**Programming Design Paradigm AE2 – Chatbox**

How to run:

In the terminal type:

Streamlit run Interface.py

**To install streamlit run:**

Pip install streamlit

**Or follow:**

<https://docs.streamlit.io/get-started/installation/anaconda-distribution>

Git Repository:

https://github.com/MatipaM/ChatBox

Introduction

The primary objective of this project was to create a simple yet robust chat application in python, seamlessly integrating an AI language model to deliver automated responses. This application not only supports real-time messaging but also demonstrates the implementation of crucial design patterns including singletons, observers, and factory methods.

The Singleton pattern ensures a single, consistent instance for managing the AI model's API configuration, enhancing efficiency and resource management. The Observer pattern allows for dynamic updates to the chat interface, ensuring that new messages are displayed in real time, thereby maintaining a smooth and interactive user experience. Meanwhile, the Factory pattern provides a flexible and scalable approach to creating various message types, accommodating the diverse needs of different conversational contexts.

The initial stage of development of the chat application was the UML class diagrams. This included sequence diagrams to depict message interactions, and activity diagrams to outline user interactions. These diagrams were instrumental in planning and visualizing the intricate components and workflows within the application.

In this report, I will delve into the problem statement, the detailed design and implementation of the chat application, the architecture illustrated through UML diagrams, and the functioning of the application. Additionally, I will discuss the user interface designed with Streamlit, the application’s current capabilities, and potential future improvements to enhance functionality and user experience.

The problem

The assessment task requires the development of a simple "Chat Application" integrated with an AI Language Model using Python. This application needs to showcase the implementation of the Singleton, Observer, and Factory design patterns. Key features include real-time messaging and AI-driven functionalities such as automated responses through an AI Language Model API. The Singleton pattern should ensure a single instance manages the AI model's API configuration and access, while the Observer pattern dynamically updates the chat interface with new messages. The Factory pattern will handle the creation of various message types, supporting flexibility and scalability. UML class diagrams must be produced to illustrate the application's structure, including user management, messaging, and AI integration. Additionally, sequence diagrams should depict the interactions for sending and receiving messages, and activity diagrams are encouraged to show user interactions and system responses

UML Diagrams

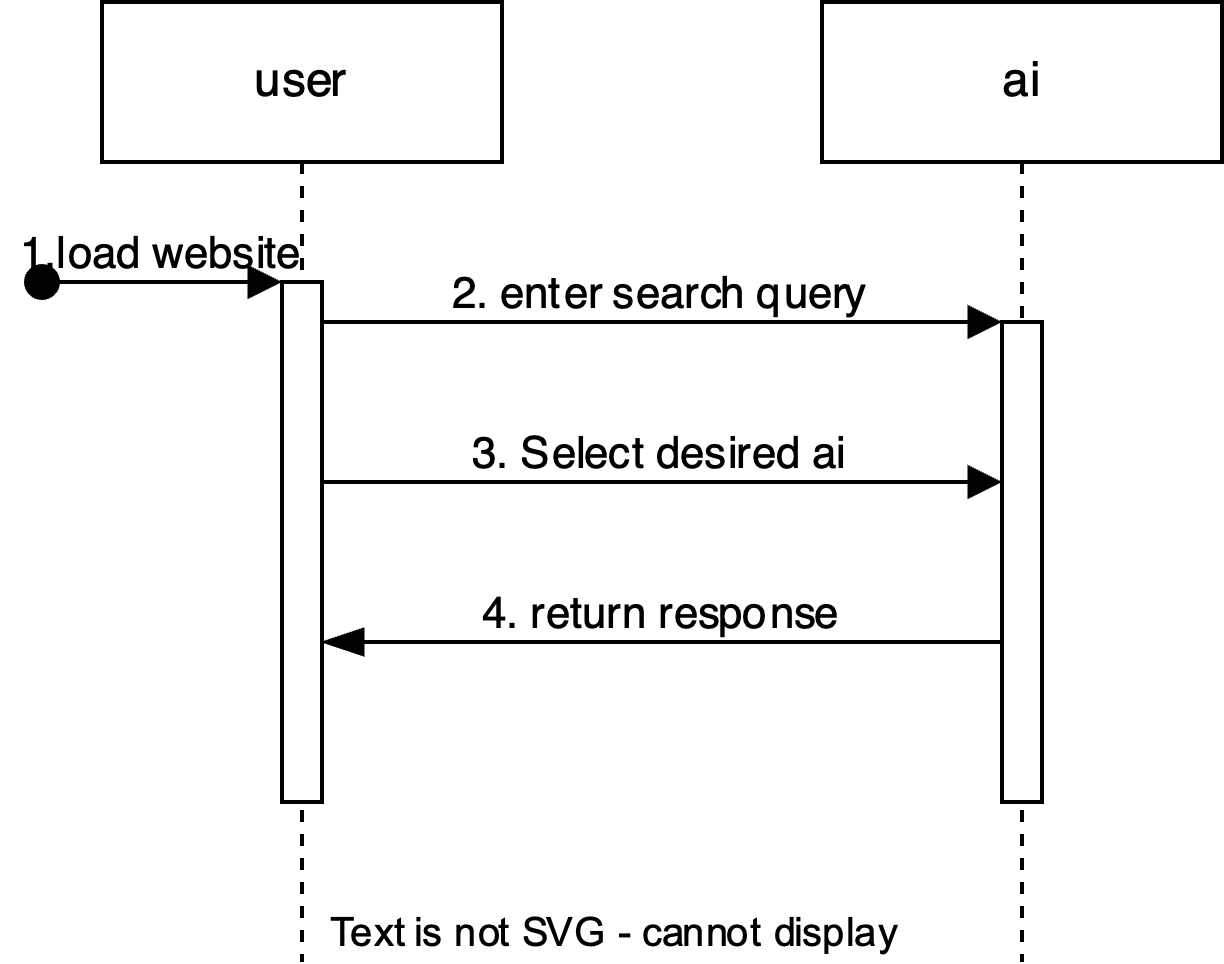


Figure 1: Sequence Diagram

The sequence diagram illustrates the interaction between a user and the AI system. It details the steps involved in processing a user query and generating an AI response. The process begins when the user loads the website (step 1), initiating the interaction with the system. Next, the user enters a search query into the input box (step 2), which is then processed by the system.

