DEVELOPMENT OF A MULTI-AGENT CHATBOT FOR USER QUERY RESOLUTION FOR UTAR

BY

HENG THEE YONG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to express thanks and appreciation to my supervisor, Prof. Ts. Dr. Liew Soung Yue and my moderator, Dr. Tan Hung Khoon who have given me a golden opportunity to involve in a multi-agent chatbot project leveraging artificial intelligence (AI) and large language models (LLMs). Besides that, they have given me a lot of guidance in order to complete this project. When I was facing problems in this project, the advice from them always assists me in overcoming the problems. Again, a million thanks to my supervisor and moderator.

ABSTRACT

Universities generate vast amount of information daily, including programme details, course structures, schedules, policies, and procedures. This information is often distributed across multiple sources such as university websites, portals, and PDF documents, making it difficult for students, staff, prospective applicants, and parents to quickly access accurate and up-to-date details. At Universiti Tunku Abdul Rahman (UTAR), this challenge highlights the need for a unified and intelligent information access system. To address this, the project proposes and develops a multi-agent Retrieval-Augmented Generation (RAG) chatbot designed specifically for UTAR. The chatbot architecture employs specialized agents namely Admissions Agent, Finance Agent, and Examinations Agent each connected to its own vector database containing structured knowledge extracted from official university sources. Data ingestion is automated through a web scraping and PDF download module that handles inconsistencies such as broken SSL certificates on UTAR domains, ensuring reliable and up-to-date knowledge collection. The system integrates OpenAI API service as the base large language model (LLM), with LangChain for orchestration, Chroma as the vector database, Flask for backend development, and React for the frontend user interface deployed on Firebase. The backend is deployed on Render to support scalability, concurrency, and real-time availability. Evaluation was carried out using technical performance testing alongside user experience testing through a structured Google Form survey. The results show that the chatbot delivers accurate and contextually relevant answers within an acceptable response time, while user feedback indicates strong satisfaction with ease of use, usefulness, and willingness to reuse the system. Open-ended responses also highlighted areas for improvement, such as expanding departmental coverage and tighter integration into UTAR's official website. By enabling 24/7 access to official university knowledge sources, the chatbot improves information accessibility and user satisfaction, demonstrating the feasibility of applying multi-agent RAG architectures in the higher education context. Future enhancements will focus on integration with UTAR's official platforms, and extending the system to additional departments and more.

Area of Study: Application of LLM, Development of Chatbot, Information Retrieval

Keywords: Chatbot, LLM, RAG, Multi-agent, Knowledge Base

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
FAQ	Frequently Asked Questions
FICT	Faculty of Information and Communication Technology
LLM	Large Language Model
NLG	Natural Language Generation
NLU	Natural Language Understanding
RAG	Retrieval Augmented Generation
RAM	Random Access Memory
UI	User Interface
UTAR	Universiti Tunku Abdul Rahman

Chapter 1
Introduction

1.1 Problem Statement and Motivation

In every university, students, university staff, prospective applicants or even students' parents often seek and expect to get quick, accurate and correct responses to a wide variety of queries which includes information related to programs offered by the university, admissions, scholarships, financial aids, facilities in the campus of university and many others. Although there exists university website with almost all information related to the university, Frequently Asked Questions (FAQ) page of university website, university front desk receptionists and email support and even live chat agent which may answer all the queries, there exists several issues. It is time consuming, lack of personalization and inefficient for a user to browse through a university website to get information about a university especially the user is trying to search for many types of information for every different university with the purpose of finding a university that best suits the user's interests, needs and budget. Moreover, there may be increase in administrative and staff workloads when a vast amount of users are asking questions to university front desk receptionists and email support. Live chat agent may not be available 24/7 and may not be able to provide real-time responses to users which will dissatisfy and frustrate the users since a non-automated live chat agent required a university staff to be online to answer user queries. According to experiment conducted in [13], developing a chatbot is an efficient way to provide users with optimal and real-time responses when users seek to find some information related to university. This solution can also reduce the workload of university staff [13].

The motivation of this project is to address the limitations mentioned above by developing a multi-agent chatbot integrated with RAG that leverages OpenAI's API and LangChain libraries for quicker, more accurate, concise and context-relevant responses in response to user queries. This also saves the time of browsing through all sources of information to search for university-related information. By providing different specific knowledge to separate agents, the system is able to provide more context-specific and concise responses to users. With RAG, each agent has its own knowledge base as source of contextual information to generate responses specific and relevant to the context of user queries. The development of the system aims to improve the overall user experience in getting information about the university which may directly or indirectly smoothen the university administrative processes. Although there already exist some multi-agent RAG chatbots developed for some universities or for other purposes, the project aims to develop multi-agent RAG chatbots with improved performance for UTAR.

1.2 Objectives

One of the objectives of the project is to develop a scalable, intelligent and robust multi-agent RAG chatbot system that is capable of providing contextually relevant and accurate responses to user queries related to the university, UTAR. The second objective is to achieve an accuracy of at least 90% in user query resolution and in user satisfaction. This ensures that the chatbot provide accurate and up-to-date information to users. The third objective of the project is to reduce the average response time of the chatbot in response to user queries to 6s. This can improve user experience and user satisfaction and even improve UTAR's reputation by allowing users who are searching for university-related information to get quick and accurate responses. The fourth objective is to deploy the chatbot so that it can be access by UTAR students and staff, parents, prospective applicants at UTAR and any person with Internet access to get quick and accurate university-related information in a convenient manner. The project will not cover the techniques and technologies employed to authenticate the user to interact with the chatbot. All the users interacting with the chatbot during testing process are assumed to be trusted users who will not misuse the resources and services of the chatbot. The project will not cover the integration of the chatbot into the UTAR website.

1.3 Project Scope and Direction

This project focuses on developing an improved multi-agent chatbot system for user query resolution, focusing on answering the queries and questions related to UTAR. A modular architecture where each agent is responsible for a specific domain will be designed for the implementation of the multi-agent chatbot. Besides that, the project will cover the integration of databases into the chatbot system. Multiple databases will be created for different agents so that each agent has its own database that stores information about the department or division that it is responsible for. Every agent is responsible for answering questions related to its own department or division by retrieving contextual information from its own knowledge base and using the contextual information to generate responses for the user query. LangChain libraries which are UnstructuredPDFLoader and RecursiveCharacterTextSplitter will be used to extract and load the contents from university PDF documents into each agent's database to improve the efficiency of context retrieval in the system. Agent orchestrator is the agent that will decide the most suitable agent to answer a user query. Conversational memory module will be developed to maintain the conversational context to allow users to ask follow-up questions. The project will also cover testing the performance of chatbot in terms of response time, accuracy of response and user satisfaction.

1.4 Contributions

The contributions of this project include the design of scalable and modular multi-agent RAG chatbot architecture for universities. An agent with a new database can be easily added into the chatbot system without having to modify many parts of the code and files. The second contribution is the demonstration and practical applications of OpenAI API and LangChain libraries in a multi-agent chatbot system to improve its context retrieval and natural language generation capabilities in response to user query. Furthermore, this project also contributes the development of a conversational memory module which allows user to ask follow-up questions and to make the conversation more natural and continuous.

By having this chatbot system, students, university staff, prospective applicants and even students' parents are able to get almost all information about a university in one place. They just have to send their queries to the chatbot and the chatbot will get the relevant and related information and generate responses in human-understandable language to them. This saves their time in getting information related to a university. The architecture of the chatbot can be referenced and used by most organisations, companies, colleges and universities in which they have different departments or faculties and each agent in the multi-agent chatbot system is designed to represent the information source of a department or faculty for the chatbot. Each department or faculty will not have access to the data of others through the system.

1.5 Report Organization

This report is structured into seven chapters. Chapter 1 introduces the project background, problem statement, objectives, scope, and contributions. Chapter 2 reviews related works and existing chatbot technologies. Chapter 3 outlines the methodology, including system architecture and design approach. Chapter 4 presents the system design, detailing the components and their interactions. Chapter 5 explains the implementation process, including setup, configuration, and deployment. Chapter 6 evaluates the system through testing, user feedback, and discussion of results. Finally, Chapter 7 concludes the project by summarizing its achievements and providing recommendations for future improvements.

1.6 Background Information

In recent years, chatbots have emerged as popular and essential tools in various sectors like e-commerce, healthcare, hospitality, tourism, banking and customer service [2]. These conversational agents simulate human-like dialogues and are available 24/7 to provide real-time responses and assistance to users. The vigorous growth and development and increase in popularity of chatbots have stimulated many gigantic companies such as Apple, Amazon, Google and Microsoft to invest in this field. The massive investment in the field causes the advent of learning models, natural language processing (NLP) and stimulates the development of technologies such as deep learning and information retrieval [2]. Despite the advancements, most chatbots are only built to focus on a single or finite set of domains [6]. Traditional chatbots are built using intent-based approach in which the chatbots are designed to identify the user's intent and reply with a predefined set of responses and patterns [1]. Based on the research in [1], traditional intent-based chatbot has some limitations which include resource-intensive and higher cost required to develop and maintain the chatbot, weaknesses in flexibility and adaptability, inability to adapt to rapidly changing user needs, and the chatbot may be unable to recognise the user's request if the question from the user contains novel words that were not part of the sampled utterances which is the list of phrases the chatbot is able to understand to identify particular intent.

To resolve the limitations of traditional chatbots, large language model (LLM)-based chatbot implementation is the promising solution [1]. LLMs such as GPT-4 are complicated and advanced forms of artificial intelligence that are trained on vast amount of textual data and can perform a wide range of tasks which include understanding and generation of natural language, research and data analysis and personalization and recommendation. RAG is integrated into the LLMs to improve its information retrieval capabilities by extracting information and generating responses from unstructured data [1] and to allow LLMs to get contextual support from predefined documented knowledge to answer user queries [4]. Furthermore, the integration of multiple agents into the LLMs makes the LLMs even more powerful and robust as each agent is endowed with specific context knowledge and will provide required contextual data to the LLMs to answer user queries.

Chapter 2
Literature Review

2.1 Previous Works on Chatbot Development

2.1.1 Review of ChatGPT as Emerging AI Technology

ChatGPT is a large language model (LLM) that is designed to understand and generate human-understandable and human-like text by processing a massive amount of data [3]. It started to become popular and gain the attention of the general public in late 2020 and early 2021, following the release of OpenAI's GPT-3 model in June 2020 [3]. ChatGPT is able to generate coherent and context-relevant responses to a broad range of prompts [3]. However, it is unable to provide in-depth and specific responses for some domains, but it will just provide the general knowledge about the domains. ChatGPT provides API that allows developers of many companies to integrate its language capabilities directly into their websites, applications or services [3]. There are some key advantages for the usage of ChatGPT API which include rapid development of a chatbot, ability to handle large amount of user queries, improving customer experience by providing fast, accurate and contextually appropriate responses to customer's queries and many other advantages. Nevertheless, ChatGPT may cause privacy breaches and confidentiality issues as company data that is used for queries is processed by OpenAI's servers. There may be possible leakage of company's and user's confidential information. In order for the chatbot built with ChatGPT API to learn a company's domain specific knowledge, extensive training is required. In addition, the company may not have full control over the AI model as the core model is managed by OpenAI.

2.1.2 Examining Local and API-based LLM Options for Chatbot Implementation

The choice of LLMs is one of the most important aspects that must be considered when designing and developing a chatbot. Deciding and choosing the LLMs that best suit a person's or and organization's needs and budget are crucial in determining the success of the project. The choice of LLMs can be divided into local models and API-based models. Local models are the LLMs such as Llama, Mistral, Qwen and many more that operate and run within an organization's infrastructure and can even be run by an individual's personal computer [14]. On the other hand, API-based models are LLMs such as Anthropic, Google's Gemini and

OpenAI's GPT models that are hosted on cloud platforms with extensive resources such as storage and processing power available [14].

[15] focused on enhancing the traditional RAG techniques for processing complex automotive documents which typically have intricate layouts such as having multicolumn formats and complex tables using locally deployed Ollama models. Ollama is a tool or framework for building and running a language model in a local machine and Llama is one of the LLMs that can be found in Ollama [12]. Incorporation of self-reflective RAG or self-RAG mechanism into the chatbot system is one advantage of the solutions proposed by the paper above. Self-RAG is an enhanced version of a normal RAG framework which aims at improving the quality and factual correctness of LLMs by using on-demand information retrieval combined with a process of self-evaluation [15]. The self-RAG mechanism can improve the accuracy and efficiency of Ollama model employed in the paper in retrieving and generating response. A local model provides an advantage of ensuring the security of confidential and sensitive data as well as data privacy and providing complete control over them as it operates locally in an organization. On the other hand, further optimization may be required for real-time performance in resource-constrained environment as local models run on an organization's infrastructure and depends heavily on the computational resources of the infrastructure. To improve the performance of local models, additional costs are required to purchase better hardware resources and maintain the infrastructure.

[16] explored the implementation of an AI health assistant designed to complement an existing eHealth data acquisition system. The AI health assistant operated locally, and it allowed users to query medical information in natural language. It generated personalized answers based on specific patient data relevant to the user query and the security of the communications were enhanced by utilising secure communications through the Matrix decentralized open protocol [16]. The AI health assistant was built using open-source software and by hosting the LLMs locally, so this reduces the costs in developing the chatbot and avoid relying on other third parties [16]. Nevertheless, the chatbot had only maximum of 2048 tokens so this reduced the amount of information can be processed by the chatbot at once.

Besides that, there was a project focusing on developing a chatbot using local LLMs to enhance adherence to exercise programs for individuals with knee osteoarthritis (KOA) [17]. Advance techniques such as incorporation of Parameter-Efficient Fine-Tuning and self-RAG which was also implemented in [15] to optimize computational efficiency and reduce hallucinations [17]. The paper demonstrated that participants took part in the conducted survey were more satisfied with the responses given by their locally developed chatbot compared to the responses given by ChatGPT [17]. However, ChatGPT was not finetuned with any domain knowledge which is related to KOA but the chatbot was provided with clinical guidelines to generate more relevant responses that were related to KOA. Hence, the comparison between ChatGPT and their chatbot seems to be not equitable or potentially biased. If ChatGPT is provided with clinical guidelines as knowledge source, it may provide better response than their chatbot as ChatGPT is hosted on cloud platforms with extensive computational resources.

There were some papers which use API-based models to build chatbots. [1] discusses the development of a teaching assistant chatbot for Data Mining and Text Analytics courses at the University of Leeds. The project used API-based model which was OpenAI GPT-3.5-Turbo model [1]. The research started with comparing the two approaches in developing an educational chatbot as teaching assistant to aid students in learning in specific fields [1]. The researchers used an open-source framework which was LangChain to integrate RAG with the OpenAI GPT-3.5-Turbo model in developing LLM-driven chatbots by utilizing LangChain's tools and APIs [1]. The Question-Answer (QA) pairs were generated using Question-Answer Generation (QAG) approach which generated a large set of QA pairs relevant to the context and content of a given passage to be referenced by the OpenAI GPT-3.5-Turbo model [1]. LLM-based chatbot integrated with RAG was more proficient in dealing with new and probably unseen queries and provided more context-relevant answers and scalable as its knowledge base could be expanded more easily [1]. The model was evaluated using the tools in a LangChain's library [1]. The results of the evaluation showed that the developed teaching assistant provided correct response, but it did not provide comprehensive answer to a particular question. The system used conversational memory to handle follow-up questions by users to maintain context throughout the interaction. Nevertheless, the model might not be able to cope with complex and lengthy dialogues as the conversation context might exceed the token limit of the model due to the usage of conversational memory in RAG system which was integrated with the model [1]. Since no proprietary and confidential information such as student and university staff records were used in the chatbot development, so there will not be any leakage of sensitive information. Thus, API-based model with extensive computational resources seemed to fit the needs of the project well. [4] was another paper that used API-based model which was GPT-4 in developing chatbot system for Decision Support in net-zero emission energy systems, leveraging LLMs and RAG as shown in Figure 2.1.1.

2.1.3 Knowledge Integration Methods: RAG and Self-RAG

A research paper by [2] tried to enhance restaurant chatbots by integrating RAG with LLM for smoother response generation and to improve context learning. RAG enhances a LLM's response generation capability by incorporating external and domain-specific data and information into the LLM in which the LLM can use to provide a more specific and relevant responses to user query. The methods employed in the development of restaurant chatbots involved constructing a Neo4j Knowledge graph using restaurant data as an external knowledge source [2]. This graph was navigated to align the user's question with suitable answer tokens by leveraging Term Frequency - Inverse Document Frequency (TF-IDF) embeddings [2]. The relevant tokens along with the user questions, were then used to enhance the context provided to the T5 language model, enabling it to deliver more nuanced and tailored responses to users [2]. The Neo4j knowledge graph played its role as graph database to store and retrieve knowledge effectively [2]. The results of the paper have shown that the model of the chatbot performed and generalized well and was suitable to be used in real-life scenarios [2]. However, the paper did not provide sampled picture illustrations on how the restaurant chatbot interact with user who will send queries and receive context-relevant answer in human-understandable language. The paper also did not mention how the chatbot perform or scale with increasing length and complexity of user queries, increasing user traffic and larger Neo4j knowledge graph.

There are many other papers such as [1] and [4] that integrated RAG system into their chatbots system to provide their chatbots with domain-specific knowledge so that their chatbots can generate responses that are more relevant and related to user queries. [1] focused on developing a teaching assistant chatbot for Data Mining and Text Analytics courses at the University of Leeds and providing all learning materials related to Data Mining and Text Analytics courses so that the teaching assistant was equipped with the knowledge to answer relevant questions. In [4], three separate document stores namely general document store, summary reports store and event notifications store were created to store unstructured knowledge for the Behavior Analyzer and Knowledge Retriever, ensuring they can source relevant and comprehensive information for decision support tasks.

There were some papers which focused on developing Self-RAG chatbot. Self-RAG is the extended version of RAG which involves incorporation of self-evaluation processes, enabling the model to constantly assess how well the retrieved information aligns with the user's need [17]. It can also automatically refine user queries to ensure the generated responses are both accurate and pertinent [17]. In implementation of Self-RAG, reflection tokens were generated and inserted into the output of a LLM and the LLM was trained to understand and use the tokens to incorporate self-reflection into its response generation process [15]. The self-reflection process allows the LLM to assess the relevance of the retrieved documents to user query to ensure that the retrieved documents are related to the context of user query. This can improve the performance of the LLM in generating correct, concise and correct responses. However, the self-reflection process may cause latency and increased computational cost as Self-RAG requires the generation of reflection tokens and there is possibility that the LLM will regenerate the response. In addition, training a LLM to generate, understand and utilize the reflection tokens effectively add complexity to the implementation of Self-RAG.

2.1.4 Multi-agent Chatbot Architecture

Gamage et al. [4] proposed a multi-agent chatbot architecture for Decision Support in net-zero emission energy systems, leveraging LLMs and RAG as shown in Figure 2.1.4.1. The architecture includes a Chatbot UI, NLU Module, Chatbot Core with four specialized agents namely Observer, Knowledge Retriever, Behaviour Analyzer and Visualizer each of which has specific task, as well as a Response Construction Module [4]. Based on the proposed architecture, Entity Extractor will first identify key entities and other attributes from user query to ensure that these entities are in uniform format understandable by the system [4]. Then, Intent Recogniser will match the input query to the most suitable agent to handle the query and the identified agent in Chatbot Core will be selected to answer the user query using the information obtained from databases [4]. Similar to [1], LangChain framework was employed to extract embeddings from databases [4]. The integration of LLMs with RAG in the paper allows the system to adapt to a wide range of queries and scenarios. By distributing different tasks among multiple agents, the chatbot can process and respond to user queries efficiently and accurately in a short period of time. In addition, the chatbot user interface (UI) was designed to produce graphical output, infographics and images as response to user queries but not just providing answer in text which was less informative and less interactive [4]. Based on the results in the paper, the architecture worked well but implementing it requires a lot of resources as running multiple agents simultaneously can be resource intensive. It requires very high computational power and large memory and the paper did not mention about it. Besides that, when it comes to scalability, adding and integrating new agents with different functionalities into the architecture may be significantly complex and scalability challenges were not mentioned in the paper.

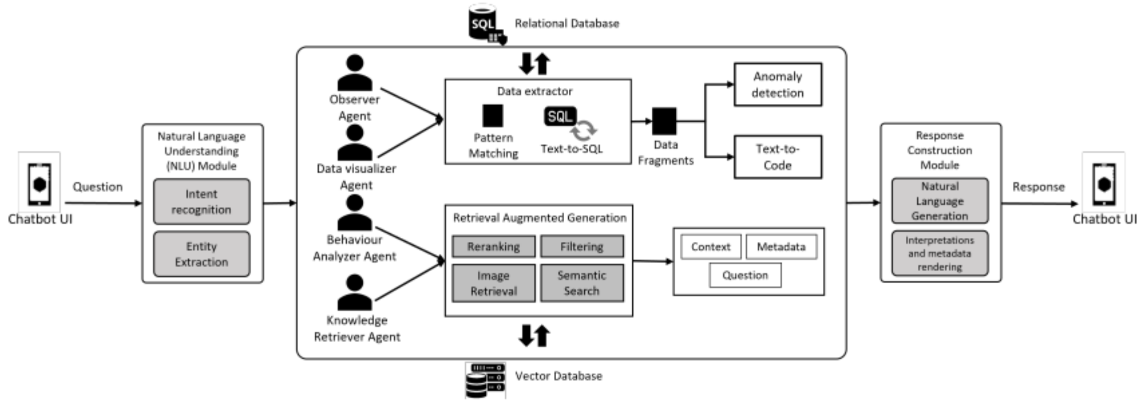


Figure 2.1.4.1 Multi-Agent Chatbot Architecture for Decision Support in Net-Zero Emissions Energy Systems [4].

Moreover, a paper introduced a chatbot framework called SMAG designed to support smoking cessation programs on social networks [5]. It details both single-agent implementation and multi-agent design to achieve a higher success rate in smoking cessation [5]. For single-agent chatbot implementation as shown in Figure 2.1.4.2, the engine core of the chatbot using the Flask framework, served through Nginx with uWSGI for parallel processing. The whole system is deployed using Docker and stores data in a PostgreSQL database [5]. Since the user will be asking query to the chatbot using Facebook Messenger, the chatbot was designed to communicate with the Facebook Messenger via JSON objects dispatched through GET/POST HTTP requests [5]. In the design of Multi-Agent SMAG, there were more than one agent and each agent was dedicated a specific knowledge to chat with users [5]. There was a gateway agent who will dispatch the incoming user messages to the agent that was related to the user [5]. In the multi-agent SMAG proposed, each agent can tailor to the specific needs of the users, providing personalized support and improving user engagement and satisfaction. The system is more robust and reliable as there will be other agents to chat with users even when one or more agents fail. On the other hand, the researchers of the paper did not implement and test the multi-agent SMAG but just proposed a design of multi-agent SMAG. Without any proof and experimental results, the paper did not provide convincing evidence about the effectiveness of multi-agent SMAG in real-world setting.

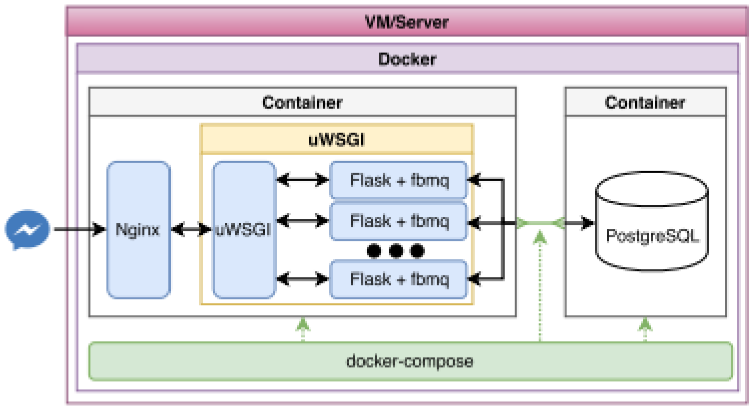


Figure 2.1.4.2 SMAG Single-Agent Deployment [5].

2.1.5 Exploring Agent Interaction Strategies in Multi-Agent Chatbots: One For All vs. Agent Selection

[6] tried to evaluate user preference on different types of interaction with AI chatbot by designing prototypes for each of the interaction types which included One For All and Agent Select for conversational AI and evaluate their ability to complete task. As shown in Figure 2.1.5.1, One For All prototype integrated a wide range of popular conversational agents which included Amazon Alexa, Google Assistant, SoundHound Houndify, and Ford Adasa and provided each of them with opportunity to respond to a user query [6]. Each agent first received the transcribed user query in text form and provide response to the user query [6]. The prototype employed a response ranking engine which will collect all the responses from the agents and select the best response based on semantic relation [6]. The best response was then converted back to speech and sent to user [6]. Another prototype, Agent Select allowed user to select agent to answer his query and hence response ranking engine was not needed [6]. The user interface of Agent Select is shown in Figure 2.1.5.2. Both prototypes have their own strengths and limitations. One For All prototype allows for automatic selection of the best agent to respond to user query but the user have to wait for all agents to provide response to the query and the best response has to be selected by response ranking engine based on semantic relation which is time-consuming and affects user experience badly. In the paper, there are agents such as Google Assistant and Amazon Alexa that covered a broad range of domains which may cause overlapping of domains and there will be a challenge to determine which agent is best suited for a specific task. On the other hand, Agent Select provides option of choice and flexibility for user to select an agent to answer his query, but the selected agent may not be the

best agent to answer the query in specific domain and may respond with “I do not have relevant knowledge” message and user will have to select other agents to get an appropriate answer.

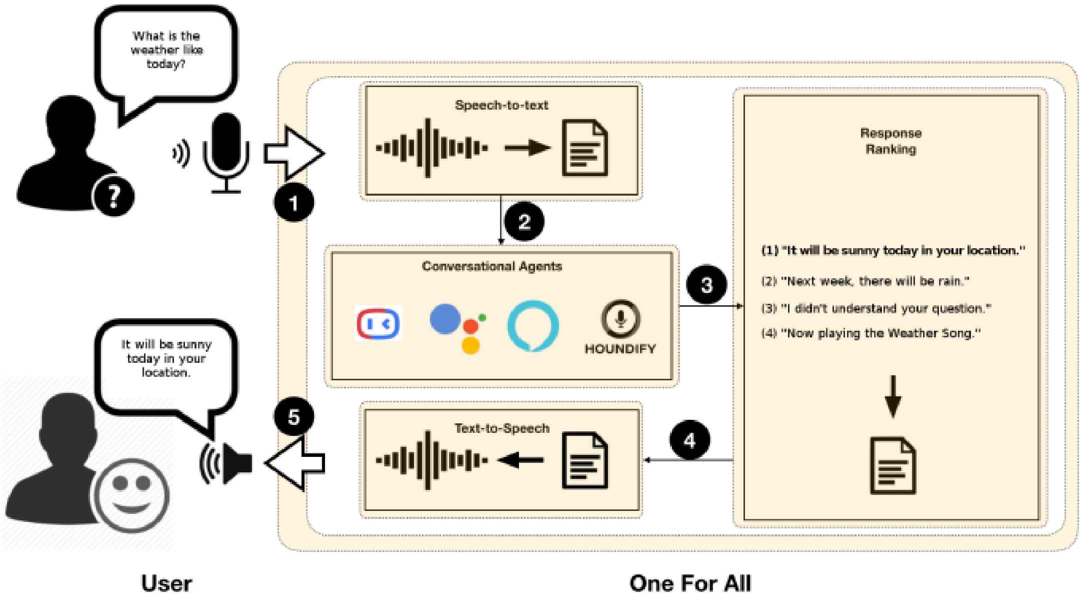


Figure 2.1.5.1 Architecture for One For All prototype [6].

