



**G L BAJAJ COLLEGE OF TECHNOLOGY AND MANAGEMENT,
GREATER NOIDA**

**A Project Report
On**

AI AGENT POWERD CHATBOT

Submitted in partial fulfillment of the requirement for the award of the degree of
Master of Computer Applications

By

RAJNISH VERMA

Roll No. – 2312000140142

**Under the Supervision
of**

SUGANDHI MAAM

(Assistant Professor)



**DR. A P J ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW
(2024-25)**

PROJECT REPORT ON
(AI AGENT POWERD CHATBOT)

(KCA-451)
Session-2024-2025

Department of Master of Computer Applications (MCA)



Submitted to:

Sugandhi maam
(Assistant Professor)

Submitted by:

RAJNISH VERMA
Roll no:2312000140142
Semester:4TH
Section:B3

G L Bajaj College of Technology & Management
Plot No 2, APJ Abdul Kalam Rd, Knowledge Park III,
Greater Noida, Uttar Pradesh

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to everyone who supported and guided me throughout the course of this project.

First and foremost, I would like to thank Sugandhi maam, whose constant encouragement, expert guidance, and valuable feedback played a crucial role in the successful completion of this project. Your insights into AI technologies and software development were incredibly helpful and inspiring.

I am also grateful to the faculty and staff of G L Bajaj College of Technology & Management for providing a supportive academic environment, as well as access to the resources and tools necessary for the research and implementation of this work.

Special thanks to my classmates and friends for their constructive discussions, testing assistance, and moral support during the development phase.

Finally, I would like to thank my family for their unwavering encouragement and support throughout my academic journey.

Rajnish Verma

(2312000140142)

CERTIFICATE OF ORIGINALITY

I hereby declare that my Project titled “**AI AGENT POWERD CHATBOT**“ submitted to **Dr. APJ ABDUL KALAM TECHNICAL UNIVERSITY, Lucknow** for the partial fulfillment of the degree of Master of Computer Applications Session 2024-2025 from **G L Bajaj College of Technology and Management, Greater Noida** has not previously formed the basis for the award of any other degree, diploma or other title.

Place:

Signature

Rajniash Verma

Date:

CERTIFICATE OF ACCEPTANCE

This is to certify that the project entitled, “ **AI AGENT POWERD CHATBOT**” submitted by **Rajnish Verma** a bonafide student of **GL Bajaj College of Technology and Management, Greater Noida** in partial fulfillment for the award of **Master of Computer Applications** affiliated to **Dr. APJ ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW** during the year 2024-25. It is certified that all corrections, suggestions indicated as per Internal Assessment have been incorporate in the project.

To the best of our knowledge, the work embodied in this report is original and has not been submitted to any other degree of discipline. The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said degree.

[Sign and Name of Internal Guide]

[Sign of External Examiner]

Head of Department
Master of Computer Applications
G L Bajaj College of Technology and Management, Greater Noida

LIST OF FIGURES

DFD Level-0	21
DFD Level-1	23
ER-Diagram	27
Implementation of code ai_agent	37
Model output	40
Implementation of code backend	41
Backend output	44
Implementation of code frontend	45
Frontend view.	49

LIST OF TABLES

Table 1.1 Related Work Overview	14
Table 1.2 Hardware Requirements	16
Tbale 1.3 Software Stack Table	19
Table 1.4 Data Flow Summary	26
Table 1.5 Relationships Summary.	29
Table 1.6 API Schema Design	34
Table1.7 Key Functional Achievements	50
Table 1.8 Future Scope	53

TABLE OF CONTENTS

ACKNOWLEDGEMENT	3
CERTIFICATE OF ORIGINALITY	4
CERTIFICATE OF ACCEPTANCE.....	5
LIST OF FIGURES	6
LIST OF TABLES	7
Chapter 1.....	9
Chapter 2.....	12
Background Study and Research Gap	12
Chapter 3.....	15
Chapter 4.....	31
Proposed work and Methodology	31
Chapter 5.....	33
Design/Development	33
Chapter 6.....	37
Testing and Implementation	37
6.1 AI_agent Implementation 6.1.1 Importing Modules and Loading Environment Variables	38
Chapter 7.....	50
Chapter 8.....	52
REFERENCES.....	54

Chapter 1

Introduction and Aim of the Project

1.1 Introduction

This project aims to build an AI-powered chatbot that interacts with users in a context-aware and intelligent manner. The chatbot will leverage the capabilities of Large Language Models (LLMs) such as Groq and OpenAI, combined with LangGraph ReAct Agents to improve decision-making and conversation flow.

In the rapidly evolving field of Artificial Intelligence, conversational agents—commonly known as chatbots—have become vital tools across industries, enhancing customer service, education, healthcare, and more. However, traditional rule-based and retrieval-based chatbots often fall short when it comes to maintaining context, adapting to dynamic queries, or providing intelligent reasoning.

With the advent of **Large Language Models (LLMs)** such as **OpenAI's GPT** or **Groq**, it is now possible to build chatbots that can simulate human-like conversation, comprehend context, and respond with insightful, accurate answers. However, even LLMs require structured frameworks to efficiently manage reasoning and integrate external tools and APIs.

This project proposes the development of an **AI Agent-powered chatbot** that overcomes traditional chatbot limitations by integrating cutting-edge tools like **LangGraph ReAct Agents**,

FastAPI, **LangChain**, and Streamlit, making the system highly responsive, intelligent, and user-friendly.

1.2 Project Description: AI Agent-Powered Chatbot System

This project focuses on developing a smart, conversational AI chatbot system using cutting-edge technologies like **LangGraph ReAct Agents**, **LLMs (OpenAI/Groq)**, **FastAPI**, and **Streamlit**. The chatbot functions as an interactive agent capable of handling user queries, executing search tasks, and offering informative and contextual responses.

The system accepts user inputs via a clean and responsive frontend UI, where users can select the LLM provider (OpenAI or Groq), choose from available models, define a custom system prompt, and optionally allow real-time web search via **Tavily API**. These inputs are sent to the backend, where the FastAPI server dynamically constructs an agent using the LangGraph framework.

Based on the selected model and settings, the agent processes the conversation by building an internal state, optionally retrieving external data, and invoking the LLM to generate a response. This response is then formatted and sent back to the frontend interface for user interaction.

Overall, the project demonstrates the integration of AI agents with modern full-stack technologies to create intelligent, dynamic, and context-aware chatbot systems—suitable for applications in education, customer service, and productivity.

1.3 Aim of the Project

- The main aim of this project is to design and develop a context-aware, intelligent chatbot system that can:
- Leverage **LLMs (OpenAI/Groq)** for generating natural language responses.
- Utilize **LangGraph ReAct Agents** to enhance reasoning, decision-making, and tool usage within conversations.
- Provide real-time communication through a **FastAPI**-based backend and a **Streamlit**-based frontend interface.
- Support API and tool integration using **LangChain**, enabling the chatbot to handle dynamic tasks and access external data.
- By combining the strengths of agent-based reasoning and LLMs, the chatbot aims to deliver smart, flexible, and scalable conversational experiences that can adapt to various real-world applications.

Chapter 2

Background Study and Research Gap

2.1 Background Study

Conversational AI has evolved significantly over the past decade, transitioning from basic rule-based systems to sophisticated neural network-based language models. Traditional chatbots, while useful, are often rigid and lack the ability to understand context, make decisions, or adapt to new situations. [1]

The emergence of **Large Language Models (LLMs)** such as OpenAI's GPT series and Groq's ultra-fast transformer models has revolutionized natural language processing. These models are capable of understanding complex language patterns, maintaining context over longer conversations, and generating human-like responses. However, LLMs on their own do not inherently manage task logic, tool usage, or structured workflows. [2]

To overcome this, frameworks such as **LangChain**, **ChainForge**, and **LangGraph** have emerged. These allow developers to create **LLM-powered agents** capable of reasoning, calling tools/APIs, & navigating complex workflows. For example, LangGraph ReAct Agents utilize a "Reasoning and Acting" loop where the agent analyzes a question, decides whether to invoke a tool, and refines its response accordingly. [3]

Projects like **ChainBuddy** (Zhang & Arawjo, 2024) and systems discussed in Auffarth's Generative AI with LangChain have demonstrated the power of such multi-agent frameworks in real-world applications, including enterprise chatbots and intelligent assistants. [4]

These advancements form the foundation for this project, which aims to build an intelligent, responsive, and modular chatbot using these state-of-the-art technologies.

2.2 Research Gap

Despite significant advancements in conversational AI, several gaps remain in current chatbot implementations:

- **Context Retention:**

Many chatbots struggle to maintain context across longer conversations, especially without memory management or session tracking.

- **Dynamic Task Execution:**

Traditional models are not capable of invoking tools or external APIs effectively. Most rely solely on pre-trained knowledge and cannot dynamically fetch or compute data.

- **Reasoning Capabilities:**

While LLMs can simulate reasoning, they often need structured agent frameworks like LangGraph to make deliberate, explainable decisions and tool calls.

- **Tool Integration Complexity:**

Integrating external tools or APIs with LLMs requires significant engineering, often leading to fragmented or inflexible solutions.

- **Real-Time Responsiveness:**

1. LLMs can be computationally intensive, which affects performance and scalability in real time applications.
2. This project addresses these gaps by:
3. Using LangGraph ReAct Agents for structured reasoning and tool integration.
4. Implementing FastAPI for efficient backend handling.
5. Deploying a Streamlit UI for an interactive, real-time experience.
6. Using LangChain to enable tool/API calls within the conversation loop.
7. Leveraging Groq/OpenAI LLMs for natural and accurate language understanding.

