A

PROJECT REPORT ON

**AI College Chatbot**

Submitted by

Under Guidence of

Professor**: Shraddha Kokate**

**EXAMINATION BSC.IT,   
SEMESTER V-VI**



**UNIVERSITY OF MUMBAI**

UNIVERSITY OF INFORMATION TECHNOLOGY

**GHANSHYAMDAS SARAF COLLEGE OF ARTS AND COMMERCE**

***(affiliated to university of Mumbai)***

**Mumbai-400067**

**Maharashtra**

**2023 – 2024**

AI COLLEGE CHATBOT

**A Project Report**

Submitted in partial fulfillment of the

Requirements for the award of the Degree of

**BACHELOR OF SCIENCE (INFORMATION TECHNOLOGY)**

**By**

ROHIT ADAK

Seat Number: \_\_\_\_\_\_\_\_

**Under the esteemed guidance of**

PROF. **SHRADDHA KOKATE**

**department of information TECHNOLOGY**

**GHANSHYAMDAS SARAF COLLEGE OF ARTS AND COMMERCE**

***(affiliated to university of Mumbai)***

**Mumbai-400067**

**Maharashtra**

**2023 – 2024**

**PROFORMA FOR THE APPROVAL PROJECT PROPOSAL**

**(Note : All entries of the proforma of approval should be filled up with appropriate and complete information. Incomplete proforma of approval in any respect will be summarily rejected.)**

PNR No**.: ……………………** Roll no**:...…………….**

1. Name of the Student
2. Title of the Project
3. Name of the Guide
4. Teaching experience of the Guide
5. Is this your first submission? Yes No

Signature of the Student Signature of the Guide

Date: …………………. Date: ………………..

Signature of the Coordinator  
Date: ………………….

**DEPARTMENT OF INFORMATION TECHNOLOGY GHANSHYAMDAS SARAF COLLEGE OF ARTS AND COMMERCE**

***(Affiliated to university of Mumbai)***

**CERTIFICATE**

This is to certify that the project entitled, **"AI College Chatbot"**, is bonafied work of **ROHIT ADAK**  bearing Seat No: submitted in partial fulfilment of the requirements for the award of degree of **BACHELOR OF SCIENCE in INFORMATION TECHNOLOGY** from University of Mumbai.

**Internal Guide** **Coordinator**

**External Examiner**

**Date: College Seal**

**ACKNOWLEDGMENT**

I would like to express our special thanks and gratitude to our project guide Prof. Shraddha Kokate Ma’am for guiding me to do the project work on time and giving us all support and guidance, which made complete our project duly. We are extremely thankful to her for providing such nice support and guidance.

I am also thankful for and fortunate enough to get constant encouragement, support and guidance from the teachers of information Technology who helped me in successfully completing our project work.

**DECLARATION**

I hereby declare that the project entitled, “**Ai College Chatbot**” done at Ghanshyamdas Saraf College of Arts and Commerce, has not been in any case duplicated to submit to any other universities for the award of any degree. To the best of my knowledge other than me, no one has submitted to any other university.

The project is done in partial fulfillment of the requirements for the award of degree of **BACHELOR OF SCIENCE (INFORMATION TECHNOLOGY)** to be submitted as final semester project as part of my curriculum.

**Name and Signature of the student**

Table of Contents

[chapter no. 1: INTRODUCTION 1](#_Toc162889489)

[1.1. Background 1](#_Toc162889490)

[1.2. Objective 1](#_Toc162889491)

[1.3. Purpose and scope 2](#_Toc162889492)

[1.3.1. Purpose 2](#_Toc162889493)

[1.3.2. Scope 2](#_Toc162889494)

[1.4. Applicability 3](#_Toc162889495)

[1.5. Organization of Report 3](#_Toc162889496)

[chapter no. 2: SURVEY OF TECHNOLOGY 4](#_Toc162889497)

[2.1. Development Software 4](#_Toc162889500)

[2.2. Front - End Technologies 7](#_Toc162889504)

[2.3. Back - End Technologies 9](#_Toc162889509)

[2.4. Hugging Face Text Generation Models 11](#_Toc162889515)

[2.5. Designing Tools 13](#_Toc162889516)

[CHAPTER NO. 3: REQUIREMENTS AND ANALYSIS 15](#_Toc162889517)

[3.1. Problem Definition for AI Chatbot 15](#_Toc162889522)

[3.2. Requirement Specification 15](#_Toc162889523)

[3.2.1. Functional Requirements 15](#_Toc162889529)

[3.2.2. Non-Functional Requirements 16](#_Toc162889536)

[3.3. Planning and Scheduling 17](#_Toc162889537)

[3.4. Software and Hardware Requirements 18](#_Toc162889538)

[3.4.1. Software requirements 18](#_Toc162889539)

[3.4.2. Hardware requirements 19](#_Toc162889540)

[3.5. Conceptual Tools 20](#_Toc162889541)

[3.5.1. Activity Diagram 20](#_Toc162889542)

[3.5.2. Class Diagram 21](#_Toc162889543)

[3.5.3. Use Case Diagram 21](#_Toc162889544)

[3.5.4. Data Flow Diagram 22](#_Toc162889545)

[3.5.5. Entity Relationship Diagram 23](#_Toc162889546)

[3.5.6. Sequence Diagram 24](#_Toc162889547)

[3.5.7. Event Table 25](#_Toc162889548)

[CHAPTER NO. 4: SYSTEM DESIGNS 26](#_Toc162889549)

[4.1. Basic Modules 26](#_Toc162889551)

[4.2. Data Design 27](#_Toc162889552)

[4.3. Flowchart 28](#_Toc162889553)

[4.4. Algorithm Design 29](#_Toc162889554)

[4.5. User-Interface Design 30](#_Toc162889555)

[4.6. Test Cases Design 31](#_Toc162889556)

[CHAPTER No. 5: IMPLEMENTATION AND TESTING 32](#_Toc162889557)

[5.1. Implementation Approach 32](#_Toc162889560)

[File Stucture 32](#_Toc162889561)

[Gemini\_QnA 33](#_Toc162889562)

[Gemini\_QnA\_API 45](#_Toc162889563)

[5.2. Testing Approch 59](#_Toc162889570)

[5.3 System Testing 61](#_Toc162889571)

[5.4. Code Efficiency 62](#_Toc162889572)

[5.4.1. Gemini API Code Efficiency 62](#_Toc162889573)

[5.4.2. Gemini GUI Code Efficiency 62](#_Toc162889574)

[CHAPTER No. 6: RESULT AND DISCUSSION 63](#_Toc162889575)

[6.1. Test Reports 63](#_Toc162889582)

[6.2. User Documentation 65](#_Toc162889583)

[User Loging Interface 65](#_Toc162889584)

[Chatting Interface 66](#_Toc162889585)

[WhatsApp Interface 67](#_Toc162889586)

[CHAPTER NO. 7 CONCLUSION 68](#_Toc162889587)

[7.1. Conclusion 68](#_Toc162889595)

[7.2. Limitation of the System 69](#_Toc162889596)

[7.3. Future Scope Of the Project 70](#_Toc162889597)

[7.4. References 71](#_Toc162889598)

List of Tables

[Table 1- Comparison Matrix - Development Software 6](#_Toc162889599)

[Table 2 - Comparison Matrix - Front-End Technologies 8](#_Toc162889600)

[Table 3- Comparison Matrix - Back-End Technologies 10](#_Toc162889601)

[Table 4 - Comparison Matrix (Hugging Face Text Generation Models) 12](#_Toc162889602)

[Table 5- Comparison Matrix - Designing Software 14](#_Toc162889603)

[Table 6 - Sequence Diagram 25](#_Toc162889604)

[Table 7- Test Cases for whatsapp\_bot.py 59](#_Toc162889605)

[Table 8 - Test Cases for app.py 60](#_Toc162889606)

[Table 9 - Test Cases for gen\_ai\_API.py: 60](#_Toc162889607)

[Table 10 - Test Cases for admin.py: 61](#_Toc162889608)

List of Figures

[Figure 1 - Gantt CHART 17](#_Toc162889609)

[Figure 2 - PERT CHART 18](#_Toc162889610)

[Figure 3 - Activity Diagram 20](#_Toc162889611)

[Figure 4 - CLASS DIAGRAM 21](#_Toc162889612)

[Figure 5 - Use Case Diagram 21](#_Toc162889613)

[Figure 6 - Data Flow Diagram 22](#_Toc162889614)

[Figure 7 - Entity Relationalship 23](#_Toc162889615)

[Figure 8 - Sequence Diagram 24](#_Toc162889616)

[Figure 9 - Gemini API ver. Code Efficiency 62](#_Toc162889617)

[Figure 10 - Gemini GUI ver. Code Efficiency 62](#_Toc162889618)

[Figure 11 - Login User Interface 65](#_Toc162889619)

[Figure 12 - Chatting Interface 66](#_Toc162889620)

[Figure 13- WhatsApp Interface 67](#_Toc162889621)

# **chapter no. 1: INTRODUCTION**

## Background

In today's technologically advanced world, colleges face a growing number of inquiries from students seeking assistance with various concerns. These queries often revolve around repetitive and commonly asked questions, which require the attention of help desk executives. However, this reliance on human support can be time-consuming and resource-intensive. To address these challenges, I am developing an AI-based chatbot for our college.

The current method of manual WhatsApp or non-AI chatbot-based query solving at our college has proven to be highly inconvenient. Students often face delays and inefficiencies when seeking assistance with their concerns. These manual processes require human intervention for each query, leading to longer response times and potential bottlenecks during peak periods. Moreover, the lack of automation and intelligence in these systems makes it challenging to provide consistent and accurate responses.

By deploying this open-source chatbot, I aim to optimise the efficiency of our college's support system. It eliminates the need for repetitive manual interventions. With the chatbot's ability to provide accurate and relevant information, students can experience improved satisfaction, faster query resolution, and enhanced overall user experience.

## **Objective**

1. **Streamline Support:** The primary objective of this project is to streamline the support process for students by automating the handling of general queries. This

automation reduces the burden on help desk executives, allowing them to focus on more complex tasks.

1. **Improve Efficiency:** By implementing an AI-based chatbot, the project aims to improve the overall efficiency of the college's support system. The chatbot provides fast and accurate responses to student queries, eliminating the need for manual intervention and reducing response times.
2. **Enhance User Experience:** The project focuses on enhancing the user experience for students by providing prompt and reliable assistance. The chatbot's availability around the clock ensures that students can receive help at any time, regardless of operational hours.
3. **Optimize Resources:** By automating repetitive and commonly asked questions, the project optimizes the allocation of resources. Help desk executives can dedicate their expertise to more specialized tasks, improving overall resource utilization within the college.

## **Purpose and scope**

### **Purpose**

* + - * **Automate Support:** Develop an AI-based chatbot to automate the handling of general queries from students, reducing the workload on help desk executives and improving overall responsiveness.
      * **Enhance User Experience:** Improve the user experience by providing fast and accurate responses to student inquiries, ensuring a seamless and satisfactory support system.
      * **Optimise Resource Allocation:** Optimise the allocation of resources by automating repetitive tasks, allowing help desk executives to focus on more complex and specialized tasks.
      * **Increase Accessibility**: Increase the accessibility of support services by providing round-the-clock availability for students, enabling them to seek assistance at any time.

### **Scope**

* + - * **General Queries:** The chatbot will handle a wide range of general queries from students, including admissions, registration, course information, campus facilities, and more.
      * **Multi-Channel Interaction**: The chatbot will be accessible through various channels, such as WhatsApp, Web application, and other social media platforms too, ensuring convenience and flexibility or students.
      * **Natural Language Processing:** The chatbot will utilise advanced natural language processing techniques to understand and interpret student queries accurately, providing relevant and meaningful responses.
      * **Integration with Existing Systems:** The chatbot will be integrated with   
        existing college systems and databases to access up-to-date information and   
         provide accurate responses.
      * **Security and Privacy:** The chatbot will prioritise data security and privacy, implementing measures to protect student information and comply with relevant regulations.

## **Applicability**

The AI-based chatbot developed for our college serves as a versatile and practical solution to address the diverse needs of students across various college departments and administrative functions. With its ability to handle a wide range of general queries, the chatbot finds applicability in key areas such as admissions, course registration, academic programs, campus facilities, and student services. By integrating the chatbot with multiple communication channels, including the college website, and social media platforms, we can ensure seamless accessibility for students, meeting them where they are most comfortable. The chatbot's advanced natural language processing capabilities enable it to accurately understand and interpret student queries, ensuring accurate and relevant responses.

## **Organization of Report**

The subsequent chapters of the project report will focus on the survey of different technologies which include a comparison of different platforms, languages, software, hardware etc.

The Survey is followed by the analysis of requirements resulting in the generation of requirement specifications and a schedule of activities. It also includes the conceptual design that visualizes the features and operations that can be performed on the system.

The final chapter of the project report includes the system design that describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudo code, and other documentation.

# **chapter no. 2: SURVEY OF TECHNOLOGY**



## **Development Software**

In the realm of development software, the project leverages a strategic combination of versatile tools to ensure efficient coding and optimal workflow.

* + - * **Visual Studio Code (VS Code):**

VS Code stands as the cornerstone of my development environment.

This lightweight and powerful source code editor offers an array of features, including intelligent coding assistance, Git integration, and an extensive marketplace for extensions. Its customizability and user-friendly interface enhance coding productivity, making it an indispensable tool for the project

.

* + - * **Anaconda:**

Anaconda, a distribution of the Python programming language, plays a vital role in the project's development process. It provides a rich ecosystem of packages and libraries, simplifying the management of dependencies. With tools like conda for environment management and Jupyter Notebook integration, Anaconda ensures a seamless workflow for data analysis and experimentation.

* + - * **Google Colab:**

Google Colab, a cloud-based Jupyter notebook environment, further enriches my development toolkit. Its integration with Google Drive facilitates collaborative coding and data sharing. The ability to execute code in a cloud-based environment eliminates hardware constraints, enabling resource-intensive tasks. With pre-installed libraries and GPUs, Google Colab empowers efficient experimentation and model training.