After entering the query, the user selects the desired AI type from the available options (step 3). This selection determines the nature of the response, whether it is a friendly, sophisticated, or normal response, as suggested by the earlier diagrams. Finally, the AI processes the user input and returns the appropriate response (step 4), completing the interaction cycle.

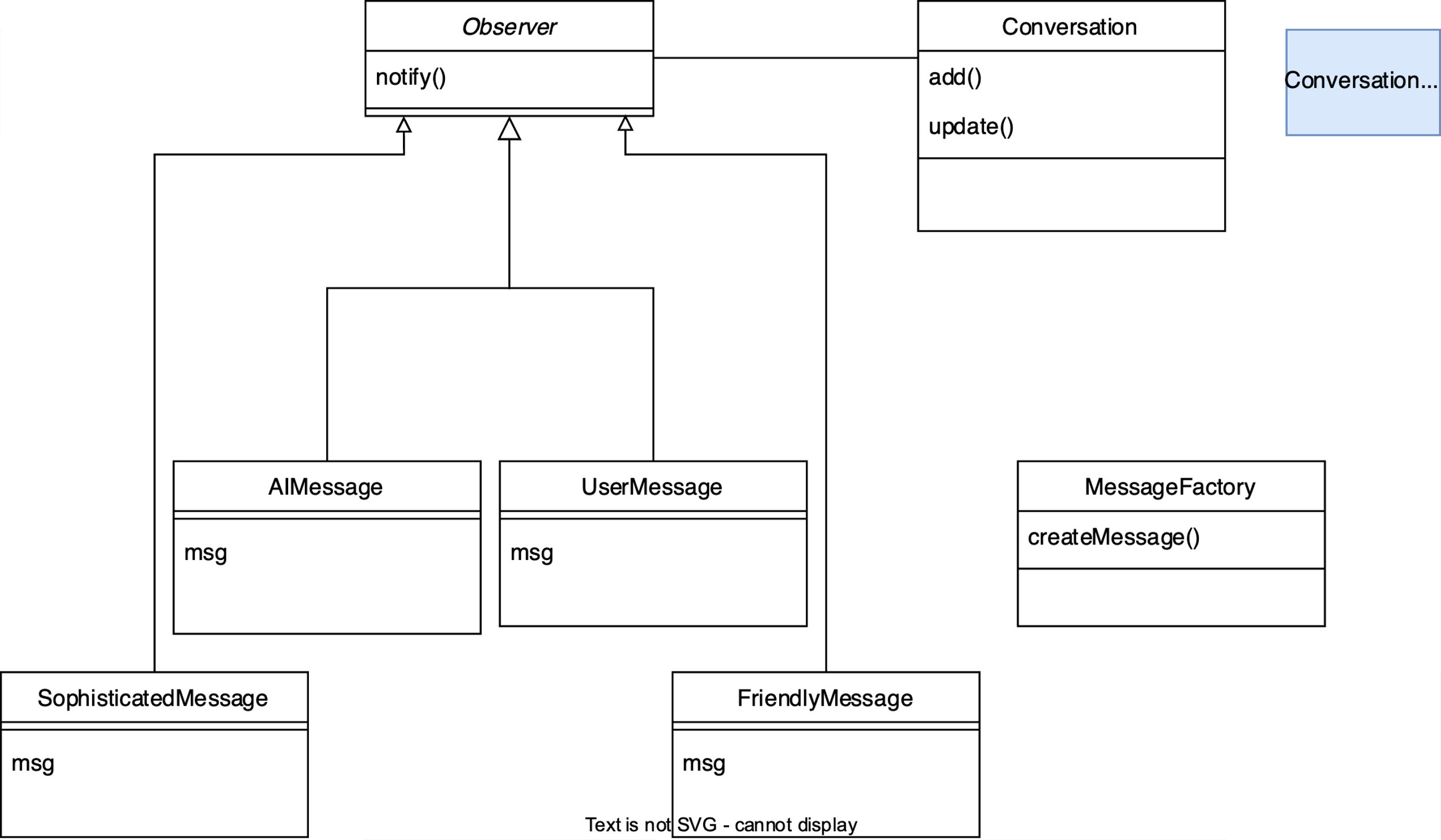


Figure 2: Class Diagram

The class diagram seeks to visualise the system’s structure. The diagram displays the classes, attributes, methods and relationships in the system. The`Conversation` class is key to the design, as it acts as the observable entity within the Observer design pattern. The ‘Conversation’ class notifies the `Observer`, which represents components that need to update dynamically when the conversation changes, ensuring real-time updates to the chat interface. In this case the Conversation class (observable) contains a list of all available observers. The list will always contain two observers. The observers are the different types of messages available in the conversation, as the conversation follows an ask and answer format. The types of messages will always include a user message and the choice of ai (responsible for the answer) which is dictated by the user. The update() method is called on the singleton observable when the user has entered their question and selected their desired AI. The update() method is responsible for telling each observer to save their data to the .json file.