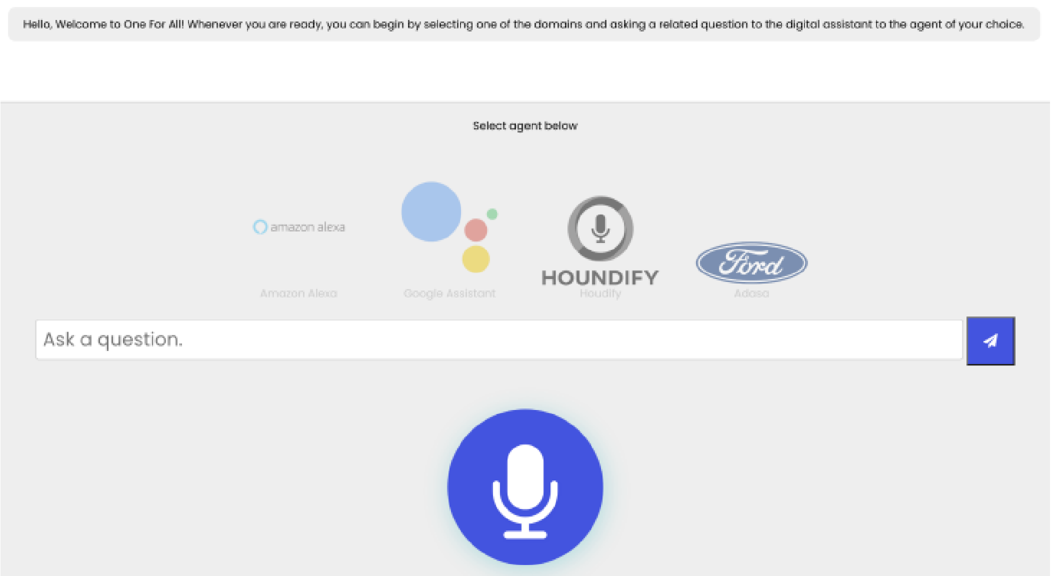


Figure 2.1.5.2 User Interface for Agent Select prototype [6].

2.2 Deployment Platforms

Firebase Hosting is a service commonly used by developers to deploy modern web applications online, especially single-page applications (SPAs) like those built with React. It comes with helpful features such as a global content delivery network (CDN), free SSL certificates, and smooth integration with other Firebase tools.

In practice, deploying a React frontend starts with the command npm run build, which creates an optimized version of the application inside the build folder. This version contains minified and bundled files that are smaller and load faster, making it suitable for production use [28]. To make these files available online, developers set up Firebase Hosting using the command firebase init hosting and select the build directory as the hosting folder. Firebase also supports custom configuration through a file called firebase.json. For SPAs like React, a rewrite rule is added so that any route will point back to index.html. This ensures that client-side navigation works correctly, even if users reload the page or access deep links directly [29], [30].

Once everything is configured, the final step is to run firebase deploy. This uploads the static files to Firebase servers, which then distribute them through Google's global CDN. As a result, the application becomes accessible worldwide with fast loading times and secure HTTPS by default [29]. This makes Firebase Hosting an appealing choice for frontend deployment because it requires little server management and is optimized for SPAs.

Render is a cloud hosting platform designed to simplify backend deployment. Unlike traditional hosting services where developers need to configure servers manually, Render automates much of the setup. It provides built-in SSL certificates, automatic scaling, and direct integration with Git repositories, which makes it easy to update applications whenever new code is pushed.

For a Flask-based backend, the deployment process is straightforward. Developers link their GitHub repository to Render, specify the environment such as Python, and provide commands to install dependencies which is commonly from a text file named requirements.txt and run the app. Render then builds and runs the application, automatically assigning it a live URL. This URL acts as the backend endpoint that frontend clients—such as the Firebase-hosted chatbot interface—can use to send requests [31], [32].

One of the main advantages of Render is that it removes the complexity of infrastructure management. Features like continuous deployment mean that developers do not need to manually redeploy the application each time changes are made; instead, updates happen automatically when code is pushed to the repository. This makes Render particularly useful for projects that are still being developed or frequently updated, as it reduces maintenance effort and speeds up the development cycle [33].

2.3 Proposed Solutions

In this project, a multi-agent RAG chatbot that uses API-based LLM namely OpenAI GPT-4o mini through OpenAI API call will be developed. API-based LLM eliminates the needs and costs in maintaining the hardware and infrastructure of hosting a LLM locally. API-based LLM is hosted in cloud platforms with extensive resources especially storage and computational power and users just have to pay per usage to use their well-trained, comprehensive LLM with high performance. Using API-based LLM also saves the time in developing the whole chatbot system by avoiding optimizing local LLM ourselves to improve performance. Some LangChain libraries will also be used to enhance the retrieval and response generation capabilities of the chatbot.

Conventional Retrieval-Augmented Generation (RAG) systems, utilizing a single agent, struggle to effectively handle numerous data sources [14]. These systems often suffer from inefficiencies like exceeding token limits and generating imprecise queries, particularly when dealing with various data storage formats [14]. Self-RAG was said to be better than RAG for LLM to generate more relevant responses to user query. However, RAG is chosen for implementation of this project but not Self-RAG. The reason is because the project aims to develop a multi-agent chatbot system with each agent representing a department of division in UTAR rather than a chatbot system with single agent. Since each agent has its own knowledge base, Self-RAG has less benefits in this case and this also reduces the complexity, time and costs of the project in implementing Self-RAG. Firebase Hosting is chosen as the platform to deploy the frontend of the chatbot because it requires only a little server management, and it is optimized for SPAs. The chatbot backend was built using Flask and so Render is used to deploy the chatbot backend as the process requires less effort compared to other platforms.

Chapter 3

System Methodology/Approach

3.1 Project Methodology

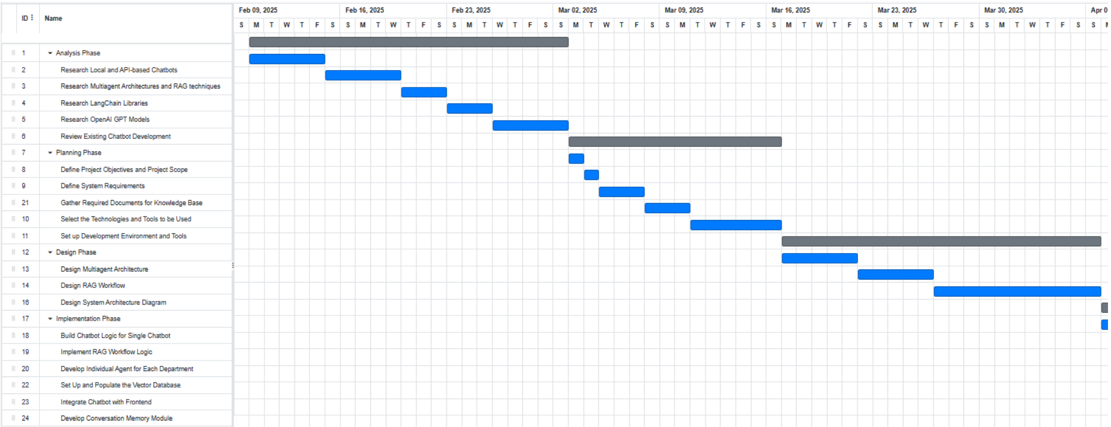


Figure 3.1.1 Gantt Chart for Final Year Project 1 Part A

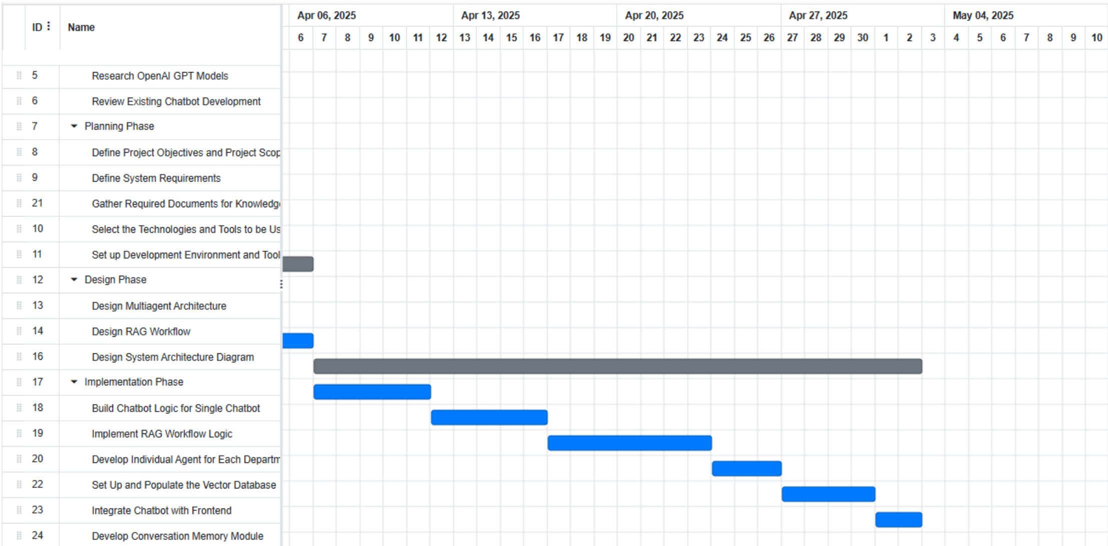


Figure 3.1.2 Gantt Chart for Final Year Project 1 Part B

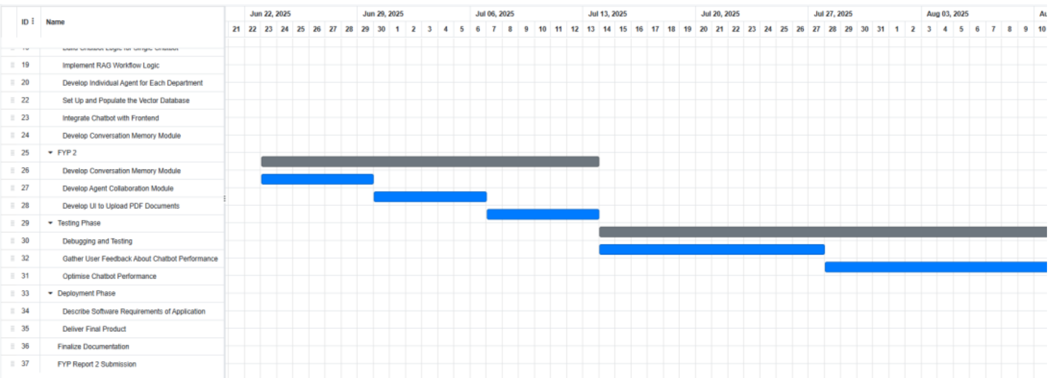


Figure 3.1.3 Gantt Chart for Final Year Project 2 Part A

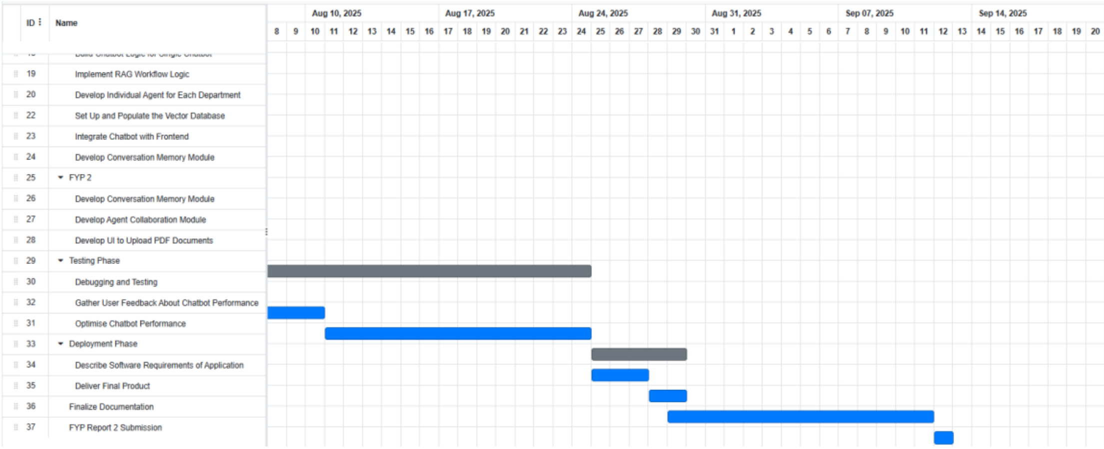


Figure 3.1.4 Gantt Chart for Final Year Project 2 Part B

Figure 3.1.1 shows the Gantt Chart for Final Year Project 1 and Figure 3.1.2 shows the remaining and incomplete parts of Gantt Chart for Final Year Project 1. Figure 3.1.3 shows the Gantt Chart for Final Year Project 2 and Figure 3.1.4 shows the remaining and incomplete parts of Gantt Chart for Final Year Project 2. Gantt Chart is used to represent the project timeline and schedule different tasks to be completed for Final Year Project 1 and Final Year Project 2.

During the analysis phase, extensive research was conducted from online sources such as online journal articles and online websites as well as getting extra knowledge, information and advice from lecturers and seniors. Research was conducted to find out the solutions that better suited stakeholders' needs, capabilities and this project. After searching for some significant amount of online sources, the research papers were studies and the proposed solutions in the papers were reviewed to determine the advantages and weaknesses of the solutions. The papers used were ensured to be five years ago but not more than five years. The purpose is to ensure exposure to the latest technology and trend but not considering an outdated solution, method or technology.

During analysis phase, research was conducted to determine whether to use local LLM or API-based LLM, multi-agent architecture, RAG techniques, LangChain libraries which could be used to simplify chatbot development, OpenAI GPT models which were API-based LLM and other existing chatbot development. In the planning phase, the project objectives, scope and system requirements were defined. The technologies and tools to be used were selected and development environment and tools were set up and prepared for development later. In the design phase, the architecture for the multi-agent RAG chatbot system and RAG workflow were designed. In the implementation phase, the development of the chatbot began by building chatbot logic, developing agents with databases, connecting frontend of the chatbot system with backend and developing additional modules. The development of conversation memory module was attempted by me in Final Year Project 1 but has not been completed so the development will be continued in Final Year Project 2.

For Final Year Project 2, the development of additional modules which was conversation memory module which can handle follow-up questions by users has been completed while the agent collaboration module which allows multiple agents to collaborate in answering a user query that involves different departments or divisions of UTAR was not implemented due to time constraints. The agent collaboration module can be implemented in the future. A UI designed for data owners or system administrators to upload PDF documents and restart the chatbot system to use the latest PDF documents to answer user queries was also not implemented but the chatbot has been deployed and it supported uploading of zipped vector databases files with smaller size to GitHub for deployment. The backend deployment platform which was Render allowed data owners or system administrators to view the logging of user interactions with the chatbot. After implementation phase, various tests have been performed to test and evaluate the chatbot system as well as gathering user feedback about the performance of the chatbot and accuracy of the responses provided by letting users to use the chatbot and filling out a Google survey form. After that, optimizing the chatbot performance was carried out before delivering the final product by various methods such as reducing the lines of code to perform an action, comparing and changing the base LLM model to other models with better performance. The next phase was the deployment phase which involved describing the software requirements of the chatbot system such as libraries required to run the chatbot which can be found in text file named requirement.txt. Finally, the final product will be delivered and the documentation related to the chatbot system will be finalized. The Final Year Project 2 Report will also be submitted.

3.2 System Design Diagram

3.2.1 System Architecture Diagram of UTAR Multi-agent RAG Chatbot

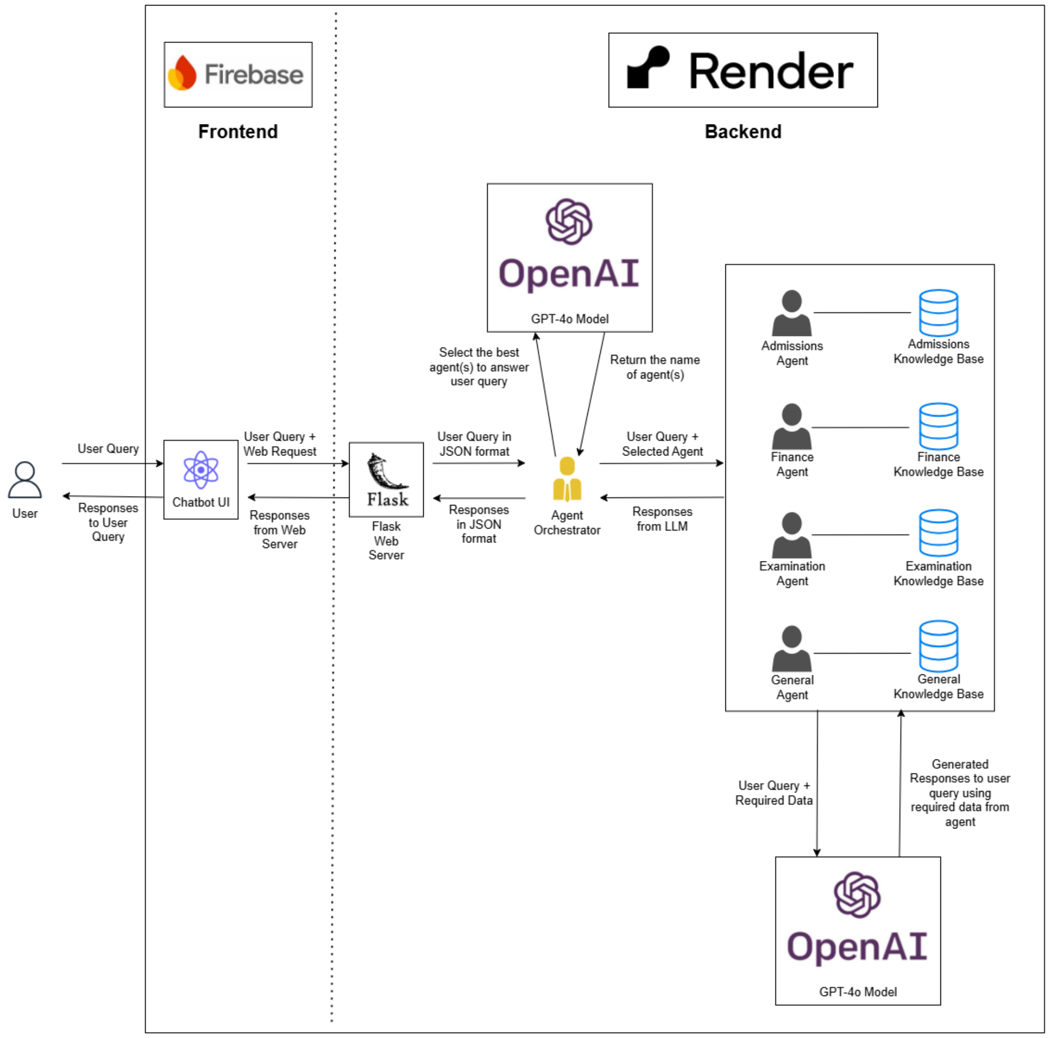


Figure 3.2.1.1 System Architecture Diagram of UTAR Multi-agent RAG Chatbot.

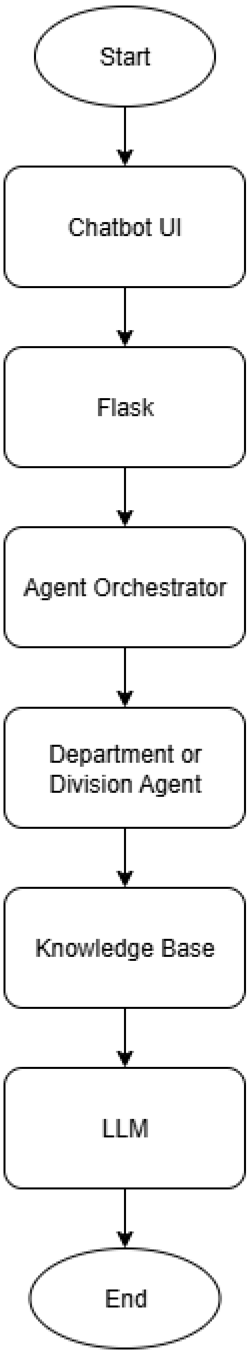


Figure 3.2.1.2 System Architecture Diagram of UTAR Multi-agent RAG Chatbot.

Figure 3.2.1.1 shows the system architecture for the multi-agent RAG chatbot developed for UTAR for user query resolution. Figure 3.2.1.2 shows the simplified pipeline of the chatbot. The frontend of the chatbot system is hosted on Firebase while the backend part is deployed to Render. A user begins interaction with the chatbot system by typing a question and sending the question to the chatbot UI. After the chatbot UI receives the input from the user, it forwards the user's question as a request to the Flask backend. The chatbot UI is built with React app and the React app uses fetch() to make a POST request to the Flask API endpoint which is configured to be /chat endpoint in this case. The request is in JSON format, and it contains the user question. The Flask backend server receives the request at /chat endpoint, and it forwards the request to the agent orchestrator which is the first agent the Flask will be interacting with. The agent orchestrator processes the request and understands the user's question and selects the best agent relevant to the query to generate responses. The chosen specialized agent representing a department or a division will get the user query. The specialized agent processes and understands the user query and retrieves required information from its knowledge base which is a vector database containing information and data about a particular department or division of UTAR. The retrieved contextual data and the user query are passed to the OpenAI GPT-4o model to generate responses to user query based on the retrieved context by API call. The generated response will then be sent back to the agent orchestrator and the agent orchestrator further forward the response in JSON format to the Flask backend server. The responses in JSON format finally reach the React app. The React app then updates the chatbot UI by appending the chatbot's response and the response is displayed to user in chatbot UI.

3.2.2 Use Case Diagram and Description

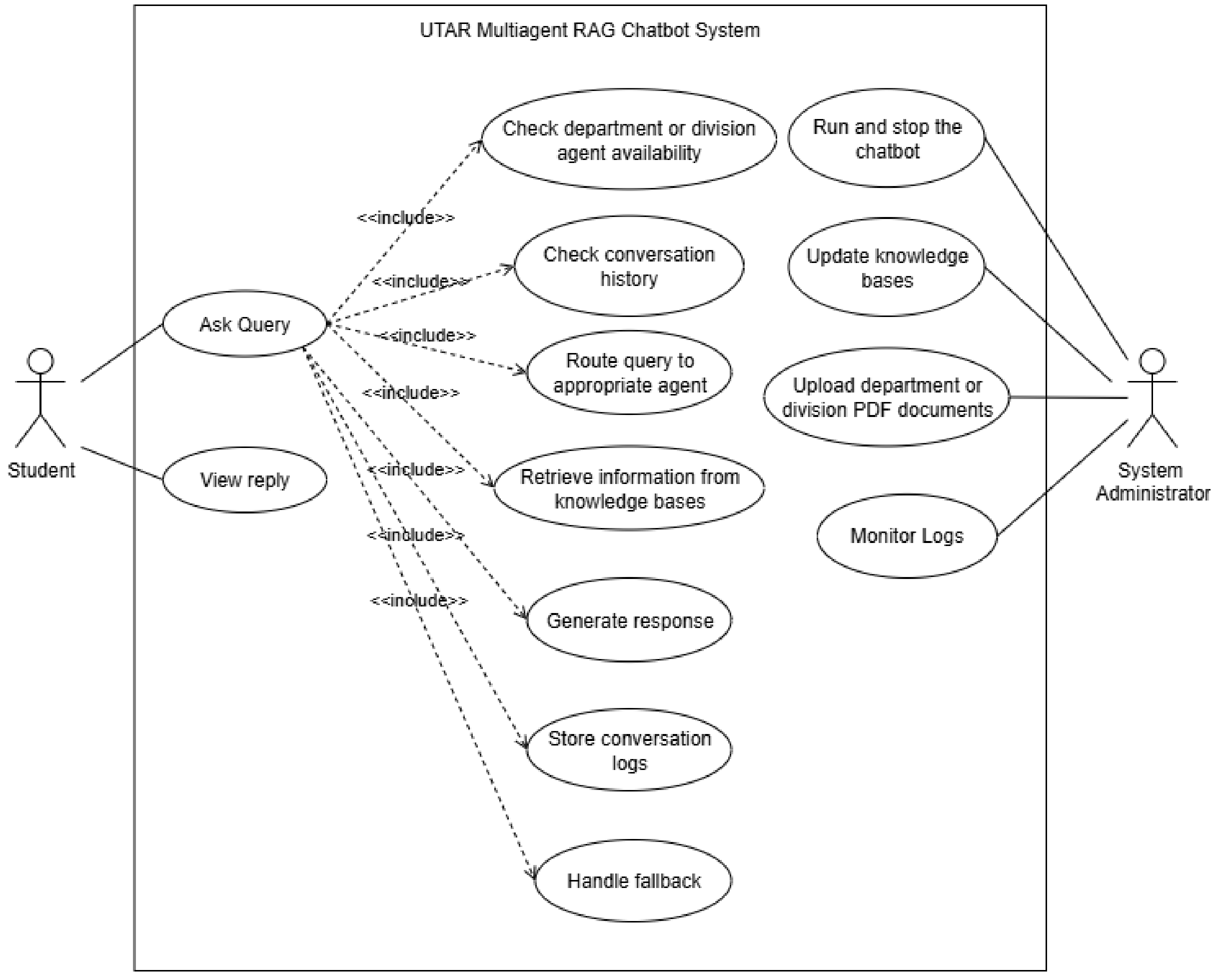


Figure 3.2.2.1 Use Case Diagram for UTAR Multiagent RAG Chatbot System.

3.2.2.1 Actors

Actors	Descriptions
Student	The main user of the chatbot
System Administrator	The system maintainer who will be maintaining the chatbot system

Table 3.2.2.1.1 Actors and their descriptions for UTAR Multiagent RAG Chatbot System.

3.2.2.2 Use Cases

Use Cases	Descriptions
Ask Query	Student submits a query through the React chatbot UI hosted on Firebase Hosting.

View reply	Student receives an automated response generated by the chatbot after retrieval from the respective knowledge base of department or division agent.
Check department or division agent availability	System checks if the correct agent is available to handle a query. If the correct agent is available, the query is routed to the agent, otherwise, the query is directed to the General Agent.
Check conversation history	System searches stored past interactions such as user queries and generated responses in a specific session to maintain context for ongoing conversations.
Route query to appropriate agent	Based on the intent of a query, the system specifically the agent orchestrator forwards the question to the appropriate agent for processing.
Retrieve information from knowledge bases	Each specialized agent has its own knowledge base that contain information and data. It will retrieves information and data from its knowledge base to be used in generating responses
Generate response	Using retrieved context from specialized agent, the system with base model OpenAI GPT-4o model generates natural and human-like responses to user queries.
Store conversation logs	Every user interaction with the chatbot system is stored securely at backend for performance tracking, error analysis and future improvements.
Handle fallback	If the system cannot find an answer, it politely responds with fallback options by

	telling the students to refer to and contact the relevant departments or divisions in UTAR.
Run and stop the chatbot	System administrator can initiate or terminate the chatbot service.
Update knowledge bases	System administrator recreates the vector database for every department and division agent using updated information and data from UTAR webpages and UTAR PDF documents to ensure students receive accurate and up to date responses.
Upload department or division PDF documents	System administrator uploads PDF documents related to admission, examination and finance departments in UTAR.
Monitor Logs	System administrator reviews system activity, including chatbot interactions, errors and usage statistics and even conversational history for each specific user session.

Table 3.2.2.2.1 Use cases and their descriptions for UTAR Multiagent RAG Chatbot System.

