# OHSL Chat Bot

## Studied About LLM Based Search Engines and Chat Bots

Finding out how large language models improve user interactions is a key component of researching LLM-based search engines and chatbots, especially ones that leverage models like LLaMA and frameworks like LangChain. These models may provide logical and contextually appropriate language since they have been trained on large datasets, which enhances the effectiveness and precision of search results and conversational answers.

A framework called LangChain makes it easier to integrate LLMs into a variety of applications, which expedites the process of creating effective AI solutions. Conversely, the LLaMA model is engineered to yield superior performance with less computing resource use.

By improving their comprehension of user queries, taking context into account, and offering more accurate results, these technologies greatly expand the potential of search engines. By preserving information over several exchanges and providing tailored replies, chatbots provide more seamless and captivating discussions.

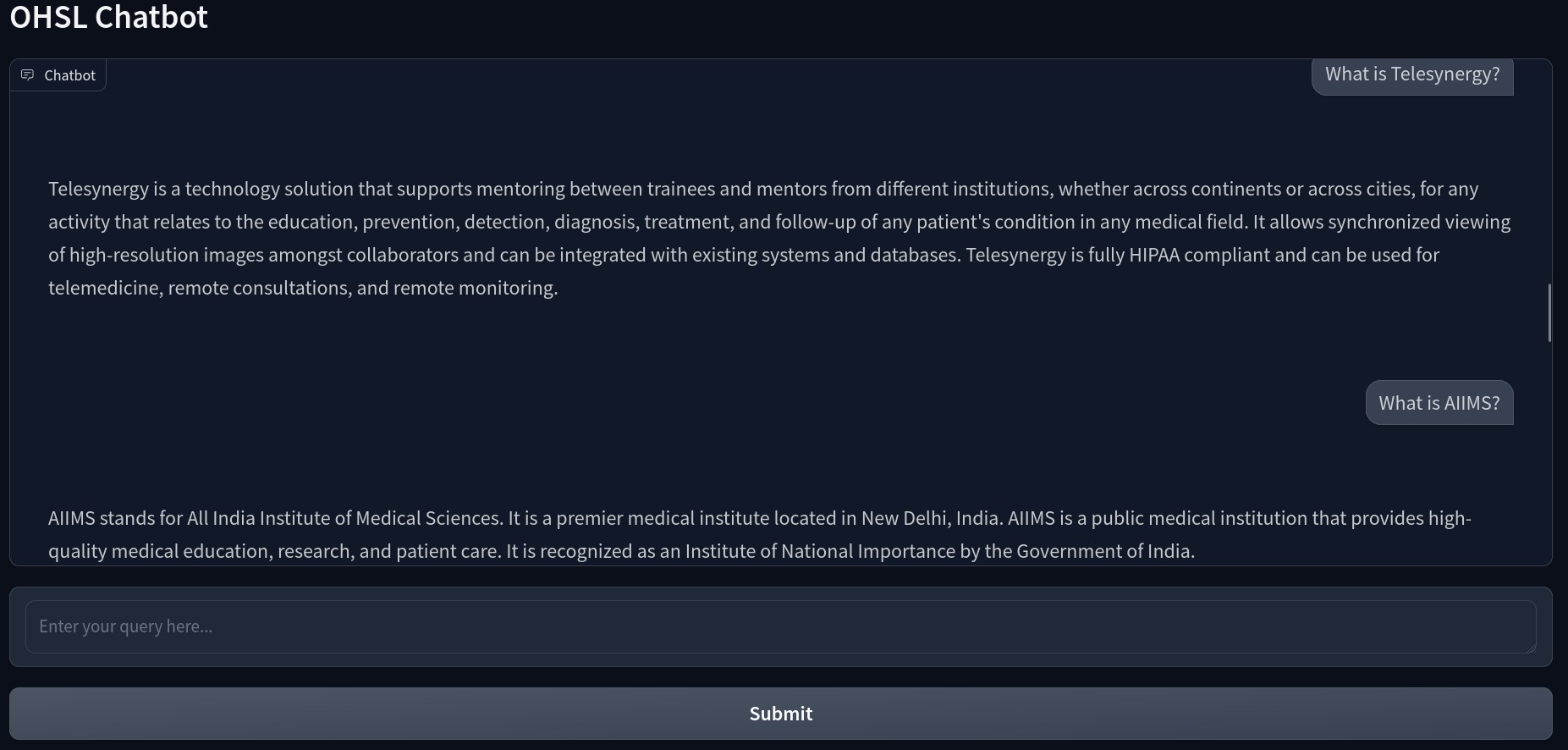


Fig: output being shown for the query asked

Research papers, online courses, and tech blogs are all resources to examine while studying these technologies since they offer insights into their creation and real-world uses. Important lessons learned include how LLMs may revolutionise information access, how they can completely change human-computer interaction, and how crucial it is to acknowledge constraints like biases and resource needs. All things considered, this research emphasises the developments in artificial intelligence and their consequences for information retrieval and communication in the future.

