A Project Report on

HEALTHCARE CHATBOT USING GPT-3

Built a chatbot fine-tuned on medical conversations for health-related queries

in partial fulfilment of the requirements for the award of the Degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

Submitted by

D. Rutwika Satya Sri	**21B81A1229**
Ch. Roopa	**21B81A1223**
B. Teja Sri	**21B81A1216**
K. Geethika	**21B81A1256**

Under the Esteemed Guidance of

Smt. N. Durga Prasanna, M.Tech

Assistant Professor, Department of IT

DEPARTMENT OF INFORMATION TECHNOLOGY

SIR CR REDDY COLLEGE OF ENGINEERING

Approved by AICTE & Accredited by NBA & NAAC

Affiliated to Jawaharlal Nehru Technological University, Kakinada

ELURU-5340007

2024-25

SIR C R REDDY COLLEGE OF ENGINEERING

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

This is to certify that this project report entitled "**Healthcare chatbot using GPT-3**" being submitted by

D. Rutwika Satya Sri	**21B81A1229**
Ch. Roopa	**21B81A1223**
B. Teja Sri	**21B81A1216**
K. Geethika	**21B81A1256**

in partial fulfilment for the award of the Degree of Bachelor of Technology in Information Technology to the JNTU KAKINADA is a record of bonafide work carried out under my guidance and supervision. The results embodied in this project report have not been submitted to any other University or Institute for the award of any Degree.

SMT. N. DURGA PRASANNA
PROJECT GUIDE
Department of IT
Sir C R R College of Engineering

DR. K. SATYANARAYANA
HEAD OF THE DEPARTMENT
Department of IT
Sir C R R College of Engineering

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We wish to express our sincere thanks to various personalities who were responsible for the successful completion of this project.

We thank our principal, **Dr. K. VENKATESWARA RAO**, for providing the necessary infrastructure required for our project.

We are grateful to **Dr. K. SATYANARAYANA,** Head of the Information Technology department, for providing the necessary facilities for completing the project in specified time.

We express our deep-felt gratitude to **Smt. N. DURGA PRASANNA**, as her valuable guidance and unstinting encouragement enabled us to accomplish our project successfully in time.

Our special thanks to librarian **Smt. D. LAKSHMI KUMARI**, and to the entire library staff Sir C.R.R College of Engineering, for providing the necessary library facilities.

We express our earnest thanks to faculty members and non-teaching staff of IT for extending their valuable support.

PROJECT MEMBERS

D. Rutwika Satya Sri (21B81A1229)

Ch. Roopa (21B81A1223)

B. Teja Sri (21B81A1216)

K. Geethika (21B81A1256)

DECLARATION

We here by declare that the Project entitled **HEALTHCARE CHATBOT USING GPT-3** submitted for the B. Tech Degree is our original work and the Project has not formed the basis for the award of any degree, associateship, fellowship or any other similar titles.

PROJECT MEMBERS

D. Rutwika Satya Sri (21B81A1229)

Ch. Roopa (21B81A1223)

B. Teja Sri (21B81A1216)

K. Geethika (21B81A1256)

ABSTRACT

With the rapid advancements in Artificial Intelligence (AI) and Natural Language Processing (NLP), the healthcare industry has witnessed revolutionary tools that enhance patient engagement and health support. This paper presents a Healthcare Chatbot using Generative AI, designed to assist users by providing medically relevant responses based on user queries. Unlike traditional symptom-checker platforms, this system leverages deep learning and vector-based search to understand and respond contextually using a pre-embedded semantic index built from medical literature.

A comprehensive literature review is carried out to assess the effectiveness of AI models in healthcare communication. The chatbot is designed to extract and preprocess healthcare data from trusted sources, convert it into embeddings using OpenAI's models, store them in Pinecone, and retrieve relevant content based on similarity with the user query. The system supports real-time user interaction through a web-based frontend built with Flask and deployed on AWS. The results demonstrate that the proposed system improves response relevance, enhances patient self-care, and minimizes the burden on healthcare professionals.

Keywords: Generative AI, Healthcare Chatbot, Semantic Search, Natural Language Processing, Pinecone, OpenAI

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The fast-paced developments in Artificial Intelligence (AI) have opened the doors for pioneering uses in the healthcare industry. One of these is using AI-based chatbots to increase accessibility of primary medical information to individuals in impoverished communities quickly and easily. This project involves developing a Healthcare Chatbot via Generative AI that can converse in natural language, understand health-related questions from users, and return correct, relevant answers drawn from valid medical papers.

Unlike traditional rule-based chatbots, this system uses state-of-the-art technologies such as Open AI's GPT models and Pinecone vector databases to enable semantic search and response generation. The chatbot retrieves information from a curated medical knowledge base created by processing medical PDFs, making it a valuable tool for patients, students, and healthcare professionals.

1.1 PURPOSE OF THE PROJECT

The major intention of this project is to create a smart healthcare chatbot with the aid of Generative AI that provides instant, credible responses to the queries of the users related to health. The aim is to enhance the ease of access of health information by presenting an easy-to-use interface and responding based on authentic sources.

The purpose of this project is to:

- Reduce the need for constant human interaction for initial health information.
- Close the gap between the user and reliable medical knowledge.
- Show the successful application of contemporary AI technology in addressing actual healthcare issues.

1.2 OBJECTIVES

- To build a conversational interface where users are able to pose medical questions in natural language.
- To extract information from medical PDFs, break it up into chunks, and transform it into vector embeddings.
- To build a semantic index using Pinecone for fast and accurate search results.
- To use LangChain and the Open AI GPT model to produce human-like answers from the indexed content.
- To make the chatbot provide credible and relevant health information in an accessible manner
- To create a scalable and deployable system that can be hosted on cloud platforms such as AWS.

2. LITERATURE SURVEY

The integration of artificial intelligence (AI) in healthcare has led to the development of healthcare chatbots aimed at enhancing patient care, education, and administrative efficiency. A review of recent literature reveals diverse approaches and applications in this domain:

Development and Technical Aspects:

- Large Language Models (LLMs): Al Nazi and Peng (2023) provide a comprehensive review of LLMs in healthcare, highlighting their capabilities in understanding and generating medical text, while also discussing challenges such as data privacy and model reliability.

- Medical Knowledge Integration: Liang et al. (2023) introduce the Medical Knowledge Assisted (MKA) mechanism, which incorporates medical knowledge graphs into generative models to enhance the performance of medical conversation tasks.

Applications in Healthcare:

- Disease Prediction: Zagade et al. (2024) describe an AI-based healthcare chatbot designed for disease prediction, utilizing machine learning and natural language processing to analyze user inputs and provide accurate health information.

- Cancer Therapy Support: Xu et al. (2021) review chatbot applications in oncology, discussing their roles in diagnosis, treatment, patient support, and workflow efficiency, while also addressing ethical and regulatory concerns.

User Perception and Acceptance:

- Physician Perspectives: A study by Bickmore et al. (2019) investigates physicians' perceptions of healthcare chatbots, revealing a mix of enthusiasm for potential benefits and concerns over patient privacy and the accuracy of AI-generated advice.

- Student Acceptance: Research at Ekiti State University (2023) explores undergraduate students' perceptions and acceptance of healthcare, highlighting factors influencing their willingness to engage with AI-driven health tools.

Challenges and Future Directions:

- Ethical and Cultural Considerations: Cho et al. (2023) emphasize the need to bridge computer science and medical perspectives in developing mental health conversational agents, addressing transparency, ethics, and cultural diversity.

- Technical Limitations: Safi et al. (2020) discuss the technical aspects of developing healthcare chatbots, identifying challenges such as integrating medical knowledge and ensuring conversational relevance.

This survey underscores the multifaceted advancements in healthcare chatbot development, from technical innovations and diverse applications to user perceptions and ongoing challenges, paving the way for more integrated and effective AI solutions in healthcare.

3. PROBLEM STATEMENT

A principal issue is access to credible medical information, particularly for those who do not have immediate access to healthcare providers. The majority of healthcare chatbots that currently exist use either static scripts or retrieval models, which do not allow them to comprehend and respond to sophisticated medical questions.

Problem

Develop and create a generative AI-powered healthcare chatbot to parse natural language questions and answer medically correct, context-sensitive queries with a tailor-made semantic knowledge base created from medical literature.

The chatbot must solve the following problems:

- Effective parsing of natural language inputs.
- Fetching contextual information from immense medical content.
- Generating correct and relevant responses using AI.
- Providing a seamless and interactive user experience.

4. SYSTEM ANALYSIS

4.1 EXISTING SYSTEM

Most current healthcare chatbots are retrieval-based or rule-based. Rule-based bots respond according to predefined scripts and decision trees, so they are not flexible and cannot cope with changes in user requests. Retrieval-based bots, being a little more flexible, still rely on a fixed repertoire of responses and are not very good at offering answers beyond what they have learned.

Limitations of Existing Systems:

- Not able to deal with context-rich, complicated conversations.
- Resistance to varied inputs from users.
- Restricted to established Q&A flows.
- Incapacity to dynamically enlarge with new medic data.

4.2 PROPOSED SYSTEM

The system envisaged employs Generative AI based on OpenAI's GPT models along with a semantic search engine implemented through LangChain and Pinecone. The medical content from reliable PDFs is converted into vector embeddings and pushed to a knowledge base. On receiving a question query, the system retrieves the most contextually appropriate chunks by means of semantic similarity and sends them to the GPT model to derive context-aware, human-sounding responses.

Key Features of proposed system

- Understands natural language questions.
- Generates answers from actual medical information.
- Scalable and easy to update with new information.
- Cloud-deployable and accessible through web interface.

5. SYSTEM REQUIREMENTS

5.1 SOFTWARE REQUIREMENTS

- Operating System : Linux, Windows, macOS
- Front-end : HTML, CSS, JavaScript
- Back-end : Python
- Frameworks : Flask, LangChain
- Database : Pinecone Vector DB
- Generative AI Frameworks/Libraries : GPT-3.5
- Development Environment : Visual Studio Code

5.2 HARDWARE REQUIREMENTS

- Processor : Intel i5 7 th gen or above
- Ram : 4gb or above
- SSD : 256gb or above
- Internet : High-speed internet connection
- Display : 1080p resolution monitor (optional)

5.3 FUNCTIONAL REQUIREMENTS

Functional requirements capture the particular behavior, features, or functions of the system. They establish what the system should accomplish and list tasks, services, or processes the system has to carry out in order to meet business requirements.

In this project, the chatbot must be capable of:

- User Query Input: Accept medical queries from users via a text-based interface.
- Medical Content Processing: Extract text from medical PDFs, segment it into chunks, and map them to vector embeddings.
- Semantic Search: Search the most pertinent information from the Pinecone vector

database according to user queries.

- Response Generation: Employ OpenAI's GPT model to produce a significant and context-aware answer from the content that has been retrieved.
- Interactive Chat Interface: Present answers in chat format and enable users to proceed with asking follow-up questions.
- Error Handling: Catch errors like invalid input, API problems, or no content match found.

5.4 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements specify the quality characteristics of the system. These specify the way in which the system operates its functions as opposed to what it does.

- Performance: The chatbot should answer within 3–5 seconds of getting a query. High throughput for processing multiple users at once (scalable through cloud).
- Reliability: System must be accurate in fetching and displaying medical information. Provide uptime of >99% on cloud deployment.
- Security: Secure API keys and sensitive backend endpoints. Provide secure data storage with authentication (if extended to patient data).
- Usability: Clean, responsive, and intuitive user interface. Simple input field and easy output format.
- Maintainability: Codebase must be modular for easy debugging and updates. Support for logging and monitoring.
- Portability: The application needs to be deployable across multiple platforms (cloud, local server, containers).

6. SYSTEM DESIGN

System design is the task of defining the architecture, components, modules, interfaces, and data flow of a system to meet specified requirements. It gives a blueprint to direct how the system will behave, interact with users, and combine different technologies.

It comprises both:

- High-Level Design (HLD): Total architecture, utilized technologies, and system structure.
- Low-Level Design (LLD): Detailed component design, class relationships, and interaction flows.

System design guarantees that the solution created is solid, scalable, maintainable, and fulfills all functional and non-functional requirements.

6.1 UML DIAGRAMS

UML (Unified Modeling Language) diagrams are standardized visual representations used to describe the design and behavior of a software system. They help in visualizing the system's architecture, its components, interactions, and flow of control among objects.

UML diagrams are broadly classified into:

- Structural Diagrams – Describe the static structure of the system (e.g., Class Diagram, Use Case Diagram).
- Behavioral Diagrams – Describe the dynamic behavior and interactions (e.g., Sequence Diagram).

UML diagrams improve understanding among developers, testers, and stakeholders, and serve as a blueprint during implementation.

6.1.1 USE CASE DIAGRAM

A Use Case Diagram is a type of UML diagram that represents the functional requirements of a system from the user's perspective. It shows the system's major functions (use cases) and how external actors (users or other systems) interact with them. It helps identify user roles and system interactions clearly.