3.2.3 Activity Diagram

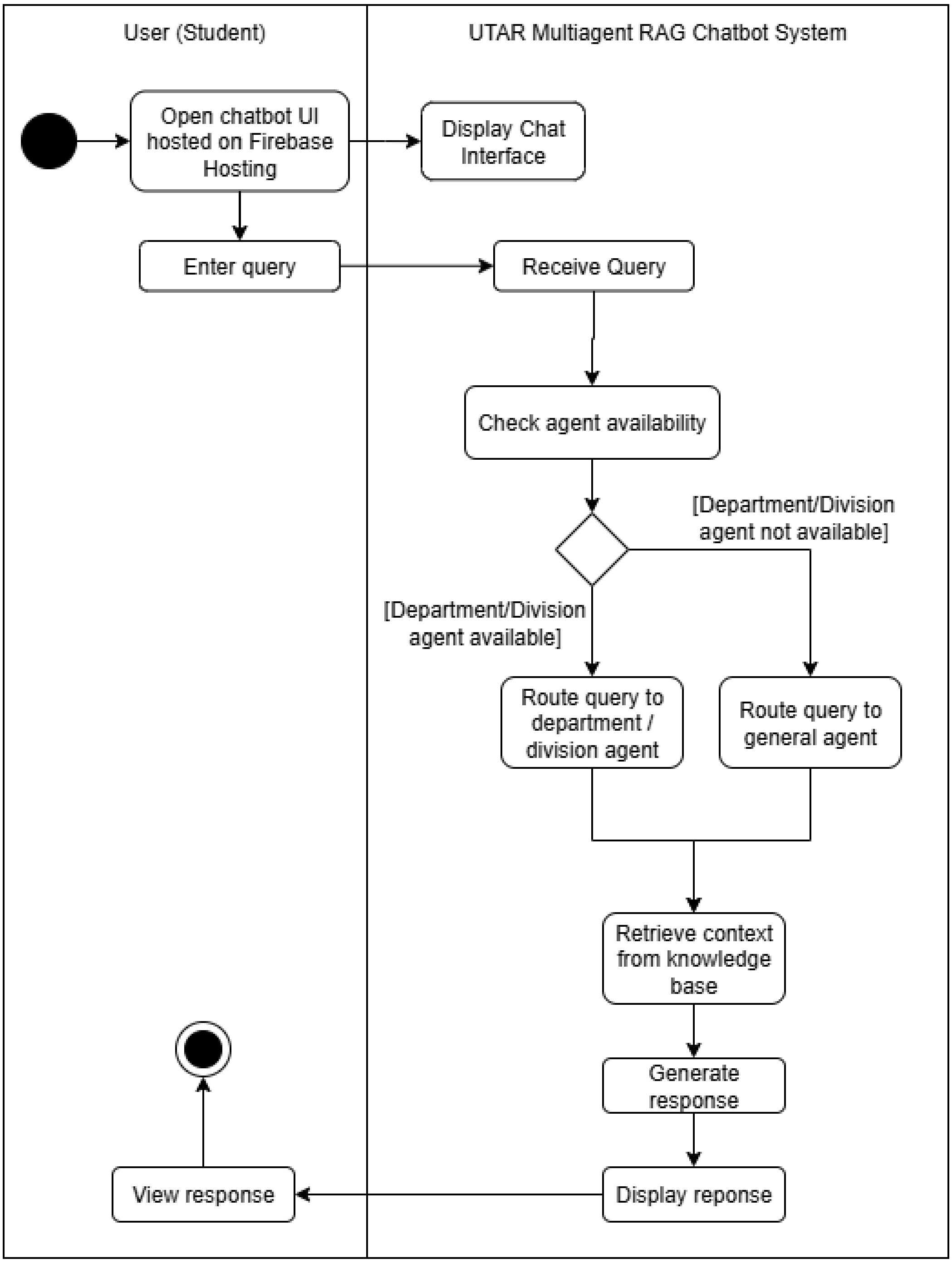


Figure 3.1.2.1 Activity Diagram for UTAR Multiagent RAG Chatbot System.

A user initiates the interaction with the chatbot by opening the chatbot UI hosted on Firebase hosting and the system renders and displays the chat interface by greeting the user and providing the user with input section to type his query and send button to submit the input query. The user types a query and clicks ‘Send’ button to submit the query to the chatbot UI. The chatbot UI receives the query and sends the query to the chatbot backend. The chatbot backend processes the query and checks for agent availability to answer the query. If relevant agent is available to answer the query, the query is routed by the agent orchestrator to the agent. On the other hand, if the relevant agent is not available, the query is routed to General Agent. When an agent receives the query, it will retrieve contextual information and data from its vector database or knowledge base to be used to generate response to the user. Based on the conversational history and the retrieved information, a response is generated by the agent and is sent to the chatbot UI. The chatbot UI then displays the generated response to the user. The user is now able to view the response displayed in chatbot UI.

Chapter 4
System Design

4.1 System Block Diagram

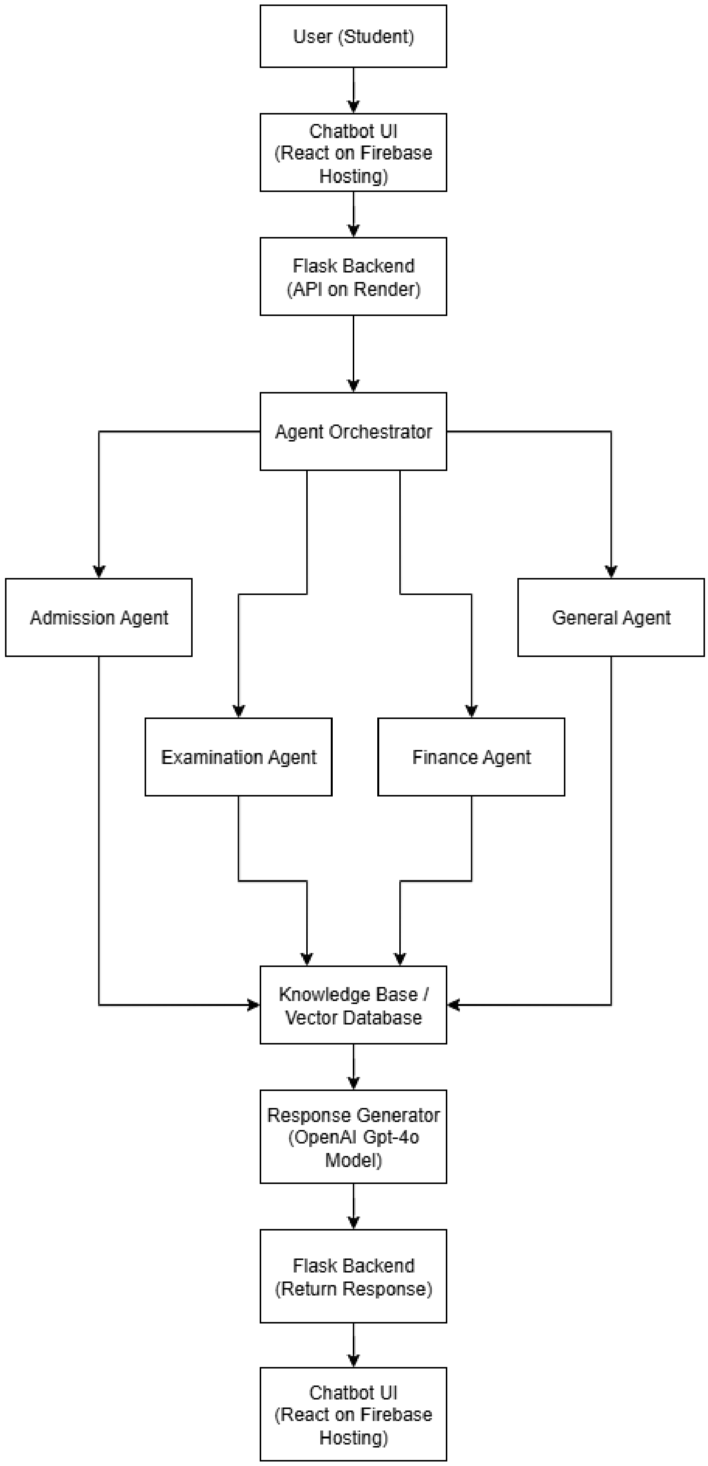


Figure 4.1.1.1 System Block Diagram of UTAR Multiagent RAG Chatbot.

4.2 Chatbot System Component Descriptions

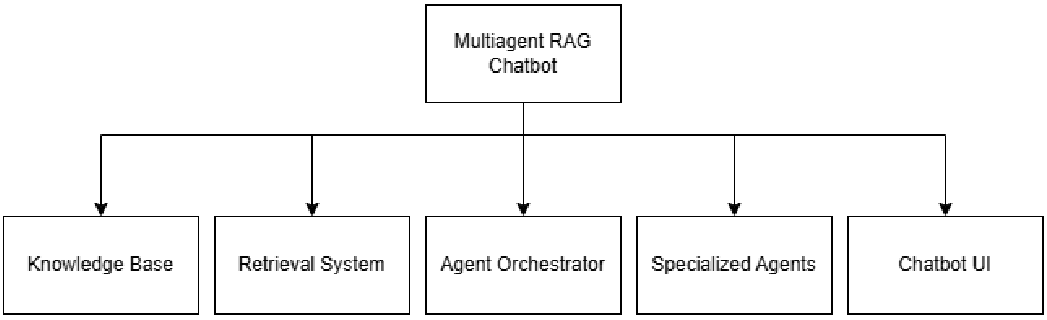


Figure 4.2.1 Five Components of UTAR Multiagent RAG Chatbot

The chatbot system consists of five main components which are the knowledge base, retrieval system, agent orchestrator, specialized agent and user interface as shown in block diagram in Figure 4.2.1.

The system workflows will be described. Firstly, a user sends a query or question to chatbot UI. The chatbot UI receives the user query and sends it to the web server which will parse the user query into JSON format and send it to agent orchestrator. Agent orchestrator receives and analyses user query and determines which specialized agent should handle the query. The selected agent will retrieve the relevant and related information from respective knowledge base using the retrieval system. The agent then processes the retrieved information and uses the information to generate responses. The agent orchestrator receives and combines the responses if multiple agents are involved in answering the query. The final response is then delivered to the user via the chatbot UI.

4.2.1 Knowledge Base

Knowledge base is defined as a centralized repository of information [7]. Each of the agents with defined scope and domain is designed to retrieve specific information or data from a knowledge base that stores information about departments and faculties in UTAR such as Faculty of Information and Communication Technology (FICT). Knowledge base here is implemented as vector database and is designed to deal with semantic search and natural language processing tasks. With implementation of vector database, the knowledge base stores and manages data in the form of vector embeddings which are numerical representations of text or other data types [8]. The distance between two vector embeddings corresponds to the semantic similarity between two input texts or data in their original form [8]. This is useful for searching of contextual information required to answer user queries.

Knowledge base plays an important role in the chatbot system to provide each agent with specific information to answer user queries. The creation of a knowledge base will be described later. First of all, PDF documents containing information about departments and divisions of UTAR are collected in a local device. The web scraping module checks whether specific UTAR webpage contains PDF documents. If PDF documents exist in the webpage, the PDF documents are downloaded to the local folder representing a UTAR department or division in the same local device. The web scraping module also extracts and stores readable text and links from specific UTAR webpage. UnstructurePDFLoader from LangChain is used to extract the text from each PDF document and RecursiveCharacterTextSplitter which is also from LangChain is used to split the extracted text from PDF documents and UTAR webpages into chunks with maximum size of 1500 characters and chunk overlap of 200 characters to reduce the loss of context. Chroma database which is a vector database is then created for each specialized agent using LangChain libraries, the embedding model used is OpenAI embedding model known as text-embedding-3-large. The chunks of extracted text from PDF documents of a department or division and extracted readable text content from UTAR webpages will be converted into vector embeddings and stored in the Chroma database of agent representing the department or division.

After creation of all vector databases, the loading of vector databases is implemented as lazy loading of vector databases. Lazy loading means the vector database for a UTAR department or division is loaded for the first time only when the department or division agent first receives a user query. This lazy loading of vector databases prevents the issues of memory exhaustion at startup. The reason is because vector databases consume large memory sizes and loading all vector databases together at startup may cause RAM memory exhaustion and timeout issues especially during deployment.

4.2.2 Retrieval System

The retrieval system in the chatbot system is important in searching for information that is relevant and related to the context of user queries from knowledge bases. When a user sends a query or question to the chatbot UI, the query is converted to vector embeddings using the same embedding function as the OpenAI embedding model used to create the Chroma database which is text-embedding-3-large which are mathematical vector calculations and representations [9]. A single or multiple agents suitable to answer the user query will be selected. The selected agents will perform similarity search to search for documents with context relevant to user query by comparing the vector embeddings of the user query with the vector embeddings in their knowledge bases which are vector databases. The retrieved context information will then be used in generation of responses to the user.

4.2.3 Agent Orchestrator

Agent orchestrator is the first agent that will receive the user query and its base LLM is OpenAI GPT-4o model. The parameters named temperature will be set to 0.0 for more deterministic result and max_tokens will be set to 10 so that the LLM only return a short response.

```
prompt = f"""You are a router that determines which university agent should handle a user query.

Available agents:
{agent_info}

User query: "{query}"

ROUTING INSTRUCTIONS:
1. Examine both the TOPIC and CONTEXT of the query carefully
2. Look for department-specific keywords and subjects (admissions, finance/fees/scholarships, exams/courses)
3. If the query relates to a department's core responsibility area, route to that department EVEN IF some terms are unfamiliar
4. Examples of routing logic:
- Questions about exam procedures, exam rules, exam requirements, or anything happening during exams → Department of Examination and Awards
- Questions about admissions process, applications, entry requirements → Division of Admissions
- Questions about fees, payments, scholarships, financial aid → Division of Finance
5. Only route to the General Agent if the query clearly doesn't relate to the core responsibilities of any specialized department

Based on these instructions, respond ONLY with the appropriate agent designation (e.g., "Agent 1", "Agent 2", etc.) without any explanation.
"""
```

Figure 4.2.3.1 Prompt for Agent Orchestrator.

Figure 4.2.3.1 shows the prompt that will be passed to the base LLM by agent orchestrator to determine the suitable agent which can answer the user query based on the description of each specialized agent. Upon receiving the user query or question, agent orchestrator will identify the intent of the query and select one most suitable agent who can answer the query based on agents' descriptions. After selecting the agents, the agent orchestrator will send API request to the selected agents to retrieve information from their respective knowledge bases, generate responses based on the retrieved information and send to the agent orchestrator. The agent orchestrator will then aggregate the responses from all selected agent, resolve the conflicts between the responses and generate a final response to user.

4.2.4 Specialized Agents

The chatbot system employs multiple specialized agents, each is equipped with a knowledge base that stores specific knowledge about a division or department of UTAR. After receiving API requests from the agent orchestrator, the selected agents will retrieve relevant information from their knowledge bases, generate responses and send to the agent orchestrator. The development of multi-agent system begins with four agents being developed. After the whole chatbot system is completed, more agents representing other divisions and departments of UTAR will be added into the chatbot system.

4.2.4.1 Types of Specialized Agents

Agent Types	Agent Description	Division or Department Name
Admissions Agent	Handles admissions-related queries	Division of Admissions and Credit Evaluation
Finance Agent	Handles finance, fees, and scholarship queries	Division of Finance
Examinations Agent	Handles course and exam queries	Department of Examination and Awards
General Agent or University Information Assistant	General knowledge about the university	None

Table 4.2.4.1.1 Types of specialized agents, their descriptions and their divisions or departments.

Table 4.2.4.1.1 shows the four specialized agents that have been developed initially. The first agent is the Admissions Agent which will be dealing with queries related to admissions of UTAR such as admission criteria and admission procedure. Its knowledge base contains information related to admissions. PDF documents from Division of Admissions and Credit Evaluation of UTAR and related to admissions of UTAR and information on webpages of the division will be used as the knowledge source for Admissions Agent.

The second agent is the Finance Agent which will be dealing with queries related to finance, tuition fees and scholarships. PDF documents related to finance and Division of Finance and information on webpages of the division will be used as the knowledge source for Finance Agent.

The third agent is the Examinations Agent which will be dealing with queries related to courses and examinations. PDF documents related to course and Department of Examination and Awards and information on webpages of the department will be used as knowledge source for the agent.

The fourth agent is the General Agent which is equipped with the general knowledge about UTAR. It serves as the last resort agent to query when there is no any suitable agent to answer the user query. PDF documents containing the general information about the university such as the history of the university are used as knowledge source for the General Agent.

4.2.4.2 Agent Implementation

In implementation of multi-agent system, a base agent class is first defined. Each agent class has attributes: name which is the name of the agent, description that describes the specific domain of the agent, vector_db which is the knowledge base for the agent, vector_db_path which is the location of the knowledge base in a computer file system and department which is the department that the agent is responsible for. Each base agent class has an attribute named urls which is a list of URLs of UTAR webpages related to the division or department to specify which webpages to scrap for information, data and PDF files for the division of department. The base agent class has methods to initialize each specialized agent, extract the contents from PDF documents, split documents into chunks, create and load vector database, retrieve contextual information from knowledge base and generate responses.

Every specialized agent class then inherit the attributes and methods from the base agent class. Each specialized agent is known as subclass while the base agent class is known as superclass. Each specialized agent is initialized with different values as each specialized agent has different name, description, vector database, path to the vector database and department. Each specialized agent overrides the method to generate responses based on the retrieved information from its knowledge base. The overriding involves accessing to OpenAI GPT-4o model using OpenAI API, combining all retrieved contexts, passing a prompt to ask base LLM model which is OpenAI GPT-4o to perform response generation to user query using the combined context.

The code snippets for the implementation of base agent, admissions agent, finance agent, examination agent and general agent classes are shown in Appendix A.

4.2.5 Chatbot User Interface

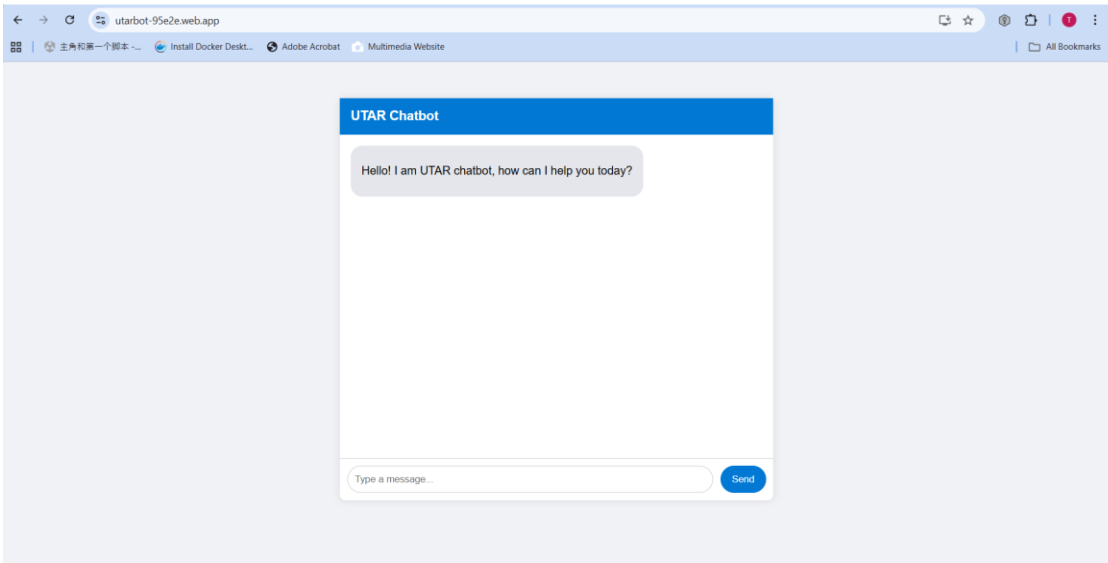


Figure 4.2.5.1 UI of UTAR multi-agent RAG chatbot.

Figure 4.2.5.1 shows the simple UI for UTAR multi-agent RAG chatbot. The UI provides user with a text input section to type query or question and a button to send the query to the chatbot.

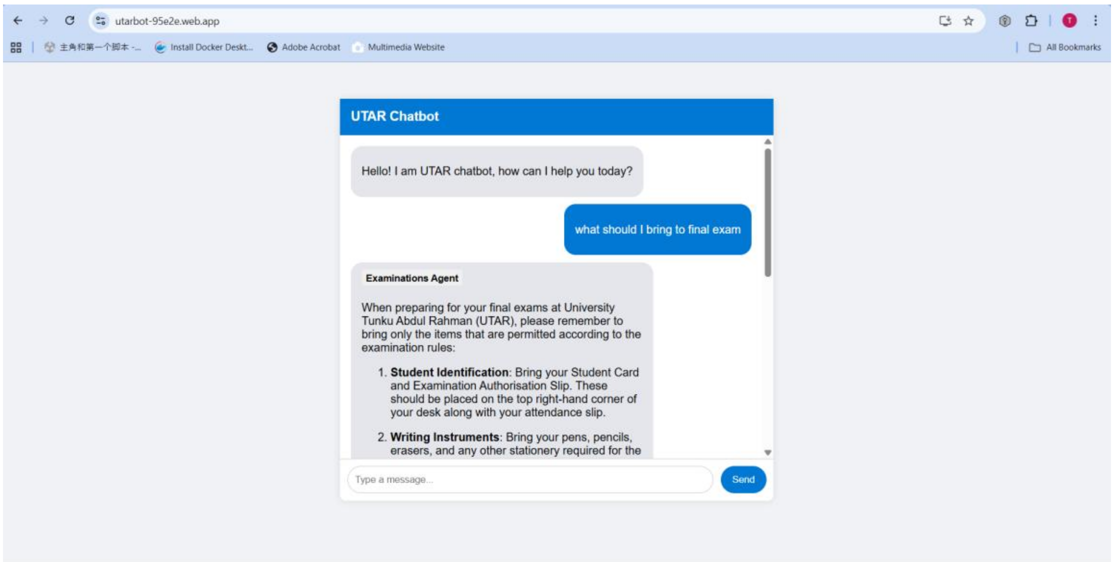


Figure 4.2.5.2 UI of UTAR multi-agent RAG chatbot with question and generated responses.

As shown in Figure 4.2.5.2, when user first enters to the web page of the chatbot, the chatbot will greets the user and user can start typing query or question in the text input section and click the button to send the query to the chatbot. The chatbot will then receive and process the query and generates the responses. The responses are then sent to the chatbot UI to be displayed to the user. The chatbot UI will display the generated response, the name of agent which is responsible for answering the question and references that is used in generating response right after the response text.

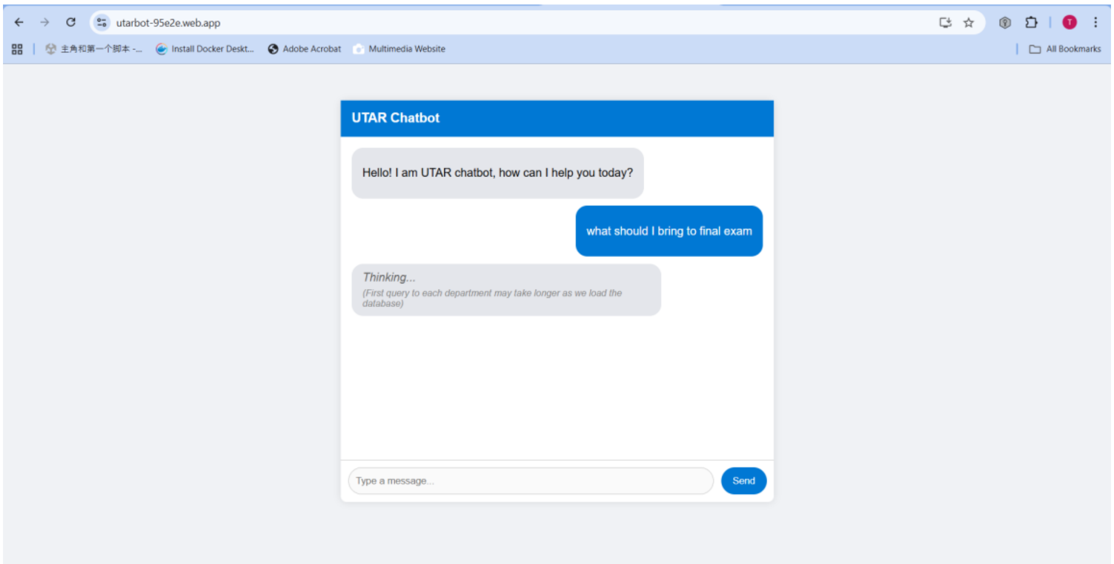


Figure 4.2.5.3 UI of UTAR multi-agent RAG chatbot with question and generated responses.

As shown in Figure 4.2.5.3, when the user query is being processed, the chatbot UI displays the status message to user to inform the user that the chatbot backend is currently processing the request. Since the deployed chatbot employs lazy loading of vector database for each specialized agent, the chatbot UI also informs that user that the first query to a department of division require extra time to load vector database before using the data from vector database to generate response to the user query.

4.3 Chatbot Module Descriptions

4.3.1 Web Scraping Module

The web scraping module in this project was designed to automatically collect relevant textual information from specified UTAR's webpages. This was implemented using Playwright, a modern headless browser automation tool which supports the interaction with JavaScript [19], combined with BeautifulSoup for HTML parsing. In this case, BeautifulSoup is used to navigate and parse the HTML content to be extracted later [20]. Most UTAR webpages contained content generated by JavaScript, so Playwright is efficient and useful in this case. Playwright is a modern headless browser automation tool that can perform actions similar to human interaction with modern websites such as clicking buttons, scrolling or filling out forms but Playwright can operate without a visible graphical user interface (GUI) [19]. The scraping process begins by launching a Chromium browser, navigating to and loading each specified URL, and waiting a few seconds to ensure that JavaScript content has fully rendered. Once the page is loaded, the HTML and Document Object Model (DOM) are captured by Playwright and are parsed by BeautifulSoup, and text is extracted from key sections of the parsed HTML and DOM content such as <div> elements with the class mg as the class mg was found to contain the important information about the UTAR webpage such as text content, JavaScript element and PDF files.

Additionally, all hyperlinks on the page are collected and filtered to ensure uniqueness, as many institutional websites often contain repetitive or nested links. The text content with class a in <div> element is extracted to get unique links that can be found on the webpage. The scraper prioritizes capturing both main textual content and relevant links so that the resulting dataset is rich in context. This ensures that the knowledge base includes both direct content and navigation cues to additional resources. By leveraging modern tools such as Playwright and BeautifulSoup, important information and data required to construct vector database for each department or division of UTAR can be extracted effectively from dynamic webpages compared to traditional static scrapers.

4.3.2 PDF Download Module

Since many university resources such as guidelines, policies, and forms are distributed in PDF format, a PDF download and ingestion module is developed to automatically scan scraped webpages for linked PDF documents and store them in structured departmental folders. The downloading process uses the Requests library with SSL handling, ensuring that broken certificates on some UTAR pages do not disrupt data collection. There is an additional challenge addressed in this module which is the presence of SSL certificate issues on UTAR's domain for certain PDF files. To handle this, the downloader bypasses SSL verification for UTAR-hosted PDFs while still enforcing standard certificate checks for external sources. This ensures reliable downloads without compromising security when dealing with external content. Once downloaded, the PDFs are parsed using UnstructuredPDFLoader, which extracts text while preserving metadata such as file source and department. This approach is consistent with recent systems that combine automated PDF scraping with content extraction for academic knowledge bases [21]. In line with best practices, the module also ensures file uniqueness, error handling, and incremental updates, which improves robustness. By integrating the scraped PDFs into the knowledge base, this module supports downstream retrieval and enhances the chatbot's ability to answer queries that depend on official university documents.

4.3.3 Conversational Memory Module

To ensure that the chatbot can maintain coherent and context-aware interactions, a conversational history module was developed. This module allows the system to recall previous conversation exchanges and integrate them into future responses, thereby making interactions feel more natural and personalized to users. The conversational memory was implemented using short-term memory. The chatbot maintains conversation continuity across sessions by storing recent interactions within the user's session [22]. When a student submits a query, the system retrieves the existing conversational history in current session or creates a new one if none exists. Each query is appended to this history, followed by the chatbot's response, ensuring that both sides of the dialogue are preserved. This module allows students to ask follow-up questions by enabling the chatbot to retain important user details across sessions [1].

To optimize performance and prevent excessive memory usage, only the last six exchanges of conversational history are retained in the session. This short-term memory is managed using Flask's built-in session mechanism, which, in this project, is configured with filesystem-based storage. As a result, the chatbot is able to recall and retrieve context during an active session, enabling more coherent and context-aware responses, although the memory is temporary and does not persist once the session ends. In order to support context-aware conversations, the chatbot incorporates a session-based design for managing user interactions. When a student submits a query through the React frontend, the request is forwarded to the Flask backend, which processes the query and simultaneously maintains a session record of the conversation.

This session data is stored temporarily in the server's filesystem (.flask_session/), allowing the chatbot to recall the last few conversational exchanges during the ongoing session. By retrieving and updating this history for each query, the chatbot is able to generate more coherent and relevant responses. Once the session ends or the server is restarted, the stored conversational history is cleared, ensuring both privacy and efficient storage management. This approach provides short-term conversational memory while avoiding the complexity of long-term data persistence.