* + - * **Other Development Software Considerations:**

While the mentioned software serve as the project's primary development tools, other notable options were also contemplated for their distinct advantages:

* + - **Visual Studio:**

Visual Studio, a comprehensive integrated development environment (IDE), is recognized for its robust features and capabilities. While not employed in the project, its extensive toolset and support for various programming languages make it a strong choice for complex software development tasks.

* + - **Jupyter Notebook:**

Jupyter Notebook, another valuable tool not currently used in the project, offers an interactive coding environment. Particularly popular in data science, Jupyter Notebook allows code execution in cells, facilitating step-by-step analysis and visualization of results.

* + - **PyCharm:**

PyCharm, a specialised IDE for Python, provides dedicated features for Python development. Its intelligent code analysis, debugging tools, and integrated testing support enhance development efficiency. Although not part of the project's current toolkit, PyCharm remains a top choice for Python-centric projects.

**Comparison Matrix**

(Development Software)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Aspect** | **Visual Studio  Code** | **Visual Studio** | **PyCharm** | **Jupyter  Notebook** | **Google  Colab** | **Anaconda** |
| **Purpose** | Code editing | Integrated IDE | Python IDE | Interactive coding | Cloud-based notebook | Distribution platform |
| **Language  Support** | Multiple languages | Various languages | Python | Multiple languages | Python | Multiple languages |
| **Features** | Extensible, lightweight | Full-featured | Robust features | Interactive execution | Cloud-based, sharing | Package management |
| **Integrated  Debugger** | Yes | Yes | Yes | No | Limited | Yes |
| **Version Control** | Built-in | Built-in | Plugin available | Not directly | Limited | Not applicable |
| **GUI  Development** | Extensions available | Full support | Limited | No | No | No |
| **Data Science Integration** | Extensions available | Limited | Limited | Built-in | Yes | Yes |
| **Cloud Integration** | Extensions available | Azure | Limited | No | Built-in | No |
| **Learning Curve** | Moderate | Moderate | Moderate | Low | Low | Low |
| **Community Support** | Large and active | Large and active | Active | Large and active | Active | Active |

Table 1- Comparison Matrix - Development Software



## **Front - End Technologies**

* **Streamlit:**

Streamlit, my chosen front-end technology, simplifies the creation of interactive web applications using Python. Its intuitive API allows developers to create dynamic visualisations, making it suitable for my chatbot's user interface. Streamlit's real-time updates and responsive design offer a seamless user experience.

* **Flask:**

Flask, another front-end option we're using, is a lightweight web framework that emphasises simplicity and flexibility. It's well-suited for building small to medium-sized web applications. Flask's modular structure and extensive library support enable developers to create feature-rich web interfaces.

* **Gradio, Django, and FastAPI:**

Gradio, Django, and FastAPI are notable contenders that were considered but not selected for the project. Gradio's focus on creating user- friendly interfaces for machine learning models offers potential for dynamic interaction with our AI-based chatbot. Django, a high-level Python web framework, provides robust tools for building complex web applications, which could prove valuable in more intricate projects. FastAPI's speed and performance capabilities are commendable, making it an attractive choice for high-performance web services. While these technologies were not integrated into the project's current development, their unique features and strengths remain noteworthy in the broader context of web application development.

Comparison Matrix

(Front-end Technologies)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Aspect** | **Flask** | **Streamlit** | **Gradio** | **Django** |
| **Use Case** | General web apps | Data apps, dashboards | UI for ML models | Full-stack web apps |
| **Learning  Curve** | Relatively simple | Very simple | Simple | Moderate to complex |
| **UI Design** | Manual HTML/CSS | Python scripting | Python scripting | Templating system |
| **Real-time  Interaction** | Possible, requires extra work | Built-in | Built-in | Possible, requires work |
| **Customization** | High flexibility | Limited flexibility | Limited flexibility | High flexibility |
| **Deployment** | Various options | Self-contained app | Various options | Flexible deployment |
| **Ecosystem** | Lightweight | Focused on data apps | Focused on ML UI | Comprehensive |
| **URL Routing** | Basic routing | - | - | Advanced routing |
| **Community Support** | Large and active | Growing community | Active community | Large and active |

Table 2 - Comparison Matrix - Front-End Technologies



## **Back - End Technologies**

* **Python:**

Python, my selected back-end language, boasts remarkable versatility and

adaptability, making it an ideal choice for the project. Its simplicity and readability expedite the development process. The extensive community support ensures swift problem-solving and access to a rich pool of libraries for diverse functionalities.

Python's object-oriented nature facilitates modular coding, enhancing code reusability. Its compatibility with various platforms streamlines deployment, and its integration with popular frameworks, such as Django and Flask, makes it seamless to build web applications

* **Java:**

Although not chosen for the project, Java is a robust, class-based, object- oriented programming language. It offers platform independence through the Java Virtual Machine (JVM), enabling applications to run on different platforms. Java's extensive standard library and rich ecosystem of frameworks and tools support various application types, from web applications to mobile apps. The strong typing and strict syntax promote code stability and maintainability.

* **C#:**

While not selected for the project, C# is a modern programming language developed by Microsoft. It excels in building Windows applications, including desktop applications and games. C#'s tight integration with the .NET framework ensures efficient memory management and runtime performance. Its features like automatic garbage collection and strong type checking enhance code quality and reliability.

* **C++:**

Though not chosen for the project, C++ is a versatile language known for its high-performance capabilities. It's widely used in system programming, game development, and embedded systems. C++ offers low-level memory manipulation and direct hardware access, making it suitable for resource-intensive applications. Its features, including object-oriented and procedural programming paradigms, provide developers with flexibility and control.

**Comparison Matrix**

(Back-end Technologies)

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **Java** | **Python** | **C++** |
| **Typing** | Static | Dynamic | Static and Dynamic |
| **Syntax** | Complex, verbose | Concise, readable | Complex, flexible |
| **Memory** | Automatic garbage collection | Automatic memory management | Manual memory management |
| **Use Cases** | Web apps, Android apps | Scripting, automation | Systems, games |
| **Libraries** | Rich ecosystem | Extensive libraries | Vast libraries |
| **Performance** | Slower than C++ | Slower than C++ | High performance |
| **Learning Curve** | Moderate | Easy | Steep |
| **Error Handling** | Explicit exceptions | Exceptions, runtime errors | Pointers, exceptions |
| **Community** | Large and active | Large and supportive | Mature and diverse |

Table 3- Comparison Matrix - Back-End Technologies



## **Hugging Face Text Generation Models**

* **Google Gemini:**

Google Gemini, my primary text generation model, is powered by Hugging Face's state-of-the-art natural language processing technology. It offers exceptional performance in generating contextually relevant and coherent responses to various queries. Google Gemini's underlying transformer architecture enables it to understand and mimic human language patterns effectively, contributing to accurate and natural-sounding output. Its adaptability allows fine-tuning for specific domains, making it an excellent fit for my college chatbot's needs.

* **Google flan:**

Flan is a capable language model, but it primarily focuses on text-based communication. This can be limiting for a college chatbot where you might want to incorporate images, diagrams, or other multimedia elements to enhance explanations or guide students through processes. If you want to go beyond basic text interactions, consider a model like Gemini that has the flexibility to handle a wise wider range of content formats.

* **GPT-2:**

Although not the chosen model for the project, GPT-2, developed by OpenAI, is renowned for its text generation capabilities. It employs a transformer-based architecture to produce coherent and contextually relevant text. GPT-2's large-scale training enables it to generate lengthy passages while maintaining logical flow. Its fine-tuning capacity enables domain-specific adaptation

* **Bloom:**

Not selected for the project, Bloom is a text generation model known for its concise and creative output. Its unique approach to language generation focuses on generating short, artistic pieces of text. While its primary use case differs from my college chatbot's requirements, Bloom's creative flair could be beneficial in certain contexts.

Comparison Matrix

(Hugging Face Text Generation Models)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Aspect** | **Hugging Face GPT-3** | **Google FALN** | **Google T5** | **Google Gemini** |
| **Model Type** | Autoregressive | Autoregressive | Encoder-Decoder | Multimodal |
| **Pre-Training Method** | Unsupervised | Unsupervised | Unsupervised | Unsupervised |
| **Fine-Tuning** | Yes | Yes | Yes | Yes |
| **Few-shot Learning** | Yes | Yes | Yes | Yes |
| **Task Diversity** | Wide range of tasks | Few-shot tasks | Text generation | Wide range (+) |
| **Input Format** | Text | Text | Text | Text, Image, Audio, Code |
| **Output Format** | Text | Text | Text | Text, Image, Code |
| **Maximum Sequence Length** | Varies | Limited | Varies | Varies |
| **Model Sizes** | Various sizes | Single size | Various sizes | Ultra, Pro, Nano |
| **Training Data** | Diverse | Diverse | General text | Diverse |
| **Community Support** | Large and active | - | - | Emerging |

Table 4 - Comparison Matrix (Hugging Face Text Generation Models)

## **Designing Tools**

* **EdrawMax:**

For design visualisation, I've chosen EdrawMax, a versatile tool offering a wide array of diagram types. EdrawMax's intuitive interface enables the creation of flowcharts, diagrams, timelines, and more. Its extensive library of symbols and templates accelerates design creation. The collaborative features ensure effective teamwork in visualising complex concepts.

* **StarUML:**

Not selected for the project, StarUML is a software modelling tool focused on UML (Unified Modeling Language) diagrams. It supports a range of diagram types, including class diagrams, sequence diagrams, and more. StarUML's features facilitate comprehensive software design and architecture representation.

Comparison Matrix

**(Designing Tools)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Aspect** | **EdrawMax** | **PlantUML** | **Google Drawings** |
| **Diagram Types** | Wide variety | Limited | Limited |
| **User Interface** | GUI-based, user-friendly | Text-based, simple | GUI-based, simple |
| **Collaboration** | Built-in collaboration | Version control | Real-time collaboration |
| **Customization** | High customization | Limited customization | Limited customization |
| **Output Formats** | Various file formats | Images, text | Images |
| **Learning Curve** | Moderate | Moderate | Very easy |
| **Integration** | Limited integrations | Limited integrations | Google Workspace |
| **Online/Offline** | Both | Offline | Online |
| **Community Support** | Active | Active | Limited |

Table 5- Comparison Matrix - Designing Software

In conclusion, the project's selected technologies are carefully curated to ensure efficient development and successful deployment. The detailed comparison tables provide insights into each application's features, aiding in making informed decisions that align with the project's goals.

# **CHAPTER NO. 3: REQUIREMENTS AND ANALYSIS**



## **Problem Definition for AI Chatbot**

In the dynamic landscape of modern education institutions, students and faculty members often encounter challenges when seeking quick and accurate information. Traditional communication channels like help desks and emails might lead to delays in responses and increased workload on administrative staff. To address these inefficiencies and enhance user experience, the AI Chatbot project aims to create an intelligent and responsive virtual assistant capable of providing prompt and relevant answers to a wide array of queries from students, faculty, and other stakeholders. This AI- powered solution seeks to streamline information retrieval, reduce reliance on manual support systems, and improve overall communication efficiency within the college environment.

## **Requirement Specification**



### **Functional Requirements**

* + **Query Processing**
  + The chatbot must analyze user queries and identify the intent.
  + It should handle queries related to general college information, student-specific inquiries, and class-related questions
  + **Response Generation**
  + The chatbot must analyze user queries and identify the intent.
  + It should handle queries related to general college information, student-specific inquiries, and class-related questions
  + **User Differentiation**
    - The chatbot should distinguish between user types (e.g., students, faculty, admin) and provide appropriate responses.

* + **Communication Channels**
    - The chatbot should support multiple communication channels, such as WhatsApp and a website interface.
  + **Data Retrieval**
    - The chatbot needs to retrieve information from embedded documents and knowledge bases to provide accurate answers.
  + **Real-time Interaction**
    - The chatbot needs to retrieve information from embedded documents and knowledge bases to provide accurate answers.



### Non-Functional Requirements

* Usability
  + The chatbot interface should be intuitive and user-friendly for individuals of varying technical expertise.
* Accuracy
  + The chatbot must provide accurate and relevant responses to maintain user trust.
* Scalability
  + The system should be capable of handling a growing number of users and queries without performance degradation.
* Security
  + User data and query history should be handled securely to protect sensitive information.
* Availability
  + The chatbot should be available and responsive around the clock to accommodate users from different time zones.
* Adaptability
  + The chatbot should be capable of adapting to changes in user behavior and query patterns over time.