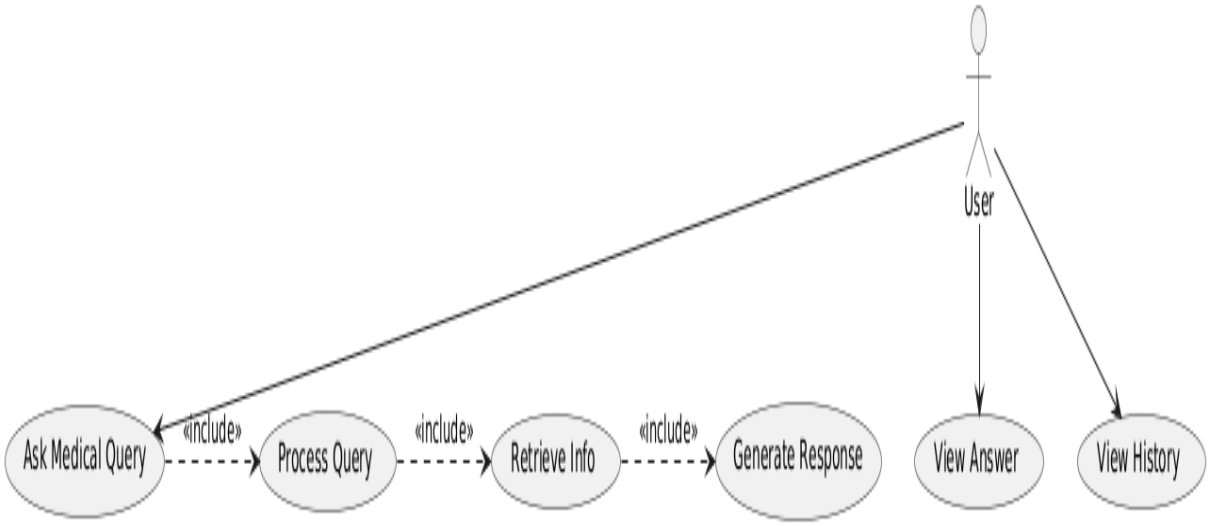

Fig 6.1.1 Use Case Diagram

- Actors
  - User: The primary actor interacting with the chatbot.
- Main Use Cases
  - Ask Medical Query
    - User initiates a medical-related question.
    - This includes:
      - Process Query
      - Retrieve Info

        - Generate Response

- View Answer
    - User receives the chatbot's response to their query.
- View History
    - User can see previously asked queries and their responses.

➢ Included Use Cases (reusable steps):

- Process Query ← included in Ask Medical Query
- Retrieve Info ← included in Process Query
- Generate Response ← included in Retrieve Info

This chaining structure effectively models how the chatbot processes a query step-by-step and ensures modularity.

6.1.2 CLASS DIAGRAM

A Class Diagram is a structural UML diagram that depicts the classes within a system, along with their attributes, methods, and the relationships among them. It helps in understanding the object-oriented structure of the system and serves as the blueprint for coding.

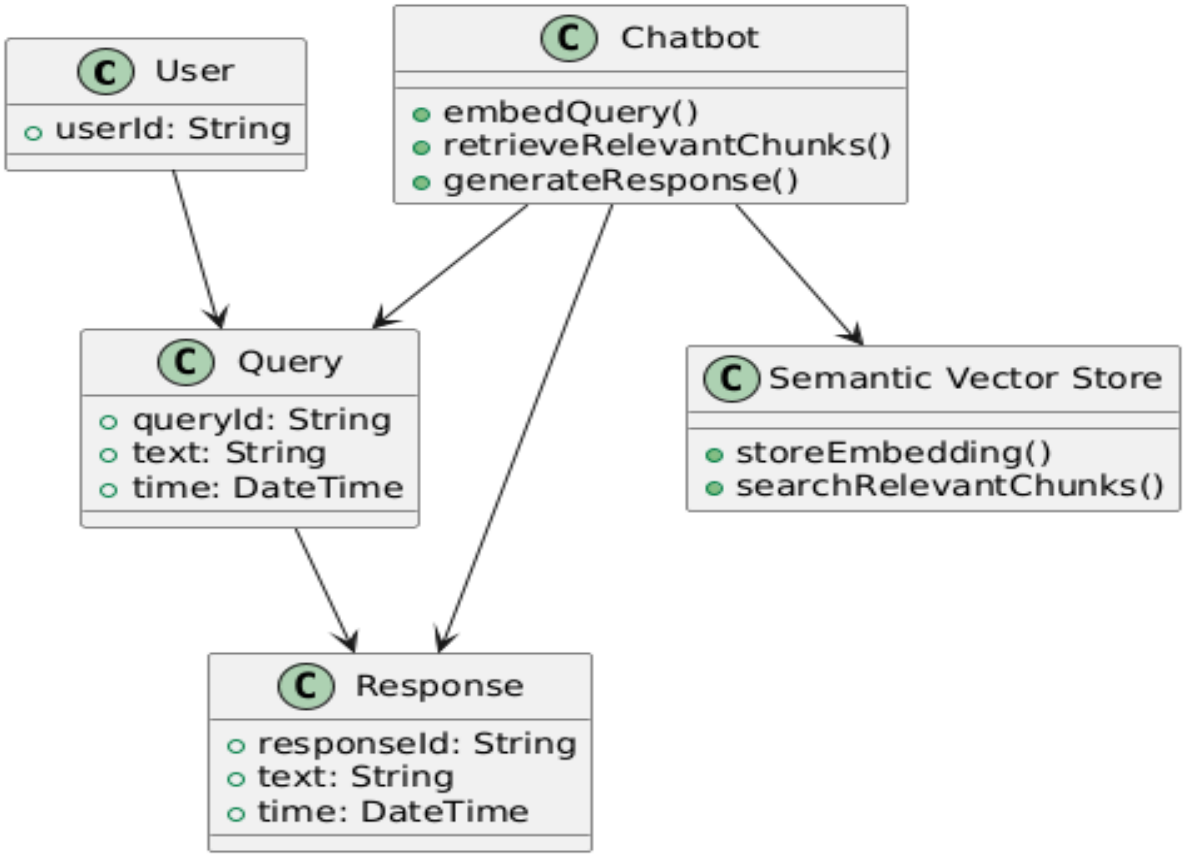

Fig 6.1.2 Class Diagram

- Classes and Their Responsibilities
  - User
    - Attributes:
      - userId: String

- Associations:
  - Can create Query instances.

- Query
  - Attributes:
    - queryId: String
    - text: String
    - time: DateTime
  - Associations:
    - Connected to both User and Response
    - Handled by the Chatbot class.

- Response
  - Attributes:
    - responseId: String
    - text: String
    - time: DateTime
  - Associations:
    - Generated from a Query

- Chatbot
  - Methods:
    - embedQuery()

- retrieveRelevantChunks()
- generateResponse()

- Associations:
  - Interacts with both Query and Response
  - Connects to Semantic Vector Store for relevant info

➢ Semantic Vector Store

- Methods:
  - storeEmbedding()
  - searchRelevantChunks()

- Associations:
  - Works closely with the Chatbot to find relevant information.

6.1.3 SEQUENCE DIAGRAM

A Sequence Diagram is a type of behavioral UML diagram that shows how objects interact in a specific sequence of events. It maps the flow of messages or actions between different components of the system, highlighting the order and timing of function calls and data transfer.

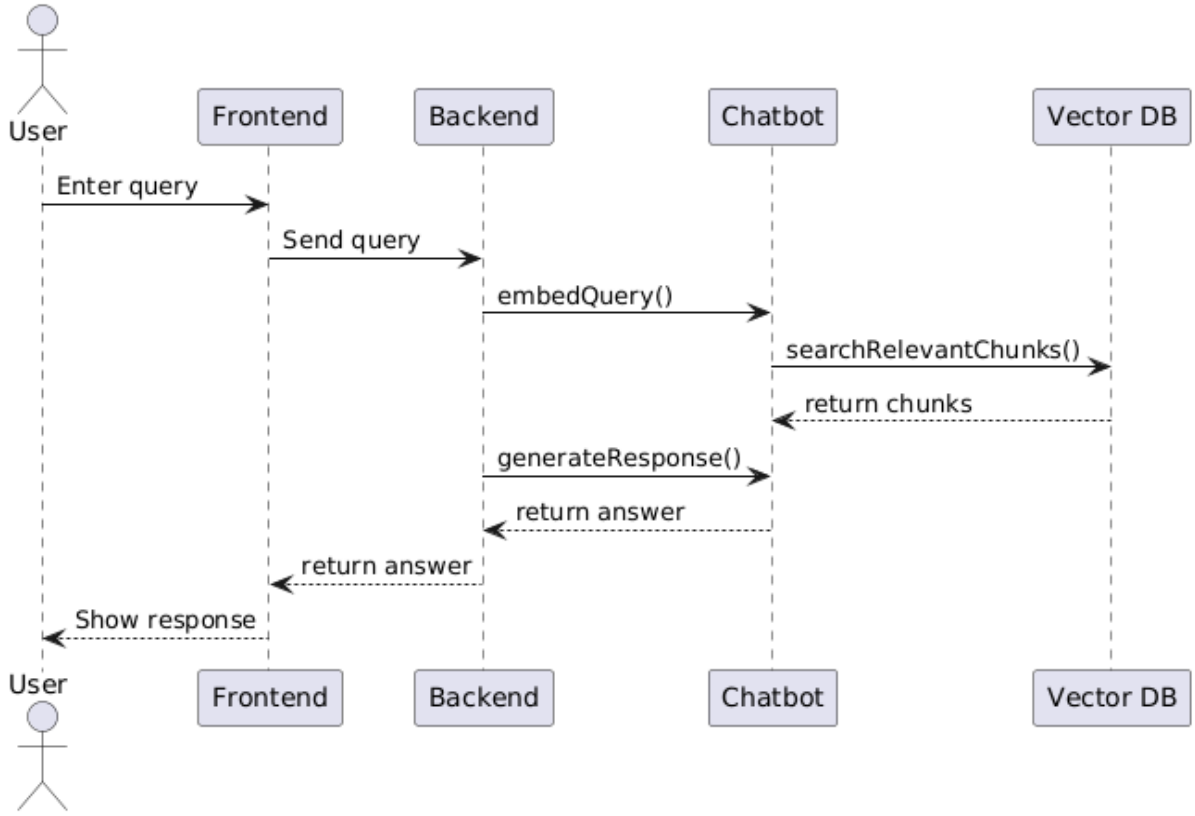

Fig 6.1.3 Sequence Diagram

- Actors and Components:
  - User
  - Frontend
  - Backend
  - Chatbot

- Vector DB

➢ Sequence Flow:

- User → Frontend:

  Enter query — User types in a medical question.

- Frontend → Backend:

  Send query — Frontend sends the input to the backend.

- Backend → Chatbot:

  embedQuery() — Backend passes the query to the chatbot for embedding.

- Chatbot → Vector DB:

  searchRelevantChunks() — Chatbot searches for relevant chunks in the vector database.

- Vector DB → Chatbot:

  return chunks — Vector DB sends back the matched chunks.

- Chatbot:

  generateResponse() — Chatbot uses the chunks to generate a medical response.

- Chatbot → Backend:

  return answer — The generated response is returned to the backend.

- Backend → Frontend:

  return answer — Backend passes it back to the frontend.

- Frontend → User:

  Show response — Final response is displayed to the user.

6.2 SYSTEM ARCHITECTURE

The system architecture of the Healthcare Chatbot is designed to provide an intelligent, scalable, and efficient flow for processing user medical queries using modern AI and NLP technologies. The architecture is modular and comprises the following key components:

- **User**
  - The end user interacts with the chatbot by entering a medical query.
  - Queries may relate to symptoms, medication, diseases, or general health.

- **Frontend (HTML, CSS, JS, Bootstrap)**
  - Acts as the user interface.
  - Collects input queries and displays chatbot responses.
  - Sends requests to the backend through REST APIs.

- **Backend (Flask / Django)**
  - Serves as the bridge between frontend and chatbot core.
  - Accepts HTTP requests and forwards them to the chatbot logic.
  - Returns the final generated answer back to the frontend.

- **Chatbot Core (Python, LangChain, OpenAI) Handles core AI logic**:
  - embedQuery(): Converts the query into vector form using language models.
  - searchRelevantChunks(): Retrieves relevant context from the knowledge base using similarity search.
  - generateResponse(): Constructs a meaningful and accurate answer using generative AI (e.g., OpenAI GPT models).

- **Semantic Vector Store (Pinecone / FAISS)**
  - Stores the vector embeddings of processed medical documents.
  - Performs fast similarity search to retrieve relevant chunks based on input queries.
  - Returns top matching document vectors for response generation.

- **Medical Knowledge Base (PDFs, Articles)**
  - Collection of domain-specific documents like medical textbooks, clinical guidelines, articles, etc.
  - Pre processed into chunks, embedded into vectors, and indexed into the vector store during setup.

- **Data Flow**
  - User inputs a query via Frontend.
  - Frontend sends the query to Backend.
  - Backend invokes Chatbot Core, which:
  - Embeds the query.
  - Searches relevant chunks via Vector Store.
  - Generates a contextual response.

The final answer is passed back through the backend to the frontend and displayed to the user.

The system follows a modular, layered architecture consisting of:

- Frontend – User interface for interacting with the chatbot.
- Backend (Flask App) – Handles requests, manages logic, and communicates with AI APIs and databases.
- PDF Processing & Embedding Layer – Extracts text from PDFs, chunks it, and generates vector embeddings.
- Vector Database (Pinecone) – Stores and retrieves embeddings based on user query similarity.
- LLM Layer (OpenAI) – Generates natural language responses using GPT.
- Cloud Deployment (AWS) – Hosts the application for accessibility and scalability.