Chapter 5

System Implementation

5.1 Software Setup

5.1.1 Flask Backend Setup

```python

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}}, supports_credentials=True)


app.secret_key = os.environ.get("FLASK_SECRET_KEY") or os.urandom(32)

app.config['SESSION_TYPE'] = 'filesystem'
app.config['SESSION_PERMANENT'] = False
app.config['SESSION_FILE_DIR'] = './.flask_session/'
app.config['SESSION_USE_SIGNER'] = True
app.config['SESSION_COOKIE_SAMESITE'] = "None"
app.config['SESSION_COOKIE_SECURE'] = True


Session(app)
```

Figure 5.1.1.1 Code snippet of Flask Backend Setup.

Figure 5.1.1.1 shows the code snippets used to setup the backend part of the UTAR multiagent RAG chatbot. The backend part of the chatbot was developed using Flask framework. Flask framework was used to setup endpoints of a web server to listen to incoming requests in the endpoints [5]. The backend setup began with the initialization of a Flask application instance and the integration of Cross-Origin Resource Sharing (CORS) to allow React-based frontend hosted on Firebase to send requests to the backend. The reason is because the backend and frontend parts of the chatbot system were deployed in different platforms. The backend part of the chatbot was deployed to Render while the frontend part of the chatbot was hosted on Firebase hosting so CORS must be used to allow the communication between two different platforms [23].

Besides CORS configuration, a secret key was configured for session management to ensure secure handling of user data. The session storage was implemented using the server's filesystem which is the line of code that shows SESSION_TYPE = 'filesystem' [24], [25]. Session signing which was SESSION_USE_SIGNER = True was enabled for added security of the session [24]. The SESSION_FILE_DIR = './.flask_session' allowed the chatbot to store the conversation histories in the .flask_session/ temporarily [26] to maintain the previous conversational exchanges so that users can ask follow-up questions. This configuration of Flask Session ensures that the conversational histories are tied to the individual session to provide personalized responses to users while maintaining users' privacy.

5.1.2 React Frontend Setup

```
const [messages, setMessages] = useState([
  { role: 'assistant', content: 'Hello! I am UTAR chatbot, how can I help you today?' }
]);
const [input, setInput] = useState('');
const [loading, setLoading] = useState(false);
```

Figure 5.1.2.1 Code snippet that shows initialization of the chatbot state.

Figure 5.1.2 above shows the segment of code in App.js which was chatbot frontend developed in React to initialize the chatbot's state. The state named messages stores the entire conversational histories for a particular user session. The state named input stores the message of current user which is being typed while the loading flag is used to update the status of the chatbot to users to indicate that the system is waiting for the backend part of the chatbot to respond. When users load the chatbot frontend developed in React, the chatbot will initialize the conversation by greeting the users with warm welcoming message and this creates a friendly starting point for the users.

```javascript

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

    const userMessage = { role: 'user', content: input };
    setMessages((prev) => [...prev, userMessage]);
    setInput('');
    setLoading(true);

    try {
      const res = await fetch('https://utarbot.onrender.com/chat', {
        method: 'POST',
        credentials: 'include',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          question: input
        })
      });

```

Figure 5.1.2.2 Code snippet that shows sending of user queries to the Flask backend.

Figure 5.1.2.2 shows the segment of code in React.js that was responsible for sending user queries to the Flask backend. When user submits a query, the handleSend function is triggered. The user’s message is firstly added to the local chat history which is setMessages and the input box is cleared. The query is then sent in JSON format to the Flask backend’s /chat endpoint using a POST request. The use of credentials: ‘include’ is an important configuration for session and it ensures the session data which is conversational history is preserved across multiple queries.

```javascript
      const data = await res.json();

      // Handle the response format properly
      const botMessage = {
        role: 'assistant',
        content: data.response,
        references: data.references || [],
        agent: data.agent || null
      };
      setMessages((prev) => [...prev, botMessage]);
```

Figure 5.1.2.3 Code snippet that shows processing of backend response.

Figure 5.1.2.3 shows the segment of code in React.js that was responsible for processing backend response. Once the backend responds, this code extracts the chatbot's reply which is named data.response and optional response such as references that the chatbot refer to when generating responses or the responding agent. The new response generated by the assistant is then appended to the conversational history to allow users to ask follow-up questions. With this modular design, it is easier to display both the generated answer and its sources.

```
        messages.map((msg, idx) =>(
          <div key={idx} className={`message ${msg.role}`}>
          {/* Show agent info if available */}
          {msg.agent && (
            <div className="agent-info">
              <small><strong>{msg.agent.name}</strong></small>
            </div>
          )}

            <ReactMarkdown>
              {typeof msg.content === 'object' && msg.content !== null
                ? msg.content.response || msg.content
                : msg.content
              }
              </ReactMarkdown>

            {msg.content.references && msg.content.references.length > 0 && (
              <div className="message-references">
                <hr />
                <h4>References:</h4>
                <ul>
                  {msg.content.references.map((ref, refIdx) => (
                    <li key={refIdx}>{ref}</li>
                  ))}
                </ul>
                </div>
```

Figure 5.1.2.4 Code snippet that shows rendering of chatbot message.

Figure 5.1.2.4 shows the segment of code in React.js that was responsible for rendering of chatbot message generated and returned by chatbot backend. The loop goes through every message in messages which is a variable storing both user and assistant messages. Each message is wrapped in a <div> with a CSS class based on its role and is different by styling such as display colors and display of message at left or right side of the chatbot UI. If the message contains metadata of department or division agent who provides answer, it displays the agent's name above the response name. Otherwise, General Agent will be displayed. This helps the users know which department of division the answer is coming from. The chatbot's answer which is msg.content is shown in ReactMarkdown. The React Markdown library allows chatbot responses to be displayed with formatting such as bold text, bullet points, headings and more [27]. This makes long or structured answers easier for users to read. If the chatbot's answer is an object, for example, a chatbot's answer in dictionary format with keys named response and references. msg.content.response is extracted to display as answer to a user query while msg.content.references is extracted to display as references that the chatbot referred to when generating response to the user query.

```
{loading && (<div className="message assistant">
  <div className="loading-indicator">
    Thinking...
    <small>(First query to each department may take longer as we load the database)</small>
    </div>
    </div>
)}
```

Figure 5.1.2.5 Code snippet that shows rendering of chatbot message.

Figure 5.1.2.5 shows the segment of code in React.js that was responsible for display of "Thinking…" indicator and messages by assistant to inform the users that first query may take longer time to get answer as the UTAR multiagent RAG chatbot system employed lazy loading of vector databases for each department and division. This improves the user experience by constantly updating the users about the chatbot status to show that the chatbot is actively working to generate response to the user query, especially during the initial database loading.

5.1.3 Chatbot Deployment

The deployment of the chatbot involved two parts: the chatbot backend which was the Flask backend was hosted on Render while the React frontend was hosted on Firebase. The frontend and backend parts of the chatbot were deployed to different platforms. The steps to outline how each component was deployed will be detailed below.

5.1.3.1 Backend Deployment on Render

The Flask backend files, including app.py, agent files, requirements.txt, render.yaml and zipped vector database file for each department and division of UTAR were organized into a GitHub repository.

File name	Description
app.py	A file that uses Flask that defines a web application by setting up a web server and specifies how it responds to user requests.

Agent files (agent_classes.py, agent_orchestrator.py)	Python files that define the structure of agent orchestrator and department and division agents such as their class, base class, attributes and methods.
requirements.txt	A text file specifies the third-party libraries and packages that need to be downloaded for the project.
render.yaml	A configuration file that defines the Render deployment setup.
Zipped vector database files	Chroma vector database file for each UTAR department and division that need to be loaded to be used by the chatbot system.

Table 5.1.3.1.1 Flask backend files.

```yaml
services:
  # Backend (Flask)
  - type: web
    name: utar-chatbot-backend
    env: python
    region: singapore
    plan: standard
    buildCommand: "pip install -r requirements.txt"
    # Increased timeout for first-time database loading per department
    startCommand: "gunicorn app:app --timeout 600 --workers 1 --threads 2 --max-requests 1000 --max-requests-jitter 100"
```

Figure 5.1.3.1.1 Code snippet that shows the configuration for Render deployment setup.

Secondly, render.yaml file with content as shown in Figure 5.1.3.1.1 was created. The buildCommand in the code snippet as shown in Figure 5.1.3.1.1 installed all required dependencies from requirements.txt for the project. startCommand tells the Render hosting platform to use Green Unicorn web server or Gunicorn to start the Python Flask web application. The timeout was increased to 600 seconds from default timeout to handle loading of vector database since the chatbot system uses lazy loading of vector database. Lazy loading of vector database means the vector database for a department or division is loaded when it receives first request relevant to the department or division, so the first request requires more time to get answer. The worker and thread setup were optimized for better memory usage.

After that, the GitHub repository for the chatbot was linked to Render and the render.yaml file was detected and used by the Render automatically. Before deployment, some environment variables which are sensitive keys such as the API keys used for connecting to the OpenAI GPT-4o model and OpenAI embeddings as well as the model to be used as base LLM model for the chatbot were configured as shown in Figure 5.1.3.1.2. Persistent disk was also configured on Render to have size of 10GB, and it was mounted to path /var/data as shown in Figure 5.1.3.1.3. The configured Render persistent disk was used to store vector database files for each department and division. With this approach, the vector database files remained attached to the chatbot service and data was preserved across deploys and restarts.

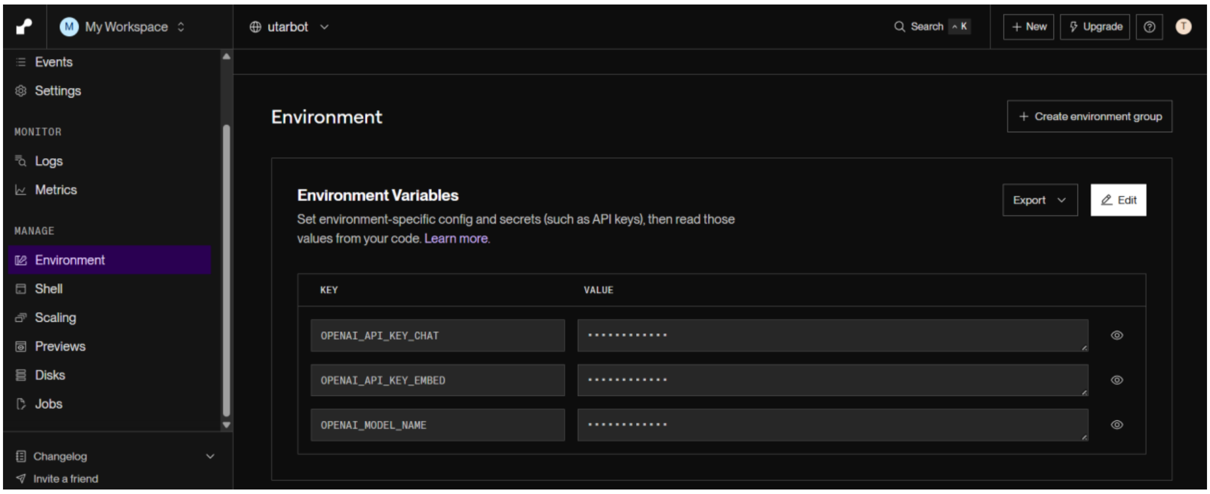


Figure 5.1.3.1.2 Environment variables configuration.

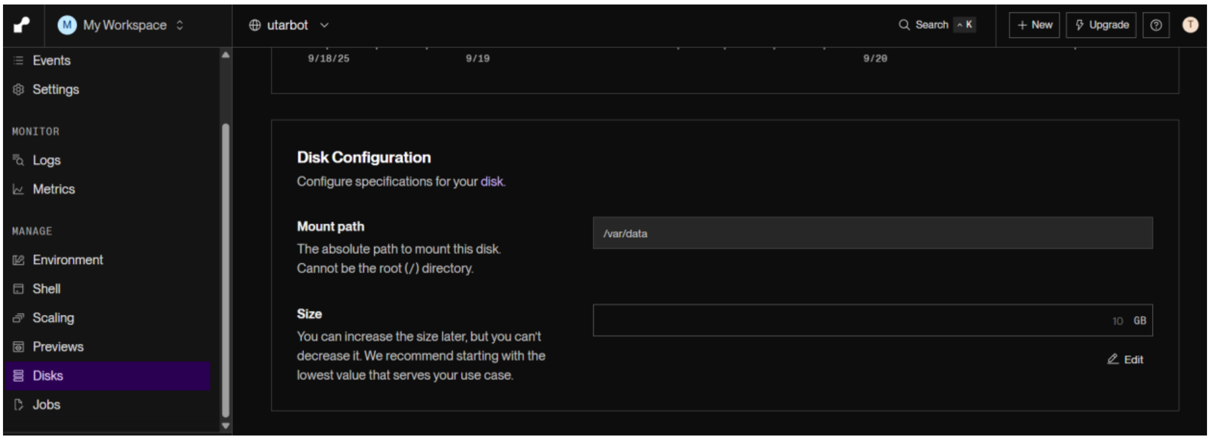


Figure 5.1.3.1.3 Disk configuration in Render.

Finally, the Flask backend was built and deployed to Render hosting platform using the Standard Render plan that provides persistent disk service as shown in Figure 5.1.3.1.4. After successful deployment, a live URL was generated, and the live URL was tested with test queries to ensure the deployed chatbot can provide correct, accurate and quick responses.

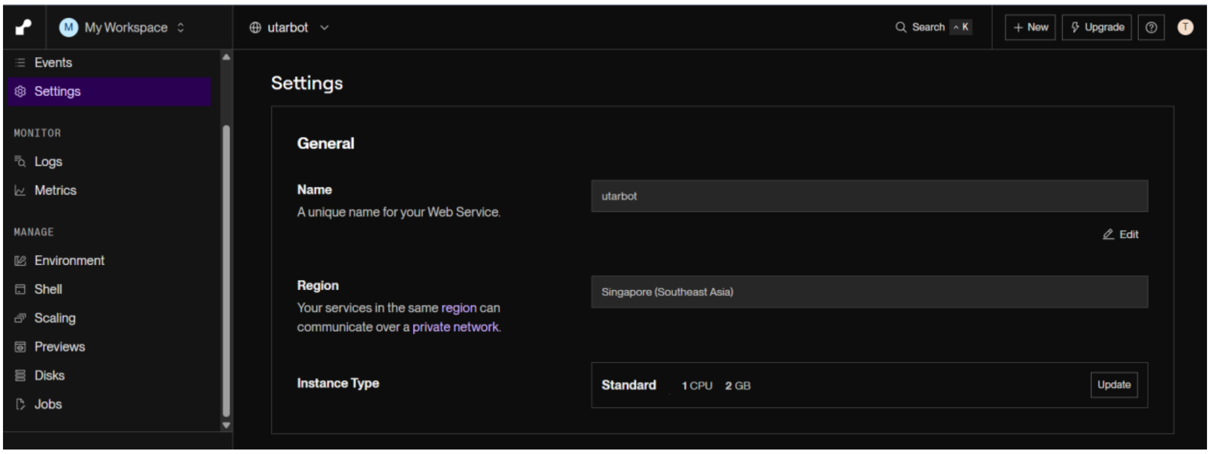


Figure 5.1.3.1.4 Settings configuration in Render platform.

5.1.3.2 Frontend Deployment on Firebase

After completing the chatbot React frontend development, the command which was npm run build was used to generate and optimize production-ready files inside the /build folder to be used for deployment later [28]. Then, to initialize a Firebase project, the command which was firebase init hosting was typed and run to set up hosting for the chatbot frontend using the Firebase Command Line Interface (CLI) [29]. The build directory was selected as the hosting folder, and automatic rewrites were configured as shown in Figure 5.1.3.2.1. The key named rewrites was configured in firebase.json to tell the Firebase to serve the index.html file which is the first page users visit when visiting a website to users trying to browse to the chatbot main page [30]. After the optimized production-ready React files were ready and Firebase project configurations were done, the command which was firebase deploy was used to upload the static files in build folder to Firebase servers, making the frontend globally available [29]. The frontend part of the chatbot system was configured to send API requests to the Render backend endpoint which was the live URL that was generated after successful backend deployment to Render. After successful frontend deployment to Firebase hosting platform, a final deployed link was obtained and it was tested to confirm that the frontend could successfully send queries and requests to the backend and the backend can receive the correct input and display responses with respond to it.

chatbot-ui > firebase.json > {} hosting > [ ] rewrites > {} 0

```json
{
  "hosting": {
    "public": "build",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}

```

Figure 5.1.3.2.1 Code snippet that shows the content of firebase.json.

5.2 Settings and Configurations

5.2.1 Backend Configurations

The backend of the chatbot was implemented using Flask, which served as the core API layer to handle requests from the frontend and return responses generated by UTAR multi-agent RAG chatbot system. There were two main endpoints created for the chatbot system which were /chat and /health as shown in Figure 5.2.1.1 and Figure 5.2.1.3. The /chat served as the endpoint to receive user queries, processed them through the Agent Orchestrator, and returned the answer generated by a department or division agent along with references and the information of the responsible agent. The /health endpoint was used to perform a simple service health check. It is used to indicate that the chatbot backend service is ready to be used when the chatbot backend has been successfully deployed.

```python

@app.route('/chat', methods=['POST'])
def chat():
    try:
        data = request.get_json()
        query = data.get('question')
        history = session.get('chat_history', [])
        sid = session.get("session_id", "NoSession")

        #
        # logging.info(f"Session ID: {session.get('_id', 'No session ID')}")
        logging.info(f"Incoming question: {query}")
        logging.info(f"Current history length: {len(history)}")
        logging.info(f"History so far: {history}")

        if not query:
            return jsonify({'error': 'No question provided'}), 400

        history.append({'role':'user', 'content':query})

        result = agent_orchestrator.process_query(query, history)

        history.append({'role':'assistant', 'content': result['response']})
        # logging.info(f"Final History List: {history}")
        session['chat_history'] = history[-6:]
        # session.modified = True

        logging.info(f"[Session {sid}] Assistant response: {result['response']}")
        logging.debug(f"[Session {sid}] Updated history: {session['chat_history']}")

```

Figure 5.2.1.1 Code snippet that shows the definition of /chat endpoint in the chatbot.

```python
        return jsonify({
            'response': result['response'],
            'agent': {
                'name': result['agent_name'],
                'description': result['agent_description']
            }
        })

    except Exception as e:
        sid = session.get("session_id", "NoSession")
        logging.error(f"[Session {sid}] Error in chat endpoint: {e}")
        return jsonify({
            'response': "I apologize, but I'm experiencing technical difficulties. "
            "If this is your first query to a specific department, the database might still be loading. "
            "Please try again in a moment.",
            'references': [],
            'agent': {'name': 'System', 'description': 'Error handler'}
        }), 500
                                You, 3 weeks ago • using lazy loading …
```

Figure 5.2.1.2 Code snippet that shows the continue definition of /chat endpoint in the chatbot.

```python
@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({'status': 'ok'})

```

Figure 5.2.1.3 Code snippet that shows the definition of /health endpoint in the chatbot.

To maintain conversational continuity, Flask-Session with server-side file storage which was .flask_session/ was used by the chatbot system to keep track of existing recent conversational history. session_id was a unique identifier configured to uniquely identify each user session, ensuring that every single user had an isolated interaction context. The conversational history should not be shared among users to provide more personalized responses to every single user. Figure 5.2.1.4 shows the method defined to generate a unique ID for each user session via UUID. Figure 5.2.1.5 shows configurations of session management. These configurations mean that the conversations between users and the chatbot were stored on the server-side filesystem which was .flask_session/. The signed session cookies were secured against tampering by configuring the SESSION_USE_SIGNER to be True. The SESSION_COOKIE_SAMESITE was configured to be None to allow cross-site access from the Firebase frontend of the chatbot system while the SESSION_COOKIE_SECURE was configured to be True to ensure the security of the communication between backend and frontend of the chatbot system by ensuring the cookies only sent over HTTPS.

```
@app.before_request
def assign_session_id():
    """Ensure every user gets a unique session ID for each session"""
    if "session_id" not in session:
        session["session_id"] = str(uuid.uuid4())

```

Figure 5.2.1.4 Code snippet that shows method to generate unique ID via UIID for each user session.

```
app.config['SESSION_TYPE'] = 'filesystem'
app.config['SESSION_PERMANENT'] = False
app.config['SESSION_FILE_DIR'] = './.flask_session/'
app.config['SESSION_USE_SIGNER'] = True
app.config['SESSION_COOKIE_SAMESITE'] = "None"
app.config['SESSION_COOKIE_SECURE'] = True


```

Figure 5.2.1.5 Code snippet that shows configurations of Flask-Session.

Since the frontend and backend of UTAR multiagent RAG chatbot system were deployed to different platforms which were Firebase and Render respectively, CORS was configured for the chatbot system as shown in Figure 5.2.1.6. The resources in the configurations of CORS mean any resource on the server of the chatbot system can be accessed by a client from any origin. Besides that, the supports_credential=True allows the session management of the chatbot system to work by allowing the browser to include credentials like cookies and session IDs in cross-origin requests [23]. In the chatbot system, the conversational history was not stored in cookies which has small storage size as conversational history may be very large. Instead, the conversational history was stored on the server filesystem. On the other side, cookies were used to store only the session ID to identify which session belongs to which user [25].

```
app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}}, supports_credentials=True)
```

Figure 5.2.1.6 Code snippet that shows the definition of /health endpoint in the chatbot.

All interactions between users and the chatbot were logged in Render backend to allow system monitoring, debugging and evaluation of how users interacted with the chatbot. Figure 5.2.1.7 shows the configurations of logging that will be used to log all the conversations between users and the chatbot. Figure 5.2.1.8 shows some examples of logging of conversations between users and the chatbot. Session ID was logged to identify which session is the current user questions, generated answers and conversational history belong to.

```
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[logging.StreamHandler()]
)
```

Figure 5.2.1.7 Code snippet that shows configurations for logging.

```
        logging.info(f"[Session {sid}] Assistant response: {result['response']}")
        logging.debug(f"[Session {sid}] Updated history: {session['chat_history']}")
```

Figure 5.2.1.8 Code snippet that shows code to load some important information at chatbot backend.

Moreover, the chatbot backend was configured with a vector database extraction process as shown in Figure 5.2.1.9. The vector databases were first created for each UTAR departments and divisions before zipping the vector database files and uploading them to Render persistent disk mounted at /var/data.

Firstly, the chatbot system checked for any department-related or division-related PDF files for each available department and division in the specified UTAR webpages. These files were then downloaded to the department or division folders via scrape_web_pdfs method as shown in Figure 5.2.1.12 and Figure 5.2.1.13. There were some UTAR PDF documents that were downloaded to department or division folders manually from UTAR portal or other UTAR webpages that required UTAR students' or UTAR staffs' login credentials. The content in the downloaded files were then extracted by parsing them with LangChain's UnstructuredPDFLoader with the method in Figure 5.2.1.14.. Metadata which was the source of the PDF files was attached to each document. With this, during the chunking process, the source of each individual chunk can be traced with the attached metadata.

The method named scrape_webpage in Figure 5.2.1.10 and Figure 5.2.1.11 loads UTAR webpages using Playwright after waiting for 3 seconds for JavaScript to render. Textual content was then extracted from specific <div> sections while unique links were extracted from <a> sections. The combined extracted data was then wrapped into LangChain objects, each with content and metadata which was its source URL.

After extracting data from PDF documents and from scrapping text content from UTAR webpages, both the data were combined into a single list of documents to provide each specialized with specific department or division information to generate department or division-specific responses.

The final created vector database file for each UTAR department and division contained vector embeddings and all the files were zipped. The zipped file was uploaded to GitHub to be used by Render. During deployment process, the content in the zipped file was extracted to /var/data using method as shown in Figure 5.2.1.9. The extracted content were the vector embeddings that will be used by Render to get information about each specific UTAR department or division to generate accurate department or division-specific responses.

```python
# Path for vector DB storage
VECTOR_DB_EXTRACT_PATH = "/var/data"
VECTOR_DB_FOLDER = "/var/data/vector_db"

BUNDLED_ZIP_PATH = "./vector_db.zip"

if os.path.exists(BUNDLED_ZIP_PATH):
    # Create parent directory
    os.makedirs(VECTOR_DB_EXTRACT_PATH, exist_ok=True)

    logging.info(f"Unzipping bundled vector DB from {BUNDLED_ZIP_PATH} to {VECTOR_DB_EXTRACT_PATH}")
    with zipfile.ZipFile(BUNDLED_ZIP_PATH, 'r') as zip_ref:
        # Extract to /var/data so that vector_db folder is created there
        zip_ref.extractall(VECTOR_DB_EXTRACT_PATH)

    # Verify the extraction worked
    if os.path.exists(VECTOR_DB_FOLDER):
        logging.info(f"Successfully extracted vector DB to {VECTOR_DB_FOLDER}")
    else:
        logging.error(f"Extraction failed - {VECTOR_DB_FOLDER} not found")
else:
    logging.warning(f"No bundled vector DB ZIP found at {BUNDLED_ZIP_PATH}. Continuing without preload.")
    # Create empty directory as fallback
    os.makedirs(VECTOR_DB_FOLDER, exist_ok=True)

```

Figure 5.2.1.9 Code snippet that shows vector database extraction process.

```python
    def scrape_webpage(self, urls):
        print("\nScrapping some UTAR Webpages.")
        seen_links = set()
        results = []

        with sync_playwright() as p:
            browser = p.chromium.launch()

            for url in urls:
                page = browser.new_page()
                page.goto(url)
                page.wait_for_timeout(3000)  # wait for JS to load
                html_content = page.content()
                page.close()

                soup = BeautifulSoup(html_content, 'html.parser')

                # ---- Collect Text ----
                page_text_parts = []
                for div in soup.find_all('div', class_='mg'):
                    section_text = div.get_text(separator='\n', strip=True)
                    if section_text:
                        page_text_parts.append(section_text)

                # ---- Collect Unique Links ----
                seen_links = set()       You, 3 weeks ago • Initial commit for utarbot project
                link_texts = []
                link_metadata_list = []
                for link in soup.find_all('a', href=True):
```

Figure 5.2.1.10 Code snippet that shows method used to scrap UTAR webpages.

```
                        full_url = urljoin(url, link['href'])
                        if full_url not in seen_links:  # ensure uniqueness
                            seen_links.add(full_url)
                            text = link.get_text(strip=True)
                            if text:
                                link_texts.append(text)
                                link_metadata_list.append({"text": text, "url": full_url})

                    # ---- Decide How to Combine ----
                    if not page_text_parts:  # if no main text, combine link texts
                        combined_text = " | ".join(link_texts)
                    else:
                        combined_text = "\n".join(page_text_parts)
                        if link_texts:
                            combined_text += "\nLinks: " + " | ".join(link_texts)

                    results.append(
                        Document(
                            page_content=combined_text,
                            metadata={"source":url}
                        )
                    )


            browser.close()
            print("\nDone Scraping UTAR Webpages.")

        return results

```

Figure 5.2.1.11 Code snippet that shows continue parts of method used to scrap UTAR webpages.

```

    def scrape_web_pdfs(self, urls, department, base_folder="/var/data"):
        """
        Scrapes a webpage for all linked PDFs and downloads them into a department folder.
        Handles UTAR's broken SSL for PDFs specifically.
        """
        print("Looking for PDF files in some UTAR Webpages to download...")
        print("department", department)
        download_folder = os.path.join(base_folder, department)
        os.makedirs(download_folder, exist_ok=True)

        pdf_files = []

        # Use Playwright to fetch rendered HTML
        with sync_playwright() as p:
            browser = p.chromium.launch()

            for url in urls:
                page = browser.new_page()
                page.goto(url)
                page.wait_for_timeout(3000)  # wait for JS to load
                html_content = page.content()
                page.close()


                soup = BeautifulSoup(html_content, 'html.parser')

```

Figure 5.2.1.12 Code snippet that shows method used to scrap PDF files from UTAR webpages.

```python
                # Download PDFs only
                for link in soup.find_all('a', href=True):
                    full_url = urljoin(url, link['href'])
                    if full_url.lower().endswith(".pdf"):
                        file_name = os.path.basename(full_url)
                        pdf_path = os.path.join(download_folder, file_name)

                        if os.path.exists(pdf_path):
                            print(f"Skipping (already exists): {file_name}")
                            continue

                        try:
                            if "utar.edu.my" in full_url.lower():
                                # Bypass SSL verification for UTAR
                                r = requests.get(full_url, stream=True, verify=False, timeout=10)
                            else:
                                r = requests.get(full_url, stream=True, verify=certifi.where(), timeout=10)
                            r.raise_for_status() # Check HTTP response for errors to avoid downloading broken files
                            with open(pdf_path, "wb") as f: # Open pdf in binary write mode
                                for chunk in r.iter_content(8192):
                                    if chunk:
                                        f.write(chunk)
                            pdf_files.append(pdf_path)
                            print(f"Downloaded PDF: {pdf_path}")
                        except Exception as e:
                            print(f"Failed to download {full_url}: {e}")

            browser.close()

        return pdf_files
```

Figure 5.2.1.13 Code snippet that shows continue parts of method used to scrap PDF files from UTAR webpages.

```python

    def ingest_pdf(self, doc_folder_path):
        scraped_pdf_files = self.scrape_web_pdfs(self.urls, self.department)

        print(f"Looking for PDFs in: {doc_folder_path}")
        if not os.path.exists(doc_folder_path):
            logging.error(f"Folder not found: {doc_folder_path}")
            return None

        all_data = []
        for pdf_file in glob.glob(os.path.join(doc_folder_path, "*.pdf")):
            try:
                print(f"Loading PDF: {pdf_file}")
                loader = UnstructuredPDFLoader(file_path=pdf_file)
                data = loader.load()

                # add source metadata
                for doc in data:
                    doc.metadata["source"] = os.path.basename(pdf_file)

                all_data.extend(data)
            except Exception as e:
                logging.error(f"Failed to load {pdf_file}: {e}")

        return all_data

```

Figure 5.2.1.14 Code snippet that shows method used to parse UTAR PDF files.