## **Planning and Scheduling**

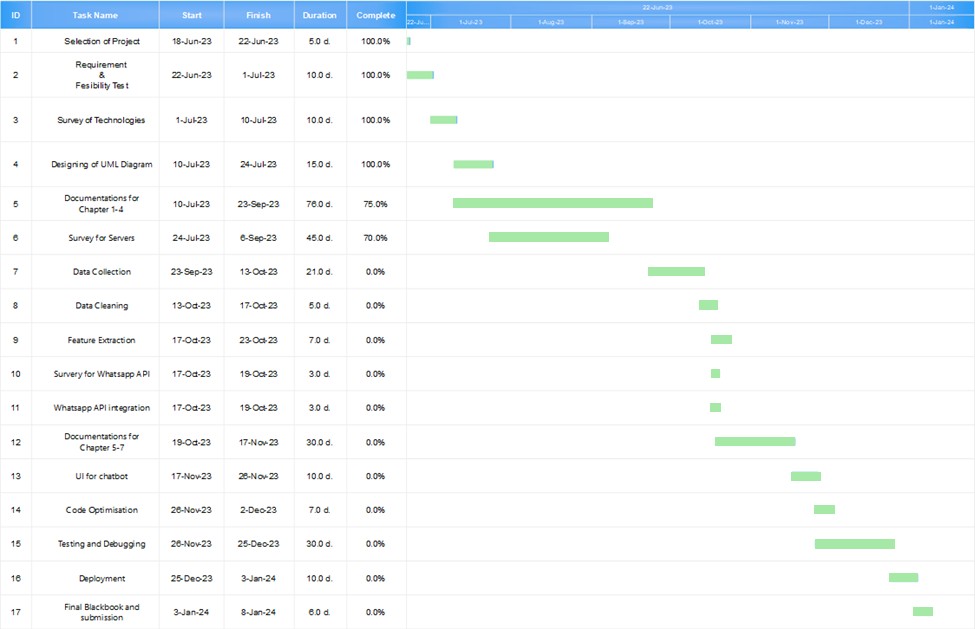
**Gantt CHART**

Figure 1 - Gantt CHART

**PERT CHART**

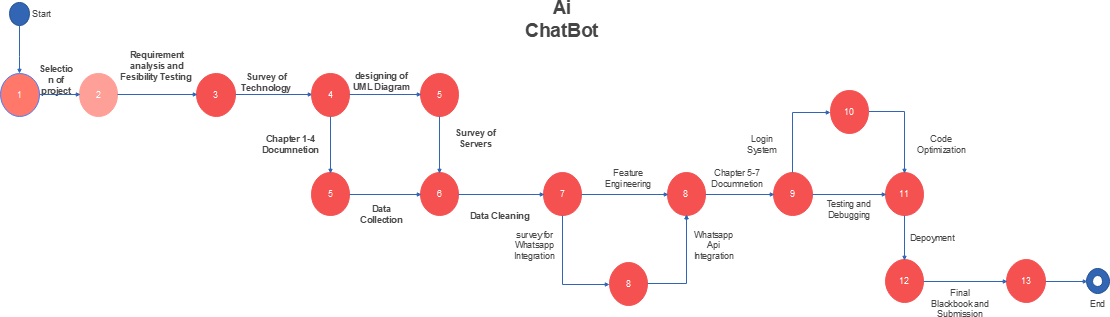


Figure 2 - PERT CHART

## Software and Hardware Requirements

In developing and deploying the AI-based chatbot for college queries, several software and hardware requirements are essential to ensure smooth operation and optimal performance.

### **Software requirements**

* **Google Colab:**

Google Colab provides a cloud-based development environment for Python, making it a convenient choice for training and testing machine learning models. Its collaborative features allow multiple team members to work together on the project in real-time.

* **Python 3.12 or more:**

Python is the primary programming language used for developing the chatbot. It's essential to have Python 3.7 or a more recent version to take advantage of the latest features and libraries.

* **Windows 10 or higher:**

The project is compatible with Windows 8 and later versions. However, it's worth noting that Python is cross-platform, and you can adapt the project for other operating systems as well.

* **Latest version of all libraries:**

To maintain compatibility and leverage the most recent advancements, it's important to keep all libraries and dependencies up to date. This includes libraries for natural language processing (NLP), machine learning, and web interactions.

### **Hardware requirements**

* **A standalone computer (i5 8th Gen, 8 GB ram or higher):**

To run the chatbot and support its underlying processes, a computer with a minimum of an 8th generation Intel Core i5 processor and 8 GB of RAM is recommended. The chatbot's natural language processing tasks and machine learning algorithms benefit from the processing power and memory provided by these specifications.

* **Secondary memory to store all the database:**

As the chatbot accumulates a knowledge base over time, a secondary storage solution is required for efficient data management. Ensure adequate storage capacity to store the database, user interactions, and model data securely.

These software and hardware requirements provide the foundation for building and deploying a robust AI-based chatbot tailored to college-specific queries. It's important to maintain these prerequisites to ensure optimal functnality and user satisfaction.

## Conceptual Tools

### Activity Diagram

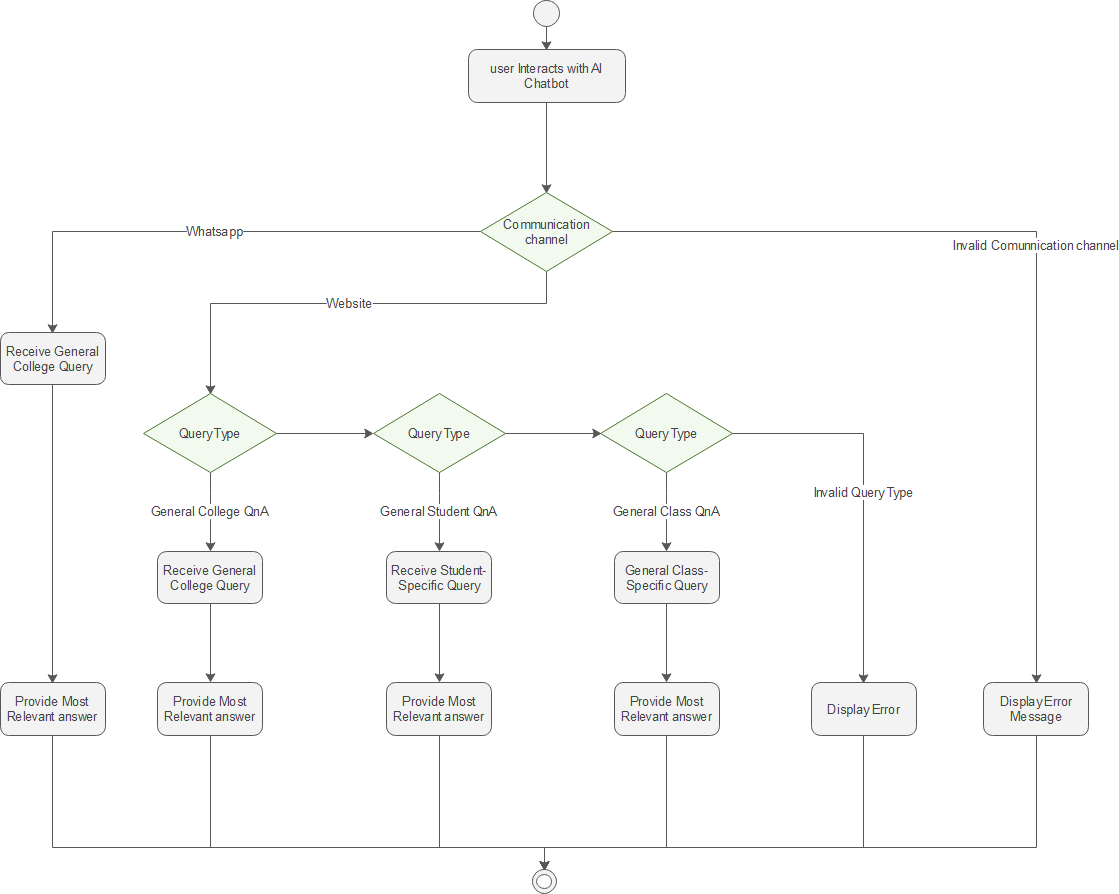


Figure 3 - Activity Diagram

### **Class Diagram**

Figure 4 - CLASS DIAGRAM

### **Use Case Diagram**

Figure 5 - Use Case Diagram

### **Data Flow Diagram**

Figure 6 - Data Flow Diagram

### **Entity Relationship Diagram**

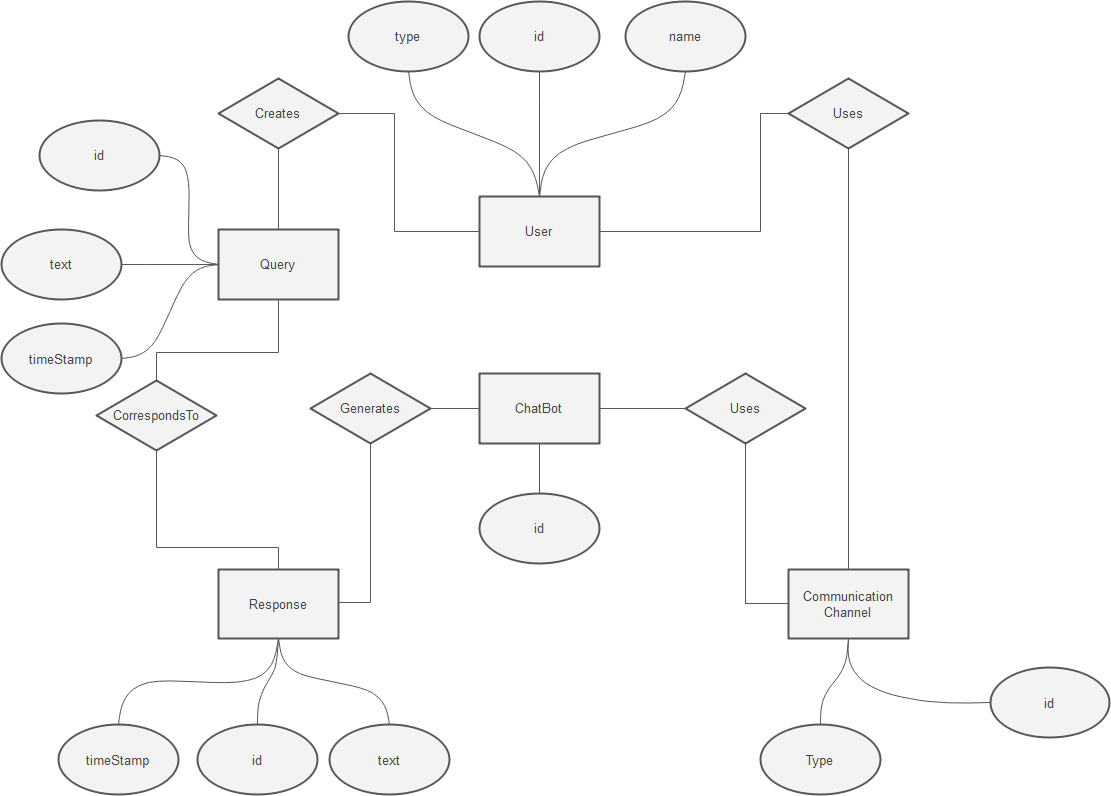


Figure 7 - Entity Relationalship

### **Sequence Diagram**

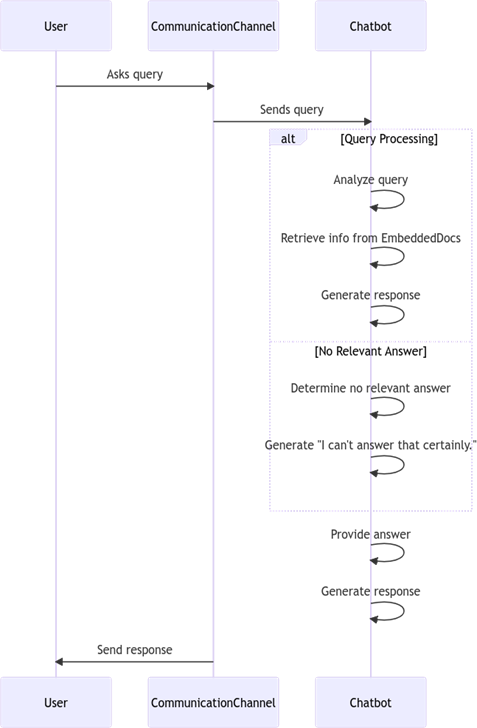


Figure 8 - Sequence Diagram

### **Event Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Event Name** | **Actor/Object** | **Action/Description** | **Possible Outcomes** |
| **User Asks Query** | User | User initiates a query from a communication channel. | Query initiated by the user. |
| **Query Sent to Chatbot** | Communication Channel | The communication channel forwards the query to the chatbot. | Query is transmitted to the chatbot. |
| **Chatbot Receives Query** | Chatbot | Chatbot receives the query and starts processing. | Chatbot receives the query for processing. |
| **Query Analysis** | Chatbot | Chatbot analyzes the query to understand its intent. | Chatbot analyzes the query for intent. |
| **Relevant Answer Found** | Chatbot | Chatbot identifies a relevant answer from its knowledge base. | Chatbot finds a relevant answer from its knowledge base and selects it for response. |
| **No Relevant Answer** | Chatbot | Chatbot determines no relevant answer is available. | Chatbot identifies that no relevant answer is found. |
| **Invalid Query Response** | Chatbot | Chatbot responds with "I can't answer that certainly." | Chatbot provides a response indicating inability to answer. |
| **Formulate Response** | Chatbot | Chatbot generates a response to the user's query. | Chatbot generates a response. |
| **Response Sent to Communication** | Chatbot | Chatbot sends the response back to the communication channel. | Response is transmitted back to the communication channel. |
| **Display Response to User** | Communication Channel | The communication channel displays the response to the user. | The communication channel presents the response to the user. |

Table 6 - Sequence Diagram

# **CHAPTER NO. 4: SYSTEM DESIGNS**



## **Basic Modules**

* **User Module:**

The User Module manages user interactions and authentication within the chatbot system. It distinguishes between different types of users based on their communication channels. These user types include unknown users, students, and administrators. Each user type has specific access and permissions within the system. For example, unknown users may have limited access, while students and administrators have broader privileges. User authentication ensures that the right individuals have access to the appropriate features and information.

* **Communication Channel Module:**

The Communication Channel Module is responsible for handling the various communication channels through which users interact with the chatbot. For instance, it manages interactions via WhatsApp and the website. This module ensures that messages and queries from users are appropriately routed to the chatbot for processing. It also manages the flow of responses back to users through their chosen communication channel, maintaining a seamless conversational experience.

* **Chatbot Module:**

At the core of the system lies the Chatbot Module. This module represents the intelligence of the chatbot. It processes user queries, generates responses, and retrieves information from embedded documents. The chatbot utilizes algorithms to understand and interpret user queries, search for relevant answers in its knowledge base, and craft informative responses. It can also access embedded documents to retrieve detailed information when needed.

