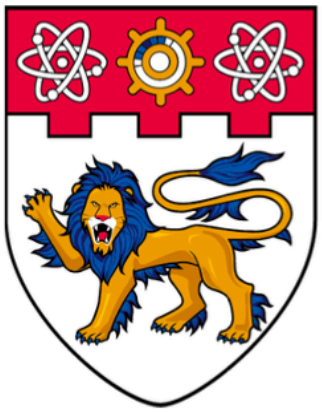


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Sustainable Multi-Model Course FAQ & Chatbot for NTU Learn V2 (CCDS24-0197)

Final Year Project Report

by

Author: Johnathan Chow Zheng Feng

Matriculation Number: U2121835D

Supervisor: Mr Ong Chin Ann

College of Computing and Data Science

2024/2025

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Professor Ong Chin Ann, for his exceptional guidance and unwavering support throughout this Final Year Project. His passion, insightful feedback, and encouragement every meeting session has made this project a fun and enjoyable experience.

I am also incredibly grateful to Professor Ong for providing me with the invaluable opportunity to conduct a workshop as part of this project. This experience not only allowed me to share my knowledge and project implementation but also helped me develop essential presentation and communication skills.

I am also grateful to my friends and participants of the online Wooclap survey and limited pilot testing whose feedback were instrumental in the implementation of the system.

1. Introduction

1.1 Background Abstract

Chatbots are an integral part of Human-Computer-Interaction as it focuses on the design, evaluation and implementation of interactive virtual systems that enhance communication between humans and computers.

The primary task of a chatbot is to produce a suitable response by contemplating natural language input provided by humans, commonly through text or speech interactions [1]. It is able to respond to user's tasks and questions automatically and its main goal is to enhance online support by using an FAQ-based system to deliver fast and accurate answers to these user queries [2]. The very first chatbot to exist was ELIZA, and it was created by Joseph Weizenbaum. ELIZA uses pattern matching and substitution methodology to simulate conversation which works by passing the words that users entered into a computer and then pairing them to a list of possible scripted responses [3].

Recent advancements in natural language processing and deep learning have significantly increased the popularity of AI-powered chatbots. Unlike traditional rule-based chatbots, these AI-powered systems leverage large language models to interpret and respond to user inputs with greater flexibility and context awareness. This enables more human-like and complex responses, making interactions smoother and more natural [4].

The adoption of such LLMs, particularly through OpenAI's ChatGPT, has surged dramatically since its introduction. Within a year of its launch, ChatGPT became one of the fastest-growing consumer internet apps ever, reaching 100 million monthly users in just two months, a pace much faster than other major platforms like Facebook, Twitter, and Instagram [5].

Chatbot technology is used by many businesses today, to initiate conversations with customers on their websites to help resolve issues, thereby improving the customer support experience. A survey was conducted for 774 online business owners and 767 customers to research current chatbot trends. The results showed that in 2022, 9 out of 10 customers interacted with a chatbot at least once and that 90% of customer inquiries are resolved in 10

messages or less [6]. This demonstrates the effectiveness of chatbots in various industries, including the education sector.

As the use of LLMs and AI continues to expand, their application in education has garnered significant attention from educators and researchers globally, owing to the wide array of possibilities they present [7]. For university students, LLMs offer significant benefits by enhancing their research and writing tasks. They can generate summaries and outlines, helping students quickly grasp key points and organise their thoughts for writing assignments. Additionally, LLMs assist in developing critical thinking and problem-solving skills by providing information, resources, and insights into unexplored aspects of topics [8].

1.2 Problem Statement

1.2.1 Overview

There are 2 related problems that need to be tackled in this FYP. Firstly, students struggle to quickly access course-specific information and materials on NTULearn, as such they resort to inefficient methods like manually searching through course documents or emailing professors. At the same time, professors and course coordinators struggle to collect timely and meaningful feedback from students for their teaching. End-of-course surveys often have low completion rates as many if not most students neglect filling up these surveys.

1.2.1 Elaboration (Student's pain points)

The current NTULearn course page allows students to access necessary course-related information and materials for registered modules. It includes a few notable pages such as announcements, information, content, discussion board, course media and discussion board.

Difficulty in finding information on NTULearn:

The current approach to accessing information and asking questions on NTULearn is inefficient. Students may struggle to locate specific information due to the volume of materials available, or because of the unstandardised organisation due to the different ways which different course coordinators arrange their course materials.

Moreover, course details such as assessment components, timetables, and venues may change at the professor's discretion. For instance, a recent initiative from CCDS offers bonus marks to students who attend tutorials weekly. This update is mentioned in the introductory notes within the course content page but is absent from the information page. As such, if students do not review the introductory notes before the lecture, they may miss out on this important information.

Delayed response/lack of immediate assistance:

When students post questions on the discussion board or email professors, professors are not always available to answer questions immediately. This might hinder students' study progress, especially if they need clarification on time-sensitive matters like assignments or exams. Professors also often receive a lot of emails from students, many of which contain repetitive or similar questions. This redundancy creates inefficiencies for both students and professors, as the same information must be communicated multiple times.

Blackboard discussion board not utilised:

Lastly, students can post their questions on the discussion board, however, they are often not actively monitored. Questions posted there may go unanswered for long periods, or students may hesitate to ask questions publicly due to fear of judgment from other peers.

1.2.2 Elaboration (Professor's pain points)

Professors in NTU require student feedback for several important reasons. Firstly, for professors, feedback is seen as an institutional requirement whereby feedback provides data about the overall student experience for internal and external quality assurance to maintain and strengthen NTU's position in the competitive education sector. For professors, feedback is required to improve the course quality, as it identifies what arrangements are effective in their course materials, teaching methods and assessments.

Timing to collect feedback:

Professors often issue end of semester surveys to gather feedback about the course from students. End of semester surveys are too late to benefit current students, and students are most busy with final exam preparations during this period. Students may have also forgotten specific issues or feedback they have from earlier in the teaching weeks.

Low participation rate of feedback:

There is not enough feedback collected from students. Professors recently resorted to granting students access to their examination results earlier if they fill up the Online Faculty Teaching Evaluation (SFT) for the semester. However, the time taken to fill up these surveys does not really provide compelling rewards for the students.

Quality of feedback collected:

Some students often complete surveys quickly without providing any thoughtful responses, either because the survey is too long or they are just after the survey incentive.

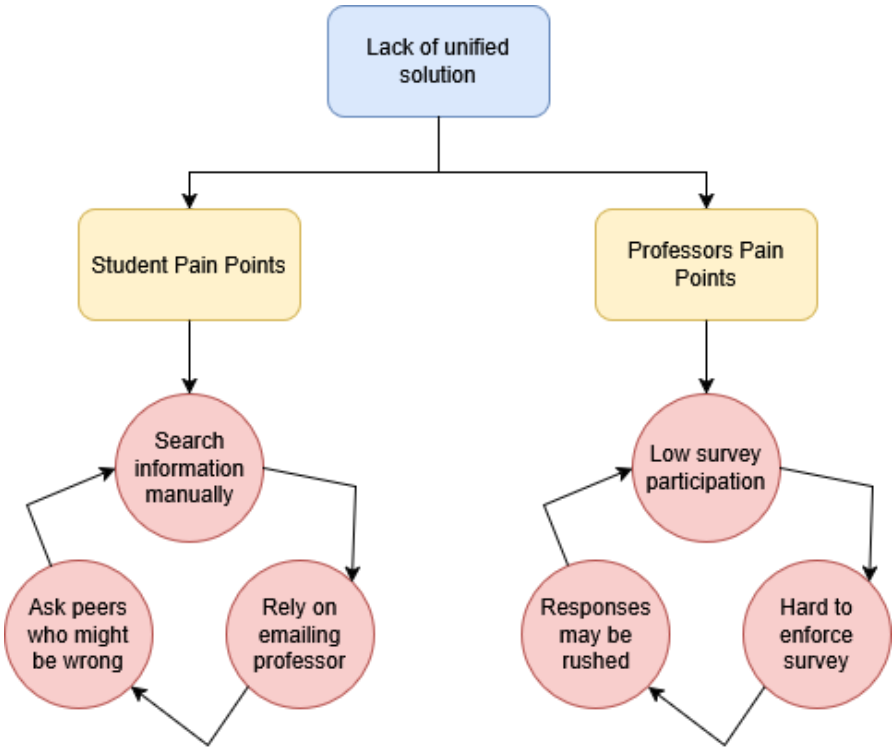


Figure 1: Summary of pain points experienced by professors and students

1.3. Purpose

The purpose of the project is to develop an LLM-based educational chatbot for students on NTULearn which helps professors collect student feedback incrementally. This then addresses the two interconnected challenges (professors' and students' pain points) elaborated in the problem statement.

For students, the chatbot will function as a centralised conversational interface by providing immediate and accurate responses to course-specific queries. This eliminates some of the pain points that students face such as frustration and time wasted in navigating through the course materials on the NTULearn platform or waiting for email replies from professors.

Firstly, the chatbot will be able to answer queries that are related to course administration such as announcements, assessment details and course FAQs. Secondly, it provides answers to course notes content by offering detailed explanations, examples, and Youtube videos if applicable to enhance students' understanding of course materials.

For professors, the chatbot will introduce a fresh approach to collect student feedback by moving beyond the traditional "filling up a survey" method. This is done by embedding the feedback collection within the students' interactions with the chatbot. This eliminates some of the pain points that professors face such as low participation rate of survey or low quality of feedback.

Firstly, survey questions are only prompted to the students when they engage with the chatbot, ensuring a natural and seamless experience. Secondly, survey questions are asked incrementally to avoid survey fatigue, with students answering 0-5 survey questions per interaction. This ensures that more authentic insights are gathered throughout the semester.

To visualise the survey responses collected, a live dashboard will also be created. Professors can make real-time adjustments to their teaching approaches if needed by looking at the dashboard, benefitting the current students immediately, rather than waiting for the next semester to implement improvements.

Ultimately, this project aims to create a continuous and sustainable cycle where improved course-related queries leads to more frequent student engagement with the chatbot, which in turn will generate incremental and more comprehensive survey feedback for professors.

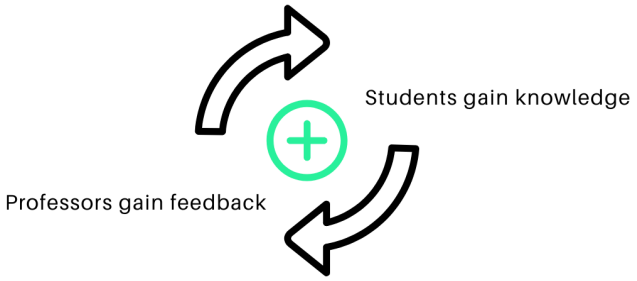


Figure 2: Synergy cycle between professor and students

1.4. Objectives

Objective 1: Design and implement a conversational chatbot system

1) Evaluate different large language models (LLMs) to identify the most suitable model for the chatbot
2) Research and select an appropriate AI framework for the project
3) Design a chatbot interface that is user-friendly for students and professors to use
4) Create a knowledge base that stores course materials and information in a way that the LLM can efficiently retrieve
5) Store conversation history logs in a database
6) Explored cloud-based solutions to deploy the chatbot for operation

Objective 2: Develop intelligent context retrieval with the use of Retrieval-Augmented Generation (RAG) techniques

1) Research on local vector stores and cloud-based vector store solutions
2) Implement semantic and keyword search functionality to retrieve and rank relevant context for the chatbot to answer questions
3) Design and integrate the RAG-based pipeline into the chatbot
4) Implement prompt engineering techniques for the LLM to utilise these context retrieved

Objective 3: Integrate an incremental survey collection mechanism in the chatbot

1) Store the survey questions in a vector database or a structured database for efficient retrieval
2) Research on optimal methods to dynamically select relevant survey questions using student's queries
3) Implement a mechanism that ensures that not more than 5 relevant survey questions are prompted per interaction
4) Create a non-intrusive design that ensures that survey questions are naturally into conversations
5) Store the survey responses into a structured database

Objective 4: Implement analytics functionality of the collected student feedback

1) Design a dashboard interface for professors to monitor survey responses
2) Enable real-time updates by refreshing survey responses every 5 minutes
3) Using charts,graphs and tables to provide data visualisation insights effectively

Objective 5: Evaluate the effectiveness of the chatbot

1) Conduct rigorous black box testing to evaluate the chatbot's ability to answer course-related and conceptual questions, including handling of edge cases
2) Collect feedback from some students for user testing to assess the chatbot's usability, intuitiveness, and effectiveness in addressing the issues elaborated in the problem statement

Objective 6: Establish a dynamic and scalable chatbot management system

1) Create a role-based permission system to differentiate functionalities of system between students and professors
2) Implement a chatbot customisation page that allows professors to upload new course materials/information and change the chatbot’s domain expertise
3) Implement a survey questions builder/uploader page that allows professors to create/upload custom feedback questions
4) Ensure that context and survey questions are updated into the chatbot automatically upon modification

Objective 7: Documentation and future work consideration

1) Ensure that all features developed are modular for easy expansion in the future
2) Write documentation for proof of concept

1.5. Scope

The chatbot will be specifically designed for CCDS students and professors who are taking or teaching the module SC1015 (Introduction to Data Science and AI). Professors will also have the flexibility to change the module to alter the chatbot context as needed. The scope encompasses several key activities, beginning with researching and selecting the most suitable LLM such as OpenAI's GPT models or open-source alternatives from Hugging Face. The selected LLM will be integrated into the chatbot by provisioning the service using Microsoft Azure and implementing it with Langchain, which will serve as the foundation for the chatbot's natural language understanding and generation capabilities. Conversational history with the chatbot will also be stored in Azure's Postgre database service using an open-sourced framework called AgentMemory from ElizaOS.

The chatbot will use Retrieval Augmented Generation (RAG) techniques such as the different search capabilities offered by Microsoft Azure AI Search to ensure relevant and context-aware responses. This approach will combine pre-defined course-specific FAQs with LLM knowledge base to provide comprehensive administrative responses to the student. Additionally, the chatbot will function as an auto-tutor, utilising internal resources such as school notes and LLM knowledge base to provide students with detailed explanations, examples, and Youtube videos to enhance their understanding of course concepts if they require them.

The survey collection feature will utilise both Azure AI search techniques and LLM's natural language understanding (NLU) capabilities. This ensures maximum relevance and accuracy of the survey questions prompted to students based on their personalised usage of the chatbot. The chatbot will use hybrid search to fetch the most relevant questions from the index based on the student's query. Additionally, it will leverage the LLM's NLU capabilities to dynamically select the most relevant survey questions based on the student's query.

For example, if a student asks a question about a specific topic (e.g., "Can you explain logistic regression?"), hybrid search will fetch the most relevant survey questions from the index. Simultaneously, the LLM will also analyse the query and choose survey questions that aligns with the topic (e.g., "How manageable do you find this course?").

The results of both methods will be cross-checked for overlapping questions, which will then be presented to the user. This two-pronged approach ensures that the survey questions are selected based on two different algorithms, and only the most relevant survey questions are prompted to the student.

The survey responses will be stored in a database using Azure Cosmos database service. An analytics visualisation interface that allows professors to interpret feedback with the help of Plotly charts and tables will be created. It will also refetch survey responses from the database every 5 minutes interval to ensure real-time reporting.

A fully functional prototype of the chatbot will be developed and deployed on Azure App Services. The frontend will be built with Streamlit, integrating all the backend features mentioned. To ensure the chatbot fulfils the project objectives, comprehensive black-box testing will be conducted. The project will also involve collecting user feedback from a small group of students to evaluate the prototype's effectiveness in addressing the problem statement.

However, the scope of this project is limited to a prototype and the chatbot will not be deployed or integrated into any existing university systems during the course of the project. The project will also not involve training or refining any LLMs but will instead rely on existing, pre-trained models. While the chatbot will include essential data visualisation for survey results, advanced analytics, reporting features or predictive modeling are beyond the scope of this project. Lastly, the project focuses on the development and testing of the prototype. Long-term maintenance, updates, or technical support for the chatbot are not included in the scope.

1.6. Project Timeline

This final year project is organised into three main stages:

1. Research & Planning (July-August 2024)
2. Features Development & Chatbot Implementation (September 2024-January 2025)
3. Testing, Evaluation & Documentation (February-March 2025)

1.6.1 Research & Planning (July-August 2024)

In this first stage, research will be conducted on LLM-based chatbots, focusing on advanced retrieval-augmented generation (RAG) techniques and features that will be useful for an educational chatbot use-case. Additional literature review will be carried out to review past FYP projects involved in creating a chatbot for NTULearn, as well as similar products available in the market.

1.6.2 Features Development & Chatbot Implementation (September 2024-January 2025)

In the second stage, chatbot development will be carried out alongside ongoing research to guide implementation. The system architecture and tech stack will be finalised, and experimentations on suitable storage solutions for vector stores and conversational history will be conducted.

Feature development will take place from October 2024 to November 2024 whereby core chatbot functionalities will first be developed to ensure a simple functional prototype. This includes implementing a conversational interface, setting up conversational history storage and also implementing vector stores for knowledge retrieval of the chatbot.

Next from November 2024 to December 2024, the survey collection module will be designed and implemented. This includes implementing survey questions storage & retrieval mechanisms. Additional research will be conducted at this stage to research on ways to optimise how feedback questions are presented to users in a non-intrusive manner. A data visualisation dashboard will also be created to enable real-time monitoring and analysis of collected survey feedback.

Further refinements to the survey collection module and context retrieval system will be made. There will be a survey question builder/uploader page to allow professors to deploy their own survey questions into the chatbot easily. Professors will also have the option to customise the chatbot's domain expertise alongside uploading their own course information and notes to be used as context for the chatbot. To support these enhancements, a role-based access control system will be implemented to manage different user permissions effectively.

1.6.3 Testing, Evaluation & Documentation (February 2025-March 2025)

From February 2025, all chatbot features components will be fully integrated, and a prototype will be deployed for a limited pilot. Some students will be invited to test the chatbot and their feedback will be collected for evaluation. Rigorous black box testing will also be conducted in this stage to evaluate chatbot performance, edge cases and usability.

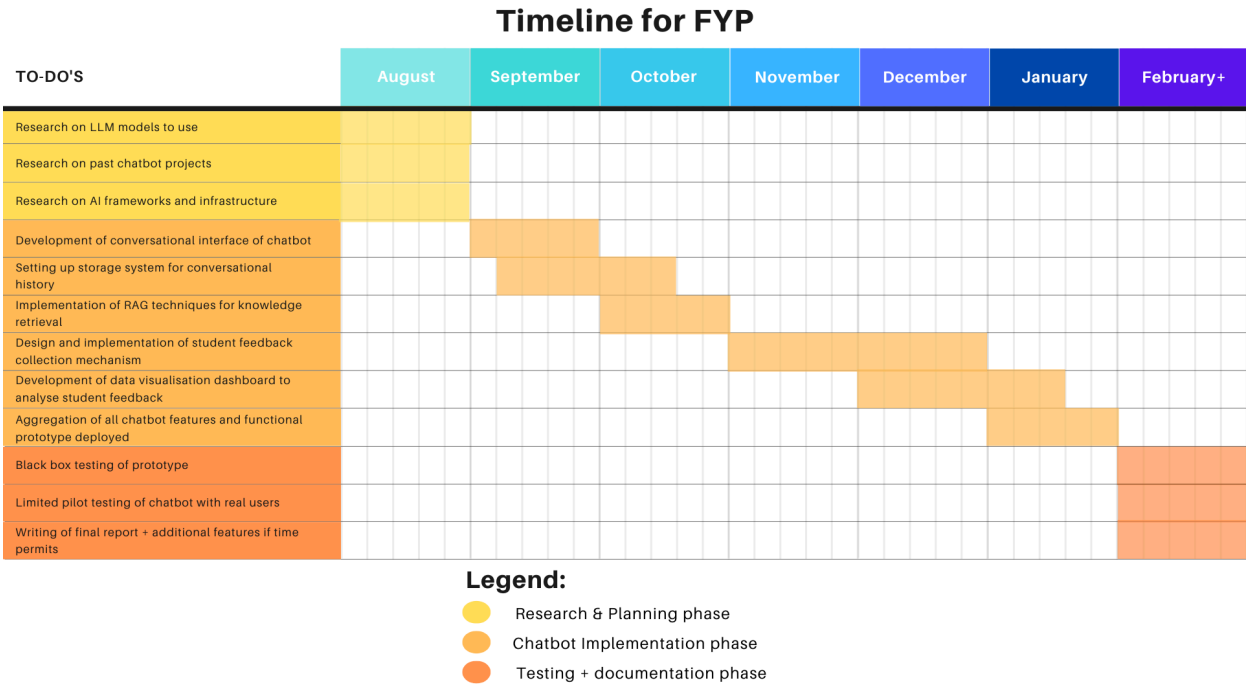


Figure 3: Project Timeline illustrated with a Gantt Chart

2. Related Works & Background Research

This section of the report reviews some previous or related works regarding the development of a NTULearn chatbot or a survey chatbot in general.

2.1 Previous Prototype of a NTULearn chatbot

There are previous studies by previous FYP students, Lew and Chee in creating a prototype NTULearn chatbot. Their chatbot prototype is designed to act as a teaching assistant for the Digital Logic course (SC1015) and the primary objective of the project was to determine the feasibility and effectiveness of using such a chatbot as an educational tool.

2.1.1 Implementation details

Lew and Chee utilised Google's Dialogflow platform to develop a rule-based chatbot prototype. The chatbot answers queries based on a set of predefined rules, which has to be manually implemented by the user. The Dialogflow framework combined intent classification, training phrases, entities and context management to create the rule-based functionality.

Intent Classification: Intents represent the user's purpose or goal behind a specific message and they are implemented using classes. These classes help the chatbot understand what the user is asking and trigger predefined responses accordingly. An example will be a user creating a `"order_pizza"`intent to match queries such as "I want to order a pizza." or "Can I order a pepperoni pizza?".

Training phrases & entities: Training phrases are example user inputs that help the framework to identify an intent class in a user's query. Entities are keywords extracted from the training phrases to further improve the classification of the intent. The purpose is to train the chatbot to understand different ways that a user's query might be phrased. An example will be, for an intent `"order_pizza"` class, training phrases will be:

"Order one cheese pizza for me", with entities `"pizza_type"=cheese,"size"=normal`
"Can I get a large beef pizza?" with entities `"pizza_type"=beef,"size"=large`

Context Management: Context management ensures that the chatbot understands the meaning of a conversation based on previous messages rather than treating each message as an isolated input, which will maintain coherence in multi turn conversations.

2.1.2 Drawbacks of rule-based framework

While this is a significant step forward in enhancing the student experience to access information without going through NTULearn, the prototype has notable limitations due to its rule-based framework. Since every user query has to be mapped to a list of possible scripted responses, the chatbot lacks the flexibility to handle complex or unexpected user queries.

Example 1: Inability to handle unexpected user queries

Lew and Chee mentioned that 52% of user queries were related to administrative matters, however there was only 1 administrative intent created to handle questions related to lab schedules [9]. Therefore, questions such as “Do I need a lockdown browser” resulted in wrong match in intent and the chatbot responded wrongly. Content questions that are short such as “Gray code” also led to no matches in intent because the chatbot required the user to have specific phrasing such as “Can you explain Gray code?”

This shows that rule-based frameworks strictly rely on predefined intents and training phrases, if a question is not explicitly accounted for and well- programmed, then the chatbot will not be able to infer meaning from that question.

Example 2: Poor Handling of Ambiguity and Short Queries

Lew and Chee mentioned queries like “Hi. May I ask why with only K bits, the maximum decimal number is expressed as 2^k-1?” were misclassified due to unstructured phrasing and entity misinterpretation (e.g., “2^k-1” confused with numbers “1” and “2”) [9].

This shows that rule-based frameworks struggle with entity ambiguity, whereby certain rigid definitions of the entity can lead to a mismatch in intent classification.

Example 3: Lack of Dynamic Learning and Adaptation

Lew and Chee mentioned that the chatbot failed to answer when students asked questions about the topics beyond week 3 as the knowledge bank they had developed only included the first 3 weeks of knowledge [9]. This meant that Lew and Chee had to manually add intents and entities to the knowledge base for every new week of school.

This shows that rule-based frameworks are static, and they cannot autonomously learn from new user inputs without any manual programming.

Example 4: Over-Reliance on FAQs and Structured Menus

Lew and Chee mentioned that a Frequently Asked Questions (FAQ) page was created as a contingency measure to queries that the chatbot failed to answer [9]. This FAQ page meant that students had to navigate the menu manually going through various menus and submenus just to find an answer for a specific query.

The FAQ page is a temporary band-aid solution for the inability of the chatbot to handle unstructured queries. Instead of the chatbot adapting to the user's request, the user is expected to adapt to the chatbot's limitation.

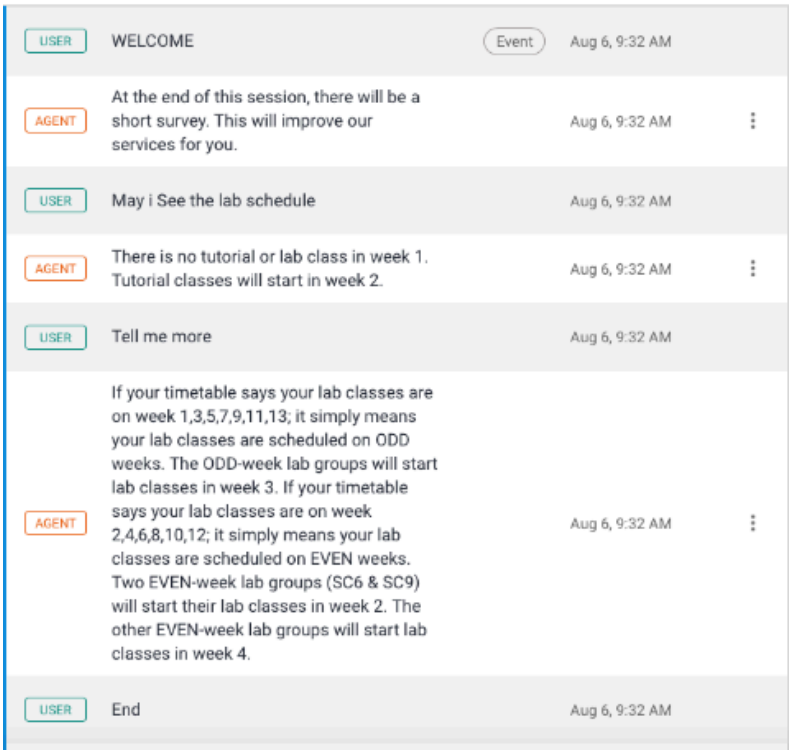


Figure 4: Example of a chat snippet from the rule-based chatbot

2.2 Current Prototype of a NTULearn chatbot

The AskNarelle chatbot is a currently deployed NTULearn chatbot created by Professor Ong Chin Ann to aid students in the modules Introduction to Data Science and AI (SC1015), Multi Disciplinary Project (SC2079) and Introduction to Computational Thinking (SC1003). It is able to answer student queries on administrative course matters using uploaded course FAQS.

2.2.1 Implementation details

Professor Ong Chin Ann utilised Microsoft Azure platform to develop a LLM-based chatbot. An LLM-based chatbot addresses the rigidity and ambiguity challenges of the rule-based chatbot specified earlier by leveraging LLM-driven natural language processing, context retrieval and also conversational memory.

AzureChatOpenAI: The chatbot uses OpenAI's GPT-3.5-turbo hosted on Microsoft Azure as the LLM model. The LLM is capable of generating responses dynamically based on the user's query using its natural language processing capabilities. This takes away the need of manually programming the chatbot to handle different types of user queries, like in traditional rule-based systems.

Knowledge Base Retrieval: `AN_Retriver`, fetches relevant documents from the knowledge base, to get relevant course materials or FAQs that can be used to answer the student's query. This solves the lack of dynamic learning and adaptation that rule-based systems face as now the LLM can dynamically alter its response depending on the documents retrieved for context.

Conversation Memory: `ConversationBufferMemory()` tracks the current messages in the conversation so that follow-up questions are correctly identified and interpreted. This solves the issue where it is hard to maintain multi-turn conversations with rule-based chatbots.

