INDUSTRY ORIENTED MINI PROJECT

Report

On

AI-POWERED CHATBOT FOR STREAMLINING ADMISSION ENQUIRIES

Submitted in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

By

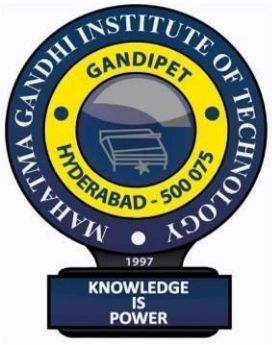
Abdul Rasheed-22261A1202

Sharon Kanaka - 22261A1229

Under the guidance of

Dr. M. RUDRA KUMAR

Professor, Department of IT



DEPARTMENT OF INFORMATION TECHNOLOGY

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

(AUTONOMOUS)

(Affiliated to JNTUH, Hyderabad; Eight UG Programs Accredited by NBA; Accredited by NAAC with 'A++' Grade)

Gandipet, Hyderabad, Telangana, Chaitanya Bharati (P.O), Ranga Reddy District, Hyderabad– 500075, Telangana

2024-2025

CERTIFICATE

This is to certify that the **Industry Oriented Mini Project** entitled **AI-Powered Chatbot For Streamlining Admission Enquiries** submitted by **Abdul Rasheed(22261A1202), Sharon Kanaka(22261A1229)** in partial fulfillment of the requirements for the Award of the Degree of Bachelor of Technology in Information Technology as specialization is a record of the bona fide work carried out under the supervision of **Dr. M. RUDRA KUMAR** and this has not been submitted to any other University or Institute for the award of any degree or diploma.

Internal Supervisor:

Dr. M. Rudra Kumar
Professor
Dept. of IT

IOMP Supervisor:

Dr. U Chaitanya
Assistant Professor
Dept. of IT

EXTERNAL EXAMINAR

Dr. D. Vijaya Lakshmi
Professor and HOD
Dept. of IT

DECLARATION

We hear by declare that the **Industry Oriented Mini Project** entitled **AI-Powered Chatbot For Streamlining Admission Enquiries** is an original and bona fide work carried out by us as a part of fulfilment of Bachelor of Technology in Information Technology, Mahatma Gandhi Institute of Technology, Hyderabad, under the guidance of **Dr. M. Rudra Kumar**, **Professor**, Department of IT, MGIT.

No part of the project work is copied from books /journals/ internet and wherever the portion is taken, the same has been duly referred in the text. The report is based on the project work done entirely by us and not copied from any other source.

Abdul Rasheed- 22261A1202

Sharon Kanaka -22261A1229

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without introducing the people who made it possible and whose constant guidance and encouragement crowns all efforts with success. They have been a guiding light and source of inspiration towards the completion of the **Industry Oriented Mini Project**.

We would like to express our sincere gratitude and indebtedness to our Internal supervisor **Dr. M. Rudra Kumar**, **Professor**, Dept. of IT, who has supported us throughout our project with immense patience and expertise.

We are also thankful to our honourable Principal of MGIT **Prof. G. Chandramohan Reddy** and **Dr. D. Vijaya Lakshmi**, Professor and HOD, Department of IT, for providing excellent infrastructure and a conducive atmosphere for completing this **Industry Oriented Mini Project** successfully.

We are also extremely thankful to our IOMP supervisor **Dr. U. Chaitanya**, Assistant Professor, Department of IT, and senior faculty **Mrs. B. Meenakshi** Department of IT for their valuable suggestions and guidance throughout the course of this project.

We convey our heartfelt thanks to the lab staff for allowing us to use the required equipment whenever needed.

Finally, we would like to take this opportunity to thank our families for their support all through the work. We sincerely acknowledge and thank all those who gave directly or indirectly their support for completion of this work.

Abdul Rasheed- 22261A1202

Sharon Kanaka -22261A1229

ABSTRACT

In today's competitive academic environment, engineering institutes are witnessing a surge in admission-related inquiries from prospective students and their parents. Managing this high volume of queries manually often leads to delays, miscommunication, and increased workload for administrative staff. To overcome these challenges, our project proposes the development of an AI-powered admission chatbot that can efficiently handle and automate the inquiry process.

This chatbot, built using FastAPI, LangChain, Qdrant, and Google Gemini, is designed to provide instant, accurate, and context-aware responses to text-based queries in English. It can answer a wide range of frequently asked questions related to admission procedures, eligibility criteria, fee structure, scholarships, academic curriculum, hostel facilities, cutoff scores, and placement records. By handling repetitive inquiries automatically, the system reduces human intervention and ensures faster, more consistent communication.

In addition to improving response time and reducing staff workload, the chatbot also collects data on user interactions to identify common concerns and query patterns. These analytics offer valuable insights that can help the institution refine its admission process and address areas of concern. Ultimately, this AI-driven solution aims to create a seamless, scalable, and efficient communication platform that enhances user experience and institutional productivity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 MOTIVATION

The increasing volume of admission-related inquiries at engineering institutes presents a significant challenge for both prospective students and administrative staff. Traditional methods of handling these queries—primarily manual responses via phone calls, emails, or in-person visits—are often slow, inconsistent, and resource-intensive. This leads to delays in providing timely information, increased workload for administrative personnel, and a frustrating experience for students and their families seeking critical admission details.

In today's digital era, prospective applicants expect instant, accurate, and round-the-clock assistance. However, existing communication systems in many institutes lack scalability and efficiency to meet this demand, often resulting in repetitive queries overwhelming the admission offices and diverting resources from other important tasks.

This project is motivated by the need to automate and enhance the admission inquiry process through an intelligent, AI-driven chatbot system. By leveraging **Retrieval-Augmented Generation (RAG)** technology, the chatbot can understand user queries and retrieve contextually relevant information from curated datasets or knowledge bases. It then uses generative models to compose accurate and natural language responses to a wide range of frequently asked questions in English. These questions may pertain to admission procedures, eligibility criteria, fee structures, scholarships, curriculum, hostel facilities, cutoff marks, and placement opportunities.

RAG allows the system to combine the benefits of real-time information retrieval and generative response capabilities, ensuring that answers are both factually grounded and user-friendly. This hybrid approach significantly improves the chatbot's ability to handle complex or dynamic queries while maintaining consistency and coherence.Such a system promises to improve accessibility and responsiveness, reduce administrative workload, and provide a consistent, high-quality user experience. Additionally, by capturing interaction data, the chatbot will enable institutions to gain valuable insights into user concerns and optimize their communication strategies. Ultimately, this project aims to transform admission counseling from a manual, labour-intensive process into an efficient, scalable, and user-centric digital solution, addressing the growing needs of modern educational institutions and their stakeholders.

.

1.2 PROBLEM STATEMENT

Engineering institutes face an overwhelming number of admission-related queries from prospective students and their parents, covering a wide range of topics such as eligibility criteria, admission procedures, fee structures, scholarships, curriculum details, hostel facilities, cutoff scores, and placement opportunities. Currently, these queries are handled manually by administrative staff through phone calls, emails, and in-person interactions, which is time-consuming, inefficient, and prone to delays.

This manual process leads to several challenges:

- **High Administrative Workload:** The repetitive nature of frequently asked questions consumes significant time and resources, limiting staff availability for more complex tasks.
- **Delayed Responses:** Prospective students often experience long wait times for their queries to be addressed, causing frustration and negatively impacting their decision-making process.
- **Inconsistent Information Delivery:** Variability in responses due to human error or lack of standardized communication can result in confusion and misinformation.
- **Limited Accessibility:** Office hours and communication channels restrict the availability of timely assistance, especially outside working hours.
- **Lack of Data Insights:** The current system does not systematically capture or analyze inquiry data to identify common concerns or improve institutional services.

These issues necessitate the development of an automated, intelligent solution capable of providing instant, accurate, and consistent responses to admission-related queries, thereby enhancing the overall efficiency of the admission process and improving the experience of prospective students and their families.

1.3 EXISTING SYSTEM

Currently, most engineering institutes rely on traditional methods for handling admission-related inquiries, including:

- **Manual Communication Channels:** Queries are addressed through telephone calls, emails, and face-to-face interactions with admission office staff. This process is labor-intensive and often results in long response times due to high query volumes.

- **Static FAQ Pages:** Institutes commonly maintain FAQ sections on their websites that list common questions and answers related to admissions. While these pages provide useful information, they lack interactivity and personalization, making it difficult for users to quickly find answers to their specific concerns.
- **Email Support Systems:** Some institutes use email ticketing systems to manage inquiries. Although these systems help in organizing queries, responses are still manual and time-consuming, leading to delays.

These existing systems do not adequately address the challenges of scalability, real-time response, and user engagement, thereby creating an opportunity for an AI-powered, NLP-based chatbot that can provide instant, accurate, and contextual answers to admission-related queries in English. This project aims to fill this gap by developing a more advanced, adaptive, and user-friendly chatbot system.

1.3.1 Limitations

- **Manual and Time-Consuming Process:** Admission inquiries are handled manually, causing delays, increased workload for staff, and inconsistent responses.

- **Limited Accessibility and Interactivity:** Existing FAQ pages and communication channels lack real-time, personalized interaction and are not available 24/7, restricting user convenience and engagement.

1.4 PROPOSED SYSTEM

The implemented backend system is a retrieval-augmented generation (RAG) chatbot API designed to deliver accurate, context-aware answers by combining vector-based document retrieval with advanced language generation models. A core feature of this system is its ability to efficiently process and update data from CSV documents, which can be refreshed to incorporate the latest information, ensuring responses remain relevant and up to date.

On startup, the system loads and preprocesses textual data from CSV files using a loader and text splitter, breaking the data into manageable chunks to optimize semantic search capabilities. These chunks are embedded using Google's Generative AI embedding model and stored in a Qdrant vector database. This setup enables rapid similarity searches that retrieve contextually relevant document fragments in response to user queries.