## **Data Design**

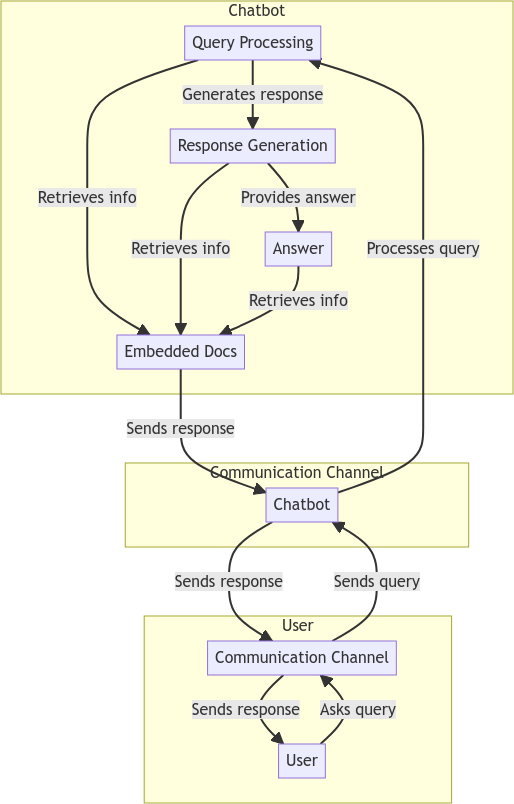
* **Database Structure:**

The database structure of the chatbot system plays a crucial role in storing and managing data. It includes tables for user information, communication history, queries, responses, and embedded documents. These tables are interconnected to facilitate efficient data retrieval and processing. For example, user information may link to communication history to maintain a record of interactions.

* **Data Flow:**

Data flow within the system is crucial for a smooth user experience. Data flows from user queries through the Communication Channel Module to the Chatbot Module, where it undergoes processing and response generation. The processed response is then sent back to the user through the Communication Channel Module. This data flow ensures that user interactions are tracked, queries are analyzed, and appropriate responses are delivered.

## **Flowchart**



## **Algorithm Design**

**Step 1.** User initiates interaction with the chatbot.

**Step 2.** If the user is registered, they log in; otherwise, proceed as an unknown user.

**Step 3.** Detect the user type (unknown, student, or administrator).

**Step 4.** The user enters a query or message via the chosen communication channel (e.g., WhatsApp or the website).

**Step 5.** The chatbot receives and analyzes the query using Natural Language Processing (NLP) techniques.

**Step 6.** Classify the query into one of the following categories: General College Question, Student-Specific Question, or Class-Specific Question.

**Step 7.** Retrieve relevant data based on the query category (knowledge base, embedded documents, etc.).

**Step 8.** Analyze the retrieved data to extract keywords and context.

**Step 9.** Generate a context-aware and informative response.

**Step 10.** Send the response back to the user through the same communication channel.

**Step 11.** Optionally, provide customized responses and actions for WhatsApp interactions.

**Step 12.** Optionally, store user data and interaction history in a database for reference.

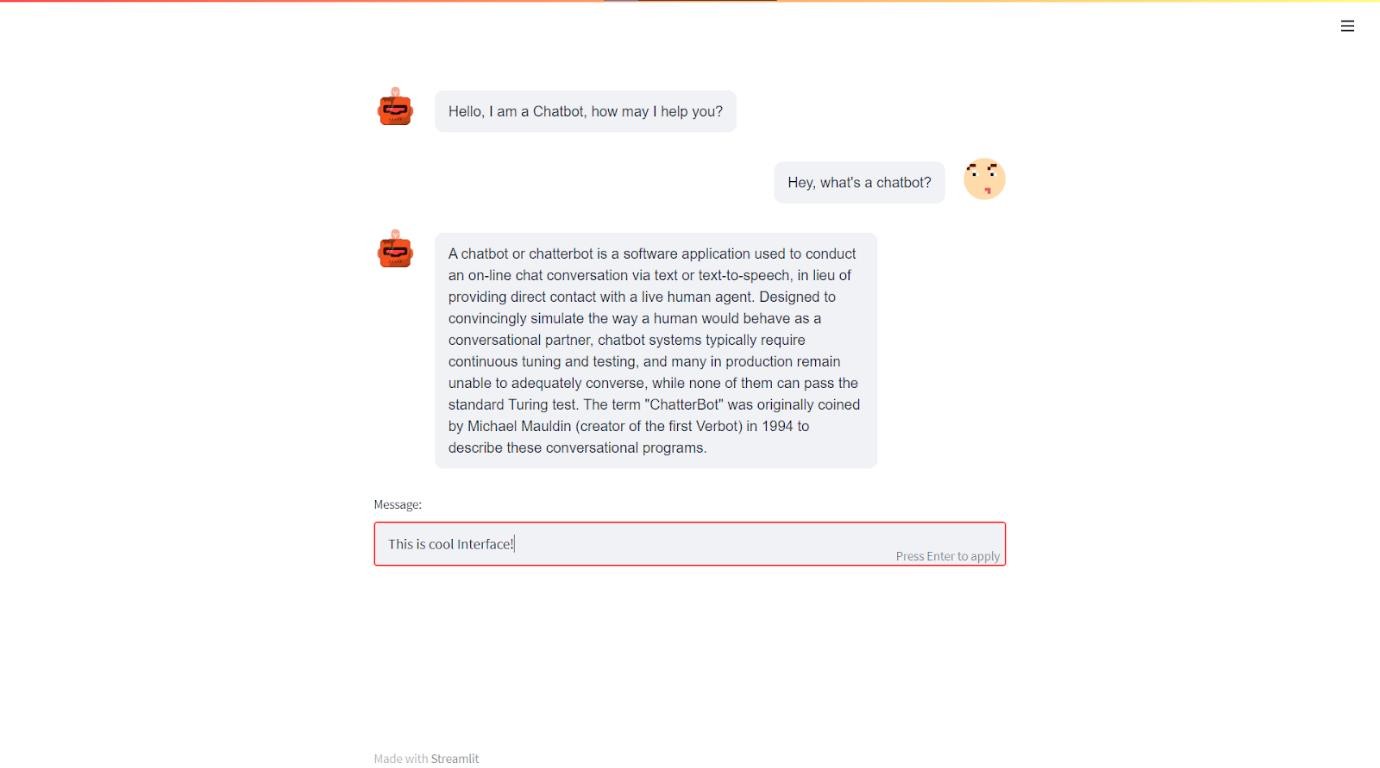
**Step 13.** Continuously monitor response times and system performance.

**Step 14.** Optionally, conduct usability testing to gather user feedback.

**Step 15.** Continually improve the chatbot's algorithms and responses based on feedback and evolving requirements.

## **User-Interface Design**

* **Website UI:**

The website-based UI design focuses on creating an intuitive and user- friendly interface for users to interact with the chatbot. It includes the layout, components, and features of the web interface. Mockups and wireframes are used to plan the user journey and design. Key elements may include a chat window, query input field, response display area, and navigation controls.

* **WhatsApp Interaction:**

While WhatsApp typically has a predefined interface, customizations are made to enhance the user experience when interacting with the chatbot. These customizations may include setting up specific response formats, enabling quick actions, and providing clear instructions to users on how to engage with the chatbot effectively.

## **Test Cases Design**

Outline the test cases and scenarios designed to ensure the functionality, reliability, and performance of your chatbot system

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Input Data** | **Expected Result** |
| **TC\_1** | User Authentication | Valid login credentials | Successful login |
| **TC\_2** | User Authentication | Invalid login credentials | Login failure |
| **TC\_3** | Query Processing | User enters a general college question | Relevant answer provided |
| **TC\_4** | Query Processing | User logs in and asks a student-specific query | Relevant answer provided |
| **TC\_5** | Query Processing | User logs in and asks a class-specific query | Relevant answer provided |
| **TC\_6** | Query Processing | User asks an unknown question | Response: “I can’t answer that certainly.” |
| **TC\_7** | Communication Channel Handling | User interacts via WhatsApp | Messages sent and received as expected |
| **TC\_8** | Communication Channel Handling | User interacts via the website | Messages sent and received as expected |
| **TC\_9** | Data Retrieval | Chatbot retrieves information from embedded documents | Accurate data retrieved |
| **TC\_10** | Data Storage | User data is stored in the database | Data successfully stored |
| **TC\_11** | Query Processing Algorithm | Complex query processing | Relevant and coherent response |
| **TC\_12** | Response Generation Algorithm | Response generation for different query types | Accurate and context-aware responses |
| **TC\_13** | Website UI Interaction | User interacts with the chatbot on the website | User-friendly interface |
| **TC\_14** | WhatsApp Customization | User interacts with the chatbot on WhatsApp | Customized chatbot behavior |
| **TC\_15** | Integration Testing | Test integration between modules | Modules integrated successfully |
| **TC\_16** | Performance Testing | Evaluate chatbot response time | Response time within acceptable limits |
| **TC\_17** | Usability Testing | Gather user feedback on the chatbot’s usability | User satisfaction and feedback |

# **CHAPTER No. 5: IMPLEMENTATION AND TESTING**



## Implementation Approach

### File Stucture

1. **Gemini\_QnA**

* .env
* .gitignore
* admin.py
* app.py
* auth.py
* chat.py
* chat\_logs.csv
* gen\_ai\_API.py
* RAG\_JUPYTER.ipynb
* requirements.txt

1. **Gemini\_QnA\_API**

* assets / general\_qna.txt
* .env
* .gitignore
* app.py
* chat\_logs.csv
* database.py
* gen\_ai\_model.py
* requirements.txt

### Gemini\_QnA

#### \Gemini\_QnA\gen\_ai\_API.py

1. **Environment Setup:**

The script starts by importing necessary libraries (`dotenv`, `requests`, `json`, `uuid`, `os`) and loading environment variables from a `.env` file using `load\_dotenv()`. These variables likely include API access tokens and URLs.

1. **User Authentication:**

The `user\_login` function is defined to handle user authentication. It sends a POST request to a login endpoint (`login\_request\_url`) with provided email and password. The credentials are sent along with access tokens and secret keys as payload. The function returns whether the user is valid or not.

1. **AI Interaction:**

The `ask\_ai` function handles user input to interact with the chatbot. It takes user's question and user's identifier (e.g., user's name) as input. It generates a unique ID if not provided (`uuid4()`). Then it constructs a payload containing the user's question, user's identifier, and a model identifier. This payload is sent as a POST request to a chat AI endpoint (`chat\_request\_url`). The function returns the response from the chatbot.

1. **Main Execution:**

In the main block (`if \_\_name\_\_ == '\_\_main\_\_':`), it prompts the user to input a question (`user\_question`). It then calls the `ask\_ai` function with the user's question and a test user identifier. The response from the chatbot is printed/displayed.

Overall, this script enables users to interact with a chatbot API by sending user questions and receiving responses, and it also provides a mechanism for user authentication before accessing the chatbot functionality.

from dotenv import load\_dotenv

import requests

import json

from uuid import uuid4

import os

load\_dotenv()

chat\_request\_url = os.environ.get('REQUEST\_URL')+'/chat\_ai'

login\_request\_url = os.environ.get('REQUEST\_URL')+'/user\_auth'

payload = {

    "access\_token": os.environ.get('API\_ACCESS\_TOKEN'),

    "secret\_key": os.environ.get('API\_SECRET\_KEY')

}

headersList = {

    "Accept": "\*/\*",

    "User-Agent": "Web UI for AI\_Chat\_bot (https://college-chatbot-1.onrender.com/)",

    "Content-Type": "application/json"

}

def user\_login(email,password):

    payload["login\_creds"] = {"email":email, "password": password}

    response = requests.request(

        'POST', login\_request\_url, data=json.dumps(payload),  headers=headersList)

    response\_json = response.json()

    # Define chat prompt template and output parser

    try:

        answer = response\_json['is\_valid\_user']

    except:

        answer = response\_json

    finally:

        return answer

# Function to handle user input and generate response

def ask\_ai(user\_question, user, id=None):

    if id == None:

        id = str(uuid4())

    payload["queries"] = [{"id": id, "prompt": user\_question, "user": user}]

    payload["model"] = "bscit\_c\_query"

    response = requests.request(

        'POST', chat\_request\_url, data=json.dumps(payload),  headers=headersList)

    response\_json = response.json()

    # Define chat prompt template and output parser

    try:

        answer = response\_json['response data'][0]['response']

    except:

        answer = response\_json

    finally:

        return answer

if \_\_name\_\_ == '\_\_main\_\_':

    user\_question = input('Please say what you wanna know ? ')

    user = 'Test user'

    response = ask\_ai(user\_question,user)

#### \Gemini\_QnA\chat.py

1. **Imported Modules:** The script imports necessary modules including `os`, `datetime`, `pandas`, and `streamlit`. It also imports functions `display\_logs` from a module named `admin` and `ask\_ai` from a module named `gen\_ai\_API`.
2. **Global Variables:** It defines global variables such as `admins` (a list of admin email addresses) and `useChat` (a boolean flag to control whether to use the chat interface or display logs).
3. **Admin Controls:** The `admin\_controls` function toggles the `useChat` flag when an admin user interacts with the interface.
4. **Logging Function:** The `logger` function logs chat interactions by appending data to a CSV file named `chat\_logs.csv`.
5. **Delete Session Function:** The `delete\_session` function deletes the user's session, effectively logging out the user.
6. **Clear Chat History Function:** The `clear\_chat\_history` function clears the chat history displayed in the UI.
7. **Chat UI Function:** The `chat\_ui` function constructs the main chat interface using Streamlit components. It displays chat messages, accepts user input, sends the input to the chatbot API (`ask\_ai` function), and displays the response. It also logs the interaction using the `logger` function.
8. **Load UI Function:** The `load\_ui` function constructs the UI layout including sidebar menu for user and admin actions. It determines whether to display the chat interface or admin controls based on the `useChat` flag.
9. **Main Execution:** In the main block (`if \_\_name\_\_ == '\_\_main\_\_':`), it calls the `load\_ui` function to run the chat interface.