The `MessageFactory` class employs the Factory design pattern to create various message types, facilitating flexibility and scalability in message management. It generates instances of `UserMessage` and `AIMessage`, which are subclasses of the broader message category. The `MessageFactory` class further branches into subclasses `SophisticatedMessage` and `FriendlyMessage`, each representing different types of AI-generated responses.

The `Conversation` class holds a list of messages and manages the overall chat flow, interacting with the `MessageFactory` to instantiate messages as needed. This structure ensures that different message types, are handled seamlessly. The relationships and interactions depicted in the diagram emphasize modularity and the use of design patterns to maintain a clean, scalable architecture for the chat application.

Overall, this class diagram illustrates system where design patterns such as Observer and Factory are utilized to manage real-time updates and message creation efficiently. The modular design supports various message types and ensures the application remains maintainable and scalable.

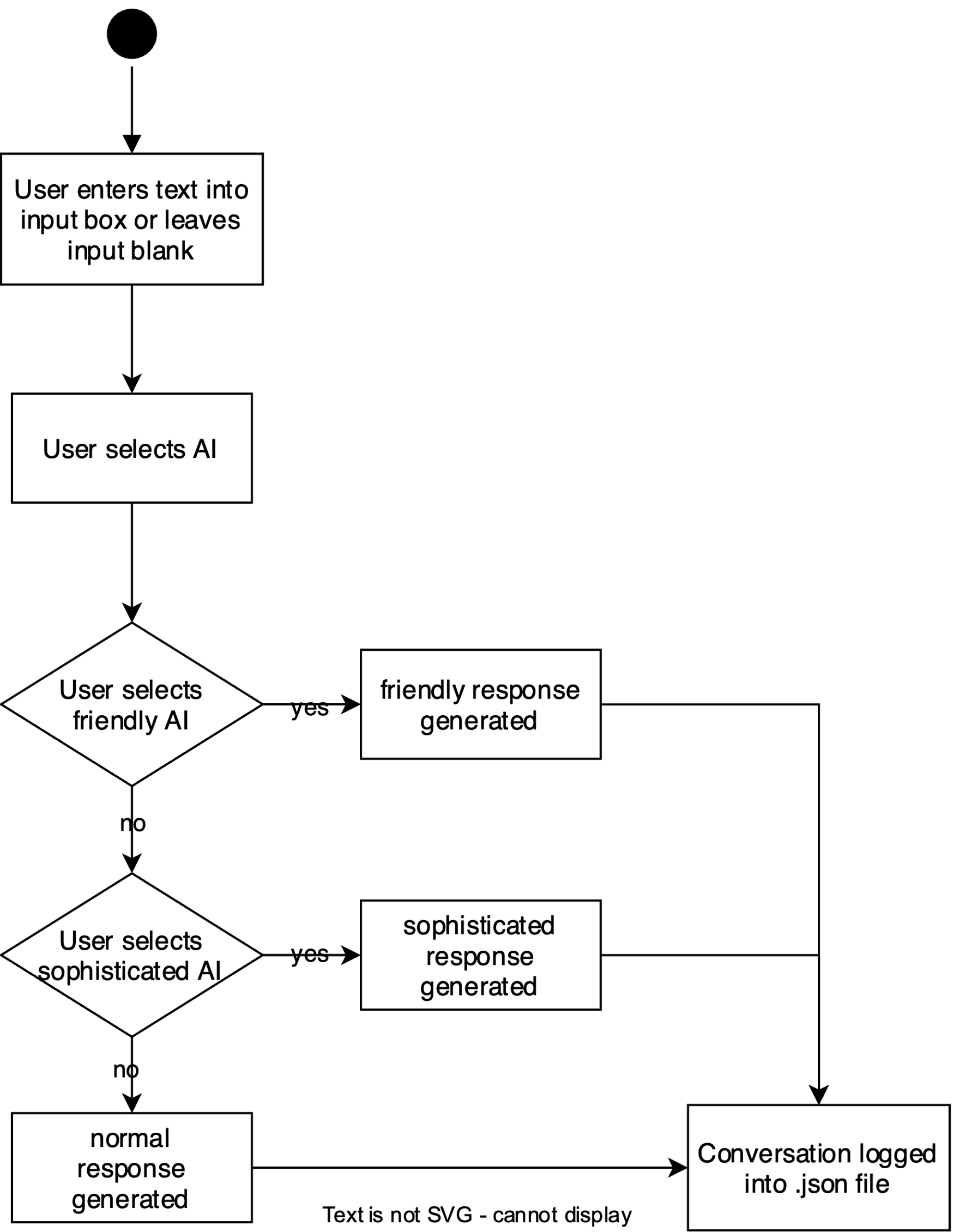


Figure 3: Activity Diagram

The activity diagram illustrates the flow of user interaction with the chat application. The process begins when the user enters text into an input box. The user has the option to either enter text or leave the input box blank. The user can then select an AI from the available options as presented on the drop down menu. The first decision point checks if the user selects the "friendly AI." If the friendly AI is selected, a friendly response is generated. If not, the flow proceeds to the next decision point, which checks if the user selects the "sophisticated AI." If the sophisticated AI is selected, a sophisticated response is generated. If neither friendly nor sophisticated AI is selected, a normal response is generated. After generating the appropriate response (friendly, sophisticated, or normal), the conversation is logged into a `.json` file.