5.2.2 Frontend Configurations

The configurations done on the React frontend was to ensure the session cookies could be passed along with API requests. The configurations were done by adding credentials: 'include' in the fetch request. This allowed the chatbot backend to match each query with the correct session conversational history. These configurations of React frontend with the configurations of backend related to session management allowed the chatbot to maintain conversational continuity across multiple queries.

```
      const res = await fetch('https://utarbot.onrender.com/chat', {
        method: 'POST',
        credentials: 'include',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          question: input
        })
      });

```

Figure 5.2.2.1 Code snippet that shows configurations in React frontend for session management.

5.3 System Operation

Once deployed, the chatbot system operated as an interactive platform accessible via a web interface hosted on Firebase platform. When a user browsed to the chatbot web application, the React frontend provided a chat window where the user can type queries as shown in Figure 5.3.1. The queries were sent in JSON format to the Flask backend through a POST request to the /chat endpoint as shown in Figure 5.3.2. The backend first retrieved and identified the session ID, retrieved any prior conversation history as shown in Figure 5.3.2, and then routed the query to the Agent Orchestrator.

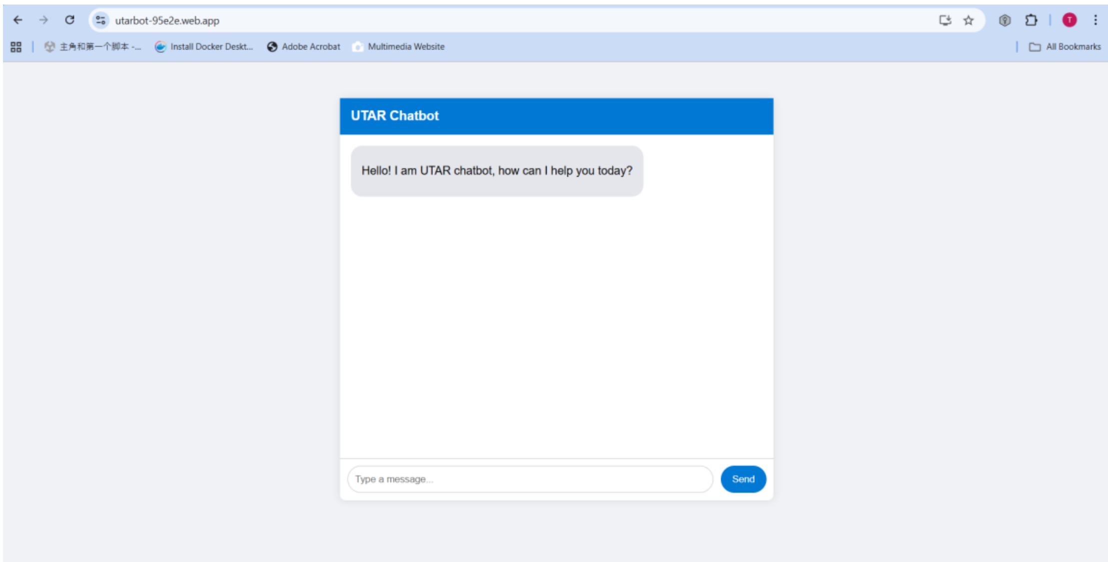


Figure 5.3.1 UI of UTAR multi-agent RAG chatbot.

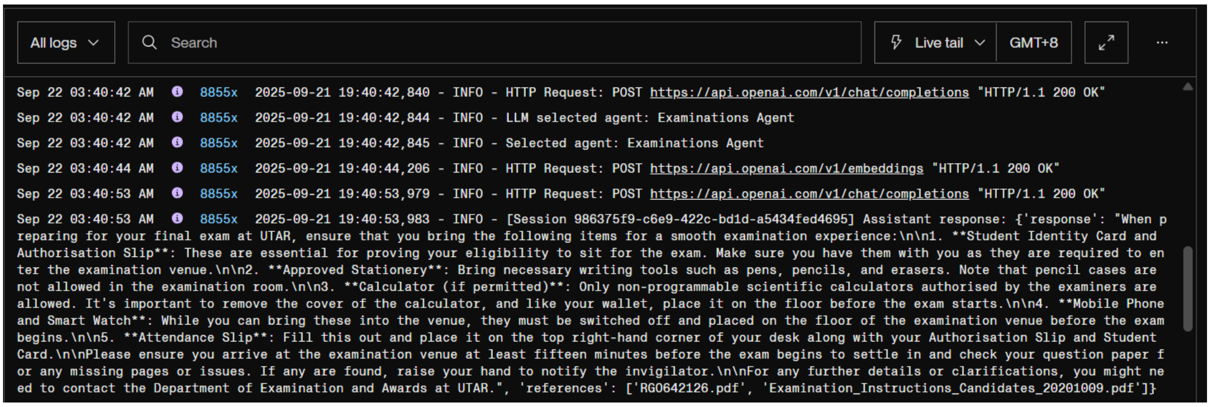


Figure 5.3.2 Logs in Flask backend displayed in Render.

The Agent Orchestrator acted as the decision-making layer, selecting the most suitable and relevant department-specific or division-specific agent, which was either Admissions, Finance, Examinations or falling back to the General Agent if no specialized match was found. As shown in Figure 5.3.2, the selected agent was Examinations Agent when the question asked by the user was related to examinations. Each agent retrieved contextual information and data from its respective vector database through OpenAI key used for OpenAI embeddings, which was pre-built from UTAR webpages and UTAR PDF documents. With both context and conversation history, the agent generated a response using the OpenAI GPT-4o LLM model. The final answer with the name of the agent who answered the question were then returned to the frontend as shown in Figure 5.3.4, where the user could view it in real time. If references were available, they were also displayed under the response for transparency. The user query and the generated response were appended to the conversational history of particular session and only the latest six conversational exchanges between a user and the chatbot system were stored for each unique session. The storing of conversational history as shown in Figure 5.3.3 was to maintain the conversation continuity and allow users to ask follow-up questions.

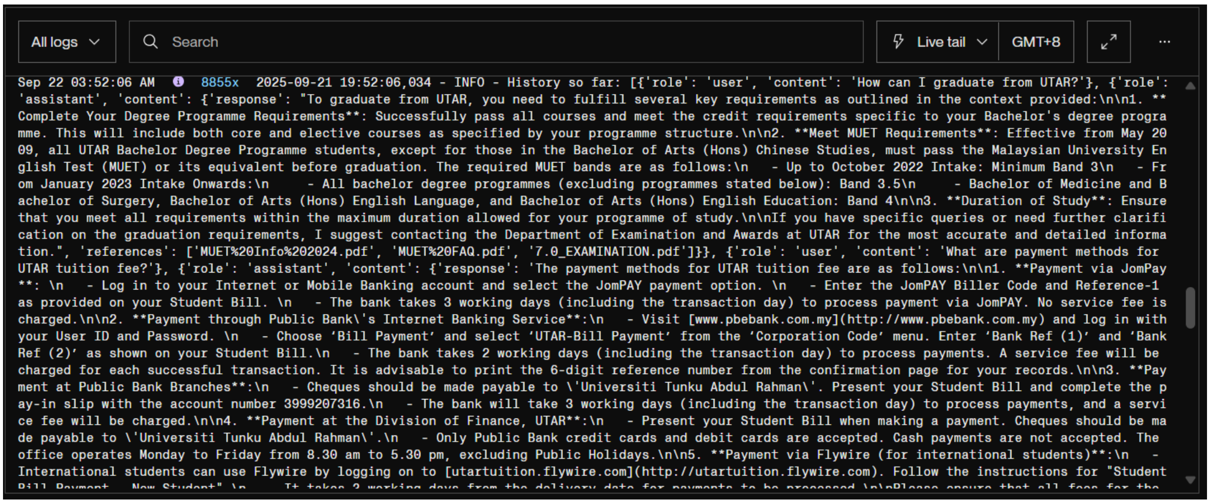


Figure 5.3.3 Logs in Flask backend displayed in Render that shows the conversational history of a specific session.

The system also included a health check endpoint, /health to verify backend availability, while the frontend displayed a “Thinking…” message to keep the user informed when the response generation took longer especially on first query when vector databases were still loading as shown in Figure 5.3.5.

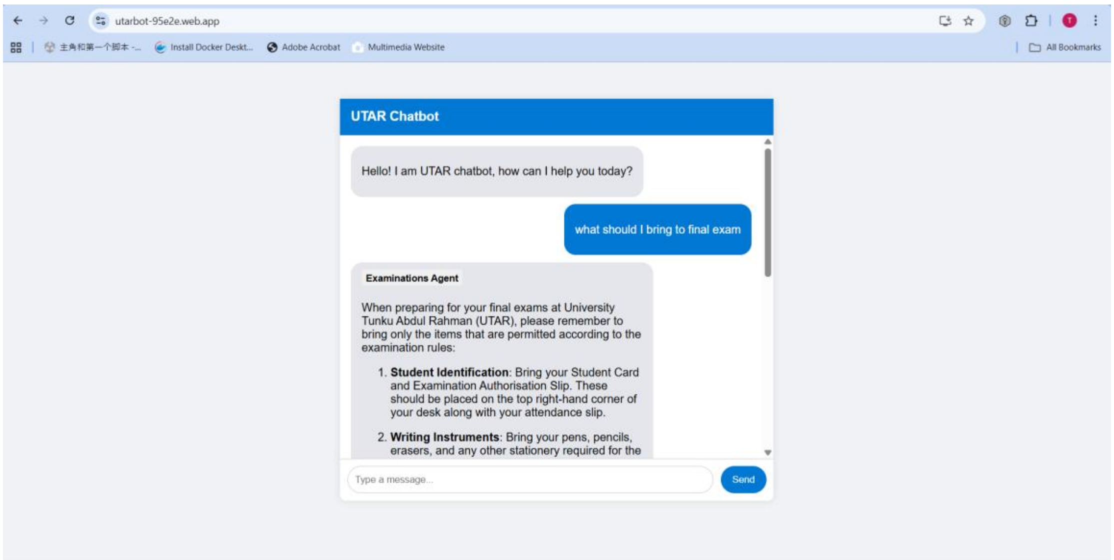


Figure 5.3.4 UI of UTAR multi-agent RAG chatbot with question and generated responses.

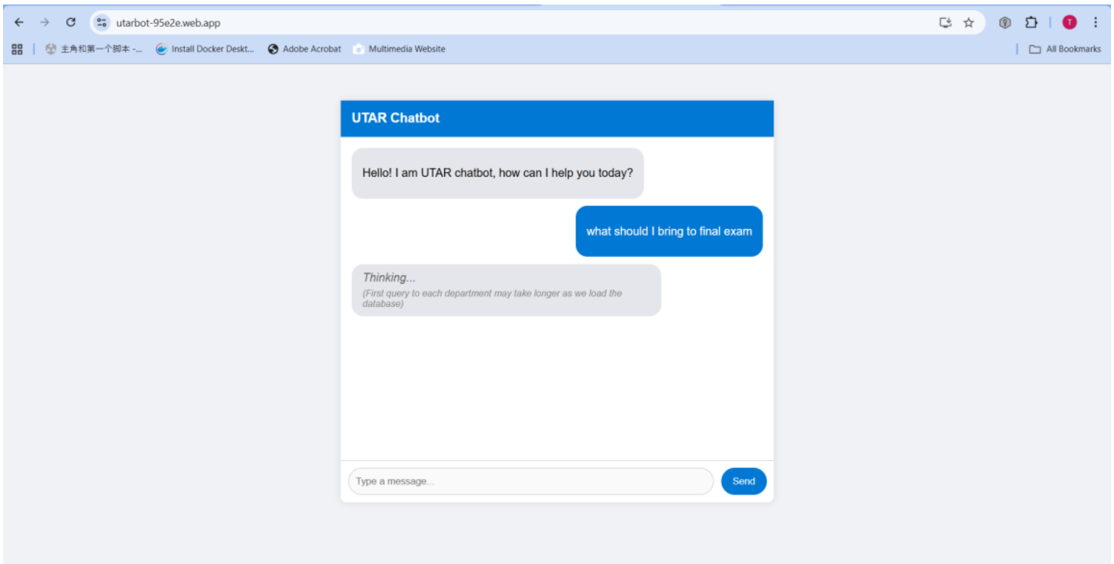


Figure 5.3.5 UI of UTAR multi-agent RAG chatbot with question and generated responses.

5.4 Implementation Issues and Challenges

There were several challenges faced during the implementation. After doing research regarding UTAR website, most UTAR webpages had broken SSL certificates for PDF downloads which would cause errors when Playwright trying to download the PDF documents from some UTAR webpages. PDF documents on UTAR webpages contained important information about UTAR departments and divisions that were required to build vector databases for specialized agents. The information and data on UTAR webpages were required to be used in generating responses to user queries. To overcome the issues, the chatbot system was designed to bypass SSL verification specifically for UTAR domains as shown in Figure 5.4.1 while enforcing proper certificate validation using certify for all other domains since UTAR domains were mostly to be trusted source and free from malicious content. This ensured both reliability in accessing UTAR's content which included UTAR PDF documents and security when downloading from external sources.

```
try:
    if "utar.edu.my" in full_url.lower():
        # Bypass SSL verification for UTAR
        r = requests.get(full_url, stream=True, verify=False, timeout=10)
    else:
        r = requests.get(full_url, stream=True, verify=certifi.where(), timeout=10)
    r.raise_for_status() # Check HTTP response for errors to avoid downloading broken files
    with open(pdf_path, "wb") as f: # Open pdf in binary write mode
        for chunk in r.iter_content(8192):
            if chunk:
                f.write(chunk)
    pdf_files.append(pdf_path)
    print(f"Downloaded PDF: {pdf_path}")
except Exception as e:
    print(f"Failed to download {full_url}: {e}")
```

Figure 5.4.1 Code snippet that shows solution to resolve broken SSL certificates in UTAR domains.

Initially, the chatbot system was designed to load all vector databases for all UTAR departments and divisions during deployment. The loading of all vector databases meaned loading millions or billions of vectors into memory and this caused rapid consumption of RAM on Render server. This caused the deployment of the web application to crash and fail due to memory exhaustion. The first way to solve the problem was by upgrading the Render plan but this method did not solve the problem in the end, the deployment still failed. Even if the deployment was successful, this method costed very much for upgrading of the Render plan. The way to fix the memory exhaustion method was by using lazy loading of vector database for each specialized agent, where vector databases only initialized and loaded when a query related to the department or division was identified. With this method, only the first query related to a department or division takes longer time to load its vector database.

There were some issues with the session management initially. The conversational history was lost between queries due to cookie restrictions. After debugging, the issue was solved by doing configurations as shown in Figure 5.4.2, setting SESSION_COOKIE_SAMESITE to "None" to allow the cookies containing session IDs to be transferred over different platform which was from the user's browser to Render backend API. Without this line of code, the conversation history for particular user session would be deleted once user asked a question to the chatbot. The SESSION_COOKIE_SECURE was configured to be True to ensure that the cookies only can be transferred over HTTPS for the security of session data.

```
app.config['SESSION_COOKIE_SAMESITE'] = "None"
app.config['SESSION_COOKIE_SECURE'] = True


```

Figure 5.4.2 Code snippet that shows configurations added to the Flask backend code to fix issues related to session management.

Moreover, Render imposed default timeout limits of thirty seconds which was not suitable for the chatbot system. The lazy loading of vector database for each department or division required more time so the timeout limits were configured to be six hundred seconds or ten minutes by adjusting gunicorn settings in render.yaml file which was shown in Figure 5.1.3.1.1. This fixing also solves the problems that a user asked a question that requires large context retrieval and prevents the timeout issues.

5.5 Concluding Remark

The system implementation phase brought together multiple components which were vector database creation, web scraping and PDF files downloading, multi-agent orchestration, React frontend chatbot UI and Flask backend services into a functional UTAR multi-agent RAG chatbot. The chatbot was now able to answer student queries related to admissions, examinations, finance and general university information by providing quick, accurate and context-relevant responses.

There were many issues and problems faced while trying to deploy the chatbot system which were SSL issues in some UTAR webpages, RAM memory constraints, session management and timeout issues. Some approaches were performed to resolve the encountered issues so that the chatbot frontend and backend can be successfully deployed to Firebase and Render respectively. After successfully deploying the chatbot system, the chatbot system was made available to students in a reliable and scalable way. Additionally, the modular design of the multi-agent system enables for addition of more agents and knowledge bases into the existing chatbot system in the future, making this project sustainable and extensible.

Chapter 6
System Evaluation and Discussion

6.1 System Testing and Performance Metrics

System testing was conducted to ensure that the UTAR multi-agent RAG chatbot system functions as intended and meets the objectives of the project. Since the chatbot interacts with students in real time and it retrieves contextual information from specialized department or division agent, functional testing, session testing, error handling, concurrent user simulation and performance testing were carried out to evaluate the performance of UTAR multi-agent RAG chatbot, correctness of responses and the overall user experience.

Functional testing was performed to verify that the chatbot direct user queries to correct UTAR department or division agent. Session testing was performed to verify that the chatbot was able to maintain short-term memory of previous interactions for the current session, allowing users to ask follow-up questions. In addition, error handling tests were performed to examine the chatbot's ability to gracefully manage invalid or unrelated queries without breaking the chatbot system. Concurrent testing simulated multiple users interacting with the chatbot at the same time in order to validate that the chatbot was able to serve simultaneous sessions independently. Finally, performance testing was performed to determine whether the system could provide responses to user queries within the targeted six seconds of project objectives by measuring the average system response time.

The evaluation also incorporated user feedback metrics to gather user feedback about the UTAR multi-agent RAG chatbot, collected using a Google Form survey distributed among UTAR students and staff. The survey included both Likert-scale questions and questions with Yes or No options to measure aspects such as usability, response accuracy, comfort in interaction and overall satisfaction. Both the technical testing and user experience feedback provided a holistic assessment of the chatbot system.

Performance Metrics	Descriptions
Response Answer Accuracy	Percentage of test cases where the chatbot selected the correct department or division

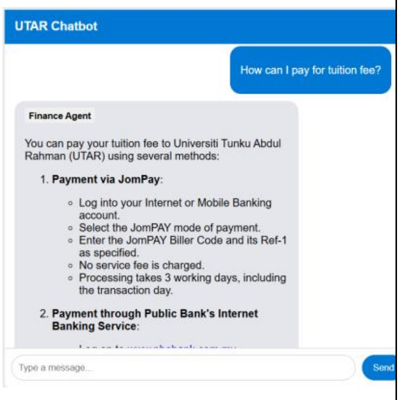
	agent and provided contextually relevant responses.
Conversational Memory Consistency	Success rate of follow-up queries being understood in context.
Average Response Time	Average time taken by the system to generate a response to a user query.
Concurrent User Support	Maximum number of users that could interact simultaneously with the chatbot without degradation of performance.
User Satisfaction Score	Average rating and feedback collected from survey responses on ease of use, response relevance, comfort and user satisfaction.

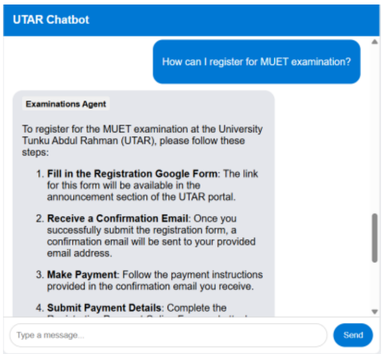
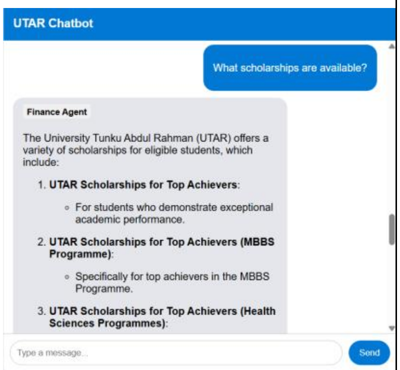
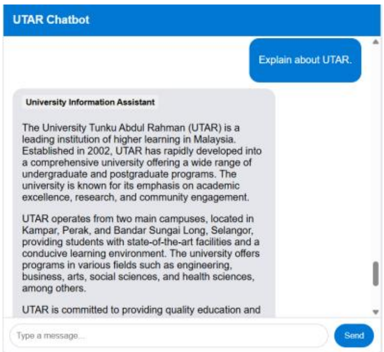
Table 6.1.1 Performance Metrics for Multi-agent RAG chatbot.

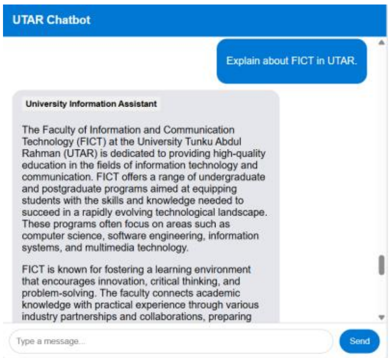
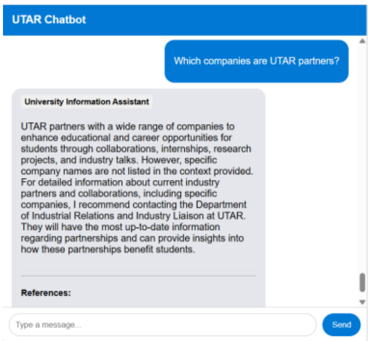
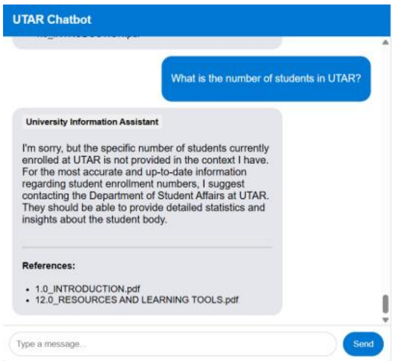
By combining both quantitative metrics which were response accuracy, conversational memory consistency, average response time, concurrent user support and qualitative feedback which were user satisfaction and user feedback, the testing and evaluation aimed to provide a comprehensive picture of the system's effectiveness in meeting the project objectives.

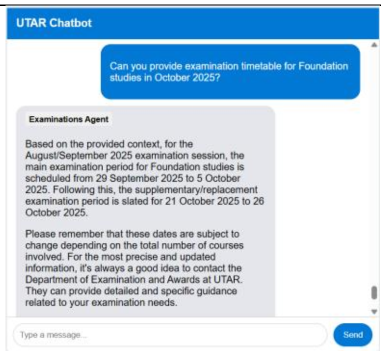
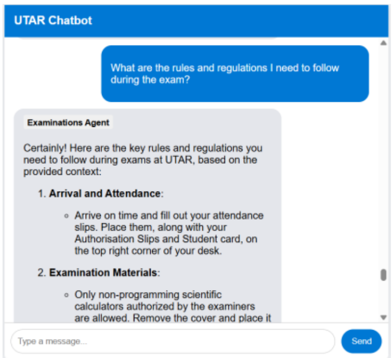
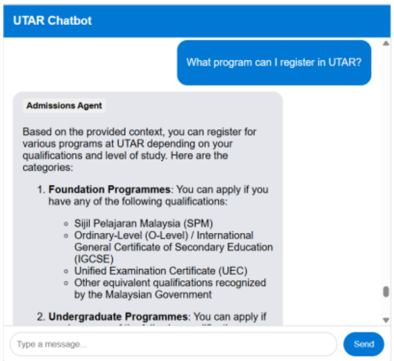
6.2 Testing Setup and Results

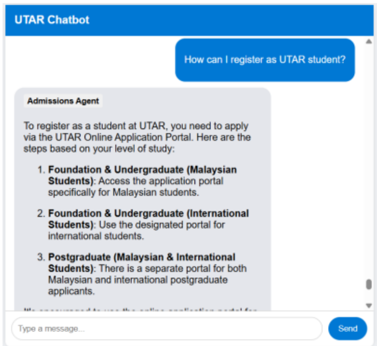
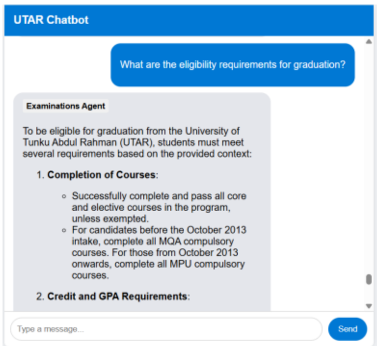
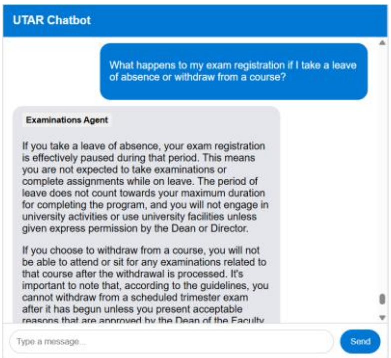
6.2.1 Functional Testing

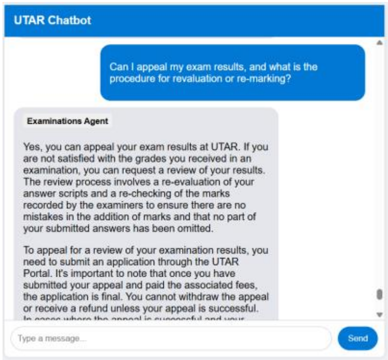
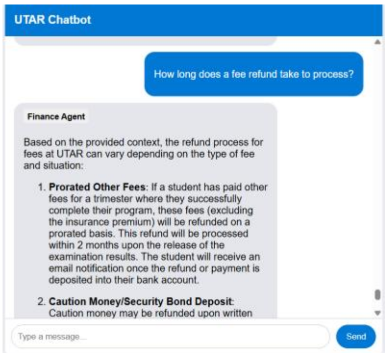
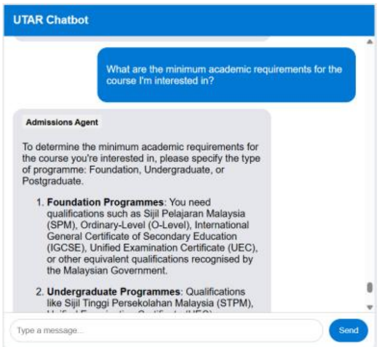
Test Case ID	Input Query	Expected Agent	Expected Output	Actual Result	Pass/Fail
T1	How can I pay for tuition fee?	Finance Agent	UTAR tuition fee payment methods are provided.	Finance agent was selected, and response was accurate. 	Pass

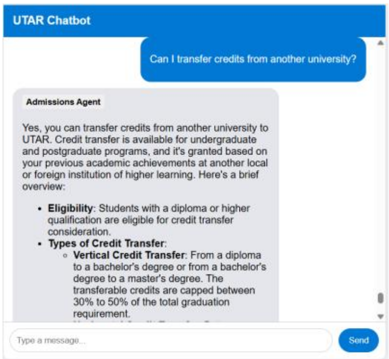
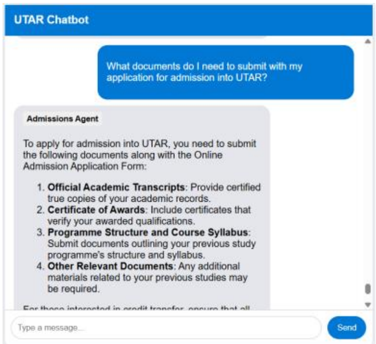
T2	How can I register for MUET examination ?	Examinations Agent	Steps on how to register for MUET examination are described.	Examinations Agent was selected and steps on how to register for MUET examination were described.  	Pass
T3	What scholarships are available?	Finance Agent	Scholarships info was provided.	Finance Agent was selected, and scholarships info was provided.  	Pass
T4	Explain about UTAR.	General Agent or University Information Assistant	Information about UTAR was provided.	General Agent was selected and information about UTAR was provided  	Pass

<table>
<tr><td>T5</td><td>Explain about FICT in UTAR.</td><td>General Agent or University Information Assistant</td><td>Information about FICT in UTAR was provided.</td><td>General Agent was selected and information about FICT in UTAR was provided.