To maximize retrieval quality, the system reformulates the user's query into multiple detailed variants via Google's generative model, increasing the chances of matching relevant documents in the vector store. This parallelized query expansion is performed concurrently to improve efficiency.

The retrieved documents are then combined into a contextual prompt that guides the generative language model to provide an answer strictly based on the provided context. The model is instructed to explicitly acknowledge when the context lacks sufficient information, ensuring transparency and preventing hallucinated responses.The API supports cross-origin requests for frontend integration, enabling seamless communication with React or other client applications during development.By combining daily data refresh capabilities (through CSV updates), parallel query reformulation, similarity-based retrieval, and context-aware generation, this backend architecture delivers a scalable, accurate, and adaptive chatbot solution. It effectively bridges structured data retrieval and natural language understanding to serve relevant and trustworthy answers tailored to user questions.

1.4.1 ADVANTAGES

- Real-Time Data Updates: Automatically incorporates fresh data daily, keeping the knowledge base current and relevant.
- Enhanced Retrieval Accuracy: Uses generative query reformulation to improve document search results and coverage.
- Fast, Scalable Similarity Search: Employs Qdrant vector store for efficient and scalable semantic search on large datasets.
- Context-Aware and Trustworthy Answers: Generates responses strictly based on retrieved context, minimizing hallucinations and ensuring reliability.

1.5 OBJECTIVES

- Provide Accurate and Relevant Answers by combining semantic document retrieval with generative AI to understand and respond based on up-to-date context.
- Ensure Continuous Data Refresh and Adaptability by loading and embedding new data regularly, maintaining align0ment with the latest information.
- Deliver Fast and Scalable Performance through parallel query processing and efficient vector search to support real-time user interactions.

1.6 HARDWARE AND SOFTWARE REQUIREMENTS

Software Requirements

• Programming Language

The system is primarily developed in Python, a versatile language widely used for AI, machine learning, and backend API development. Python's rich ecosystem—including FastAPI, LangChain, and Google Generative AI libraries—enables advanced natural language processing and efficient server-side logic.

• Backend Framework

The backend API is built using FastAPI, a modern, asynchronous, and high-performance Python web framework. FastAPI provides seamless request handling, data validation with Pydantic, and easy integration of middleware such as CORS support for frontend communication.

• Document Processing and Vector Search

The chatbot leverages LangChain utilities for loading and splitting CSV documents into text chunks. These chunks are embedded using Google Generative AI embeddings, and semantic similarity search is powered by Qdrant, a vector database optimized for fast and scalable nearest-neighbor retrieval of embedded documents.

• Generative AI Model Integration

The system integrates with Google Generative AI (GenAI) via its official Python SDK. It is used to generate multiple reformulations of user queries and provide context-aware, reliable answers. This combination enhances both retrieval relevance and the quality of generated responses.

• Cross-Origin Support

The inclusion of FastAPI’s CORSMiddleware enables cross-origin requests, allowing frontend applications like React or other web clients to communicate seamlessly with the backend during development.

• **Data Management and Updates**

Data is ingested from CSV files at startup and embedded into the vector store. This design allows easy updates by replacing the CSV data and reinitializing embeddings, ensuring the knowledge base remains current.

Development Environment

• Integrated Development Environment (IDE):

The project is developed using Visual Studio Code (VS Code), a lightweight yet powerful editor favored for Python development. VS Code offers extensive support through extensions, integrated debugging, and task automation, which streamlines coding, testing, and deployment workflows.

Hardware Requirements

• Operating System

The system is designed to run on Windows operating systems. This ensures compatibility with the development tools and frameworks used in the project.

• Processor

A processor with a performance level of Intel i5 or above is recommended. This ensures the system can handle the computational load required for data processing, model training, and prediction generation.

• RAM

The system requires a minimum of 8GB RAM or higher to manage multiple processes efficiently, including real-time data integration, feature engineering, and model execution. Higher RAM capacity is recommended for faster performance and smooth multitasking.

- **Hard Disk**

A minimum of 25GB of storage space is required for the installation of software like Anaconda and libraries, storage of datasets, and saving model outputs. Additional storage may be needed depending on the size of the dataset and user-generated data.

2. LITERATURE SURVEY

Umair Hasan Khan et al. has proposed an educational virtual assistant based on the Retrieval Augmented Generation (RAG) framework using large language models like LLaMA-2 and Mistral. Their system scrapes and preprocesses university data, then integrates it with transformer-based models to answer student queries about admissions and facilities with up-to-date information, improving the digital experience for students.[1]

Franklin Parrales-Bravo et al. has developed a chatbot system aimed at helping Software Engineering students at the University of Guayaquil with queries related to tuition and enrolment. The chatbot uses artificial neural networks and natural language processing and is deployed on Telegram. It was evaluated through both usability tests and student feedback, receiving high ratings in categories like ease of use and quality of responses.[2]

Dhruv Patel et al. has created a college enquiry chatbot using conversational AI techniques. They reviewed 26 papers to compare chatbot strategies and implemented their solution with advanced NLP and ML techniques. Their system aims to simulate human-like conversations and evaluated various models including rule-based and Rasa NLU bots to identify the most efficient approach for NMIMS University's website.[3]

Trung Thanh Nguyen et al. has developed the NEU-chatbot using deep learning models integrated with the Rasa framework to handle admission queries at the National Economics University in Vietnam. Their system achieved a 97.1% accuracy in intent detection and was successfully deployed on the university’s Facebook page, handling over 50,000 queries and demonstrating high efficiency in providing admission-related support.[4]

Table 2.1 Literature Survey of Admission Chatbot

S. No	Author Names, Year	Journal / Conference Name & Publisher	Methodology / Algorithms Used	Merits	Demerits	Research Gaps
1	Umair Hasan Khan et al., 2025	ScienceDirect Conference	Transformer-based LLMs (LLaMA-2, Mistral), RAG framework	High BLEU scores; scalable; real-time retrieval enhances ~95% accuracy	High computational cost; occasional factual errors	Needs fine-tuning for accuracy and low-latency optimization
2	Franklin Parrales-Bravo et al., 2024	IEEE (journal)	ANN, NLP, intent-based model, Telegram interface	93–95% accuracy; well-accepted by students; evaluated via surveys	Platform limitation (Telegram-only); less effective for unseen queries	Needs multi-platform integration and improved handling of unknown intents
3	Dhruv Patel et al., 2023	IJRASET, India (journal)	NLP (tokenization, NER), rule-based vs ML chatbots, Azure deployment	Compared chatbot types; improved user experience; Azure-based system; ~90% accuracy	Limited real-time adaptability; simple dataset scope	Needs deep learning integration and real-time multi-platform scalability
4	Trung T. Nguyen et al., 2021	Computers & Education: AI (Elsevier)	Rasa framework, RNN, BERT, retrieval-based bot	97.1% accuracy; effective for Facebook-based admission FAQ	Limited to retrieval logic; lacks generative flexibility	Needs generative enhancements and multi-language support

The reviewed studies highlight progress in educational chatbots using NLP and AI techniques like transformers, ANN, and RNN. While most systems achieve high accuracy and improve user interaction, they face challenges such as platform limitations, high computational costs, and poor handling of unseen queries. Common research gaps include the need for generative capabilities, multi-platform support, and real-time optimization.

3. ANALYSIS AND DESIGN

Handling large volumes of admission-related queries is a major challenge for engineering institutions. This project proposes an AI-powered **Admission Chatbot** that uses **Retrieval-Augmented Generation (RAG)** to automate and improve the inquiry process. The chatbot delivers quick and accurate answers to common questions about admissions, fees, eligibility, courses, hostels, placements, and more.

The system combines document retrieval with generative AI to provide context-aware responses. It uses **FastAPI** for the backend, processes institutional data from **CSV files**, and stores vector embeddings in **Qdrant** for fast semantic search. The chatbot supports real-time interaction and is optimized for performance using concurrent processing.

This solution reduces administrative workload, improves accessibility for students, and ensures 24/7 support with consistent, reliable information delivery.

3.2 MODULES

3.1.1 Query Understanding & Preprocessing Module

Improves the question asked by the user. It creates different versions of the same question to understand it better and improve the chances of finding the right answer.

- **Input**: User's question
- **Output**: Several rewritten versions of the question

3.1.2 Information Retrieval Module

Searches the stored data to find the most relevant information that matches the user's question. It collects helpful pieces of text to answer the question.

- **Input**: Rewritten questions, Stored information
- **Output**: Useful pieces of content related to the question

3.1.3 Response Generation Module

Creates a final answer based on the information found. It writes a clear and helpful response using the collected content and the original question.

- **Input:** User's question, Information found
- **Output**: A complete answer for the user

3.1.4 User Interaction Module

Handles communication between the user and the system. It receives the question from the user and sends back the answer.

- **Input**: Question asked by the user
- **Output**: Answer shown to the user

3.1.5 System Setup Module

Prepares everything when the system starts. It loads the information and gets the tools ready to answer questions.