The Factory design pattern was used to generate different types of responses based on the user’s selection. The singleton pattern is used in multiple classes to manage the configuration and access of the AI model's API, ensuring a single control point for API interactions. The singleton method is also used to ensure that only one conversation can take place at a single point throughout the user’s interaction with the application. The Observer Pattern is employed to update .json file as new messages are sent or received. This diagram aims to effectively outline the user interaction flow and response generation in the chat application, emphasizing decision-making based on AI selection and ensuring all interactions are logged for record-keeping.

How the application runs

I added extra information to the sophisticated and friendly prompts in order to ensure persistent types of replies. How responses to a prompt changes based on the different types of the ai:

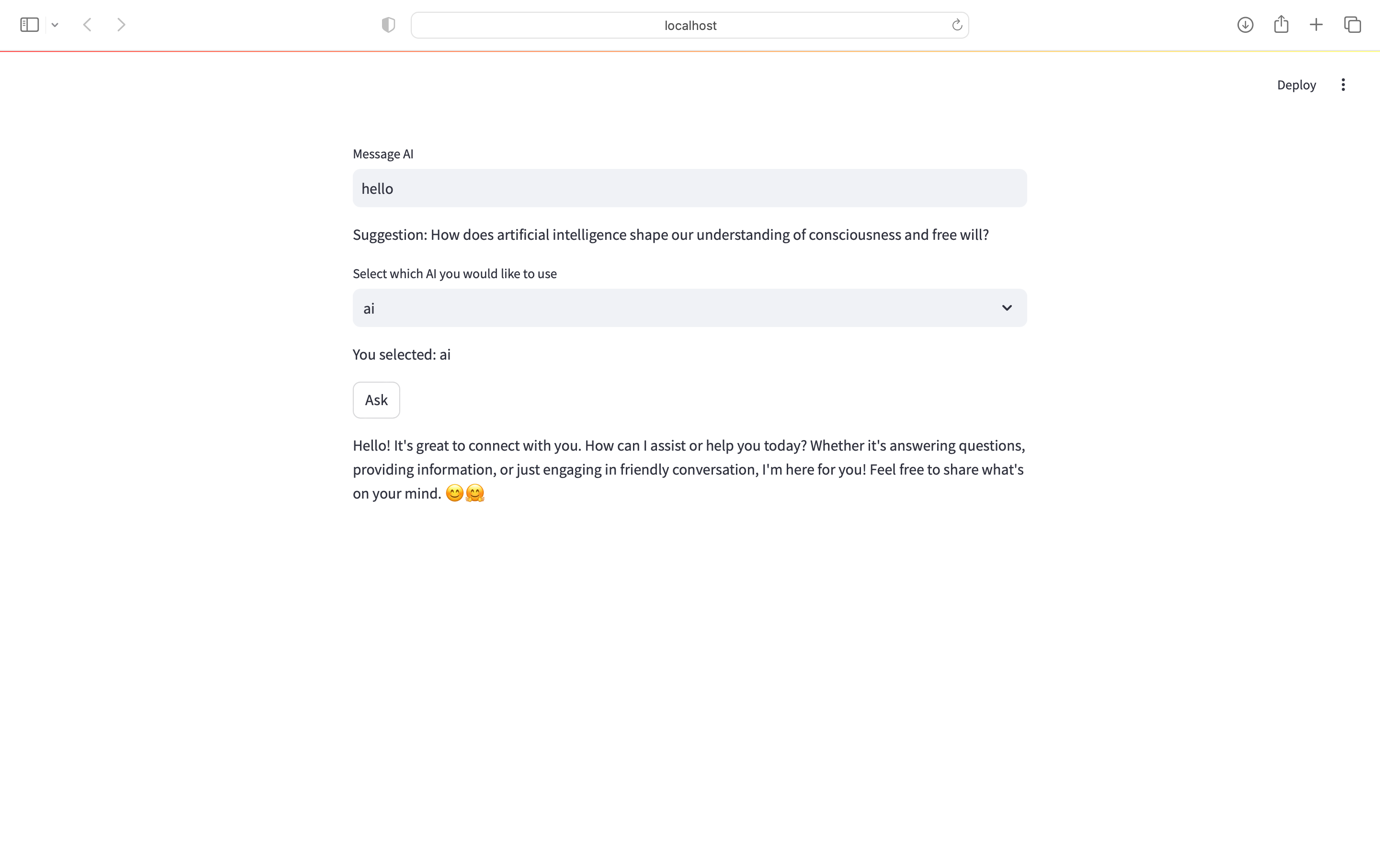


Figure 4: Normal AI

A screenshot of a chat

Description automatically generated

Figure 5: Friendly AI

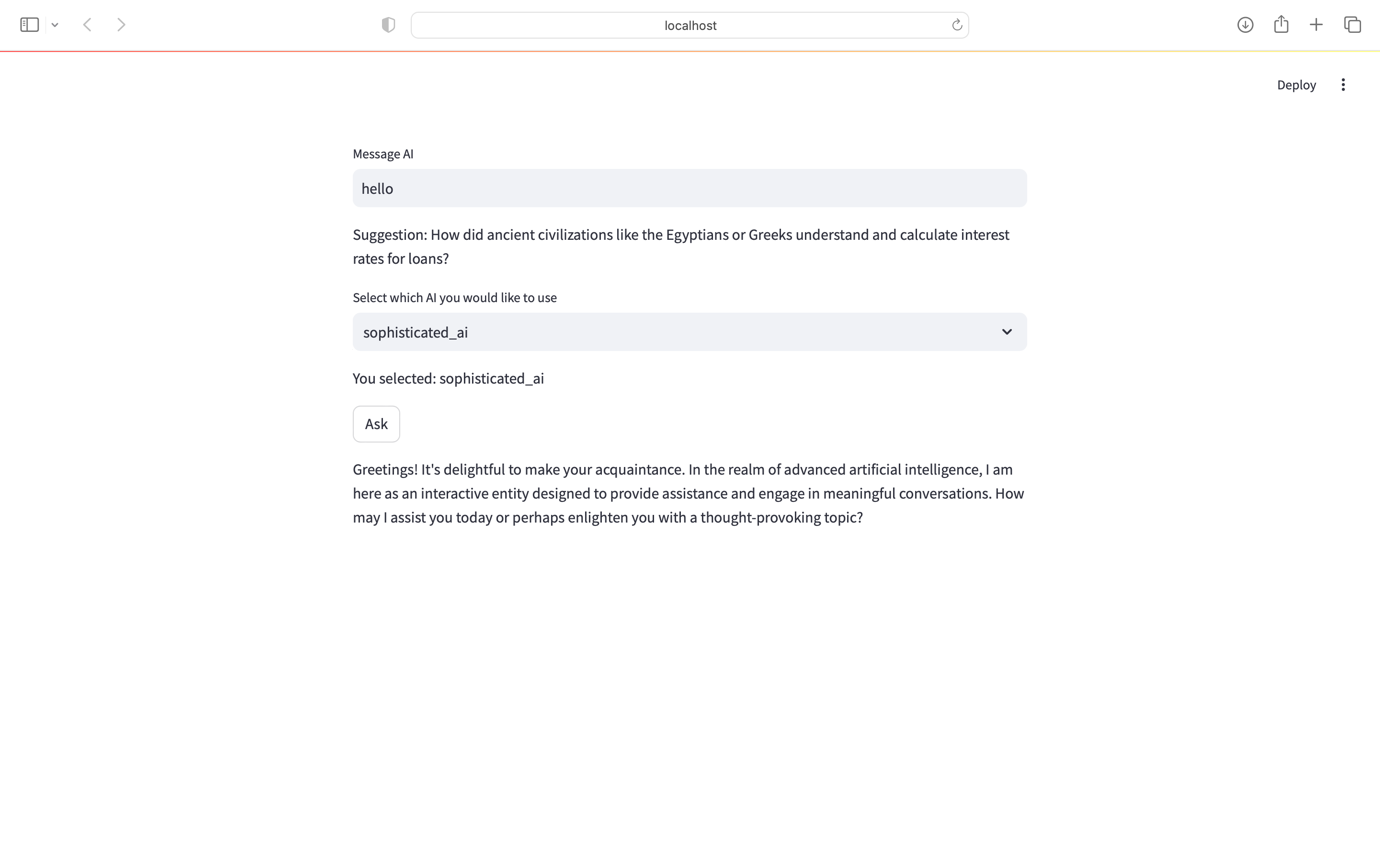


Figure 6: Sophisticated AI

The resulting responses align well with the intended personas of the different AIs, but there are some issues that occur when comparing the responses of each of the ai’s to one another. The sophisticated AI response is well-crafted, offering a formal and detailed greeting. It successfully conveys a sense of advanced capability and readiness to engage in meaningful conversation, appropriately matching the expectation for a sophisticated AI that aims to provide an elevated and intellectual interaction. The friendly AI response is warm, inviting, and supportive, featuring a positive tone and an emoji to enhance the friendly atmosphere. It effectively creates a welcoming environment, making the user feel at ease, which is ideal for users seeking a more personable and approachable interaction.

On the other hand, the normal AI response, while polite and informative, might be slightly too friendly given its intended neutral position between sophisticated and friendly AIs. Although it maintains professionalism, it could benefit from a more balanced tone that is less effusive than the friendly AI but still approachable. A slight reduction in the warmth of the language could help distinguish it more clearly from the friendly AI, ensuring it serves as a middle ground that is neither too formal nor overly friendly.