2.3 Comparison between LLM and rule-based chatbots

Feature	**Rule-based chatbots (eg. Dialogflow)**	**LLM-based chatbots (eg. AskNarelle)**
Handling Ambiguity	Relies on predefined intents and training phrases, fails on vague queries	Uses natural language processing to infer intent for ambiguous inputs
Scalability	Requires manual updating of intents, training phrases and entities	Only need to upload new documents to knowledge base
Context Management	No context considered, only rigid input-output	Tracks conversation history and allows multi-turn conversations
Response Generation	Predetermined responses from knowledge base	Generates dynamic responses depending on user's query

Table 1: Table to illustrate different chatbot frameworks

2.4 Background Research on motivations in completing surveys

2.4.1 Research motivation and collection methodology

As elaborated in the problem statement earlier, many professors face many challenges in collecting student feedback from surveys. Understanding the factors and mentality that affect survey participation is important to improve the response rates and ensure quality feedback. This includes recognising that some participants may be motivated by rewards, such as gift cards or vouchers, while others may engage out of personal interest or a sense of contribution.

This section presents responses from students who participated in a Wooclap survey on their willingness to complete surveys. This survey was conducted at the end of a workshop on Azure AI Search to gather insights for the FYP project.

2.4.2 Results from Wooclap survey

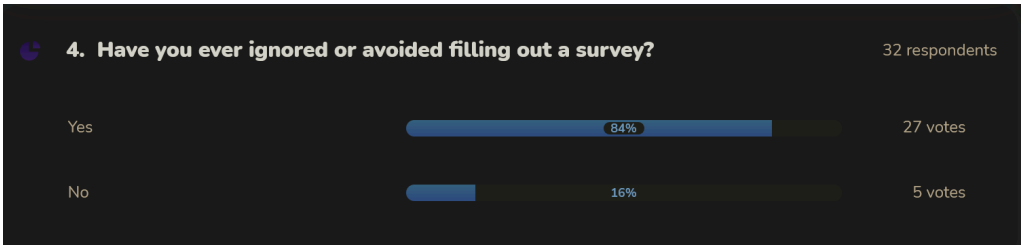


Figure 5: Wooclap survey question 1

Firstly, to understand why people are reluctant to fill up surveys, we must first explore how many have previously avoided doing so. Figure 4 shows that many participants have previously ignored skipped surveys before, and only a small group consistently completes them.

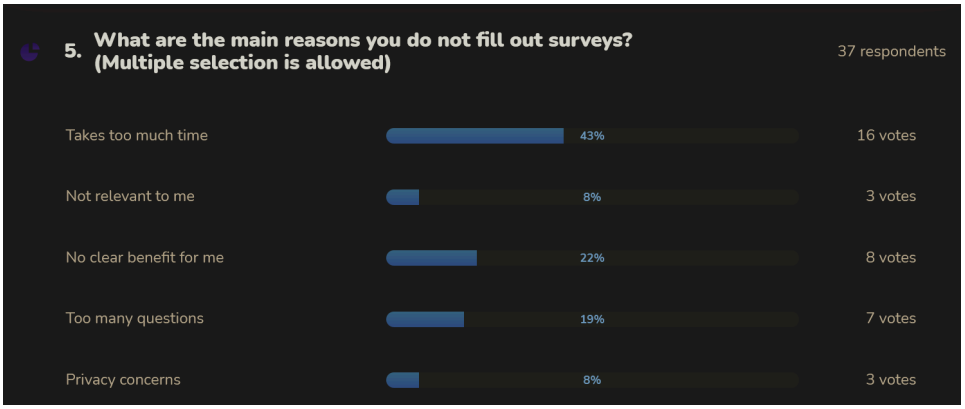


Figure 6: Wooclap survey question 2

Figure 5 shows the distribution of reasons for not completing surveys. The top results are that they take too much time, present no clear benefits and contain too many questions. Commenting is enabled for this Wooclap question and some participants gave responses like "Lazy" or "I don't benefit from filling in the survey. Even if there is a benefit, it is minimal." and "Sometimes really too many questions".

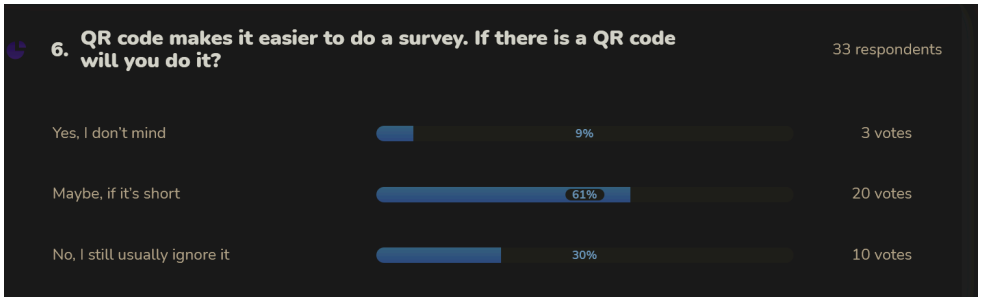


Figure 7: Wooclap survey question 3

A barrier to survey participation may be because of the inconvenience of accessing the survey. Sometimes, filling up some survey questions may require the user to type out a lengthy survey link or even sign up for an account. While the convenience of QR codes might make this process more bearable, Figure 6 actually shows many participants still choose to ignore survey questions.

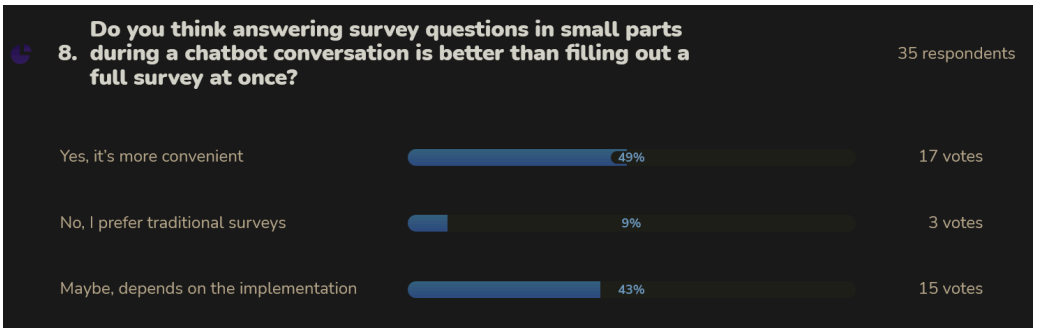


Figure 8: Wooclap survey question 4

By integrating surveys into chatbots, this FYP project aims to address the issues of traditional survey methods like filling up Google Forms. From Figure 8, it showed that more than half of participants believe that filling up surveys while using the chatbot is more convenient, which is a positive indication. However, some also expressed that the effectiveness of this method can vary based on how it is implemented.

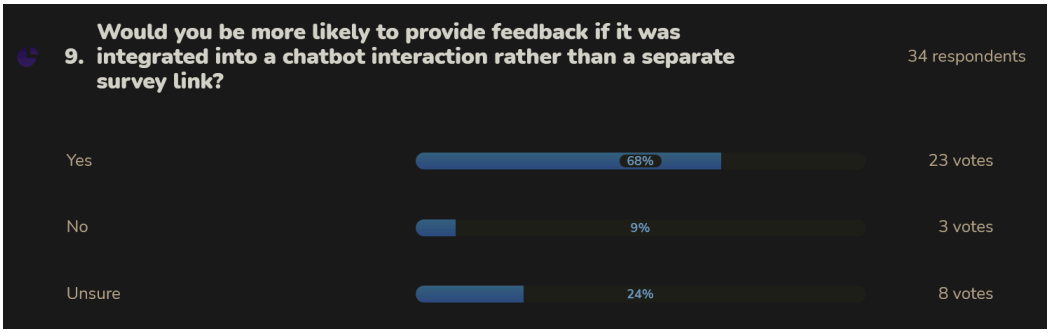


Figure 9: Wooclap survey question 5

From figure 9, the largest group of participants are worried about survey questions interrupting the chatbot experience, suggesting that users value smoothness of conversation with the chatbot. Likewise, many also experienced concerns over privacy. Therefore, it is important to take these factors into account when designing the survey module of the chatbot.

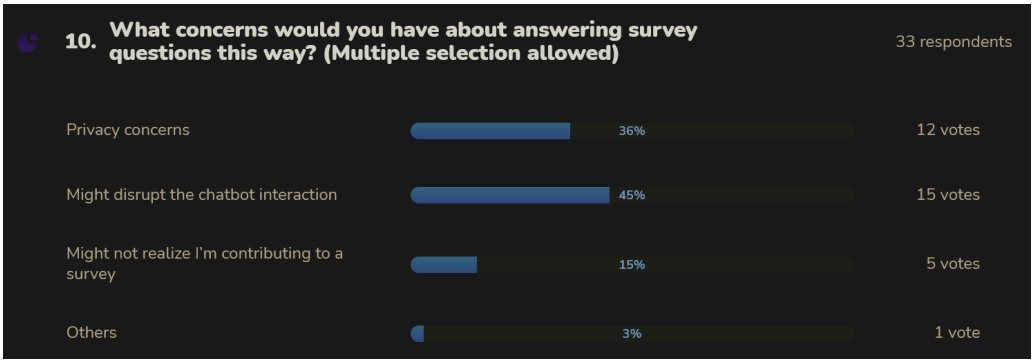


Figure 10: Wooclap survey question 6

2.5 Relevance and Summary

The related works and background research reveals important insights that directly addresses the issues that were presented in the problem statement earlier.

From the case study of the previous NTULearn chatbot by Lwe and Chee, it showed that rule-based chatbots are no longer effective. In order to create an effective chatbot that can help students quickly access course-specific information and provide conceptual knowledge, newer LLM-based solutions are required.

Therefore, there will be improvements made to the current AskNarelle chatbot. This is done by creating new features that build on its framework, to ensure that the chatbot can better serve students' needs.

The Wooclap findings provide important insights for tackling the issue of low survey completion rates among students. The data indicates that traditional survey methods fail due to time urgency, unclear benefits, and excessively long questions.

If the implementation of the survey module in the chatbot is done well, having students fill up small snippets of the survey whilst they are interacting the chatbot can prove to be more effective as compared to traditional surveys methods.

3. System Design & Architecture

This section entails the technical architecture and software stack overview used to develop the chatbot's design and functionality. It also provides the functional, non-functional requirements of the system, and a high level overview of how the chatbot works through flow diagrams.

3.1 System overview

The backend architecture is responsible for managing the functionality of all the chatbot necessary features, which includes the following:

- Initialising essential Azure services including LLM, search indexes, databases etc.
- Storing and retrieving login information in a Azure Cosmos DB for MongoDB collection
- Processing student queries using Azure OpenAI and Langchain
- Managing conversational memory with AgentMemory and Azure Database for postgreSQL
- Retrieving relevant course information/content for query context from the Azure AI Search index
- Retrieving relevant survey questions to prompt students using Azure OpenAI and Azure AI Search
- Storing and retrieving survey responses in a Azure Cosmos DB for MongoDB collection
- Allowing professors to upload custom survey questions to the Azure AI Search index
- Allowing professors to upload course information/content to the Azure AI Search index

The frontend architecture is built entirely on the Streamlit interface, which is responsible for bridging all the backend functionalities in a presentable manner, which includes the following:

- Login Page for to separate roles between students and professors
- Chatbot page for students to ask queries
- Conversational history tab to allow students to switch between previous conversations
- Survey questions prompting page to collect students' feedback
- Chatbot customisation page for professors to upload course information/content
- Survey creation page for professors to create/upload survey questions
- Survey visualisation dashboard for faculty members

Essentially, this is a "cloud-native" three-tiered architecture, because it maintains the fundamental separation of concerns of the three-tiered model while taking full advantage of Azure cloud services.

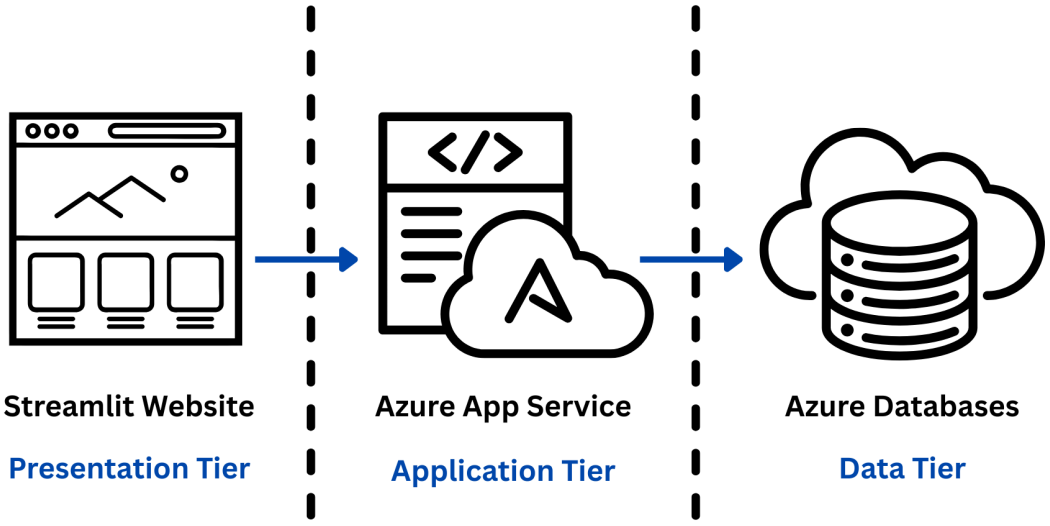


Figure 11: "Cloud-native" Three Tier System Architecture

3.1.1 Azure Ecosystem for backend

Microsoft Azure is chosen to be the main provider for the backend cloud services due to several reasons. Firstly, Azure provides $200 student credits and also free access to popular Azure services. The student credits allows for experimentation and development without incurring any personal costs, while the free access to popular services including Azure Cosmos DB and Azure AI Search will be widely used in the project. The Azure Portal is also particularly beneficial as it serves as a central hub to access all services provisioned.

Secondly, Azure OpenAI service offers usage of most GPT models through its simple provisioning service, which eliminates the need for any additional setup requirements or reliance on third-party methods.

Thirdly, Azure AI Search provides more powerful search capabilities as compared to other vector store solutions as it can use search methods like hybrid search (combining keyword and semantic search). This enhances the chatbot's ability to retrieve more relevant course information, notes and survey questions.

Lastly, Azure App service also provides a reliable cloud infrastructure to host the chatbot. This cloud infrastructure is scalable anytime, which means that the user load of the chatbot can be scaled incrementally at will to accommodate hundreds of students who may interact with the chatbot simultaneously, if it is successfully implemented into the NTULearn website.

3.1.2 Streamlit for frontend

Streamlit is chosen for the frontend due to several reasons. Firstly, Streamlit is designed for rapid prototyping as changes to the code will be reflected immediately on the interface without the need for synchronous page refreshes.

Secondly, Streamlit allows users to build web applications with just Python, removing the need for many conventional web frameworks such as HTML and CSS [10]. Since the backend architecture is also built on Python, Streamlit allows for easy integration between the frontend and the backend code.

Lastly, Streamlit has many interactive UI components and widgets, which are easily customisable to match the chatbot’s design requirements.

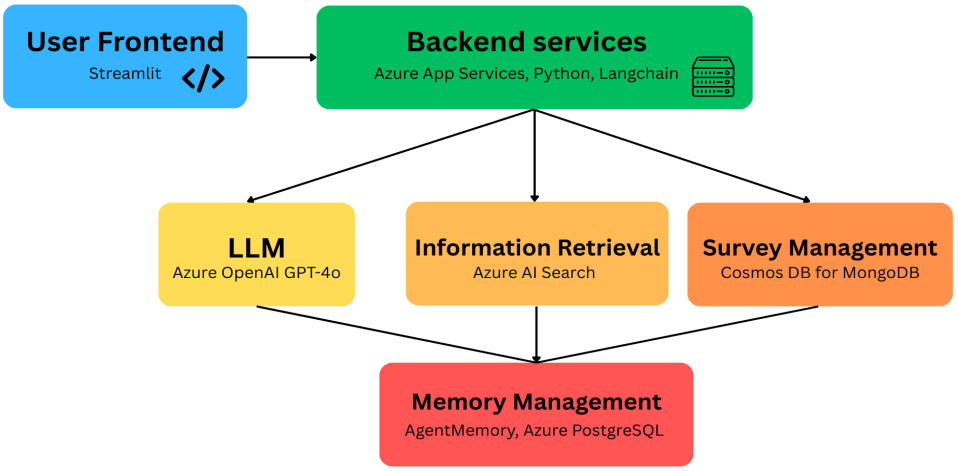


Figure 12: Illustrative Solution Architecture Diagram

3.2 Use case Diagram for chatbot

The use case diagram below illustrates the interaction of the users with the chatbot system.

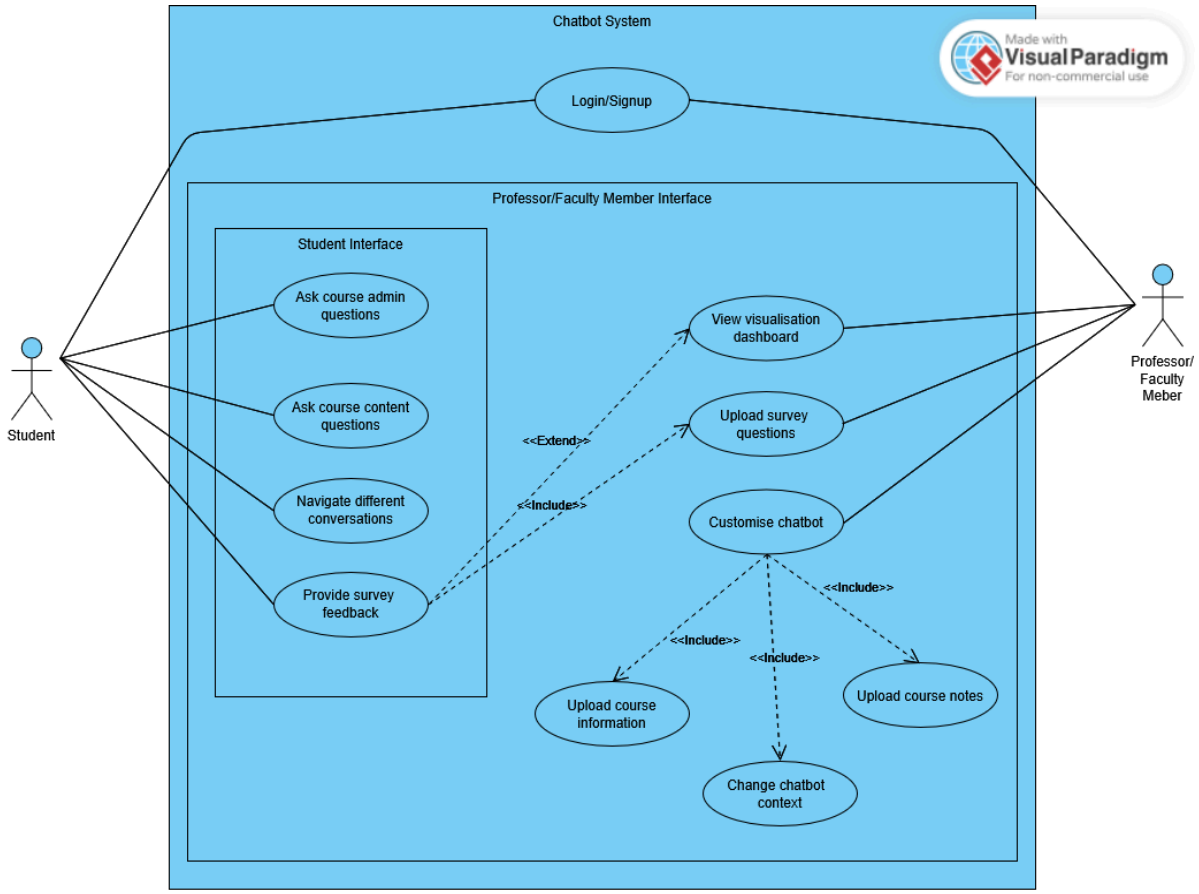


Figure 13: Use case diagram of chatbot system

3.3 Functional Requirements

Chatbot System:

1. Registration
    a. Students must be to register with their email, username and password
    b. Students information must be stored in the mongoDB database
    c. Students cannot register with an email or username that is already present in the database
2. Login
    a. Students can only login with their credentials if they already have an account
    b. Students must login only with a correct username and password
    c. Students will be denied access if username is not in mongoDB database
    d. Students will be denied access if password is invalid
    e. Professors can login to the system with a special “staff” account
3. System Initialisation
    a. System must initialise all session states required after student has logged in
    b. System must establish connection with all Azure Services provisioned after user has logged in
    c. System must establish connection with the Postgre database and fetch conversational history to a local cache after user has logged in
    d. System must only allow access to the “Survey Visualisation”, “Survey Generator” and “Chatbot Customisation” pages to professors who have the “staff” account

Student Interface:

1. Chatbot Input/Output
    a. Students must be able to send a query to the chatbot using the text input field
    b. System must be able to send the query to the Azure OpenAI LLM
    c. System must be able to display the LLM response back to the user

2. Course Admin queries

    a. The LLM must be able to identify queries related to course FAQs using the "Information" index in Azure AI Search
    b. The LLM must structure a response with the information context retrieved from the search index
    c. The LLM must reply that it does not have an answer to that question when it is not confident to provide an appropriate response

3. Course notes queries

    a. The LLM must be able to identify queries related to conceptual questions using the "Notes" index in Azure AI Search
    b. The LLM must structure a school notes explanation with the notes context retrieved from the search index
    c. The LLM must also provide a LLM-based explanation of the conceptual question asked
    d. The LLM must support these explanations with relevant resources using links
    e. The LLM must also suggest supplementary Youtube videos if applicable

4. Conversational History

    a. System must keep track of each individual student query and LLM response
    b. System must store student queries and LLM responses along with timestamp into Azure Postgre database
    c. System must use the conversational history to display a conversation tab to allow students to switch to different conversations

5. Student survey feedback

    a. System must do a search in the "Survey" index in Azure AI Search with the student's query to find relevant survey questions
    b. The LLM must also use the student's query to find relevant survey questions
    c. System must identify overlapping survey questions from the index and the LLM

d. System must ensure that questions have not been answered by the user before by querying the mongoDB database
e. System must display the overlapping survey questions if applicable after the LLM has provided a response for the student's query
f. System must ensure that the student will not be able to query the chatbot if the survey questions are not completed

Professor/Faculty Interface:

1. Data visualisation dashboard

    a. Professors with the "staff" account must be able to enter the dashboard page by clicking the "Survey Visualisation" button at the side bar
    b. System fetches and caches students' survey responses from the mongoDB database every 5 minutes interval
    c. System displays survey response data in the form of tables and charts using Plotly

2. Survey Creation

    a. Professors with the "staff" account must be able to enter the survey creation page by clicking the "Survey Generator" button at the side bar
    b. System provides two options, either uploading survey questions or generating survey questions using AI
    c. The LLM must be able to process the uploaded survey questions and do validation checks to ensure that survey questions are in the right format
    d. The LLM must be able to generate survey questions with AI
    e. System must allow the generated survey questions with AI to be edited individually
    f. System must upload the confirmed survey questions for both options to the "Survey" index in Azure AI Search

3. Chatbot customisation

    a. Professors with the “staff” account must be able to enter the chatbot customisation page by clicking the “Chatbot Customisation” button at the side bar
    b. System provides three options, to change the chatbot’s context, upload information and upload notes
    c. System must update the chatbot’s new context in the mongoDB database after the professor has changed it
    d. System must update/upload new course information uploaded by the professor to the “Information” index in Azure AI Search
    e. System must update/upload new course notes uploaded by the professor to the “Notes” index in Azure AI Search

3.4 Non-functional Requirements

1. Performance
    a. The system must respond to students' queries within 10 seconds under normal load
    b. The visualisation dashboard must fetch survey responses every 5 minutes with a maximum delay of 10 seconds
    c. Any database uploading and querying must complete within 5 seconds
2. Reliability
    a. The system must function correctly on the latest versions of Chrome, Firefox, Safari, and Edge browsers
3. Scalability
    a. The system must be scalable, by ensuring that each feature is modular
    b. The system must have proper error handling mechanisms such that the failure of one feature will still allow the user to use the system
4. Usability
    a. The interface must be user-friendly and easily navigable without training of students or professors
    b. Administrative functions such as creation of survey questions or chatbot customisation shall require no more than 10 clicks to complete tasks
    c. Each administrative page has instructions to guide users of the functionality
5. Security
    a. All students' password data is hashed before storing into database
    b. Survey response data policy must be clearly defined and enforced

3.5 Flow of student chatbot usage

The flow diagram below illustrates the complete user journey and interaction process for the students with the NTULearn chatbot, from initial query input through response generation to survey question integration and feedback collection.

The blue rectangle boxes represent the start and end of the flowchart, the yellow diamond boxes represent the types of action available, and the green rectangle boxes represent the intermediary steps.

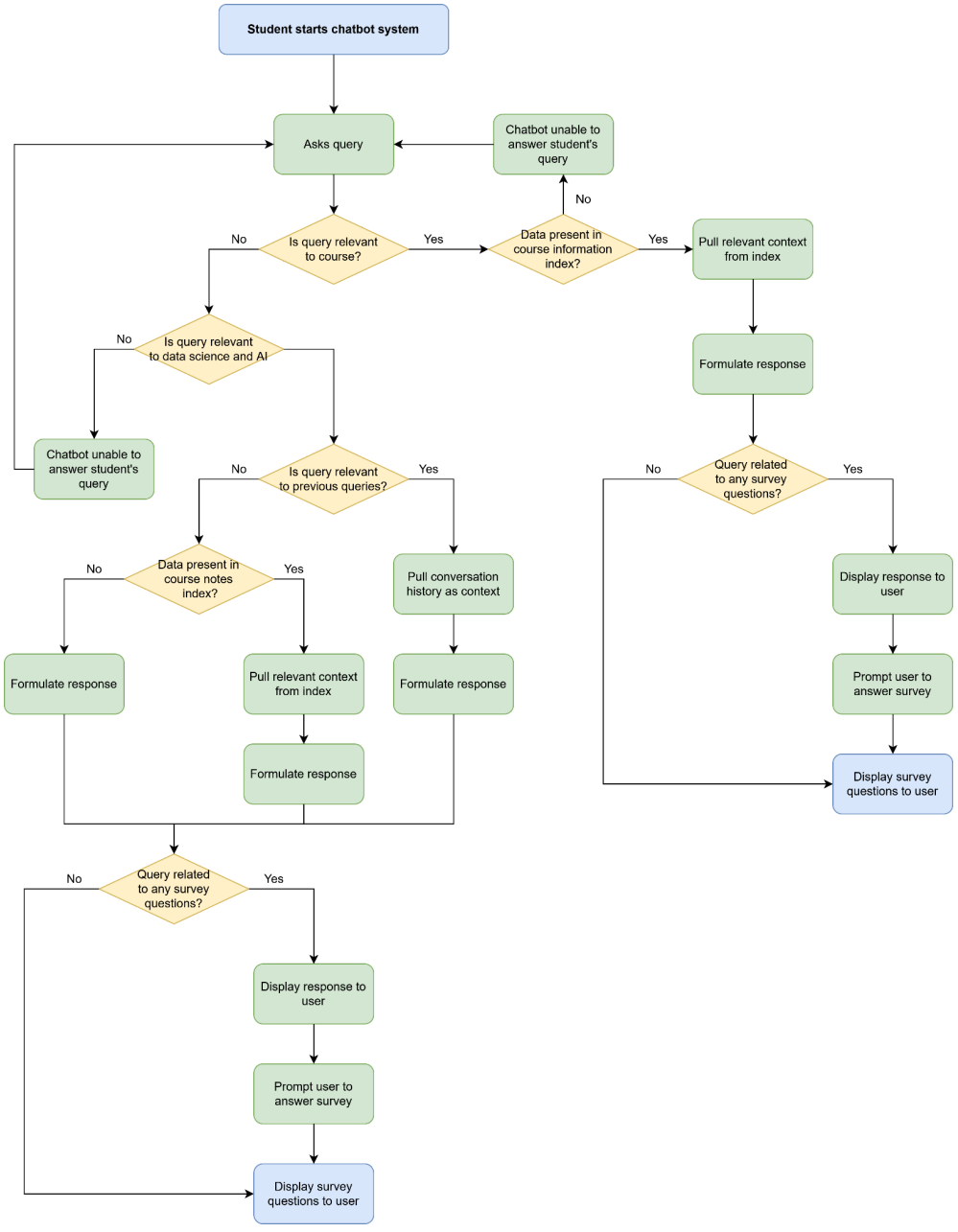


Figure 14: Flow diagram of student chatbot system

3.6 Flow of professor/faculty member chatbot usage

The flow diagram below illustrates the complete user journey and interaction process for professors/faculty members with the NTULearn chatbot, detailing the steps they can take in viewing the survey dashboard, survey creation and also chatbot customisation.