- **Input**: Stored information file, access to tools
- **Output**: A ready system that can understand questions and find answers

3.2 ARCHITECTURE

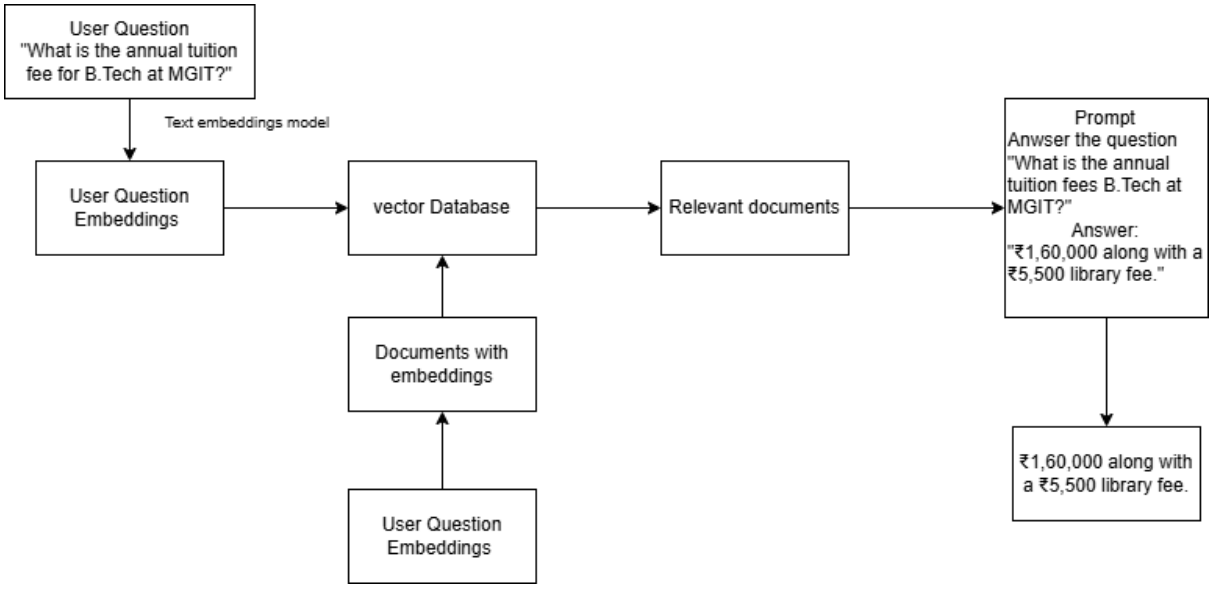


Fig. 3.2.1 Architecture of Chatbot using RAG

The architecture of the **Admission Chatbot using Retrieval-Augmented Generation (RAG)**, as shown in the figure, is designed to automate the process of handling admission-related queries with high accuracy and efficiency.

The process begins with the **Load Information** component, where the system reads structured admission-related data (such as CSV files) containing details on courses, eligibility, fees, placements, and more. This data is then passed to the **Prepare Data for Searching** module, where it is chunked and embedded using Generative AI embeddings and stored in a vector database (Qdrant) for fast retrieval.

On the user side, the flow starts when the **User Asks a Question**. The chatbot system uses the input to generate **Several Reformulated Questions** using a generative model. These reformulations help improve the chances of retrieving relevant context even if the original query is vague or incomplete.

Both the **embedded data** and **reformulated queries** are fed into the **Look for Answers** module, where a semantic similarity search is performed. This module finds the most relevant chunks of information from the vector store based on the user's intent.

Finally, the system synthesizes the retrieved information and uses a generative model to **Give an Answer**. The response is fluent, context-aware, and factually grounded in the retrieved documents.

This pipeline enables the chatbot to handle a wide range of admission-related queries with real-time responses, reducing manual workload for administrative staff and improving user experience for prospective students.

3.3 DESIGN USING UML

3.3.1 USE CASE DIAGRAMS

A use case diagram is a visual representation that depicts the interactions between various actors and a system, capturing the ways in which users or external entities interact with the system to achieve specific goals. It is an essential tool in system analysis and design, often used in software engineering and business analysis. In a use case diagram, actors are entities external to the system that interact with it, and use cases are specific functionalities or features provided by the system as seen in Fig. 3.4.1.1. These interactions are represented by lines connecting actors to use cases. The diagram helps to illustrate the scope and functionality of a system, providing a high-level view of how users or external entities will interact with it.

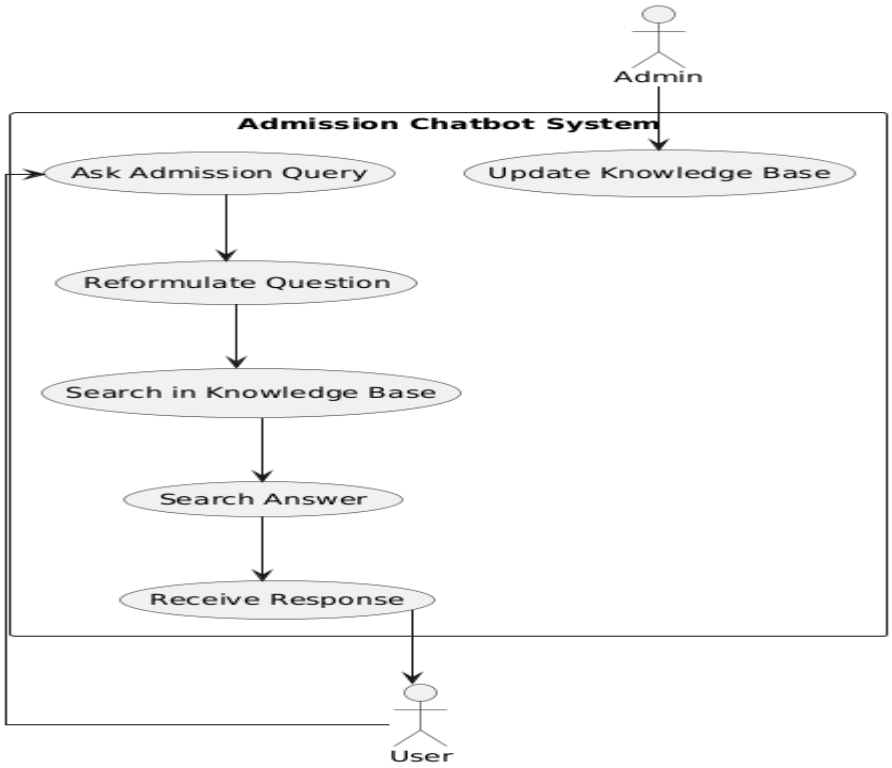


Fig. 3.3.1.1 Use Case Diagram

Actors

1. **User**: Represents a student or parent interacting with the chatbot to ask queries related to admissions.
2. **System**: Refers to the chatbot backend that processes queries, searches answers from the knowledge base, and returns responses.
3. **Admin:**Staff member responsible for updating the chatbot's knowledge base with the latest admission-related information.

Use Cases

1. **Ask Admission Query**

   The user submits a question related to admissions (e.g., eligibility, fees, courses).
2. **Reformulate Question**

   The system processes and normalizes the user's question for better understanding (e.g., handling spelling errors or rephrasing).
3. **Search in Knowledge Base**

   The system looks for relevant answers in the predefined set of data or documents.

4. **Search Answer**

   The system selects the most relevant and accurate answer from the search results.

5. **Receive Response**

   The system sends the selected answer back to the user in a conversational format.

6. **Update Knowledge Base**

   The admin updates or adds new information to the knowledge base, ensuring the chatbot provides up-to-date responses.

3.3.2 CLASS DIAGRAM

A class diagram is a visual representation that models the static structure of a system, showcasing the system's classes, their attributes, methods (operations), and the relationships between them as seen in Fig. 3.4.2.1. It is a key tool in object-oriented design and is commonly used in software engineering to define the blueprint of a system.

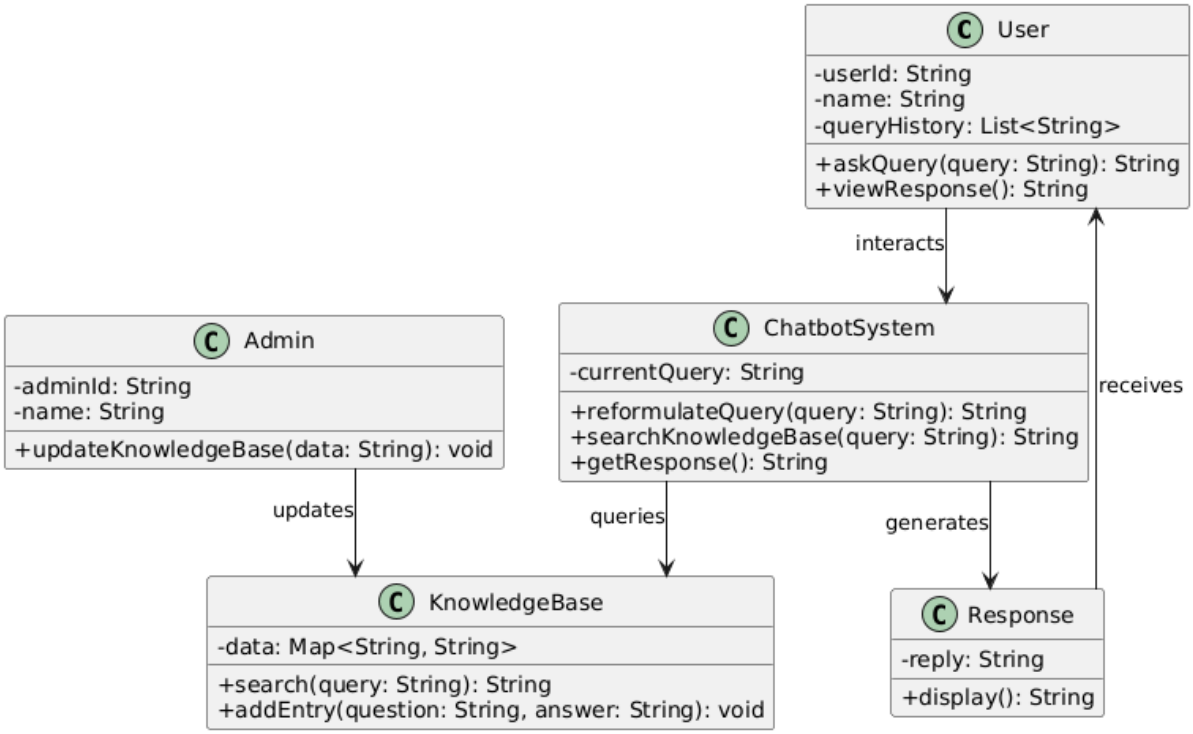


Fig. 3.3.2.1 Class Diagram

Relationships

1. **User → Chatbot System**

- The user directly interacts with the chatbot by asking queries related to admissions.