Overall, the responses are well-crafted for their respective AI personas. However, slight adjustments to the normal AI's tone could enhance the distinction between the different types, ensuring each AI response aligns perfectly with its intended character. The sophisticated AI performs exceptionally well in maintaining a formal and intellectual tone, while the friendly AI excels in creating a positive and welcoming interaction.

All of the present AI options allow support for multiple language input. The responses are. fully in the desired language.

A screenshot of a computer

Description automatically generated

Figure 7: Multi-language support

Front End

The user interface of the chat application is clean and user-friendly, designed to facilitate seamless interaction between the user and the AI. At the top of the interface, a text input field labelled "Message AI" allows the user to enter their message. Beneath this, a suggestion is displayed, providing the user with a potential topic or question to ask, such as "How does artificial intelligence shape our understanding of consciousness and free will?" Below the suggestion, there is a dropdown menu labelled "Select which AI you would like to use," allowing the user to choose the desired AI type. After selecting the AI, the user's choice is confirmed with a message stating "You selected: ai." An "Ask" button initiates the query. The AI's response is displayed below, providing a friendly and engaging message that acknowledges the user's input and encourages further interaction. The overall design is intuitive, ensuring that users can easily navigate the application and communicate with the AI effectively.

In order to develop the interface rapidly, the development of the chat application's interface was done with the help of Streamlit. Streamlit was chosen because it is simple and efficient in building interactive web applications. Streamlit allows for rapid prototyping by enabling the use of Python scripts directly to create a user-friendly interface without the need for extensive front-end coding. One of the primary advantages of using Streamlit is its ability to update the interface in real-time, making it ideal for applications requiring dynamic content such as a chat application. However, there are some disadvantages that come with the use of streamlit, such as limited customization options compared to traditional web frameworks and potential scalability issues for more complex applications. Despite these drawbacks, Streamlit’s speed and simplicity made it an excellent choice for quickly developing and deploying the chat application's interface. It allowed me to focus more on refining the core functionalities of the system as well as creating a desirable user experience.