2.3 Related Work Overview

Table 1.1 Related Work Overview

S.No.	Title / Area	Author(s)	Year	Scope / Focus
1	ChainBuddy: An AI Agent System for Generating LLM Pipelines	Zhang, J., & Arawjo, I.	2024	Proposes an AI agent system to automate and generate complex LLM pipelines
2	Agent-Based Chatbot in Corporate Environment (Master’s Thesis)	Dybedokken, O., & Nilsen, H.	2024	Explores how agent-based chatbots improve enterprise communication & automation
3	Generative AI with LangChain	Auffarth, B.	2023	A practical guide on building LLM-based applications using LangChain and Python
4	Large Language Model and AI-Based Human Conversation Agent	Farooq, M. O., Aziz, U., & Ullah, M. Z.	2024	Research on designing AI conversation agents using LLMs for natural interactions
5	ChainBuddy (Duplicate)	Zhang, J., & Arawjo, I.	2024	Duplicate of Entry 1 – AI LLM pipeline agent system
6	LangChain, FastAPI, Streamlit, OpenAI, Groq, Uvicorn (Documentation)	Official Contributors	-	Technical documentation for libraries used in LLM-based AI agent development

Chapter 3

Tools/Platform, Hardware and Software Requirement

Specification (DFD, ER/UML)

3.1 Hardware Requirements

Developing and deploying an AI Agent Chatbot that integrates advanced LLMs (Large Language Models) and real-time web search capabilities requires a decent hardware setup for efficient performance. The following hardware configuration ensures smooth development, testing, and interaction with APIs and external tools:

3.1.1 Intel Core i5/i7 or Equivalent Processor

A multi-core processor such as Intel Core i5 or i7 (or AMD Ryzen 5/7) is essential to run backend servers, handle concurrent requests, and manage development environments efficiently. These processors provide enough computational power to support real-time API calls, code compilation, and Streamlit frontend rendering.

3.1.2 8GB+ RAM

A minimum of 8GB RAM is required to run all essential processes simultaneously — including the FastAPI server, Streamlit UI, IDE (e.g., VS Code), and local testing of the chatbot. However, 16GB RAM or more is recommended for better multitasking, especially when working with heavier Python libraries like langchain, openai, and transformers.

3.1.3 256GB SSD

A solid-state drive (SSD) ensures faster read/write speeds for loading development tools, storing virtual environments, and caching temporary data. A 256GB SSD provides enough space for all dependencies, model files, logs, and project files.

3.1.4 GPU (Optional but Recommended)

While the project primarily relies on external APIs for LLM inference (like Groq or OpenAI), having a GPU can be beneficial during local fine-tuning, testing smaller models, or exploring offline inference using open-source LLMs. A CUDA-enabled NVIDIA GPU would help speed up local model execution if needed.

Table 1.2 Hardware Requirements

Component	Specification	Purpose
Processor (CPU)	Intel Core i5/i7 or AMD Ryzen 5/7	Ensures fast and efficient processing for backend servers and development.
RAM	Minimum 8GB (16GB Recommended)	Supports smooth multitasking during API hosting, frontend rendering, etc.
Storage	256GB SSD	Fast read/write speeds, stores code, virtual environments, dependencies.
GPU (Optional)	NVIDIA GPU (Preferred)	Enhances performance if using local LLM inference or model fine-tuning.

3.2 Software Stack

This AI chatbot project utilizes a modern and efficient software stack, combining full-stack development tools, API frameworks, and agent orchestration systems.

3.2.1 Python (Programming Language)

Python serves as the primary language due to its strong ecosystem for machine learning, API development, and web frameworks. It supports key libraries like fastapi, langchain, and streamlit used in this project.

3.2.2 FastAPI (Backend Framework)

FastAPI is used to build the REST API backend. It enables high-performance asynchronous request handling and supports auto-generated documentation (Swagger UI). It processes requests from the frontend, constructs the AI agent, and returns model responses.

- Fast and lightweight
- Supports Pydantic for data validation
- Ideal for microservices architecture

3.2.3 Streamlit (Frontend/UI)

Streamlit is a Python-based tool used to build interactive UIs quickly. It provides a simple way to gather user input, display chatbot results, and toggle features like model selection and web search.

- Easy UI creation with minimal code
- Integrates seamlessly with Python backend
- Real-time display of responses

3.2.4 Groq / OpenAI (LLM Providers)

These are external providers of powerful Large Language Models. OpenAI offers models like GPT-4o-mini, while Groq provides blazing-fast inference for models like LLaMA 3 and Mixtral.

- Used via API calls

- Offers real-time intelligent responses
- No local model hosting required

3.2.5 Uvicorn (ASGI Server)

Uvicorn is a lightning-fast ASGI server used to host the FastAPI app. It handles HTTP requests and ensures the API endpoints are accessible to the Streamlit frontend.

- Asynchronous support
- Lightweight and efficient
- Runs on localhost or production servers

3.2.6 LangGraph ReAct (Agent Framework)

LangGraph is a powerful orchestration library used to build ReAct-style agents. It allows the chatbot to reason and act by interacting with tools like web search or databases during conversations.

- State-based reasoning
- Supports LangChain tools
- Modular and composable architecture

3.2.7 LangChain (Tool/Chain Integration)

LangChain enables easy integration of external tools (e.g., search APIs) with LLMs. It helps the agent use tools intelligently during conversation flow using the ReAct (Reason + Act) pattern.

- Tool calling and chaining
- Prompt templates and memory
- Integrates with LangGraph

3.2.8 Visual Studio Code (Development IDE)

VS Code is the preferred integrated development environment used for writing, testing, and debugging the Python code. It supports extensions for Python, Git, Docker, and REST clients.

- Lightweight and extensible
- Terminal and linting support
- GitHub integration

Tbale 1.3 Software Stack Table

Tool / Technology	Category	Description / Role
Python	Programming Language	Base language for coding backend, agent logic, and frontend scripts.
FastAPI	Backend Framework	Builds REST API endpoints to serve responses from the chatbot agent.
Streamlit	Frontend/UI Framework	Creates a simple and interactive web-based user interface for chatting with the agent.
Groq / OpenAI	LLM Providers	Provides access to powerful cloud-hosted language models like GPT-4o, LLaMA3, etc.
Uvicorn	ASGI Server	Lightweight async server for running FastAPI services.
LangGraph ReAct	Agent Architecture	Enables reasoning agents using tools like web search, step-by-step thought processing.
Langchain	Tool/Chain Integration	Facilitates tool invocation, memory, prompt chaining, and modular conversation flows.
VS Code	Development IDE	Full-featured coding environment with terminal, Git, and Python support.

Summary of Use

- **Python, FastAPI, and Langchain** drive the backend logic and agent orchestration.
- **Streamlit** handles the front-end user interface for easy interaction with the bot.
- **Groq/OpenAI APIs** are used for high-speed and intelligent LLM-based responses.
- **Uvicorn** serves the backend, making endpoints accessible to the frontend.
- **LangGraph ReAct** gives the bot reasoning capabilities through tool integration and memory.
- **VS Code** provides a developer-friendly IDE to manage the entire codebase.

3.3 DFD LEVEL 0

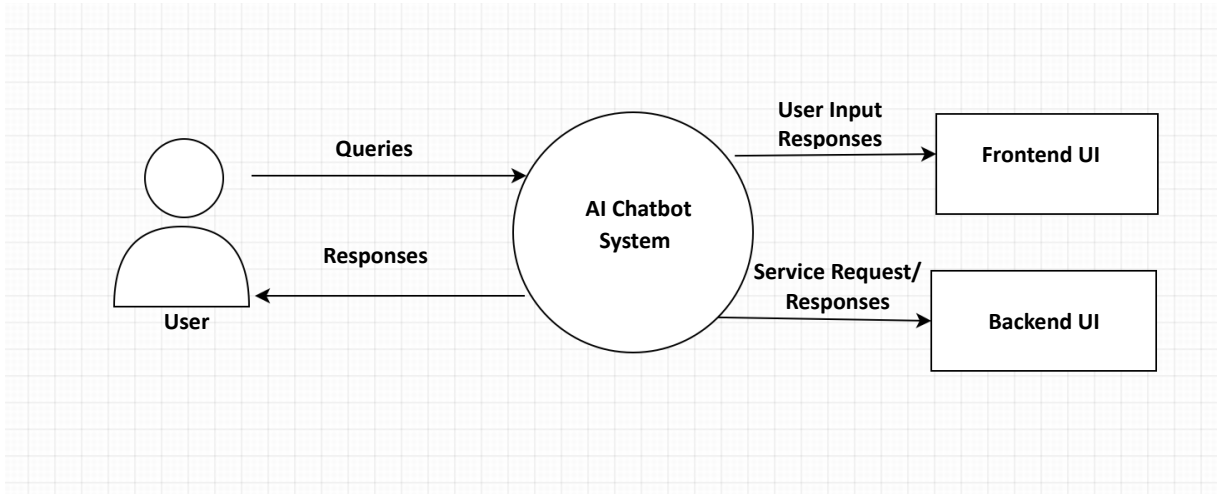


fig.1.1 Data Flow Diagram

Explanation of the DFD Components

1. External Entity: User

Role: Initiates communication with the chatbot system.