The blue rectangle boxes represent the start and end of the flowchart, the yellow diamond boxes represent the types of action available, and the green rectangle boxes represent the intermediary steps.

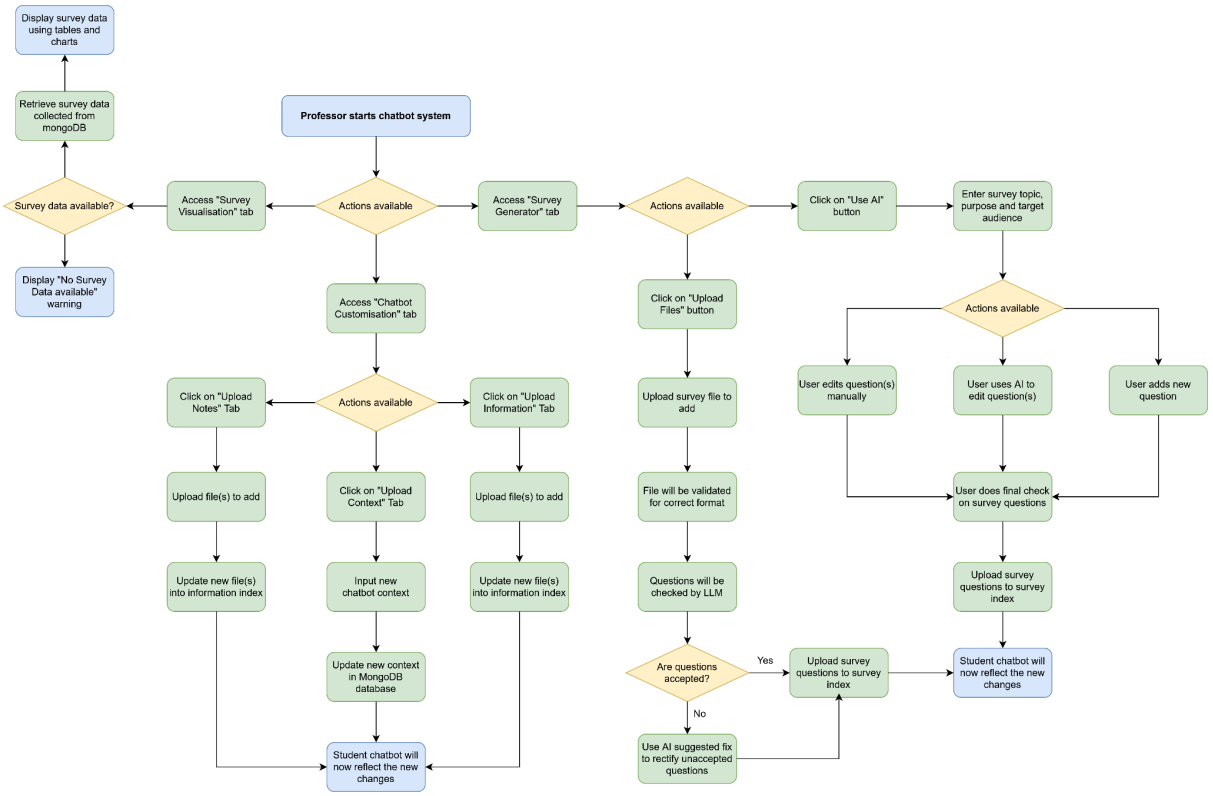


Figure 15: Flow diagram of professor administrative tools system

4. Backend Methodology

This section details the technical methodologies used to implement the chatbot's core functionalities and features. Each subsection highlights a feature of the chatbot. The tools, algorithms and mechanisms if any, and implementation process used to develop these features will be elaborated upon. **Azure AI Search and prompt engineering techniques are used heavily in most of the features of the chatbot, and therefore will be introduced first.**

4.1 Azure OpenAI

To enable students to ask the chatbot queries, Azure OpenAI is used for its NLU (natural language understanding) abilities. This allows the student to ask the chatbot complex queries in conversational language, without relying on keyword matching in traditional search systems like a rule-based chatbot.

4.1.1 LLM Model Selection Process

The first step in the project was to determine the LLM model to be used to handle student's queries in the chatbot. The LLM providers researched were (1) Azure OpenAI GPT models, (2) Google Gemini models and also (3) open-sourced Hugging Face models.

Azure OpenAI GPT models	Azure OpenAI GPT models, such as GPT-4o, are integrated into Microsoft Azure. GPT models are the most popular LLM in the market and are often used as benchmarks against other models
Google Gemini Models	Google Gemini models, such as Gemini 1.5 Pro, are integrated into Google Cloud service. They are known for their multimodal capabilities to handle images and audio. Gemini models are increasingly being implemented in smartphones as an AI service tool.
Hugging Face models	Hugging Face models, such as Falcon and Zephyr are open source models developed by the Hugging Face community. They are free to use, making them a cost effective alternative to other paid LLM services.

Table 2: Descriptions of the different LLMs considered for prototyping

	Azure OpenAI	Google Gemini	Hugging Face models
Performance	High	High	Variable (depends on model/parameters)
Integration	Requires Microsoft Azure subscription	Requires Google Cloud subscription	Manual setup required
Cost	Credit-based ($200 free for students)	Credit-based ($200 free for students)	Free
Scalability	High	High	Resource-intensive to scale to a better model

Table 3: Comparisons between the different LLMs considered for prototyping

After experimenting with the different LLMs available, **Azure OpenAI 4o model is chosen as the main provider** to process student queries using the chatbot.

Decision Rationale:

1. Range of additional services that are already included in Azure

Having a student account in Azure provides many complementary resources that will benefit the development of the prototype, such as Azure App Services, Azure Databases and Azure AI Search. Since all these services are managed by the Azure ecosystem, it ensures that all they work together seamlessly with one another, reducing potential development complexity in the future when integrating various software and frameworks together.

Hugging Face provides an inference API for some of its models, however not all models are available through this API. It is limited to lightweight models, which have smaller parameters (distilled/quantised versions) that have reduced computational costs. Using these smaller models will mean that the responses generated will not be as good. One possible way to use these HuggingFace models without the inference API is to self host these models. However, self hosting will require powerful hardware such as high-end GPUs which are not readily accessible. Cloud-based GPU rentals are also not a viable solution due to the cost required to sustain a large model over the span of the project.

4.1.2 Provisioning of Azure OpenAI service

Firstly, a dedicated resource group (eg. `chatbot_fyp`) is created on Microsoft Azure to logically isolate all the project related assets, ensuring that there is localised cost tracking for all project related deployments.

Next, the Azure OpenAI service is located and deployed within this designated resource group. This service is given a custom name (eg. `<yourname>_azureopenai`), followed by selecting the free price tier (Azure for Students). The Azure OpenAI Studio is launched, and under the service deployment, select and manage different language models. The gpt-4o model is selected and given a name (`<yourname>-azureopenai-gpt4o`), and the standard deployment type and other settings are set as default.

Another text embedding model will also be deployed under the same Azure OpenAI service. This text embedding model is to convert user queries into embeddings to search on the Azure AI Search index. The text-embedding-ada-002 model is selected and given a name (`<yourname>-azureopenai-text-embedding`), and the standard deployment type and other settings are set as default.

In order to use the models that are created, navigate to the deployment page, and under keys and endpoint, the keys are copied and securely stored into an .env file to decouple credentials from code and adhere to security best practices.

```
AZURE_OPENAI_ENDPOINT=""
AZURE_OPENAI_APIKEY=""
AZURE_OPENAI_DEPLOYMENT_NAME=""
AZURE_OPENAI_MODEL_NAME=""
AZURE_OPENAI_API_VERSION = ""

TEXT_EMBEDDING_DEPLOYMENT_NAME=""
TEXT_EMBEDDING_MODEL_NAME=""
```

Figure 16: Sample of .env file

```
llm = AzureChatOpenAI(
    azure_endpoint=os.environ['AZURE_OPENAI_ENDPOINT'],
    api_key=os.environ['AZURE_OPENAI_APIKEY'],
    deployment_name=os.environ['AZURE_OPENAI_DEPLOYMENT_NAME'],
    model_name=os.environ['AZURE_OPENAI_MODEL_NAME'],
    api_version=os.environ['AZURE_OPENAI_API_VERSION'],
    temperature=0)

llm_response = llm.invoke('Example query')
```

Figure 17: Sample of LLM calling function

```
embeddings = AzureOpenAIEmbeddings(azure_endpoint=os.environ['AZURE_OPENAI_ENDPOINT'],
api_key=os.environ['AZURE_OPENAI_APIKEY'],
azure_deployment=os.environ['TEXT_EMBEDDING_DEPLOYMENT_NAME'],
model=os.environ['TEXT_EMBEDDING_MODEL_NAME'])

embedding = embeddings.embed_query('Example query')
```

Figure 18: Sample of text embedding function

Figure 16 and 17 shows examples of how the LLM calling function and text embedding function are implemented along with its usage to invoke both of the LLM and embedding models.

4.2 Azure AI Search

To enable context-awareness for queries relating to course information and notes, Azure AI Search is used for its indexing and search capabilities. This allows the chatbot to get relevant chunks of context to formulate a response to a student's query. In the later sections, Azure AI Search is also used for storing and retrieving survey questions.

4.2.1 Vector Store Selection Process

Initially, local vector stores such as FAISS (Facebook AI Similarity Search) were evaluated for storing and retrieving course material and information. The vector database is created in the local project folder, to store the embeddings of these documents.

```
faiss_vectorstore = FAISS.from_documents(alldocs, embeddings)
faiss_vectorstore.save_local("vectorstores/faiss_vs_notes")
loaded_faiss_vs = FAISS.load_local("vectorstores/faiss_vs_notes/", embeddings=embeddings,
allow_dangerous_deserialization=True)
```

Figure 19: Local vector store creation and loading

However, there are flaws associated with FAISS. Firstly, its local storage means that the vector store has to be rebuilt whenever new course information/notes is added. For example, adding a single lecture note to the existing vector requires regenerating embeddings and rebuilding the vector store, which will lead to downtime of the vector store when new content is added.

Secondly, FAISS only supports vector similarity search and lacks native integration with keyword-based retrieval, making it unable to efficiently handle keyword-specific queries such as course codes (e.g., "SC1015").

On the other hand, Azure AI Search supports hybrid retrieval, which combines both keyword-based and vector-based search, improving the accuracy of both exact keyword matches (e.g., "SC1015") and semantic queries (e.g., "Explain linear regression"). New course information and notes can also be added incrementally to the index without requiring a rebuild and these updates are immediately reflected in the search results, as the service is cloud-based.

Therefore, **Azure AI Search is chosen as the main provider** for vector stores in this project.

4.2.2 Provisioning of Azure AI Search in chatbot

To provision the service, under the dedicated resource group (eg. `chatbot_fyp`) created earlier, Azure AI Search service is located and deployed. The service is given a name (eg. `<yourname>_azureaisearch`), followed by selecting the free price tier (Azure for Students). Go to the deployment, and under keys and endpoint, copy and securely store them into the same .env file.

```
search_index_client =
SearchIndexClient(os.environ.get('AZURE_AI_SEARCH_ENDPOINT'),
AzureKeyCredential(os.environ.get('AZURE_AI_SEARCH_API_KEY')))
```

Figure 20: Sample of search index function

4.2.3 Creating a search index in Azure AI Search

A search index defines the schema and configuration for storing and querying data. There are three primary field types, `simpleField`, `searchableField` and `searchField` that can be used to store data in index creation.

Firstly, a `SimpleField` allows for basic storage of non-searchable metadata or identifiers. This can be ideal for storing data fields like IDs, timestamps etc.

```
SimpleField(
     name="id",
     type=SearchFieldDataType.String,
     key=True,
     filterable=True,
     retrievable=True,
     stored=True,
     sortable=False,
     facetable=False
),
```

Figure 21: SimpleField Implementation

Secondly, a `SearchableField` allows for full-text search over textual content (eg. keywords, phrases). This can be ideal for storing and searching data fields which are highly specific like "SC1015 assignment 3".

```
SearchableField(
    name="content",
    type=SearchFieldDataType.String,
    searchable=True,
    filterable=False,
    retrievable=True,
    stored=True,
    sortable=False,
    facetable=False
),
```

Figure 22: SearchableField Implementation

Lastly, a `SearchField` allows for vector search capabilities by storing embeddings. It requires `vector_search_dimensions` to match the output size of the embedding model used.

```
SearchField(
    name="content_vector",
    type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
    searchable=True,
    vector_search_dimensions=embedding_dimension,
    vector_search_profile_name="my-vector-config"
),
```

Figure 23: SearchField Implementation

Summary for the three different field types in index creation:

	Use Case	Searchable	Vector Support
SimpleField	Metadata/IDs	No	No
SearchableField	Text search (eg. content)	Yes	No
SearchField	Vector search (eg. content_vector)	Yes, with vectors	Yes

Table 4: Difference between index fields

The `VectorSearch` configuration enables vector and hybrid search (keyword + vector) capabilities for searchableField and searchField.

The configuration defines:

- **Algorithm:** This defines the algorithm used for vector retrieval. In this case, the HNSW algorithm is used (`HnswAlgorithmConfiguration`) and given a name of `"myHnsw"`. HNSW uses approximate nearest neighbour search to find similar vectors. Azure AI search also supports other vector search algorithms such as Flat Algorithm, Exhaustive KNN algorithm etc. depending on the dataset requirements. In this project, since we have a relatively small index, HNSW is chosen because it provides a balance of speed and accuracy in finding similar vectors.

- **Profiles:** This links a named search profile (`"myHnswProfile"`) to a specific algorithm (`"myHnsw"`) and vectorizer (`"myVectorizer"`), creating a pipelines for vector search operations, as queries will be first converted into embeddings with the vectorizer, followed by using the search algorithm to find similar vectors in the index.

- **Vectorizer:** The `AzureOpenAIVectorizer` handles the conversion of text to embeddings internally using the **Azure OpenAI embedding model** previously defined, which eliminates the need for extra code to manually convert user queries before performing any vector search operations.

Figure 23 shows the implementation of the `Algorithm`, `Profiles` and `Vectorizer` components in a `VectorSearch` function.

```
vector_search = VectorSearch(
    algorithms=[
        HnswAlgorithmConfiguration(name="myHnsw")
    ],
    profiles=[
        VectorSearchProfile(
            name="myHnswProfile", algorithm_configuration_name="myHnsw",
            vectorizer_name="myVectorizer")
    ],
   vectorizers=[
       AzureOpenAIVectorizer(
           vectorizer_name="myVectorizer",
           parameters=AzureOpenAIVectorizerParameters(
               resource_url=os.environ.get('AZURE_OPENAI_ENDPOINT'),
               api_key=os.environ.get('AZURE_OPENAI_APIKEY'),
               model_name=os.environ.get('TEXT_EMBEDDING_MODEL_NAME),
               deployment_name=os.environ.get('TEXT_EMBEDDING_DEPLOYMENT_NAME')
           )
       )
  ]
)
```

Figure 24: VectorSearch Implementation

The documents to be stored in the index are then loaded using Langchain document loaders (`PyPDFLoader`, `Docx2txtLoader`). In figure 24, the `add_or_update_docs` function firstly uses `CharacterTextSplitter` to break the document contents into smaller chunks. It then checks whether any documents with the same file name already exist in the index, if there are existing documents, they will be deleted to avoid duplication. Lastly, using the OpenAI embedding model, embeddings are generated for each chunk, and a structure containing a unique UUID, the chunk content, the chunk embedding and the filename will be created to be inserted into the search index.

```
def add_or_update_docs(documents, index_name):
        ...
        first_doc = documents[0]
        filename = Path(first_doc.metadata['source']).name
        combined_text = "\n".join([doc.page_content for doc in documents])

        text_splitter = CharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200,
            separator="\n"
        )
        chunks = text_splitter.split_text(combined_text)
```

```
        search_results = list(search_client.search(filter=f"filename eq '{filename}'"))
        existing_ids = [result['id'] for result in search_results]

        if existing_ids:
            search_client.delete_documents([{"id": id} for id in existing_ids])

        chunk_embeddings = embeddings.embed_documents(chunks)

        docs_to_upload = [
            {
                "id": str(uuid.uuid4()),
                "content": chunk,
                "content_vector": chunk_embeddings[i],
                "filename": filename
            }
            for i, chunk in enumerate(chunks)
        ]

        if docs_to_upload:
            search_client.upload_documents(docs_to_upload)

        return True
```

Figure 25: Adding documents to search index example function

The index with the uploaded documents can then be queried to retrieve relevant chunks of data using a simple `search` method. More advanced search techniques will be explored later.

```
search_client = SearchClient(
                endpoint=os.environ.get('AZURE_AI_SEARCH_ENDPOINT'),
                index_name="example",
                credential= AzureKeyCredential(os.environ.get('AZURE_AI_SEARCH_API_KEY'))
                )

search_results = search_client.search(query,top=5)

for result in search_results:
    print("Filename:", result['filename'])
    print("Similarity Score", result['@search.score'])
    print("Content: " , result['content'], "\n\n")
```

Figure 26: Example of basic search function

Using the index creation implementation, 2 indexes will be created in the Azure AI Search service provisioned. The first index, “Information” index will be used to store course information such as course FAQs, and the second index “Notes” index will be used to store course notes such as lecture materials from Week 1 to Week 13.

4.3 Chatbot response prompt engineering

Prompt Engineering is the careful designing of prompts or instructions given to LLMs to guide them to produce either specific or more accurate responses.

By default, LLMs often try to extend user queries (e.g., turning "Explain neural networks" into a longer question) instead of answering directly to the point. Prompt engineering overcomes this by designing clear and specific instructions that “guides” the AI to generate a response.

4.3.1 Researching and adapting prompt engineering templates

To optimise the chatbot’s response to student’s queries using prompt engineering, repositories of pre-engineered prompt templates in Langchain Hub are explored. By studying popular and widely used prompt structures, certain prompting techniques were identified and adapted.

`rlm/rag-prompt` on Langchain Hub, is a prompt template that structures how a chatbot should answer questions using context retrieved from a knowledge base. From Figure 26, we can see that the prompt template follows a distinct structure, which is the instructions first, followed by the user query and supporting context fetched from a vector store.

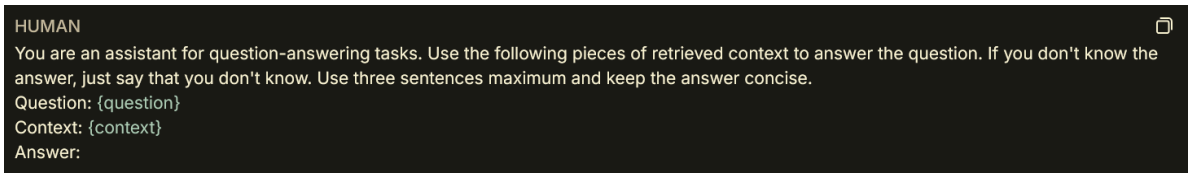


Figure 27: RAG template from Langchain Hub

`ohkgi/superb-system-instruction_prompt` on Langchain Hub, is a prompt template with a set of 10 detailed instructions to guide the LLM to create an effective prompt based off the user's query:

Figure 28: Instruction template from Langchain Hub

Lessons learnt:

Effective prompt templates have several key characteristics. Firstly, the prompt template should have a clear and specific objective. For example, if a prompt template is designed for a summarisation task, it should include a section for clear instructions on how to summarise, followed by the user query and context (such as the article to be summarised).

Secondly, instructions given to the model should be explicit and unambiguous. If certain tasks require more guidance, few-shot prompting can be further implemented giving the LLM few examples to learn the pattern of response.

Lastly, error handling and edge cases should also be accounted for when the LLM cannot find a suitable response to the query. An example of this will be "If you don't know the answer to the question, respond with…"

4.3.2 Overview of response generation flow

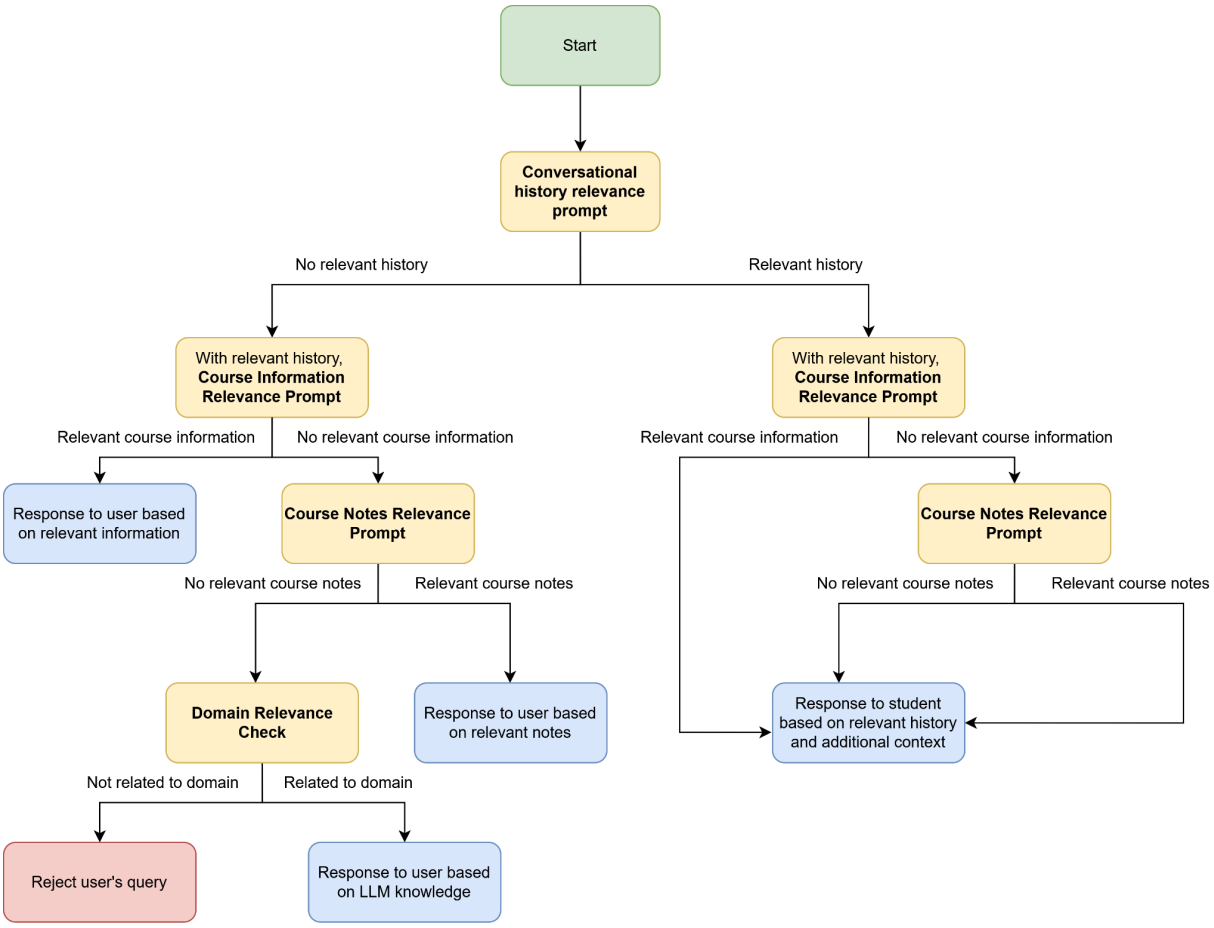


Figure 29: Flow of prompt execution in a query

High-level flowchart explanation:

The response generation flow is designed using a multi-stage decision process that evaluates several key factors before formulating a response.

1. Firstly, with the current user's query, the LLM checks whether it is related to past conversation history using the `if_related_to_past_conversations()` function.

2. Next, depending on the results:

    - For the path **with** past conversation history, the LLM attempts to retrieve relevant course information using the previous student's query from the "Information" index.
    - For the path **without** past conversation history, the LLM attempts to retrieve relevant course information using the current student's query from the "Information" index.

    This is done by using the `is_relevant_to_information()` function. If the context fetched is enough to answer the user's query then, it indicates that the student has **asked a course information query**, and the context is used to formulate a response back to the student.

3. Else, if course information is insufficient (meaning that the student did not ask a course information query):

    - For the path **with** past conversation history, the LLM attempts to retrieve relevant course notes using the previous student's query from the "Notes" index.
    - For the path **without** past conversation history, the LLM attempts to retrieve relevant course notes using the current student's query from the "Notes" index.

    This is done by using the `is_relevant_to_notes()` function. If the context fetched is enough to answer the user's query then, it means that the student has **asked a course notes query**, and the context is used to formulate a response back to the student.

4. Lastly, based on the latest decision path, one of several response generation methods is used:

    - For queries with relevant history and/or relevant course information and notes, the `continue_from_previous_conversation()` function will be used to generate a response that considers the previous messages
    - For queries without relevant history, no relevant course information and notes, a domain relevance check will be performed with `is_relevant_to_context()`. This is a final fallback check to determine whether the query is related to the domain, although there is no relevant context retrieved from both "Information" and "Notes" index.

      If it is relevant to the domain specified, then a pure LLM-based response will be generated to the user. Else, the query will be rejected.

4.3.3 Implementation of prompting techniques

In this section, all the different prompting techniques used in the response flow are explored. All these prompts work together to generate a suitable final response to the student depending on the type of question asked (eg. course administration/ conceptual queries).

Past query relevance check, `if_related_to_past_conversations()`**:**

To determine whether the past conversations are related to the current query, a few-shot prompt with examples is used. The LLM evaluates the relationship between the current query and the previous messages focusing on **linguistic**, **semantic**, and **functional** continuity. The response is requested to be in a structured JSON format for further manipulation.

Linguistic Continuity: Language patterns to indicate that the current query is building on the previous messages
Semantic Continuity: References to indicate that the current query is based on previous messages based on previous topics, terms, or concepts
Functional Continuity: Assesses whether the query serves to respond to a previous action or request

```
context_query = f"""
    Analyze if the current query continues or relates to the previous conversations.

    ### Previous Conversations [ordered chronologically from the earliest (top) to the
    latest (bottom)]:
    {past_messages}

    ### Current Query: "{query}"

    ### Analysis Criteria:
    1. LINGUISTIC CONTINUITY
       - Contains follow-up phrases: "tell me more", "explain further", "elaborate", etc.
       - Uses pronouns referring to previous content (it, that, this, those, etc.)
       - Contains elliptical expressions
         (incomplete sentences that rely on previous context)

    2. SEMANTIC CONTINUITY
       - References specific terms, concepts, or entities mentioned in previous messages
       - Asks questions about previously discussed topics
       - Requests examples or details about a previously introduced subject
       - Builds upon or challenges information provided earlier

    3. FUNCTIONAL CONTINUITY
       - Answers a question posed in the previous messages
       - Provides requested information from previous turns
       - Refers to a requested action mentioned earlier

    Return a JSON object with the following structure:
    {{
        "is_related": true/false,
        "relevant_query": "If related, include the specific query or queries that are
         relevant to the current query. If not related, leave this empty.",
        "relevant_response": "If related, include the specific parts of previous
         conversations that are relevant to the current query. Extract only the most
         essential information needed to understand the context of the current query.
         If not related, leave this empty."
    }}

    Important: Ensure your response is valid JSON that can be parsed programmatically.
    """
```

Figure 30: Conversation continuity template example

The final JSON object will return a `relevant_query` and `relevant_response` field if there is relevant context to the current query. The `relevant_query` will be used to query the indexes later on, and the `relevant_response` will be used to provide context about the past conversations for the LLM to generate an appropriate response.

Course information/course notes relevance check,

`is_relevant_to_information() or is_relevant_to_notes()`**:**

To determine whether the query is related to the course's administration or course notes, a few-shot prompt is used. The response is requested to be in a structured JSON format for further manipulation.