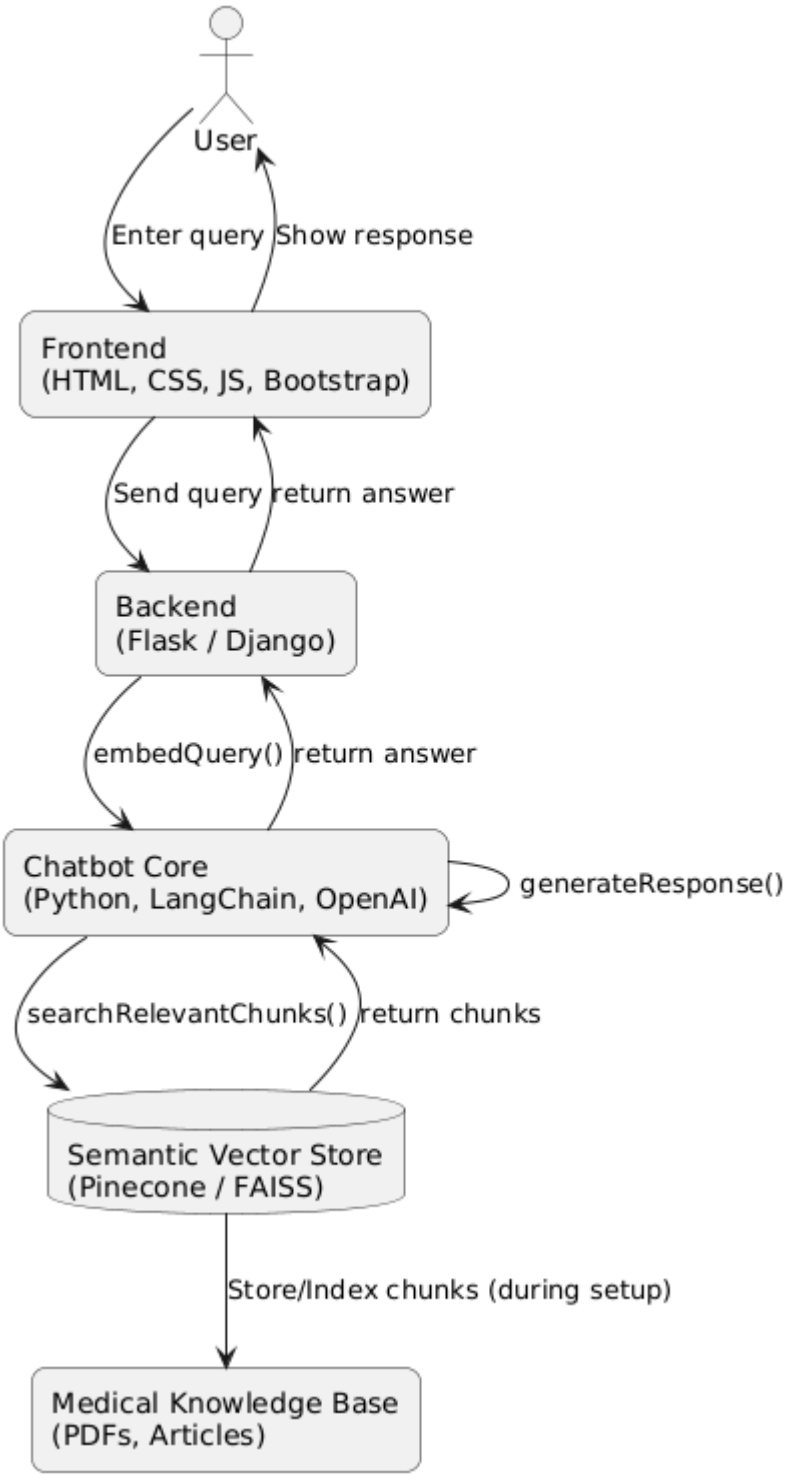

Fig 6.2 System Architecture

6.3 I/O DESIGN

The Input/Output (I/O) design phase plays a crucial role in defining how data enters the system and how results are presented to the user. In our healthcare chatbot system, the I/O design focuses on creating user-friendly, efficient, and error-free data entry and result display interfaces. It ensures data is collected in a structured format and presented in an understandable and interactive manner, enhancing user experience and system reliability.

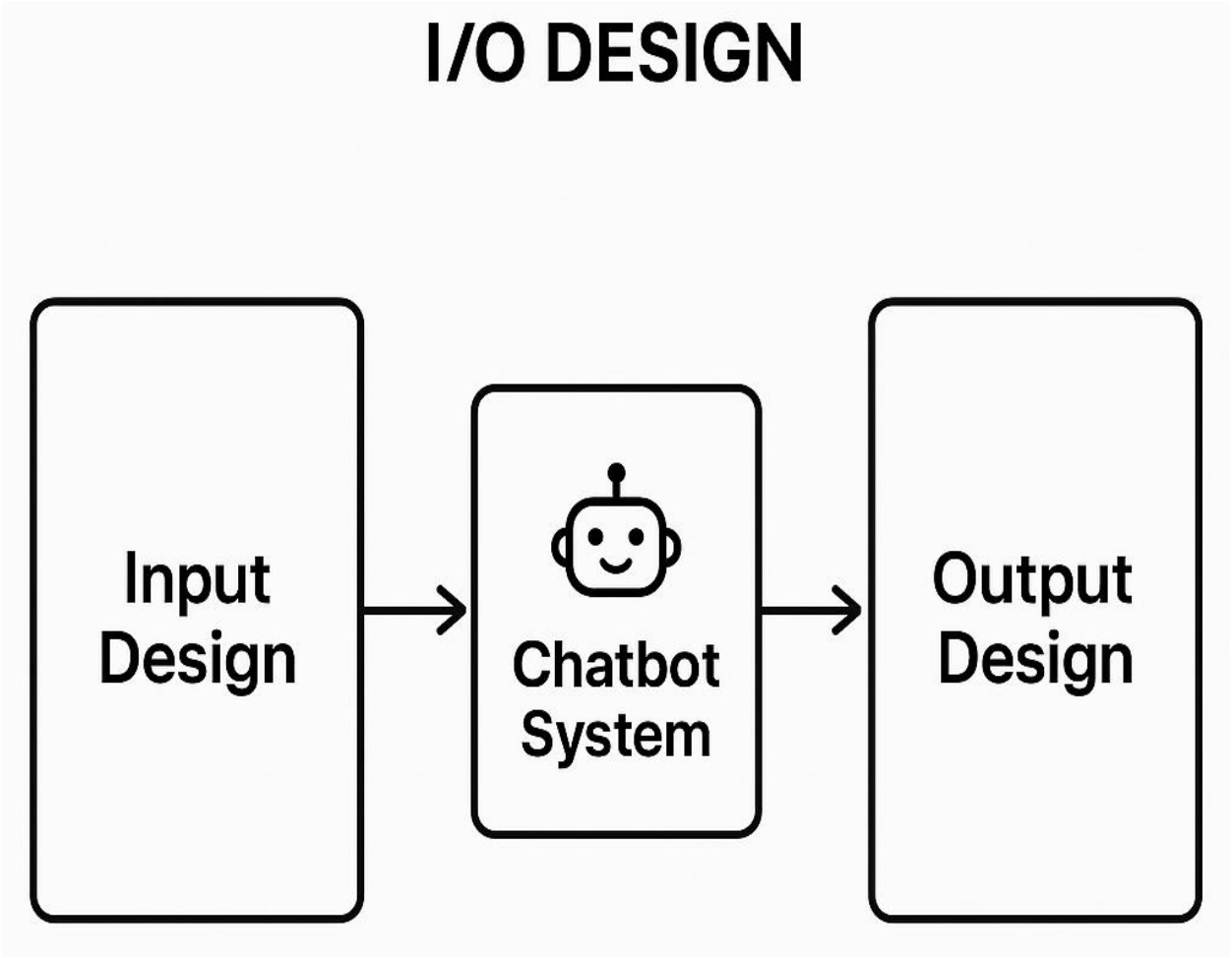

Fig 6.3 I/O Design

6.3.1 INPUT DESIGN

Input design defines the process and formats through which users provide data to the system. The system accepts inputs through a web-based interface where users can type their health-related queries in natural language. The primary goals of input design in this project are simplicity, validation, and user-friendliness.

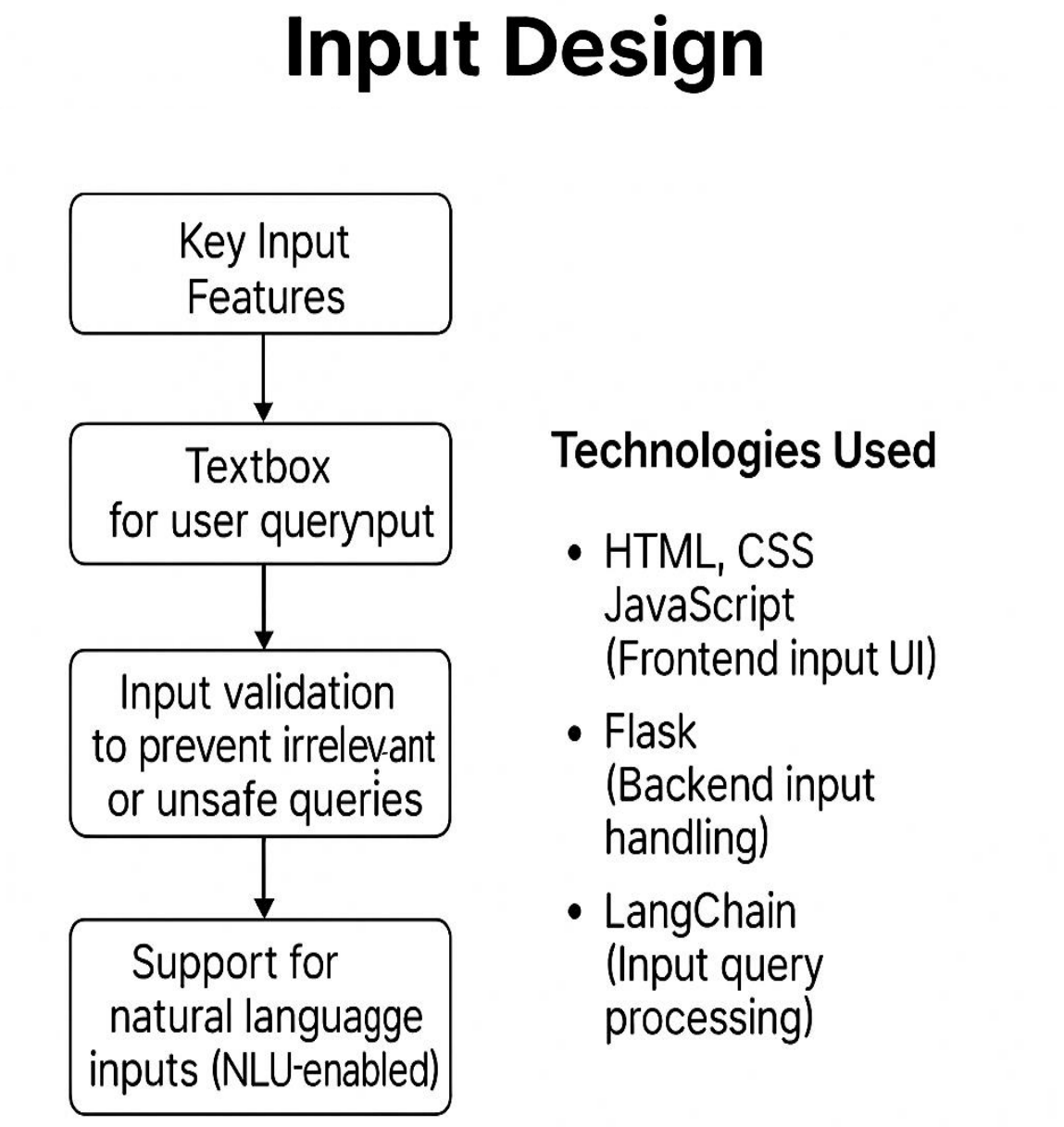

Fig 6.3.1 Input Design

- **Key Input Features:**
  - Textbox for user query input.
  - Input validation to prevent irrelevant or unsafe queries.
  - Input length control to ensure model performance.
  - Support for natural language inputs (NLU-enabled).

- **Technologies Used:**
  - HTML, CSS, JavaScript (Frontend input UI)
  - Flask (Backend input handling)
  - LangChain (Input query processing)

6.3.2 OUTPUT DESIGN

Output design focuses on how the system presents the retrieved and generated information to the user. The chatbot generates responses using the semantic search mechanism on the embedded knowledge base and delivers natural language answers.