</td><td>Pass</td></tr>
<tr><td>T6</td><td>Which companies are UTAR partners?</td><td>General Agent or University Information Assistant</td><td>Companies partner with UTAR were listed.</td><td>General Agent was selected, and companies partner with UTAR were not listed.

</td><td>Fail</td></tr>
<tr><td>T7</td><td>What is the number of students in UTAR?</td><td>General Agent or University Information Assistant</td><td>Number of students in UTAR was provided.</td><td>General Agent was selected and number of students in UTAR was not provided.

</td><td>Fail</td></tr>
<tr><td>T8</td><td>Can you provide examination timetable</td><td>Examinations Agent</td><td>Examination period for Foundation</td><td>Examinations Agent was selected and examination period for Foundation studies was provided.</td><td>Pass</td></tr>
</table>

	for Foundation studies in October 2025?		studies was provided.		
T9	What are the rules and regulations I need to follow during the exam?	Examinations Agent	Rules and regulations that need to be followed during the exam were provided.	Examinations Agent was selected and rules and regulations that need to be followed during the exam were provided. 	Pass
T10	What program can I register in UTAR?	Admissions Agent	Programs at UTAR were provided.	Admissions Agent was selected and categories of programs at UTAR were provided. 	Pass
T11	How can I register as	Admissions Agent	Steps to register as a UTAR	Admissions Agent was selected and steps to register	Pass

	UTAR student?		student were provided.	as a UTAR student were provided  	
T12	What are the eligibility requirements for graduation?	General Agent or University Information Assistant	Requirements for graduation were provided.	General Agent was selected and requirements for graduation were provided.  	Pass
T13	What happens to my exam registration if I take a leave of absence or withdraw from a course?	Examinations Agent	The effects of taking a leave or withdrawing from a course on exam registration were described.	Examinations Agent was selected and the effects of taking a leave or withdrawing from a course on exam registration were described.  	Pass
T14	Can I appeal my exam results, and	Examinations Agent	Answer is provided to inform the	Examinations Agent was selected and answer is provided to inform the user	Pass

<table>
<tr>
<td></td>
<td>what is the procedure for revaluation or re-marking?</td>
<td></td>
<td>user whether he or she can appeal his or her exam results and procedure for revaluation or remarking is described.</td>
<td>whether he or she can appeal his or her exam results and procedure for revaluation or remarking is described.

</td>
<td></td>
</tr>
<tr>
<td>T15</td>
<td>How long does a fee refund take to process?</td>
<td>Finance Agent</td>
<td>The time required for refund for some cases was stated.</td>
<td>Finance Agent was selected and the time required for refund for some cases was stated.

</td>
<td>Pass</td>
</tr>
<tr>
<td>T16</td>
<td>What are the minimum academic requirements for the course I'm interested in?</td>
<td>Admissions Agent</td>
<td>The minimum academic requirements for courses are provided.</td>
<td>Admissions Agent was selected and the minimum academic requirements for courses were provided.

</td>
<td>Pass</td>
</tr>
</table>

T17	Can I transfer credits from another university?	Admissions Agent	Answer is provided to determine whether credits can be transferred from another university.	Admissions Agent was selected, and answer is provided to determine whether credits can be transferred from another university.  	Pass
T18	What documents do I need to submit with my application for admission into UTAR?	Admissions Agent	Documents that need to be submitted with application into UTAR are listed.	Admissions Agent was selected and documents that need to be submitted with application into UTAR are listed.  	Pass
T19	Where can I find my student bill?	Finance Agent	Place to find student bill is provided.	Finance Agent was selected and place to find student bill is provided.	Pass

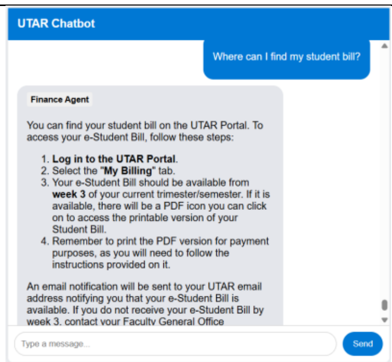
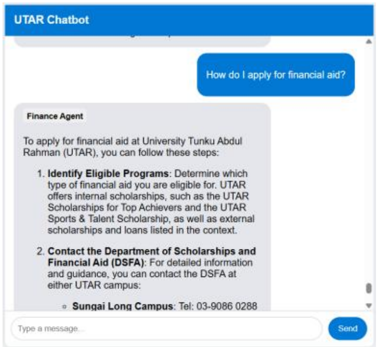
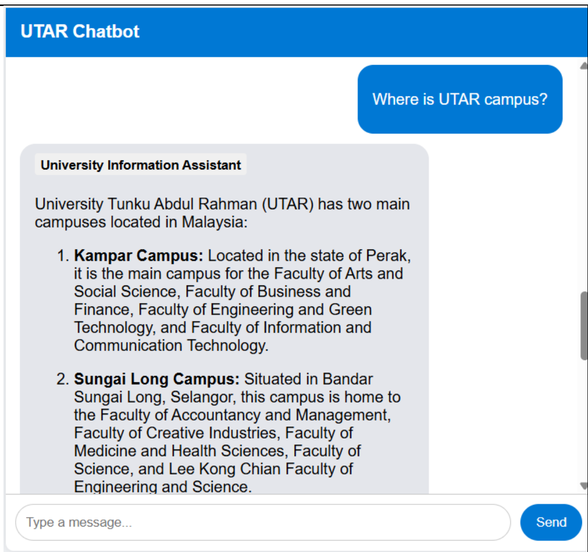
					
T20	How do I apply for financial aid?	Finance Agent	Steps to apply for financial aid are provided.	Finance Agent was selected and steps to apply for financial aid were provided. 	Pass

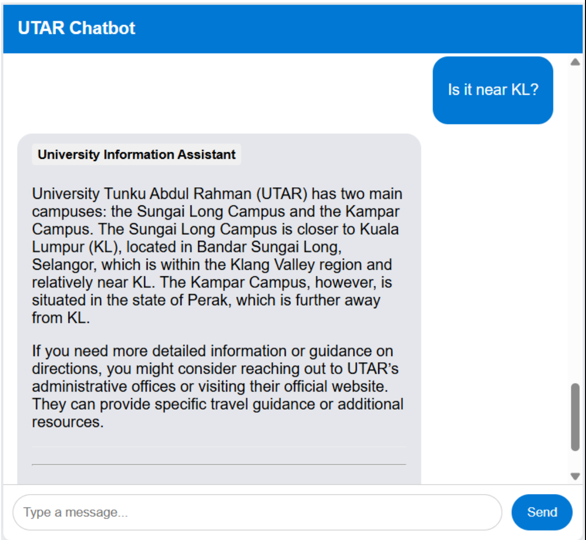
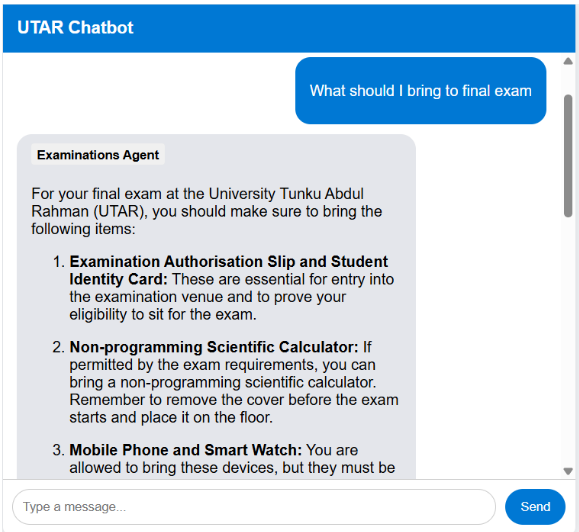
Table 6.2.1.1 Test cases for functional testing and their results.

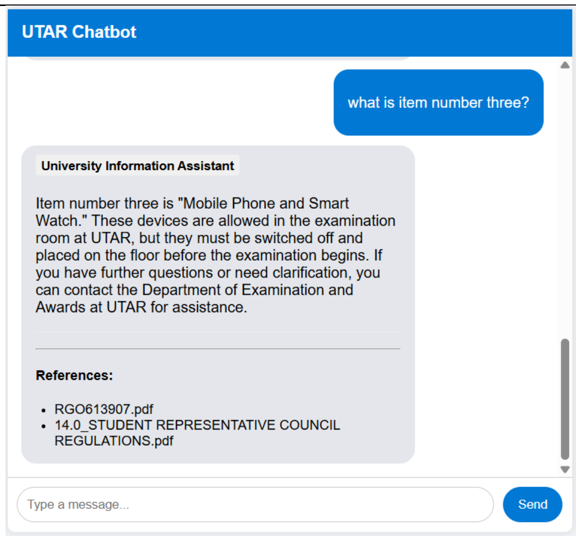
20 test cases were used for functional testing and the results obtained were shown in Table 6.2.1.1. In the 20 test cases, 5 test cases were related to Admission Agent, 5 test cases were related to Examinations Agent, 5 test cases were related to Finance Agent and the last 5 test cases were related to General Agent or University Information Assistant. The test cases were conducted to determine whether a user query was routed to the correct agent and whether the agent can provide correct response.

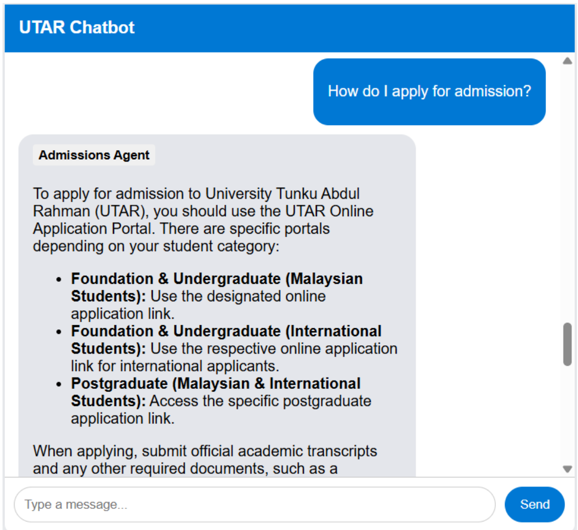
6.2.2 Session Testing

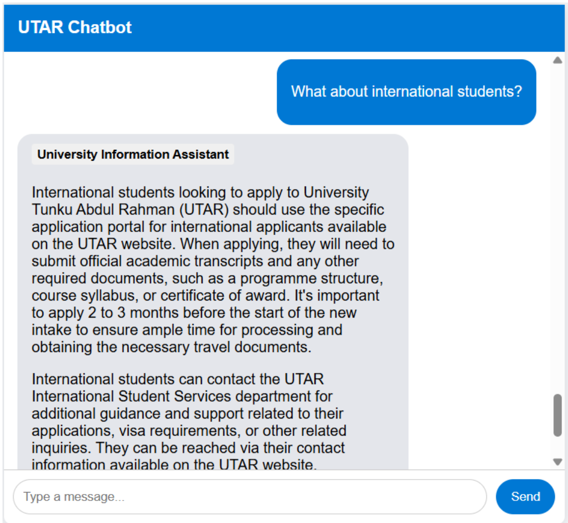
Test Case ID	Conversation Flow	Expected Behavior	Actual Behavior	Pass/Fail
S1	User: "Where is UTAR campus?"	Should continue remember the	Chatbot responded with correct responses.	Pass

<table>
<tr>
<td></td>
<td>User: “Is it near KL?”</td>
<td>General Agent’s context information</td>
<td>


</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>User: “What should I bring to final exam?”
User: “What is item number three?”</td>
<td>Should describe the number three item to be brought to final exam.</td>
<td>Chatbot provided information about the third item to be brought to final exam.

</td>
<td>Pass</td>
</tr>
</table>

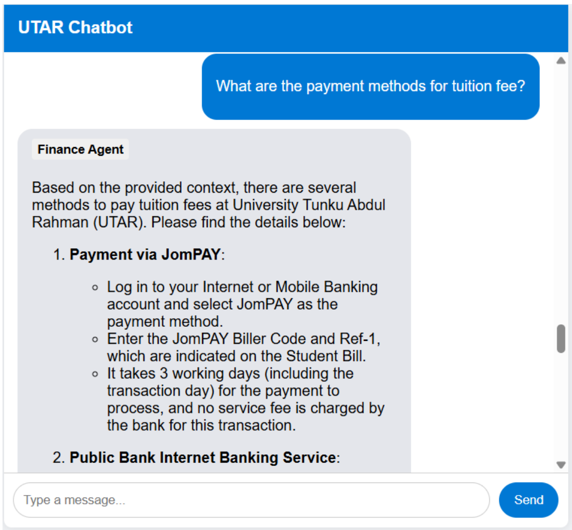
<table>
<tr><td></td><td></td><td></td><td>


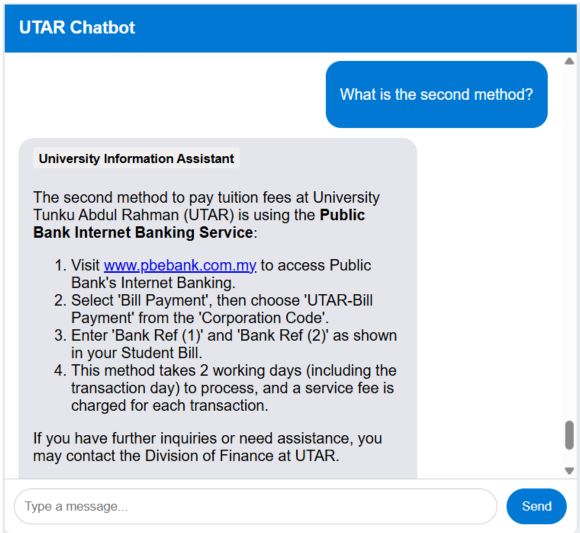
</td><td></td></tr>
<tr><td>S3</td><td>User: “How do I apply for admission?” User: “What about international students?”</td><td>Should remember the context of admission</td><td>Chatbot gave correct admission information for international students.




</td><td>Pass</td></tr>
</table>

<table>
<tr>
<td>S4</td>
<td>User: “What are the payment methods for tuition fee?”
User: “What is the second method?”</td>
<td>Should remember the context of payment methods for tuition fee</td>
<td>Chatbot described about the second method to pay for UTAR tuition fee.





</td>
<td>Pass</td>
</tr>
<tr>
<td>S5</td>
<td>User: “Tell me about scholarships.”
User: “Do they cover hostel fees?”</td>
<td>Should follow-up under Finance Agent</td>
<td>Chatbot provided scholarships information but the chatbot did not provide information about hostel fee coverage by scholarships.</td>
<td>Partial Pass</td>
</tr>
</table>

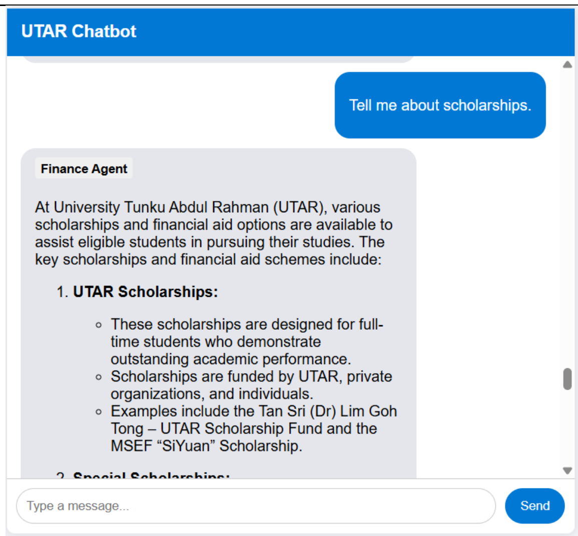
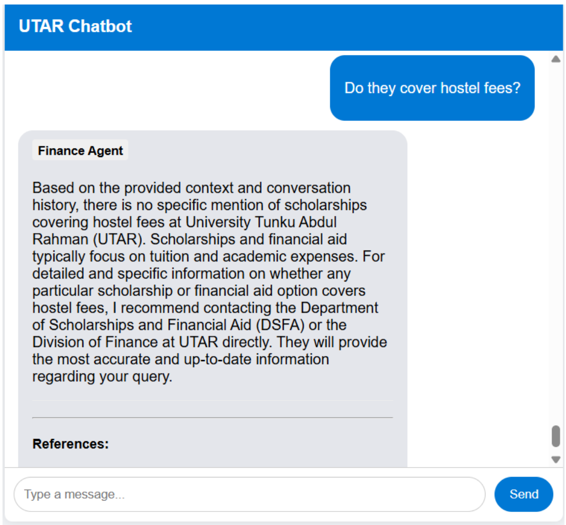
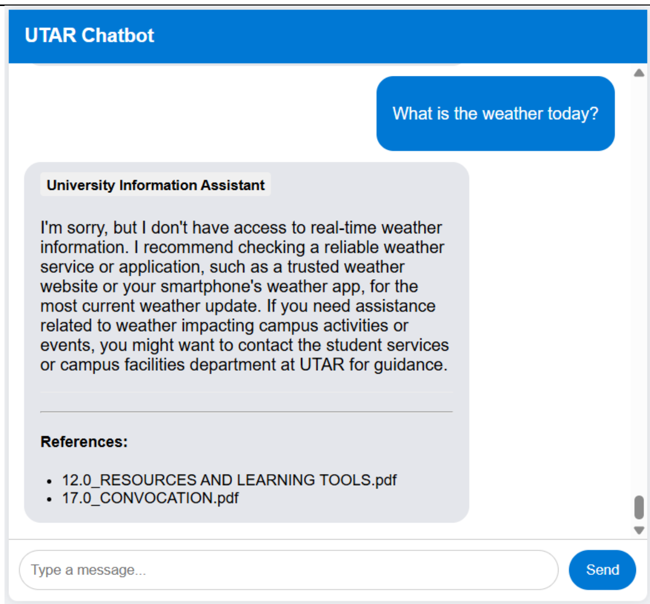
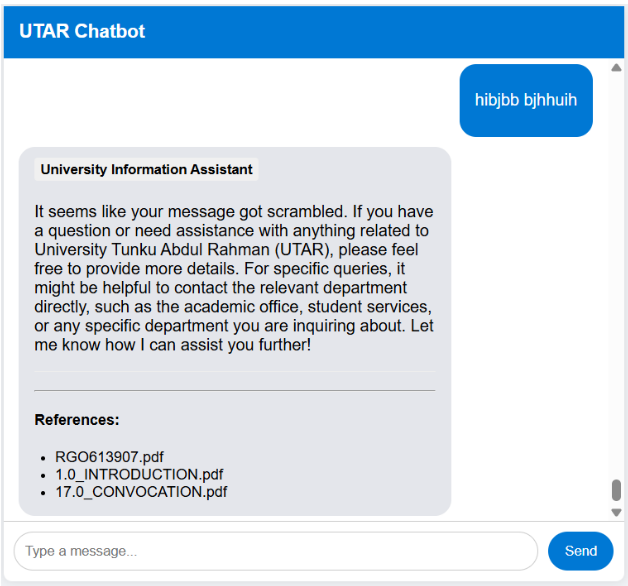
			 	

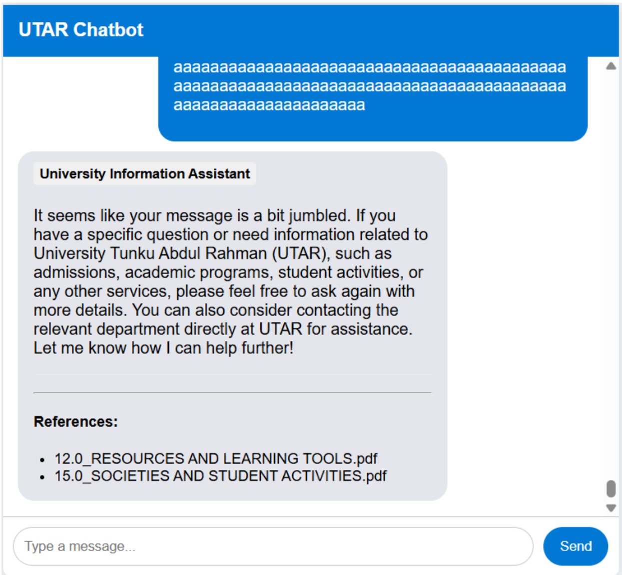
Table 6.2.2.1 Test cases for Session testing and their results.

5 test cases which were related to all available department and division agents were conducted to determine whether conversational context was maintained in a session and that the selected agent can provide correct response and the results were shown in Figure 6.2.2.1.

6.2.3 Error Handling

Test case ID	Input Query	Expected Behavior	Actual Behavior	Pass/Fail
E1	What is the	Chatbot should not	Chatbot gave fallback response.	Pass

	weather today?	have access to current weather information.		
E2	hibjbb bjhhuih	Chatbot should request further clarification.	Chatbot responded that it did not understand the question and required further clarification. 	Pass

E3	Overly long input which are more than five hundred words	Chatbot should handle the query gracefully without crashing	Chatbot responded that it did not understand the question and required further clarification. 	Pass
E4	Empty input	Chatbot should not send the empty input or input with whitespace only for processing	Chatbot did not provide any response if user tried to send empty input or input with only whitespace.	Pass
E5	pizza delivery at UTAR?	Chatbot should not have information about pizza delivery at UTAR	Chatbot did not have information about pizza delivery at UTAR and it directed users to refer to the relevant UTAR departments.	Pass

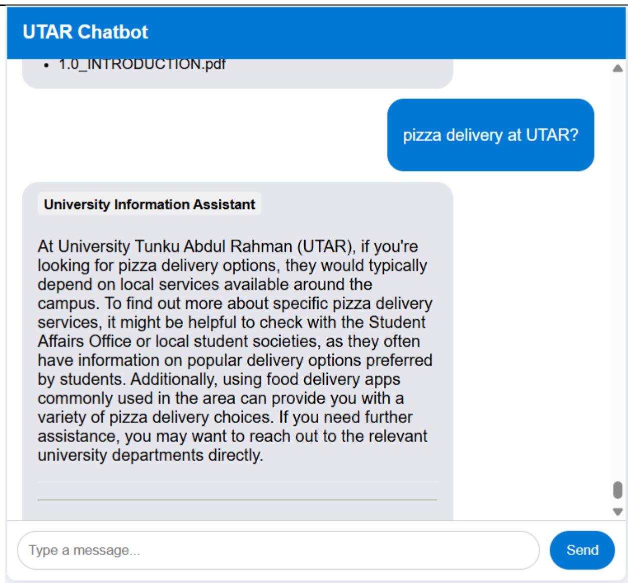
				

Table 6.2.3.1 Test cases for error handling and their results.

These 5 test cases were conducted to check whether the chatbot was able to handle inappropriate user queries and the results obtained were shown in Figure 6.2.3.1.

6.2.4 Concurrent User Testing

Test Case ID	User Role	Input Query	Expected Behavior	Actual Behavior	Pass/Fail
C1	Student A	What are the exam dates for October 2025 examination?	Examinations Agent should answer the query.	Examinations Agent provided correct response.	Pass
C2	Student B	How can I get admitted into UTAR?	Admissions Agent should answer the query.	Admissions Agent provided correct response.	Pass
C3	Student C	How can I apply for scholarships?	Finance Agent should answer the query.	Finance Agent provided correct response.	Pass

C4	Staff A	What are the facilities offered by UTAR?	General Agent should answer the query.	General Agent provided correct response.	Pass
C5	Three users sending queries to chatbot at the same time		Each user session remains independent and separate.	All user queries were handled independently and correctly.	Pass

Table 6.2.4.1 Test cases for concurrent user testing and their results.

The 5 test cases were conducted to check whether each user session was independent and whether many users can access different agents at the same time and the results were shown in Figure 6.2.4.1.

6.2.5 Performance Testing

Test Case ID	Specialized Agent	Input Query	Measured Response Time (seconds)	Expected Response Time (Less than 6 seconds)	Pass/Fail
R1	General Agent or University Information Assistant	Where is UTAR Sungai Long campus?	3.56	Yes	Pass
R2	Examinations Agent	Can I bring mobile phone to exam hall?	4.01	Yes	Pass
R3	Finance Agent	How can I pay for	8.27	No	Fail

		UTAR tuition fee?			
R4	Admissions Agent	What are the entry requirements for UTAR Foundation studies?	4.29	Yes	Pass
R5	Examinations Agent	How can I graduate from UTAR?	7.39	Yes	Fail

Table 6.2.5.1 Test cases for performance testing and their results.

The 5 test cases were conducted to measure the response time of the chatbot in providing responses to user queries and the results were shown in Figure 6.2.5.1. The average response was calculated as follows:

$$Average\ Response\ Time = \frac{3.56 + 4.01 + 8.27 + 4.29 + 7.39}{5}$$

= 5.50 seconds

6.2.6 Evaluation of Results of Five Type of Tests

Type of Tests	Number of Test Cases Run	Pass	Fail	Success Rate (%)
Functional Testing	20	18	2	90
Session Testing	5	4.5	0.5	95
Error Handling	5	5	0	100
Concurrent User Testing	5	5	0	100
Performance Testing	5	3	2	60

Table 6.2.6.1 Summarization of results obtained after performing five types of tests on the chatbot.

Table 6.2.6.1 shows the results obtained after conducting functional testing, session testing, error handling, concurrent user testing and performance testing on the deployed UTAR multi-agent RAG chatbot. For the tests except for the performance testing, the success rates were 90% and above. Performance testing had a success rate of quite low which was 60%. This might be due to the reason that the number of test cases run for performance test was less. More complex questions required longer time to retrieve more contextual information from vector databases. The two questions that took more than six seconds to generate response were likely complex questions. Besides that, the success rates for error handling and concurrent user testing were 100%, which was unrealistic. This might also be due to the small number of test cases. In future, more test cases should be carried out for all test types.

6.2.7 Evaluation of Google Form Survey Results

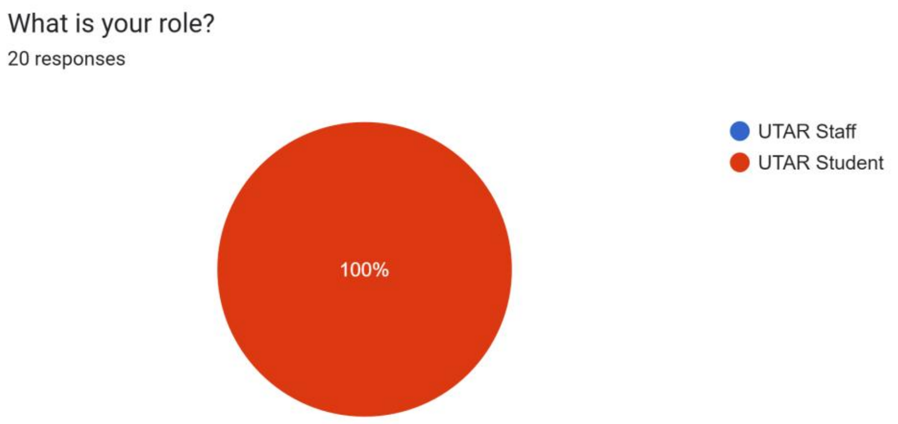


Figure 6.2.7.1 Pie chart shows the results of a survey question that asks for respondents' role.