Action:

Sends Queries to the AI Chatbot.

Receives Responses from the system.

2. Central Process: AI Chatbot System

Core of the system which processes user input and routes it appropriately.

It acts as an intermediary between the frontend UI and backend services.

3. Frontend UI

Connected via: “User Input/Responses” arrow.

Function:

Collects user inputs like queries, prompts, selected models, etc.

Displays responses returned by the chatbot (through APIs).

Tools Involved: Streamlit (in your project).

4. Backend UI / Service Layer

Connected via: “Service Request/Responses” arrow.

Function:

Handles the actual AI logic, model inference, search integration, etc.

Processes data sent from the frontend and generates appropriate replies.

Tools Involved: FastAPI, LangGraph, Groq/OpenAI models.

Flow of Data

User → AI Chatbot System: User sends a query.

AI Chatbot System → Frontend UI: User input is routed to the UI for validation and display.

AI Chatbot System → Backend UI: The request is sent to the backend for processing via APIs.

Backend UI → AI Chatbot System: Returns the model’s response.

AI Chatbot System → User: Delivers the final response back to the user via frontend.

Summary

The DFD shows a simple interaction between user, frontend, and backend.

It separates concerns clearly:

Frontend handles UI/UX.

Backend handles logic and AI processing.

It reflects a **modular design**, making the system scalable and maintainable.

3.4 DFD LEVEL 1

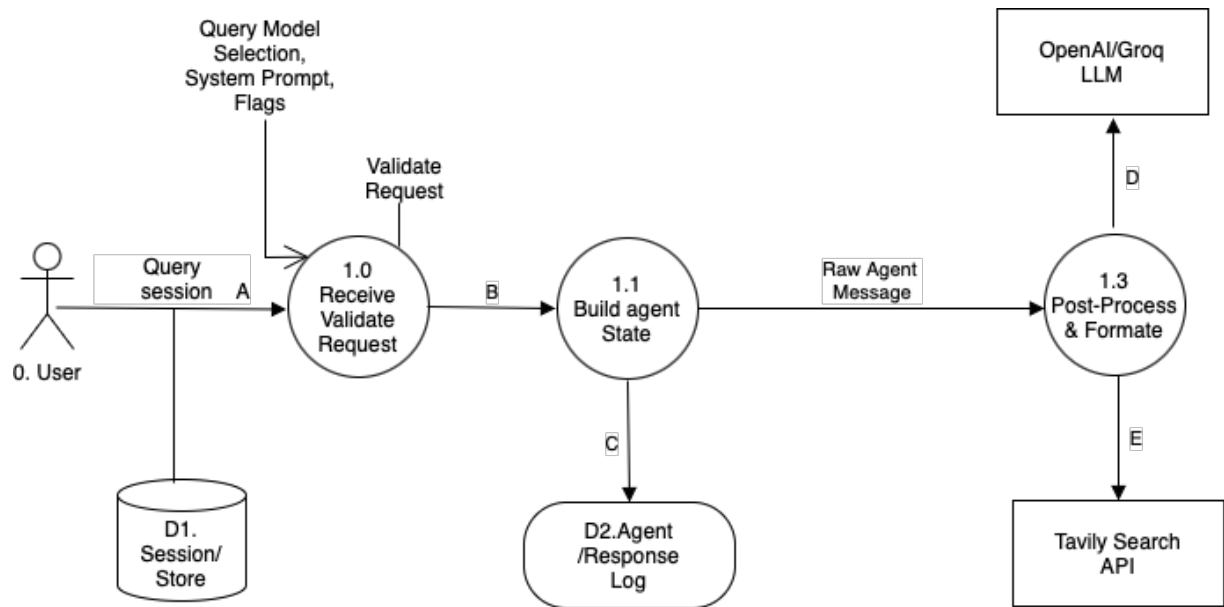


fig.1.2 AI Agent Chatbot Model

External Entity: User

Input: Sends a **query/session request**.

Goal: To receive a meaningful response from the chatbot.

Process Blocks

1.0 – Receive & Validate Request

Purpose: Parses the incoming request, checks:

If the model selected is valid

If the input (query, system prompt, etc.) is complete and safe

Interaction with D1: May retrieve past session data to support continuity.

1.1 – Build Agent State

Purpose: Prepares the state for the LangGraph ReAct agent using:

User's message

Selected model and provider

System prompt

Tools used: This step defines how the agent will reason or act.

1.2 – Run Agent (LLM + Tools)

Purpose: Executes the actual logic:

The agent can invoke the **OpenAI or Groq LLM**

If enabled, it may use the **Tavily Search API** for real-time information

Input: Structured agent state

Output: Raw response from the agent

Logs: This raw response is logged to D2.Agent/Response Log

1.3 – Post-process & Format

Purpose: Cleans, formats, and finalizes the response to return to the user

Handles Markdown formatting

Handles display-ready structuring (like bolding, bulleting, etc.)

Result: A user-friendly message ready to show on the frontend

Data Stores

D1 – Session/Store

Purpose: Stores user session history and contextual data (e.g., previous questions).

Helps in: Maintaining memory or continuous interaction feel.

D2 – Agent/Response Log

Purpose: Stores each interaction:

For auditing

For analytics or learning purposes (e.g., improving future agent behavior)

External Systems

OpenAI/Groq LLM

These are the Large Language Models that generate the core response based on the prompt and context.

Tavily Search API

This external API is called **only if web search is allowed**, to fetch real-time information.

Table 1.4 Data Flow Summary

ID	Description
A	User input/query
B	Validated data passed to build agent
C	Agent state prepared
D	Interaction with LLMs and Tavily API
E	Final formatted output to the user

Conclusion

This Level 1 DFD shows the **step-by-step internal workflow** of your chatbot system. It highlights:

Proper modular structure (validation, state creation, execution, formatting)

Logging and session support

Usage of external LLMs and optional web search tools

Structured design with future extensibility

3.4 ER Diagram

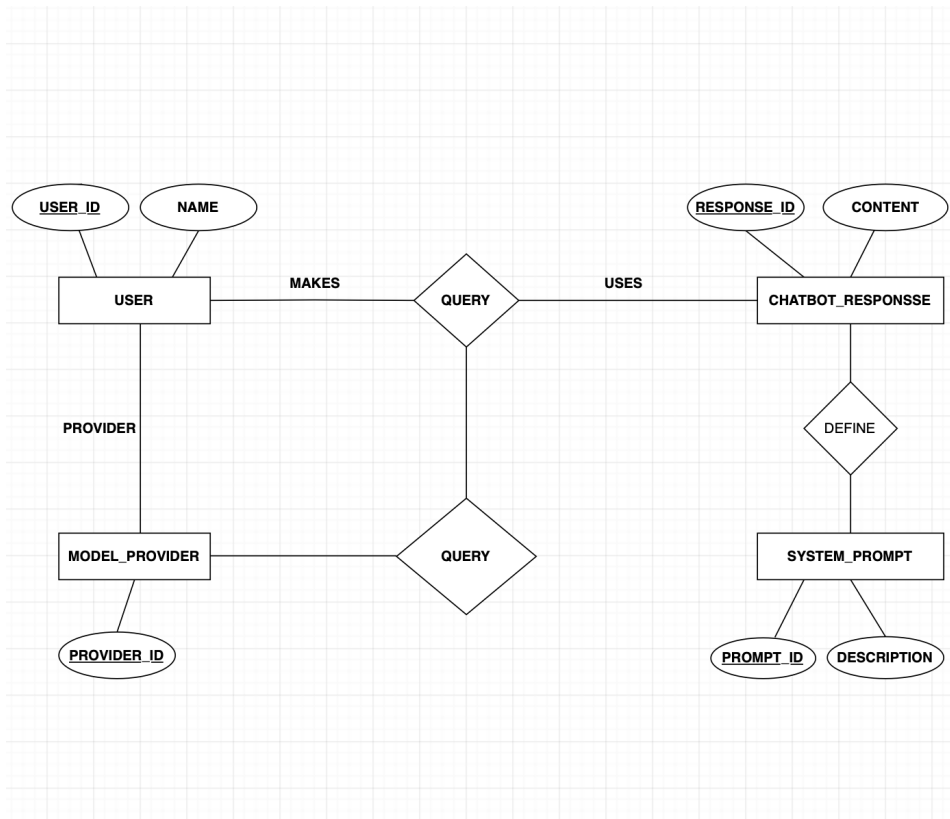


fig1.3 ER Diagram

Explanation of Components

Entity: USER

Attributes:

USER_ID (Primary Key): Uniquely identifies a user.

NAME: Stores the name of the user.

Relationship:

A user makes queries.

A user is linked to a model provider (could indicate user preference or usage source).

Entity: MODEL_PROVIDER

Attributes:

PROVIDER_ID (Primary Key): Unique identifier for each provider (e.g., Groq, OpenAI).

Relationship:

Provides models that users access when making queries.

Relationship: QUERY

Associations:

MAKES: A user initiates a query.

USES: A query is handled using a chatbot model and **yields a response**.

A query is also associated with a **model provider** (indirectly through the user).

Purpose: Central action that ties users, model providers, and chatbot responses.

Entity: CHATBOT_RESPONSE

Attributes:

RESPONSE_ID (Primary Key): Unique identifier for a chatbot's output.

CONTENT: The actual response text from the AI agent.

Relationship:

Used in response to a query.

Defined by a system prompt (through the DEFINE relationship).