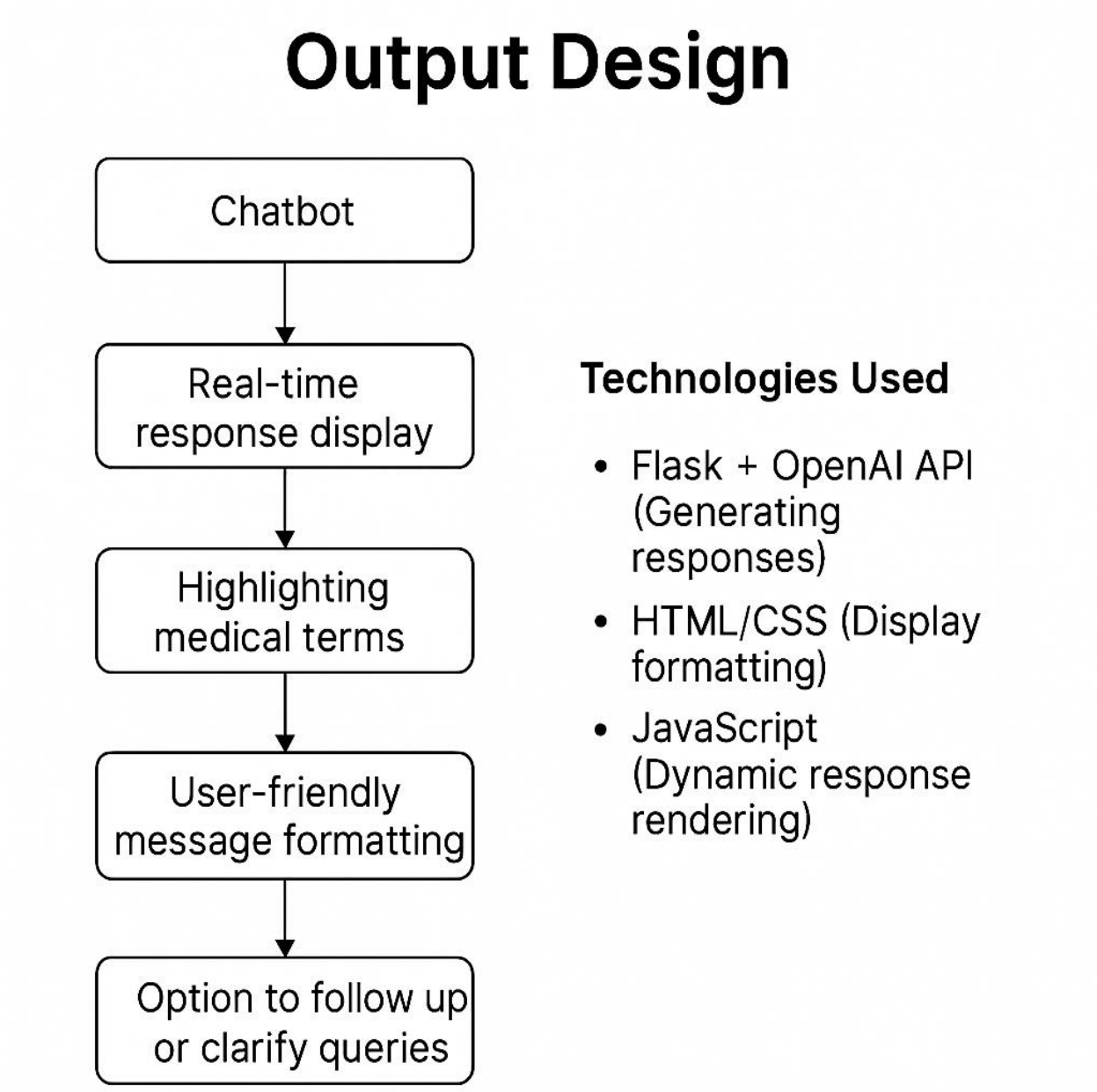

Fig 6.3.2 Output Design

- **Key Output Features:**
  - Real-time response display from the chatbot.
  - Highlighted medical terms or keywords for better understanding.
  - User-friendly message formatting.
  - Option to follow up or clarify queries.

- **Technologies Used:**
  - Flask + OpenAI API (Generating responses)
  - HTML/CSS (Display formatting)
  - JavaScript (Dynamic response rendering)

7. IMPLEMENTATION

The implementation phase involves developing the chatbot system using selected technologies and integrating various components like AI, embeddings, and database systems.

The project is implemented using Python (Flask) for backend development, HTML,CSS, JavaScript for frontend design, and OpenAI API with LangChain for generating intelligent medical responses. The system processes user queries, converts them into embeddings, retrieves relevant context from Pinecone Vector Database, and generates responses using GPT.The application is deployed using Microsoft Azure Cloud services. The development environment used for building and testing the application is Visual Studio Code (VS Code).

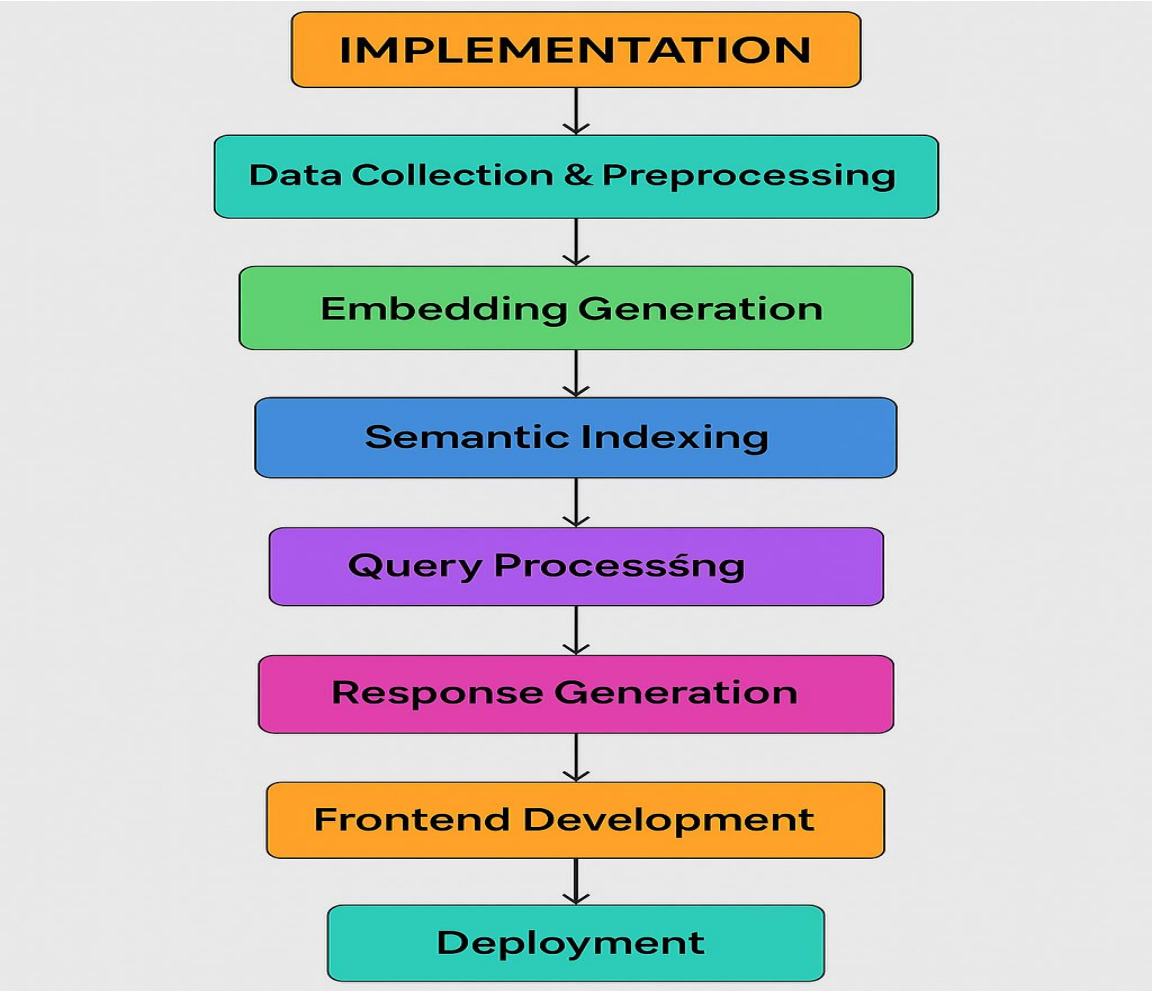

Fig 7 Implementation

This project is implemented using the following steps:

- **Data Collection & Preprocessing**

  In this phase, medical knowledge was extracted from authoritative textbooks, research articles, and medical PDFs. The data was cleaned by removing unnecessary characters, correcting formatting issues, and standardizing terminology. The text was then split into manageable chunks of consistent size to ensure compatibility with embedding generation models. Tokenization and normalization techniques were applied to improve downstream processing accuracy.

- **Embedding Generation**

  Each preprocessed chunk of medical text was converted into numerical vectors (embeddings) using state-of-the-art embedding models such as OpenAI's text-embedding-ada-002. These embeddings capture the semantic meaning of the text and form the basis for semantic search and retrieval tasks.

- **Semantic Indexing**

  The generated embeddings were indexed using Pinecone, a vector database optimized for similarity search. This semantic index enables fast and accurate retrieval of the most contextually relevant text chunks based on user queries. Metadata such as source titles and chunk positions were also stored to maintain context.

- **Query Processing**

  When a user inputs a query, it is converted into an embedding vector using the same embedding model. This vector is then used to perform a similarity search against the indexed vectors in Pinecone. The top-k relevant results (text chunks) are retrieved for generating a meaningful response.

- **Response Generation**

  The retrieved chunks are combined and passed to a language model (such as OpenAI's GPT-3.5/4) to generate a coherent and informative answer. Prompt engineering techniques are used to guide the language model to provide responses that are accurate, helpful, and contextually appropriate within the medical domain.

- **Frontend Development**

  The user interface was built using HTML, CSS, Bootstrap, and JavaScript. The frontend allows users to enter queries and view chatbot responses in a clean, responsive layout. The design ensures ease of use, especially for non-technical users such as patients or healthcare workers.

- **Deployment**

  The application backend, built using Flask, was deployed on AWS Cloud to ensure high availability and scalability. Pinecone was used as the vector database in production, and API calls to OpenAI were managed securely. The system was tested for responsiveness, reliability, and efficiency under various loads.

7.1 TECHNOLOGY STACK

A technology stack refers to the collection of tools, frameworks, programming languages, libraries, and platforms used to develop, run, and manage a software application.

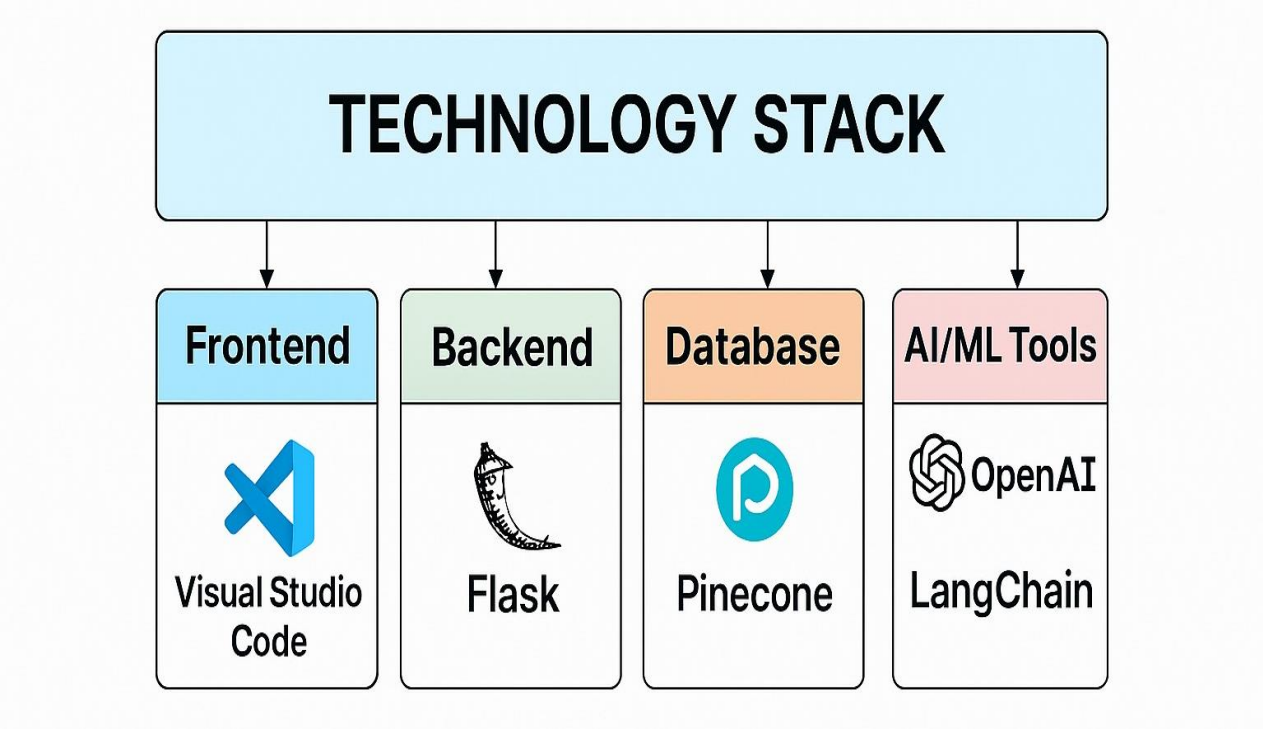

Fig 7.1 Technology Stack

It typically includes:

- Frontend technologies – Tools and languages used to build the user interface.
- Backend technologies – Frameworks and languages that handle the logic, database, and server-side operations.
- Database technologies – Systems used to store, manage, and retrieve data.
- AI/ML tools – APIs or frameworks used for implementing intelligence in applications.
- Deployment platforms – Services used to host and run the application (e.g., Azure, AWS).