Streamlit automatically makes sure the website is designed responsively. Ensuring that no elements are cutoff and all elements are fully visible on almost all device sizes.

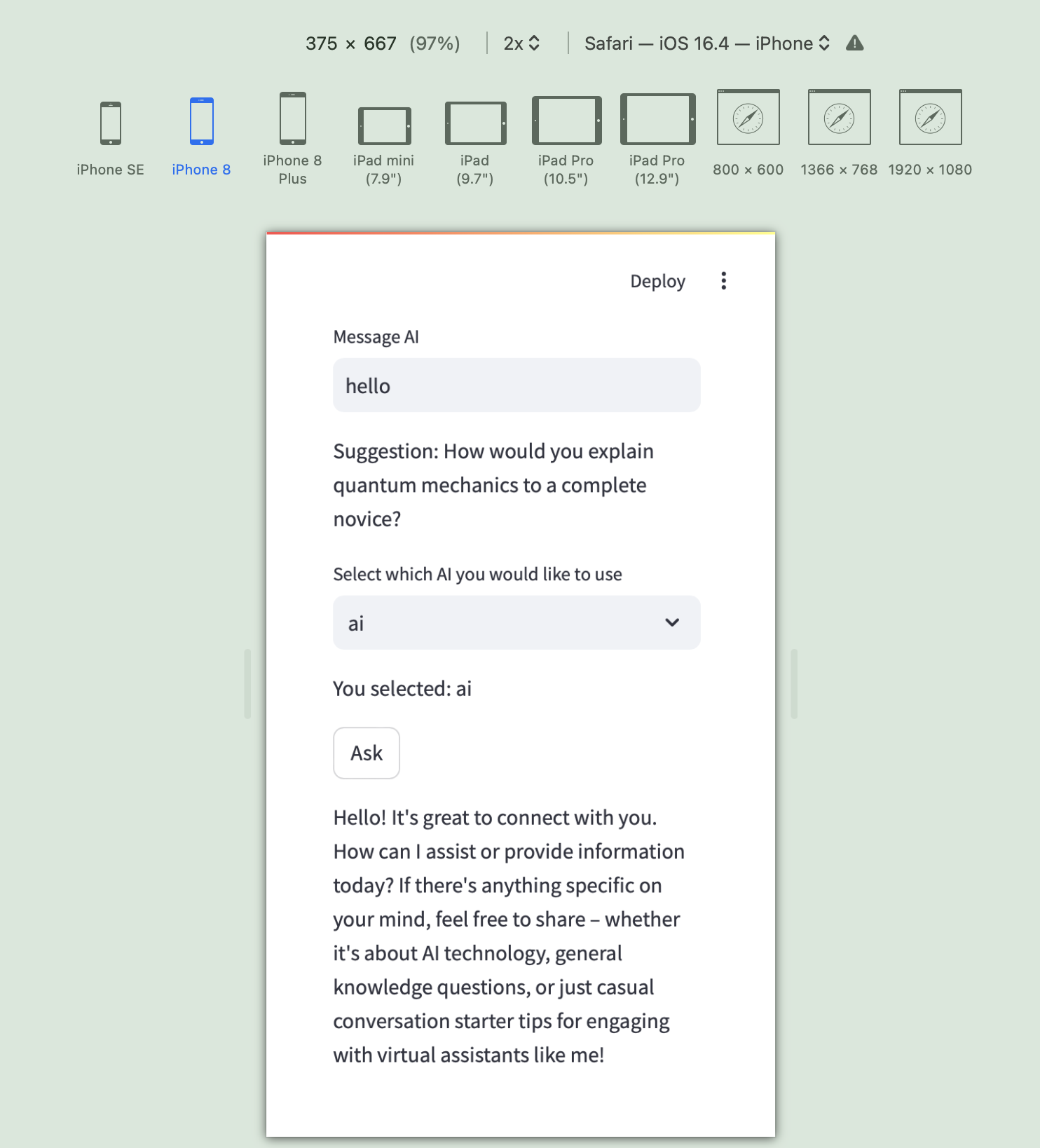


Figure 8: Mobile View

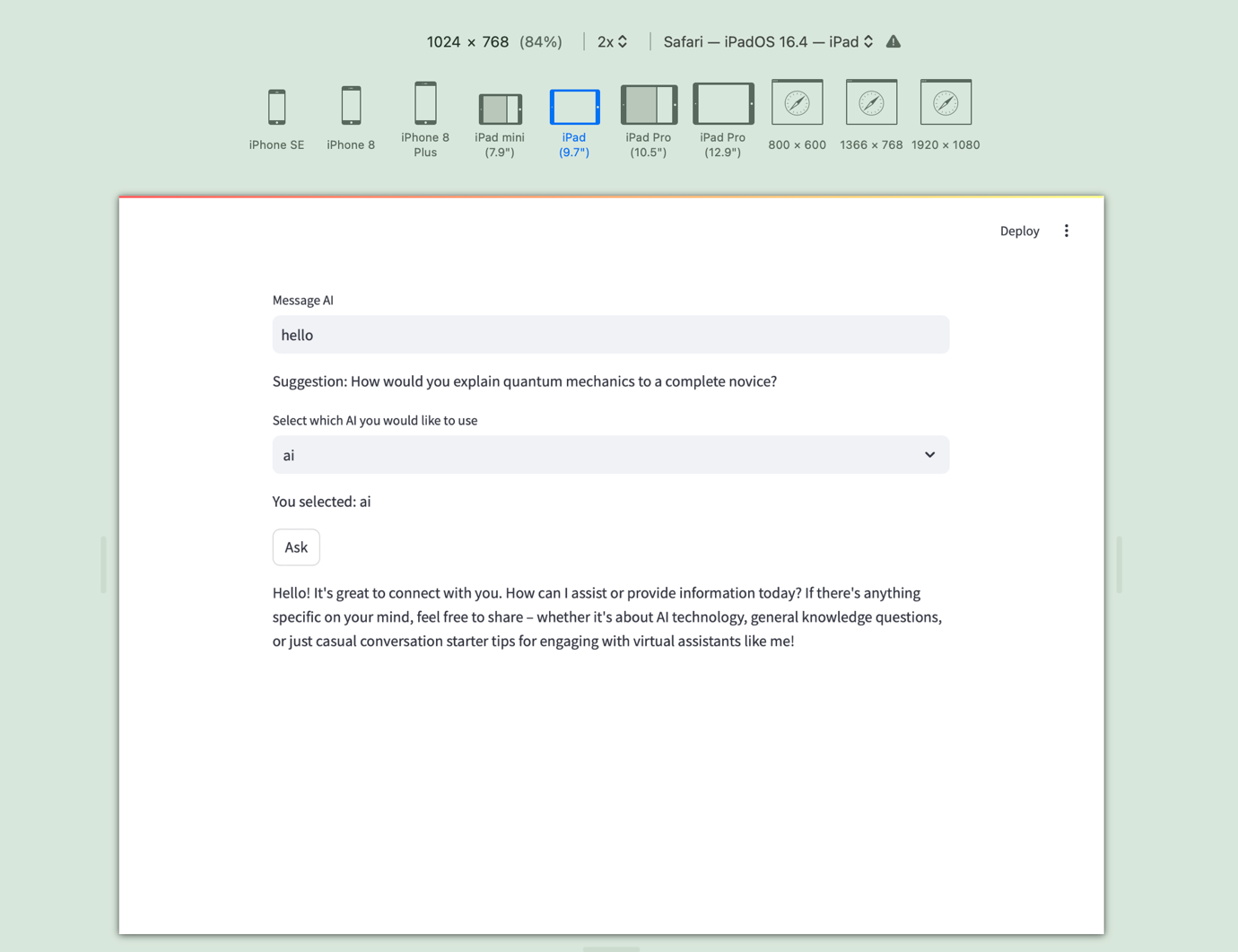


Figure 9: Ipad View

Future Improvements

Improvements to the AI chat application in its current state would include remembering users' past responses and ensuring that future AI responses take into account the context of previous interactions. This would make the conversation more coherent and contextually relevant . Additionally, the application could offer suggestions for follow-up questions based on the users' previous queries, guiding them towards deeper and more meaningful questions. These features would collectively enhance the usability and responsiveness of the chat application, making interactions more seamless and intuitive.

Another enhancement would be to implement a typing indicator instead of showing a ‘running symbol’ in the corner of the webpage, while the AI is generating a response, providing a more interactive and engaging user experience.

One future improvement for the chat application involves making better use of the saved conversations stored in the JSON file. Currently, while conversations are logged, they are not utilized beyond storage. By implementing a feature that keeps a list of past conversations, similar to how ChatGPT maintains chat histories, users would be able to continue previous conversations seamlessly. This enhancement would allow users to revisit and resume different types of conversations, enhancing the continuity and personalization of their interactions with the AI. It would also enable the AI to provide more contextually aware responses, as it could reference prior exchanges, leading to more coherent and meaningful dialogues.

Another area for improvement is addressing the Streamlit application's current behavior of reloading the entire screen when the user selects a different option. The reload interrupts the user experience, as they cannot interact with the screen while it is loading, and it presents a blank or static page during the process. By incorporating an image or animation to indicate that the application is processing, users would have a visual cue that the system is working, which can reduce frustration and enhance user engagement. Moreover, optimizing the application to minimize reload times or implementing partial updates where only the relevant part of the screen refreshes could significantly improve the overall user experience, making interactions smoother and more efficient.