Entity: SYSTEM_PROMPT

Attributes:

PROMPT_ID (Primary Key): Unique identifier for each prompt.

DESCRIPTION: The instructional or behavioral guidance provided to the AI.

Relationship:

Each response is generated based on a defined prompt.

Relationships Summary

Table 1.5 Relationships Summary

Relationship Name	Involves	Meaning
MAKES	USER – QUERY	A user can make one or more queries
PROVIDER	USER – MODEL_PROVIDER	Each user is linked to a model provider
USES	QUERY – CHATBOT_RESPONSE	Each query results in a chatbot response
DEFINE	CHATBOT_RESPONSE – SYSTEM_PROMPT	Each response is generated using a specific system prompt

Design Justification

This ER diagram provides a normalized schema for building a backend database that:

Tracks **who made a query** and **what the AI responded**

Logs **which model provider** was used

Captures **system prompts** used for generating those responses

Allows future scalability for audit, analytics, personalization, and prompt engineering.

Chapter 4

Proposed work and Methodology

The proposed system aims to build a robust, intelligent, and context-aware chatbot that can:

- Understand and process natural language queries using state-of-the-art **Large Language Models (LLMs)** like OpenAI or Groq.
- Utilize **LangGraph ReAct Agents** for reasoning, decision-making, and structured responses.
- Provide a real-time conversational interface using **FastAPI** for backend processing and **Streamlit** for frontend interaction.
- Enable **tool invocation and knowledge retrieval** using LangChain to enhance the chatbot's capability beyond static LLM answers.
- Ensure modularity, scalability, and real-time performance for practical deployment.

4.1 Methodology Flow

The development methodology for this project follows a structured and modular approach comprising the following stages:

4.1.1. Requirement Analysis

Identify the core functionalities needed: user query input, LLM integration, agent flow management, response generation, and frontend interface. Determine suitable technologies and frameworks (OpenAI/Groq, LangGraph, FastAPI, Streamlit, LangChain).

4.1.2 . System Architecture Design

1. Design an architecture that includes:
2. LLM Interface
3. LangGraph ReAct Agent manager
4. FastAPI backend
5. Streamlit frontend
6. Tool connectors via LangChain

4.1.3 . Implementation

- **Backend Development:**

Develop FastAPI endpoints to receive and process user queries.

Integrate OpenAI/Groq LLMs via API calls for generating responses.

Agent Integration:

Implement LangGraph ReAct Agent flow to handle query parsing, reasoning, and tool usage.

- **Frontend Development:**

Build an interactive UI in Streamlit to accept queries and display responses in real-time.

- **Tool Integration:**

Use LangChain for integrating APIs, computational tools, and custom logic.

Testing and Evaluation

- Perform functionality testing for:
 - Input validation
 - LLM response accuracy
 - Agent flow logic
 - User interface performance
- Conduct performance testing under different loads and types of queries.

Optimization:

- Refine prompt engineering and agent logic for better context awareness.
- Tune LLM parameters (temperature, max tokens) for optimal output.

Documentation and Deployment:

- Prepare complete documentation for users and developers.
- (Optional) Deploy the system on a cloud platform with Uvicorn.

This structured methodology ensures that the chatbot system is not only functional and efficient but also scalable, maintainable, and adaptable for future extensions like memory, multimodal input, and multilingual support

Chapter 5

Design/Development

This chapter details the architectural design, agent graph structure, API schema, LLM prompting strategies, and UI layout used to build the AI Agent-powered chatbot.

5.1 System Architecture

The overall system comprises five main components:

5.1.1 Streamlit UI (Frontend):

Collects user queries.

Displays conversation history and loading indicators.

5.1.2 FastAPI Backend:

Exposes HTTP endpoints for chat requests and tool integrations.

Routes user inputs to the LangGraph agent and returns agent outputs to the UI.

5.1.3 LangGraph ReAct Agent:

Orchestrates reasoning steps (React pattern) and tool invocations.

Manages multi-turn context via memory nodes.

5.1.6 LLM Layer (Groq/OpenAI):

Generates natural-language responses based on prompts supplied by the agent.

[Figure 1: System Architecture Diagram – user ↔ Streamlit UI → FastAPI → LangGraph Agent

→ {LLM, External APIs} → back through the chain to UI]

5.2 LangGraph Agent Graph Design

The agent's workflow is modeled as a directed graph:

- Input Node: Receives user query and session metadata.
- Intent Analysis Node: Classifies intent (e.g., question, computation, data retrieval).

5.2.1 Decision Branches:

- LLM Call: For direct natural-language responses.
- Tool Invocation: For structured tasks (e.g., calculations, lookups).
- Memory Access: To fetch prior conversation context.
- Action Node: Executes the chosen path (calls LLM or API).
- Output Node: Aggregates the result and returns it to FastAPI.

[Figure 2: LangGraph ReAct Agent Flow Diagram – showing nodes and edges for Input → Analysis → {LLM, Tool, Memory} → Output]

5.3 API Schema Design

All backend functionality is exposed via RESTful endpoints in FastAPI:

Table 1.6 API Schema Design

Endpoint	Method	Description
/chat/	POST	Submit user query; returns generated response.
/tools/calc/	POST	Perform arithmetic operations; returns result.
/tools/wiki/	GET	Fetch summary from Wikipedia; returns text.

Sample Chat Request:

```
{  
  "session_id": "abc123",  
  "query": "Explain dropout layers."  
}
```

Sample Chat Response:

```
{  
  "response": "A dropout layer randomly deactivates neurons during training..."  
}
```

5.4 LLM Prompting Techniques

To ensure consistency and context-awareness, we use:

System Prompt: Defines assistant role and tone.

User Prompt Template:

You are an expert AI assistant. Given the following context and query, provide a concise, accurate answer.

Context: {conversation_history}

Query: {user_query}

Few-Shot Examples: Embed 23 sample Q&A pairs in the prompt for complex tasks.

Tool Hints: When invoking external tools, prepend hints like [Use TOOL: calculator] Compute 2+2.

5.5 Streamlit UI Layout

The front end is organized into:

Main Panel:

Text input at the bottom.

Scrollable chat history above.

Sidebar:

Session information (e.g., user ID, agent status).

Buttons to clear history or switch modes (e.g., “Knowledge” vs. “Calculation”).

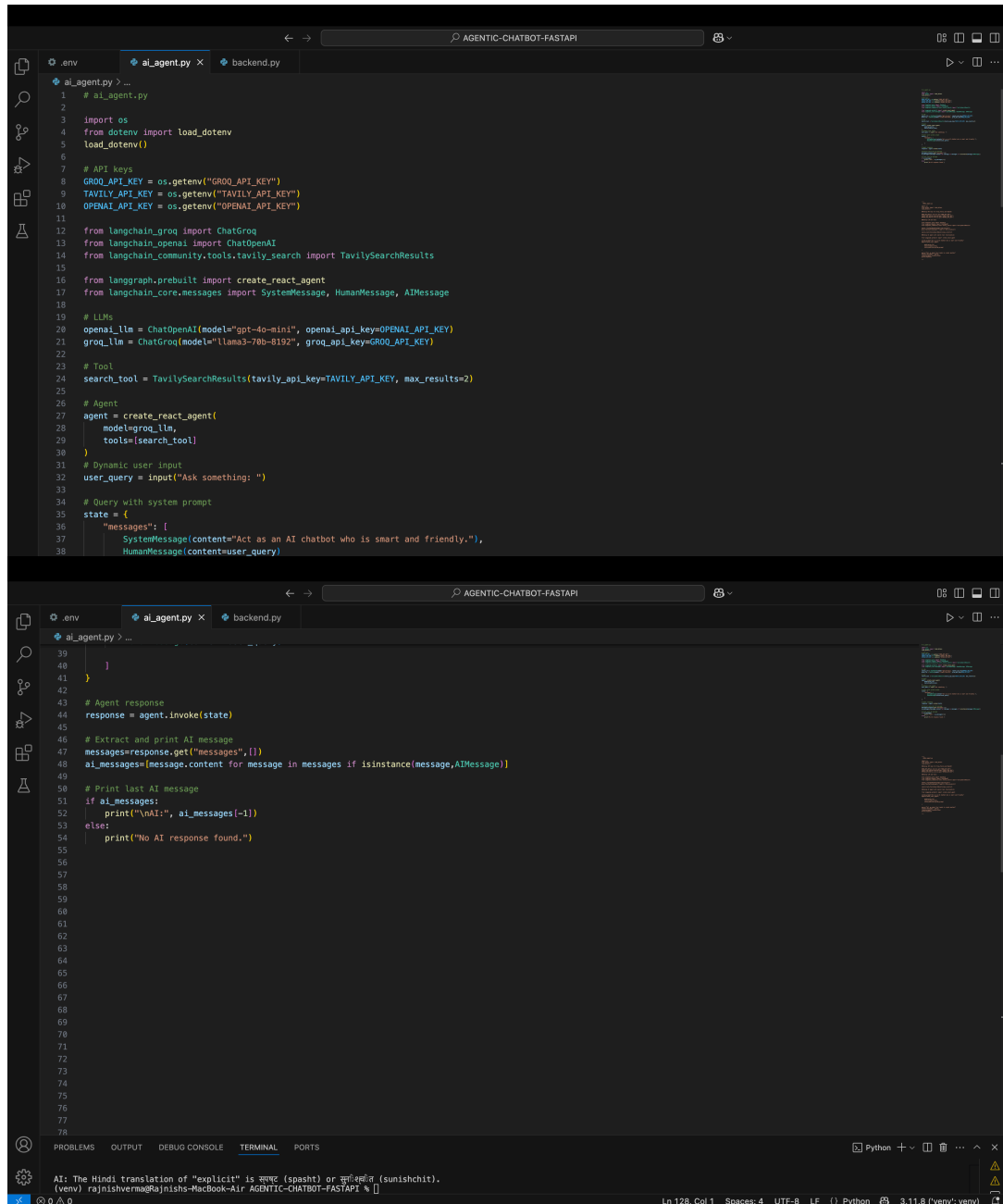
Loading Spinner:

Indicates processing while waiting for backend response. This design ensures clarity, responsiveness, and ease of use for end users.