In this Project, the Technology Stack Includes:

- **OpenAI (Natural Language Processing)**

  OpenAI provides state-of-the-art language models that empower the chatbot with advanced natural language understanding and generation capabilities. It interprets user queries related to medical topics and generates accurate, context-aware responses. This forms the core intelligence of the chatbot, making it interactive and human-like in its communication.

- **LangChain (AI Integration Framework)**

  LangChain acts as a powerful framework that connects OpenAI models with external data sources. In this project, it is used to:

  - Embed user queries and context documents
  - Manage prompt templates
  - Build chains that connect vector search results with OpenAI for contextual response generation This allows the chatbot to deliver reliable and accurate answers by grounding its responses in factual, pre-indexed medical data.

- **Pinecone (Cloud-Based Vector Database)**

  Pinecone is used to store and manage vector embeddings generated from medical PDFs. It provides fast, real-time semantic search capabilities by matching user query embeddings with the most relevant information chunks. This ensures the responses are accurate, contextually relevant, and based on verified medical knowledge.

- **Flask (Backend Framework)**

  Flask is a lightweight Python web framework used to handle server-side logic. It:

  - Accepts user inputs from the frontend
  - Sends queries to the AI model via LangChain

- Receives the generated response and displays it on the UI
- It acts as the communication bridge between the user interface and the AI system.

➢ **Visual Studio Code (Development Environment)**

Visual Studio Code is the primary IDE used for developing the chatbot. It offers:

- Syntax highlighting and debugging tools.
- Integrated terminal for running Flask server.
- Seamless handling of both frontend and backend code.

VS Code ensures an efficient and smooth development and testing experience throughout the project lifecycle.

7.2 MODULES

Modules in a project refer to the divided functional components that work together to build the complete system. Each module handles a specific task or feature of the project.

The system is divided into several interdependent modules, each performing a specific function to enable seamless interaction between the user and the chatbot.

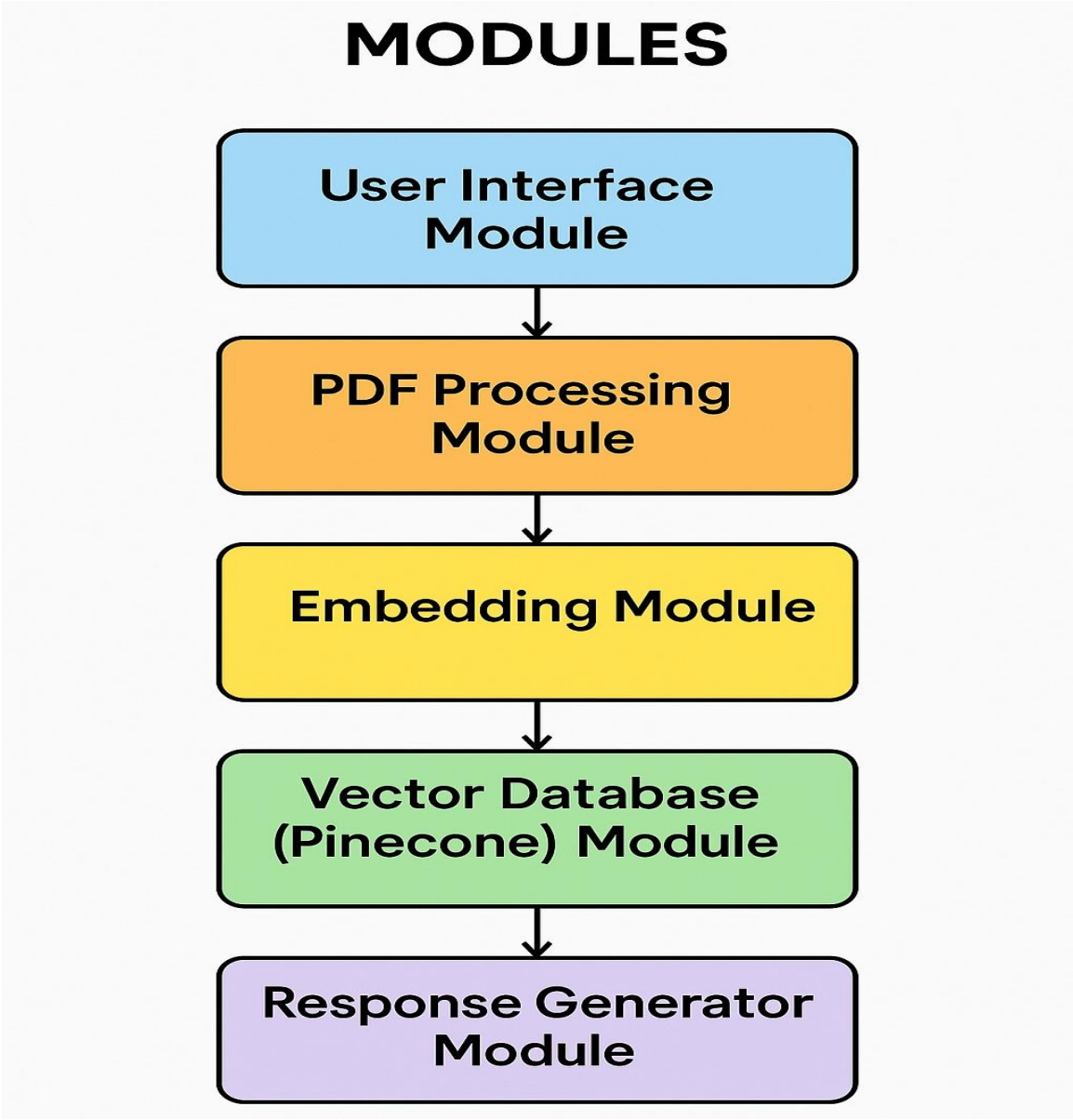

Fig 7.2 Modules

- **User Interface Module**

  This module is responsible for the interaction between the user and the system.

  - Provides a simple, clean interface where users can type their medical queries.
  - Ensures real-time display of chatbot responses using frontend technologies like HTML, CSS, Bootstrap, and JavaScript.
  - Handles input validation to prevent empty or irrelevant queries.

- **PDF Processing Module**

  This module handles the ingestion of medical knowledge sources.

  - Allows the admin to upload medical PDFs containing verified content.
  - Extracts raw text from PDFs using libraries like PyMuPDF or pdfminer.
  - Splits the extracted text into smaller, manageable chunks to ensure better understanding and embedding quality.

- **Embedding Module**

  This module prepares the textual data for semantic search.

  - Converts each text chunk into a vector embedding using OpenAI's Embedding API.
  - These embeddings capture the semantic meaning of the content, making it easier for the model to understand context.
  - Optimizes the text format to ensure high-quality embeddings.

- **Vector Database (Pinecone) Module**

  Handles fast and accurate retrieval of relevant information.

  - Stores all generated embeddings in the Pinecone vector database.
  - When a user asks a question, the system converts it to an embedding and performs a similarity search to retrieve the most relevant text chunks.
  - Enables low-latency, high-accuracy semantic search functionality.

- **Response Generator Module**

  This is the intelligence layer of the system.

  - Uses LangChain to combine the retrieved context with the user's query.
  - Passes the combined data to OpenAI's GPT model, which generates a natural, context-aware response.
  - Ensures the response is medically relevant, concise, and easy to understand.

8. CODING

The coding phase involves translating the design and logic into a functional application using various programming languages and tools. The codebase is organized into modular components to maintain clarity and scalability.

Key Components Covered in Code:

- **Backend (Flask – Python)**
  - Integrates with LangChain, OpenAI, and Pinecone
  - Processes PDF files and manages embeddings
- **Frontend (HTML/CSS/Bootstrap/JavaScript)**
  - Provides a simple interface for user interaction
  - Sends queries to the backend and displays responses
  - Ensures a responsive and user-friendly experience
- **AI & NLP Integration**
  - Uses OpenAI API to generate responses
  - Embeds content using OpenAI's embedding model
  - Uses LangChain to construct prompt chains
- **Vector Database Interaction (Pinecone)**
  - Uploads and retrieves embeddings
  - Performs similarity search based on user input
- **PDF to Text Conversion**
  - Extracts and chunks text data from uploaded PDFs
  - Prepares content for embedding and storage

Backend Code – app.py

The backend logic for the healthcare chatbot is implemented in the app.py file using Flask**,** OpenAI**,** LangChain**,** and Pinecone**.** This script handles the entire flow from receiving user queries, processing them through vector search, and generating accurate responses using GPT models.

Key Functionalities:

- Initializes the Flask application and defines the route for handling user input.
- Extracts text from medical PDFs and splits them into smaller chunks.
- Converts text chunks into embeddings using OpenAI.
- Stores and retrieves embeddings from Pinecone vector database.
- Uses LangChain's ConversationalRetrievalChain to manage context and generate responses.

Below is the complete source code of the app.py file:

```
from flask import Flask, render_template, jsonify, request
from langchain_openai import AzureChatOpenAI
from src.helper import download_hugging_face_embeddings
from langchain_pinecone import Pinecone
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from dotenv import load_dotenv
from src.prompt import *
import os
app = Flask(__name__)
# Load environment variables
load_dotenv()
# Fetch API keys safely
```

```
PINECONE_API_KEY=
"pcsk_43dbdq_BFamaizKSRo6uBjRPigeHLkW2BLVTLJNUWXMuiDC8U9R5KkJygecffhx
NkxnofE"
AZURE_OPENAI_KEY=
"Aku4PbwrUxVVFGdtIuzDMUAiHVV1HE1dA1uDHuwDMsxMDswx7lOvJQQJ99BCACf
hMk5XJ3w3AAAAACOGzuaI"
AZURE_OPENAI_ENDPOINT = "https://rutwi-m8oatrp1-swedencentral.openai.azure.com/"
AZURE_OPENAI_DEPLOYMENT = "gpt-4"
AZURE_OPENAI_VERSION = "2024-05-01-preview"  # Updated to a valid API version
# Set environment variables
os.environ["PINECONE_API_KEY"] = PINECONE_API_KEY
os.environ["AZURE_OPENAI_KEY"] = AZURE_OPENAI_KEY
# Load embeddings
embeddings = download_hugging_face_embeddings()
index_name = "test"
# Load Pinecone index
docsearch = Pinecone.from_existing_index(
    index_name=index_name,
    embedding=embeddings
)
retriever = docsearch.as_retriever(search_type="similarity", search_kwargs={"k": 3})
# Initialize Azure OpenAI
llm = AzureChatOpenAI(
    openai_api_key=AZURE_OPENAI_KEY,
    openai_api_version=AZURE_OPENAI_VERSION,
    azure_deployment=AZURE_OPENAI_DEPLOYMENT,
    azure_endpoint=AZURE_OPENAI_ENDPOINT,
    temperature=0.4,
    max_tokens=500
)
# Updated medical check using Runnable interface
```

```
medical_check_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a medical query classifier.
    Determine if the user's question is related to medical topics, health, diseases, treatments,
    medications, health conditions, wellness, or medical procedures.
    Respond with ONLY 'yes' if it's medical or 'no' if it's not medical."""),
    ("human", "{input}")
])
medical_check_chain = medical_check_prompt | llm
# Main RAG chain for medical answers
prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    ("human", "{input}"),
])
question_answer_chain = create_stuff_documents_chain(llm, prompt)
rag_chain = create_retrieval_chain(retriever, question_answer_chain)
@app.route("/")
def index():
    return render_template('chat.html')
@app.route("/get", methods=["GET", "POST"])
def chat():
    try:
        user_message = request.form['msg']
        print("Received message:", user_message)
        # First check if the question is medical
        medical_check = medical_check_chain.invoke({"input": user_message})
        print("Medical check result:", medical_check)
        # Get just the yes/no answer, trim whitespace and lowercase
        is_medical = medical_check.content.strip().lower() if hasattr(medical_check, 'content')
else str(medical_check).strip().lower()
        # Only process medical questions with RAG
        if is_medical == "yes":
```

```
            response = rag_chain.invoke({"input": user_message})
            bot_reply = response.get("answer", "No answer found.")
        else:
            bot_reply = "I'm a medical assistant and can only answer medical-related questions. Could you please ask a health-related question instead?"
        return jsonify({'response': bot_reply})
    except Exception as e:
        print("Error in /get endpoint:", str(e))
        return jsonify({'error': str(e)}), 500
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8080, debug=True)
```

Testing and Trials – trials.ipynb

To validate and test the core components of the chatbot before full integration, experimental implementations were carried out in a Jupyter Notebook named trials.ipynb.

Purpose of Trials:

- To test PDF text extraction and chunking strategies.
- To experiment with OpenAI embedding models and observe embedding generation.
- To connect and verify Pinecone indexing and retrieval functionality.
- To validate LangChain integration with the OpenAI model for response generation.

The trials ensured each module works independently and correctly before final integration into the app.py. This iterative approach allowed safe debugging and improved the reliability of the final system.