1. **Input**: The user provides a query ("What are the course deadlines?", "What is linear regression" etc.) and the model retrieves information relevant from the indexes
2. **Task**: The model must determine whether the retrieved context contains enough relevant information to answer the query. If so, the model will provide a response; otherwise, it will indicate that it doesn't have enough context.

```
information_relevance_prompt = f"""
    You are given the context and the query. The context contains tailored information
    from an Azure AI Search index.
    Your task is to determine whether the context contains sufficient information
    to properly answer the query.

    Query: "{query}" Context: {contexts}

    Respond in JSON format with two fields:
    1. "relevant": Boolean value (true/false) indicating whether you can answer
    the query using the context.
    2. "response": If relevant is true, provide a well-formatted, helpful response
    based on the context.

    If relevant is false, set this to "I'm sorry, I don't have enough context to
    answer that question."

    Example response for sufficient context:
    {{
        "relevant": true,
        "response": [WELL-FORMATTED RESPONSE]
    }}
    Example response for insufficient context:
    {{
        "relevant": false,
        "response": "I'm sorry, I don't have enough context to answer that question."
    }}
    Important: Ensure your response is valid JSON that can be parsed programmatically.
    """
```

Figure 31: Course information example relevance prompt template

Primary domain relevance check, `is_relevant_to_context()`**:**

Zero-shot prompt is used to ask the LLM to make a binary classification after determining whether a query is relevant to the domain (eg. Data Science and AI). This generates a response with an explicit output format requirement ("Yes" or "No").

```
primary_prompt = f"""
    Determine if this query is relevant to {domain}.

    Query: "{query}"

    Consider relevant if it relates to:
    1. Core concepts or practices in {domain}
    2. Common tools/technologies used in {domain}
    3. Standard procedures in {domain}
    4. Typical user needs in {domain}

    Respond ONLY with 'Yes' or 'No'.
    """
```

Figure 32: Initial domain relevance prompt template example

Secondary domain relevance check, `is_relevant_to_context()`**:**

A fallback check is used in case of any false negatives on whether a query is relevant to the domain. Step-based zero-shot prompting is used here, as the prompt template explicitly outlines steps the LLM should take to come to a conclusion. This fallback check improves accuracy of classification by careful relevance reconsideration and forces the LLM to double check the previous response it has given.

```
fallback_prompt = f"""
    You marked the query "{query}" as irrelevant to "{domain}" in an earlier query.
    Please do a more thorough secondary check to
    determine if the query is relevant to {domain}.

    Evaluation Steps:
    1. Identify any implicit references to {domain} concepts
    2. Check for domain-specific terminology patterns
    3. Consider common paraphrasing of {domain} topics
    4. Assess if non-domain items could relate to {domain} workflows

    Respond ONLY with 'Yes' or 'No'.
    """
```

Figure 33: Secondary domain relevance prompt template example

Response formulation of student conceptual queries,

`generate_relevant_answer_with_links()`**:**

To generate a response to explain concepts, a structured response prompt template with certain constraints is implemented. The prompt template requires the LLM to provide multiple types of explanation/examples, such as explanation from notes, explanation from the LLM model itself, and also link resources to relevant websites or videos.

```
answer_prompt = f"""
You are an experienced educator explaining concepts to students at various learning levels.

Address this query: "{query}"
Provided explanation from notes: "{ai_notes_response}"

**Response Requirements:**
1. Provide two distinct explanations:
    - First, create your OWN clear, concise explanation using simple language
    - Then integrate key points from provided notes as supplemental information

2. Structure your response as:
 ### Conceptual Explanation:
   - 1-2 paragraph plain-language overview
   - Include real-world examples/analogies where applicable
   - Highlight common misunderstandings
   - Use bullet points for key takeaways

### Supplemental Explanation From Notes:
   - [Only if notes exist] Summarize key points from reference materials
   - Never repeat your original explanation verbatim

### Verified Resources:
   - Suggest 3 authoritative sources meeting these criteria:
        * Government/educational domains (.gov/.edu) or established platforms
          (Khan Academy, Britannica)
        * Directly relevant to query subtopics
        * Active links (test URL formatting)
   - Include 1 YouTube video from official educational channels
    (CrashCourse, TED-Ed, MIT OpenCourseWare) or videos with >100k views & >90% likes

    **Format Example:**
    ### Conceptual Explanation:
    [Your original explanation here...]

    ### Supplemental Explanation From Notes:
    - [Condensed note point 1]
    - [Condensed note point 2]
```

```
    ### Verified Resources:
    1. [Resource Title] - [Full URL]
    - Domain type/credentials (e.g., "NIH medical resource")
    2. [Interactive Simulation: Topic] - [URL]
    3. [Research Paper Summary] - [URL]

    ### Recommended Video:
    - [Video Title] by [Creator] - [URL]
    - Duration & content focus (e.g., "12-min visual guide to X process")
    """
```

Figure 34: Conceptual response prompt template example

Response formulation with relevant conversation history,

`generate_relevant_answer_with_links()`**:**

Using the `relevant_response` field in the past query relevance check, the LLM is given information about the contents discussed earlier. Using the current query, and additional contexts fetched from either the "Information" or "Notes" index, a response continuation response is generated.

```
context_query = f"""
    In our previous conversation, we discussed {conversation_history}.

    The current query is: "{query}"

    Additional information (if needed): {info_reponse}

    Your task is to continue the conversation from the previous context
    using the additional information if required.

    Return a response that seamlessly connects the previous conversation
    with the current query. Feel free to elaborate as much as needed.
    """
```

Figure 35: Conversation history response prompt template example

4.4 Conversational history

From the LLM's perspective, conversational history is important as it helps the chatbot provide a coherent response by understanding the past context, which simulates a real-life conversation flow. For instance, remembering user preferences, prior questions asked, or unresolved issues makes interactions with the chatbot feel more natural and intelligent.

From the user's perspective, conversational history is important as it provides practical utility through the side navigation tab to allow users to:

1. Navigate between different conversations
2. Resume previous conversations
3. Reference previous conversations if needed
4. Organise their conversations by separating conversations for different needs

4.4.1 Conversational Memory Storage selection process

Various methods for storing conversational data in AI chatbot systems have been researched.

Firstly, session-based storage is explored. Streamlit's session state is used as a temporary storage solution for the conversational logs. However, data persistence is only limited to the active session duration. There needs to be a permanent storage solution such that all historical conversation logs will not be irreversibly lost upon session termination.

Next, traditional database systems such as MySQL (relational) and NoSQL solutions are explored. They offer permanent storage capabilities for conversation logs. These database systems offer data storage, data retrieval and also scalable solutions if needed. However, direct querying of the database can be troublesome. There are many SQL operations to remember for different use cases, such as inserting (INSERT) and updating (UPDATE). This will eventually lead to code cluttering and potential maintenance issues.

Direct database approach requiring multiple complex queries:

```
INSERT INTO conversations (id, text, metadata, embeddings) VALUES (...);
SELECT * FROM conversations WHERE text LIKE '%search_term%';
UPDATE conversations SET text = 'new_text' WHERE id = 1;
DELETE FROM conversations WHERE id = 1;
```

Figure 36: Example of direct queries from traditional database

Finally, the AgentMemory framework by open-source project ElizaOS is explored. This solution solves the issues faced by traditional database systems as the framework optimises these queries to the database.

Firstly, AgentMemory eliminates the need for raw SQL queries to retrieve conversation logs from the database. Secondly, AgentMemory offers advanced memory features, such as storing the conversation logs as embeddings if needed. Database operations are also simplified into just 1 line of code, making it very easy for programmers to understand.

Simplified AgentMemory approach:

```
create_memory("conversation", "I can't do that, Dave.", metadata={"speaker": "AI"})
search_memory("conversation", "Dave")
update_memory("conversation", 1, "New response")
delete_memory("conversation", 1)
```

Figure 37: Example of queries from AgentMemory approach

Usage guide for the AgentMemory framework is available on https://github.com/elizaOS/agentmemory/tree/main.

4.4.2 Implementation of AgentMemory in chatbot

AgentMemory uses a local ChromaDB instance by default. However, it also offers the ability to switch to a Postgres instance by adding `CLIENT_TYPE=POSTGRES` and specifying the Postgres connection string `POSTGRES_CONNECTION_STRING` in the .env file.

To provision the PostgresQL service, under the dedicated resource group (eg. `chatbot_fyp`) created earlier, Azure Database for PostgreSQL flexible server is located and deployed. The service is given a name (eg. `<yourname>_postgre`), followed by selecting the free price tier (Azure for Students). Go to the deployment, and under connection string, copy and securely store it into the same .env file.

The `create_memory(category, text, id=None, embedding=None, metadata=None)` function creates a new memory in a PostgreSQL collection. The required parameters are the category (str): Category of the collection and text (str): Document text. An example will be `create_memory("ai", "I do not have an answer to that question")`, which creates a memory to store one AI response.

Memory creation using `create_memory` is implemented using the `append_chat` function. This separates the conversational logs into two distinct categories, 'human' (user queries) and 'ai' (LLM responses). Each message also holds several important metadata, such as username, conversational ID and timestamp. Additionally, a text embedding can be generated internally if there is a need to store the responses as vectors in the Azure PostgreSQL database.

```
def append_chat(role, msg):

    current_datetime = generate_chat_timestamp()

    metadata = {
        'username': st.session_state.get('username', 'Anonymous'),
        'conversation': st.session_state.conversation_id,
        'conversation_title': st.session_state.conversation_title,
        'datetime': current_datetime
    }

    if role == 'human':
        ...
        #Permanent storage of the human message in the database
        create_memory("human", msg, metadata=metadata)

    elif role == "ai":
        ...
        #Permanent storage of the human message in the database
        create_memory("ai", msg, metadata=metadata)
```

Figure 38: Create_memory usage

Simple memory retrieval using `get_memories` loads the memories from the Azure Postgre database. This retrieves a list of memories from a given category and sorted by ID, the `sort_order` parameter controls whether the memories are fetched from the beginning or end of the list. Below is an example of the output.

```
memory = get_memories(category='ai', sort_order= "asc", n_results=1)
for mem in memory:
    print(mem)

Example Output:
{'metadata': {'username': 'user123', 'conversation': 'H9kRDbRO', 'conversation_title':
'"Identifying the Course Coordinator"', 'created_at': '1741343869.688998', 'updated_at':
'1741343869.688998', 'datetime': '2025-03-07 18:37:49'}, 'document': 'The course
coordinator is Dr. Smitha K G.', 'embedding':[...], 'id': 1}
```

Figure 39: Example output from get_memories

Memory fetching using `get_memories` is implemented using the `get_combined_memories` function. This retrieves and sorts the human and AI messages related to the specific conversation ID and current user, and then sorts the messages in chronological order to ensure the correct flow of the conversation.

Steps taken:

1) Call `get_memories` to load the conversations retrieved from the Postgre database, and use `session_state` to cache the results to reduce the overhead of fetching the memories everytime the function is called
2) Filter the conversations based on the different conversation ID and current username
3) Ensure that the messages in each conversation are sorted by timestamp
4) Iterates through the sorted messages, and then pairs each human and corresponding AI message using `zip()`, before loading them into the messages to display to the user

As mentioned in section 4.4, this way of retrieving the conversations is important as it allows the LLM to get the conversational history log for every interaction that the user had with the chatbot. This enables the LLM to generate more contextually relevant current responses by understanding past exchanges with the user, ensuring continuity in the conversation.

```
if 'cached_human_memories' not in st.session_state:
            st.session_state.cached_human_memories =
            get_memories(category='human', sort_order="desc")
if 'cached_ai_memories' not in st.session_state:
            st.session_state.cached_ai_memories =
            get_memories(category='ai', sort_order="desc")
```

Figure 40: Retrieval of conversational history using get_memories

```
def get_combined_memories(conversation_id):
    # Filter and sort messages by timestamp in ascending order (oldest first)
    human_memories = sorted(
        [human for human in st.session_state.cached_human_memories
         if human.get('metadata', {}).get('conversation') == conversation_id
         and human.get('metadata', {}).get('username') ==
         st.session_state.username], key=lambda x: x['metadata']['datetime']
    )
    ai_memories = sorted(
        [ai for ai in st.session_state.cached_ai_memories
         if ai.get('metadata', {}).get('conversation') == conversation_id
         and ai.get('metadata', {}).get('username') ==
         st.session_state.username], key=lambda x: x['metadata']['datetime']
    )
    # Pair messages and add to session state
    for human, ai in zip(human_memories, ai_memories):
    #Append to conversation history and chat messages here
    st.session_state.messages.append({"role": "human", "content": human['document']})
    st.session_state.messages.append({"role": "ai", "content": ai['document']})
```

Figure 41: get_memories implementation for usage with LLM

Conversation ID fetching using `get_memories` is also implemented using the `get_conversation_ids` function. This retrieves and organises all conversation metadata for the current user. Then, the complete list of user conversations will be displayed in the UI sidebar to allow users to navigate different conversations.

Steps taken:

1) Retrieve the cached AI responses from `session_state` earlier
2) Filters conversation data to include only those that belong to the current user and skip those that belonging to other users
3) For the conversation data that belongs to the current user, extract the conversation ID, timestamp and the title for each conversation from the metadata

4) For the conversations that have multiple messages, the earliest message timestamp is found. This ensures each conversation is categorised based on its start time and not the most recent message in the conversation.

As mentioned in section 4.4, this way of retrieving the conversations is important for the user as it allows the users to efficiently navigate and restore their past interactions with the chatbot.

```
def get_conversation_ids():
    ai_memories = st.session_state.cached_ai_memories
    # Dictionary to store the earliest timestamp for each conversation ID
    conversations = {}

    for ai in ai_memories:
        username = ai.get('metadata', {}).get('username')
        if username != st.session_state.username:
            continue  # Skip other users' conversations
        conversation_id = ai.get('metadata', {}).get('conversation')
        datetime_str = ai.get('metadata', {}).get('datetime')
        conversation_title = ai.get('metadata', {}).get('conversation_title')

        if conversation_id:
            current_datetime = datetime.strptime(datetime_str, '%Y-%m-%d %H:%M:%S')

            if conversation_id not in conversations:
                # If this is the first time seeing this conversation ID,
                # initialize it with the current timestamp
                conversations[conversation_id] = {
                    'conversation_id': conversation_id,
                    'datetime': datetime_str,
                    'conversation_title': conversation_title,
                    'username': st.session_state.username}
            else:
                #If not, Keep the earlier timestamp
                existing_datetime =
                datetime.strptime(conversations[conversation_id]['datetime'],
                '%Y-%m-%d %H:%M:%S')
                if current_datetime < existing_datetime:
                    conversations[conversation_id]['datetime'] = datetime_str
                # Update title if current entry has no title
                if not conversations[conversation_id].get('title') and conversation_title:
                    conversations[conversation_id]['title'] = conversation_title

    return list(conversations.values())
```

Figure 42: get_memories manipulation to display conversations to user

4.5 Survey questions prompting algorithm

As highlighted in the problem statement and objective, the survey selection algorithm needs to ensure that for each student query, only the most appropriate survey questions are prompted, balancing feedback collection with user experience to benefit both the students and the professors. In order to do this, we will utilise both Azure AI search techniques and also LLM NLU capabilities.

4.5.1 Creation of survey search index in Azure AI Search

The implementation of the "survey_questions" index follows the same index creation approach outlined in section 4.2.3. However, there is an inclusion of a new `SearchableField` and `SearchField` for tags. Tags are categorical labels that can be associated with each survey question. An example will be:

Survey question: "On a scale of 1-5, how will you rate the professor's teaching?"
Corresponding tags: "teaching effectiveness", "professor knowledge" and " expectations"

By assigning tags to all of the survey questions, the search index gains an additional dimension for retrieving relevant results. This enhancement allows the search technique to find relevant survey questions not just using the survey question text, but also by the contextual attributes of the tags.

```
SearchableField(
    name="tags",
    type=SearchFieldDataType.String,
    searchable=True,
    filterable=True,
    retrievable=True,
    stored=True,
    sortable=False,
    facetable=False),

SearchField(
    name="tags_vector",
    type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
    searchable=True,
    vector_search_dimensions=embedding_dimension,
    vector_search_profile_name="myHnswProfile"),
```

Figure 43: Addition of tags into index creation

4.5.2 Azure AI search's hybrid search

The simple index search described in section 4.2.3 uses a simple text-based method by looking for exact keyword matches.

However, to achieve greater precision in the survey selection algorithm, a more advanced search technique is required. Azure AI Search offers hybrid search. Hybrid search combines the text-based search with the vector search. Text-based search helps capture exact matches and specific terminology, while vector search captures semantic similarity and conceptual relationships between words.

In this implementation, vector search is performed on two fields:

1. `content_vector` field which represents the embeddings of the survey question. The vector search performed on this field looks for semantically similar survey questions based on the student's query
2. `tags_vector` field which represents embeddings of the associated tags to the question. As the tags provide contextual attributes to each survey question, the vector search performed on this field looks for survey questions that are related to the categories of the tags based on the student's query

Weights are also factored into these vector searches:

1. `content_vector` field has a weight of 1.5, which ensures that the results there is greater emphasis on the survey questions itself

2. `tags_vector` field has a weight of 0.5, which ensures that the tags contribute lesser to the ranking of the results and do not overpower the survey questions itself

Lastly, using `search_text=query` also does a full-text search based on the student's query. It looks for exact matches or partial matches to retrieve those survey questions that contain the exact words or phrases from the user's query.

As a result, the final ranking score blends all factors, leading to greater precision in selecting relevant survey questions.

Example:

If the user performs a query of "What will the professor cover in his SC1015 lectures?":

1. The `content_vector` field identifies the semantic similarity between the vectorised query to similar survey questions like "how effective is the professor in his lectures", and assigns the results with a similarity score

2. The `tags_vector` field then identifies the semantic similarity between the vectorised query to survey question tags such as "teaching effectiveness", "professor knowledge" or "lecture pace" and assigns the results with a similarity score

3. The text-based search using `search_text` then searches for similar or exact keywords in the query such as "SC1015 lectures" to the survey questions

Finally, a final ranking is calculated to show the most related survey questions, and the number of results returned will be determined by `top_k`.

```
def hybrid_search(query, top_k=5):

    vector_query_content = VectorizableTextQuery(text=query,
                                                 k_nearest_neighbors=50,
                                                 fields="content_vector",
                                                 weight =1.5)
    vector_query_tags = VectorizableTextQuery(text=query,
                                              k_nearest_neighbors=50,
                                              fields="tags_vector",
                                              weight =0.5)
    try:
        results = st.session_state.surveyquestions_search.search(
            search_text=query,
            vector_queries=[vector_query_content, vector_query_tags],
            select=["id", "content", "tags"],
            top=top_k
        )
        content= [result['content'] for result in results]
        return content

   except Exception as e:
        print(f"Error performing vector search: {e}")
        return []
```

Figure 44: Hybrid search to fetch relevant survey questions using student's query

4.5.3 LLM-based contextual understanding search

A prompt-based approach similar to section 4.3.3 is also used to select relevant survey questions based on a user query using a natural language approach.

`search_all()` first retrieves all of the available survey questions from the "survey_questions" index. Then, using a prompt template, four criteria are specified relating to the choosing of the relevant survey question:

1. **Entity matching** requires the LLM to identify key entities in the user query and consider these entities (eg. subject matter in the query)
2. **Intent recognition** requires the LLM to determine the type of question the user is asking (eg. domain relevance of query)
3. **Contextual relevance** encourages the LLM consider beyond just the important keywords in the query
4. **Question diversity** allows the LLM to ensure selected questions cover different but related aspects of the query

```
relevance_prompt = f"""
    Analyze the following user query: "{query}"

    Below is a list of available survey questions:
    {all_questions}

    Select 3-5 survey questions that are most relevant to the user's query by considering:

    1. Entity matching: Identify key entities in the query
       (e.g., people, products, services, processes, experiences)
       and prioritize questions about those specific entities.
       - Example: If the query mentions a specific person,
         prioritize questions about individual performance or characteristics.
       - Example: If the query is about a product or service,
         prioritize questions about features, quality, or satisfaction.

   2. Intent alignment: Determine what the user is trying to learn or evaluate,
      and select questions that address that intent.
      - Example: If the query suggests the user wants feedback on performance,
        prioritize evaluation questions.
      - Example: If the query suggests the user wants information about preferences,
        prioritize questions about likes/dislikes.
```

```
   3. Contextual relevance: Consider the broader context implied by the query,
      not just explicit keywords.
      - Example: A query about a "manager" suggests interest in leadership,
        communication, and decision-making.
      - Example: A query about a "checkout process" suggests interest in
        user experience and efficiency.

   4. Question diversity: Include a mix of question types to
      provide comprehensive feedback while staying relevant to the query focus.

   Example response:
   {{
   "selected_questions": [
       {{
       "question_text": "Selected survey question",
       "entity_match": "Selected entity",
       "reasoning": "Short reason for selection"
       }},
       // Additional questions...
   ]
   }}

   Important: Prioritize questions that match the most significant entities
   and intents in the query, even if they don't share exact keywords.
   Important: Ensure your response is valid JSON that can be parsed programmatically.
   """
```

Figure 45: Example of prompt to select similar survey questions

4.5.4 Intersection-based filtering of both search techniques

Using the results from the survey questions from using the hybrid search and LLM-based prompting, `find_exact_matches_intersection(query)` combines the results from the two different methods to identify survey questions that both methods agree are relevant.

1. It calls `hybrid_search(query,top_k=5)` to get the top 5 results using the hybrid search approach
2. It calls `query_search(query)` to get top 3-5 results from the LLM-based prompting approach
3. Both result are converted to Python sets and `intersection(survey_set)` is used to find the intersection based on these sets
4. Finally, the mongoDB database is queried to find out whether the user has completed these survey questions before, if not, display these questions to the user

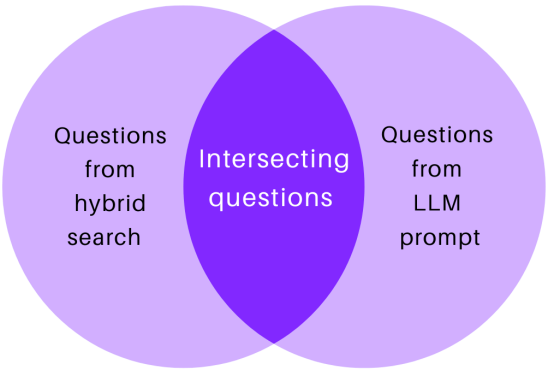


Figure 46: Illustration of survey questions set

```python
def find_exact_matches_intersection(query):
    matches = []

    search_contents = hybrid_search(query, top_k=5)
    print("Search content:", search_contents)
    survey_questions = query_search(query)
    print("LLM response:", survey_questions)

    # Create sets to match questions
    search_set = set(search_contents)
    survey_set = set(survey_questions)

    # Find the intersection of both sets
    matches = list(search_set.intersection(survey_set))

    sql_client = st.session_state.sql_client
    chatbot_db = sql_client['chatbot']
    surveys_collection = chatbot_db['surveys']

    # Find all survey questions answered by this user
    user_surveys = surveys_collection.find({"user": st.session_state.username})

    # Collect all answered questions
    answered_questions = set()
    for survey in user_surveys:
        answered_questions.update(survey.get("questions", []))

    # Remove answered questions from matches
    matches = [q for q in matches if q not in answered_questions]

    return matches
```

Figure 47: Finding intersection between survey questions with search_set

4.6 Survey questions generation/uploading methodology

The survey questions generation/uploading methodology provides the backend processes that allows professors to create/upload their own survey questions to be used to collect student feedback in the chatbot.

There are 2 processes that the professors can take:

- **Survey question generator:** Prompt engineering is used to allow professors to generate AI-generated survey questions based on parameters inputted
- **Survey question uploader:** Prompt engineering is used to validate and improve survey questions that professors have uploaded

4.6.1 Multi Agent Approach for survey questions generation

The survey questions are generated using a series of prompting techniques. Professors are required to specify the survey topic, purpose and the target audience. These will be used as parameters for the agents to come up with specialised survey questions based on the user's requirements.

Example:

If a professor wants to create survey questions to collect student feedback for the course "SC1015 Data Science and AI":

Survey Topic: Data Science and AI module, SC1015
Survey Purpose: To collect student feedback for the course to research on ways to improve teaching and course materials
Target Audience: Students taking the SC1015 module

Then, using a series of prompt engineering techniques, the LLM will generate the professor a list of possible survey questions.

The entire survey creation pipeline uses a collaborative multi-agent framework:

1. Agent 1 is a **Survey Design Agent** - The prompt analyses the parameters inputted by the professor and considers factors such as optimal question count and key dimensions to cover before providing a structured list of recommended survey questions

```
prompt_1 = f"""
    You are designing a survey with the following details:
    - **Topic:** {topic}
    - **Purpose:** {purpose}
    - **Target Audience:** {audience}

    Your task is to analyze the given context and provide recommendations for
    structuring the survey based on the metrics below:
    - **Optimal Number of Questions:** Suggest a suitable number of questions
      based on best practices.
    - **Key Dimensions:** Identify the essential themes or aspects the survey
      should cover to achieve its purpose.

    Using these recommendations, generate only Likert scale question based on
    the metrics below:

    - **Ensure Comprehensive Coverage:** Each identified dimension must have at least
      one question.
    - **Craft Clear and Neutral Questions:**
    - Use concise and unambiguous language.
    - Avoid biased, leading, or double-barreled questions.
    - Ensure appropriateness for the target audience.
    - **Maintain Proper Formatting:**
    - **Question Text**: Clearly stated.

    ## Output Format:
    Return a structured list of survey questions, aligned with the original
    survey purpose and audience.
    """
```

Figure 48: Survey questions generation prompt

2. Agent 2 is a **Tags Design Agent** - Analyses the likert scale questions generated by Agent 1 and generates relevant tags for each question. These tags ensure that they capture the subject categories of each question

```
prompt_2= f"""
    A structured list of survey questions: {response_1} has been provided.
    Your task is to analyze each question and extract relevant **tags** that
    capture its core themes.
    If there is a section on invalid questions, do not ignore it.

    ### **Your Role: Tag Generation Expert**
    You specialize in identifying key themes from survey questions.
    For each question, extract a set of concise, relevant tags that reflect the
    main ideas, ensuring that they:

    - **Capture the Essence:** Tags should summarize the key aspects of the question.
    - **Remain Concise:** Each tag should be a short, descriptive phrase (1-3 words).
    - **Be Non-Redundant:** Avoid repeating similar terms.
    - **Follow a consistent Format:** Use lowercase and separate multiple-word tags with
      spaces (e.g., "student engagement", "teaching effectiveness").
    - **Cover Multiple Perspectives:** Include aspects such as different perspectives,
      roles, or contexts where relevant.

    ## Input Format:
    A structured list of survey questions.

    ## **Example Output Format**
    **Question:** This faculty member encouraged engagement in the course;
    as a result of the teaching approaches taken by this faculty member
    I was involved and interested in the course.

    **Tags:** student engagement, teaching approach, interactivity,
    classroom dynamics, motivation, learning environment, student participation

    **Question:** This faculty member communicated clearly: this faculty
    member was easy to understand in all forms of communication including
    in classes, online, and in writing.

    **Tags:** communication skills, clarity, understanding, teaching effectiveness,
    online communication, written communication, verbal communication
    """
```

Figure 49: Tag generation prompt

3. Agent 3 is a **Quality Control Agent** - Performs final review and refinement of the questions and its associated tags if needed to ensure that it aligns with the survey parameters specified by the professor

```
prompt_3 = f"""
    You have received the following draft of survey questions along with its tags
    and the purpose of the survey:
    Questions & tags: {response_2}
    Purpose: {purpose}

    Perform a final review ensuring the survey is well-structured according to these
    guidelines:
    1. Questions should strictly use Likert scale format (1-5 agreement scale)
    2. Check for Redundancy: Identify and remove any duplicate or overlapping questions
    3. Ensure Logical Flow: Organize questions in a natural progression
    4. Verify Alignment: Confirm all questions relate to: {purpose}
    5. Refine Language: Use clear, simple phrasing appropriate for all respondents

    Provide only the finalized Likert-scale questions with their tags, formatted properly.
    """
```

Figure 50: Questions and tags checking prompt

4. Agent 4 is a **JSON converter agent** - Takes the list of questions and associated tags provided by agent 3 to convert the final survey questions into a structured JSON format for further processing

```
json_prompt = f"""
    Convert this survey data to JSON: {final_output}
    Required structure:
    {{
        "questions": [
            {{
                "question": "text",
                "tags": ["tag1", "tag2"]
            }},
        ]
    }}
    Rules:
    1. Include only the JSON output
    2. Preserve all original tags
    3. Maintain question order
    4. Ensure questions align with: {purpose}
    Important: Ensure your response is valid JSON that can be parsed programmatically.
    """
```

Figure 51: JSON converter prompt

After the survey questions are generated in JSON format, there is also an option which allows the user to regenerate specific questions that are unsatisfactory.