## Scraping the OHSL Website

The Python script is designed to efficiently scrape websites, download images, and extract textual content, organising the data in a structured manner. It leverages libraries for file management, making HTTP requests, parsing HTML, and handling URLs. The overall goal is to store data in a text file and save images in a designated folder.

The process begins with setting up a folder for downloaded images and a file for extracted text. It also maintains a list of visited URLs to prevent revisiting the same pages, ensuring efficient and non-redundant scraping. The script prompts the user to input a website URL, validating its format to ensure it begins with http:// or https://. This step ensures that the base URL is correctly identified, which is crucial for resolving relative links during the scraping process.

Once the URL is validated, the script initiates the scraping process. It first makes an HTTP request to the specified page and checks for a successful response. If successful, the page content is parsed using BeautifulSoup, a library that helps navigate and search through the HTML structure. The script specifically looks for key sections within the HTML, such as headers, paragraphs, lists, images, and tables. These elements are extracted and formatted into a markdown-like structure in the text file, providing a clean and organised output.

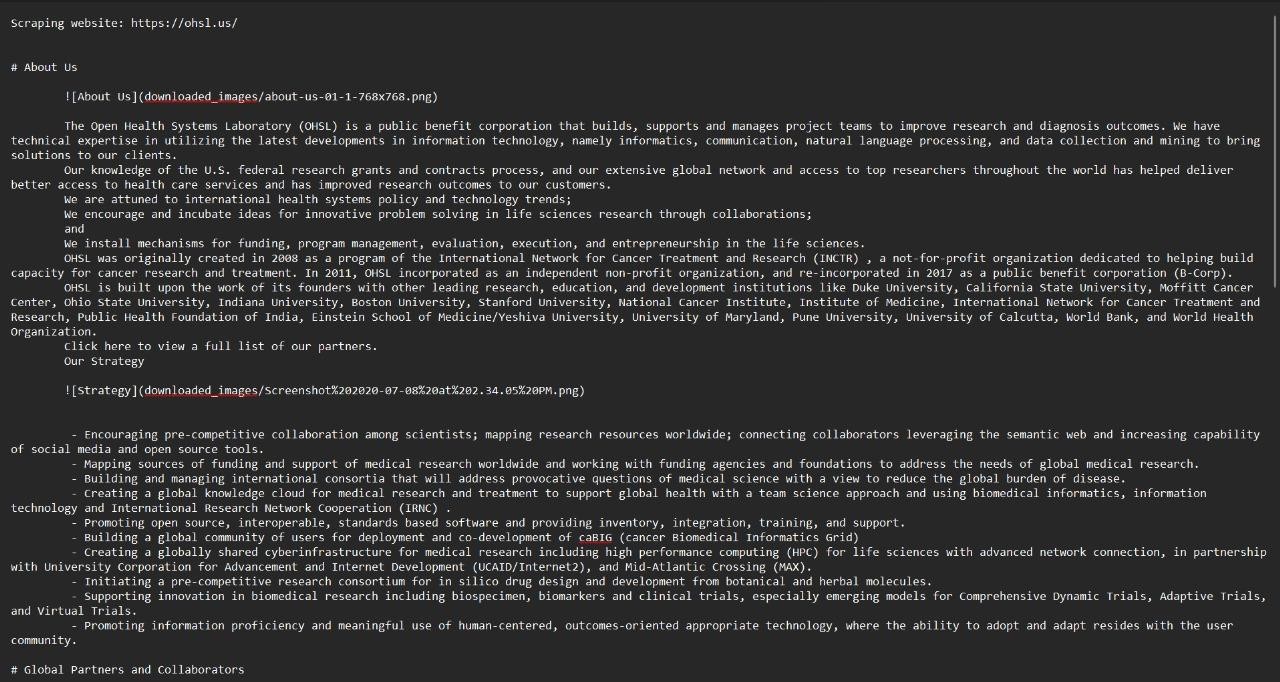


Fig: Scraped Data of OHSL Site

For images, the script downloads them from their respective URLs, ensuring that they are saved in the designated folder. The paths to these images are included in the text file, maintaining a connection between the text and visual content. This integration of images enhances the comprehensiveness of the extracted data, making the output more informative.