Overall, this script provides a user-friendly interface for interacting with a chatbot, includes admin controls for managing chat functionality, and logs chat interactions for future reference.

import os

from datetime import datetime

from admin import display\_logs

from gen\_ai\_API import ask\_ai

import pandas as pd

import streamlit as st

# GLOBAL Variables

admins = ['rohitadak02@gmail.com']

useChat = True

# Controls for ADMINS

def admin\_controls():

    global useChat

    useChat = not useChat

# Function for logging chat interactions

def logger(user, prompt, response):

    current\_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    if not os.path.exists('chat\_logs.csv'):

        # Create an empty DataFrame with the specified columns

        df = pd.DataFrame(columns=['Timestamp', 'User', 'Prompt', 'Response'])

        # Write the DataFrame to a CSV file

        df.to\_csv('chat\_logs.csv', index=False)

    data = {'Timestamp': current\_time, 'User': [

        user], 'Prompt': [prompt], 'Response': response}

    print(data)

    pd.DataFrame(data, columns=['Timestamp', 'User', 'Prompt', 'Response']).to\_csv(

        'chat\_logs.csv', mode='a', index=False, header=False)

# logout user by deleting user session

def delete\_session():

    del st.session\_state['useremail']

# Function to clear chat history

def clear\_chat\_history():

    st.session\_state.messages = [{"role": "assistant", "content": "Hey Hi!!"}]

# Load UI for chatting

def chat\_ui():

    # Main chat interface

    st.title("Ask me your college doubts")

    st.write("Welcome to the chat!")

    # Initialize chat history if not exists

    if "messages" not in st.session\_state.keys():

        st.session\_state.messages = [

            {"role": "assistant", "content": """Hey Hello!! \n

 \*\*You can ask me Questions like:\*\*

 - Syllabus for 2nd semester

 - Time table for regular examination schedules for Sem-6

 - Time table and ATKT examination schedules for Sem-6

 - Need notes for ITSM"""}]

    # Display chat messages

    for message in st.session\_state.messages:

        with st.chat\_message(message["role"]):

            st.markdown(message["content"])

    # User input

    if prompt := st.chat\_input():

        st.session\_state.messages.append({"role": "user", "content": prompt})

        with st.chat\_message("user"):

            st.markdown(prompt)

    # Bot response

    if st.session\_state.messages[-1]["role"] != "assistant":

        with st.chat\_message("assistant"):

            with st.spinner("Thinking..."):

                response = ask\_ai(prompt,user=st.session\_state['useremail'])

                placeholder = st.empty()

                placeholder.markdown(response)

        if response is not None:

            message = {"role": "assistant", "content": response}

            logger(st.session\_state['useremail'], prompt, response)

            st.session\_state.messages.append(message)

# Function to clear chat history

def clear\_chat\_history():

    st.session\_state.messages = [{"role": "assistant", "content": "Hey Hi!!"}]

# UI for chat interaction

def load\_ui():

    global useChat

    # Sidebar menu

    with st.sidebar:

        st.title("Menu:")

        user\_tab, admin\_tab = st.tabs(['user', 'admins'])

        with user\_tab:

            # Check if user is logged in

            if st.session\_state['useremail']:

                # CSS to position elements at the bottom

                st.markdown(

                    """

                    <style>

                    .sidebar .element-container:last-child {

                        position: absolute;

                        bottom: 0;

                        width: 100%;

                    }

                    </style>

                    """,

                    unsafe\_allow\_html=True

                )

            st.selectbox('Course', options=['BSCIT'])

            st.selectbox('Mode', options=['QnA'])

            clear\_chat\_container = st.container()

            with clear\_chat\_container:

                st.button('Clear Chat History', on\_click=clear\_chat\_history)

                st.button('Log out', on\_click=delete\_session)

        with admin\_tab:

            if not st.session\_state.useremail in admins:

                useChat = True

            btn\_name = "Check Logs" if useChat else "Use ChatBot"

            st.button(btn\_name, on\_click=admin\_controls,

                      disabled=not st.session\_state.useremail in admins, help='Hope you are an ADMIN')

    if useChat:

        chat\_ui()

    else:

        display\_logs()

if \_\_name\_\_ == '\_\_main\_\_':

    # Run the chat UI

    load\_ui()

#### \Gemini\_QnA\auth.py

This script is a Streamlit application designed for user authentication and login functionality. Here's a breakdown of its components:

1. **Library Imports:** The script imports the Streamlit library (`streamlit`). Additionally, it imports the `user\_login` function from the `gen\_ai\_API` module. This function likely handles user authentication.
2. **Session Management:** The `start\_session` function sets the user's email as a session variable (`st.session\_state.useremail`). This allows the application to keep track of the logged-in user.
3. **User Authentication Function:** The `user\_auth` function handles user authentication. It takes the user's email and password as input, calls the `user\_login` function with these credentials, and starts a session if authentication is successful. Otherwise, it displays an error message.
4. **Login Function:** The `login` function sets up the UI for user login. It provides input fields for email and password, a dropdown for course selection, and a login button. When the login button is clicked, it triggers the authentication process by calling the `user\_auth` function.
5. **Main Execution:** The script runs the `login` function when executed.

Overall, this script creates a simple user authentication interface using Streamlit, allowing users to log in with their email and password. Upon successful authentication, it starts a session for the user.

import streamlit as st  # Importing the Streamlit library

from gen\_ai\_API import user\_login

# logging in user by setting session

def start\_session(useremail):

    st.session\_state.useremail = useremail

def user\_auth(userEmail, userPass):

    user\_auth = user\_login(email=userEmail, password=userPass)

    if user\_auth:

        start\_session(userEmail)

    else:

        st.error("Invalid email or password -- RETRY ")

def login():

    """

    Function to handle user login.

    This function displays an email input field and a course selection dropdown.

    When the user clicks the login button, it stores the entered email in the cookie\_manager.

    """

    # User input for email

    useremail = st.text\_input(

        "EMAIL:", autocomplete='email', placeholder='(Please enter your mail ID)')

    password = st.text\_input(

        "PASSWORD:",autocomplete='password', placeholder='(Please enter your password)',type='password')

    # Dropdown for course selection

    course = st.selectbox('Course', options=['BSCIT'], key='login\_course')

    # Button to submit login information

    if  st.button('Log in', disabled= not (password and useremail) ):

        user\_auth(userEmail=useremail, userPass=password)

#### \Gemini\_QnA\app.py

1. **Streamlit Import:** The script imports the Streamlit library, which is used for building web applications with Python.
2. **User Authentication:** The `user\_login` function is imported from the `gen\_ai\_API` module, which presumably handles the authentication process.
3. **Session Management:** The `start\_session` function is defined to set the user's email as a session variable (`st.session\_state.useremail`). This allows the application to keep track of the logged-in user.
4. **User Authentication UI:** The `user\_auth` function is defined to handle user authentication. It takes the user's email and password as input, sends a request to the `user\_login` function, and starts a session if authentication is successful. Otherwise, it displays an error message.
5. **Login UI:** The `login` function sets up the UI for user login. It provides input fields for email and password, a dropdown for course selection, and a login button. When the login button is clicked, it triggers the authentication process.
6. **Main Execution:** The `if \_\_name\_\_ == '\_\_main\_\_':` block calls the `login` function to run the login UI.

Overall, this script creates a simple user authentication interface using Streamlit, allowing users to log in with their email and password. Upon successful authentication, it starts a session for the user.

import streamlit as st

from auth import login  # Importing the login function from the auth module

from chat import load\_ui  # Importing the chat\_ui function from the chat module

def main():

    """

    Main function to handle the user interaction flow.

    If the user is not logged in, it prompts for login, otherwise starts the chat interface.

    """

    if not st.session\_state.get('useremail',default=None):  # If useremail is not present in the cookie\_manager

        login()  # Prompt user to login

    else:

        load\_ui()  # Start the chat interface

if \_\_name\_\_ =='\_\_main\_\_':

    main()  # Execute the main function if this script is run directly

#### \Gemini\_QnA\admin.py

This script is a Streamlit application designed to display chat logs retrieved from an API endpoint. Here's a breakdown of its functionality:

1. **Library Imports:** The script imports necessary libraries:
   * 1. `**pandas**` for data manipulation.
     2. `**streamlit**` for building web applications.
     3. `**requests**` for making HTTP requests.
     4. `**json**` for handling JSON data.
     5. `**dotenv**` for loading environment variables from a `.env` file.
2. **Environment Setup:** It loads environment variables from a `.env` file using `load\_dotenv()`. These variables likely include the request URL, API access token, and secret key.
3. **Streamlit Configuration:** The script sets the page layout configuration to 'wide' using `st.set\_page\_config(layout='wide')`. This configuration affects the layout of the Streamlit application.
4. **Request Setup:** It constructs the payload and headers necessary for making requests to the API endpoint. The payload includes access tokens and secret keys retrieved from environment variables.
5. **Display Logs Function:** The `display\_logs` function is defined to display chat logs. It sends a POST request to the specified request URL with the payload and headers. It then converts the JSON response into a pandas DataFrame and displays it using Streamlit.
6. **Main Execution:** In the `if \_\_name\_\_ == '\_\_main\_\_':` block, the script calls the `display\_logs` function to run the application and display the chat logs.

Overall, this script creates a Streamlit application that fetches chat logs from an API endpoint and presents them in a tabular format using pandas and Streamlit.

import pandas as pd

import streamlit as st

import requests,os

import json

from dotenv import load\_dotenv

load\_dotenv()

st.set\_page\_config(layout='wide')

request\_url = os.environ.get('REQUEST\_URL')+'/get\_logs'

payload = {

    "access\_token": os.environ.get('API\_ACCESS\_TOKEN'),

    "secret\_key": os.environ.get('API\_SECRET\_KEY')

}

headersList = {

    "Accept": "\*/\*",

    "User-Agent": "Web UI for AI\_Chat\_bot (https://college-chatbot-1.onrender.com/)",

    "Content-Type": "application/json"

}

def display\_logs():

    st.header("Chat Logs")

    response = requests.request(

        'POST', request\_url, data=json.dumps(payload),  headers=headersList)

    response\_json = response.json()

    df = pd.DataFrame(response\_json)

    st.write(df)

if \_\_name\_\_ == '\_\_main\_\_':

    display\_logs()

### Gemini\_QnA\_API



#### \Gemini\_QnA\_API\database.py

A Python module for connecting to a PostgreSQL database using the psycopg2 library. Here's a breakdown of its components:

1. **Library Imports**: The script imports the `psycopg2` library for interacting with PostgreSQL databases and the `dotenv` library for loading environment variables from a `.env` file.
2. **SQL Query**: It defines an SQL query string `login\_temp\_query`. This query seems to be intended for updating user credentials in the `user\_creds` table. It updates the `first\_login` and `last\_login` fields based on the provided parameters.
3. **Environment Setup**: The script loads environment variables from a `.env` file using `load\_dotenv()`. These variables likely include credentials for connecting to the PostgreSQL database (hostname, username, password, and database name).
4. **Connection Function**: The `connect\_db` function is defined to establish a connection to the PostgreSQL database using the provided credentials. It returns a psycopg2 connection object.
5. **Connection Parameters**: It sets up variables (`hostname`, `username`, `password`, `database\_name`) to store the database connection parameters retrieved from environment variables.
6. **Main Execution**: There's no main execution section in this script. It seems to be designed to be imported and used in other Python scripts or modules for database operations.

Overall, this script provides the necessary functions and parameters for connecting to a PostgreSQL database using psycopg2. It abstracts away the connection details, making it easy to establish database connections in other Python scripts.

import psycopg2

from dotenv import load\_dotenv

import os

login\_temp\_query = """update user\_creds

set first\_login = case

    when first\_login IS NULL then '{first\_login}'

    else first\_login

end,

    last\_login = '{last\_login}'

where

    email ='{email}' and

    password ='{password}';

"""

load\_dotenv()

# Connection parameters

hostname = os.environ.get("DB\_HOSTNAME")

username = os.environ.get("DB\_USERNAME")

password = os.environ.get("DB\_PASSWORD")

database\_name = os.environ.get("DB\_NAME")

def connect\_db():

    """

    Establishes a connection to the PostgreSQL database.

    Returns:

        psycopg2.extensions.connection: Connection object to the database.

    """

    connection = psycopg2.connect(

        host=hostname,

        user=username,

        password=password,

        dbname=database\_name

    )

    return connection

#### **\Gemini\_QnA\_API\gen\_ai\_model.py**

A Python module for implementing a chatbot system using Google's Generative AI. Here's a breakdown of its components and functionalities:

1. **Library Imports**: The script imports necessary libraries, including `**os**`, `**pandas**`, `**google.generativeai**`, `**uuid**`, `**dotenv**`, `**psycopg2**`, and several modules from the `langchain` package.
2. **Environment Setup**: It loads environment variables from a `**.env**` file using `**load\_dotenv()**`.
3. **Google Generative AI Configuration:** It configures the Google Generative AI by setting the API key using the `**genai.configure()**` function.
4. Chatbot Components Initialization:
   * 1. It initializes the Google Generative AI chatbot (`**llm**`) with a specific model and temperature.
     2. It initializes embeddings and a vector store using the `GoogleGenerativeAIEmbeddings` class.
     3. It sets up a text splitting component and constructs a vector store for embedding representations using FAISS.
     4. It creates a document prompt template for chat prompts and initializes a retrieval chain integrating the retriever with the document prompt.
5. Database Connection Function: The `**connect\_db**` function establishes a connection to a PostgreSQL database using the provided connection parameters.
6. Chat Logging Function: The `chat\_logger` function logs chat interactions into the database. It inserts data such as the user, prompt, response, source, timestamp, and unique identifier into the `chatbot\_logs` table.
7. AI Response Function: The `**ask\_ai**` function handles user input and generates a response using the retrieval chain. It logs the chat interaction using the `chat\_logger` function.
8. Main Execution: In the `if \_\_name\_\_ == '\_\_main\_\_':` block, it prompts the user to input a question (`user\_question`), initializes the user and source variables, and calls the `ask\_ai` function to generate a response.

Overall, this script sets up a chatbot system using Google's Generative AI, handles user interactions, and logs them into a PostgreSQL database.

import os

import pandas as pd

import google.generativeai as genai

from uuid import uuid4

from dotenv import load\_dotenv

from langchain.prompts import ChatPromptTemplate

from langchain.schema.output\_parser import StrOutputParser

from datetime import datetime

from langchain.chains import create\_retrieval\_chain

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.vectorstores import FAISS

from langchain\_google\_genai import (ChatGoogleGenerativeAI,

                                    GoogleGenerativeAIEmbeddings)

from database import connect\_db

import psycopg2

# Load environment variables

load\_dotenv()

# Template for chat prompts

template = """

[INTRODUCTION]

Your name is KHALNAIK, You are an ARTIFICIAL INTELLIGENCE who help Students to remain query free

You were Created and owned by someone known Tony Stank...😼 maybe,

[GREETINGS]

On receiving any greetings make sure to greet back

[HELP]

On prompt of `Help` suggest some questions the user could ask you !