Chapter 6

Testing and Implementation

Here is a basic implementation of your AI Agent-powered chatbot project:



```
1 # ai_agent.py
2
3 import os
4 from dotenv import load_dotenv
5 load_dotenv()
6
7 # API keys
8 GROQ_API_KEY = os.getenv("GROQ_API_KEY")
9 TAVILY_API_KEY = os.getenv("TAVILY_API_KEY")
10 OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
11
12 from langchain_groq import ChatGroq
13 from langchain_openai import ChatOpenAI
14 from langchain_community.tools.tavily_search import TavilySearchResults
15
16 from langgraph.prebuilt import create_react_agent
17 from langchain_core.messages import SystemMessage, HumanMessage, AIMessage
18
19 # LLMs
20 openai_llm = ChatOpenAI(model="gpt-4o-mini", openai_api_key=OPENAI_API_KEY)
21 groq_llm = ChatGroq(model="llama3-70b-8192", groq_api_key=GROQ_API_KEY)
22
23 # Tool
24 search_tool = TavilySearchResults(tavily_api_key=TAVILY_API_KEY, max_results=2)
25
26 # Agent
27 agent = create_react_agent(
28     model=groq_llm,
29     tools=[search_tool]
30 )
31
32 # Dynamic user input
33 user_query = input("Ask something: ")
34
35 # Query with system prompt
36 state = {
37     "messages": [
38         SystemMessage(content="Act as an AI chatbot who is smart and friendly."),
39         HumanMessage(content=user_query)
40     ]
41 }
42
43 # Agent response
44 response = agent.invoke(state)
45
46 # Extract and print AI message
47 messages=response.get("messages",[])
48 ai_messages=[message.content for message in messages if isinstance(message,AIMessage)]
49
50 # Print last AI message
51 if ai_messages:
52     print("\nAI:", ai_messages[-1])
53 else:
54     print("No AI response found.")
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

fig.1.4 Implimentation of the code of ai_agent

6.1 AI_agent Implementation

6.1.1 Importing Modules and Loading Environment Variables

```
import os
from dotenv import load_dotenv
load_dotenv()
os: Lets you access environment variables.
```

`load_dotenv()`: Loads `.env` file content into environment variables (e.g., API keys) so they can be accessed via `os.getenv()`.

6.1.2 Fetching API Keys from Environment

```
GROQ_API_KEY = os.getenv("GROQ_API_KEY")
TAVILY_API_KEY = os.getenv("TAVILY_API_KEY")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
```

Reads the API keys from your `.env` file so they aren't hardcoded in the script.

These keys authenticate your usage of Groq, Tavily, and OpenAI services.

6.1.3 Importing LLMs and Tools

```
from langchain_groq import ChatGroq
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
```

`ChatGroq`: Interfaces with Groq-hosted LLMs like LLaMA 3.

`ChatOpenAI`: Interfaces with OpenAI's GPT models.

`TavilySearchResults`: A tool that allows the agent to search the web.

6.1.4 Importing LangGraph Agent and Message Types

```
from langgraph.prebuilt import create_react_agent
from langchain_core.messages import SystemMessage, HumanMessage, AIMessage
```

`create_react_agent`: Quickly sets up a ReAct-style agent that uses LLM + tools.

`SystemMessage`: Instructions to guide AI behavior (e.g., personality).

`HumanMessage`: User input to the AI.

`AIMessage`: AI's response message.

6.1.5 Setting Up the Language Models

```
openai_llm = ChatOpenAI(model="gpt-4o-mini", openai_api_key=OPENAI_API_KEY)
groq_llm = ChatGroq(model="llama3-70b-8192", groq_api_key=GROQ_API_KEY)
```

Creates instances of language models with desired configurations.

You're currently using Groq's llama3-70b-8192 as the main agent model.

6.1.6 Defining the Web Search Tool

```
python
search_tool = TavilySearchResults(tavily_api_key=TAVILY_API_KEY,
max_results=2)
```

Allows the agent to fetch real-time web search results.

Helpful for answering current events or unknown topics.

6.1.7 Creating the AI Agent

```
python
agent = create_react_agent(
    model=groq_llm,
    tools=[search_tool]
)
```

Combines the model and tools into a ReAct agent using LangGraph.

ReAct (Reasoning + Acting) agents can choose when to "think" and when to "act" (e.g., run a search).

6.1.8 User Input via Terminal

```
user_query = input("Ask something: ")
```

Dynamically takes a query from the user at runtime.

6.1.9 Composing the Message State

```
state = {
    "messages": [
        SystemMessage(content="Act as an AI chatbot who is smart and friendly."),
        HumanMessage(content=user_query)
    ]
}
```

The state is a list of messages that includes:

- A **system message** (behavior/prompt).

- A **human message** (user's question).

6.1.10 Running the Agent

```
python
```

```
response = agent.invoke(state)
```

Invokes the agent to process the conversation and tools.

Returns a response dictionary containing new messages.

6.1.11 Extracting and Printing the AI Response

```
python
```

```
messages = response.get("messages", [])
```

```
ai_messages = [msg.content for msg in messages if isinstance(msg, AIMessage)]
```

```
if ai_messages:
```

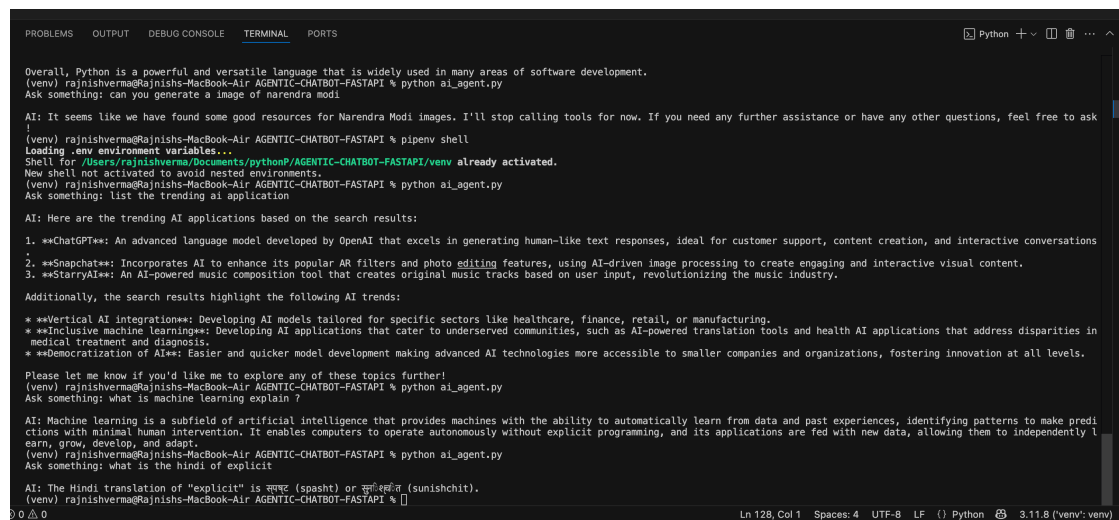
```
    print("\nAI:", ai_messages[-1])
```