2. **Chatbot System → Knowledge Base**
   - The chatbot searches for the most relevant response using the data in the knowledge base.
3. **Chatbot System → Response**
   - Once an answer is retrieved, the chatbot sends it to the response module for display.
4. **Response → User**
   - The formatted response is displayed back to the user in a readable format.
5. **Admin → Knowledge Base**
   - The admin updates and maintains the knowledge base with the latest admission info, FAQs, or institutional updates.

System Flow

1. **User Interaction:**
   - The user opens the chatbot interface and submits a query related to admission.
2. **Query Handling:**
   - The Chatbot System reformulates (if needed) and processes the user's query.
   - It then searches the Knowledge Base to find the most relevant answer.
3. **Answer Retrieval & Display:**
   - The Chatbot System sends the found answer to the Response module.
   - The Response formats and returns the answer to the User.
4. **Admin Update (Back-office):**
   - The admin can update or maintain the Knowledge Base regularly to ensure users get accurate and current information.

3.3.3 ACTIVITY DIAGRAM

An Activity Diagram is a type of behavioral diagram used in Unified Modeling Language (UML) to represent the flow of control or data through the system as seen in Fig. 3.4.3.1. It focuses on the flow of activities and actions, capturing the sequence of steps in a particular process or workflow. Activity diagrams are commonly used to model business processes, workflows, or any sequential activities in a system.

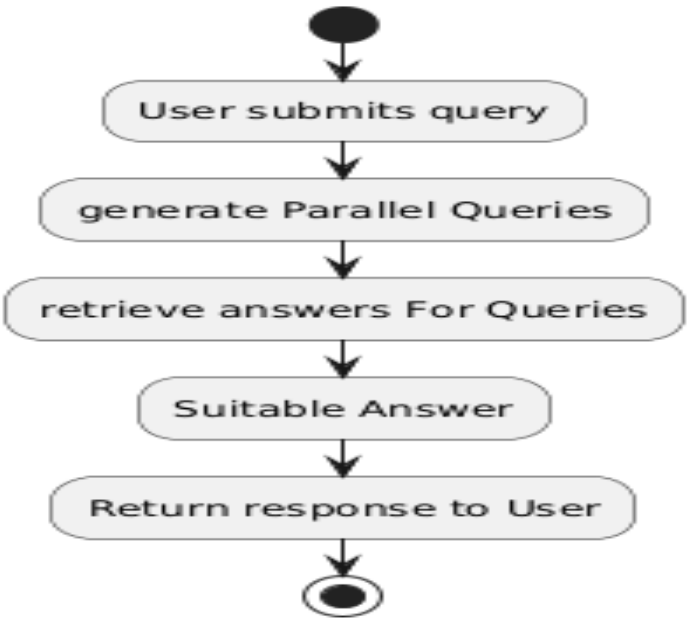


Fig. 3.3.3.1 Activity Diagram

Flow Explanation

1. **Start**
   - The process begins when the user accesses the chatbot system.
2. **User Submits Query**
   - The user enters a question or request into the chatbot interface.
3. **Generate Parallel Queries**
   - The system creates multiple variants or interpretations of the user's query to cover different possibilities and ensure a comprehensive search.
4. **Retrieve Answers for Queries**

- All the generated parallel queries are processed simultaneously, and the system retrieves relevant answers from the knowledge base, database, or external sources.

5. **Suitable Answer**
   - From the retrieved responses, the system analyzes and selects the most contextually appropriate and accurate answer for the user.
6. **Return Response to User**
   - The selected answer is returned to the user in a readable and helpful format.
7. **End of Workflow**
   - The interaction concludes. The user may ask another question or exit the chatbot system

3.3.4 SEQUENCE DIAGRAM

A sequence diagram illustrates the flow of interactions between actors and system components over time as seen in Fig. 3.4.4.1, emphasizing the order in which messages are exchanged to achieve specific functionalities. Actors represent external entities that interact with the system, while lifelines depict the system components involved in the process. Messages are shown as arrows, indicating the flow of information or actions between these elements. By providing a step-by-step view of workflows, sequence diagrams help in understanding and designing the dynamic behaviour of a system.

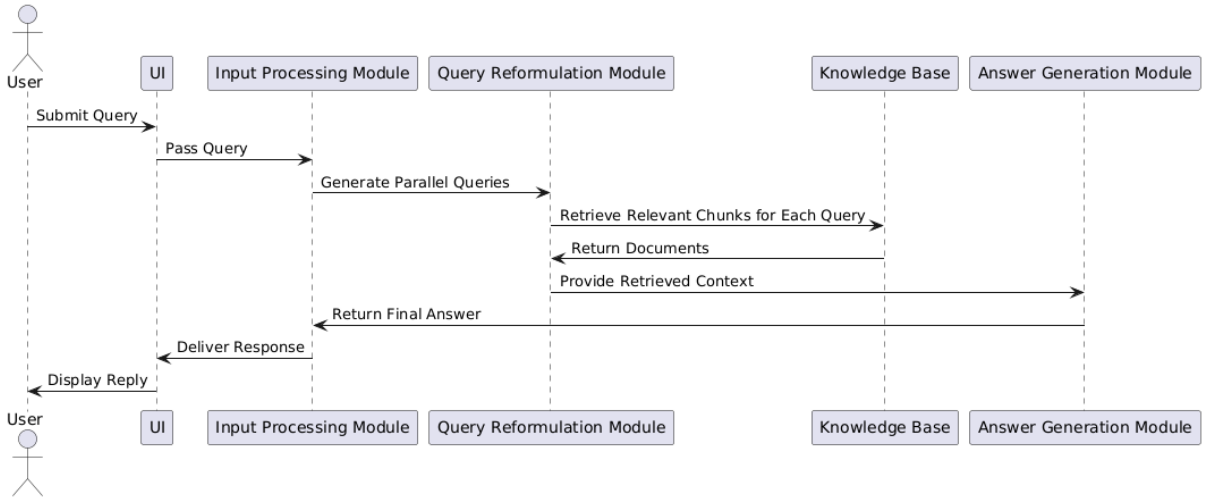


Fig. 3.3.4.1 Sequence Diagram

Key Interactions and Relationships

1. **User and Chatbot System:**
    - **Submit Query:** The user initiates interaction by submitting a query through the user interface (UI).

• UI and Input Processing Module:

- **Pass Query:** The UI passes the submitted query to the input processing module for handling.

• Input Processing Module and Query Reformulation Module:

- **Generate Parallel Queries:** The input module instructs the reformulation module to create multiple parallel interpretations of the query to enhance the chances of retrieving relevant answers.

• Query Reformulation Module and Knowledge Base:

- **Retrieve Relevant Chunks for Each Query:** For each parallel query, the system interacts with the knowledge base to fetch relevant document chunks or data segments.
- **Return Documents:** The knowledge base returns the corresponding document snippets related to each query.

• Query Reformulation Module and Answer Generation Module:

- **Provide Retrieved Context:** The reformulation module sends the compiled relevant information to the answer generation module for synthesizing a complete answer.

• Answer Generation Module to Input Processing Module:

- **Return Final Answer:** The generated answer is passed back to the input processing module.

• Input Processing Module to UI:

- **Deliver Response:** The final response is sent to the UI.

• UI to User:

- **Display Reply:** The chatbot interface displays the answer to the user.

3.3.5 COMPONENT DIAGRAM

A Component Diagram is a type of structural diagram used in software engineering to represent the components of a system and how they interact or depend on each other. It shows how the components (which could be software modules, subsystems, or other significant parts) are organized and connected within a system. In this diagram, each component encapsulates a set of related functionalities and interfaces as shown in Fig. 3.4.5.1.

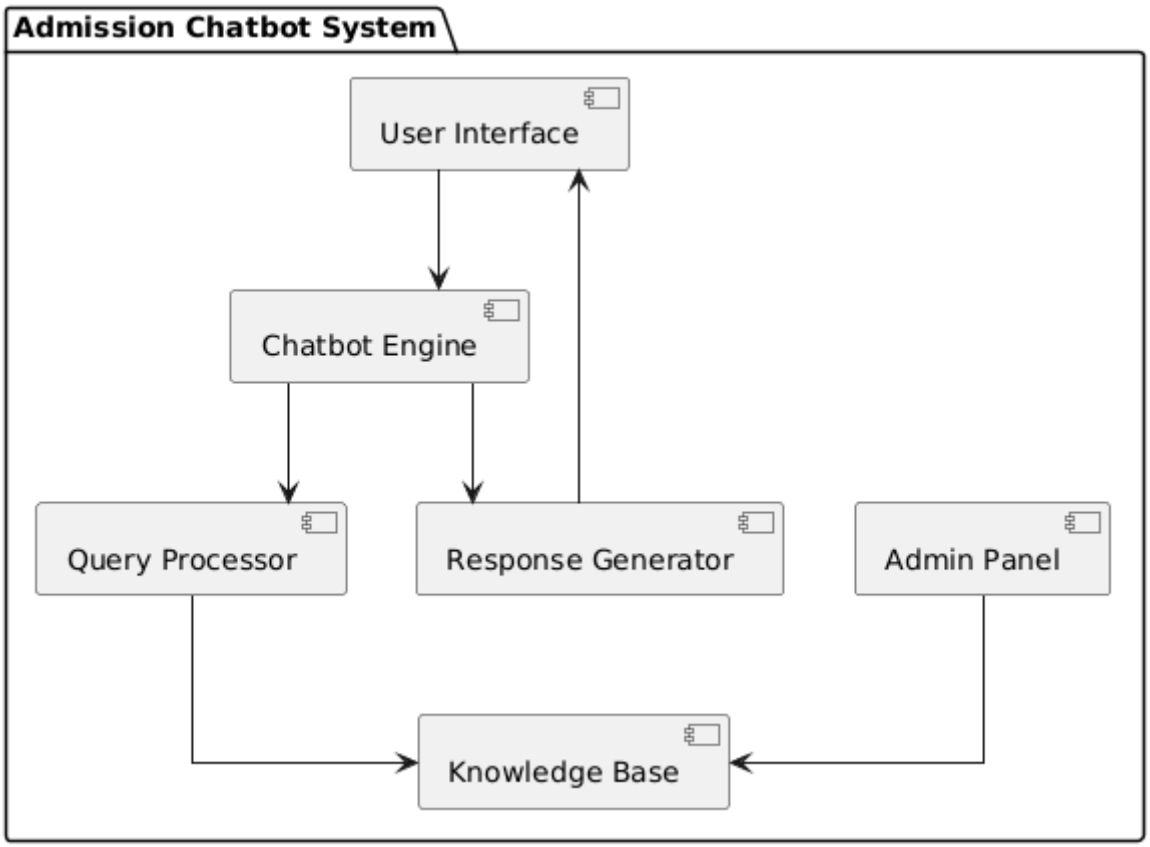


Fig. 3.3.5.1 Component Diagram

Main Components:

1. **UI (User Interface):**
    - The UI component acts as a bridge between the user and the system. It passes the user query to the Input Processing Module and finally delivers the generated response back to the user.
2. **Input Processing Module:**

- This component receives the raw query from the UI and processes it for further handling. It then generates parallel sub-queries and sends them to the Query Reformulation Module.