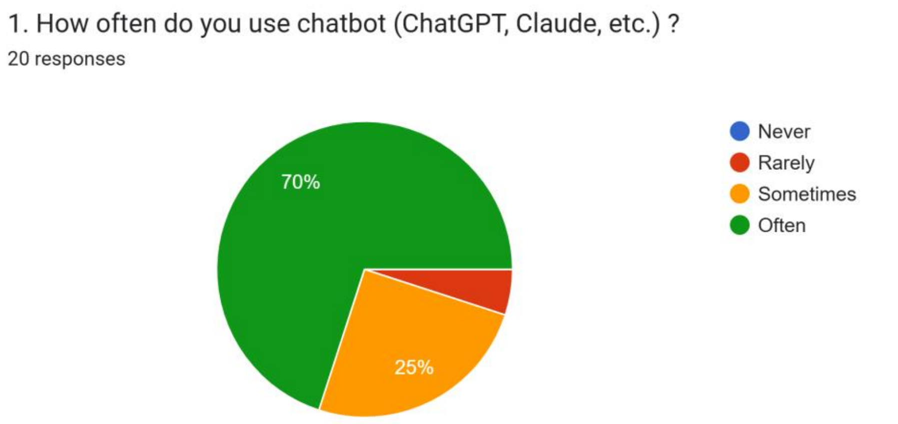


Figure 6.2.7.2 Pie chart shows the results of a survey question that asks how often the respondent using chatbot.

Google Form Survey Question	Average Score (Score 1 to 5)
2. The chatbot was easy to use.	4.40
3. I felt comfortable interacting with the chatbot.	4.40
4. The chatbot responded in a reasonable amount of time.	4.40
5. The chatbot understood my questions well.	4.10
6. The responses were relevant and accurate.	4.20
7. The chatbot provided useful information about the university.	3.95
8. The chatbot helped me find the information I was looking for.	4.00
9. Overall, I am satisfied with my experience using the chatbot.	4.15

Table 6.2.7.1 Calculated average score for each 5-point Likert scale question.

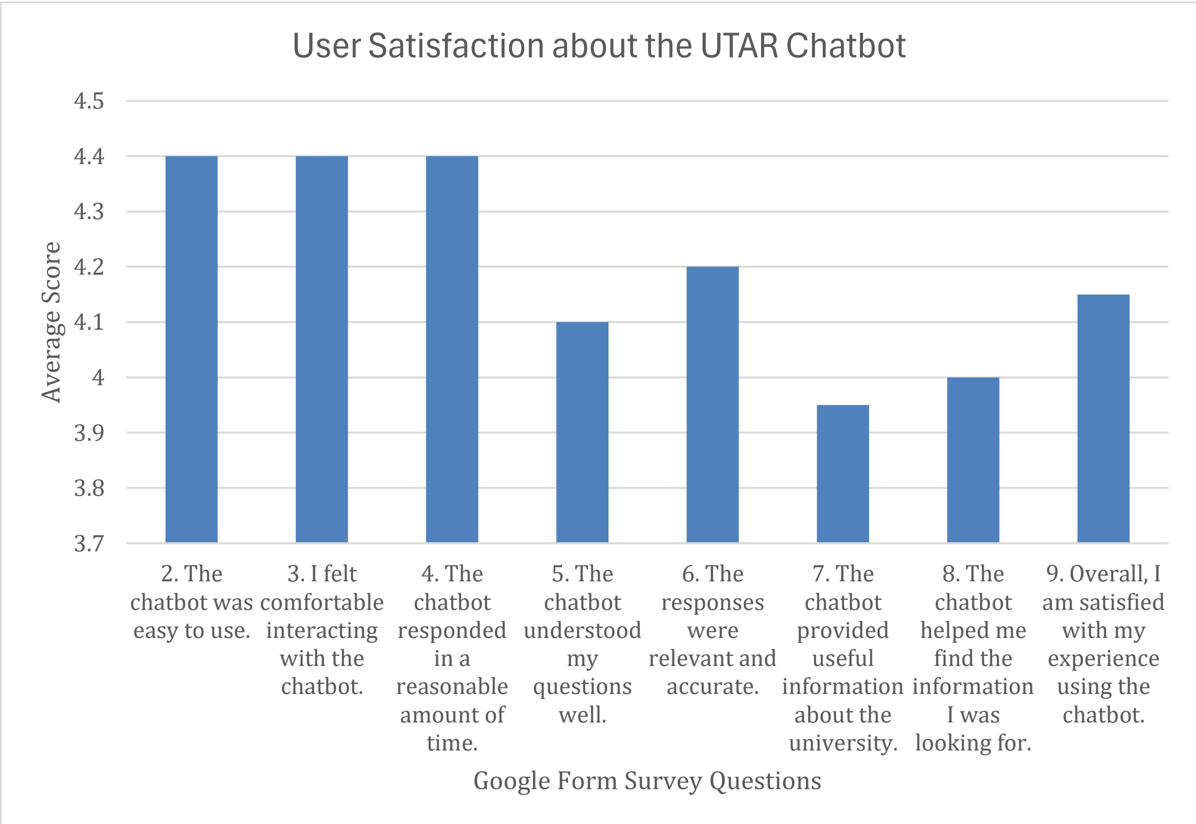


Figure 6.2.7.3 Bar chart that was plotted using calculated average scores in Table 6.2.7.1.

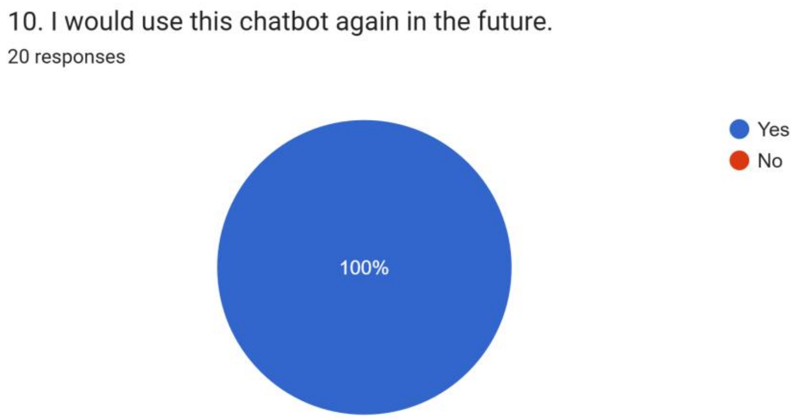


Figure 6.2.7.4 Pie chart shows the results of a Yes or No Google Form survey question.

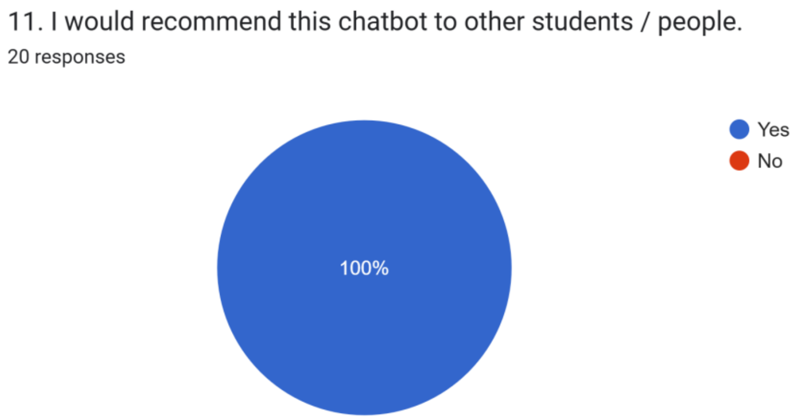


Figure 6.2.7.5 Pie chart shows the results of another Yes or No Google Form survey question.

Figure 6.2.7.1 shows that all the respondents who responded to the Google Form survey were UTAR students and there were total of 20 respondents that filled out the survey form. Figure 6.2.7.2 shows that 70% of respondents often used chatbots such as ChatGPT and Claude, 25% used chatbots sometimes, 5% rarely used chatbots and there were not any respondents who did not use chatbots before.

Table 6.2.7.1 shows Google Form survey questions from question number 2 to question number 9 which were 5-point Likert scale questions with respective calculated average score. Figure 6.2.7.3 shows the bar chart which was constructed using the results in Table 6.2.7.1. the bar chart shows that most respondents were satisfied with the chatbot after using it as most average scores of questions were above 4.00 and only one question which was question number

7 had average score of 3.95 which was less than 4.00. The average satisfaction score was calculated to be 4.2/5. The average satisfaction score and its percentage were calculated as follows:

$$Average\ satisfaction\ score = \frac{4.4 + 4.4 + 4.4 + 4.1 + 4.2 + 3.95 + 4 + 4.15}{8}$$

$$= 4.2$$

$$Percentage\ of\ User\ Satisfaction = \frac{4.2}{5.0} \times 100\%$$

$$= 84\%$$

Figure 6.2.7.4 shows that all respondents were willing to use the chatbot again in the future which mostly means that they were having good experience when using the chatbot. Figure 6.2.7.5 shows all users would suggest the chatbot to other students or people because they might think that the chatbot would benefit other students in searching for university-related information quickly and accurately.

Finally, there were three open-ended questions in the Google Form survey to ask the respondents what they liked most about the chatbot, what they disliked or found confusing as well as suggested improvements and additional features. The respondents stated that the chatbot was easy and convenient to use, provide accurate answers and the chatbot was able to provide fallback responses. On the other hand, some respondents found the chatbot provided explanations that were hard to understand in some cases and the accuracy of some answers provided by the chatbot was not believed to be completely true. The respondents suggested adding more department and division agents into the chatbot system, adding emotions to the chatbot and giving the chatbot access to UTAR webpages.

6.3 Project Challenges

There were many challenges faced when developing a UTAR multi-agent RAG chatbot system. The challenges include technical issues such as session management issues in which the conversational history was lost every time a user asked a question in the same session. The problem was found to be related to cookie restrictions and so the solution was to configure cookie to store session ID of a session and allow the cookie to be securely transferred over different platforms over HTTPS only. Furthermore, maintaining all the conversational history in a session may consume too much memory and may overload the model context so the chatbot system was designed to keep the last conversational exchanges between users and the chatbot for each session.

Besides that, most UTAR webpages were found to have broken SSL certificates for PDF downloads which would cause errors when Playwright trying to download the PDF documents from some UTAR webpages. To overcome the issues, the chatbot system was designed to bypass SSL verification specifically for UTAR domains while enforcing proper certificate validation using certify for all other domains since UTAR domains were mostly to be trusted source and free from malicious content. This ensured both reliability in accessing UTAR's content which included UTAR PDF documents and security when downloading from external sources.

Building and loading vector databases for multiple departments and divisions was time-consuming and consumed a lot of memory. Initially, the chatbot system employed preloading of all vector databases after successful deployment to ensure that every department and division agent was ready to provide instant response to user queries. This approach consumed a lot of RAM memory and caused RAM memory exhaustion which further caused the deployment of the chatbot to fail. The solution was to use lazy loading of vector database for each department and division. The chatbot system only loaded vector database for a department or division when there was a first query sent to the relevant agent for the first time.

There were also problems related to deployment of the chatbot system in which the Render backend had long startup times due to vector database extraction. The chatbot system was provided with zipped vector database files and the files were extracted to persistent disk on Render to be loaded during deployment. This process took significant amount of time. In addition, render.yaml was created and some settings, specifically the timeout and worker and thread settings need to be done correctly in order to prevent timeout issues during loading of vector databases. CORS setup was another problem that needed to be resolved in order for Firebase frontend to communicate with Flask backend.

There were also some challenges associated with user testing. There were only limited number of testers which were UTAR students and staff to test the chatbot system and some testers were unfamiliar with chatbots. This may affect the ratings of the chatbot system. There were only 20 respondents responding to the Google Form survey due to time constraints and this made the results less convincible. The open-ended feedback in the Google Form survey was difficult to analyse as the responses were highly varied.

6.4 Objectives Evaluation

The first project objective which was to develop a scalable, intelligent and robust multi-agent RAG chatbot system that is capable of providing contextually relevant and accurate responses to user queries related to the university, UTAR was met. The reason is because a UTAR multi-agent RAG chatbot has been successfully developed. The chatbot has been able to be used by UTAR students to get quick, accurate and contextually relevant university-related responses.

The second project objective was to achieve accuracy of at least 90% in user query resolution and user satisfaction. The accuracy of response was found to be exactly 90% as shown in Table 6.2.6.1 but the percentage of user satisfaction was only 84% which was less than 90%, so this means that the second project objective was partially met. The reason may be due to small number of test cases conducted and limited number of respondents for Google Survey form. This makes the percentage in response accuracy to be easily affected by one failed test case and the percentage in user satisfaction to be easily affected by a user's dissatisfaction.

Besides that, the third project objective was to reduce the average response time of the chatbot in response to user queries to 6s was met. Based on the results of performance testing, an average response time of 5.50 seconds was obtained, and this shows that the third objective was met. However, there were only limited number of test cases conducted, in the future, more test cases should be conducted in order to ensure that the results were convincible and accurate.

The fourth objective was to deploy the chatbot so that UTAR students and staff, students' parents and other people with Internet access are able to access the UTAR chatbot to get quick, correct and accurate information related to UTAR. The fourth objective was met as the chatbot was proven to be deployed in previous chapters and testing was already conducted on the deployed chatbot to evaluate it.

6.5 Concluding Remark

In summary, the testing and evaluation of the developed and deployed UTAR multi-agent RAG chatbot demonstrate that the chatbot system is functional and is able to meet its intended objectives. The combination of a Flask backend, a React frontend and deployment on Firebase and Render provided a stable and scalable foundation for real-world use. The chatbot is intended to benefit UTAR students and staff, students' parents and other people searching for information related to UTAR. Through the Google Form survey responses, the chatbot was proven to provide accurate responses to users and that most users were satisfied with the chatbot. Most respondents agreed that the chatbot was easy to use and provided accurate university-related information within reasonable amount of time.

With open-ended feedback provided by respondents, the advantages, limitations and areas for improvement for the chatbot were obtained. Users liked the easy-to-use and accessible chatbot interface while also providing some areas of improvement which were to add more specialized agents into the chatbot system and provide the chatbot with access to UTAR websites to get more information about UTAR which is useful in answering user queries. These insights provide a valuable direction for future development and improvement of the project.

Overall, the evaluation confirmed that the chatbot is able to answer most user queries accurately while maintaining conversational context through the use of conversational history. While there remain technical challenges such as improving handling of ambiguous questions and ensuring the university data sources remain up to date, the results indicate that the chatbot system is a viable solution to support UTAR students and staff in accessing university-related information more efficiently. The project serves as the foundation for the development of a powerful chatbot for UTAR equipped with knowledge of all departments and divisions in the future.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

This project sets out to address a recurring challenge faced by UTAR students, staff, prospective applicants at UTAR and parents in accessing accurate, timely and relevant university-related information in a convenient manner. Traditional channels such as university websites, phone calls to university receptionist desk and sending of email often result in delays and frustration. Motivated by the need for a more efficient information retrieval system, this project proposed and developed a multi-agent RAG chatbot system for UTAR, designed to serve as an intelligent and scalable solution for users to get university-related information conveniently by only accessing the chatbot.

The system combined several key components which were a Flask backend that orchestrates queries and manages conversation sessions, a React frontend deployed via Firebase for user interaction, and department and division vector databases that store scraped website and PDF content from UTAR's official sources. Specialized agents namely Admissions Agent, Finance Agent, and Examinations Agent, were implemented to handle department-specific queries, while a General Agent handled broader university-related questions. Deployment of the backend was carried out using Render [8], ensuring scalability and 24/7 availability.

The system was tested and evaluated using both technical testing and user-based evaluation through a Google Form survey. Performance testing confirmed that the chatbot maintained an average response time close to the targeted six seconds, even under concurrent user load. The system also demonstrated robustness in handling multiple queries without significant service degradation. The survey collected responses from UTAR students and staff. Results indicated that most users found the chatbot easy to use, felt comfortable interacting with it, and agreed that the chatbot's responses were accurate, useful and relevant. Importantly, a majority of respondents expressed satisfaction with their experience in using the chatbot and indicated that they would be willing to use the chatbot again in the future.

These outcomes demonstrate that the primary objectives of the project were largely achieved. The system succeeded in delivering a scalable, intelligent, and robust chatbot that supports multiple users concurrently while maintaining contextual accuracy. It also provided response times that were within acceptable limits, thereby meeting user expectations for efficiency.

In conclusion, this project successfully developed a multi-agent chatbot tailored for UTAR that not only addresses the need for accessible and accurate university-related information but also contributes a scalable prototype for intelligent information systems in higher education.

7.2 Recommendation

Although the system met its primary goals, further work can improve its robustness and adoption. In the future, the knowledge base can be scaled and expanded by adding more department and division agents into the chatbot system and integrate APIs and real-time data sources from UTAR websites to keep the chatbot's knowledge up to date. This allows the chatbot to automatically retrieves the latest UTAR information online to be used in answering user queries.

Besides that, the agent collaboration module which was not implemented in this project due to time constraints can be implemented in the future. This module allows multiple agents to involve in generating responses which will then be combined by the agent orchestrator before sending the final response to the user. This module is useful when a user asks a question that requires the contextual information from two departments or divisions.

Besides that, the chatbot can be integrated into the UTAR official websites to allow all users browsing to UTAR website to access the chatbot. The chatbot can be embedded within the UTAR website to be used by anyone or embedded within UTAR student and staff portals to only allow UTAR students and staff to access the UTAR chatbot with login credentials. The integration of UTAR chatbot into UTAR student portal can ensure the confidentiality of university data since the chatbot may be compromised by attackers to obtain UTAR sensitive information. However, this limits the public access to UTAR chatbot.

An interface can be designed for data owner or system administrators to upload UTAR PDF documents to the chatbot system so that the chatbot system can use the latest and new UTAR PDF documents to improve its knowledge and to be able to answer more questions.

REFERENCES

[1] B. Alsafari, E. Atwell, A. Walker, and M. Callaghan, “Towards effective teaching assistants: from intent-based chatbots to LLM-powered teaching assistants,” *Natural Lang. Process. J.,* Aug. 2024, doi: https://doi.org/10.1016/j.nlp.2024.100101

[2] J. Gao et al., “Retrieval augmented generation (RAG) based restaurant chatbot with AI testability,” in *The 10th IEEE Int. Conf. on Big Data Comput. and Mach. Learn. Appl.*, Shanghai, China, Jul. 2024, [Online]. Available: https://www.researchgate.net/publication/381461839_Retrieval_Augmented_Generation_RAG_based_Restaurant_Chatbot_with_AI_Testability

[3] OpenAI, “ChatGPT,” Accessed: Sep. 5, 2024. [Online]. Available: https://openai.com/chatgpt/.

[4] G. Gamage *et al.*, “Multi-Agent RAG Chatbot Architecture for Decision Support in Net-Zero Emission Energy Systems,” in *2024 IEEE International Conference on Industrial Technology (ICIT)*, Bristol, United Kingdom, Mar. 2024, pp. 1-6, doi: 10.1109/ICIT58233.2024.

[5] D. Calvaresi, J. Calbimonte, F. Dubosson, A. Najjar, and M. Schumacher, “Social network chatbots for smoking cessation: agent and multi-agent frameworks,” in *IEEE/WIC/ACM Int. Conf. on Web Intell.*, New York, USA, Oct. 2019, pp. 286-292, doi: 10.1145/3350546.3352532.

[6] C. Clarke, K. Krishnamurthy, W. Talamonti, Y. Kang, L. Tang, and J. Mars, “One agent too many: user perspectives on approaches to multi-agent conversational AI,” *J. of Latex Class Files*, vol. 14(8), Aug. 2021, doi:10.48550/arXiv.2401.07123.

[7] B. Lutkevich, “Knowledge base,” TechTarget. Accessed: Sep. 7, 2024. [Online]. Available: https://www.techtarget.com/whatis/definition/knowledge-base.

[8] J. Codella, S. Andrews, T. Shibamura, and R. Millsap, “Vector search in Azure Cosmos DB for NoSQL (preview),” Microsoft. Accessed: Sep. 7, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/vector-search#what-is-a-vector-store.

[9] Amazon Web Services, “What is RAG (retrieval-augmented generation)?,” Accessed: Sep. 7, 2024. [Online]. Available: https://aws.amazon.com/what-is/retrieval-augmented-generation/.

[10] M. Syed, and E. Russi, “What is vector search?,” IBM. Accessed: Sep. 7, 2024. [Online]. Available: https://www.ibm.com/topics/vector-search.

[11] LangChain, “Introduction,” Accessed: Sep. 8, 2024. [Online]. Available: https://python.langchain.com/v0.2/docs/introduction/.

[12] Ollama, “llama3.1,” Accessed: Sep. 8, 2024. [Online]. Available: https://ollama.com/library/llama3.1.

[13] T. T. Nguyen, A. D. Le, H. T. Hoang, and T. Nguyen, “NEU-chatbot: chatbot for admission of National Economics University,” *Comput. and Educ.: Artif. Intell.*, vol. 2, Oct. 2021, doi: https://doi.org/10.1016/j.caeai.2021.100036.

[14] A. Salve, S. Attar, M. Deshmukh, S. Shivpuje, and A. M. Utsab, "A collaborative multi-agent approach to retrieval-augmented generation across diverse data," 2024. [Online]. Available: https://arxiv.org/abs/2412.05838.

[15] F. Liu, Z. Kang, and X. Han, "Optimizing RAG Techniques for Automotive Industry PDF Chatbots: A Case Study with Locally Deployed Ollama Models ," 2024. [Online]. Available: https://arxiv.org/abs/2408.05933.

[16] I. A. Pap, and S. Oniga, "eHealth Assistant AI Chatbot Using a Large Language Model to Provide Personalized Answers through Secure Decentralized Communication," *Sensors*, vol. 24(18), Sept. 2024, doi: https://doi.org/10.3390/s24186140.

[17] H. Farias, J. G. Aroca, and D. Ortiz, "Chatbot Based on Large Language Model to Improve Adherence to Exercise-Based Treatment in People with Knee Osteoarthritis: System Development," *Technologies*, vol. 13(4), 140, Apr. 2025, doi: https://doi.org/10.3390/technologies13040140.

[18] Microsoft. "What is Azure OpenAI Service." Accessed: Apr. 28, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/ai-services/openai/overview.

[19] A. Khan, "Design and implementation of an automated web scraping system: enhancing accuracy and efficiency for Nettileasing Finland OY," M.S. thesis, LUT School of Engineering Science. 2023.

[20] M. A. Wahed, M. S. Alzboon, M. Alqaraleh, J. Ayman, M. Al-Batah, and A. F. Bader, "Automating web data collection: challenges, solutions, and python-based strategies for effective web scraping," 2024 7th Int. Conf. on Internet Appl, Protocols, and Services (NETAPPS), Kuala Lumpur, Malaysia, 2024, pp. 1-6, doi: https://doi.org/10.1109/NETAPPS63333.2024.10823528.

[21] I. Naing, S. T. Aung, K. H. Wai, and N. Funabiki, "A reference paper collection system using web scraping," *electronics*, vol. 13(14), Jul. 2024, doi: https://doi.org/10.3390/electronics13142700.

[22] A. Maharana, D. H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang, "Evaluating very long-term conversational memory of LLM Agents," 2024. Available: https://arxiv.org/abs/2402.17753.

[23] Flask-CORS, "*Flask-CORS Documentation*," Accessed: Sept. 22, 2025. [Online]. Available: https://flask-cors.readthedocs.io/en/latest/configuration.html.

[24] Flask-Session, "*Flask-Session Documentation*," Accessed: Sept. 22, 2025. [Online]. Available: https://flask-session.readthedocs.io/en/latest/config.html.

[25] GeeksforGeeks, "How to use Flask-Session in Python Flask?," *GeeksforGeeks*, Accessed: Sept. 22, 2025. [Online]. Available: https://www.geeksforgeeks.org/python/how-to-use-flask-session-in-python-flask/.

[26] GeekPython, "What are Sessions? How to use Sessions in Flask," *GeekPython*. Accessed: Sept. 22, 2025. [Online]. Available: https://geekpython.in/how-to-use-sessions-in-flask.

[27] React-Markdown Team, "*react-markdown: Markdown component for React*," GitHub. Accessed: Sept. 22, 2025. [Online]. Available: https://github.com/remarkjs/react-markdown.

[28] Create React App, *"Creating a Production Build."* Accessed: Sept. 22, 2025. [Online]. Available: https://create-react-app.dev/docs/production-build/.

[29] Firebase, *"Get started with Firebase Hosting — Quickstart,"* Accessed: Sept. 22, 2025. [Online]. Available: https://firebase.google.com/docs/hosting/quickstart.

[30] Firebase, *"Configure Hosting behavior — Firebase Hosting,"* Accessed: Sept. 22, 2025. [Online]. Available: https://firebase.google.com/docs/hosting/full-config.

[31] Render, *"Flask Deployment Guide — Render Documentation,"* Accessed: Sept. 22, 2025. [Online]. Available: https://render.com/docs/deploy-flask.

[32] Render, *"Web Services — Render Documentation,"* Accessed: Sept. 22, 2025. [Online]. Available: https://render.com/docs/web-services.

[33] M. McGrath, *Cloud Deployment Simplified: Serverless, Containers, and Render*. New York, NY, USA: Apress, 2021.