To handle this question regeneration, an agent will take an existing question, its associated tags and the complete survey as input to generate a completely new question that:

- Is distinct from the other existing questions
- Ensures focus on different aspects of the same survey topic (eg. if category is “Course Feedback”, then new question generated must be in this category)
- Maintains Likert scale format
- Creates new tags specific to the regenerated question
- Returns the result in JSON format

```
prompt = f"""
    A user wants to regenerate a survey question:
    - **Original Question:** {question}
    - **Tags:** {tags}
    - **All survey questions:** {survey}

    Your task is to use the contexts from the original question, tags, and all survey
    questions to **generate a completely new question** that is:
      - Distinct from the original question and existing survey questions
      - Not a rewording/rephrasing of the original
      - Focused on different aspects/angles of the topic
      - Strictly Likert scale format (not open-ended)

    - **Generate new tags** that:
        - Align with the **new question** you generate.
        - Tags should be **concise**, **contextually relevant**,
          and **cover the core themes** of the new question.
        - Use **2 to 5 tags** per question, ensuring they capture key aspects
          of the question's focus.
        - Avoid using the same tags as the original question unless they
         remain highly relevant.
        - If the question shifts focus to a related but distinct aspect,
          adjust the tags accordingly.

    Create a JSON object with:
        - 'question': Unique Likert-scale question
        - 'tags': A list of relevant tags (2-5) based on the newly generated question.
    Important: Ensure your response is valid JSON that can be parsed programmatically.
    """
```

Figure 52: Single question regeneration prompt

4.6.2 Full Process of survey questions generation

After the professor is satisfied and finalises the survey questions generated, then the survey questions will be processed into the "survey_questions" index in Azure AI Search.

The uploading processing will use the implementation of the "survey_questions" index creation outlined in section 4.5.1 earlier.

The whole process of survey questions generation using LLM is illustrated with a simple diagram:

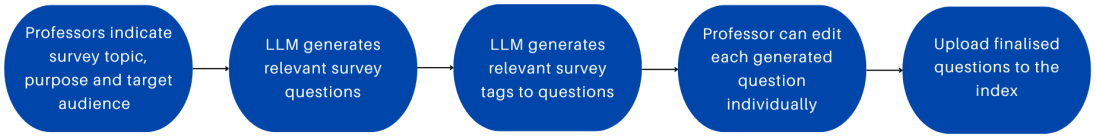


Figure 53: Survey generation full process

4.6.3 Multi Agent Approach for survey uploading

Professors are first given four options to upload their survey questions into the chatbot: PDF, DOCX, JSON or text. The system checks the type of file uploaded, extracts the text, before passing it through an LLM-based survey checker to ensure it contains valid survey questions.

```
# Read PDF file
pdf_reader = PdfReader(st.session_state["uploaded_file"])
text = "\n".join([page.extract_text() for page in pdf_reader.pages if
page.extract_text()])

# Read Word document
doc = docx.Document(st.session_state["uploaded_file"])
text = "\n".join([para.text for para in doc.paragraphs])

# Read JSON file
file_content = st.session_state["uploaded_file"].getvalue().decode("utf-8")
json_data = json.loads(file_content)
processed_json_data = json.dumps(json_data, indent=2)

# Read text file
stringio = StringIO(st.session_state["uploaded_file"].getvalue().decode("utf-8"))
string_data = stringio.read()
```

Figure 54: Process each uploaded file type

```
prompt=f"""
    Content: {text}

    **Task:** Analyze the user-provided content and check whether they are legitimate
    survey questions or not.
    If they are valid survey questions, respond with "Yes". Otherwise, respond with "No".
    Strictly follow the format "Yes" or "No".
    """
```

Figure 55: Survey questions checker prompt

If the text contains legitimate survey questions, then they are checked using a series of prompting techniques.

The entire survey questions checking pipeline uses a collaborative multi-agent framework:

1. Agent 1 is a **Likert Scale Checker** - The prompt is responsible for the first stage of questions validation, ensuring that they are only answerable with a 1-5 Likert scale. For those questions that fail the check, they will be marked as "`invalid_questions`".

```
step1_prompt = f"""
    Perform INITIAL validation of these survey questions:
    {raw_questions}
    1. For each question, determine if it can be reasonably answered
       using a 1-5 Likert scale
    2. Only flag questions that are CLEARLY incompatible with Likert scales:
    - Questions explicitly requiring binary yes/no answers
      (e.g., "Did you enjoy the workshop?")
    - Questions specifically asking to choose from multiple categories
      (e.g., "Which menu item did you prefer?")
    - Questions explicitly requesting free-text responses
      (e.g., "Please describe your experience.")
    - Questions with multiple embedded prompts requiring different response formats
    3. Statements of opinion or assessment that a respondent can agree or disagree with
       ARE valid for Likert scales, even if they could hypothetically be answered yes/no
    4. Give questions the benefit of doubt and be lenient
      - if a question could potentially work with a Likert scale, consider it valid

    Return JSON format:
    {{
        "valid_questions": [],
        "invalid_questions": [{{"question": "...", "reason": "..."}}]
    }}
    """
```

Figure 56: Likert scale checker prompt

An example of invalid questions in JSON format, with the LLM-generated reasoning on why they were rejected will be:

```
{
   "valid_questions": [...],
   "invalid_questions": [
       {
           "question": "O4: This faculty member helped students understand important
            concepts: this faculty member explained how the course content fits
            together with principles and concepts.",
           "reason": "This question is too general and would be better suited for
            a free-text response. It asks for a broad evaluation of concepts."
       },
       {
           "question": "A5: The intended learning outcomes given at the beginning of
           the course helped me understand what I was expected to learn.",
           "reason": "This question is asking for a yes/no answer
           ('Did the outcomes help you understand?'),
            which is incompatible with Likert scale."
       },
       {
           "question": "D4: This faculty member conducted valuable [type of class*]
            *Lectures, tutorials, seminars, online.",
           "reason": "This question asks for a categorical response (type of class),
            which is incompatible with a Likert scale."
       },
       {
           "question": "C5: This course challenged me to make decisions about
           different perspectives in the subject area.",
           "reason": "This question includes multiple embedded prompts asking for
           different types of response (e.g., challenge and decision-making),
           which is incompatible with a Likert scale."
       }
   ]
}
```

Figure 57: Invalid survey questions example

2. Agent 2 is a **Domain Consistency Checker** - The prompt is responsible for ensuring that all the survey questions are relevant under a single domain

```
step2_prompt = f"""
    Analyze these pre-validated questions:
    {step1_output}

    1. Verify questions are broadly related to a similar assessment domain
    2. Flag questions only if they:
    - Clearly belong to an entirely different evaluation category
    - Use drastically inconsistent perspectives that would confuse respondents
    - Contain evaluation criteria so ambiguous they cannot be meaningfully answered
    3. Be lenient - slight variations in focus or perspective are acceptable
    Maintain previous validity status where possible.

    Update JSON structure with new findings.
    """
```

Figure 58: Domain Consistency prompt

3. Agent 3 is a **Final Adjustment Agent** - The prompt is responsible for ensuring that the survey questions are clear, well-structured and properly formatted so it can be processed by the "survey_questions" index later

```
step3_prompt = f"""
    Process final validation adjustments:
    {step2_output}
    1. Do not preserve question codes (e.g., "D4:" or "Q5:" etc) if applicable
    2. For questions mentioning "scale of 1-5":
    - Keep verbatim - these are almost always valid
    3. Fix only severe grammar issues that impede understanding
    4. Err on the side of keeping questions valid - only flag questions with major
incompatibilities

    5. Ensure final output format:
    {{
        "valid_questions": [],
        "invalid_questions": [{{"question": "...", "reason": "..."}}]
    }}
    """
```

Figure 59: Final survey questions adjustment prompt

4. Agent 4 is a **Duplicate Invalid Questions Checker** - Since the LLM may hallucinate in the previous steps, this could generate duplicate questions. This prompt ensures that the final survey questions set contains no duplicate questions that appear both in the valid and invalid questions section.

```
repeat_question_prompt = f"""
    **Question Repeat Check:**
    - DO NOT include any invalid questions in the `valid_questions` list
    - Ensure that each question appears ONLY ONCE in the output
    - If a question is flagged as invalid, it MUST NOT appear in the `valid_questions` list

    **Important Notes:**
    1. If a question is invalid, it MUST ONLY appear in the `invalid_questions` list.
    2. If a question is valid, it MUST ONLY appear in the `valid_questions` list.
    3. If there are no invalid questions, leave the `invalid_questions` list empty.
    4. ALWAYS ensure NO DUPLICATES exist between the `valid_questions`
       and `invalid_questions` lists.

    Questions to check: {llm_response}

   **Output Format:**
   (Strictly follow this format, with no additional text or explanations)
        {{
            "valid_questions": [
                "Question 1 text",
                "Question 2 text",
                ...
            ],
            "invalid_questions": [
                {{"question": "Bad question text", "reason": "Explanation..."}},
                ...
            ]
        }}
        """
```

Figure 60: Duplicate questions flagging prompt

After the survey questions are checked, any invalid questions will be flagged if they violate any of the validation criteria, such as Likert scale incompatibility, difference in domain etc.

There will also be an option which allows the user to regenerate specific questions that are unsatisfactory.

To handle this question regeneration, an agent will take an existing question to generate a fix the mistakes that were flagged such that:

- Maintain the original question meaning/intent
- Maintains Likert scale format
- Returns the result in JSON format

```
fix_prompt = f"""
    Act as a survey question editor specializing in Likert scale assessment items.
    For each invalid question below, generate a corrected version that meets these
    requirements:

    1. If there is a question code, maintain the original question code (e.g., "D4:")
    at the beginning exactly as it appears
    2. Fixes the specific issue identified in the reason
    3. Format as a statement that can be rated on a 1-5 Likert scale
    (strongly disagree to strongly agree)
    4. Preserves the core meaning and assessment aspect of the original question,
    DO NOT CHANGE THE QUESTION DRASTICALLY

    Return ONLY a JSON array with corrected questions in their original order.

    Example Input:
    ["D4: I would highly recommend the person.
    (Reason: This question is a statement that would be better suited for a
    binary yes/no answer.)]

    Example Output:
    ["D4: How much would recommend the person?"]

    Invalid Questions to Fix:
    {[f"{q['question']} (Reason: {q['reason']})" for q in invalid_questions]}
    """
```

Figure 61: Invalid questions regeneration prompt

4.6.4 Full Process of survey questions uploading

After the professor is satisfied and finalises the survey questions they uploaded, then the survey questions will be processed into the "survey_questions" index in Azure AI Search.

The uploading processing will use the implementation of the "survey_questions" index creation outlined in section 4.5.1 earlier.

The whole process of survey questions uploading using LLM is illustrated with a simple diagram:

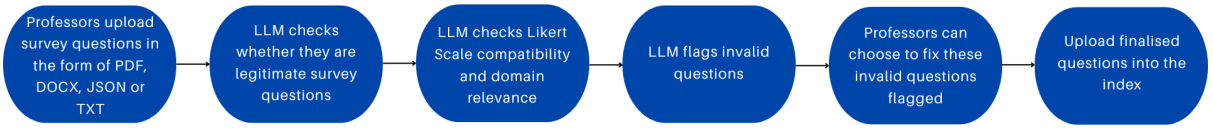


Figure 62: Survey uploading full process

4.7 Chatbot customisation using Azure AI Search index

Chatbot customisation is achieved through an extension of storing and fetching the chatbot context in mongoDB and also updating of the indexes created in Azure AI Search in Section 4.2.3.

4.7.1 Updating chatbot context

Updating chatbot context allows the professor to specify a domain constraint for the chatbot. If the professor is teaching the module SC1015, Introduction to Data Science and AI, then the chatbot context will be “Data Science and AI”. This restricts the chatbot from answering any question outside of this domain.

To fetch the current chatbot context, the `chatbot_context` collection in mongoDB is queried. If the context field is found, it is assigned to `st.session_state.chatbot_context`, which is then used in the “domain relevance" prompt specified earlier in section 4.3.3.

```
if 'chatbot_context' not in st.session_state:
            st.session_state.chatbot_context = None
            chatbot_db = st.session_state.sql_client['chatbot']
            context_collection = chatbot_db['chatbot_context']
            st.session_state.chatbot_context = (existing_context :=
            context_collection.find_one({}, {"_id": 0, "context": 1})) and
            existing_context.get("context", "")
```

Figure 63: Fetching the current context of the chatbot

```
def is_relevant_to_context(query):
    # Fetch chatbot context from database
    domain = st.session_state.chatbot_context
    # Step 1: Initial Relevance Check
    primary_prompt = f"""
    Determine if this query is relevant to {domain}.
...
...
```

Figure 64: Snippet of a prompt function that uses the dynamic chatbot context

The context field of the `chatbot_context` collection in mongoDB can also be edited dynamically. This is handled by either updating the context using `update_one()` if a context already exists, or inserting the new context into the collection using `insert_one()` if it does not.

```python
if st.button("Save Context"):
            new_context = context_phrase.strip()
            if new_context and new_context != st.session_state.chatbot_context:
                if st.session_state.chatbot_context:
                    context_collection.update_one({}, {"$set": {"context": new_context}})
                else:
                    context_collection.insert_one({"context": new_context})

                st.session_state.chatbot_context = new_context
                st.session_state.context_success = True
                st.rerun()
```

Figure 65: Updating the new context of the chatbot

4.7.2 Updating information and notes index in Azure AI Search

Building upon the Azure AI Search index creation in Section 4.2.3, this section extends the index creation capabilities to process and upload user-provided documents through a front-end interface. While the index creation implementation process remains consistent, several enhancements have been made to accommodate professor's needs to upload both course information and course notes to the chatbot.

Therefore, a two indexes approach is found to be most suitable to separate the course content into 2 categories.

- **Information Index:** Designed for storage of administrative or informational content such as course FAQs etc.
- **Notes Index**: Designed for storage of supplementary materials such as school notes (Week 1-13) etc.

This separation of indexes uses the same field types detailed in Section 4.2.2 (`SimpleField`, `SearchableField`, and `SearchField`) but applies them to different content domains. This helps to improve retrieval accuracy depending on whether the user asks the chatbot an administrative or conceptual question.

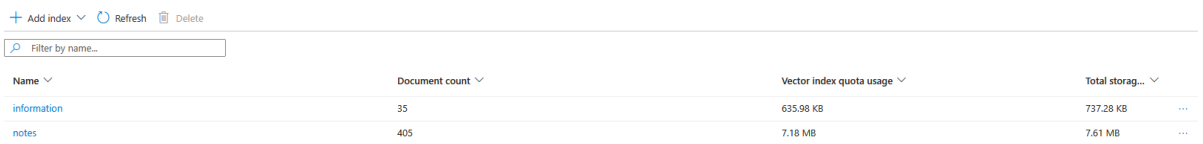


Figure 66: Separation of indexes in Azure AI Search

4.8 Visualisation of survey responses

After collecting responses of the survey questions from students who are using the chatbot, professors need to understand these data. Rather than manually analysing individual survey responses, a dashboard will help professors visualise patterns in survey responses to identify trends for areas of improvement in teaching and course materials. This section focuses on the backend implementation of processing the survey responses for visualisation. The frontend visualisation dashboard which includes Plotly graphs and tables, will be covered later in the later section.

4.8.1 Survey responses data processing

Survey data is first retrieved and cached with a 10 minute time-to-live, this means that fresh survey responses will be fetched from the mongoDB database every 10 minutes, to ensure that the dashboard is always updated.

```
@st.cache_data(ttl=600, show_spinner="Loading survey data...")
def get_survey_data():
    """Cache survey data with 5 minute freshness"""
    client = get_db_connection()
    return list(client['chatbot']['surveys'].find({}))
```

Figure 67: Caching of survey responses

Next, each survey response is extracted to create a distribution of responses for each question.

```
all_questions = list(set([q for doc in survey_data if 'questions'
in doc for q in doc['questions']]))
sorted_questions = sorted(all_questions)
dist_data = []
for question in sorted_questions:
    answers = []
    for doc in survey_data:
        if 'questions' in doc and question in doc['questions']:
            idx = doc['questions'].index(question)
            answers.append(doc['answers'][idx])

    answer_counts = pd.Series(answers).value_counts().reindex(range(1, 6), fill_value=0)
    dist_data.append({
        'Question': question,
        **{f'Rating {i}': answer_counts[i] for i in range(1, 6)}
    })
df_dist = pd.DataFrame(dist_data)
```

Figure 68: Storing survey responses into a Dataframe

4.8.2 Visualisation components

Using the structured Dataframe showing the response distribution for each question, the data is displayed using several components.

Summary Statistics:

The "Total Responses" metric calculates the total number of survey responses submitted by all students. The "Unique Question" metric calculates the number of distinct survey questions answered by all students.

```
st.metric("Total Responses", sum(len(doc.get('answers', [])) for doc in survey_data))
st.metric("Unique Questions", len(sorted_questions) -1)
```

Figure 69: Summary statistics counting of survey responses

Survey question tracker:

A matrix visualisation table component is created which shows which questions each individual student has answered. If the student has responded to a particular question, then it will be marked as "Not Answered". This table helps to reveal patterns in how students interact with the chatbot. As outlined in section 4.5, survey questions are prompted based on the student's query. This means that survey questions are very personalised since each student uses the chatbot differently which allows only the most relevant survey questions to be prompted to them.

```
data = []
    for question in all_questions_list:
        row = {'Question': question}
        for user in user_responses:
            if question in user_responses[user]:
                row[user] = '✔️'
            else:
                row[user] = '❌'
        data.append(row)
```

Figure 70: Question processing to track individual survey responses

```
df_raw = pd.DataFrame(data)
styled_df =
df_raw.style.applymap(lambda x: 'color: #d62728' if x == '❌' else 'color: #2ca02c')

st.dataframe(
    styled_df,
    use_container_width=True,
    column_config={
        "Question": st.column_config.TextColumn(
            "Survey Question",
            width="medium"
        ),
        **{user: st.column_config.Column(
            user,
            help="Shows ✔️ if answered or ❌ if not",
            width="small"
        ) for user in user_responses}
    }
)
```

Figure 71: Matrix table implementation

Survey responses timeline:

A survey response timeline component is created which is an interactive scatter plot that displays when each student submits their survey responses over time. Through this module, professors can identify when the most survey responses are recorded. This will reveal usage patterns of the chatbot, such as just before deadlines/finals, or during specific times of day/week.

```
timeline_data = []
for doc in survey_data:
    if 'timestamp' in doc:
        try:
            ts = pd.to_datetime(doc['timestamp'])
            timeline_data.append({
                'User': doc['user'],
                'Timestamp': ts
            })
        except (TypeError, ValueError):
            pass
```

Figure 72: Timestamp processing of each submitted survey response

```
df_timeline = pd.DataFrame(timeline_data)

    fig = px.scatter(
        df_timeline,
        x='Timestamp',
        y='User',
        color='User',
        labels={'Timestamp': 'Submission Time', 'User': 'Respondent'},
        category_orders={"User": sorted(df_timeline['User'].unique())},
        color_discrete_sequence=px.colors.qualitative.Plotly
    )

    fig.update_traces(
        marker=dict(size=10, line=dict(width=2, color='DarkSlateGrey')),
        hovertemplate="<b>%{y}</b><br>%{x|%Y-%m-%d %H:%M:%S}<extra></extra>"
    )

    fig.update_layout(
        xaxis=dict(
            rangeselector=dict(
                buttons=list([
                    dict(count=1, label="1d", step="day", stepmode="backward"),
                    dict(count=7, label="1w", step="day", stepmode="backward"),
                    dict(count=1, label="1m", step="month", stepmode="backward"),
                    dict(step="all")
                ])
            ),
            rangeslider=dict(visible=True),
            type="date"
        ),
        height=600,
        showlegend=False
    )

    st.plotly_chart(fig, use_container_width=True)
```

Figure 73: Survey responses timeline implementation

Survey answers distribution visualisation:

A survey answers distribution component creates a stacked horizontal bar chart that displays how survey responses are distributed across different Likert scale levels (1-5) for each survey question. Professors can use this visualisation to quickly scan which questions received mostly positive (higher ratings) versus negative responses.

```
fig = go.Figure()
for i, rating in enumerate(range(5, 0, -1)):
    fig.add_trace(go.Bar(
        y=df_dist['Short Question'],
        x=df_dist[f'Rating {rating}'],
        name=f'Rating {rating}',
        orientation='h',
        marker_color=colors[i],
        text=df_dist[f'Rating {rating}'],
        textposition='inside',
        texttemplate='%{text}'
    ))
```

Figure 74: Survey answers processing for horizontal bar chart implementation

Detailed individual question analysis:

This component creates expandable sections for every survey question to view more in-depth visualisations of the response data, using bar charts and pie charts. This allows professors to examine individual questions in detail if they require without feeling overwhelmed by the survey answers distribution visualisation mentioned earlier.

```
for question in sorted_questions:

        with st.expander(f"{question}", expanded=False):
            answers = []

            for doc in survey_data:
                if 'questions' in doc and question in doc['questions']:
                    idx = doc['questions'].index(question)
                    answers.append(doc['answers'][idx])

            if not answers:
                st.write("No responses for this question")
                continue

            tab1, tab2 = st.tabs(["Distribution", "Breakdown"])
```

```
        with tab1:
            # Bar chart
            fig1 = px.bar(
                x=[f'Rating {i}' for i in range(1, 6)],
                y=[(pd.Series(answers) == i).sum() for i in range(1, 6)],
                color=[f'Rating {i}' for i in range(1, 6)],
                color_discrete_sequence=px.colors.sequential.Reds[::-1],
                labels={'x': 'Rating', 'y': 'Count'},
                title="Response Distribution"
            )
            st.plotly_chart(fig1, use_container_width=True, key=f"dist_{question}")


        with tab2:
            # Pie chart
            fig2 = px.pie(
                names=[f'Rating {i}' for i in pd.Series(answers).value_counts().index],
                values=pd.Series(answers).value_counts().values,
                hole=0.3,
                title="Response Breakdown",
                color=[f'Rating {i}' for i in pd.Series(answers).value_counts().index],
                color_discrete_sequence=px.colors.sequential.Reds[::-1]
            )
            fig2.update_traces(textposition='inside', textinfo='percent+label')
            st.plotly_chart(fig2, use_container_width=True, key=f"pie_{question}")
```

Figure 75: Individual question analysis implementation

5. Frontend Implementation

Streamlit seamlessly integrates all of the backend methodology that was elaborated in Section 4, providing a unified interface where students can interact with the chatbot and submit feedback on survey questions. Meanwhile, professors/faculty members with "staff" access can upload their own custom chatbot context and create dynamic survey questions.

5.1 Login system

Landing page:

Users will see this page when they first visit the website. It provides options to log in for existing users or sign up for new users. Currently, the "Continue As Guest" option can be toggled on and off, depending if the professor/faculty member allows students to proceed without an account.

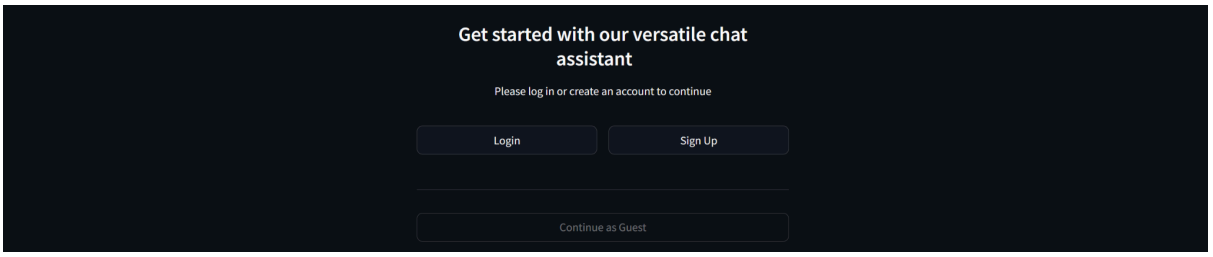


Figure 76: Interface for landing page

Signup page:

Users will see this when they click the "Sign Up" button on the landing page. They are required to fill up a structured form asking them for their email, username and password. When the user tries to submit the form, the system:

1. Checks for a valid email format
2. Verify that email and username does not exist in the MongoDB database
3. Ensures both password are entered correctly
4. Hashes password before storing into database
5. Displays error messages for validation errors

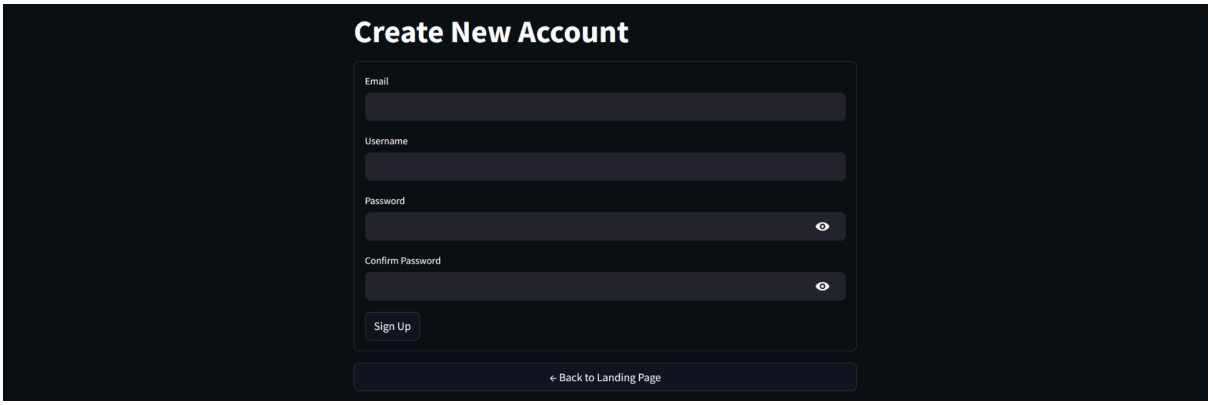


Figure 77: Interface for sign up page

Login page:

Users will see this when they click the "Login" button on the landing page. Users are required to fill up their credentials to enter the chatbot system.

When the user tries to submit the form, the system:

1. Checks whether username exists in database
2. Use secure verification method to cross-check stored hash password
3. Shows specific error messages for invalid username or password

Upon successful login, the system will store the authentication and username in session state.

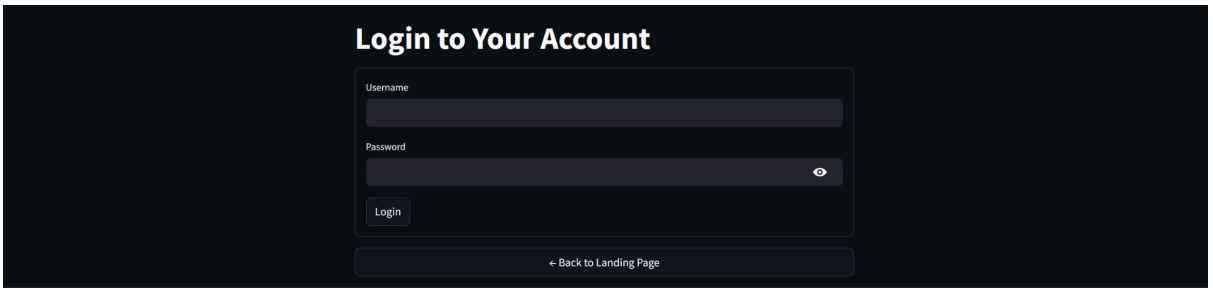


Figure 78: Interface for login page

5.2 Chatbot Page

Student login:

Upon successful login, the user will be brought to the initialisation page. The system will load the necessary services required to run the chatbot, connect to the Postgre database to retrieve conversation history, establish connection with mongoDB for prompting survey questions and also initialise all the session states required.

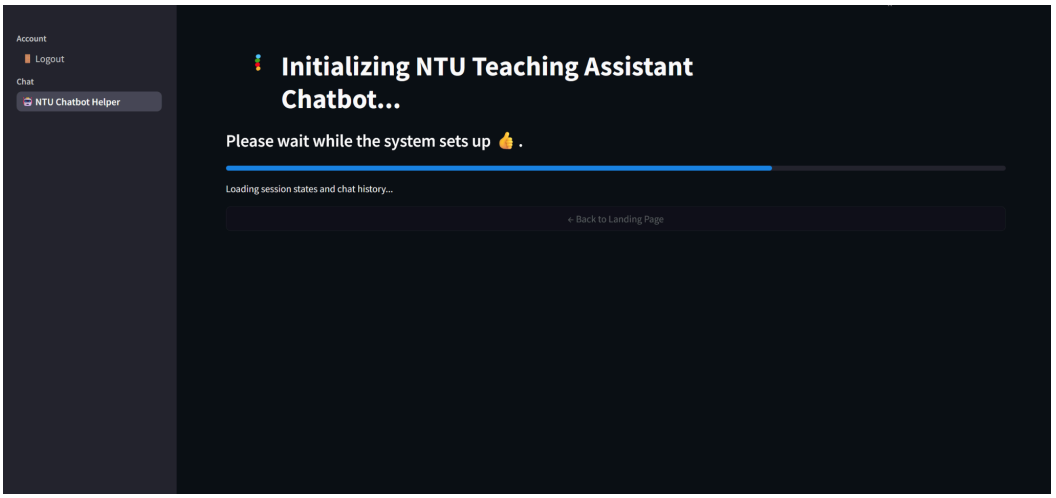


Figure 79: Interface for student initialisation page

Staff Login:

For professors/faculty members with the "staff" account, they will see an additional section called "Tools" on the sidebar. These tools will provide them with customisation options for configuring the chatbot which will be shown later.