Below is the complete source code of the trails.ipynb file:

```
# Print statement for initial test
print("Ok")
# Change the current working directory
```

```
%pwd
import os
os.chdir("../")
%pwd
# ------------------------------
# Load PDFs and Split into Chunks
# ------------------------------
from langchain.document_loaders import PyPDFLoader, DirectoryLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
# Extract Data From the PDF Files in 'Data/' directory
def load_pdf_file(data):
    loader = DirectoryLoader(data,
                             glob="*.pdf",
                             loader_cls=PyPDFLoader)
    documents = loader.load()
    return documents
# Load the documents
extracted_data = load_pdf_file(data='Data/')
# Split the PDF text into manageable chunks for embedding
def text_split(extracted_data):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=20)
    text_chunks = text_splitter.split_documents(extracted_data)
    return text_chunks
# Split the loaded documents into chunks
text_chunks = text_split(extracted_data)
print("Length of Text Chunks", len(text_chunks))
# ------------------------------
# Download Sentence Transformers Embedding Model
# ------------------------------
from langchain.embeddings import HuggingFaceEmbeddings
# Load Hugging Face sentence-transformers model for embeddings
```

```
def download_hugging_face_embeddings():
    embeddings=HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-
L6-v2")
    return embeddings
# Instantiate the embeddings model
embeddings = download_hugging_face_embeddings()
# Test the embedding model
query_result = embeddings.embed_query("Hello world")
print("Length", len(query_result))
# ------------------------------
# Set API Keys and Pinecone Initialization
# ------------------------------
PINECONE_API_KEY=os.environ.get('PINECONE_API_KEY')
AZURE_OPENAI_KEY=os.environ.get('AZURE_OPENAI_KEY')
AZURE_OPENAI_DEPLOYMENT=os.environ.get('AZURE_OPENAI_DEPLOYMENT')
AZURE_OPENAI_VERSION=os.environ.get('AZURE_OPENAI_VERSION')
AZURE_OPENAI_ENDPOINT=os.environ.get('AZURE_OPENAI_ENDPOINT')
from pinecone import Pinecone, ServerlessSpec
# Initialize Pinecone client
pc=
Pinecone(api_key="pcsk_43dbdq_BFamaizKSRo6uBjRPigeHLkW2BLVTLJNUWXMuiDC
8U9R5KkJygecffhxNkxnofE")
# Define Pinecone index name
index_name = "test"
# Create the Pinecone index if it doesn't exist already
if index_name not in pc.list_indexes().names():
    pc.create_index(
        name=index_name,
        dimension=384,
        metric="cosine",
        spec=ServerlessSpec(
```

```
            cloud="aws",
            region="us-east-1"
        )
    )
    print(f"Index '{index_name}' created successfully.")
else:
    print(f"Index '{index_name}' already exists.")
# ------------------------------
# Upload Embeddings to Pinecone
# ------------------------------
from langchain_community.vectorstores import Pinecone
# Upload text chunks as vector embeddings to Pinecone
docsearch = Pinecone.from_documents(
    documents=text_chunks,
    index_name=index_name,
    embedding=embeddings,
)
# OR: Load existing index (if already uploaded)
docsearch = Pinecone.from_existing_index(
    index_name=index_name,
    embedding=embeddings
)
# ------------------------------
# Set Up Retriever
# ------------------------------
# Create retriever to fetch top 3 relevant chunks
retriever = docsearch.as_retriever(search_type="similarity", search_kwargs={"k":3})
# Test the retriever with a sample query
retrieved_docs = retriever.invoke("What is Acne?")
print(retrieved_docs)
# ------------------------------
```

```
# Configure Azure OpenAI LLM
# -----------------------------
from langchain_openai import OpenAI
# Set and check the Azure OpenAI API Key
import os
os.environ["AZURE_OPENAI_KEY"] = "847a00b5e2044e26a82689414d331c87"
api_key = os.getenv("AZURE_OPENAI_KEY")
if api_key:
   print("API Key is set successfully!")
else:
   print("API Key is not set.")
from langchain_openai import AzureOpenAI
# Initialize Azure OpenAI LLM (text generation model)
llm = AzureOpenAI(
   azure_endpoint="https://scremer-ai.openai.azure.com/",
   api_key="847a00b5e2044e26a82689414d331c87",
   azure_deployment="gpt-35-turbo",
   azure_model="gpt-35-turbo",
   api_version="2024-05-01-preview",
   temperature=0.4,
   max_tokens=500
)
# -----------------------------
# Prompt Template for QA
# -----------------------------
from langchain.prompts import ChatPromptTemplate
# System prompt template
system_prompt = (
   "You are an assistant for question-answering tasks. "
   "Use the following pieces of retrieved context to answer "
   "the question. If you don't know the answer, say that you "
```

```
    "don't know. Use three sentences maximum and keep the "
    "answer concise.\n\n"
    "{context}"
  )
  # Full prompt for conversation
  prompt = ChatPromptTemplate.from_messages(
    [
      ("system", system_prompt),
      ("human", "{input}"),
    ]
  )
  # ------------------------------
  # Conversational Retrieval Chain (RAG)
  # ------------------------------
  from langchain.chains import ConversationalRetrievalChain
  from langchain_openai import AzureChatOpenAI
  # Use AzureChatOpenAI (chat-style) instead of basic LLM
  llm = AzureChatOpenAI(
    azure_endpoint="https://scremer-ai.openai.azure.com/",
    api_key="847a00b5e2044e26a82689414d331c87",
    azure_deployment="gpt-35-turbo",
    api_version="2024-05-01-preview",
    temperature=0.4,
    max_tokens=500
  )
  # Build the Conversational RAG Chain
  rag_chain = ConversationalRetrievalChain.from_llm(llm, retriever=retriever)
  # ------------------------------
  # Ask a Question via RAG Chain
  # ------------------------------
  # Example 1
```

```
chat_history = []
response = rag_chain.invoke({
   "question": "What is Acromegaly and gigantism?",
   "chat_history": chat_history
})
print(response["answer"])
# Example 2
chat_history = []
response = rag_chain.invoke({
   "question": "What is stats?",
   "chat_history": chat_history
})
print(response["answer"])
```

Frontend Code – chat.html

The chat.html file provides the user interface for the healthcare chatbot. It is responsible for collecting user input, displaying the conversation, and interacting with the backend via AJAX. This page is styled using Bootstrap and custom CSS to ensure a responsive and visually appealing layout.

Key Functionalities:

- Displays a chat interface where users can input medical queries.
- Sends user queries to the backend using jQuery and AJAX without reloading the page.
- Displays both user messages and chatbot responses in a conversational format.
- Integrates seamlessly with Flask for dynamic content rendering like images and styles.

Below is the complete source code of the chat.html file:

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Chatbot</title>
    <link                                                                rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
        integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css"
        integrity="sha384-
B4dIYHKNBt8Bc12p+WXckhzcICo0wtJAoU8YZTY5qE0Id1GSseTk6S+L3BlXeVIU"
crossorigin="anonymous">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css')}}"/>
</head>
<body>
<div class="container-fluid h-100">
    <div class="row justify-content-center h-100">
        <div class="col-md-8 col-xl-6 chat">
            <div class="card">
                <div class="card-header msg_head">
                    <div class="d-flex bd-highlight">
                        <div class="img_cont">
                            <!-- Replace this image with your downloaded image -->
                            <img src="{{ url_for('static', filename='images/icon.png') }}"
                                class="rounded-circle user_img">
                            <span class="online_icon"></span>
                        </div>
                        <div class="user_info">
                            <span>Healthcare Chatbot</span>
                            <p>Ask me anything!</p>
                        </div>
                    </div>
```

```
            </div>
            <div id="messageFormeight" class="card-body msg_card_body">
            </div>
            <div class="card-footer">
               <form id="messageArea" class="input-group">
                  <input type="text" id="text" name="msg" placeholder="What is your
Query..." autocomplete="off"
                        class="form-control type_msg" required/>
                  <div class="input-group-append">
                     <button type="submit" id="send" class="input-group-text send_btn"><i
                           class="fas fa-location-arrow"></i></button>
                  </div>
               </form>
            </div>
         </div>
      </div>
   </div>
   <script>
      $(document).ready(function() {
         // Auto-scroll to bottom of chat
         function scrollToBottom() {
            $("#messageFormeight").scrollTop($("#messageFormeight")[0].scrollHeight);
         }
         $("#messageArea").on("submit", function(event) {
            event.preventDefault();
            const date = new Date();
            const hour = date.getHours();
            const minute = date.getMinutes();
            const str_time = hour + ":" + minute;
            var rawText = $("#text").val().trim();
```

```
        if (!rawText) return;  // Don't send empty messages
        // Append user message to chat
        var userHtml = '<div class="d-flex justify-content-end mb-4"><div class="msg_cotainer_send">' + rawText + '<span class="msg_time_send">' + str_time + '</span></div><div class="img_cont_msg"><img src="{{ url_for('static', filename='images/user.png') }}" class="rounded-circle user_img_msg"></div></div>';
        $("#text").val("");
        $("#messageFormeight").append(userHtml);
        scrollToBottom();
        // Send message to backend
        $.ajax({
            url: "/get",
            type: "POST",
            data: {msg: rawText},
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded'
            },
            success: function(data) {
                console.log("Response from backend:", data);
                // Display bot response
                var botHtml = '<div class="d-flex justify-content-start mb-4"><div class="img_cont_msg"><img src="{{ url_for('static', filename='images/icon.png') }}" class="rounded-circle user_img_msg"></div><div class="msg_cotainer">' + (data.response || data.error) + '<span class="msg_time">' + str_time + '</span></div></div>';
                $("#messageFormeight").append(botHtml);
                scrollToBottom();
            },
            error: function(xhr, status, error) {
                console.error("AJAX error:", error);
                var botHtml = '<div class="d-flex justify-content-start mb-4"><div class="img_cont_msg"><img src="{{ url_for('static', filename='images/icon.png') }}"
```

```
class="rounded-circle user_img_msg"></div><div class="msg_cotainer">Sorry, I encountered an error. Please try again.<span class="msg_time">' + str_time + '</span></div></div>';
                $("#messageFormeight").append(botHtml);
                scrollToBottom();
            }
        });
    });
});
</script>
</body>
</html>
```

Styling File – style.css

The style.css file defines the visual appearance of the chatbot's user interface. It enhances the HTML elements with aesthetic design, providing a responsive and modern chat layout. The styles are built on top of Bootstrap with additional customization using gradients, rounded elements, and animation-friendly formatting.

Key Functionalities:

- Applies a smooth gradient background and modern font for improved readability.
- Styles the chat cards, message containers, and user avatars with rounded corners and semi-transparent overlays.
- Customizes input fields, send buttons, and search bars for a sleek, interactive look.
- Ensures responsiveness across screen sizes with media queries.
- Differentiates between user messages and chatbot responses using distinct colors (e.g., green for user, blue for bot).

Below is the complete source code of the style.css file:

```
body, html {
    height: 100%;
    margin: 0;
    background: #F67379; /* Light Coral */
    background: -webkit-linear-gradient(to right, #F67379, #F57278, #EEA3FE, #e2afed); /* Gradient with provided colors */
    background: linear-gradient(to right, #F67379, #F57278, #EEA3FE, #e2afed); /* Gradient with provided colors */
    font-family: 'Poppins', sans-serif;
}
.chat{
        margin-top: auto;
        margin-bottom: auto;
}
.card{
        height: 500px;
        border-radius: 15px !important;
        background-color: rgba(0,0,0,0.4) !important;
}
.contacts_body{
        padding:  0.75rem 0 !important;
        overflow-y: auto;
        white-space: nowrap;
}
.msg_card_body{
        overflow-y: auto;
}
.card-header{
        border-radius: 15px 15px 0 0 !important;
```

```
        border-bottom: 0 !important;
    }
    .card-footer{
    border-radius: 0 0 15px 15px !important;
        border-top: 0 !important;
    }
    .container{
        align-content: center;
    }
    .search{
        border-radius: 15px 0 0 15px !important;
        background-color: rgba(0,0,0,0.3) !important;
        border:0 !important;
        color:white !important;
    }
    .search:focus{
         box-shadow:none !important;
      outline:0px !important;
    }
    .type_msg{
        background-color: rgba(0,0,0,0.3) !important;
        border:0 !important;
        color:white !important;
        height: 60px !important;
        overflow-y: auto;
    }
        .type_msg:focus{
         box-shadow:none !important;
      outline:0px !important;
    }
    .attach_btn{
```

```
        border-radius: 15px 0 0 15px !important;
        background-color: rgba(0,0,0,0.3) !important;
        border:0 !important;
        color: white !important;
        cursor: pointer;
    }
    .send_btn{
        border-radius: 0 15px 15px 0 !important;
        background-color: rgba(0,0,0,0.3) !important;
        border:0 !important;
        color: white !important;
        cursor: pointer;
    }
    .search_btn{
        border-radius: 0 15px 15px 0 !important;
        background-color: rgba(0,0,0,0.3) !important;
        border:0 !important;
        color: white !important;
        cursor: pointer;
    }
    .contacts{
        list-style: none;
        padding: 0;
    }
    .contacts li{
        width: 100% !important;
        padding: 5px 10px;
        margin-bottom: 15px !important;
    }
    .active{
        background-color: rgba(0,0,0,0.3);
```

```
}
.user_img{
    height: 70px;
    width: 70px;
    border:1.5px solid #f5f6fa;
}
.user_img_msg{
    height: 40px;
    width: 40px;
    border:1.5px solid #f5f6fa;
}
.img_cont{
    position: relative;
    height: 70px;
    width: 70px;
}
.img_cont_msg{
    height: 40px;
    width: 40px;
}
.online_icon{
    position: absolute;
    height: 15px;
    width:15px;
    background-color: #4cd137;
    border-radius: 50%;
    bottom: 0.2em;
    right: 0.4em;
    border:1.5px solid white;
}
.offline{
```