The script also processes links found on the page. It follows these links and scrapes their content recursively, provided they haven’t been visited before. This recursive scraping ensures that all relevant pages linked from the main URL are also explored and extracted, contributing to a thorough data collection process.

Throughout the execution, the script handles various exceptions and errors, such as failed HTTP requests or inaccessible images, by printing relevant error messages. This makes the script robust and capable of handling a range of potential issues during web scraping.

Python script serves as a powerful tool for web scraping, effectively gathering and organising both text and images from websites. It provides a structured output, integrating text and visual elements in a user-friendly manner. By ensuring that URLs are correctly processed and managing visited links, the script offers a comprehensive approach to data extraction, suitable for a variety of applications in data analysis, research, and content curation.

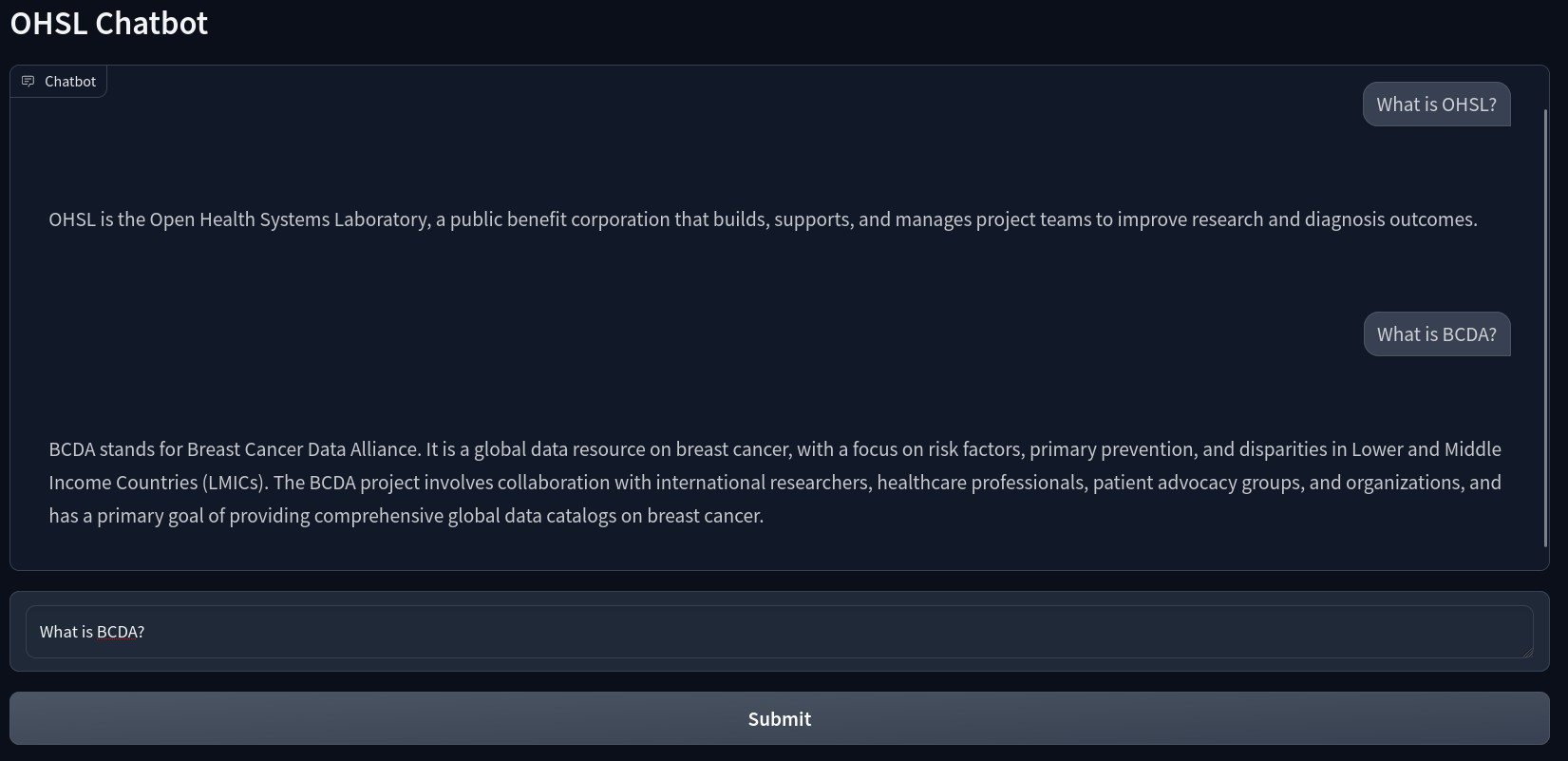


Fig: output for OHSL and BCDA

## Reading and Loading the Scraped Data

After getting the scraped data via BeautifulSoup4 we have to load this data for the model and feed this to the model that will be loaded. Text file will be loaded which has the scraped data and this will be used for all the further query-resolution tasks. The code that we developed extracted the data and stored it into a text file. The LLM model which takes in the data is supposed to be in document or PDF format. The file was then converted to a document format via an inbuilt component of the LlamaIndex data framework. This allows the model to easily access the relevant data without accessing any garbage data, unwanted ascii characters and any unnecessary terminating symbols. The function to read and load the data is combined into one. After this, the custom system prompt is defined which allows the model to understand how it should handle every query and then generate the response.

## Defining the LLM Model

The model used is the **daryl149/llama-2-7b-chat-hf** scaled variant of the original model which was Meta/llama-2-7b-chat-hf. This model was chosen after trying around 14 models out of which a few were unable to produce the intended results, a few others had their Hugging Face Inference API disabled by the creators and loading the model directly was not feasible as they were quite large. The models which we used to build the chatbot were: **OpenAssistant/oasst-sft-1-pythia-12b** | **facebook/bart-large-cnn** | **openai-community/gpt2** | **google-t5/t5-small** | **google-t5/t5-base** | **google/flan-t5-large** | **distilbert-distilgpt2** | **tiiuae/falcon-7b-instruct** | **meta-llama/Llama-2-7b-hf** |