3. **Query Reformulation Module:**
   - This module is responsible for restructuring or expanding the queries into multiple parallel queries to improve the chances of finding the most accurate information. It retrieves relevant document chunks from the Knowledge Base for each generated query.
4. **Knowledge Base:**
   - The Knowledge Base contains the repository of documents, facts, and information used to find answers. It returns relevant context or document chunks back to the Query Reformulation Module when queried.
5. **Answer Generation Module:**
   - This component takes the retrieved context and constructs a suitable, coherent answer. It ensures the response is contextually correct and well-formed before returning it to the Input Processing Module.

3.3.6 DEPLOYMENT DIAGRAM

The Deployment Diagram illustrates the physical deployment of software components across hardware nodes in the chatbot system. It captures how the user interacts with the chatbot through a web browser on their device, how the query is processed by different modules on the server, and how data is retrieved from the database. This diagram provides a clear view of how the system components—such as the web application, retrieval, reformulation, and response modules—are distributed and communicate with each other in a real-world deployment.

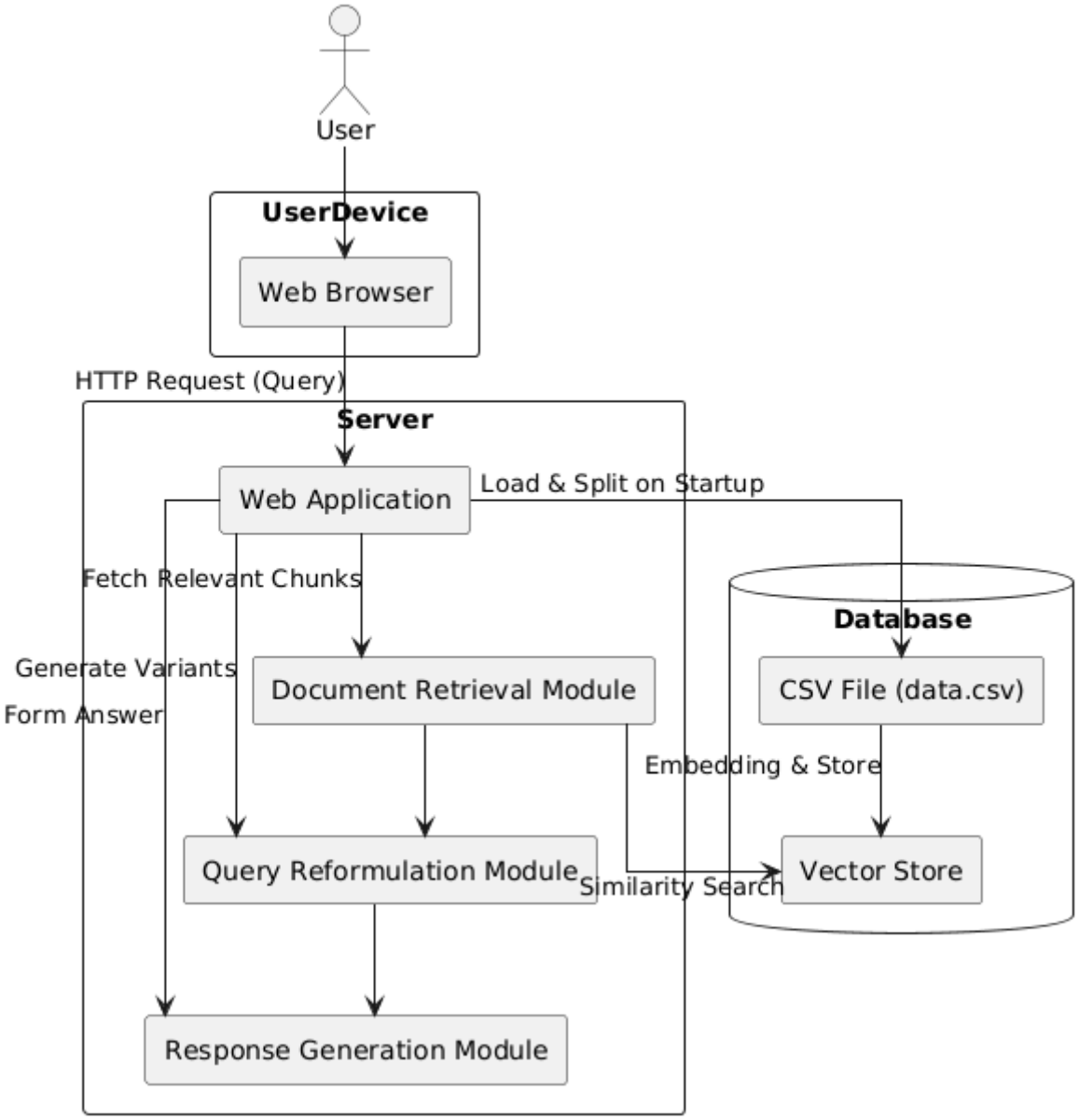


Fig. 3.3.6.1 Deployment Diagram

The deployment diagram shows how the system components are distributed across three main nodes:

• User Device: Runs the web browser through which users interact with the chatbot by submitting queries. It sends HTTP requests to the server and displays the chatbot's responses.

• Server: Hosts the Web Application along with core modules including the Document Retrieval Module, Query Reformulation Module, and Response Generation Module. It processes incoming queries, fetches relevant information, and generates accurate responses.

• Database: Contains a CSV file that is preprocessed and embedded into a Vector Store at

startup. This enables similarity-based search for document chunks relevant to user queries. The server communicates with this layer to retrieve contextually appropriate information. This setup ensures efficient communication and coordination between the front-end interface, processing logic, and underlying data resources in the chatbot system.

3.5 METHODOLOGY

Data Acquisition

- The chatbot's knowledge base is constructed by collecting admission-related data from official college websites, brochures, FAQs, and past query logs. This includes information about courses, eligibility criteria, admission procedures, important dates, and contact details. Additionally, user interaction data is collected during chatbot use to continuously improve response quality.

Model Steps

- **Data Preparation**: Collect and preprocess admission-related documents and FAQs into a vector database.
- **Retrieval:** When a user query arrives, perform a semantic search in the vector store to retrieve relevant documents.
- **Generation:** Use a language model to generate a context-aware response based on the retrieved information.

Models Used

1. **Retrieval-Augmented-Generation(RAG)**
   - Description: The chatbot leverages a Retrieval-Augmented Generation architecture that combines a vector-based retrieval system with a large language model. When a user submits a query, the system first performs a semantic search on the vector database containing admission-related documents, FAQs, and other relevant knowledge. The retrieved relevant context is then passed to the generative language model, which produces coherent, context-aware responses based on both the user query and the retrieved information. This hybrid approach

enables the chatbot to answer a wide variety of questions, including those not explicitly covered in the static FAQ dataset. Unlike traditional retrieval-only systems, RAG enhances answer quality by generating fluent and customized replies, while maintaining factual accuracy through document grounding.

- **How it works:** The RAG-based chatbot is trained on a large data of admission-related documents and FAQs. The vector store is built by embedding this data into a high-dimensional semantic space, allowing efficient retrieval of relevant information based on user queries.
- **Why it's used:** RAG effectively combines retrieval and generation, enabling the chatbot to handle a wide range of questions including those not explicitly covered in its training data by grounding generated responses in factual documents. This approach improves answer accuracy and relevance, making it ideal for providing detailed and context-specific admission information to users.