```
        background-color: #c23616 !important;
    }
    .user_info{
        margin-top: auto;
        margin-bottom: auto;
        margin-left: 15px;
    }
    .user_info span{
        font-size: 20px;
        color: white;
    }
    .user_info p{
        font-size: 10px;
        color: rgba(255,255,255,0.6);
    }
    .video_cam{
        margin-left: 50px;
        margin-top: 5px;
    }
    .video_cam span{
        color: white;
        font-size: 20px;
        cursor: pointer;
        margin-right: 20px;
    }
    .msg_cotainer{
        margin-top: auto;
        margin-bottom: auto;
        margin-left: 10px;
        border-radius: 25px;
        background-color: rgb(82, 172, 255);
```

```
        padding: 10px;
        position: relative;
    }
    .msg_cotainer_send{
        margin-top: auto;
        margin-bottom: auto;
        margin-right: 10px;
        border-radius: 25px;
        background-color: #58cc71;
        padding: 10px;
        position: relative;
    }
    .msg_time{
        position: absolute;
        left: 0;
        bottom: -15px;
        color: rgba(255,255,255,0.5);
        font-size: 10px;
    }
    .msg_time_send{
        position: absolute;
        right:0;
        bottom: -15px;
        color: rgba(255,255,255,0.5);
        font-size: 10px;
    }
    .msg_head{
        position: relative;
    }
    #action_menu_btn{
        position: absolute;
```

```
        right: 10px;
        top: 10px;
        color: white;
        cursor: pointer;
        font-size: 20px;
    }
    .action_menu{
        z-index: 1;
        position: absolute;
        padding: 15px 0;
        background-color: rgba(0,0,0,0.5);
        color: white;
        border-radius: 15px;
        top: 30px;
        right: 15px;
        display: none;
    }
    .action_menu ul{
        list-style: none;
        padding: 0;
        margin: 0;
    }
    .action_menu ul li{
        width: 100%;
        padding: 10px 15px;
        margin-bottom: 5px;
    }
    .action_menu ul li i{
        padding-right: 10px;
    }
    .action_menu ul li:hover{
```