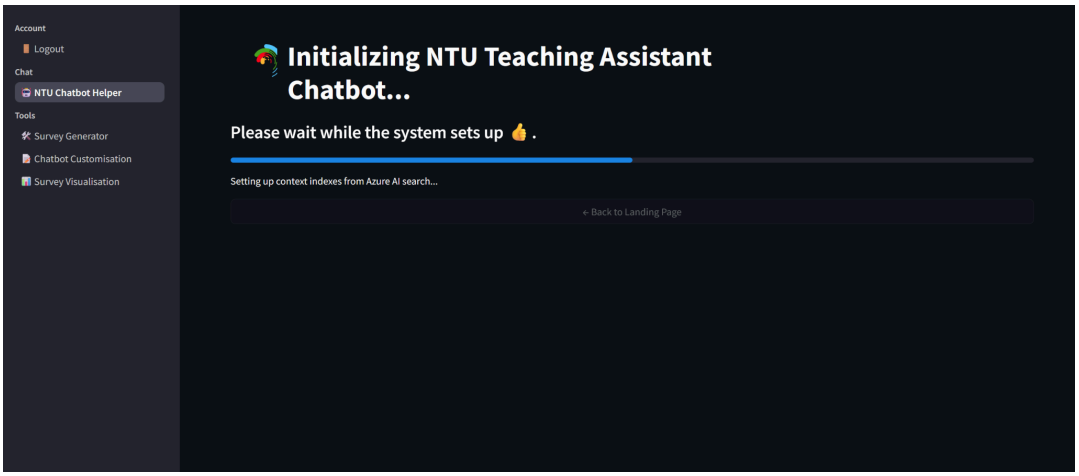


Figure 80: Interface for staff initialisation page

After initialisation is complete, the user will be redirected to the chatbot page, where they can interact with the chatbot. Their previous conversation history with the chatbot will also be displayed in the sidebar.

The "Chat History" sidebar uses the **backend conversational history** methodology outlined in Section 4.4.2. This allows users to switch between past conversations by clicking on the conversation buttons, or start a new conversation by clicking on "New conversation".

Figure 81: Interface for chatbot page

5.2.1 Querying the chatbot

The user can ask the chatbot questions relating to the course administration and logistics, which triggers the **course information relevance check** outlined in Section 4.3.2. The system retrieves context from the "information" index outlined in section 4.2.2 to supplement the response provided by the LLM.

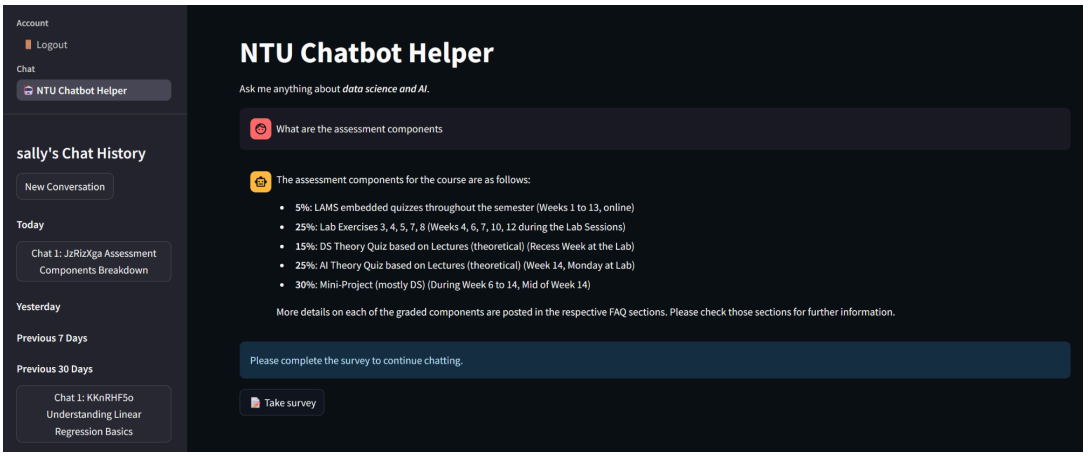


Figure 82: When user asks administrative questions

The user can also ask the chatbot questions relating to conceptual knowledge, triggering the **domain relevance check, past query relevance check and conceptual prompt** outlined in Section 4.3.2. The system retrieves context from the "notes" index outlined in Section 4.2.2 and structures the response in a standard format. The response will always include an explanation from course notes, followed by the LLM's explanation of the concept, along with suggested links for further references. If there is a relevant video resource, then a Youtube video will also be displayed.

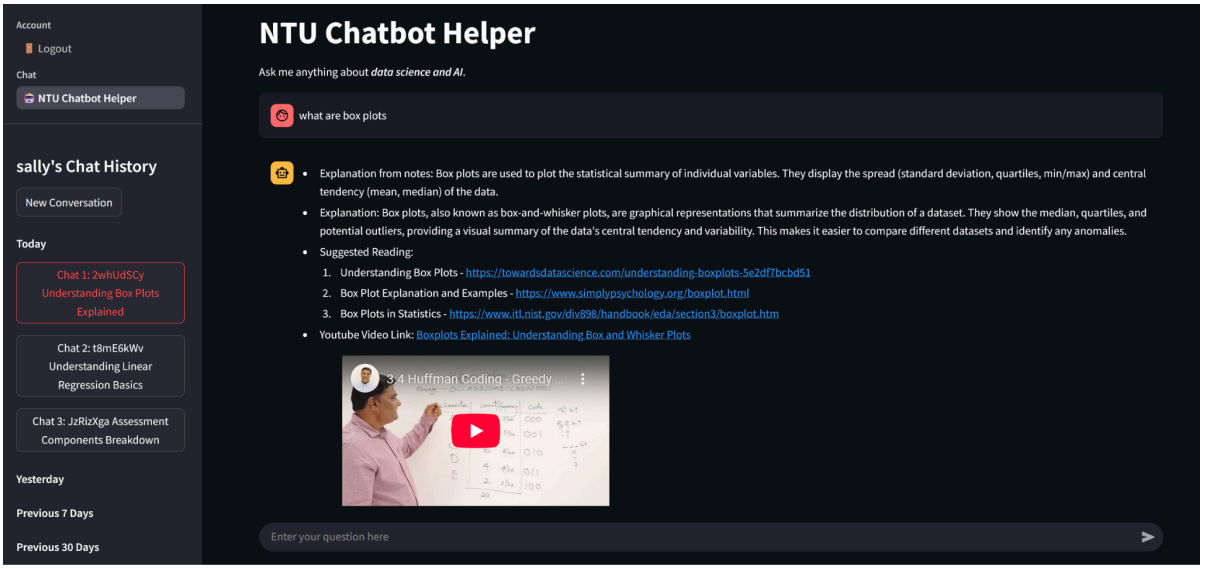


Figure 83: When user asks conceptual questions

5.2.2 Survey interface embedded in chatbot

After the response is displayed to the user, the **survey prompting algorithm** outlined in Section 4.5 is executed to check for any relevant survey questions from the "survey" index to prompt the user. If there are relevant survey questions, the text input field will be hidden to prevent further input and a "Take Survey" button will appear, requiring the user to complete the survey questions before continuing the conversation with the chatbot.

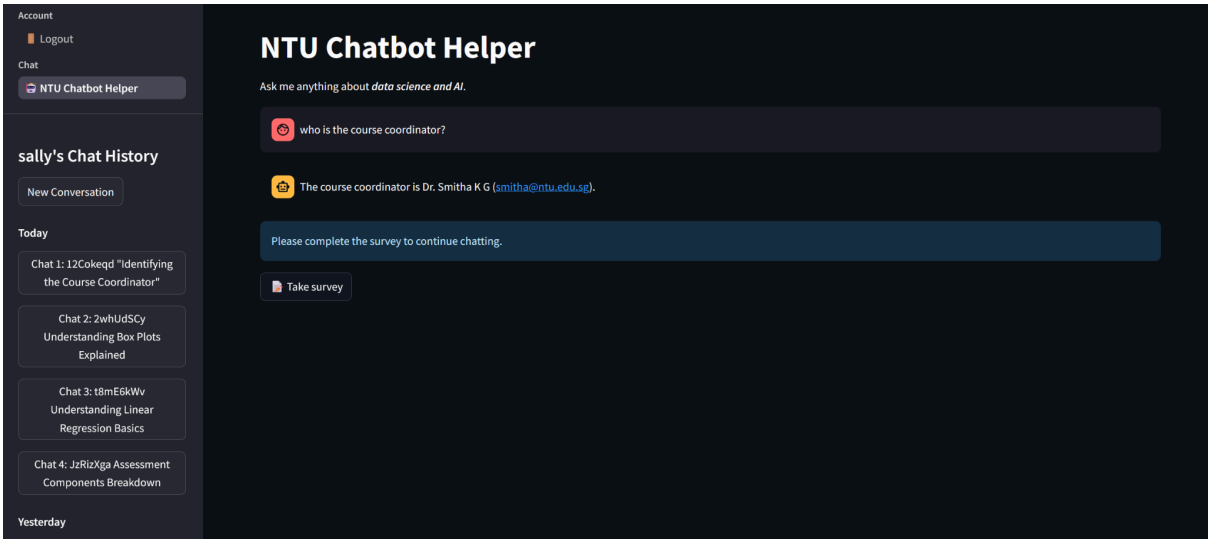


Figure 84: When there are relevant survey questions to the user's query

When the user clicks on the "Take Survey" button, the system will display the relevant survey questions to the users. This approach aligns with the project objective of preventing survey fatigue by limiting the number of questions presented to the student at a time, ensuring a more user-friendly experience.

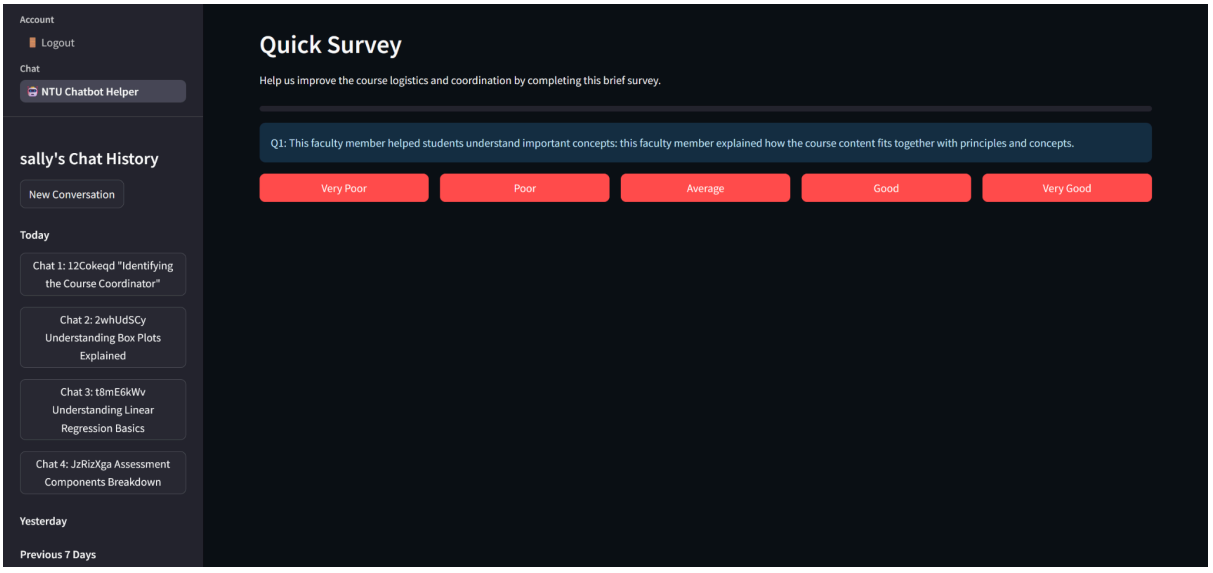


Figure 85: Survey questions interface

After the user finishes filling up the survey questions, they will be redirected back to the chatbot interface, where they can continue the conversation with the chatbot.

5.3 Survey Generator Page

For professors/faculty members with the "staff" account, they will be given special access to the survey generator page. This access is managed through role-based permissions, ensuring that only authorised users can create surveys

The Survey Generator module offers two pathways which users can take to develop survey questions, catering to different needs and expertise levels:

1) Users can use the **left option** to upload their own survey files, and this module is intended for users with existing survey content
2) Users can use the **right option** to use an LLM to help them come up with survey questions and this module is intended for users who require guidance coming up with survey questions from scratch

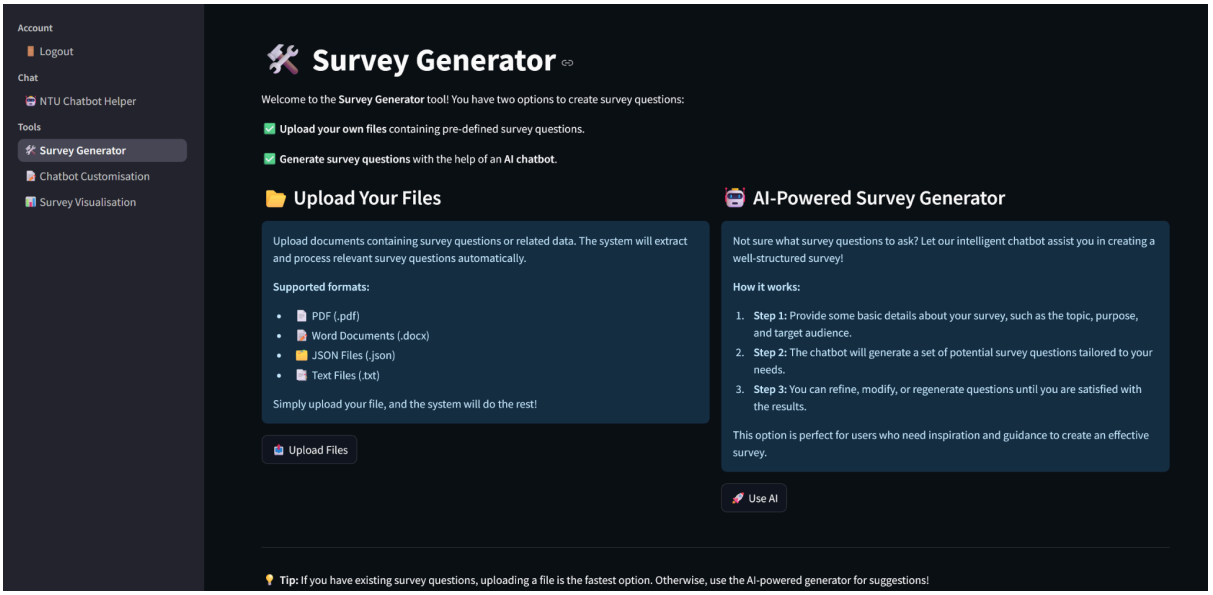


Figure 86: Main survey generator interface

5.3.1 When user clicks on the "Use AI" (right option) button:

Users provide basic information about their survey objectives, target audience, and topic. The LLM will then analyse these requirements and generate tailored survey questions for the user using the methodology outlined in section 4.7.

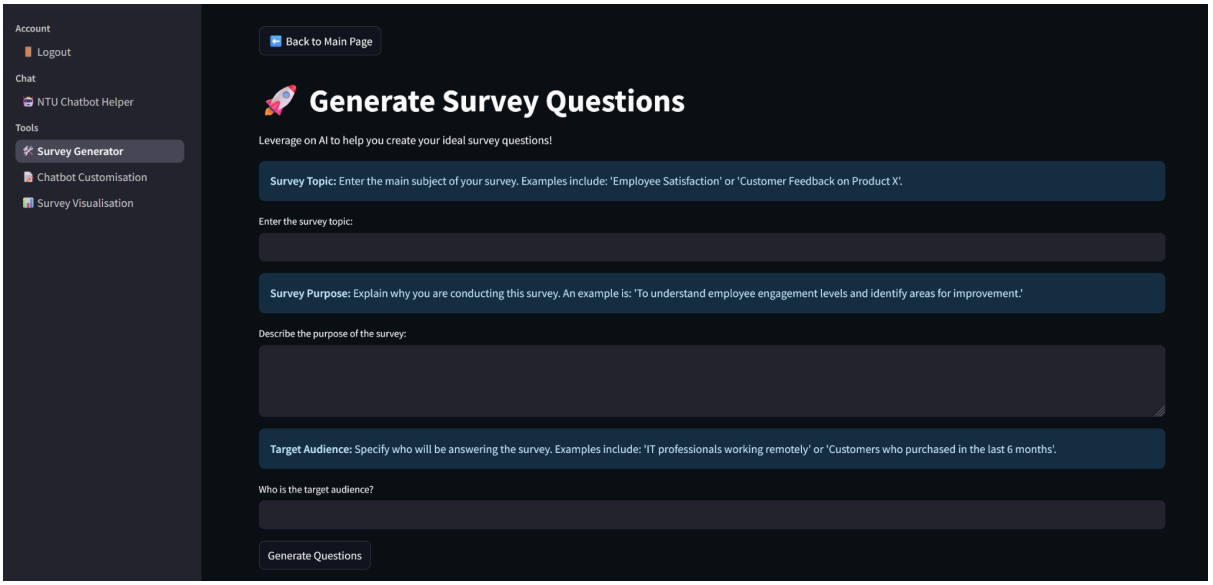


Figure 87: Users are required to fill up some basic details to generate their survey

After the LLM generates a set of survey questions based on the requirements specified, the user will be brought to the "Generated Survey Questions" page. The main section displays all the AI-generated survey questions with an expandable/collapsible interface, allowing the user to view the question text along with its associated tags.

Under each question card, the user has 3 options:

1) Edit the question manually by clicking on **"Edit Question/Tags manually"**
2) Delete the question if it is not satisfactory by clicking on **"Delete Question"**
3) Regenerate the question with AI by clicking on **"Regenerate Question with AI"**

There are also other useful tools at the top of the page such as:

1) The**"Back to Main Page"** button at the top which allows users to return to the main interface
2) The **"Start Over"** button which brings the user through the process to create a new set of questions again.
3) The **"Add Question"** button which allows the user to manually add a new question to existing question set
4) The **"Next"** button which proceeds to the next step when the user is satisfied with the question set

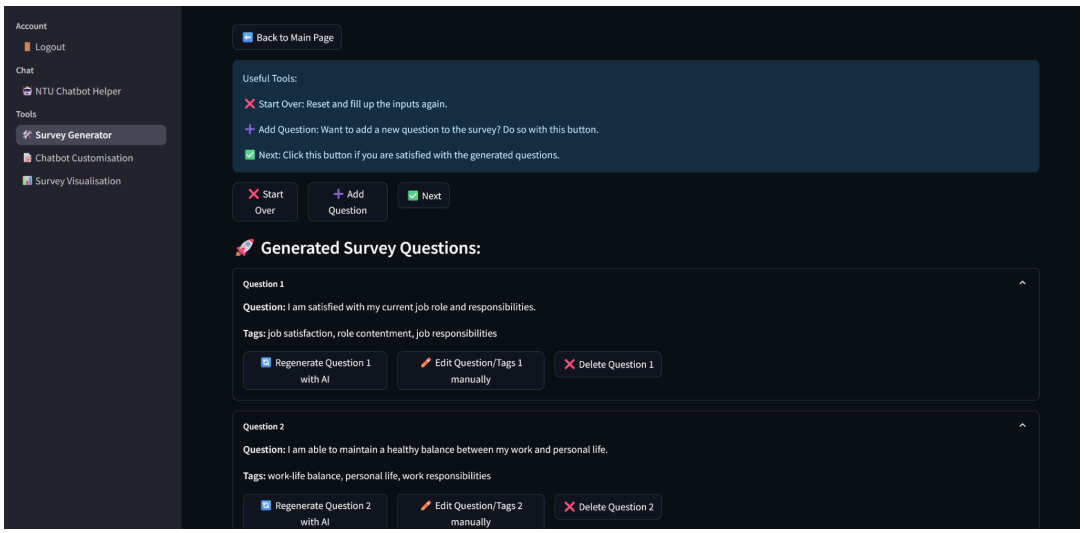


Figure 88: Generated survey questions page

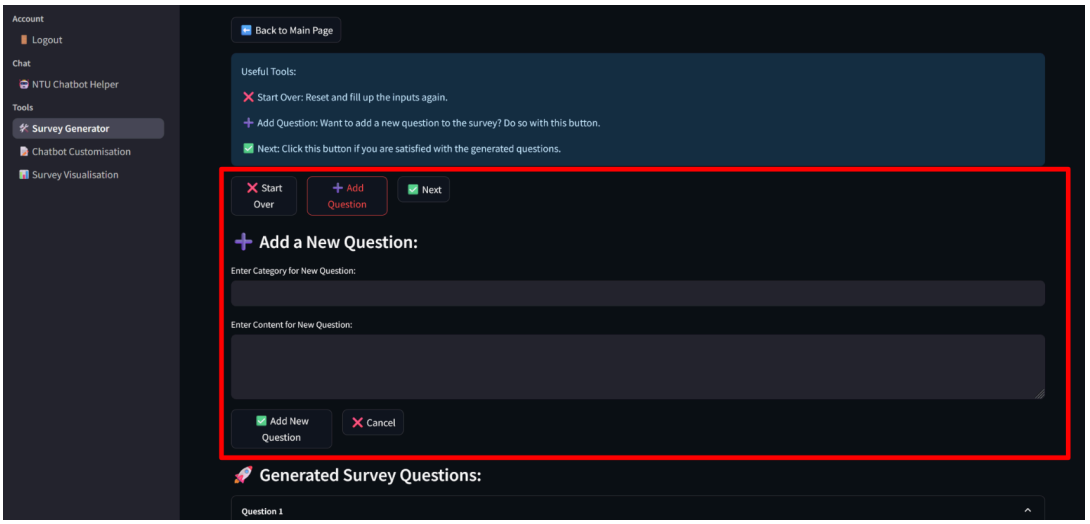


Figure 89: When the user clicks on the Add Question button

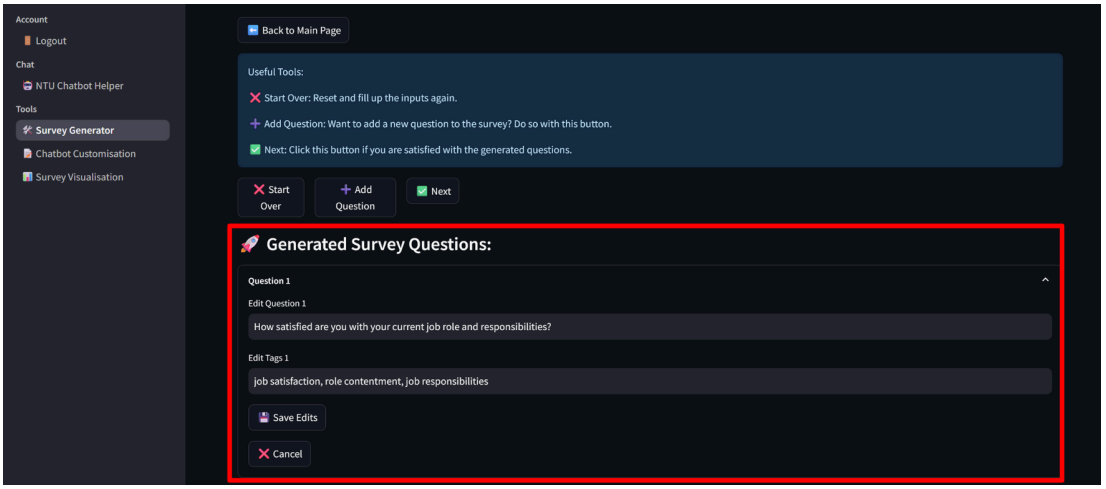


Figure 90: When the user clicks on the Edit Question/Tags manually button

If the user is satisfied with the questions and clicks “Next”, they will be brought to the “Survey Questions Confirmation” page, where they can do one last check on the list of survey questions with their associated tags that they have generated/edited.

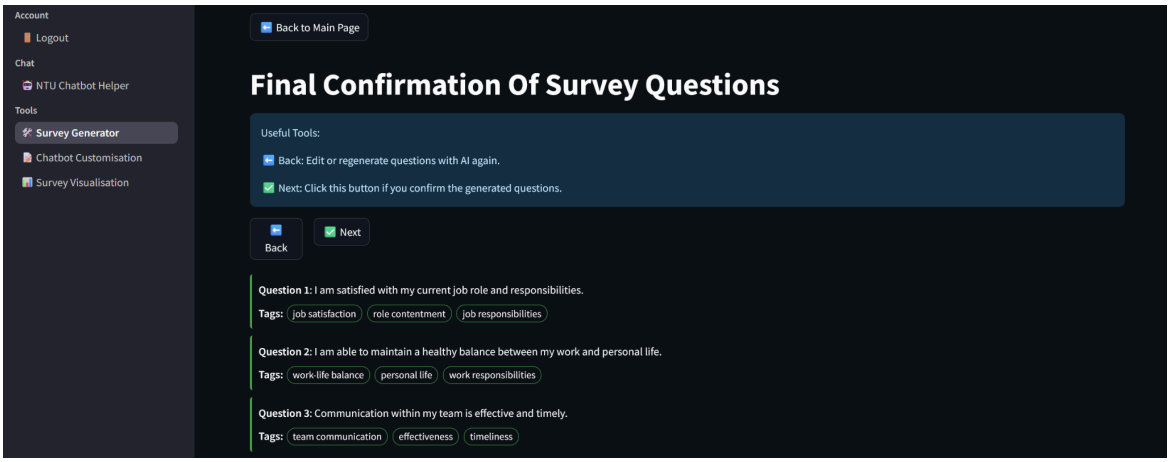


Figure 91: Survey questions confirmation page

The system will then help the user to upload these survey questions to the survey index on Azure AI Search which was outlined in section 4.7. When this process is done, the user will be given a confirmation success message and prompted to go to the chatbot, to try out the survey prompting algorithm.

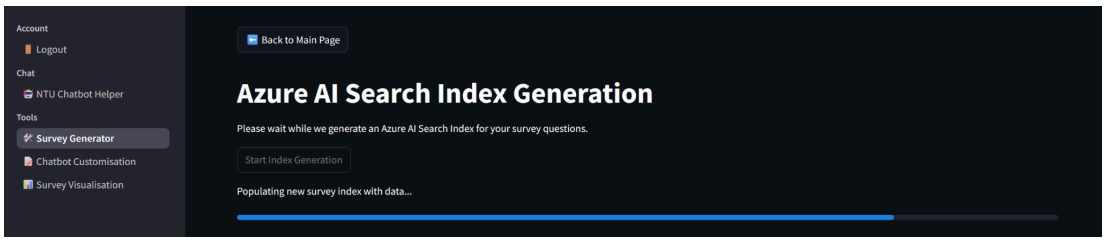


Figure 92: Survey questions index generation page

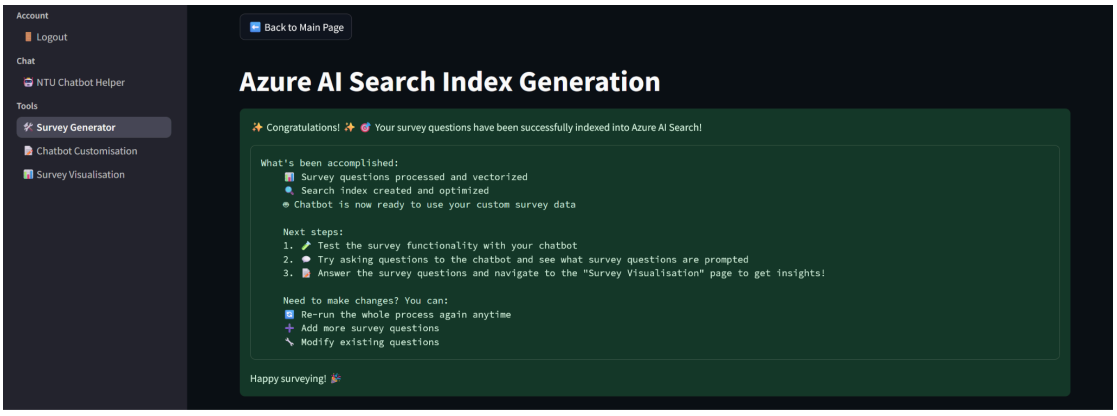


Figure 93: Survey questions successfully populated in survey index

5.3.2 When user clicks on the "Upload Files" (left option) button:

Users can upload their own survey files on this page, with support for four common file formats: **JSON, TXT, PDF, and DOCX**. A guide at the bottom of the page provides instructions on formatting survey questions in these file types if needed. However, manual formatting is optional, as the LLM will process and standardise the questions automatically later.