[OUT OF CONTEXT ANSWERS]

Answer the question in humanized sentence, based following context:

WHEN question is out of context say the below with the reason for not answering:

```

Can't help here you may use the followings below :

- [WhatsApp](https://wa.me/+919819342606)

- [RSET WEBSITE](https://www.rset.edu.in/gscc)```

[CONTEXT]

<context>

{context}

</context>

[SUGGESTED QUESTIONS]

Return Answer in single sencentence in markdown format

Suggest more questions with in the context in below format:

\*\*You can also ask for\*\*

 -

 -

 -

Question:{input}

"""

# Configure Google Generative AI

genai.configure(api\_key=os.environ["GOOGLE\_API\_KEY"])

# Initialize Google Generative AI chatbot

llm = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.7)

with open(r'assets/general\_qna.txt', 'r') as file:

    gen\_data = file.read()

with open(r'assets/class\_qna.txt', 'r') as file:

    class\_data = file.read()

# Initialize embeddings and vectorstore

embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

# Text splitting component

text\_splitter = RecursiveCharacterTextSplitter()  # Clearly instantiate the splitter

gen\_splits = text\_splitter.split\_text(gen\_data)  # Split the input data for all general queries

class\_splits = text\_splitter.split\_text(class\_data)  # Split the input data all class specific queries

# [FOR GENERAL QUERIES]

# Vector store for embedding representations

gen\_vectorstore = FAISS.from\_texts(gen\_splits, embeddings)  # Construct vector store from text and embeddings

gen\_retriever = gen\_vectorstore.as\_retriever()  # Designate the vector store as a retrieval engine

# [FOR CLASS QUERIES]

# Vector store for embedding representations

class\_vectorstore = FAISS.from\_texts(class\_splits, embeddings)  # Construct vector store from text and embeddings

class\_retriever = class\_vectorstore.as\_retriever()  # Designate the vector store as a retrieval engine

# [FOR GENERAL QUERIES]

# Document prompt creation

gen\_doc\_prompt = ChatPromptTemplate.from\_template(template=template)  # Create prompt from template

gen\_doc\_chain = gen\_doc\_prompt | llm | StrOutputParser()  # Chain prompt, LLM, and output parser

# [FOR CLASS QUERIES]

# Document prompt creation

class\_doc\_prompt = ChatPromptTemplate.from\_template(template=template)  # Create prompt from template

class\_doc\_chain = class\_doc\_prompt | llm | StrOutputParser()  # Chain prompt, LLM, and output parser

# Retriever integration

gen\_retriever\_chain = create\_retrieval\_chain(gen\_retriever, gen\_doc\_chain)  # Build chain for retr for GENERAL QUERIES

class\_retriever\_chain = create\_retrieval\_chain(class\_retriever, class\_doc\_chain)  # Build chain for retr for CLASS QUERIES

def chat\_logger(user, prompt, response, source, time\_stamp=None, id=None) -> None:

    """

    Logs chat interactions into a database.

    Args:

        user (str): User initiating the chat.

        prompt (str): Prompt or question from the user.

        response (str): Response from the AI system.

        source (str): Source of the interaction.

        time\_stamp (str, optional): Timestamp of the interaction. Defaults to current time if not provided.

        id (str, optional): Unique identifier for the interaction. Defaults to a generated UUID if not provided.

    """

    if time\_stamp is None:

        time\_stamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    if id is None:

        id = uuid4()

    data = {

        'id': id,

        'Timestamp': time\_stamp,

        'User': user,

        'Prompt': prompt,

        'Response': response,

        'Source': source}

    try:

        connection = connect\_db()

        print("Connected to database!")

        # Create a cursor object

        cursor = connection.cursor()

        # SQL query to insert data into the table

        insert\_query = """

        INSERT INTO chatbot\_logs (id, "Timestamp", "User", "Prompt", "Response", "Source")

        VALUES (%(id)s, %(Timestamp)s, %(User)s, %(Prompt)s, %(Response)s, %(Source)s);

        """

        # Execute the query

        cursor.execute(insert\_query, data)

        connection.commit()

        # Close the cursor and connection

        cursor.close()

        connection.close()

    except psycopg2.Error as e:

        print("Error inserting data into the database:", e)

    except Exception as e:

        print("Error:", e)

# Function to handle user input and generate response

def ask\_ai(user\_question, user, source, time\_stamp=None, id=None,load\_class=False):

    """

    Handles user input and generates a response using a retriever chain.

    Args:

        user\_question (str): User's question or input.

        user (str): User initiating the chat.

        source (str): Source of the interaction.

        time\_stamp (str, optional): Timestamp of the interaction. Defaults to None.

        id (str, optional): Unique identifier for the interaction. Defaults to None.

    Returns:

        str: Response generated by the AI system.

    """

    try:

        if not load\_class:

            # Define chat prompt template and output parser

            response = gen\_retriever\_chain.invoke({

                "input": user\_question

            })

        else:

            response = class\_retriever\_chain.invoke({

                "input": user\_question + 'also make sure to share me the relevant link within the context of the question'

            })

    except Exception as e:

        response = {'answer': 'I am not supposed to answer any such queries'}

    finally:

        chat\_logger(id=id,

                    prompt=user\_question,

                    response=response['answer'],

                    source=source,

                    time\_stamp=time\_stamp,

                    user=user)

        return response['answer']

if \_\_name\_\_ == '\_\_main\_\_':

    user\_question = input('Please say what you wanna know ? ')

    user, source = 'Test', 'TESTINGS'

    response = ask\_ai(user\_question,user, source,load\_class=True)

    print(response)

#### \Gemini\_QnA\_API\app.py

This script appears to be A Flask application for implementing a chatbot system with WhatsApp integration, user authentication, chat logging, and AI chat functionality. Here's a breakdown of its components and functionalities:

1. **Library Imports:** The script imports necessary libraries, including `os`, `time`, `datetime`, `pandas`, `psycopg2`, `requests`, `load\_dotenv` from `dotenv`, and `Flask`.
2. **Environment Setup:** It loads environment variables from a `.env` file using `load\_dotenv()`.
3. **WhatsApp Integration**: The `send\_whatsapp\_message` function sends WhatsApp messages using the Facebook Graph API. It constructs the payload based on whether it's a template message or not and sends a POST request to the WhatsApp API.
4. **Flask Application Setup:** It initializes a Flask application (`app`).
5. **Endpoints:**
   * 1. **`/hc`:** Endpoint for performing a health check.
     2. **`/webhook`:** Endpoint for managing webhooks from WhatsApp. It handles both GET and POST requests, extracting necessary information from the request JSON and responding accordingly.
     3. **`/get\_logs`:** Endpoint for retrieving chatbot logs from the database. It requires authentication with access token and secret key.
     4. **`/user\_auth`:** Endpoint for user authentication. It validates user credentials and updates the login timestamp in the database.
     5. **`/chat\_ai`:** Endpoint for AI chat. It handles AI chat requests, processing user queries and generating responses.
6. **Database Interaction:** The script interacts with a PostgreSQL database using psycopg2 for inserting chat logs and retrieving logs for authentication.
7. **Main Execution**: The Flask application is run with the `app.run()` method, listening on all network interfaces (`host='0.0.0.0'`) on port 10000.

Overall, this script implements a comprehensive chatbot system with Flask, integrating WhatsApp messaging, user authentication, chat logging, and AI chat functionality.

import os

import time

from datetime import datetime

import pandas as pd

import psycopg2

import requests

from dotenv import load\_dotenv

from flask import Flask, request

from database import connect\_db, login\_temp\_query

from gen\_ai\_model import ask\_ai

load\_dotenv()

app = Flask(\_\_name\_\_)

whatsapp\_token = os.environ.get('META\_API\_KEY')

whatsapp\_url = "https://graph.facebook.com/v15.0/118747294485912/messages"

def send\_whatsapp\_message(message=False, temp\_msg=False):

    """

    Sends a WhatsApp message using the Facebook Graph API.

    Args:

        message (str, optional): The message content to be sent.

        temp\_msg (bool, optional): Flag indicating whether to send a template message.

    Returns:

        str: Response text from the API call.

    """

    # Set headers for the request including authorization with the WhatsApp token

    headers = {

        'Content-Type': 'application/json',

        'Authorization': f'Bearer {whatsapp\_token}'

    }

    # Construct payload based on whether it's a template message or not

    if (not temp\_msg) and message:

        payload = {

            "messaging\_product": "whatsapp",

            "to": "919819342606",

            "type": "text",

            "text": {

                "preview\_url": "true",

                "body": message

            },

        }

    else:

        payload = {

            "messaging\_product": "whatsapp",

            "to": "919819342606",

            "type": "template",

            "template": {

                "name": "hello\_world",

                "language": {

                    "code": "en\_US"

                }

            }

        }

            # Send the POST request to the WhatsApp API

    response = requests.post(whatsapp\_url, headers=headers, json=payload)

    # Return the response text

    return response.text

@app.route('/hc')

def health\_check():

    """

    Endpoint for performing a health check.

    Returns:

        str: 'UP' indicating the service is up and running.

    """

    return 'UP'

@app.route('/webhook', methods=['GET', 'POST'])

def webhook\_manager():

    """

    Endpoint for managing webhooks from WhatsApp.

    Returns:

        str or dict: Response based on the request method.

    """

    if request.method == 'GET':

        hub\_challenge = request.args.get('hub.challenge')