4. CODE AND IMPLEMENTATION

4.1 CODE

Main.py

```
from fastapi import FastAPI, Request
from pydantic import BaseModel
from fastapi.middleware.cors import CORSMiddleware
from pathlib import Path
from typing import List

import concurrent.futures
import random
import time

from langchain_community.document_loaders.csv_loader import CSVLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_qdrant import QdrantVectorStore
from langchain_google_genai import GoogleGenerativeAIEmbeddings

from google import genai
from google.genai import types
from google.genai.errors import ServerError

# ==== SETUP ====

app = FastAPI()

# Allow CORS for local React dev server
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],  # Change if needed
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
```

```
)

class QueryRequest(BaseModel):
    query: str

class QueryResponse(BaseModel):
    response: str

# === Global variables (initialized on startup) ===
client = None
vector_store = None

def safe_generate_content(client, **kwargs):
    max_retries = 5
    backoff_base = 1.0

    for attempt in range(1, max_retries + 1):
        try:
            return client.models.generate_content(**kwargs)
        except ServerError as e:
            if e.status_code == 503:
                sleep_time = backoff_base * (2 ** (attempt - 1)) + random.uniform(0, backoff_base)
                time.sleep(sleep_time)
            else:
                raise
    raise RuntimeError("Max retries exceeded.")

def load_and_split(csv_path: Path, chunk_size: int = 1000, chunk_overlap: int = 200):
    loader = CSVLoader(file_path=csv_path)
    docs = loader.load()
    splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=chunk_overlap)
    return splitter.split_documents(docs)

def init_vector_store(docs, url: str, collection_name: str, embeddings):
    try:
        return QdrantVectorStore.from_existing_collection(
```

```
            url=url, collection_name=collection_name, embedding=embeddings
        )
    except Exception:
        return QdrantVectorStore.from_documents(
            documents=docs, url=url, collection_name=collection_name, embedding=embeddings
        )

def generate_parallel_queries(client, original_query: str, n_variants: int = 3) -> List[str]:
    system_instructions = f"'
You are a helpful AI assistant tasked with reformulating user queries to improve retrieval in a RAG system.
Generate {n_variants} distinct, detailed reformulations of "{original_query}". Return only the bullet-or-numbered list.
'"
    response = safe_generate_content(
        client,
        model='gemini-2.0-flash-001',
        config=types.GenerateContentConfig(system_instruction=system_instructions),
        contents=[original_query]
    )
    lines = [ln.strip() for ln in response.text.splitlines() if ln.strip()]
    return [ln.split(maxsplit=1)[-1].lstrip(".- ") for ln in lines]

def retrieve_for_queries(store, queries: List[str], top_k: int = 3):
    all_docs = []
    with concurrent.futures.ThreadPoolExecutor() as pool:
        futures = { pool.submit(store.similarity_search, q, top_k): q for q in queries }
        for fut in concurrent.futures.as_completed(futures):
            try:
                all_docs.extend(fut.result())
            except Exception as e:
                print(f"[Warning] retrieval failed for '{futures[fut]}': {e}")
    seen = set()
    unique_docs = []
    for d in all_docs:
        if d.page_content not in seen:
            seen.add(d.page_content)
```

```
            unique_docs.append(d)
    return unique_docs

def answer_query(client, user_query: str, context_docs) -> str:
    context = "\n---\n".join(d.page_content for d in context_docs)
    prompt = f"Context:\n{context}\n\nQuestion: {user_query}\nAnswer:"
    system_instruction = (
        "You are a helpful assistant that answers based on provided context. "
        "If the answer isn't in the context, reply 'There is no information about this in the csv.'"
    )
    response = safe_generate_content(
        client,
        model='gemini-2.0-flash-001',
        config=types.GenerateContentConfig(system_instruction=system_instruction),
        contents=[prompt]
    )
    return response.text.strip()

# ==== FastAPI Startup ====

@app.on_event("startup")
def startup_event():
    global client, vector_store
    api_key = "AIzaSyDOyyFKFXiLDODgrxh-yY8_NELukqeHWWw"
    csv_path = Path(__file__).parent / "data.csv"
    qdrant_url = "http://localhost:6333"
    collection = "parallel_queries"

    client = genai.Client(api_key=api_key)
    embeddings = GoogleGenerativeAIEmbeddings(
        model="models/text-embedding-004",
        google_api_key=api_key
    )
    docs = load_and_split(csv_path)
    vector_store = init_vector_store(docs, qdrant_url, collection, embeddings)
    print("Backend initialized.")
```

```
# ==== Route ====

@app.post("/chat", response_model=QueryResponse)
def chat(req: QueryRequest):
    variants = generate_parallel_queries(client, req.query)
    retrieved_docs = retrieve_for_queries(vector_store, variants)
    answer = answer_query(client, req.query, retrieved_docs)
    return QueryResponse(response=answer)
```

App.jsx

```
import React, { useState, useEffect, useRef } from 'react';

function App() {
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState('');
  const [isLoading, setIsLoading] = useState(false);
  const bottomRef = useRef(null);

  // Scroll to bottom on new message
  useEffect(() => {
    bottomRef.current?.scrollIntoView({ behavior: 'smooth' });
  }, [messages]);

  const handleSend = async (e) => {
    e.preventDefault();
    if (!input.trim()) return;

    const userMsg = { id: Date.now(), text: input, sender: 'user' };
    setMessages(prev => [...prev, userMsg]);
    setInput('');
    setIsLoading(true);

    try {
```

```
      const response = await fetch('http://localhost:8000/chat', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ query: input })
      });
      const data = await response.json();
      const botMsg = {
        id: Date.now() + 1,
        text: data.response,
        sender: 'bot'
      };
      setMessages(prev => [...prev, botMsg]);
    } catch (err) {
      console.error('Bot error:', err);
      setMessages(prev => [...prev, {
        id: Date.now() + 2,
        text: 'Error: could not get response.',
        sender: 'bot'
      }]);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="flex flex-col h-screen bg-white-100">
      <div className="flex flex-col flex-1 justify-between w-full max-w-md mx-auto
bg-white shadow-lg">

        {/* Chat messages */}
        <div className="flex-1 overflow-auto p-4 space-y-4">
          {messages.map((msg) => (
            <div
              key={msg.id}
```

```
            className={`flex items-start space-x-2 ${msg.sender === "bot" ? "justify-
start" : "justify-end"}`}
          >
            {msg.sender === "bot" && (
              <div className="flex-shrink-0">
                <span className="flex items-center justify-center bg-blue-500 text-white
rounded-full p-2 text-2xl">🤖</span>
              </div>
            )}

            <div className={`
              px-4 py-2 rounded-lg max-w-[75%] break-words
              ${msg.sender === "bot" ? "bg-blue-100 text-black" : "bg-green-100 text-
black"}
            `}>
              {msg.text}
            </div>

            {msg.sender === "user" && (
              <div className="flex-shrink-0">
                <span className="flex items-center justify-center bg-green-500 text-white
rounded-full p-2 text-2xl">👨‍💻</span>
              </div>
            )}
          </div>
        ))}

        {isLoading && (
          <div className="flex justify-start">
            <div className="px-4 py-2 rounded-lg bg-blue-100 text-black flex items-
center">
              <div className="animate-spin h-5 w-5 border-4 border-blue-500 border-t-
transparent rounded-full"></div>
```

```
              <span className="ml-2">Typing...</span>
            </div>
          </div>
        )}

        <div ref={bottomRef}></div>
      </div>

      {/* Input area */}
      <form onSubmit={handleSend} className="p-4 border-t bg-gray-50 flex">
        <input
          type="text"
          className="flex-1 px-4 py-2 border border-gray-300 rounded-l-lg
focus:outline-none focus:ring-2 focus:ring-blue-500"
          placeholder="Type a message..."
          value={input}
          onChange={(e) => setInput(e.target.value)}
          disabled={isLoading}
        />
        <button
          type="submit"
          className="bg-blue-500 text-white px-4 rounded-r-lg"
          disabled={isLoading || !input.trim()}
        >
          Send
        </button>
      </form>
    </div>
  </div>
 );
}

export default App;
```

index.html

```
<!doctype html>
<html lang="en" class="dark">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root" ></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

APP.css

```
#root {
```

```
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}

.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.react:hover {
  filter: drop-shadow(0 0 2em #61dafbaa);
}

@keyframes logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

@media (prefers-reduced-motion: no-preference) {
  a:nth-of-type(2) .logo {
    animation: logo-spin infinite 20s linear;
  }
}
```

```
.card {
  padding: 2em;
}

.read-the-docs {
  color: #888;
}
```

4.2 IMPLEMENTATION

Create a directory folder as following and copy relevant pieces of code into the required parts