APPENDIX A

A.1 Base Agent Class Implementation

```
class BaseAgent:
    """Base agent class with common functionality"""

    def __init__(self, name, description, vector_db_path=None,
department=None, urls=[]):
        self.name = name
        self.description = description
        self.vector_db = None
        self.vector_db_path = vector_db_path
        self.department = department
        self.urls = urls
        self._is_initialized = False  # Track initialization state

    def initialize(self):
        """Load the Vector Database for the current agent (lazy
loading)"""
        if self._is_initialized:
            logging.info(f"Agent {self.name} already initialized,
skipping...")
            return

        if self.vector_db_path:
            logging.info(f"Lazy loading vector database for
{self.name}...")
            self.vector_db = self._load_vector_db()
            self._is_initialized = True
            if self.vector_db:
                logging.info(f"Successfully loaded vector database
for {self.name}")
            else:
                logging.warning(f"Failed to load vector database for
{self.name}")

    def scrape_webpage(self, urls):
        print("\nScrapping some UTAR Webpages.")
        seen_links = set()
        results = []

        with sync_playwright() as p:
            browser = p.chromium.launch()

            for url in urls:
                page = browser.new_page()
                page.goto(url)
```

```
                page.wait_for_timeout(3000)  # wait for JS to load
                html_content = page.content()
                page.close()

                soup = BeautifulSoup(html_content, 'html.parser')

                # ---- Collect Text ----
                page_text_parts = []
                for div in soup.find_all('div', class_='mg'):
                    section_text = div.get_text(separator='\n',
strip=True)
                    if section_text:
                        page_text_parts.append(section_text)

                # ---- Collect Unique Links ----
                seen_links = set()
                link_texts = []
                link_metadata_list = []
                for link in soup.find_all('a', href=True):
                    full_url = urljoin(url, link['href'])
                    if full_url not in seen_links:  # ensure
uniqueness
                        seen_links.add(full_url)
                        text = link.get_text(strip=True)
                        if text:
                            link_texts.append(text)
                            link_metadata_list.append({"text": text,
"url": full_url})

                # ---- Decide How to Combine ----
                if not page_text_parts:  # if no main text, combine
link texts
                    combined_text = " | ".join(link_texts)
                else:
                    combined_text = "\n".join(page_text_parts)
                    if link_texts:
                        combined_text += "\nLinks: " + " |
".join(link_texts)

                results.append(
                    Document(
                        page_content=combined_text,
                        metadata={"source":url}
                    )
                )

            browser.close()
            print("\nDone Scraping UTAR Webpages.")
```

```
        return results

    def scrape_web_pdfs(self, urls, department,
base_folder="/var/data"):
        """
        Scrapes a webpage for all linked PDFs and downloads them
into a department folder.
        Handles UTAR's broken SSL for PDFs specifically.
        """
        print("Looking for PDF files in some UTAR Webpages to
download...")
        print("department", department)
        download_folder = os.path.join(base_folder, department)
        os.makedirs(download_folder, exist_ok=True)

        pdf_files = []

        # Use Playwright to fetch rendered HTML
        with sync_playwright() as p:
            browser = p.chromium.launch()

            for url in urls:
                page = browser.new_page()
                page.goto(url)
                page.wait_for_timeout(3000)  # wait for JS to load
                html_content = page.content()
                page.close()

                soup = BeautifulSoup(html_content, 'html.parser')

                # Download PDFs only
                for link in soup.find_all('a', href=True):
                    full_url = urljoin(url, link['href'])
                    if full_url.lower().endswith(".pdf"):
                        file_name = os.path.basename(full_url)
                        pdf_path = os.path.join(download_folder,
file_name)

                        if os.path.exists(pdf_path):
                            print(f"Skipping (already exists):
{file_name}")
                            continue

                        try:
                            if "utar.edu.my" in full_url.lower():
                                # Bypass SSL verification for UTAR
                                r = requests.get(full_url,
stream=True, verify=False, timeout=10)
                            else:
```

```
                                r = requests.get(full_url,
stream=True, verify=certifi.where(), timeout=10)
                            r.raise_for_status() # Check HTTP
response for errors to avoid downloading broken files
                            with open(pdf_path, "wb") as f: # Open
pdf in binary write mode
                                for chunk in r.iter_content(8192):
                                    if chunk:
                                        f.write(chunk)
                            pdf_files.append(pdf_path)
                            print(f"Downloaded PDF: {pdf_path}")
                        except Exception as e:
                            print(f"Failed to download {full_url}:
{e}")

            browser.close()

        return pdf_files


    def ingest_pdf(self, doc_folder_path):
        scraped_pdf_files = self.scrape_web_pdfs(self.urls,
self.department)

        print(f"Looking for PDFs in: {doc_folder_path}")
        if not os.path.exists(doc_folder_path):
            logging.error(f"Folder not found: {doc_folder_path}")
            return None

        all_data = []
        for pdf_file in glob.glob(os.path.join(doc_folder_path,
"*.pdf")):
            try:
                print(f"Loading PDF: {pdf_file}")
                loader = UnstructuredPDFLoader(file_path=pdf_file)
                data = loader.load()

                # add source metadata
                for doc in data:
                    doc.metadata["source"] =
os.path.basename(pdf_file)

                all_data.extend(data)
            except Exception as e:
                logging.error(f"Failed to load {pdf_file}: {e}")

        return all_data

    def split_documents(self, documents):
```

```
        text_splitter =
RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=200)
        return text_splitter.split_documents(documents)

    def _load_vector_db(self):

        """Load vector database from the defined path"""
        try:
            if os.path.exists(self.vector_db_path):
                print(f"Loading vector database for
{self.department} from {self.vector_db_path}...")

                vector_database = Chroma(
                    persist_directory=self.vector_db_path,
                    embedding_function=embedding_model
                )
            else:
                print(f"Creating vector database for
{self.department}...")
                doc_folder_path = f"/var/data/{self.department}"

                # Load PDF data
                pdf_data = self.ingest_pdf(doc_folder_path)
                if pdf_data is None:
                    return None

                # Scrape data from UTAR website
                scraped_data = self.scrape_webpage(self.urls)

                # Combine both
                combined_texts = []
                if pdf_data:
                    combined_texts.extend(pdf_data)
                if scraped_data:
                    combined_texts.extend(scraped_data)

                # Chunk the combined text content
                chunks = self.split_documents(combined_texts)

                for i, chunk in enumerate(chunks[:5]):
                    print(f"Chunk {i+1}")
                    print("Text:", chunk.page_content[:200])
                    print("Source:", chunk.metadata.get("source"))
                    print("Metadata:", chunk.metadata)
                    print("------")

                vector_database = Chroma.from_documents(
                    documents=chunks,
                    embedding=embedding_model,
```

```
                    persist_directory=self.vector_db_path
                )

            return vector_database

        except Exception as e:
            logging.error(f"Failed to load Vector Database for
{self.name}: {e}")
            return None

    def retrieve_context(self, query, k=3):
        """Retrieve context relevant and related to the user
query"""
        if not self.vector_db:
            logging.warning(f"There is no Vector Database available
for {self.name}")
            return []

        try:
            docs = self.vector_db.similarity_search(query, k=k)
            return docs
        except Exception as e:
            logging.error(f"Failed to retrieve context for the
query: {e}")
            return []

    def generate_response(self, query, contexts, history):
        """Generate a response based on the query and contexts"""
        # Base implementation
        return f"Hello, I'm {self.name}. I don't have specific
information related and relevant to the context of the query."
```

A.2 Admissions Agent Class Implementation

```
class AdmissionsAgent(BaseAgent):
    """An Agent class which is specialized in handling admissions
queries and questions"""

    def __init__(self):
        super().__init__(
            name="Admissions Agent",
            description="Handles admissions-related queries.",
            vector_db_path="/var/data/vector_db/admissions",
            department="Division of Admissions and Credit
Evaluation",
```

```
            urls=[
                "https://admission.utar.edu.my/About_DACE.php",
                "https://admission.utar.edu.my/Entry-Qualifications-
and-English-Language-Requirements.php"
            ]
        )

    def generate_response(self, query, contexts, history):
        """Generate responses that is related to admissions
queries"""
        if not contexts:
            return {
                "response": "I don't have specific information about
that admissions question. Please contact the Division of Admissions
directly.",
                "references": []
            }

        # Retrieve and store references used to generate a response
        references = []
        for doc in contexts:
            source = doc.metadata.get("source", None)
            if source and source not in references:
                references.append(source)

        # Retrieve conversation history of the session
        formatted_history = "\n".join([f"{msg['role'].capitalize()}:
{msg['content']}" for msg in history[-6:]])

        combined_context = "\n\n---\n\n".join(doc.page_content for
doc in contexts)
        prompt = f"""You are an admissions assistant at a university
named University Tunku Abdul Rahman or UTAR. Your name is
{self.name}.
        Use the following context to answer the question concisely
and helpfully, you need to answer the question
        based on context.

        Conversation history:
        {formatted_history}

        Context:
        {combined_context}

        Question: {query}

        Respond as a knowledgeable admissions professional. Be
helpful but concise.
```

```
        Only answer based on the given context and the given
conversation history.
        When interpreting questions, refer back to the conversation
history to resolve pronouns or implied references.
        If you cannot find an answer, politely tellthe user to
contact the Division of Admissions and Credit Evaluation.
        """

        try:
            response = chat_client.chat.completions.create(
                model=OPENAI_MODEL_NAME,
                messages=[
                    {"role": "system", "content": "You are a helpful
university admissions assistant."},
                    {"role": "user", "content": prompt}
                ]
            )
            return {
                "response":
response.choices[0].message.content.strip(),
                "references": references
            }
        except Exception as e:
            logging.error(f"Failed to generate response: {e}")
            return {
            "response": "Sorry, something went wrong while
generating the answer.",
            "references": []
        }
```

A.3 Examinations Agent Class Implementation

```
class ExaminationAgent(BaseAgent):
    """An Agent class which is specialized in handling examination,
academic and course queries"""

    def __init__(self):
        super().__init__(
            name="Examinations Agent",
            description="Handles course and exam queries", ##TODO
handle the problem when convocation is directed to exam agent
            vector_db_path="/var/data/vector_db/examinations",
            department="Department of Examination and Awards",
            urls=[
                "https://deas.utar.edu.my/Announcement.php",
```

```
                "https://deas.utar.edu.my/Home.php"
            ]
        )

    def generate_response(self, query, contexts, history):
        """Generate academic-specific responses"""
        if not contexts:
            return {
                "response": "I don't have specific information about
that academic question. Please contact the Department of Examination
and Awards directly.",
                "references": []
            }

        # Retrieve and store references used to generate a response
        references = []
        for doc in contexts:
            source = doc.metadata.get("source", None)
            if source and source not in references:
                references.append(source)

        # Retrieve conversation history of the session
        formatted_history = "\n".join([f"{msg['role'].capitalize()}:
{msg['content']}" for msg in history[-6:]])

        print("Formatted history for agent: ", formatted_history)

        combined_context = "\n\n---\n\n".join(doc.page_content for
doc in contexts)
        prompt = f"""You are an academic coordinator at a university
named University Tunku Abdul Rahman or UTAR. Your name is
{self.name}.
        Use the following context to answer the question clearly and
informatively.

        Conversation history:
        {formatted_history}

        Context:
        {combined_context}

        Question: {query}

        Respond as a knowledgeable academic professional. Be
educational but approachable.
        Only answer based on the given context and based on the
given conversation history.
        When interpreting questions, refer back to the conversation
history to resolve pronouns or implied references.
```

```
        If you can't find an answer, politely direct the user
        to contact the Department of Examination and Awards.
        """

        try:
            response = chat_client.chat.completions.create(
                model=OPENAI_MODEL_NAME,
                messages=[
                    {"role": "system", "content": "You are a helpful
university academic coordinator."},
                    {"role": "user", "content": prompt}
                ]
            )
            return {
                "response":
response.choices[0].message.content.strip(),
                "references": references
            }
        except Exception as e:
            logging.error(f"Failed to generate response: {e}")
            return {
            "response": "Sorry, something went wrong while
generating the answer.",
            "references": []
        }
```

A.4 Finance Agent Class Implementation

```
class FinanceAgent(BaseAgent):
    """An Agent class which is specialized in handling finance
queries"""

    def __init__(self):
        super().__init__(
            name="Finance Agent",
            description="Handles finance, fees, and scholarship
queries.",
            vector_db_path="/var/data/vector_db/finance",
            department="Division of Finance",
            urls=[
                "https://dfn.utar.edu.my/DFN.php",
                "https://dfn.utar.edu.my/DFN-3.php"
            ]
        )
```

```
    def generate_response(self, query, contexts, history):
        """Generate finance-specific responses"""
        if not contexts:
            return {
                "response": "I don't have specific information about
that financial question. Please contact the Division of Finance
directly.",
                "references": []
            }

        # Retrieve and store references used to generate a response
        references = []
        for doc in contexts:
            source = doc.metadata.get("source", None)
            if source and source not in references:
                references.append(source)

        # Retrieve conversation history of the session
        formatted_history = "\n".join([f"{msg['role'].capitalize()}:
{msg['content']}" for msg in history[-6:]])

        combined_context = "\n\n---\n\n".join(doc.page_content for
doc in contexts)
        prompt = f"""You are a financial advisor at a university
named University Tunku Abdul Rahman or UTAR. Your name is
{self.name}.
        Use the following context to answer the question precisely
and accurately.

        Conversation history:
        {formatted_history}

        Context:
        {combined_context}

        Question: {query}

        Respond as a precise and detail-oriented finance
professional. Mention specific
        numbers and dates when available. Only answer based on the
given context and the given conversation history.
        When interpreting questions, refer back to the conversation
history to resolve pronouns or implied references.
        If you can't find an answer, politely direct the user to
contact the Division of Finance.
        """

        try:
            response = chat_client.chat.completions.create(
```

```
                model=OPENAI_MODEL_NAME,
                messages=[
                    {"role": "system", "content": "You are a precise
university financial advisor."},
                    {"role": "user", "content": prompt}
                ]
            )
            return {
                "response":
response.choices[0].message.content.strip(),
                "references": references
            }
        except Exception as e:
            logging.error(f"Failed to generate response: {e}")
            return {
            "response": "Sorry, something went wrong while
generating the answer.",
            "references": []
        }
```

A.5 General Agent Class Implementation

```
class GeneralAgent(BaseAgent):
    """General agent for handling queries that do not fit any
specific department"""

    def __init__(self):
        super().__init__(
            name="University Information Assistant",
            description="General knowledge about the university",
            vector_db_path="/var/data/vector_db/general",
            department="General"
        )

    def generate_response(self, query, contexts, history):
        """Generate general responses for the user query"""

        # Retrieve and store references used to generate a response
        references = []
        for doc in contexts:
            source = doc.metadata.get("source", None)
            if source and source not in references:
                references.append(source)

        # Retrieve conversation history of the session
```

```
        formatted_history = "\n".join([f"{msg['role'].capitalize()}:
{msg['content']}" for msg in history[-6:]])

        prompt = f"""You are a general university information
assistant for a university named University Tunku Abdul Rahman or
UTAR. Your name is {self.name}.
        Use the following context and conversation history to answer
the question concisely and helpfully, you need to answer the
question
        based on context.

        Conversation history:
        {formatted_history}

        Question: {query}

        Respond as a helpful university assistant. For this query,
if you do not have any specific information,
        then you should provide a general response and suggest which
department might help.
        """

        try:
            response = chat_client.chat.completions.create(
                model=OPENAI_MODEL_NAME,
                messages=[
                    {"role": "system", "content": "You are a helpful
university information assistant."},
                    {"role": "user", "content": prompt}
                ]
            )
            return {
                "response":
response.choices[0].message.content.strip(),
                "references": references
            }
        except Exception as e:
            logging.error(f"Failed to generate response: {e}")
            return {
            "response": "Sorry, something went wrong while
generating the answer.",
            "references": []
        }
```

A.6 Agent Orchestrator Class Implementation

```
class AgentOrchestrator:
    """An agent that manages multiple specialized agents and routes
queries to the appropriate one"""


    def __init__(self):
        self.agents = []
        self.initialize_agents()


    def initialize_agents(self):
        """Initialize all available agents"""
        # Add specialized agents
        self.agents.append(AdmissionsAgent())
        self.agents.append(FinanceAgent())
        self.agents.append(ExaminationAgent())

        # Add the general agent as the last candidate to answer
query if no relevant and related specified agents found
        self.agents.append(GeneralAgent())

        # Initialize each agent
        for agent in self.agents:
            logging.info(f"Initialized agent: {agent.name}")

    def get_agent_for_query(self, query):
        """Find the most appropriate agent to handle a query using
LLM"""
        try:
            # Create agent descriptions for the LLM
            agent_descriptions = []
            for i, agent in enumerate(self.agents):
                agent_descriptions.append(f"Agent {i+1}:
{agent.name} - {agent.description}")

            agent_info = "\n".join(agent_descriptions)

            prompt = f"""You are a router that determines which
university agent should handle a user query.

            Available agents:
            {agent_info}

            User query: "{query}"

            ROUTING INSTRUCTIONS:
```

```
            1. Examine both the TOPIC and CONTEXT of the query
carefully
            2. Look for department-specific keywords and subjects
(admissions, finance/fees/scholarships, exams/courses)
            3. If the query relates to a department's core
responsibility area, route to that department EVEN IF some terms are
unfamiliar
            4. Examples of routing logic:
            - Questions about exam procedures, exam rules, exam
requirements, or anything happening during exams → Department of
Examination and Awards
            - Questions about admissions process, applications,
entry requirements → Division of Admissions
            - Questions about fees, payments, scholarships,
financial aid → Division of Finance
            5. Only route to the General Agent if the query clearly
doesn't relate to the core responsibilities of any specialized
department

            Based on these instructions, respond ONLY with the
appropriate agent designation (e.g., "Agent 1", "Agent 2", etc.)
without any explanation.
            """

            # Call the LLM to determine the appropriate agent
            response = chat_client.chat.completions.create(
                model=OPENAI_MODEL_NAME,
                messages=[
                    {"role": "system", "content": "You are a helpful
router assistant that determines which specialized agent should
handle a query."},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.0,  # Use low temperature for more
deterministic results
                max_tokens=10     # We only need a short response
            )

            agent_selection =
response.choices[0].message.content.strip().lower()

            # Extract the agent number from the response
            try:
                if "agent 1" in agent_selection:
                    selected_index = 0
                elif "agent 2" in agent_selection:
                    selected_index = 1
                elif "agent 3" in agent_selection:
                    selected_index = 2
```

```
                else:
                    # Default to the general agent
                    selected_index = 3

                selected_agent = self.agents[selected_index]
                logging.info(f"LLM selected agent:
{selected_agent.name}")
                return selected_agent

            except (ValueError, IndexError) as e:
                logging.error(f"Error parsing agent selection: {e}.
Using general agent.")
                return self.agents[-1]  # Return the general agent
as fallback

        except Exception as e:
            logging.error(f"Error in LLM agent selection: {e}")
            # Fallback to the general agent if there's any error
            return self.agents[-1]

    def process_query(self, query, history):
        """Process a user query through the appropriate agent"""
        # Select the appropriate agent
        agent = self.get_agent_for_query(query)
        logging.info(f"Selected agent: {agent.name}")

        # Lazy load the agent's vector database if not already
loaded
        if not agent.vector_db:
            logging.info(f"Loading vector database for
{agent.name}...")
            agent.initialize()

        # Get context information and data from the agent's
knowledge base
        contexts = agent.retrieve_context(query)

        # Generate and return the response from the agent
        return {
            "agent_name": agent.name,
            "agent_description": agent.description,
            "response": agent.generate_response(query, contexts,
history)
        }

    # def preload_all_databases(self):
    #     """Preload all vector databases for faster response
times"""
    #     for agent in self.agents:
```

```
    #         agent.initialize()

    def preload_all_databases(self):
        """DEPRECATED: This method is kept for backward
compatibility but does nothing"""
        logging.warning("preload_all_databases() is deprecated.
Using lazy loading instead.")
        pass
```

A.7 Google Form Survey Questions

Section 2 of 7

Section 1: Demographics

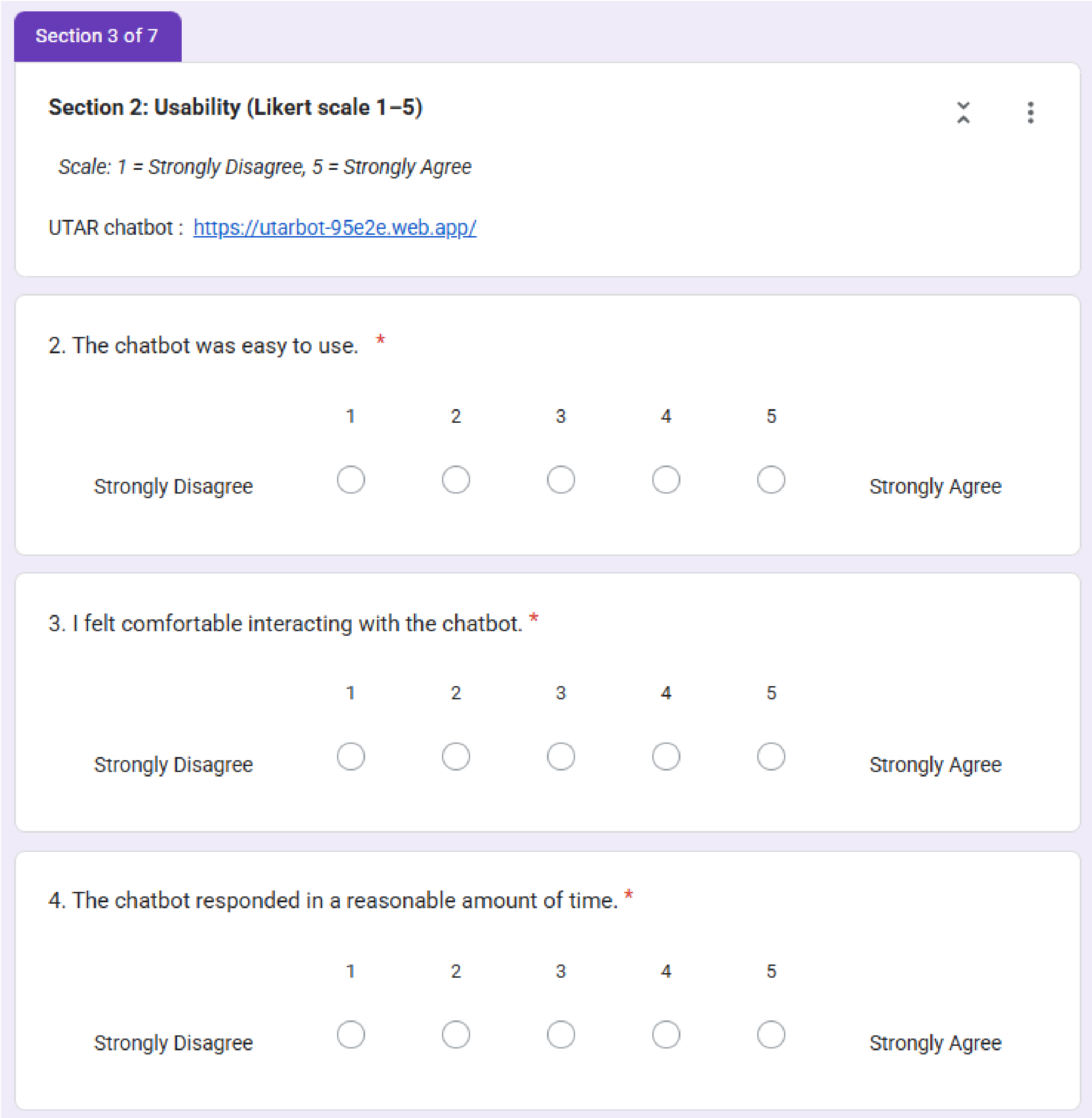
UTAR chatbot : https://utarbot-95e2e.web.app/

What is your role? *

- UTAR Staff
- UTAR Student
- Other: ____________

1. How often do you use chatbot (ChatGPT, Claude, etc.) ? *

- Never
- Rarely
- Sometimes
- Often



Section 3 of 7

Section 2: Usability (Likert scale 1–5)

Scale: 1 = Strongly Disagree, 5 = Strongly Agree

UTAR chatbot : https://utarbot-95e2e.web.app/

2. The chatbot was easy to use. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

3. I felt comfortable interacting with the chatbot. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

4. The chatbot responded in a reasonable amount of time. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Section 4 of 7

Section 3: Performance & Accuracy (Likert scale 1–5)

Scale: 1 = Strongly Disagree, 5 = Strongly Agree

UTAR chatbot : https://utarbot-95e2e.web.app/

5. The chatbot understood my questions well. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

6. The responses were relevant and accurate. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

7. The chatbot provided useful information about the university. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

8. The chatbot helped me find the information I was looking for. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Section 5 of 7

Section 4: Overall Satisfaction

UTAR chatbot : https://utarbot-95e2e.web.app/

9. Overall, I am satisfied with my experience using the chatbot. *
(Scale: 1 = Strongly Disagree, 5 = Strongly Agree)

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

10. I would use this chatbot again in the future. *

○ Yes

○ No

11. I would recommend this chatbot to other students / people. *

○ Yes

○ No

Section 6 of 7

Section 5: Open-ended Feedback

UTAR chatbot : https://utarbot-95e2e.web.app/

12. What did you like most about the chatbot? (Optional)

Long answer text

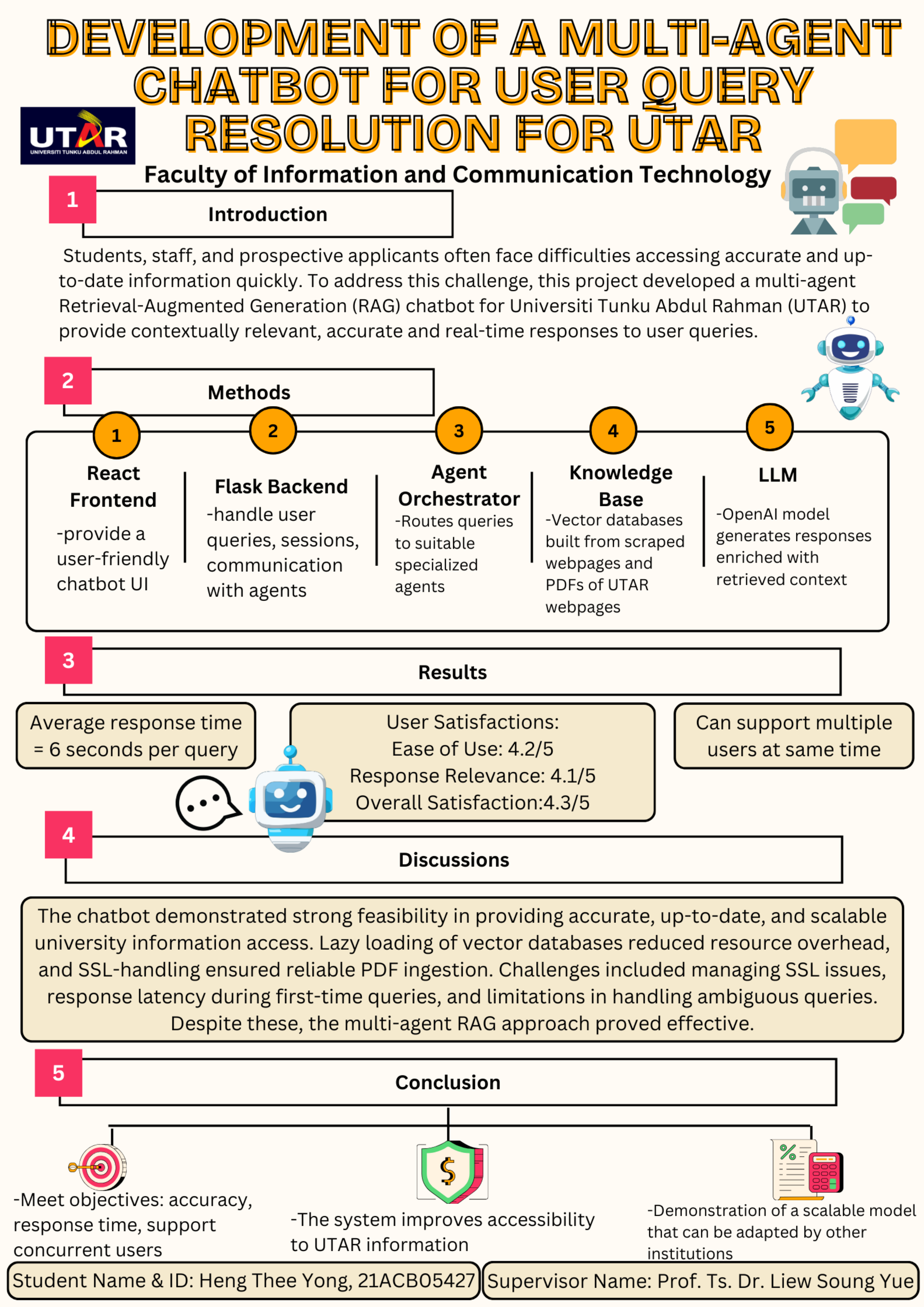
13. What did you dislike or find confusing? (Optional)

Long answer text

14. What improvements or additional features would you suggest? (Optional)

Long answer text

POSTER


DEVELOPMENT OF A MULTI-AGENT CHATBOT FOR USER QUERY RESOLUTION FOR UTAR
Faculty of Information and Communication Technology
1 Introduction
Students, staff, and prospective applicants often face difficulties accessing accurate and up-to-date information quickly. To address this challenge, this project developed a multi-agent Retrieval-Augmented Generation (RAG) chatbot for Universiti Tunku Abdul Rahman (UTAR) to provide contextually relevant, accurate and real-time responses to user queries.
2 Methods
1 React Frontend -provide a user-friendly chatbot UI
2 Flask Backend -handle user queries, sessions, communication with agents
3 Agent Orchestrator -Routes queries to suitable specialized agents
4 Knowledge Base -Vector databases built from scraped webpages and PDFs of UTAR webpages
5 LLM -OpenAI model generates responses enriched with retrieved context
3 Results
Average response time = 6 seconds per query
User Satisfactions: Ease of Use: 4.2/5 Response Relevance: 4.1/5 Overall Satisfaction:4.3/5
Can support multiple users at same time
4 Discussions
The chatbot demonstrated strong feasibility in providing accurate, up-to-date, and scalable university information access. Lazy loading of vector databases reduced resource overhead, and SSL-handling ensured reliable PDF ingestion. Challenges included managing SSL issues, response latency during first-time queries, and limitations in handling ambiguous queries. Despite these, the multi-agent RAG approach proved effective.
5 Conclusion
-Meet objectives: accuracy, response time, support concurrent users
-The system improves accessibility to UTAR information
-Demonstration of a scalable model that can be adapted by other institutions
Student Name & ID: Heng Thee Yong, 21ACB05427
Supervisor Name: Prof. Ts. Dr. Liew Soung Yue