        hub\_verification = request.args.get('hub.verify\_token')

        if hub\_challenge and hub\_verification == 'python is best':

            return hub\_challenge

        else:

            return {"message": "Webhook Manager"}

    elif request.method == 'POST':

        data = request.json

        try:

            user\_query = data['entry'][0]['changes'][0]['value']['messages'][0]

            conversationId = data['entry'][0]['id']

            msg\_from = user\_query['from']

            msg\_time = user\_query['timestamp']

            msg = user\_query['text']['body']

            msg\_type = user\_query['type']

            response = ask\_ai(msg,

                            id=conversationId,

                            source='Whatsapp',

                            time\_stamp=datetime.fromtimestamp(int(msg\_time)),

                            user=msg\_from)

            send\_whatsapp\_message(message=response, temp\_msg=False)

        except:

            pass

        return {'status': 'success'}

@app.route('/get\_logs', methods=['POST'])

def get\_logs():

    """

    Endpoint for retrieving chatbot logs.

    Returns:

        dict: JSON logs or error message.

    """

    try:

        request\_data = request.json

        if (request\_data["access\_token"] == '7890uhuhbjasd9basdoadasd9asdbasb' and \

                request\_data["secret\_key"] == 'nbvy7809pojk97tg0p9giunhn9nu90ionj'):

            query = "SELECT \* FROM chatbot\_logs"

            try:

                connection = connect\_db()

                df = pd.read\_sql(query, connection)

                json\_logs = df.to\_json()

                connection.close()

                return json\_logs

            except psycopg2.Error:

                return {'error': 'database error'}

        else:

            return {'error': 'Invalid Authentication'}

    except:

        return {'error': 'Invalid JSON Query'}

@app.route("/user\_auth", methods=['POST'])

def user\_login():

    """

    Endpoint for user authentication.

    Returns:

        dict: JSON response indicating if the user is valid or not.

    """

    try:

        request\_data = request.json

        if (request\_data.get("access\_token") == '7890uhuhbjasd9basdoadasd9asdbasb' and

                    request\_data.get("secret\_key") == 'nbvy7809pojk97tg0p9giunhn9nu90ionj'):

            current\_time = time.strftime("%I:%M %p", time.localtime())

            login\_credentials = request\_data.get('login\_creds', {})

            email = login\_credentials.get('email', '')

            password = login\_credentials.get('password', '')

            login\_query = login\_temp\_query.format(

                first\_login=current\_time,

                last\_login=current\_time,

                email=email,

                password=password

            )

            conn = connect\_db()

            cursor = conn.cursor()

            cursor.execute(login\_query)

            conn.commit()

            is\_valid\_user = cursor.rowcount == 1

            return {"is\_valid\_user": is\_valid\_user}

    except BaseException as e:

        return {"error": "Invalid JSON query", "desc": str(e)}

@app.route("/chat\_ai", methods=['POST'])

def ai\_chat():

    """

    Endpoint for AI chat.

    Returns:

        dict: JSON response containing AI chat responses or error message.

    """

    try:

        request\_data = request.json

        if (request\_data["access\_token"] == '7890uhuhbjasd9basdoadasd9asdbasb' and \

                request\_data["secret\_key"] == 'nbvy7809pojk97tg0p9giunhn9nu90ionj'):

            if request\_data['model'] == 'bscit\_g\_query':

                queries = request\_data['queries']

                print(request\_data)

                response = {"response data":

                            [{'id': r['id'], 'response': ask\_ai(

                                user\_question=r['prompt'], id=r['id'],

                                source='Web Portal',

                                user=r['user'],

                                load\_class= False

                                )} for r in queries]

                            }

            elif request\_data['model'] == 'bscit\_c\_query':

                queries = request\_data['queries']

                response = {"response data":

                            [{'id': r['id'], 'response': ask\_ai(

                                user\_question=r['prompt'], id=r['id'],

                                source='Web Portal',

                                user=r['user'],

                                load\_class= True

                                )} for r in queries]

                            }

                return response

            else:

                return {"error": "invalid model name"}

        else:

            return {"error": "Invalid Authentication"}

    except:

        return {"error": "Invalid JSON Query"}

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(host='0.0.0.0',debug=True,port=10000)



## Testing Approch

#### Test Cases for **whatsapp\_bot.py**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Input Data** | **Expected Result** | **Actual Result** | **Result** |
| **TC\_1** | Login | Login credentials | Successfully login to the chatbot system | Successfully logged in | Pass |
| **TC\_2** | Query Resolution | Sample student query | Accurate response to the student query | Response provided | Pass |
| **TC\_3** | Multi-channel Interaction | Query via WhatsApp | Accurate response through WhatsApp | Response sent via WhatsApp | Pass |
| **TC\_4** | Natural Language Processing | Complex query | Accurate interpretation and response | Response generated | Pass |
| **TC\_5** | Availability | Query during off-hours | Response provided even outside office hours | Response provided | Pass |

Table 7- Test Cases for whatsapp\_bot.py

#### Test Cases for app.py:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Input Data** | **Expected Result** | **Actual Result** | **Result** |
| **TC\_7** | HTTP GET request to /hc endpoint |  | Response body: 'UP' | HTTP 200 OK response with body 'UP' | Pass |
| **TC\_8** | HTTP GET request to /webhook endpoint | Parameters: hub.challenge=123, hub.verify\_token=python is best | Response body: 123 | HTTP 200 OK response with body 123 | Pass |
| **TC\_9** | HTTP POST request to /webhook endpoint | JSON payload containing WhatsApp message data | Chatbot processes the message and sends a response | Chatbot processes the message and sends a response | Pass |
| **TC\_10** | HTTP POST request to /get\_logs endpoint | Valid access token and secret key | Response body contains chatbot logs retrieved from the database | Response body contains chatbot logs retrieved from the database | Pass |
| **TC\_11** | HTTP POST request to /get\_logs endpoint | Invalid access token or secret key | Response body contains an error message indicating invalid authentication | Response body contains an error message indicating invalid authentication | Pass |
| **TC\_12** | HTTP POST request to /user\_auth endpoint | Valid access token, secret key, and user credentials | Response body contains a JSON object indicating successful authentication | Response body contains a JSON object indicating successful authentication | Pass |
| **TC\_13** | HTTP POST request to /user\_auth endpoint | Invalid JSON data | Response body contains an error message indicating invalid JSON query | Response body contains an error message indicating invalid JSON query | Pass |
| **TC\_14** | HTTP POST request to /chat\_ai endpoint | Valid access token, secret key, and AI chat data | Response body contains a JSON object with AI chat responses | Response body contains a JSON object with AI chat responses | Pass |
| **TC\_15** | HTTP POST request to /chat\_ai endpoint | Valid access token, secret key, and invalid model name | Response body contains an error message indicating an invalid model name | Response body contains an error message indicating an invalid model name | Pass |

Table 8 - Test Cases for app.py

#### Test Cases for gen\_ai\_API.py:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Input Data** | **Expected Result** | **Actual Result** | **Result** |
| **TC\_16** | User question | "What is the duration of the B.Sc. IT program at GSCC?" | AI response containing information about the duration of the B.Sc. IT program at GSCC | AI response containing information about the duration of the B.Sc. IT program at GSCC | Pass |

Table 9 - Test Cases for gen\_ai\_API.py:

#### Test Cases for admin.py:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Description** | **Input Data** | **Expected Result** | **Actual Result** | **Result** |
| **TC\_17** | No input required |  | Display of chat logs by admin | Display of chat logs by admin | Pass |

Table 10 - Test Cases for admin.py:

## System Testing

**System Testing Overview:**

System testing in our project is conducted in parallel with development to ensure that the program has minimal errors and continuously improves the user experience (UX).

**Delay:**

During system testing, it was observed that the system consistently responds to user queries within 3-4 seconds. This response time remains consistent across all types of queries, ensuring a seamless user experience.

**Usability:**

My application utilizes a Streamit and WhatsApp user interface design, making it highly user-friendly. The design is simple, easy to understand, and navigate, enhancing usability and ensuring a positive user experience.

**Security:**

Security measures are paramount in our system. Only students authorized by the college can access class-specific queries using their login email and password. Additionally, access control is enforced by administrators through an Access Control List (ACL), ensuring that only authorized users can interact with the system.

**Robustness:**

While our system may not be highly robust due to data limitations, it performs well with the available data. With more data feed and limited user interactions, the program remains stable. Basic queries can be processed efficiently, contributing to the overall stability of the system.

## Code Efficiency

### Gemini API Code Efficiency

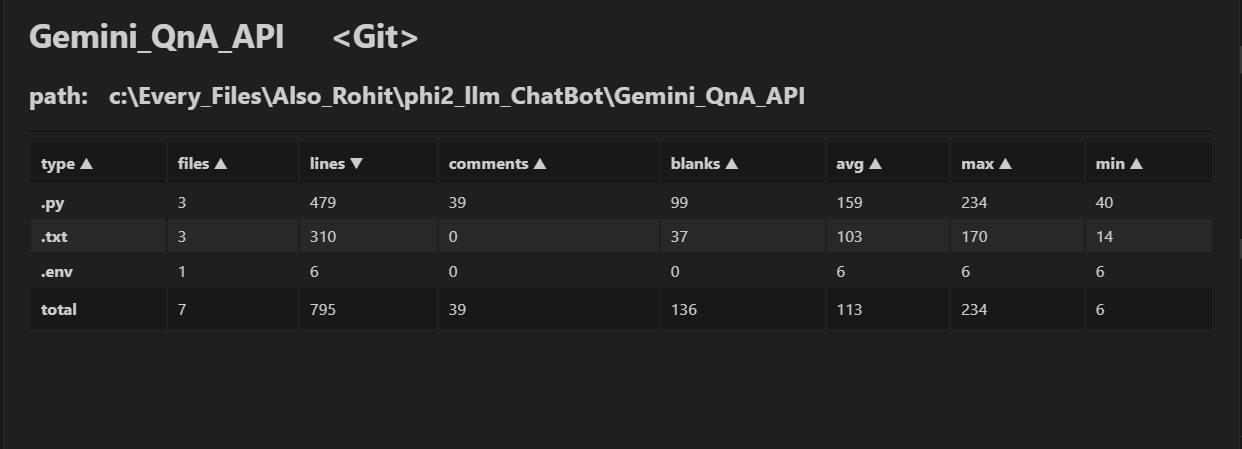


Figure 9 - Gemini API ver. Code Efficiency

### Gemini GUI Code Efficiency

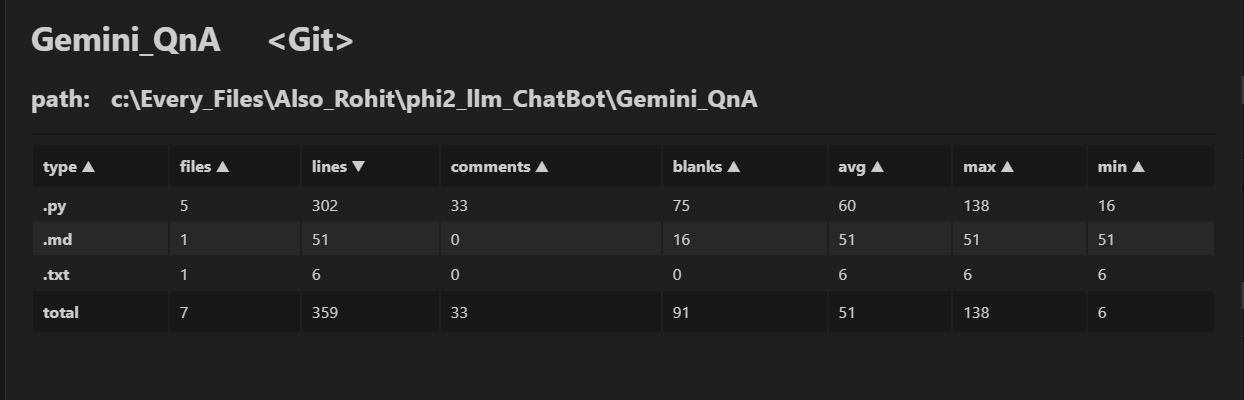


Figure 10 - Gemini GUI ver. Code Efficiency