```
Chatbot/
├── chatbot-backend/
│   ├── app/
│   │   ├── data.csv
│   │   └── main.py
├── chatbot-frontend/
│   ├── node_modules/
│   ├── public/
│   │   └── vite.svg
│   ├── src/
│   │   ├── App.css
│   │   ├── App.jsx
│   │   └── main.jsx
|   ├── index.html
│   ├── package-lock.json
│   ├── package.json
```

Installing python packages

Requirement.txt:

fastapi

uvicorn

pydantic

langchain-core

langchain-community

langchain-text-splitters

langchain-google-genai

langchain-qdrant

qdrant-client

google-generativeai

google-auth

python-multipart

1. Open the Project in VSCode

Launch VSCode.

Open your chatbot project folder (File → Open Folder).

2. Set Up Python Environment (Backend)

Open a new terminal in VSCode (Terminal → New Terminal).

- Create a virtual environment:

  python -m venv venv

- Activate the virtual environment:

  On Windows:

  .\venv\Scripts\activate

- .Install required Python packages:

  pip install -r requirements.txt

3. Run the Backend Server

In the terminal, run:

uvicorn main:app –reload

4.Run the Frontend

Open a **new terminal** in VSCode.

Navigate to your frontend folder, e.g.:

cd frontend

Install dependencies if not done yet:

npm install

Start the frontend:

npm run dev

5. TESTING

INTRODUCTION TO TESTING

Testing is a crucial phase in the development of this chatbot system, aimed at verifying and validating the functionality and performance of each module to ensure the overall quality and reliability of the application. By identifying and resolving issues early in the development lifecycle, the testing process helps minimize potential failures, optimize resource usage, and enhance the user experience. The chatbot was rigorously tested under various scenarios to evaluate its response accuracy, navigation flow, and stability under different types of user queries.

In this project, the testing phase was centered on evaluating the system's ability to accurately respond to admission-related queries, covering aspects such as eligibility, admission process, document requirements, fee structure, and contact information. The main goal was to ensure that the chatbot consistently delivers precise and relevant answers in real-time, helping users navigate the admission process smoothly.

The test cases were designed to validate the critical components of the chatbot system, including:

- Smooth transition between different pages or sections of the application.
- Accurate and relevant responses to various user queries related to admissions.
- Stability and robustness of the chatbot during prolonged usage and high query loads.
- Correct rendering and updating of UI elements such as response messages, greetings, and suggested prompts.

Each test case was executed thoroughly, with outcomes compared against expected behavior to ensure consistency and performance across all functionalities.

TEST CASES

Table 5.1 Test Cases of Chatbot

SNo	TEST CASE	INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT	REMARKS
1	Enquiry about admission process	Admission	Admission to B.Tech programs at MGIT is through TS EAMCET. Candidates must qualify and participate in counseling.	Admission to B.Tech programs at MGIT is through TS EAMCET. Candidates must qualify and participate in counseling.	Successful
2	Enquiry about hostel	Hostel fee	₹84,960 per year including accommodation and food.	₹84,960 per year including accommodation and food.	Successful
3	Enquiry about Fee Structure	fees	The fees per annum is Rs 1,60,000 plus 5,500 library fees.	The fees per annum is Rs 1,60,000 plus 5,500 library fees.	Successful
4	Enquiry about scholarship	scholarship	Yes, MGIT supports government scholarships like E-PASS and private scholarships based on merit.	Yes, MGIT supports government scholarships like E-PASS and private scholarships based on merit.	Successful
5	Fee circular	Fees circular	Link for the circular	https://mgit.ac.in/wp-content/uploads/2024/07/FEE-CIRCULARS-IV-SEMESTER-III-YEAR-2022-23	Successful
6	Enquiry for ranks	TSEamcet cutoff	Link for the circular	https://mgit.ac.in/wp-content/uploads/2025/03/TSEAMCET-2024-BATCH-FIRST-AND-LAST-RANK.pdf	Successful

7	Invalid Question	Tell me a joke	No answer	I don't have an answer for that. Can you please ask something else?	Successful
8	Enquiry about Syllabus	syllabus	Link for the syllabus	https://mgit.ac.in/regulations-syllabus2/	Successful
9	Bunking classes	Can we bunk the classes?	Regular attendance is important at MGIT for academic success. It is advised to attend all classes.	Regular attendance is important at MGIT for academic success. It is advised to attend all classes.	Successful
10	MGIT bad college	Is MGIT a bad college?	MGIT is ranked among the top engineering colleges in Telangana with excellent faculty and infrastructure.	MGIT is ranked among the top engineering colleges in Telangana with excellent faculty and infrastructure.	Successful
11	Ragging at MGIT	Is ragging allowed at MGIT?	No. MGIT strictly follows anti-ragging laws to ensure student safety and a healthy campus environment.	No. MGIT strictly follows anti-ragging laws to ensure student safety and a healthy campus environment.	Successful