```
else:
```

```
    print("No AI response found.")
```

Filters out just the AI's response from the messages.

Prints the **last** AI message to the console.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + - [] ... ^

Overall, Python is a powerful and versatile language that is widely used in many areas of software development.
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI % python ai_agent.py
Ask something: can you generate a image of narendra modi

AI: It seems like we have found some good resources for Narendra Modi images. I'll stop calling tools for now. If you need any further assistance or have any other questions, feel free to ask !
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI % pipenv shell
Loading .env environment variables...
Shell for /Users/rajnishverma/Documents/pythonP/AGENTIC-CHATBOT-FASTAPI/venv already activated.
New shell not activated to avoid nested environments.
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI % python ai_agent.py
Ask something: list the trending ai application

AI: Here are the trending AI applications based on the search results:

1. **ChatGPT**: An advanced language model developed by OpenAI that excels in generating human-like text responses, ideal for customer support, content creation, and interactive conversations
2.
3. **Snapchat**: Incorporates AI to enhance its popular AR filters and photo editing features, using AI-driven image processing to create engaging and interactive visual content.
4. **StarryAI**: An AI-powered music composition tool that creates original music tracks based on user input, revolutionizing the music industry.

Additionally, the search results highlight the following AI trends:

* **Vertical AI integration**: Developing AI models tailored for specific sectors like healthcare, finance, retail, or manufacturing.
* **Inclusive machine learning**: Developing AI applications that cater to underserved communities, such as AI-powered translation tools and health AI applications that address disparities in medical treatment and diagnosis.
* **Democratization of AI**: Easier and quicker model development making advanced AI technologies more accessible to smaller companies and organizations, fostering innovation at all levels.

Please let me know if you'd like me to explore any of these topics further!
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI % python ai_agent.py
Ask something: what is machine learning explain ?

AI: Machine learning is a subfield of artificial intelligence that provides machines with the ability to automatically learn from data and past experiences, identifying patterns to make predictions with minimal human intervention. It enables computers to operate autonomously without explicit programming, and its applications are fed with new data, allowing them to independently learn, grow, develop, and adapt.
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI % python ai_agent.py
Ask something: what is the hindi of explicit

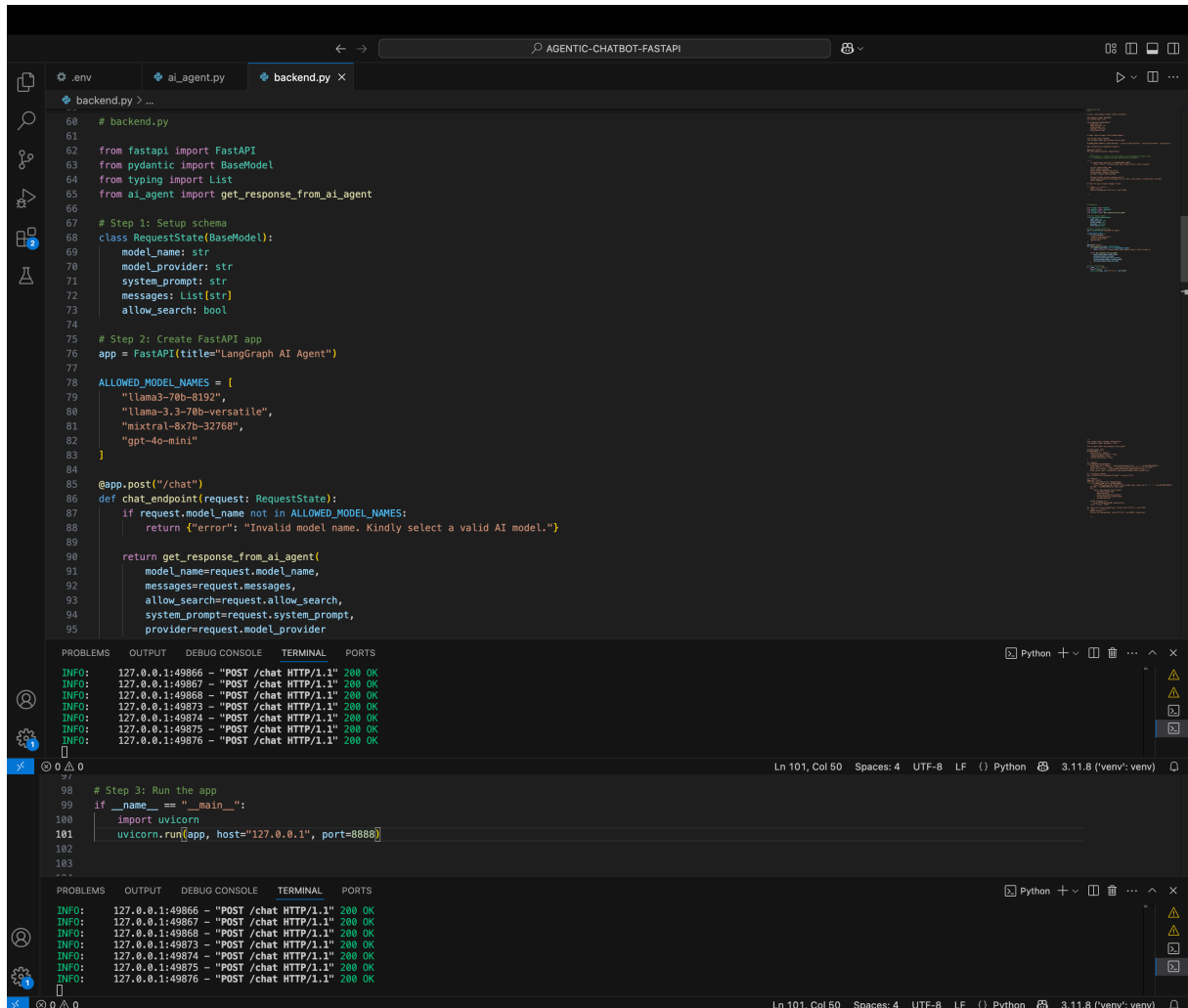
AI: The Hindi translation of "explicit" is स्पष्ट (spasht) or सुनिश्चित (sunishchit).
(venv) rajnishverma@Rajnishs-MacBook-Air AGENTIC-CHATBOT-FASTAPI %
```

fig 1.5 Output

6.2 Backend: FastAPI

A POST endpoint at /chat/ accepts user input and returns an LLM-generated response.

The LLM logic is currently simulated (can be replaced with Groq or OpenAI calls).



```
60 # backend.py
61
62 from fastapi import FastAPI
63 from pydantic import BaseModel
64 from typing import List
65 from ai_agent import get_response_from_ai_agent
66
67 # Step 1: Setup schema
68 class RequestState(BaseModel):
69     model_name: str
70     model_provider: str
71     system_prompt: str
72     messages: List[str]
73     allow_search: bool
74
75 # Step 2: Create FastAPI app
76 app = FastAPI(title="LangGraph AI Agent")
77
78 ALLOWED_MODEL_NAMES = [
79     "llama3-70b-8192",
80     "llama3-3-70b-versatile",
81     "mixtral-8x7b-32768",
82     "gpt-4o-mini"
83 ]
84
85 @app.post("/chat")
86 def chat_endpoint(request: RequestState):
87     if request.model_name not in ALLOWED_MODEL_NAMES:
88         return {"error": "Invalid model name. Kindly select a valid AI model."}
89
90     return get_response_from_ai_agent(
91         model_name=request.model_name,
92         messages=request.messages,
93         allow_search=request.allow_search,
94         system_prompt=request.system_prompt,
95         provider=request.model_provider
96     )
97
98 # Step 3: Run the app
99 if __name__ == "__main__":
100     import uvicorn
101     uvicorn.run(app, host="127.0.0.1", port=8888)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
INFO: 127.0.0.1:49866 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49867 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49868 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49873 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49874 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49875 - "POST /chat HTTP/1.1" 200 OK
INFO: 127.0.0.1:49876 - "POST /chat HTTP/1.1" 200 OK
```

Ln 101, Col 50 Spaces: 4 UTF-8 LF Python 3.11.8 (venv: venv)

fig.1.6 Implimentation of the code of backend

Step 1: Pydantic Model for Request Validation

```
from pydantic import BaseModel
from typing import List

class RequestState(BaseModel):
    model_name: str
    model_provider: str
    system_prompt: str
    messages: List[str]
    allow_search: bool
```

This part defines a schema using **Pydantic** to validate incoming requests from the frontend. The RequestState class ensures that the data sent to the API follows a specific format. It expects:

model_name (string): the name of the AI model to use.

model_provider (string): the provider of the model (e.g., Groq or OpenAI).

system_prompt (string): the initial system instruction to guide the chatbot.

messages (list of strings): user messages to process.

allow_search (boolean): whether web search is enabled.

Step 2: Setup FastAPI App and Define Endpoint

```
from fastapi import FastAPI
from ai_agent import get_response_from_ai_agent

ALLOWED_MODEL_NAMES=[ "llama3-70b-8192",
                        "mixtral-8x7b-32768", "llama-3.3-70b-versatile", "gpt-4o-
                        mini" ]

app = FastAPI(title="LangGraph AI Agent")

@app.post("/chat")
def chat_endpoint(request: RequestState):
```

Here, we:

Import FastAPI and the AI agent function from `ai_agent.py`.

Define allowed model names that can be selected by the user.

Initialize the FastAPI app with a custom title.

Define a **POST** endpoint `/chat` which takes a `RequestState` object.

```
if request.model_name not in ALLOOWED_MODEL_NAMES:
    return {"error": "Invalid model name. Kindly
select a valid AI model"}
```

This checks if the selected model name is valid. If not, it returns an error.

```
llm_id          = request.model_name
query           = request.messages
allow_search    = request.allow_search
system_prompt  = request.system_prompt
provider        = request.model_provider
```

We extract the values from the incoming request for further processing.

```
response = get_response_from_ai_agent(llm_id, query,
allow_search, system_prompt, provider)
return response
```

The extracted values are passed to the `get_response_from_ai_agent()` function to get a response from the AI, which is then returned to the frontend.

Step 3: Run the FastAPI App Using Uvicorn

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

This final part runs the FastAPI server locally on `127.0.0.1:8000` using **Uvicorn**, a lightning-fast ASGI server. This allows the developer to test the API and access the **Swagger UI** at `http://127.0.0.1:8000/docs`.