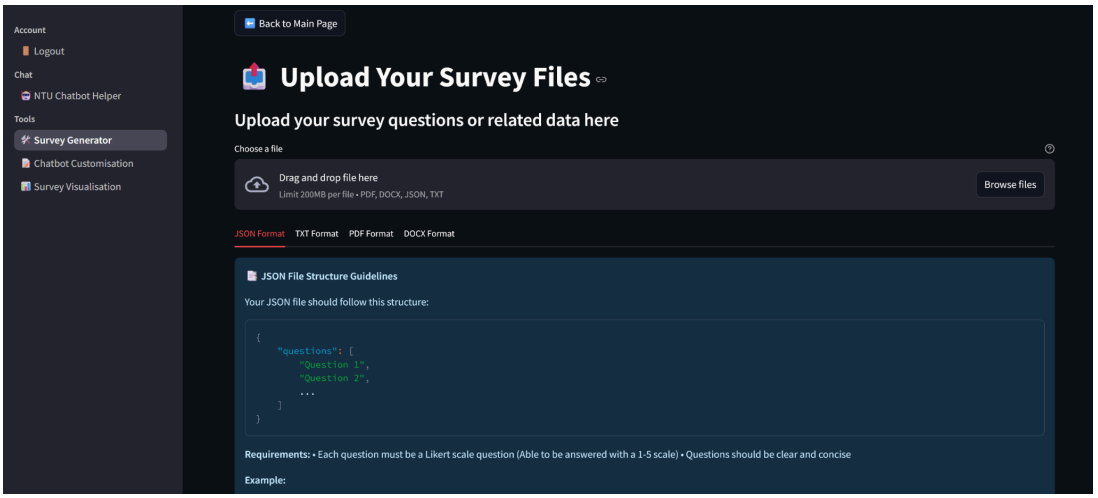


Figure 94: Upload survey files page

After the user uploads the file, the LLM will perform a check to ensure that only survey documents are uploaded to prevent unrelated files from being submitted. Moreover, the user can validate the extracted text from the uploaded file to confirm its accuracy before proceeding.

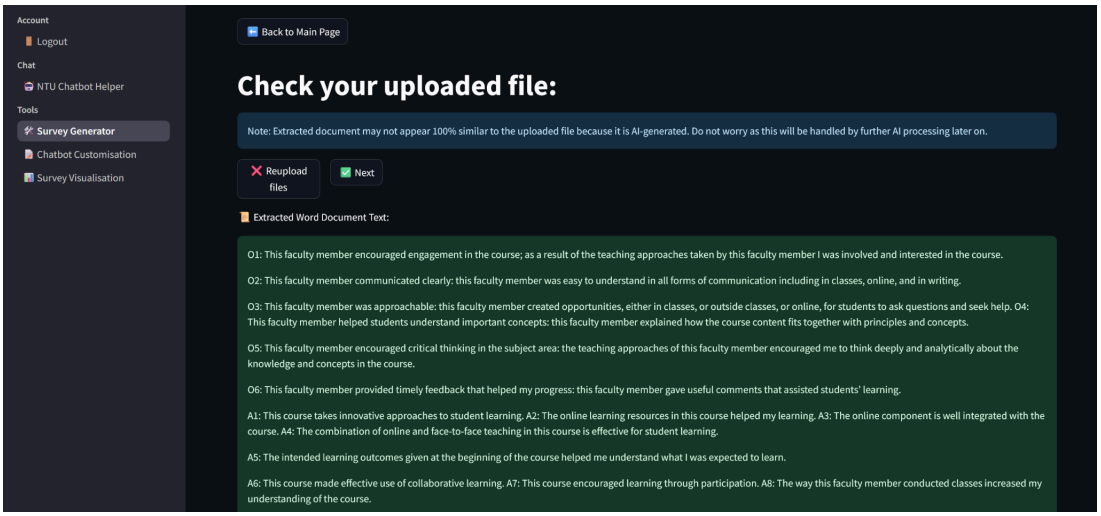


Figure 95: Upload survey files page

When the user clicks “Next”, they will be brought to the “AI processing” page. In this page, the LLM will perform several quality checks to ensure that:

1) Questions are answerable on a Likert scale of 1-5 only
2) Questions do not have grammatical errors
3) All questions are in the same domain or category

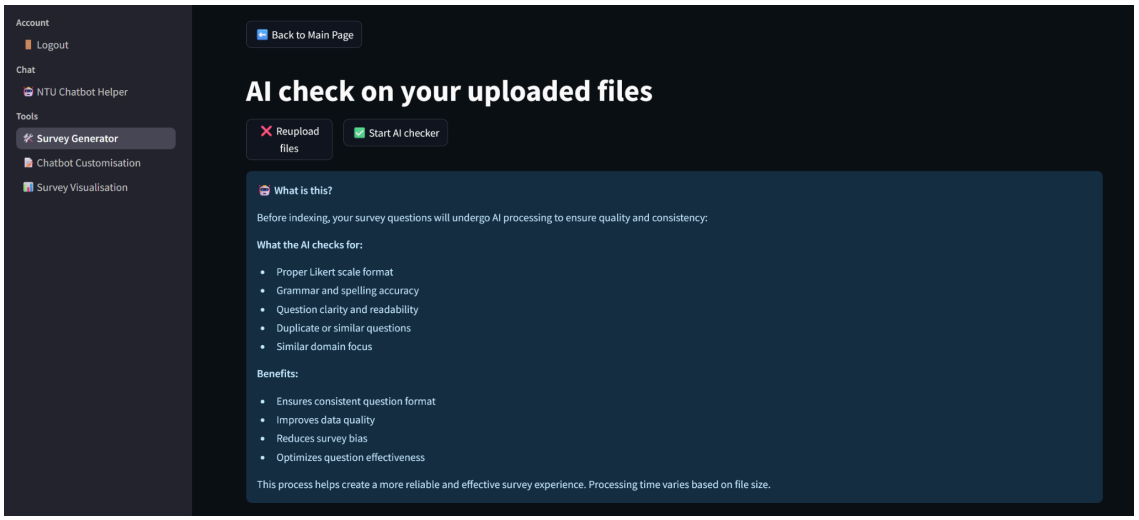


Figure 96: AI processing page to ensure quality of survey questions uploaded

If any questions are flagged by the LLM, users can choose to edit them either by manually changing the question or using an AI-generated fix. However, this process is not compulsory. If the user believes that the AI’s flagged issue is not important, they proceed without making any changes.

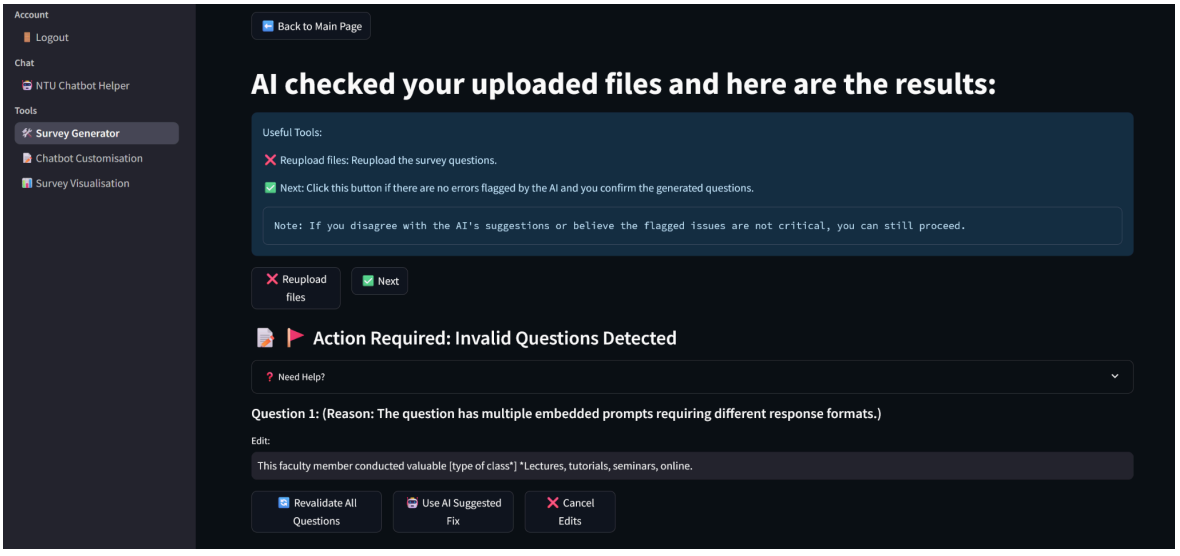


Figure 97: AI processing page when there are flagged questions

If all questions are acceptable, then this interface will be shown to the user instead.

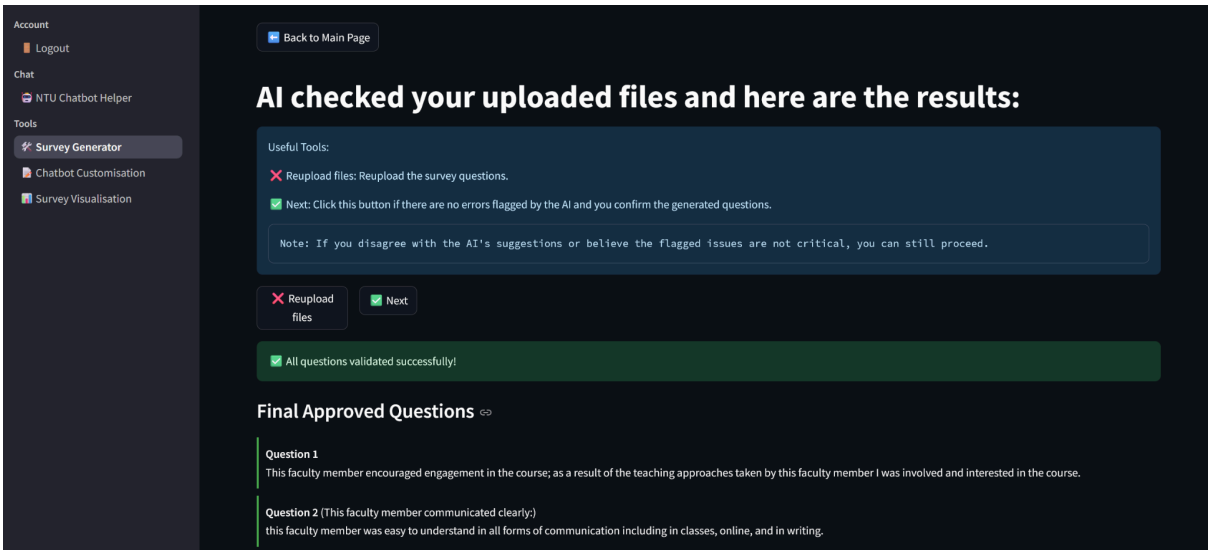


Figure 98: AI processing page when all survey questions are acceptable

After this, the system will generate the tags for each associated question and help the user to upload these survey questions to the survey index on Azure AI Search which was outlined in section 4.7. After this process is complete, the user will be given a confirmation success message and prompted to go to the chatbot to try out the survey prompting algorithm.

5.4 Chatbot Customisation Page

For professors/faculty members with the "staff" account, they will be given special access to the chatbot customisation page. This access is managed through role-based permissions, ensuring that only authorised users can change system settings.

On this page, users have three options. They can modify the chatbot context, upload course administration and information details, and also upload course notes to the relevant indexes.

5.4.1 Modifying chatbot context

Defining the chatbot context is crucial because it ensures that the chatbot only responds to questions that are within the scope that is set. Professors/faculty members can tailor the chatbot context to the current modules that they are teaching. For example, if the chatbot is designed for students taking SC1015 (Introduction to data science and AI), then the context would be set as "data science and AI". This guarantees that the chatbot is restricted to only answering questions in this scope, preventing users from asking unrelated questions.

Users can see the current context of the chatbot, and also specify a new context for the chatbot if they require it by filling up the text input.

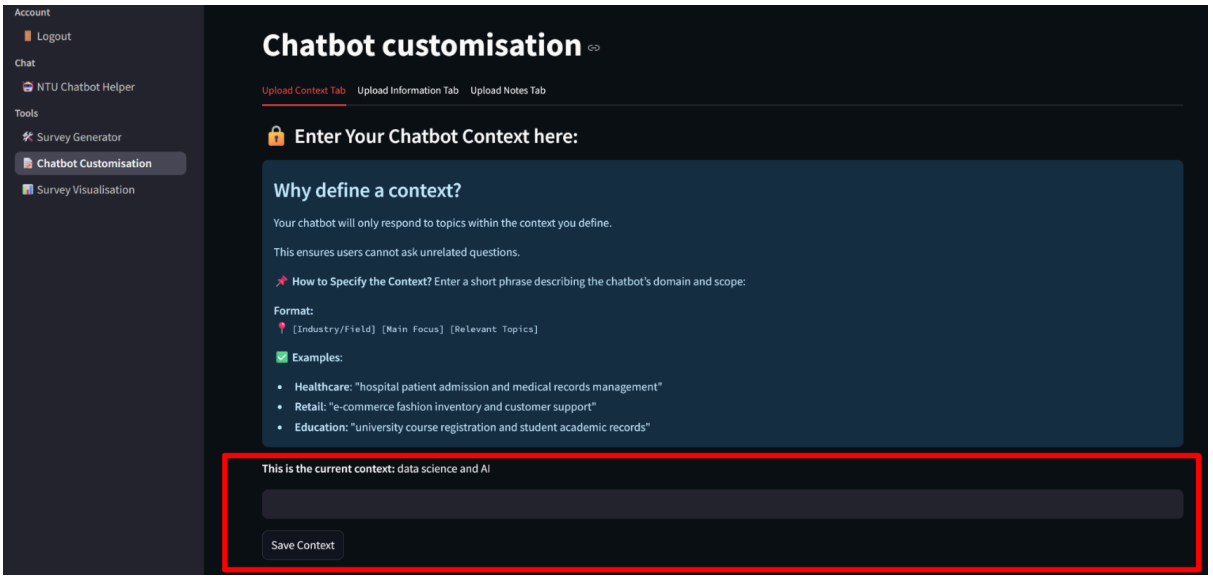


Figure 99: Chatbot customisation page with context tab

5.4.2 Uploading information to the chatbot

Uploading information to the chatbot is crucial as it will equip the chatbot with specific knowledge to address user's queries on administrative or logistic matters. Professors/faculty members can upload course information to the current modules that they are teaching.

For example, if the chatbot is designed for students taking SC1015 (Introduction to data science and AI), then the information uploaded can be course FAQs and guidelines, such as assessment components, final exam dates, contact information etc. Without this knowledge, the chatbot will lack access to unique content, limiting its ability to provide customised responses catered for different courses.

Users can upload their information files on this page, with support for two common file formats: **PDF and DOCX**. After the user uploads the files, the system is responsible for chunking these documents using Langchain text splitters and uploading them onto the "information" index on Azure AI Search.

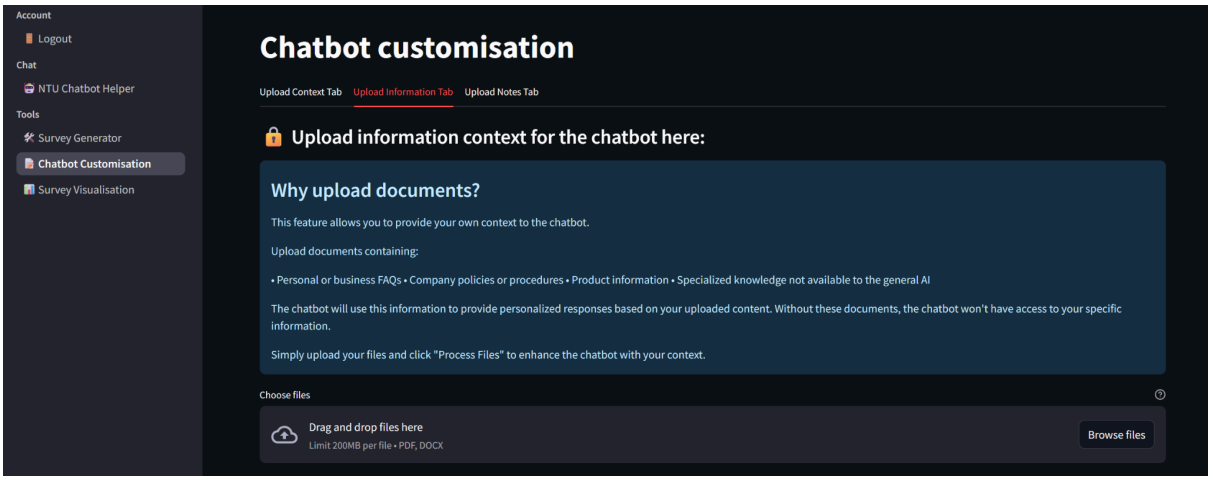


Figure 100: Chatbot customisation page with information tab

5.4.3 Uploading notes to the chatbot

Uploading notes to the chatbot is crucial as it will equip the chatbot with course notes knowledge to guide users with conceptual queries. Students can ask conceptual questions and the chatbot will reference the course notes to provide answers, helping students prepare for final exams. Professors/faculty members can upload course notes such as study notes, research papers, books, or any text-based material relevant to the current modules that they are teaching.

For example, if the chatbot is designed for students taking SC1015 (Introduction to data science and AI), then the notes uploaded can be weekly notes from Week 1 to Week 13. With this knowledge, the students will receive more relevant responses tailored to the specific course content.

Users can upload their information files on this page, with support for two common file formats: **PDF and DOCX**. After the user uploads the files, the system is responsible for chunking these documents using Langchain text splitters and uploading them onto the "notes" index on Azure AI Search.

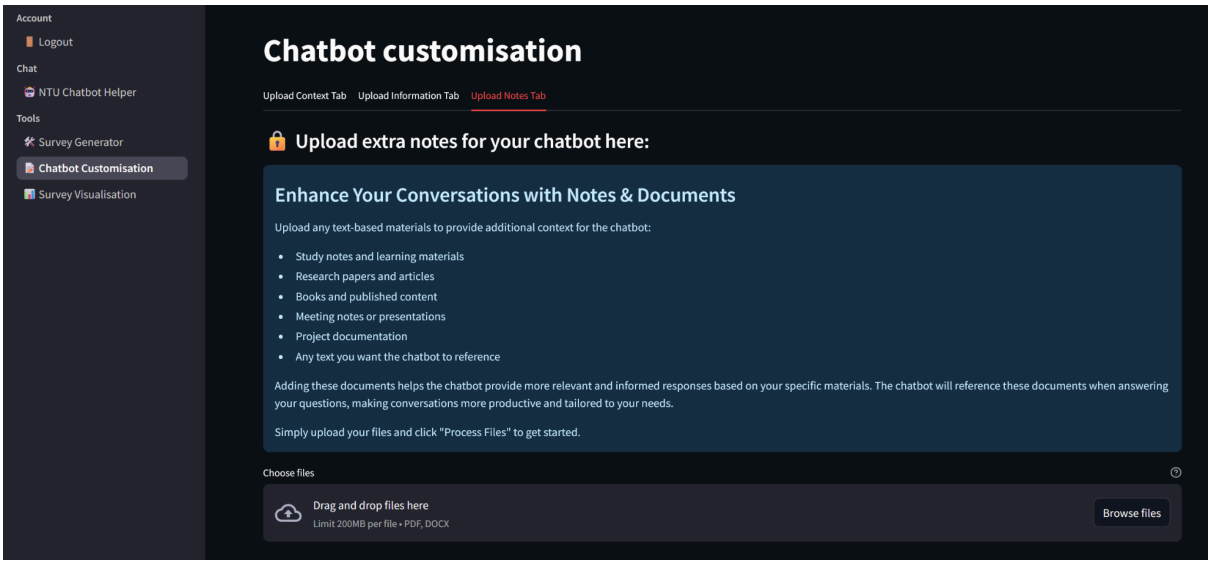


Figure 101: Chatbot customisation page with notes tab

5.5 Survey Visualisation Page

For professors/faculty members with the "staff" account, they will be given special access to the survey visualisation page.

When the professor first opens this page, the survey responses will be fetched from the MongoDB database and cached for processing and visualisation. There will be a loader displayed when this process is happening.

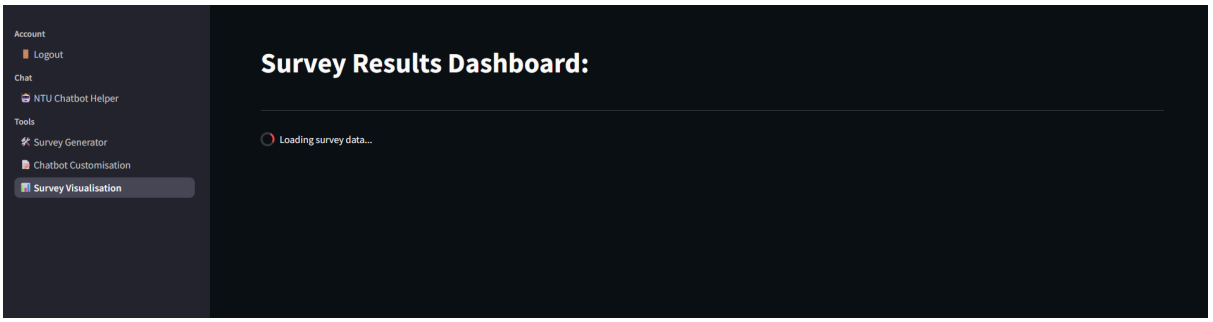


Figure 102: Survey responses dashboard initialisation

After the survey responses are fetched and cached, professors will see a high-level overview of the survey responses collected. This includes the total number of survey responses submitted by all students and the distinct types of questions answered.

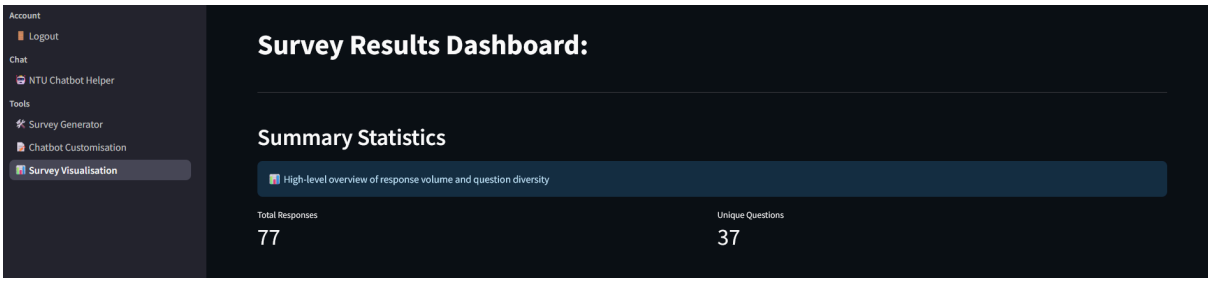


Figure 103: Survey responses dashboard initialisation

Scrolling down, professors can access some visualisation components including:

1. **Survey question tracker visualisation** that tracks which survey questions have been answered by each student, to reveal patterns in how students interact with the chatbot

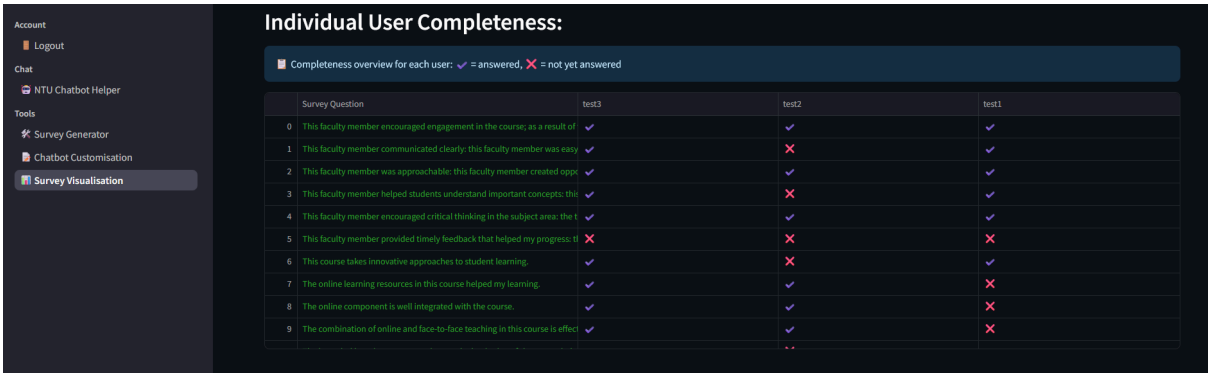


Figure 104: Survey question tracker interface

2. **Survey responses timeline visualisation** that displays a timeline of submission patterns, allowing professors to analyse when students are engaging with the chatbot the most (eg. finals period, weekdays, weekends etc.)

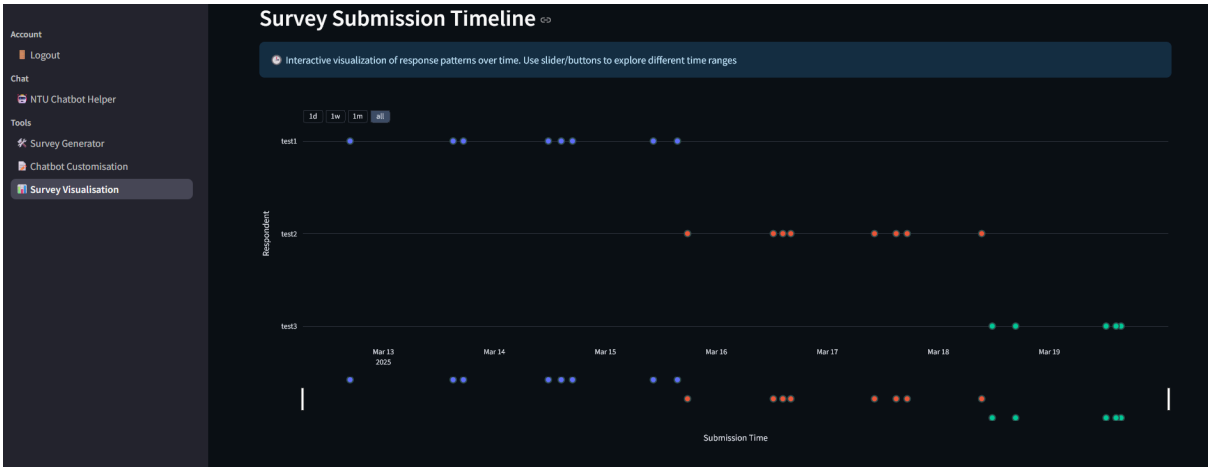
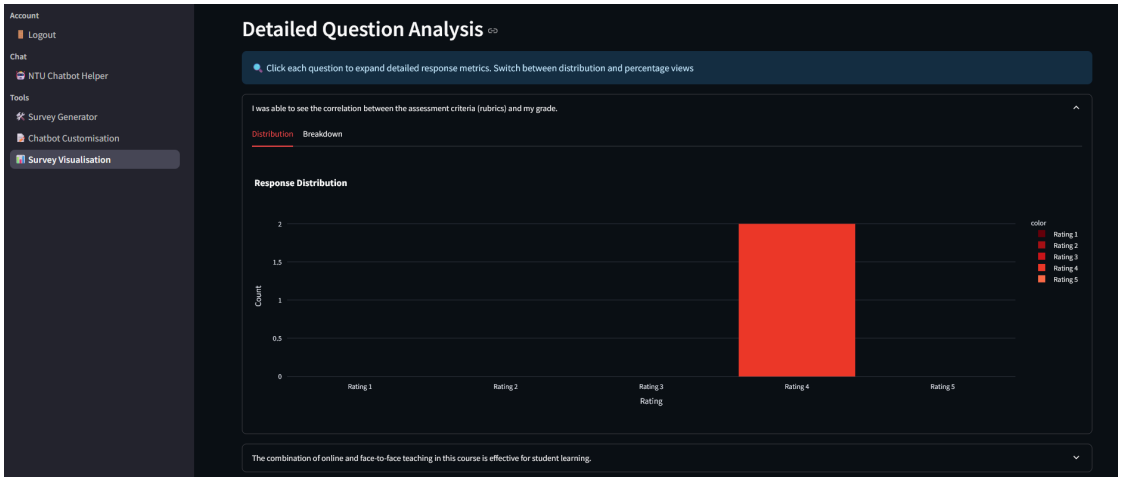


Figure 105: Survey responses timeline interface

3. **Survey answers distribution visualisation** that displays a stacked horizontal bar chart showing how survey responses are distributed across the different ratings of 1-5 for each question. This makes it easy to identify trends and outliers, such as glancing quickly to find out which questions have the most positive/negative responses

Figure 106: Survey answers distribution interface

4. **Detailed question analysis sections** that provide bar and pie charts visualisation for individual questions through expandable tabs



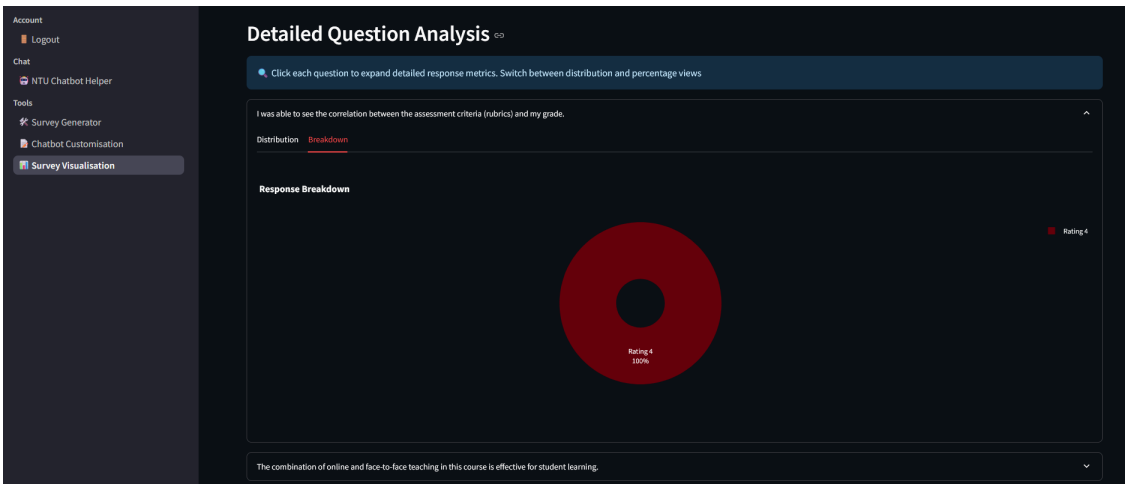


Figure 106 & 107: Detailed question analysis section interface

6. User Testing & Feedback

Deployment Environment

The system was deployed and hosted on Azure App Services under the domain customised-chatbot.azurewebsites.net for a limited-time pilot test. For this evaluation period, the module scope has been shifted from SC1015 (Introduction to Data Science and AI) to CS0888(AI & New Technology Law), a BDE that I am currently enrolled in.