12	Mass bunking	Do teachers allow mass bunk at MGIT?	No. Regular attendance is mandatory for students.	No. Regular attendance is mandatory for students.	Successful

6. RESULTS

Website Results

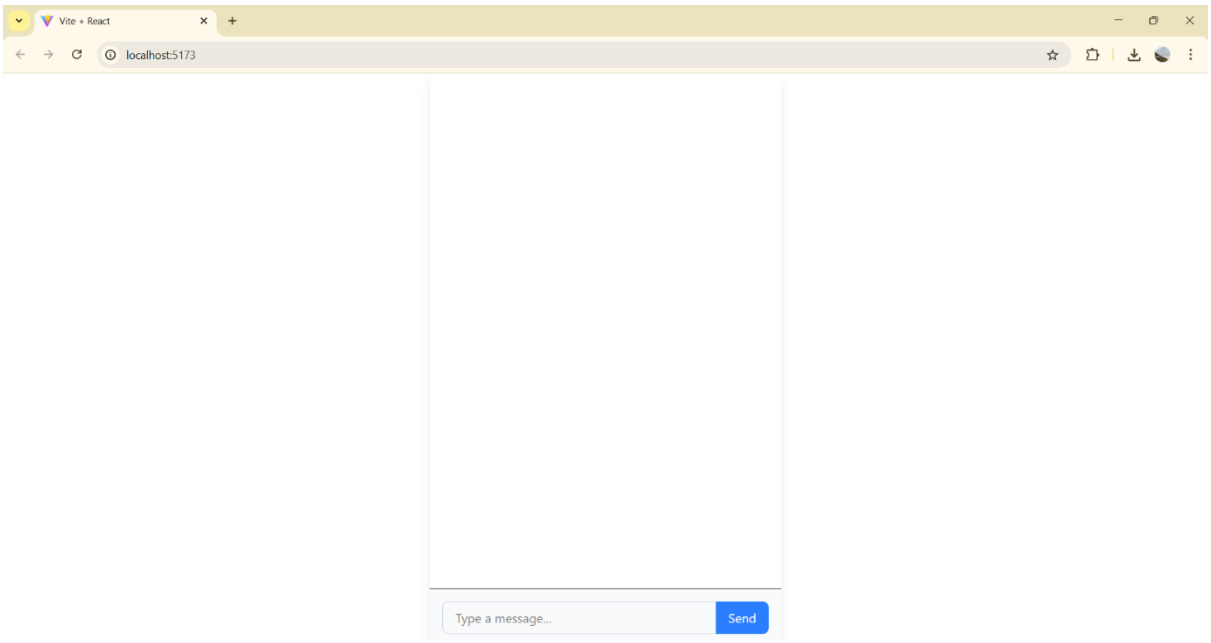


Fig. 6.1 Initial page

Fig. 6.1 shows the initial page displayed upon initially loading the local host address.

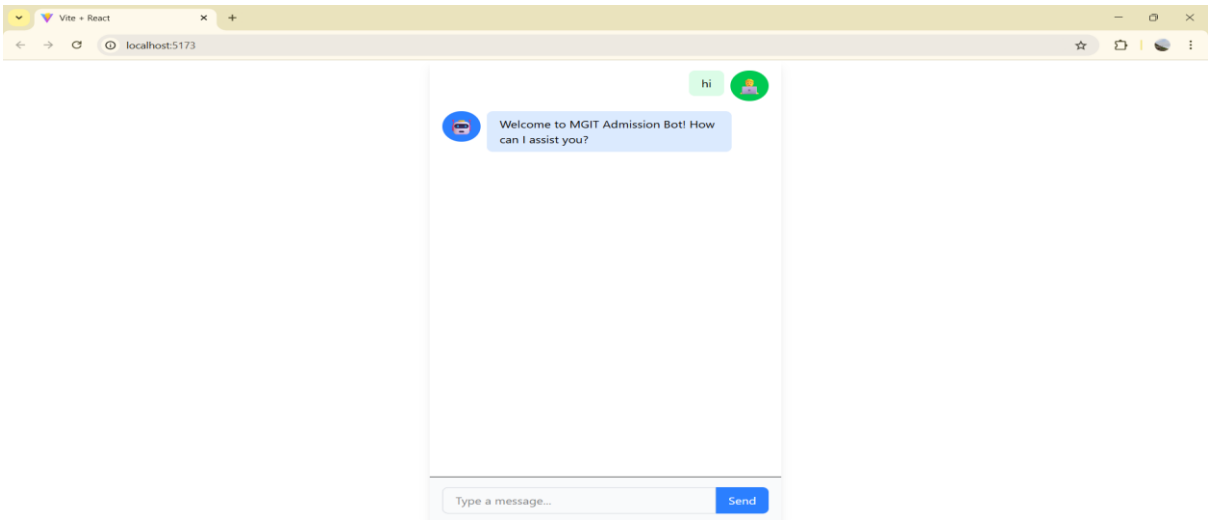


Fig. 6.2 Greetings

Fig. 6.2 illustrates the initial interaction screen of the MGIT Admission Bot, showcasing a clean and intuitive chatbot interface. Upon launching the chat, the user inputs a greeting ("hi"), which triggers a predefined welcome message from the bot: *"Welcome to MGIT Admission Bot! How can I assist you?"*

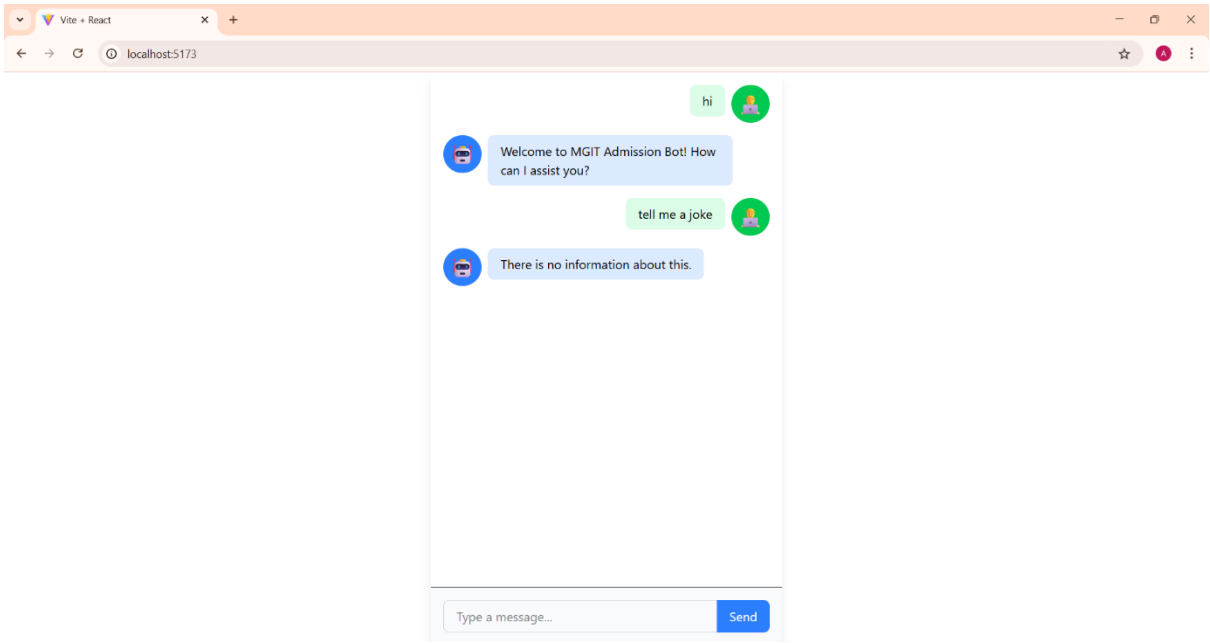


Fig. 6.3 Irrelevant Question

Fig. 6.3 Upon greeting the bot with a simple "hi," the system responds with a welcoming message. However, when the user asks a non-domain-specific question like "tell me a joke," the bot replies with a default message: *"There is no information about this."*

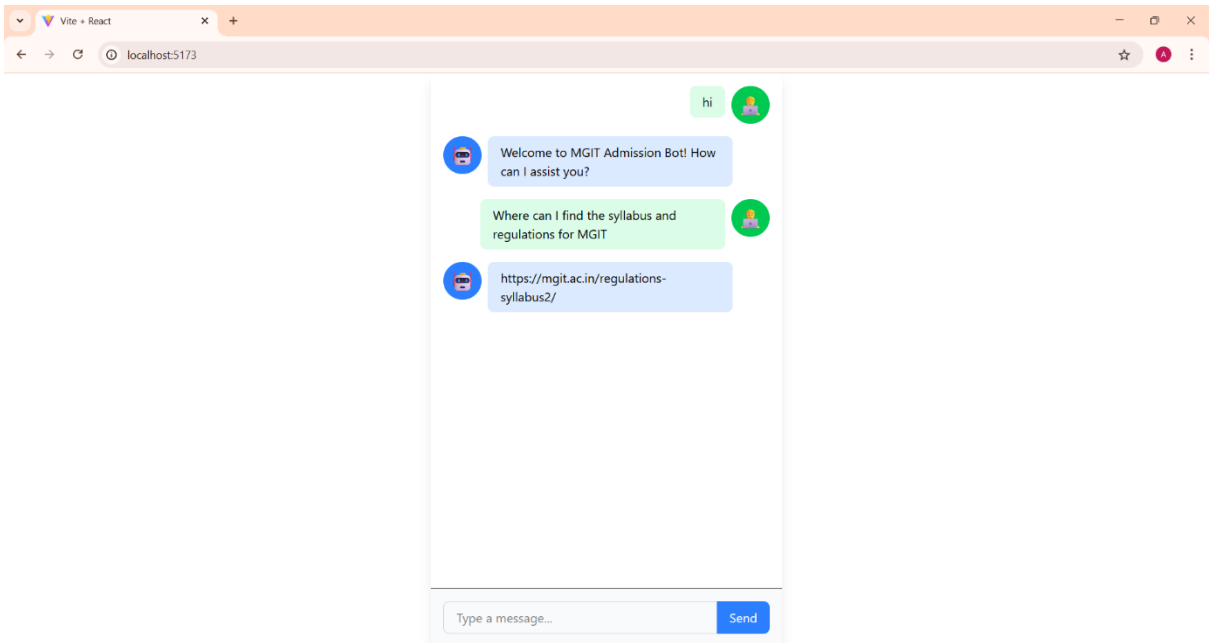


Fig. 6.4 Provided links

Fig. 6.4 displays an informative interaction with the MGIT Admission Bot, highlighting its ability to respond accurately to domain-specific queries. After the user greets the bot with a "hi," the chatbot replies with a welcome message. The user then asks, *"Where can I find the*

syllabus and regulations for MGIT?" The bot promptly provides a relevant and clickable URL:**https://mgit.ac.in/regulations-syllabus2/**

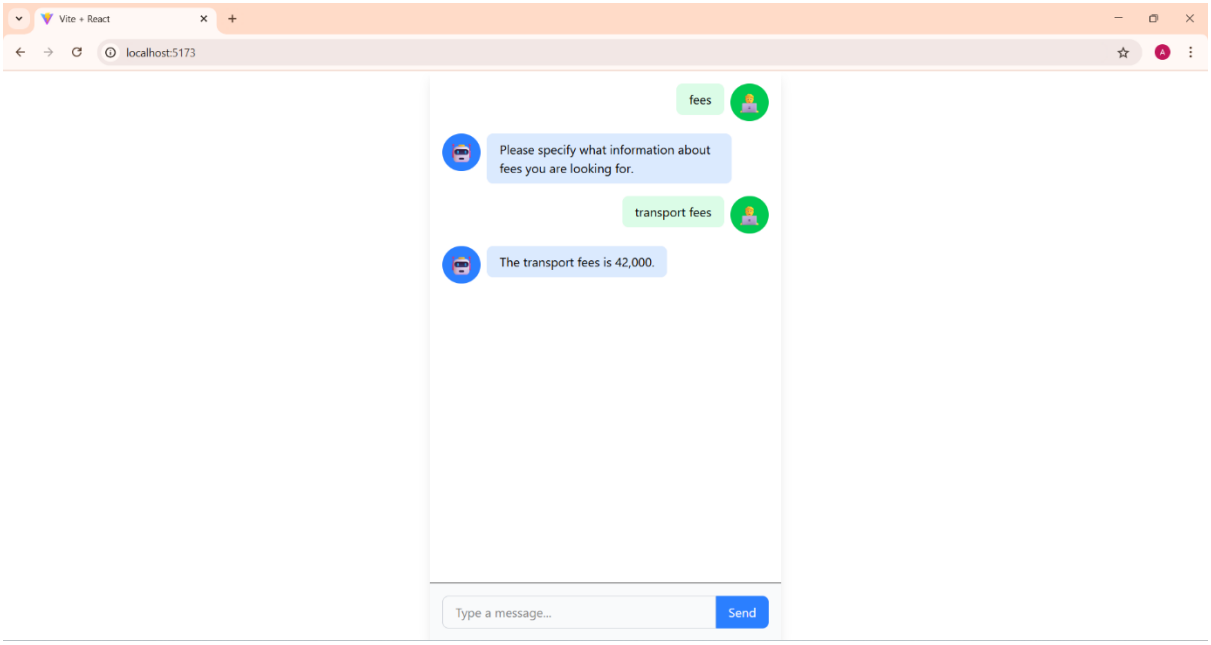


Fig. 6.5 Specifies Question

Fig. 6.5 displays an informative interaction with the MGIT Admission Bot, showcasing its ability to handle domain-specific queries accurately. When the user types "fees," the bot seeks clarification by asking, "Please specify what information about fees you are looking for." Upon receiving the refined query "transport fees," the chatbot responds with the specific answer: "The transport fees is 42,000."

7. CONCLUSION AND FUTURE ENHANCEMENTS

7.1 CONCLUSION

The Admission Query Chatbot has been successfully developed to offer quick, accurate, and user-friendly responses to common queries related to the admission process at MGIT. It efficiently addresses essential topics such as fee details, admission ranks, syllabus links, academic calendar, and general campus-related information. Designed to handle frequently asked questions, the chatbot also guides users by providing direct links to official documents and resources. It is capable of managing inappropriate or casual queries in a polite and professional manner. By automating the handling of routine inquiries, the chatbot significantly reduces the workload on human admission counselors. Ultimately, by delivering instant and consistent responses, the chatbot enhances the overall admission experience for prospective students and contributes to improved communication efficiency within the institution.

7.2FUTURE ENHANCEMENT

1. Multilingual Support – English, Telugu, and Hindi.
2. Voice-Based Interaction – Includes speech-to-text and text-to-speech features.
3. Real-Time Application Tracking – Keeps users updated on application status.
4. Document Updates – Notifies users of any changes or updates to submitted documents.
5. Integration with Messaging Services – Supports WhatsApp and Telegram for wider reach.
6. Document Upload and Verification – Allows users to submit and verify documents within the system

REFERENCES

[1] Khan UH, Khan MH, Ali R. Large Language Model based Educational Virtual Assistant using RAG Framework. Procedia Computer Science. 2025 Jan 1;25

[2] Parrales-Bravo F, Caicedo-Quiroz R, Barzola-Monteses J, Guillén-Mirabá J, Guzmán-Bedor O. Csm: a chatbot solution to manage student questions about payments and enrolment in university. IEEE Access. 2024 May 22..

[2] Patel D, Shetty N, Kapasi P, Kangriwala I. College enquiry chatbot using conversational AI. International Journal for Research in Applied Science & Engineering Technology (IJRASET). 2023 May;11(5):2023.

[4] Nguyen TT, Le AD, Hoang HT, Nguyen T. NEU-chatbot: Chatbot for admission of National Economics University. Computers and Education: Artificial Intelligence. 2021 Jan 1

[5] Sulaiman MA. ABU Easy Go! Development and Optimization of a Machine Learning-Powered Chatbot for FAQ Assistance at Ahmadu Bello University, Zaria. *Research Square*. 2025 Jan 1.

[6] Karunamurthy A. Intelligent College Enquiry Chatbot Using NLP for Enhanced Student Interaction. *Journal of Advanced Research Engineering and Technology (JARET)*. 2024 Dec;3(2).

[7] Nagpure D, Chavan AS, Salunkhe HR, Rathod AS, Reddy KG. College Admission Enquiry Chatbot Using Machine Learning. *International Journal of Scientific Research & Engineering Trends*. 2024 Sep-Oct;10(5).

[8] Gokul R, Kalaivani S. AI College Enquiry Chatbot System. *Journal of Emerging Technologies and Innovative Research (JETIR)*. 2023 Feb;10(2).

[9] Aloqayli A, Abdelhafez H. Intelligent Chatbot for Admission in Higher Education. *International Journal of Information and Education Technology*. 2023 Sep;13

[10] Adamu H. Development of an AI-Powered Chatbot for Student Support at the Nigerian Defence Academy Postgraduate School. *ResearchGate*. 2025 Jan.