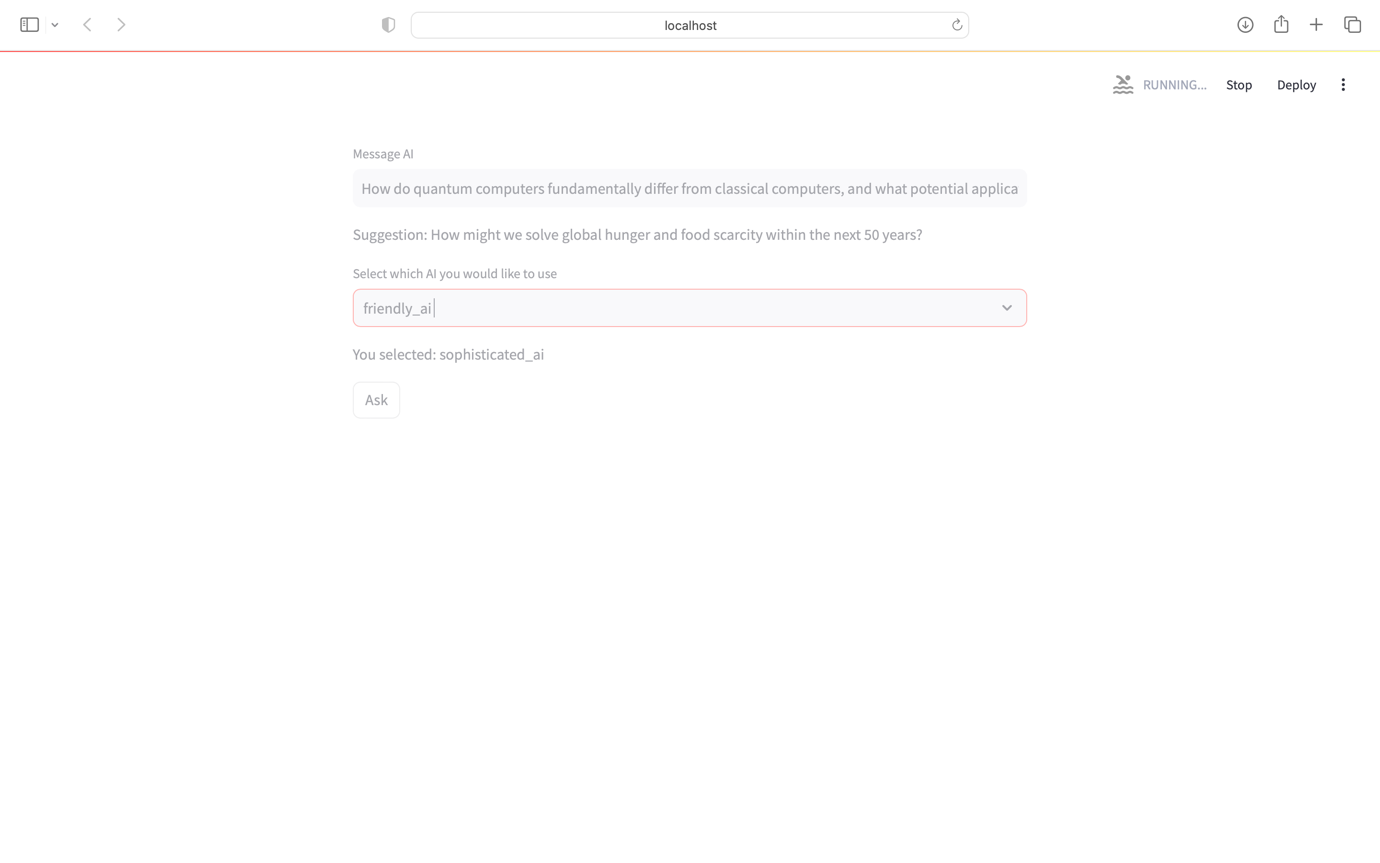


Figure 10: Loading Screen

Introducing user profiles in order to personalize responses would help to enhance the user experience. The system would function similarly to how ChatGPT sometimes prompts users to choose between two response options. By creating and saving a user profile, the machine could offer a more tailored interaction. During the initial interactions, the machine might present side-by-side options, allowing the user to select their preferred communication method. Over time, the machine would learn how the user best comprehends information and how they prefer their answers to be formatted. This adaptive learning process would enable the machine to continually update its approach, providing increasingly personalized and effective responses as it interacts with the user.

A dashboard designed to analyze user interactions, popular queries, and AI performance could be instrumental in understanding user behavior and enhancing the AI's responses. By tracking and evaluating how users interact with the system, which queries are most frequently asked, and how effectively the AI addresses these queries, developers could gain valuable insights into user preferences and pain points. This data-driven approach would enable continuous refinement of the AI, ensuring it becomes more intuitive and responsive to user needs over time. By leveraging this analysis, the AI could be fine-tuned to deliver more accurate, relevant, and personalized responses, thereby improving the overall user experience.

Allowing users to customize the appearance and behaviour of the chat interface could significantly enhance the user experience. By offering options for themes, font sizes, and layout adjustments, users could tailor the interface to their personal preferences and needs. This level of customization would not only make the chat more visually appealing but also improve usability by accommodating different accessibility requirements. Such personalization ensures that the interface is comfortable and intuitive for each user, ultimately leading to a more engaging and satisfying interaction with the AI.

Integrating voice recognition to allow users to input queries via speech would make the application significantly more accessible and convenient. This feature would enable users to interact with the AI hands-free, which is especially beneficial for those who may have difficulty typing or who are multitasking. Voice input could streamline the user experience, making it faster and more natural to ask questions and receive responses. By accommodating a broader range of user needs and preferences, voice recognition technology would enhance the inclusivity and usability of the application, making it more versatile and user-friendly.

Conclusion

The development of the AI Chatbox application exemplifies a robust implementation of key programming design patterns. These design patterns include the singleton, observer, and factory patterns, highlighting their utility in creating a modular, scalable, and efficient system. By ensuring a single instance for API management with the Singleton pattern, dynamically updating the json files with the Observer pattern, and facilitating flexible message creation with the Factory pattern, the application successfully integrates real-time messaging and AI-driven functionalities.

The UML diagrams effectively illustrates the structural and interactional aspects of the application, providing clarity on user management, messaging flow, and AI integration. The sequence, class, and activity diagrams demonstrate the systematic approach taken to manage user interactions, message processing, and the overall architecture.

The front-end, developed using Streamlit, achieves a clean and user-friendly interface, enabling rapid prototyping and real-time updates. Despite some limitations in customization and scalability, Streamlit's simplicity allowed a focus on core functionalities and user experience. Future improvements such as enhanced context-awareness, a more interactive loading indicator, user profiles, a detailed analytics dashboard, interface customization, and voice recognition will further elevate the application's functionality and user engagement. In conclusion, the application demonstrates the effective use of design patterns and a thoughtful approach to user interface design, ensuring a maintainable, scalable, and user-centric solution..