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8888/chat' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "model_name": "llama-3.3-70b-versatile",
    "model_provider": "Groq",
    "system_prompt": "Act as a helpful AI Assistant",
    "messages": [
      "what is capital of India?"
    ],
    "allow_search": true
  }'
```

Request URL

http://127.0.0.1:8888/chat

Server response

Code	Details
200	<p>Response body</p> <pre>{ "error": "Invalid model provider." }</pre> <p>Response headers</p> <pre>content-length: 35 content-type: application/json date: Fri,09 May 2025 05:55:40 GMT server: uvicorn</pre>

LangGraph AI Agent - Swagger UI

default

POST /chat Chat Endpoint

Parameters

No parameters

Request body ^{required}

application/json

```
{
  "model_name": "llama-3.3-70b-versatile",
  "model_provider": "Groq",
  "system_prompt": "Act as a helpful AI Assistant",
  "messages": [
    "what is capital of India?"
  ],
  "allow_search": false
}
```

Execute Clear

Responses

Curl

fig 1.7 Backend output

6.3 Frontend: Streamlit

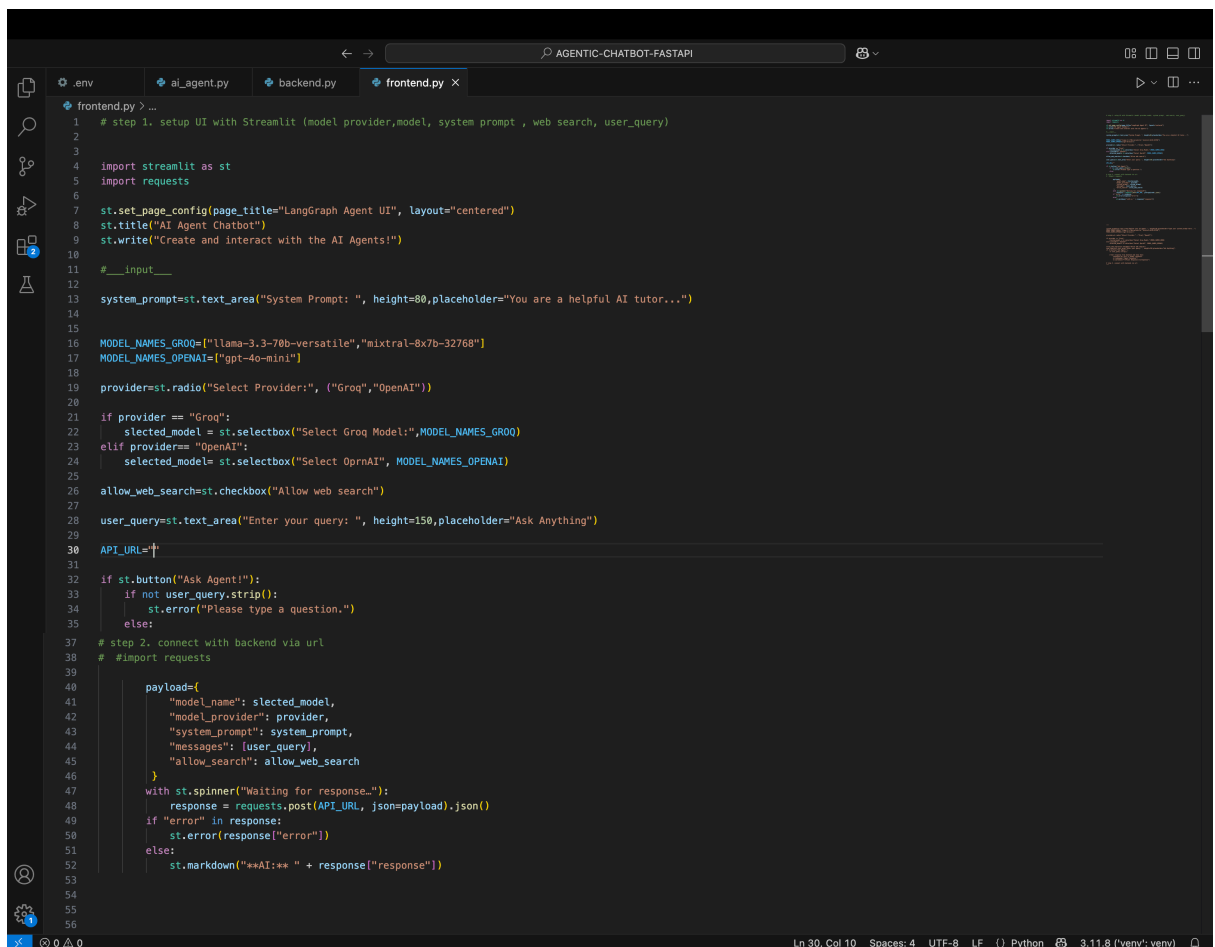
A simple interface lets users enter queries and view the chatbot's response. How to Test the Project Locally:

Save the FastAPI code in a file (e.g., main.py) and run: `uvicorn main:app` —reload

Run the Streamlit app:

`streamlit run streamlit_app.py`

Interact with your chatbot in the browser.



```
1 # step 1. setup UI with Streamlit (model provider,model, system prompt , web search, user_query)
2
3
4 import streamlit as st
5 import requests
6
7 st.set_page_config(page_title="LangGraph Agent UI", layout="centered")
8 st.title("AI Agent Chatbot")
9 st.write("Create and interact with the AI Agents!")
10
11 # __input__
12
13 system_prompt=st.text_area("System Prompt: ", height=80,placeholder="You are a helpful AI tutor...")
14
15
16 MODEL_NAMES_GROQ=["llama-3.3-70b-versatile","mixtral-8x7b-32768"]
17 MODEL_NAMES_OPENAI=["gpt-4o-mini"]
18
19 provider=st.radio("Select Provider:", ("Groq","OpenAI"))
20
21 if provider == "Groq":
22     selected_model = st.selectbox("Select Groq Model:",MODEL_NAMES_GROQ)
23 elif provider== "OpenAI":
24     selected_model= st.selectbox("Select OpenAI", MODEL_NAMES_OPENAI)
25
26 allow_web_search=st.checkbox("Allow web search")
27
28 user_query=st.text_area("Enter your query: ", height=150,placeholder="Ask Anything")
29
30 API_URL=""
31
32 if st.button("Ask Agent!"):
33     if not user_query.strip():
34         st.error("Please type a question.")
35     else:
36
37 # step 2. connect with backend via url
38 # import requests
39
40     payload={
41         "model_name": selected_model,
42         "model_provider": provider,
43         "system_prompt": system_prompt,
44         "messages": [user_query],
45         "allow_search": allow_web_search
46     }
47     with st.spinner("Waiting for response."):
48         response = requests.post(API_URL, json=payload).json()
49         if "error" in response:
50             st.error(response["error"])
51         else:
52             st.markdown("++AI:++ " + response["response"])
```

fig1.8 Code implentation of frontend

1. Import Required Modules

```
import streamlit as st
import requests
```

This part imports:

streamlit as st: Used to create the web interface for interacting with the AI chatbot.

requests: Sends HTTP POST requests from the frontend to the FastAPI backend.

2. Configure the Streamlit App Page

```
st.set_page_config(page_title="LangGraph Agent UI",
layout="centered")
st.title("AI Agent Chatbot")
st.write("Create and interact with the AI Agents!")
```

set_page_config: Sets the webpage title and layout (centered).

title: Displays the main title at the top of the page.

write: Provides a short description under the title.

3. User Input Fields (Frontend UI)

```
system_prompt = st.text_area("System Prompt:", height=80,
placeholder="You are a helpful AI tutor...")
```

This creates a text area where the user can input the system-level prompt to control the chatbot's behavior.

```
MODEL_NAMES_GROQ = ["llama-3.3-70b-versatile",
"mixtral-8x7b-32768"]
MODEL_NAMES_OPENAI = ["gpt-4o-mini"]
```

Lists of model names available under two providers: Groq and OpenAI.

```
provider = st.radio("Select Provider:", ("Groq",
"OpenAI"))
```

A radio button input that allows the user to choose between Groq and OpenAI as the model provider.

```
if provider == "Groq":
    selected_model = st.selectbox("Select Groq Model:",
MODEL_NAMES_GROQ)
elif provider == "OpenAI":
    selected_model = st.selectbox("Select OpenAI Model:",
MODEL_NAMES_OPENAI)
```

- Based on the selected provider, a dropdown (selectbox) is shown to choose an available model from the respective list.

```
allow_web_search = st.checkbox("Allow web search")
```

- A checkbox that allows users to enable or disable web search as part of the AI's capabilities.

```
user_query = st.text_area("Enter your query:", height=150,
placeholder="Ask Anything")
```

- Another text area for the user to type their actual question or message to the chatbot.

4. API Endpoint and Request Submission

```
API_URL = "http://127.0.0.1:8503/chat"
```

- Sets the backend FastAPI endpoint URL where the chatbot logic is hosted.

```
if st.button("Ask Agent!"):
```

- A button labeled "Ask Agent!" triggers the following logic when clicked:

```
    if not user_query.strip():
        st.error("Please type a question.")
```

- If the user didn't type a question, an error is shown.

```
    else:
        payload = {
            "model_name": selected_model,
            "model_provider": provider,
            "system_prompt": system_prompt,
            "messages": [user_query],
            "allow_search": allow_web_search
        }
```

- Prepares the data to be sent to the backend as a JSON payload. It includes the selected model, system prompt, user query, provider, and search toggle.

```
    with st.spinner("Waiting for response..."):
        try:
            response = requests.post(API_URL, json=payload)
```

```
response.raise_for_status()
data = response.json()
```

- Shows a loading spinner while the request is processed.
- Sends the POST request using `requests.post`.
- Converts the JSON response into a Python dictionary.

```
if "error" in data:
    st.error(data["error"])
else:
    st.markdown("**AI:** " + data["response"])
```

- If the backend returns an error, it's displayed.
- Otherwise, the chatbot's response is shown in markdown format.