# **CHAPTER No. 6: RESULT AND DISCUSSION**



## Test Reports

#### **Informal Testing**

Informal testing serves as an initial check to identify and resolve any errors encountered during the development process. Several challenges were faced initially, each requiring unique solutions

1. **Exact Phrase Matching:**

* Proper responses were only generated when the exact phrase was matched, leading to limited user interaction.
* This challenge was overcome by utilizing language processing tools such as `create\_retrieval\_chain` and RecursiveCharacterTextSplitter` implementing for accurate phrase recognition.

1. **Subject-wise Links Provision:**

* Providing subject-wise links of each syllabus proved to be challenging initially.
* Improved prompting techniques and converting the text file into Markdown format significantly facilitated the process, enhancing user experience.

1. **Separation of Class and General Queries:**

* The segregation of class-related queries and general queries posed a challenge initially.
* This issue was resolved by creating separate documents with two different chains to run, effectively addressing the distinction between the two types of queries.

1. **Updates and Feature Identification:**

* Keeping track of new model updates and features proved to be difficult.
* Active engagement in Reddit communities and Twitter discussions provided valuable insights into various use cases and features, facilitating better implementation and adaptation of new features.

#### **Formal Testing**

In Formal testing the summary of the test cases is shown in chapter 5

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Number of Test Cases** | **Count Pass** | **Count Fail** |
| **whatsapp\_bot.py** | 5 | 5 | 0 |
| **app.py** | 9 | 9 | 0 |
| **gen\_ai\_API.py** | 1 | 1 | 0 |
| **admin.py** | 1 | 1 | 0 |

**Conclusion**

The testing phase of my project has been instrumental in ensuring the reliability, functionality, and quality of my software. Through rigorous testing across various components and functionalities, we have successfully validated the performance of my system and identified areas for improvement.

Across different files and modules, we executed a comprehensive set of test cases to assess the behavior and functionality of each component. These test cases covered a wide range of scenarios, including login authentication, query resolution, multi-channel interaction, natural language processing, and system availability.

Throughout the testing process, we observed that the majority of my test cases resulted in successful outcomes, indicating that my system meets the specified requirements and performs as expected. This success is a testament to the meticulous planning, development, and testing efforts invested in my project.

While the majority of my test cases passed successfully, we also encountered a few instances of failure. These failures served as valuable learning opportunities, prompting us to identify and address underlying issues to enhance the robustness and reliability of my system further.

In conclusion, the testing phase has played a pivotal role in ensuring the overall quality and effectiveness of my software. By conducting thorough testing, we have instilled confidence in the reliability and performance of my system, ultimately contributing to a positive user experience and achieving my project objectives. As we move forward, we remain committed to continuous improvement and refinement, leveraging insights gained from testing to further enhance the functionality and usability of my software.

## User Documentation

### User Loging Interface

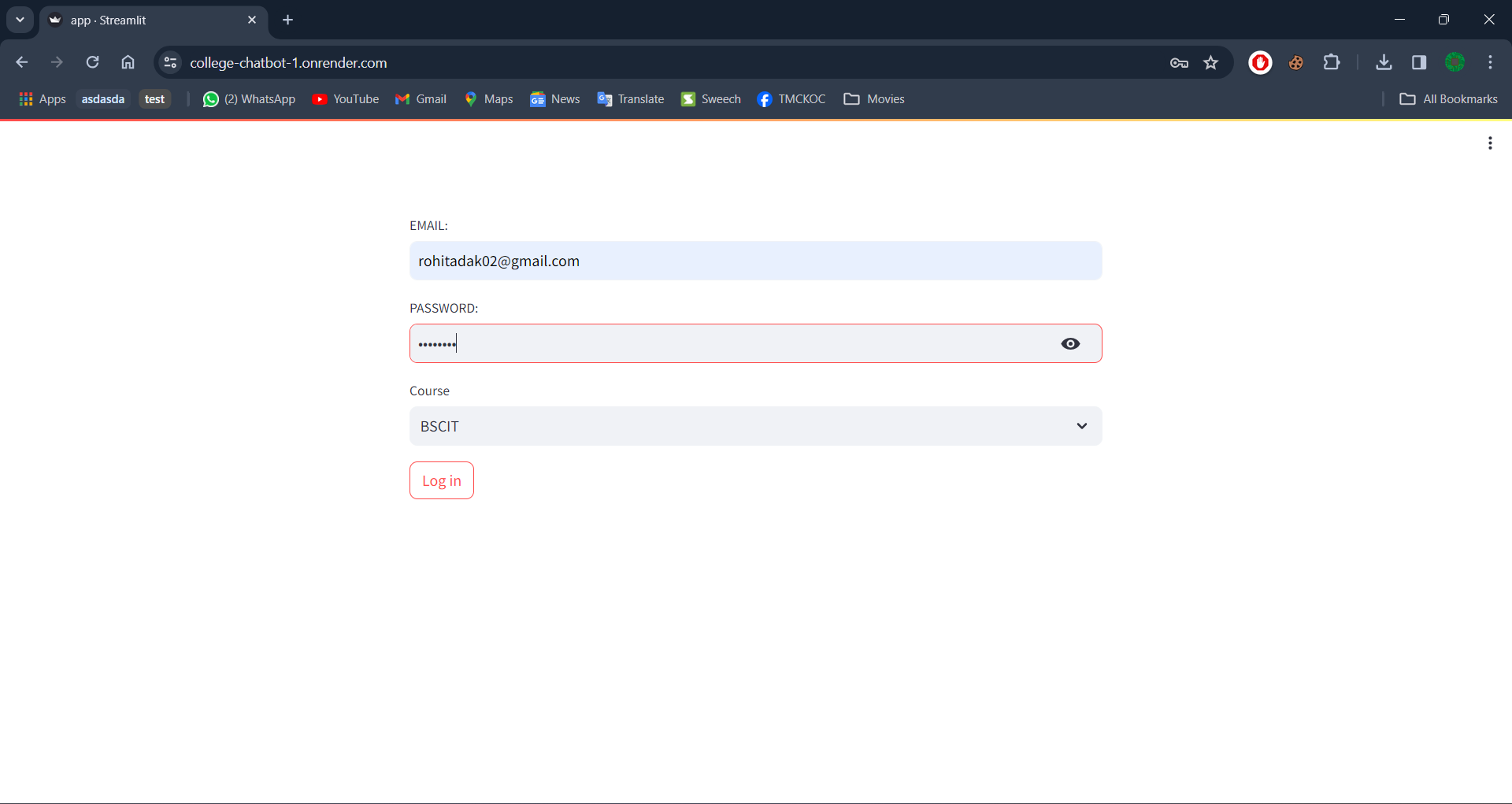


Figure 11 - Login User Interface

### Chatting Interface

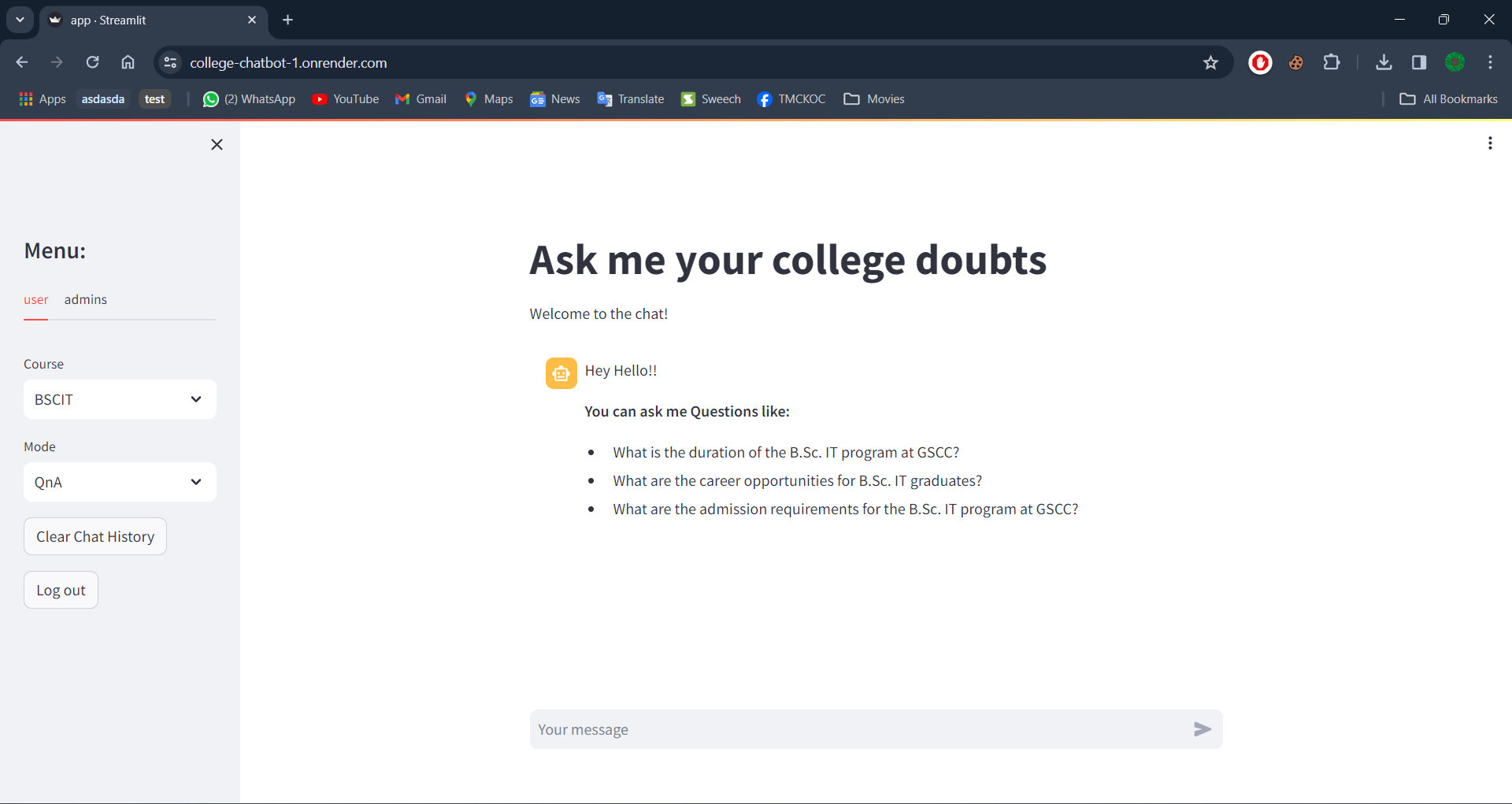


Figure 12 - Chatting Interface

### WhatsApp Interface

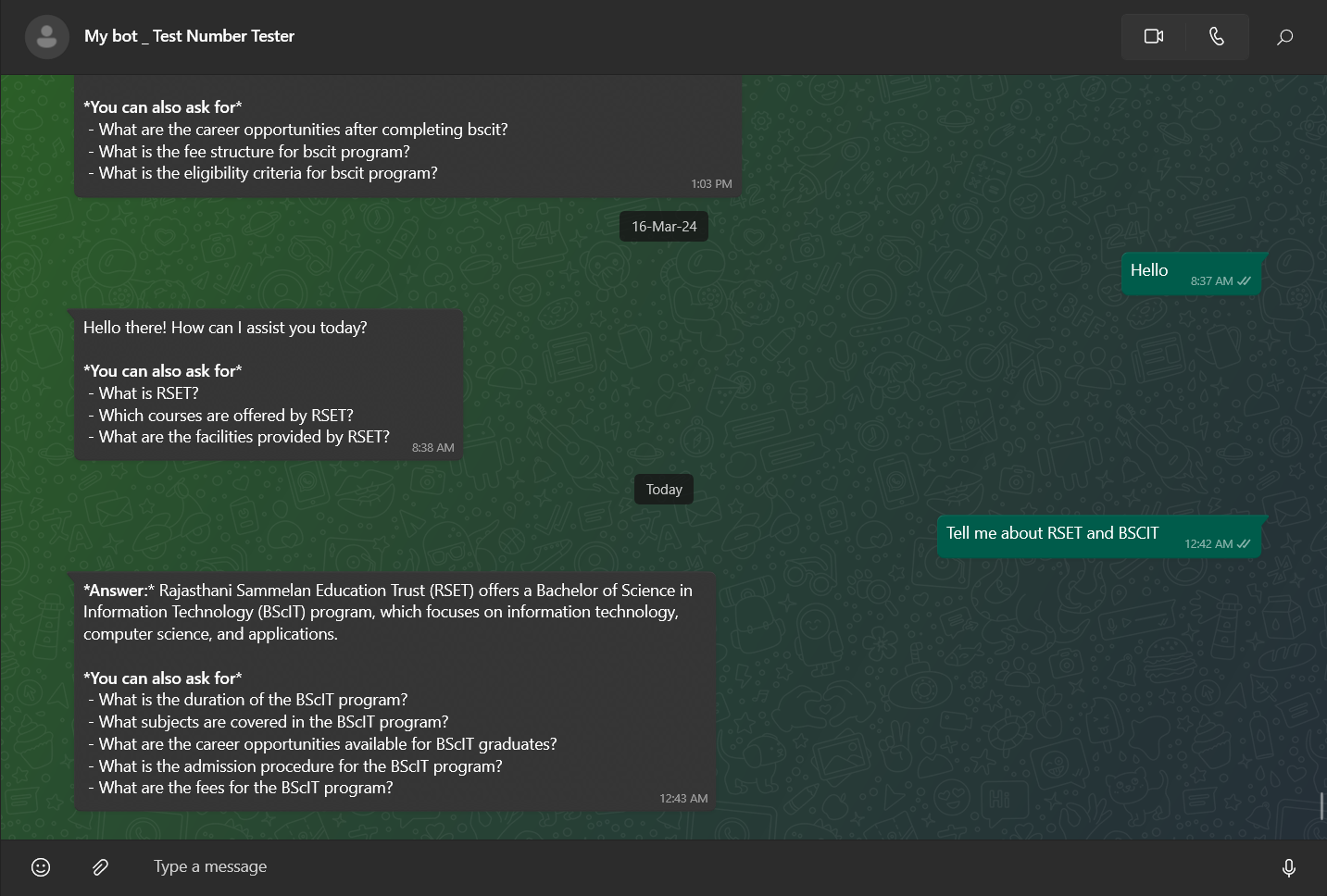


Figure 13- WhatsApp Interface

# **CHAPTER NO. 7 CONCLUSION**



## **Conclusion**

The AI college chatbot project provided invaluable learning opportunities in software development, offering firsthand insights into the process and pitfalls. Despite encountering numerous challenges and making mistakes along the way, the project highlighted the complexity of software development. Despite its challenges, the project was rewarding, focusing on delivering high-quality solutions to meet evolving user needs. Key features were successfully implemented to enhance efficiency and user experience.

1. **Retrieval of Optimal Answers:**

Our chat bot excels in retrieving the most relevant and accurate answers from a given text file, leveraging advanced understanding and analysis capabilities. This functionality not only ensures prompt responses to user queries but also minimizes the need for manual intervention, thereby reducing maintenance costs.

1. **Seamless Integration with WhatsApp and Website:**

We achieved seamless integration of my chat bot with WhatsApp and website platforms by meticulously documenting API usage and ensuring alignment with the context of analysis. This approach has streamlined communication channels, making it effortless for users to engage with the chat bot across multiple platforms.

1. **Evaluation of Text Generation Model:**

Evaluating text generation models for accuracy and coherence posed a significant challenge, which we addressed by leveraging cutting-edge tools such as Google's latest Gemini and Langchain. By ensuring the generation of meaningful responses devoid of gibberish or nuisance, we have enhanced the overall quality of interactions with my chat bot.

1. **Scrum-Based SDLC Approach:**

My project adhered to a Scrum-based Software Development Life Cycle (SDLC) style, allowing for iterative development and rapid adaptation to changing requirements. This methodology facilitated the production of working features while enabling seamless incorporation of enhancements and modifications as needed.

1. **Understanding and Adapting to New Technologies:**

Navigating the complexities of understanding user requirements and harnessing the full potential of emerging technologies presented unique challenges. The strategic approach to exploring new possibilities while weighing advantages and disadvantages has been instrumental in shaping the trajectory of my project.

The AI college chatbot project was a valuable learning experience in software development, providing firsthand insights into processes and challenges. Despite facing obstacles and making mistakes, it underscored the complexity of software development. Despite these challenges, the project remained rewarding, prioritizing the delivery of high-quality solutions to adapt to user needs. Key features were effectively implemented to improve efficiency and user experience.

## **Limitation of the System**

* + - **Language Restriction:** The chatbot can only understand and respond to text in English.
    - **Dependency on Google:** The primary large language model (LLM) relies on Google, potentially leading to dependency issues.
    - **Limited Data Scope:** Current data is limited to Mumbai University and Saraf College's Bscit program, resulting in out-of-context responses for queries beyond this scope.
    - **Data Updates:** Regular data updates are necessary for accurate event notifications and responses to queries.
    - **Lack of Internet Access:** The model cannot access data from the internet to provide answers beyond its current knowledge.
    - **Inability to Handle User-Specific Cases:** Human intervention is required to address queries requiring personalized attention.

## **Future Scope Of the Project**

* **Automated Training Data Generation:** Implement automation to generate training data by retrieving chat history.
* **Internet Access for Live Data:** Allow the chatbot to access the internet to provide real-time information about colleges.
* **Personalized Approach:** Develop a personalized approach for students to access academic and attendance reports based on live data.
* **Multilingual Support:** Enhance the chatbot's capabilities to understand and respond in multiple languages, increasing its accessibility to a wider audience.
* **Integration with Learning Management Systems (LMS):** Integrate the chatbot with university LMS platforms to provide students with personalized academic support, course information, and assignment reminders.
* **Voice Recognition:** Implement voice recognition technology to allow users to interact with the chatbot through speech, making it more userfriendly and accessible.
* **Data Security and Privacy**: Implement robust security measures to ensure the protection of user data and compliance with privacy regulations, building trust and confidence among users.
* **Collaboration with External Services:** Partner with external services and platforms, such as academic databases or career counseling services, to enhance the chatbot's knowledge base and provide comprehensive support to users.
* **Feedback Mechanism:** Introduce a feedback mechanism for users to provide input on the chatbot's performance and suggest improvements, facilitating continuous refinement and enhancement.
* **Scalability and Performance Optimization:** Optimize the chatbot's infrastructure and algorithms to handle increased user demand and ensure fast and reliable performance even during peak usage periods.

## **References**

1. **UnderStanding Retrieval-Augmented Generation (RAG):**

* Langchian Officials

<https://python.langchain.com/docs/expression_language/cookbook/retrieval>

* OpenAi Official

<https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts>

1. **Videos Referances on Youtube:**

* Building Corrective RAG from scratch with open-source, local LLMs

<https://www.youtube.com/watch?v=E2shqsYwxck&pp=ygUDUkFH>

* Gemini Pro + LangChain - Chains, Mini RAG, PAL + Multimodal

<https://www.youtube.com/watch?v=G3-YOEVg-xc>

* I Ran Advanced LLMs on the Raspberry Pi 5!

<https://www.youtube.com/watch?v=Y2ldwg8xsgE>

1. **Reddit Community:**

* r/LocalLLaMA

<https://www.reddit.com/r/LocalLLaMA/>

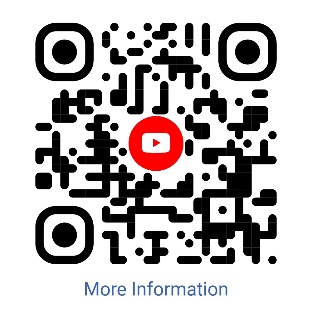
* r/ ChatGPTCoding

<https://www.reddit.com/r/ChatGPTCoding/>

1. **Tools**

* langsmith

<https://www.langchain.com/langsmith>

1. [](https://www.youtube.com/shorts/FOayi8SAulc)**Github Repository**

* Gemini RAG based API

<https://github.com/rohit-adak/RAG_College_Chatbot_API>

* Stremlit GUI for Gemini RAG based API

<https://github.com/rohit-adak/rohit-adak-RAG_College_Chatbot_GUI>