Testing environment system configuration:

Under the chatbot customisation page, the chatbot was reconfigured to cater to the shift in module.

Firstly, the chatbot context has been changed to “AI & New Technology Law”,restricting its responses to this specific domain.

Secondly, to ensure that the chatbot has indepth knowledge of the course content:

- A PDF containing the course FAQs and syllabus curated by the course coordinator was uploaded to the “Information” index of Azure AI Search.
- Complete weekly course materials, including Week 1-13 lecture notes and tutorial handouts, were added to the "Notes" index of Azure AI Search

This configuration ensured that the chatbot now has full context awareness of any queries that are relating to this module, to address both course administration or course conceptual queries effectively.

Evaluation methodology:

The survey questions used for the SC1015 module were retained for this pilot test. as its questions were designed to be generic and adaptable to various courses. *(The list of survey questions used throughout this project is available in the appendix)*

Three students currently enrolled in the CS0888 module were invited for the testing of the prototype. The students were:

- Told the problem statement and objectives of this project
- Given a comprehensive walkthrough on the navigation and functionalities of the chatbot
- Asked to use the chatbot to prepare for their upcoming mid-term examination (covering concepts from weeks 1-8)
- Asked to complete survey questions that were prompted whilst interacting with the chatbot

Testing process:

The students utilised the chatbot to ask both administrative and conceptual queries as part of their mid-term examination preparation. After the mid-term examination, feedback was collected from the students about the effectiveness of the chatbot in answering queries and the survey responses provided were extracted for visualisation analysis.

Analysis and effectiveness of user interactions:

The analysis of user interactions with the chatbot revealed distinct usage patterns::

Student 1 using the test1 account primarily queried the chatbot for conceptual understanding. Some queries asked included questions like "*Can u elaborate on the 1990s scotus antitrust case?*" or "*Lawful processing: 2 possible bases what are the 2 possible bases*". From this usage pattern, it shows that the chatbot has fulfilled its objective to function as an effective study aid, allowing students to access course concepts through a conversational interface while receiving both course notes-based information and LLM-enhanced explanations.

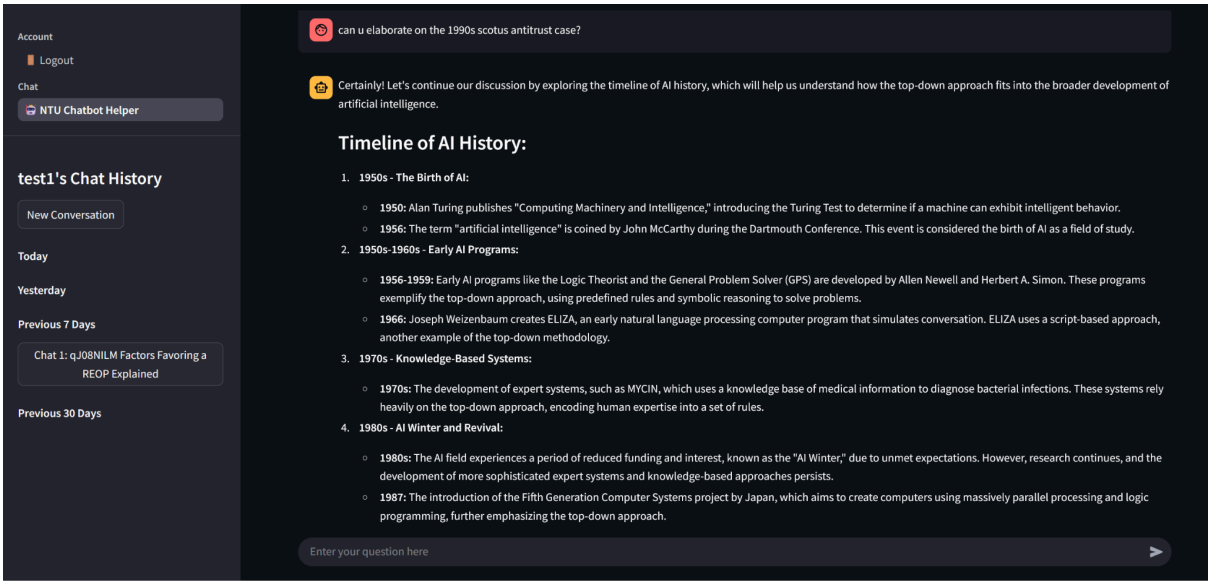


Figure 108: Example of test1's conceptual query to the chatbot

Student 2 using the test2 account primarily queried the chatbot for administrative queries. Some queries included questions like "*what is the course outline?*", " *how do i score well in exams*" and "*when is test 1 and test 2?*". From this usage pattern, it also shows that the chatbot has fulfilled its objective of providing immediate access to course administration information, eliminating the need for students going to NTULearn to locate this information in the course syllabus document manually.

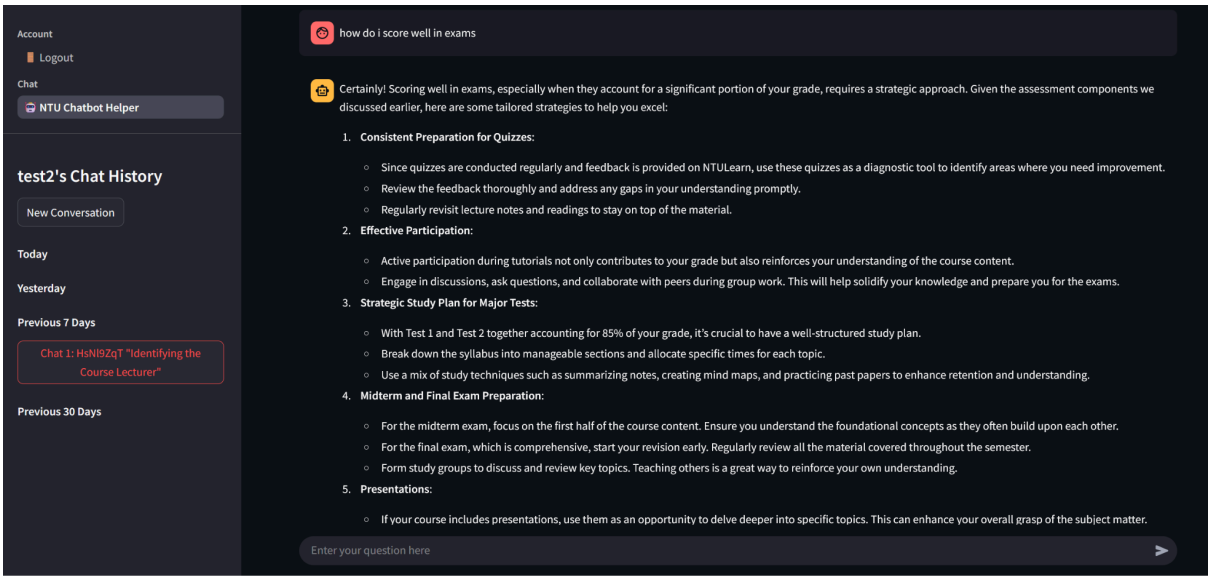


Figure 109: Example of test2's administrative query to the chatbot

Student 3 using the test3 account used the chatbot for a hybrid approach, asking both administrative and conceptual queries. Some queries included questions like "*what are the intended learning outcomes of the course*" and "*how does tort of negligence work for both the plaintiff and defendant?*".

The diverse usage patterns of the chatbot observed across all three students provided evidence in suggesting that the chatbot has successfully addressed the pain points of students identified in the problem statement. These include:

1) Eliminated the frustration of students of navigating through the many course information and materials available on NTULearn, as the chatbot served as a all in one hub for any queries related to the course
2) Provided immediate responses to students' queries without waiting for any professor intervention

Survey collection mechanism effectiveness:

The survey questions module provided evidence in collecting promising survey responses during the pilot testing. The system has successfully prompted different survey questions to each student based on their personalised usage patterns of the chatbot, enabling the professors a collection of diversified feedback data. This addresses the objective to allow professors to collect personalised and relevant student feedback rather than generic responses in filling up traditional survey forms.

However, it is also important to note certain limitations in the survey questions mechanism's implementation. Throughout the one-week limited pilot testing phase:

1) Not all 37 available survey questions were prompted to each student
2) Each student only answered approximately 10-20 survey questions during their one week interaction with the chatbot

These limitations imply that for the chatbot to effectively collect timely and relevant student feedback throughout an entire semester, sustained and regular engagement of the students with the chatbot would be necessary. Essentially, the incremental collection nature of the survey mechanism requires consistent individual student interaction to build a complete feedback profile over time.

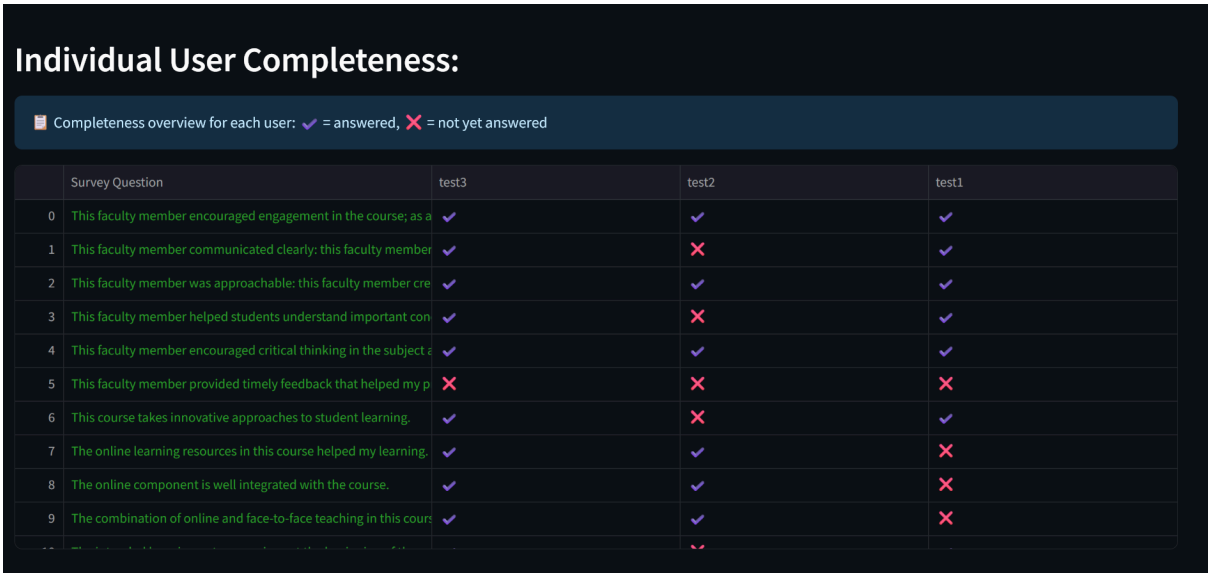


Figure 110: Example of survey questions responded to by each student in the pilot testing

7. Project Extension

Since all of the system components are modular and adaptable, it can potentially address universal needs across many different industries to collect structured feedback for either their products or services and provide domain-specific knowledge assistance in the chatbot.

The proposed solution is not limited to the education sector but can be extended to industries such as **healthcare, retail, finance, and human resources** by leveraging its two core modules:

Survey generator module:

Using the survey generator module, surveys that are related to the particular industry can be created. This can range from:

1. **Healthcare:** Generate patient feedback surveys, symptom tracking questionnaires, post-treatment follow-ups etc.
2. **Retail**: Create customer satisfaction surveys, product feedback surveys etc.
3. **HR**: Employee engagement surveys, company satisfaction feedback etc.

Chatbot customisation module:

Using the chatbot customisation module, the chatbot can be given industry specific knowledge awareness to equip it with context to answer user's queries. This can range from:

1. **Healthcare:** Restrict chatbot to only answer healthcare related questions, upload clinic/hospital FAQs, upload medical symptoms notes etc
2. **Retail:** Restrict chatbot to only answer product/retail-specific related questions, upload product information sheet etc
3. **HR:** Restrict chatbot to only answer company/work related questions, upload job listing documents etc

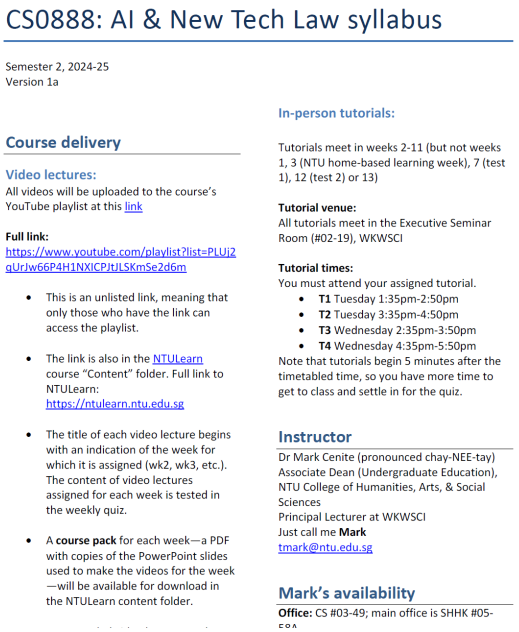
8. Future works

While the chatbot prototy[e implementation successfully addressed the students' and professors' pain points in the problem statement, several areas for improvement emerged during the implementation and testing phases. This section outlines potential enhancements for future versions of the system.

Enhanced document processing capabilities:

The current implementation uses Langchain document loaders (`PyPDFLoader`, `Docx2txtLoader`) to process the uploaded files before uploading them to the "Information" and "Notes" index when professors use the chatbot customisation page. However, limitations were observed when processing documents with complex layouts, images, and mathematical equations. These limitations resulted in broken text flow and incomplete text extraction.

Hence, it is suggested that more sophisticated document processing algorithms capable of handling multi-modal content are implemented by using advanced document loaders with image processing capabilities like PaddleOCRLoader or TesseractOCRLoader. It is also possible to use an LLM-based approach to intelligently parse and structure documents. Google's Gemini models are capable of handling multi-modal content effectively.



CS0888: AI & New Tech Law syllabus

Semester 2, 2024-25
Version 1a

Course delivery

Video lectures:
All videos will be uploaded to the course's YouTube playlist at this link

Full link:
https://www.youtube.com/playlist?list=PLUi2qUrJw66P4H1NXICPJtJL5KmSe2d6m

- This is an unlisted link, meaning that only those who have the link can access the playlist.
- The link is also in the NTULearn course "Content" folder. Full link to NTULearn: https://ntulearn.ntu.edu.sg
- The title of each video lecture begins with an indication of the week for which it is assigned (wk2, wk3, etc.). The content of video lectures assigned for each week is tested in the weekly quiz.
- A **course pack** for each week—a PDF with copies of the PowerPoint slides used to make the videos for the week—will be available for download in the NTULearn content folder.

In-person tutorials:

Tutorials meet in weeks 2-11 (but not weeks 1, 3 (NTU home-based learning week), 7 (test 1), 12 (test 2) or 13)

Tutorial venue:
All tutorials meet in the Executive Seminar Room (#02-19), WKWSCI

Tutorial times:
You must attend your assigned tutorial.
- **T1** Tuesday 1:35pm-2:50pm
- **T2** Tuesday 3:35pm-4:50pm
- **T3** Wednesday 2:35pm-3:50pm
- **T4** Wednesday 4:35pm-5:50pm

Note that tutorials begin 5 minutes after the timetabled time, so you have more time to get to class and settle in for the quiz.

Instructor

Dr Mark Cenite (pronounced chay-NEE-tay)
Associate Dean (Undergraduate Education), NTU College of Humanities, Arts, & Social Sciences
Principal Lecturer at WKWSCI
Just call me **Mark**
tmark@ntu.edu.sg

Mark's availability

Office: CS #03-49; main office is SHHK #05-

Figure 111: Example of document with complex layout

Improved response generation for conceptual queries:

In the current implementation, the system generates conceptual explanations to student''s query by retrieving and explaining context from the "notes" index in Azure AI Search, providing LLM-based explanations, followed by providing reference links and relevant YouTube videos. However, there are instances of broken or outdated links in the generated responses.

Hence, it is suggested that future enhancements focus on implementing link validation methods to detect broken links such as integrating Langchain Agent Tools, including YouTube and Google Search APIs, to dynamically locate these relevant reference resources.

More features to encourage sustained engagement with chatbot:

From the testing feedback, it indicated that not all survey questions from the "survey_questions" index were prompted to the students during a one-week interaction period. Continuous chatbot usage is crucial, to ensure that students using the chatbot complete most if not all of the survey questions, to increase the quality of feedback for the professors.

Therefore, additional features should be developed to incentivise students to use the chatbot regularly.

Hence, some extra features suggested:

- Implementing a MCQ question generation system that creates questions to test the student's conceptual understanding of concepts based on content from the "notes" index
- Implementing a learning path feature, where the chatbot categorises students based interaction patterns/questions asked and identified knowledge gaps

Advanced search algorithms for survey questions prompting:

The current implementation uses hybrid search to search both content and tag fields, which is an improvement over basic keyword-based searches. However, more advanced search techniques could further enhance the relevance of prompted survey questions.

Hence, it is suggested to implement Semantic search with query rewriting as a premium service through Azure. Semantic search has capabilities to better understand the contextual meaning of student queries, which will improve the accuracy of the results fetched.

Mobile application deployment:

In the current implementation, this system is only currently available through a web interface. In order to make it more accessible and user-friendly for students to incentivise them to use it regularly, it is suggested to develop a responsive, mobile-first interface design since most users nowadays are on their phone screens more than computer screens.

9. Conclusion

This project successfully designed and implemented an LLM-based educational chatbot with an inbuilt survey collection mechanism for NTULearn that addresses both challenges in the problem statement: students' difficulty in accessing course and conceptual information and professors' struggle to collect meaningful feedback. The analysis and feedback of limited user testing suggests the effectiveness of this chatbot to function as both an auto-tutor for students and a survey collection tool for professors. In fact, professors can customise the chatbot to their needs for any modules and this prototype can also be catered for users in different industries elaborated in section 7, under project extension.

However, it is also important to consider the limitations of the implementation for a holistic evaluation of the project. The chatbot prototype was only implemented for a limited time and tested with a small group of students, which led to some constraints in the overall data collection and analysis. Additionally, the prototype was not integrated with the existing NTULearn platform which limits its real-world testing environment.

A further comprehensive study can be conducted for future iterations to fully ascertain the effectiveness of the chatbot in addressing the identified pain points. For instance, professors could introduce the chatbot during the first lecture, demonstrate its capabilities, or even integrate its usage into course activities to ensure continuous usage of the system throughout the semester.

It would be beneficial for this project to continue, with enhanced features developed for the chatbot and survey collection mechanism so that it can be deployed across multiple courses for NTULearn. With further refinement and time, this solution has the potential to create a truly continuous and sustainable cycle where improved LLM-based course-related assistance leads to more frequent student engagement, which in turn generates more comprehensive feedback for teaching enhancement.

REFERENCES

[1] R. Agarwal and M. Wadhwa, “Review of State-of-the-Art Design Techniques for Chatbots,” *SN Computer Science*, vol. 1, no. 5, Jul. 2020, doi: https://doi.org/10.1007/s42979-020-00255-3.

[2] F. Aslam, “The Impact of Artificial Intelligence on Chatbot Technology: A Study on the Current Advancements and Leading Innovations,” *European Journal of Technology*, vol. 7, no. 3, pp. 62–72, Aug. 2023, doi: https://doi.org/10.47672/ejt.1561.

[3] Marie Gobiet, “The History of Chatbots - From ELIZA to Alexa,” *Chatbots and Voice Assistants from Onlim*, Feb. 15, 2024. https://onlim.com/en/the-history-of-chatbots/

[4] Feriel Khennouche, Youssef Elmir, Yassine Himeur, Nabil Djebari, and A. Amira, “Revolutionizing generative pre-traineds: Insights and challenges in deploying ChatGPT and generative chatbots for FAQs,” *Expert Systems with Applications*, vol. 246, pp. 123224–123224, Jul. 2024, doi: https://doi.org/10.1016/j.eswa.2024.123224.

[5] J. Porter, “ChatGPT continues to be one of the fastest-growing services ever,” *The Verge*, Nov. 06, 2023. https://www.theverge.com/2023/11/6/23948386/chatgpt-active-user-count-openai-developer-conference

[6] M. Fokina, “11 Amazing Chatbots Statistics and Trends You Need to Know in 2020,” *Tidio*, Apr. 04, 2023. https://www.tidio.com/blog/chatbot-statistics/

[7] A. Lieb and T. Goel, “Student Interaction with NewtBot: An LLM-as-tutor Chatbot for Secondary Physics Education,” May 2024, doi: https://doi.org/10.1145/3613905.3647957.

[8] E. Kasneci *et al.*, “ChatGPT for good? On opportunities and challenges of large language models for education,” *Learning and Individual Differences*, vol. 103, no. 102274, Apr. 2023, doi: https://doi.org/10.1016/j.lindif.2023.102274.

[9] K. Jian, “Chatbot as a teaching assistant,” *Handle.net*, 2021, doi: https://hdl.handle.net/10356/153230.

[10] Suma Katabattuni, “Why Choose Streamlit? A Comprehensive Guide - Suma Katabattuni - Medium,” *Medium*, Aug. 20, 2024. https://medium.com/@sumakbn/why-choose-streamlit-a-comprehensive-guide-bc4779ff678c

APPENDIX:

Code Implementation can be found on:

https://github.com/john14759/fyp_ntu_chatbot

Use cases of system:

1. **Ask About Course Logistics**
    a. Actor: Student
    b. Description: Students can ask questions about course administration, scheduling, venues, assessment components, and other logistical topics related to SC1015.
    c. Example: "When is the deadline for Assignment 2?"

2. **Ask About Course Content**
    a. Actor: Student
    b. Description: Students can ask questions about Data Science and AI concepts covered in the SC1015 course (e.g., machine learning, data visualization).
    c. Example: "What is the difference between classification and regression?"

3. **Receive Context-Based Answers**
    a. Actor: Student
    b. Description: The chatbot categorizes questions (logistics vs. course content) and responds appropriately.
    c. Example: If a student asks, "What is reinforcement learning?", the chatbot recognizes it as a Data Science/AI concept.

4. **Retrieve Course Materials**
    a. Actor: Student
    b. Description: The chatbot uses Azure AI Search to pull relevant materials (e.g., lecture notes, assignments) based on the student’s query.

c. Example: The chatbot includes the explanation from the notes if there is any in its response.

5. **Provide Feedback**
    a. Actor: Student
    b. Description: The chatbot collects feedback from students through predefined questions triggered by their interactions.
    c. Example: After answering a question, the chatbot asks a question from the survey list

6. **Continue Previous Conversation**
    a. Actor: Student
    b. Description: The chatbot remembers past interactions and provides context-aware responses.
    c. Example: A student asks, "Can you explain more?"

7. **View chat log history**
    a. Actor: Course coordinator
    b. Description: The course coordinator uses the chat history tab to view the chat history between students and chatbot

8. **View visualisation of feedback**
    a. Actor: Course Coordinator
    b. Description: The course coordinator uses the visualisation tab to see charts and dashboard to visualize survey results for reporting or decision-making.

List of survey questions used:

1) This faculty member encouraged engagement in the course; as a result of the teaching approaches taken by this faculty member I was involved and interested in the course.
2) This faculty member communicated clearly: this faculty member was easy to understand in all forms of communication including in classes, online, and in writing.

3) This faculty member was approachable: this faculty member created opportunities, either in classes, or outside classes, or online, for students to ask questions and seek help.
4) This faculty member helped students understand important concepts: this faculty member explained how the course content fits together with principles and concepts.
5) This faculty member encouraged critical thinking in the subject area: the teaching approaches of this faculty member encouraged me to think deeply and analytically about the knowledge and concepts in the course.
6) This faculty member provided timely feedback that helped my progress: this faculty member gave useful comments that assisted students' learning.
7) This course takes innovative approaches to student learning.
8) The online learning resources in this course helped my learning.
9) The online component is well integrated with the course.
10) The combination of online and face-to-face teaching in this course is effective for student learning.
11) The intended learning outcomes given at the beginning of the course helped me understand what I was expected to learn.
12) This course made effective use of collaborative learning.
13) This course encouraged learning through participation.
14) The way this faculty member conducted classes increased my understanding of the course.
15) This faculty member often illustrated or explained the subject matter with real-world examples that gave more meaning to the material.
16) This faculty member gave useful information on assessment, such as clear instructions, past exam questions and how to solve them, marking criteria, or rubrics.
17) I was able to see the correlation between the assessment criteria (rubrics) and my grade. B3: This faculty member set thought-provoking assessment tasks.
18) This faculty member clearly explained the requirements of assessment tasks to enable me to understand the purpose of the task.
19) This faculty member weighted the assessment tasks relative to the amount of work required to complete the task.

20) This faculty member set assessment tasks that enabled me to demonstrate my achievement of the intended learning outcomes.
21) This course challenged me to think deeply about the concepts.
22) This course challenged my creativity.
23) This course challenged me to engage in discussion and debate.
24) This course challenged me to find my own solutions to problems.
25) This course challenged me to make decisions about different perspectives in the subject area.
26) This course challenged me to reflect on my thinking to formulate a new way of seeing things.
27) This faculty member stressed the most important parts of the course so that I could see how the material fitted together.
28) This faculty member communicated enthusiasm for the subject.
29) This faculty member used strategies to get feedback from students on their understanding before moving on to new material.
30) This faculty member conducted valuable [type of class*] *Lectures, tutorials, seminars, online.
31) This faculty member increased my interest in the course.
32) This faculty member presented the subject matter in a methodical and logical way.
33) This faculty member indicated how the topics and theories in the course are related to other course offerings in the degree programme.
34) This faculty member highlighted current research and development in the field.
35) This faculty member encouraged the development of practical skills.
36) This faculty member demonstrates enthusiasm for teaching.