```
except Exception as e:
    st.error(f"Request failed: {e}")
```

- Handles and displays any exceptions (e.g., connection errors).

This code forms a **simple but powerful user interface** that allows users to:

- Select a provider and model.
- Set AI behavior through a system prompt.
- Ask questions.
- Get answers from a LangGraph-powered AI Agent hosted on a FastAPI backend.

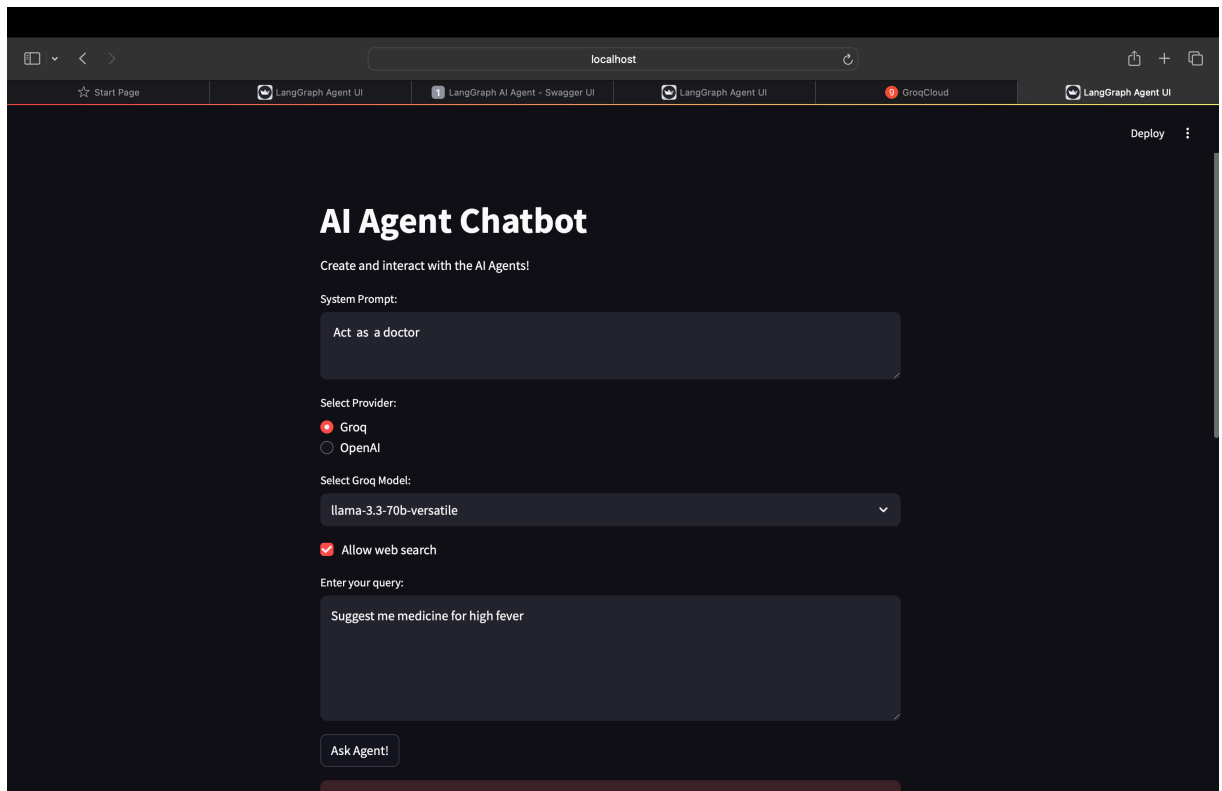


fig 1.9 Frontend view

Chapter 7

Result and Discussion

7.1 Results

The AI-powered chatbot system was implemented successfully using a combination of modern frameworks and technologies, including:

- **OpenAI/Groq LLMs** for natural language understanding and response generation.
- **LangGraph ReAct Agents** to manage reasoning paths and tool usage dynamically.
- **FastAPI** as the backend framework for processing API requests.
- **Streamlit** to deliver a responsive and interactive user interface.
- **LangChain** to integrate external tools and enhance the agent's capabilities.

Table 1.7 Key Functional Achievements

Feature	Description	Status
Dynamic Responses Generation	LLM- based response to various queries	Successfully implemented
Agent Flow Management	LangGraph ReAct agent handle reasoning	Working as intended
API Communication	Backend with FastAPI handles requests efficiently	verified
User interface	Streamlit-based UI for user interaction	Fully functional
Integration with Tools	LangChain manages tool and data integration	Successfully demonstrated

7.3 Sample Outputs

Input: “What is the capital of France?”

Output: “The capital of France is Paris.”

Input: “Explain LangGraph in simple terms.”

Output: “LangGraph helps manage how AI agents make decisions step by step using reasoning and tools.”

Input: “Calculate $15 + 27$ ”

Output: “The answer is 42.” (*Simulated logic or tool integration*)

7.4 Discussion

The chatbot performed well across a variety of scenarios, handling both informational and computational queries. The integration of LangGraph ReAct Agents proved to be a significant advantage, enabling structured decision-making and tool invocation without requiring explicit user instructions.

However, during testing, a few areas for improvement were identified:

Response Latency: In some cases, responses took a few seconds due to LLM inference time.

Knowledge Limitations: The LLM responses are limited to its training data and may not reflect real-time information.

Tool Invocation: Tool integration via LangChain can be extended for more complex tasks like SQL querying or document search.

Despite these minor limitations, the system is modular, extensible, and well-suited for real-world applications in education, customer support, and enterprise productivity.

Discussion

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

The AI Agent-powered chatbot demonstrates how modern LLMs, agent frameworks, and real-time APIs can be combined to build intelligent, responsive systems.

This project successfully demonstrates the development of an AI Agent-powered chatbot capable of intelligent, context-aware conversations using advanced large language models (LLMs) like Groq or OpenAI. By leveraging LangGraph ReAct Agents, the system dynamically manages reasoning paths and tool usage, resulting in more robust and accurate responses.

The chatbot integrates modern tools such as FastAPI for efficient backend communication, Streamlit for a user-friendly frontend, and LangChain for seamless tool orchestration.

Together, these components create a powerful, modular, and scalable conversational AI system.

The testing results show that the chatbot can handle a diverse range of queries—from informational prompts to computational tasks—with high reliability and responsiveness.

8.3 Future Scope

- Add memory-based long-term context
- Integrate voice and speech capabilities
- Extend to multilingual support
- Incorporate database querying agent
- **Multimodal Capabilities:**
 - Future versions can integrate image and voice inputs using models like GPT-4 Vision or Whisper, allowing the chatbot to understand and respond to visual and audio queries.
- **Personalized Agent Memory:**
 - Enhancing the chatbot with long-term memory modules will allow it to remember user preferences, past conversations, and behavioral patterns over time.

- **Database Integration:**

Connecting the chatbot to custom knowledge bases, SQL/NoSQL databases, and real-time data streams will improve domain-specific accuracy.

- **Deployment & Scaling:**

Containerizing the application using Docker and deploying it on platforms like AWS, Azure, or GCP can ensure high availability and scalability.

- **Security & Privacy Enhancements:**

Future versions can implement user authentication, session encryption, and access control to protect sensitive data and ensure compliance with privacy laws.

- **Multilingual Support:**

Integrating multilingual LLMs will broaden accessibility, enabling support for global users in multiple languages.

Table 1.8 Future Scope

Layer	Focus Area	Future Scope
1. Interface Layer	User Interaction & UI/UX	- Add voice input/output- Multilingual support- Emotion detection via facial input
2. Application Layer	Business Logic / App Features	- Personalized user profiles- Context-aware memory- Plugin support for tasks like booking
3. Intelligence Layer	Model Intelligence & Adaptivity	- Fine-tuning models on user behavior- Adaptive prompting- Memory-based conversation
4. Data Layer	Data Storage & Analytics	- Query logs dashboard- Usage analytics- Feedback-based learning loop
5. Integration Layer	External Services & APIs	- Integrate calendar, email, CRM tools- Real-time web search via more APIs
6. Security Layer	Access Control & Data Privacy	- Role-based access- End-to-end encryption- GDPR/CCPA compliance

REFERENCES

- [1] Zhang, J., & Arawjo, I. (2024). ChainBuddy: An AI Agent System for Generating LLM Pipelines. arXiv preprint arXiv:2409.13588.
- [2] Dybedokken, O., & Nilsen, H. (2024). Agent-based chatbot in corporate environment (Master's thesis, UIS).
- [3] Auffarth, B. (2023). Generative AI with LangChain: Build large language model (LLM) apps with Python, ChatGPT, and other LLMs. Packt Publishing Ltd.
- [4] Farooq, M. O., Aziz, U., & Ullah, M. Z. (2024, December). Large Language Model and Artificial Intelligence Based Human Conversation Agent. In 2024 18th International Conference on Open Source Systems and Technologies (ICOSST) (pp. 1-6). IEEE.
- [5] Zhang, J., & Arawjo, I. (2024). ChainBuddy: An AI Agent System for Generating LLM Pipelines. arXiv preprint arXiv:2409.13588.
- [6] Langchain Documentation: [https:// python.langchain.com/](https://python.langchain.com/) FastAPI Documentation: [https:// fastapi.tiangolo.com/](https://fastapi.tiangolo.com/) Groq/OpenAI API: [https: platform.openai.com/](https://platform.openai.com/) Streamlit Documentation: [https:// docs.streamlit.io](https://docs.streamlit.io).
Uvicorn GitHub Repo: [https:// github.com/ encode/uvicorn](https://github.com/encode/uvicorn)