```
        cursor: pointer;
        background-color: rgba(0,0,0,0.2);
    }
    @media(max-width: 576px){
        .contacts_card{
        margin-bottom: 15px !important;
    }
    }
```

9. OUTPUT SCREENS

This section provides a comprehensive overview of the system’s visual output during runtime, highlighting key moments of interaction within the healthcare chatbot application. Each screenshot captures a critical state or user interaction that showcases the seamless integration of the frontend interface, backend processing, and AI-driven response engine. These visual outputs collectively validate the successful communication between all system components and illustrate the chatbot’s ability to understand, process, and respond to a wide range of medical queries in a coherent and user-friendly manner.

The screenshots demonstrate the complete functional flow—from initial user input to intelligent response generation—emphasizing the system’s responsiveness, accuracy, and natural language understanding capabilities. They also highlight error handling mechanisms, loading states, and AI response formatting, ensuring the chatbot not only delivers correct information but also maintains a smooth and intuitive user experience. Overall, these visual outputs serve as evidence of the system's effectiveness, reliability, and real-time performance in providing accessible and context-aware medical assistance.

➢ **Chatbot Home Interface**

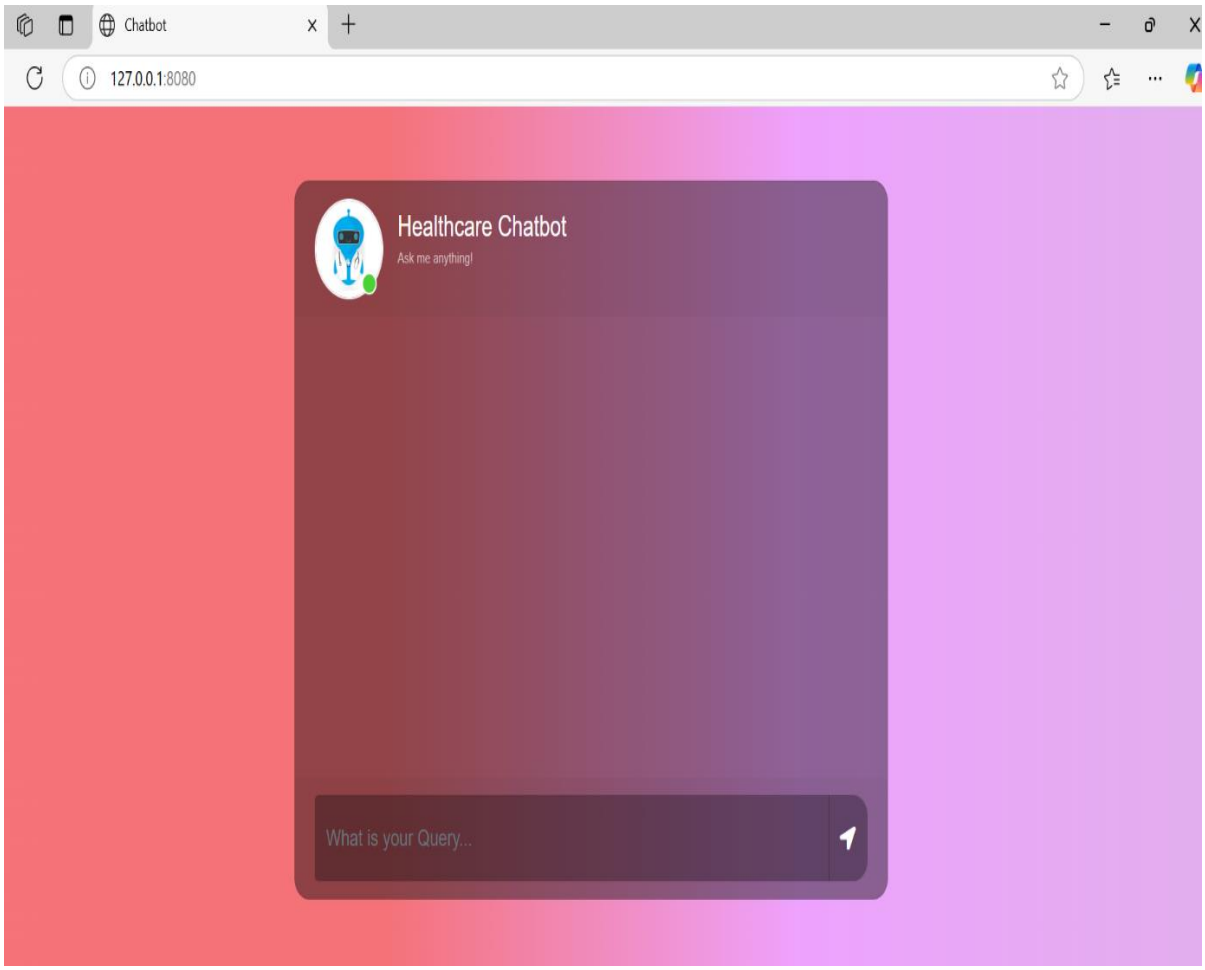

Fig 9.1 Chatbot Home Interface

- Description: This is the default screen that loads when the user visits the chatbot page. It features a clean, centered chat card with a medical avatar and welcome message.

- UI Elements:
  - Chat area with conversation bubbles
  - Input text box labeled "What is your Query..."
  - Send button with a paper-plane icon
  - Online indicator on the avatar

- Purpose: Creates a user-friendly and professional-looking interface that encourages interaction.

➢ **User Message Submission**

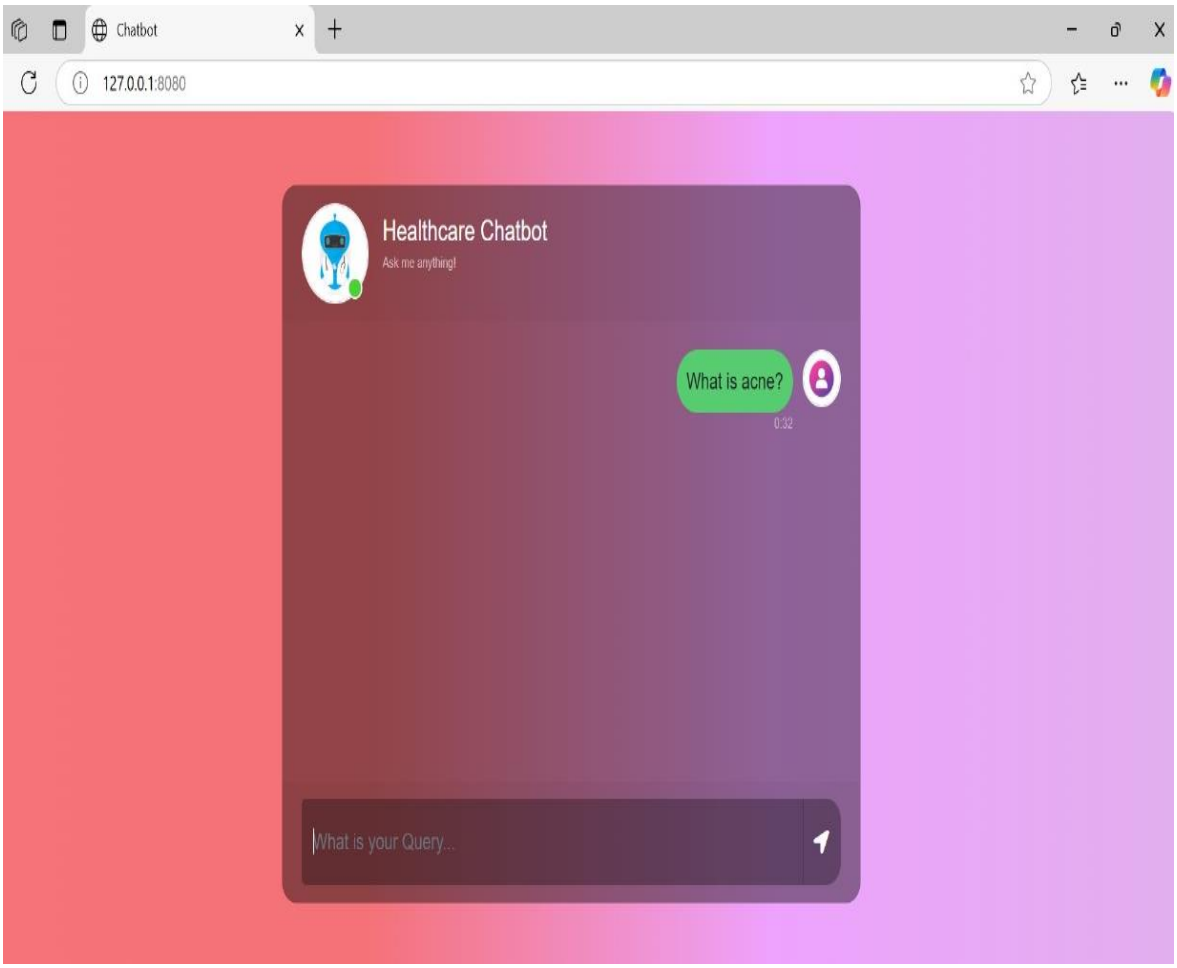

Fig 9.2 User Message Submission

- Description: After entering a medical query, the user clicks the send button. The message appears in the chat window aligned to the right with a green background to indicate it is from the user.

- Visual Details:
  - Green message bubble for user query
  - Timestamp displayed under the message
  - Avatar representing the user

- Example Input: "What is acne?"

➢ **AI-Generated Medical Response**

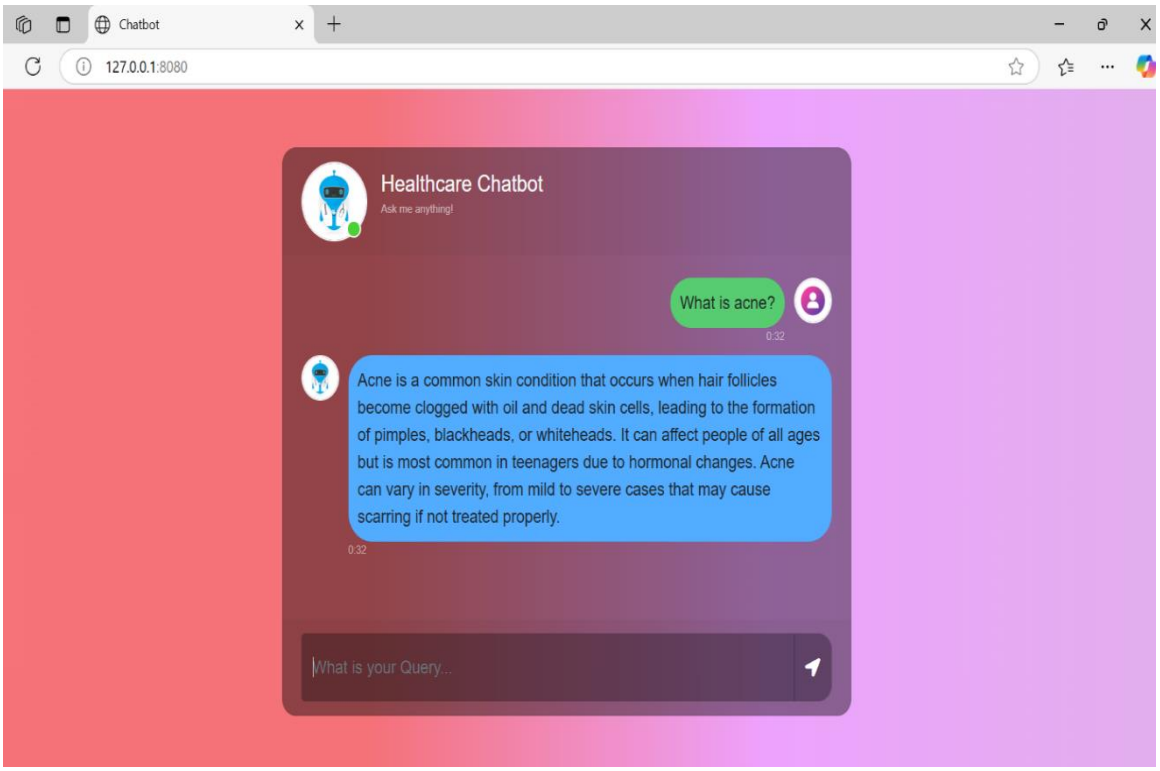

Fig 9.3 AI-Generated Medical Response

- Description: The system receives the user query, processes it through the LangChain + GPT-based RAG pipeline, and displays a relevant, context-aware response.

- Visual Details:
  - Bot message bubble in blue
  - Medical avatar on the left
  - Accurate, structured response with timestamp
  - Scrollable chat interface for longer conversations

- Example Output: " Acne is a common skin condition that occurs when hair follicles become clogged with oil and dead skin cells, leading to the formation of pimples, blackheads, or whiteheads. It can affect people of all ages but is most common in teenagers due to hormonal changes. Acne can vary in severity, from mild to severe cases that may cause scarring if not treated properly."

- Significance: Demonstrates the chatbot's ability to return medically sound, AI-generated responses grounded in real medical documents.

➢ **Non-Medical Query Handling**

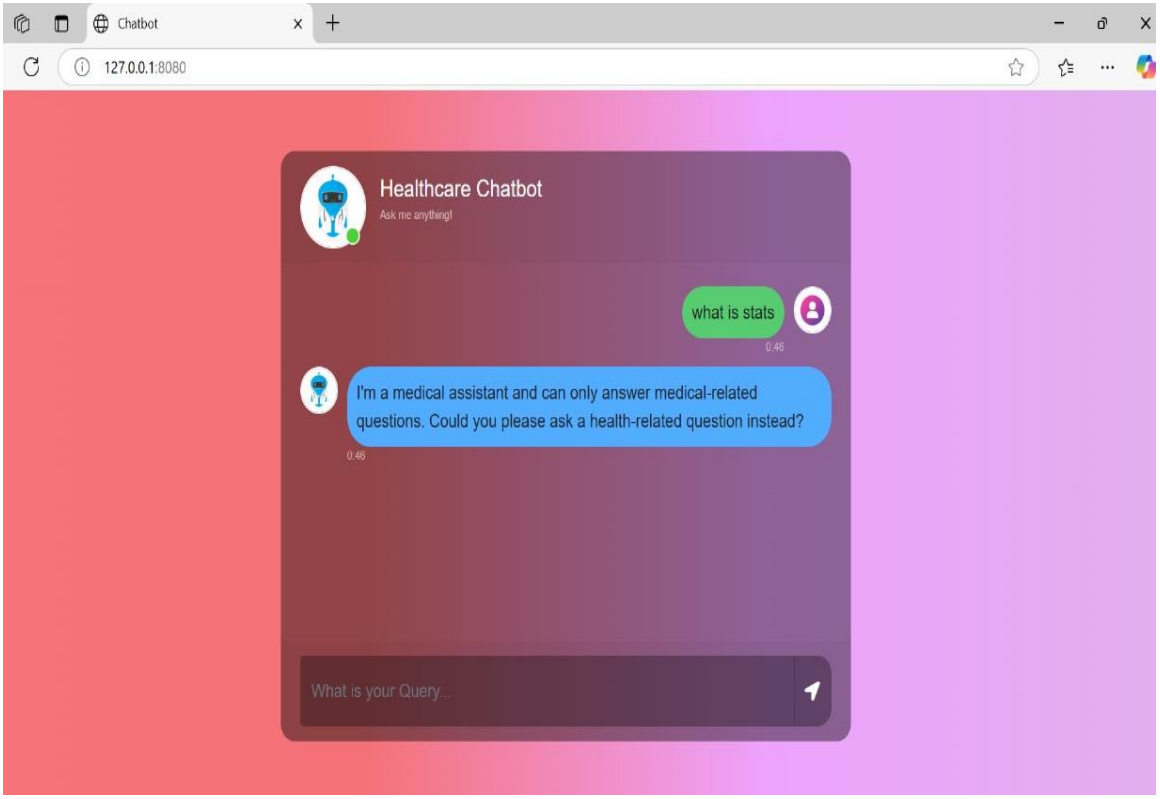

Fig 9.4 Non-Medical Query Handling

- Description: When a user inputs a non-health-related question, the system performs a classification check and responds accordingly.

- Visual Details:
  - Bot returns a filtered message
  - Message bubble in blue with a polite error notification

- Example Output: "I'm a medical assistant and can only answer medical-related questions. Could you please ask a health-related question instead?"

- Purpose: Maintains the integrity of the chatbot by staying focused on verified medical information only.

➢ **Backend Console (Developer View)**

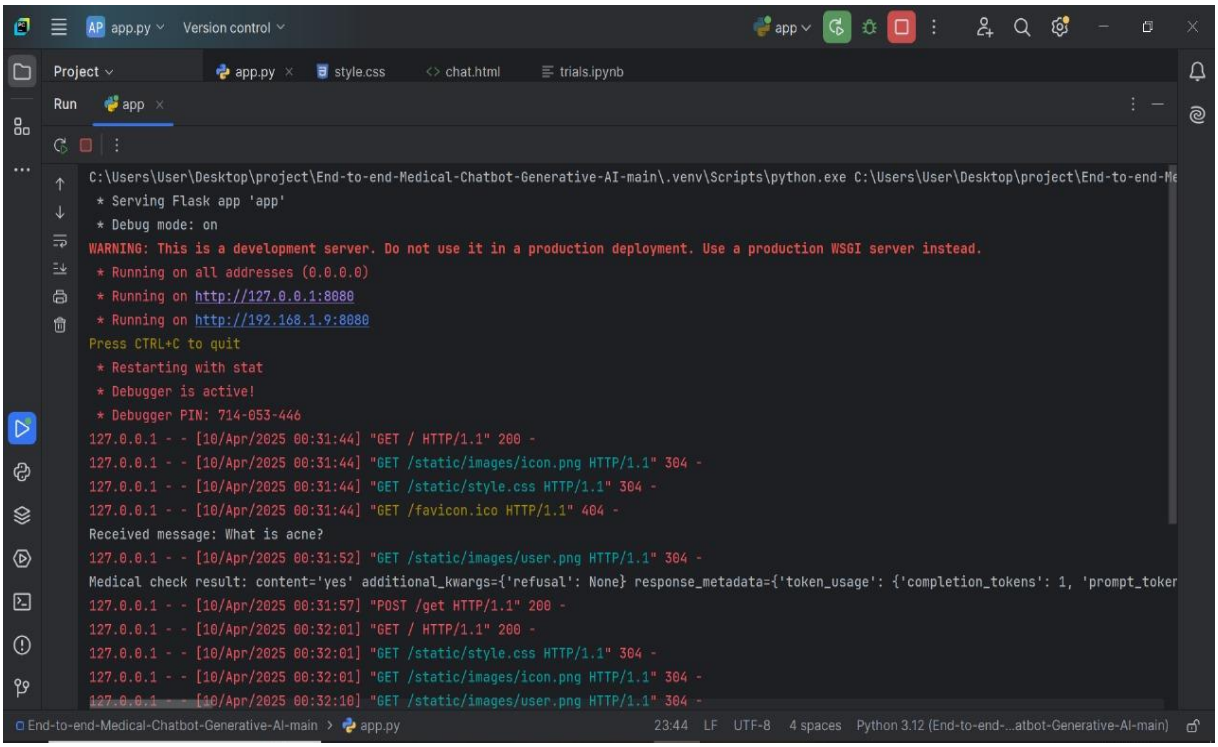

Fig 9.5 Backend Console

- Description: Shows the backend logs in the development console.

- Example Log Entries:
  - Received message from user
  - Medical check result: yes
  - Pinecone retrieval success
  - GPT response generated successfully

- Visual Details:
  - Chatbot message is shown in a blue message bubble.
  - The bot avatar appears on the left, maintaining conversation flow.
  - The message contains a friendly and clear explanation.
  - A timestamp is shown below the message, similar to standard responses.

- **Medical PDF Insertion**

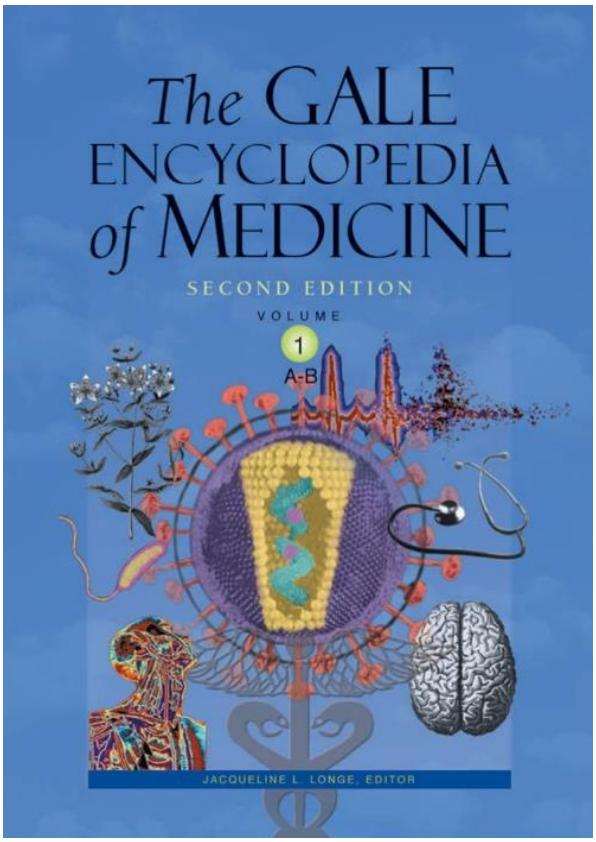

Fig 9.6 Medical PDF Insertion

- Description: In the current implementation, new medical PDFs are added manually through the development environment (Visual Studio Code) rather than a user-facing admin upload interface. Developers place the PDFs into a designated directory within the project before processing them through the embedding pipeline.

- Significance:
  - Allows for real-time updates and scalability of the knowledge base.
  - Allows for scalable expansion of the knowledge base by simply placing additional medical documents into the codebase.
  - Simplifies control over content accuracy and quality, as only trusted documents are manually reviewed and inserted.

10. CONCLUSION

The healthcare chatbot using Generative AI successfully demonstrates how advanced natural language processing can assist users in obtaining relevant and accurate medical information. By integrating technologies such as OpenAI, LangChain, and Pinecone, the system is capable of understanding complex medical queries, retrieving contextual information, and generating intelligent responses in real time.

This project showcases the seamless collaboration between frontend design, backend logic, and AI capabilities. The modular architecture ensures flexibility and scalability, while the intuitive interface enhances user interaction.

Overall, the project proves to be a reliable and efficient tool for basic medical query assistance, and lays the foundation for future enhancements in healthcare AI solutions.

11. FUTURE ENHANCEMENT

To further improve the functionality and usability of the healthcare chatbot, the following enhancements can be considered in future versions:

- Voice Input Integration: Allow users to speak their queries for hands-free interaction using speech-to-text technology.
- Multilingual Support: Enable the chatbot to understand and respond in multiple languages to cater to a wider audience.
- Medical Image Analysis: Integrate basic medical image recognition (like X-rays or skin conditions) for more comprehensive diagnostics.
- Doctor Connect Feature: Provide an option to connect users with real medical professionals based on their queries.
- User Authentication and History Tracking: Implement login systems and store chat history for returning users to maintain personalized experiences.
- Mobile Application Deployment: Develop Android/iOS versions for broader accessibility.
- Feedback and Learning Loop: Allow users to rate responses and use that data to fine-tune the chatbot's accuracy and relevance.

These future enhancements will make the chatbot more intelligent, user-friendly, and capable of providing broader healthcare support.

12. REFERENCES

- AI Inquiry Bot in Scientific Education – Asia-Pacific Science Education (2023) https://doi.org/10.1163/23641177-bja10062
- Healthcare Chatbot using Artificial Intelligence – IJRASET https://www.ijraset.com/research-paper/healthcare-chatbot-using-artificial-intelligence
- AI Conversational Chatbot for Primary Healthcare Diagnosis – ResearchGate https://www.researchgate.net/publication/357162231
- AI Based Healthcare Chatbot using NLP & Pattern Matching – IJNRD (2023) https://www.ijnrd.org/papers/IJNRD2305598.pdf

Tool/Library References (for Tech Stack)

- Python: https://www.python.org/
- NLTK: https://www.nltk.org/
- Scikit-learn: https://scikit-learn.org/stable/
- Pandas: https://pandas.pydata.org/
- Chatterbot (basic NLP): https://chatterbot.readthedocs.io/en/stable/
- RASA (Advanced NLP chatbot framework): https://rasa.com/

Textbooks

1. HTML & CSS: The Complete Reference, fifth Edition
2. Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming
3. Flask Web Development, 2nd Edition
4. Fundamentals of Python First Programs, Kenneth. A. Lambert, Cengage.
5. Python Programming: A Modern Approach, Vamsi Kurama, Pearson.