**TheBloke/Llama-2-7B-Chat-GGUF** | **TheBloke/Llama-2-7B-Chat-GGML** | **google/flan-t5-base** | **mistralai/Mistral-7B-v0.1** | **daryl149/llama-2-7b-chat-hf.** Loading the aforementioned model was possible because of its relatively smaller size as compared to others. Besides the configuration and the code format used for the present chat bot, we tried other methods of developing a chatbot including using chainlit, streamlit, separate scripts and end-to-end approaches. All of them could not be integrated into our current project and were failing in one way or the other and thus we decided to stick to the defined code configuration and model. This model was capable of clearly understanding the service context and the data that was provided to it. This helped the model to accurately provide resolution to the mentioned queries.

## Converting the Scraped Data Into Vector Embeddings

Initialising the Embedding Model: We begin by initialising our embedding model using the LangchainEmbedding class from Langchain. Here, we utilise the "sentence-transformers/all-MiniLM-L6-v2" model from HuggingFace, which is known for its efficiency and accuracy in generating sentence embeddings.

Creating the Service Context: Next, we set up our service context, which includes configurations such as the chunk size and the language model (LLM) to be used. Here, we specify a reduced chunk size of 512 to ensure that our documents are divided into manageable pieces for embedding generation. We also integrate the previously initialised embedding model into this context.

Building the Vector Store Index: With our service context ready, we can now construct the vector store index from our documents. This step involves converting each document into vector embeddings using the service context we defined. The VectorStoreIndex class handles the creation of this index, which allows for efficient querying based on vector similarities.

Setting Up the Query Engine: Finally, we transform our vector store index into a query engine. This query engine leverages the vector embeddings to perform semantic searches, enabling us to retrieve relevant information from the indexed documents based on the similarity of their embeddings to the query embeddings.

## Gradio Interface

First, we define a function that handles chat requests and returns responses. This function accepts a user query and an optional history of past encounters. It runs the query through the query engine, which searches the vector index for relevant responses based on the embeddings. The inquiry and response are then added to the interaction history, providing a record of the conversation. The function eventually returns the updated history.

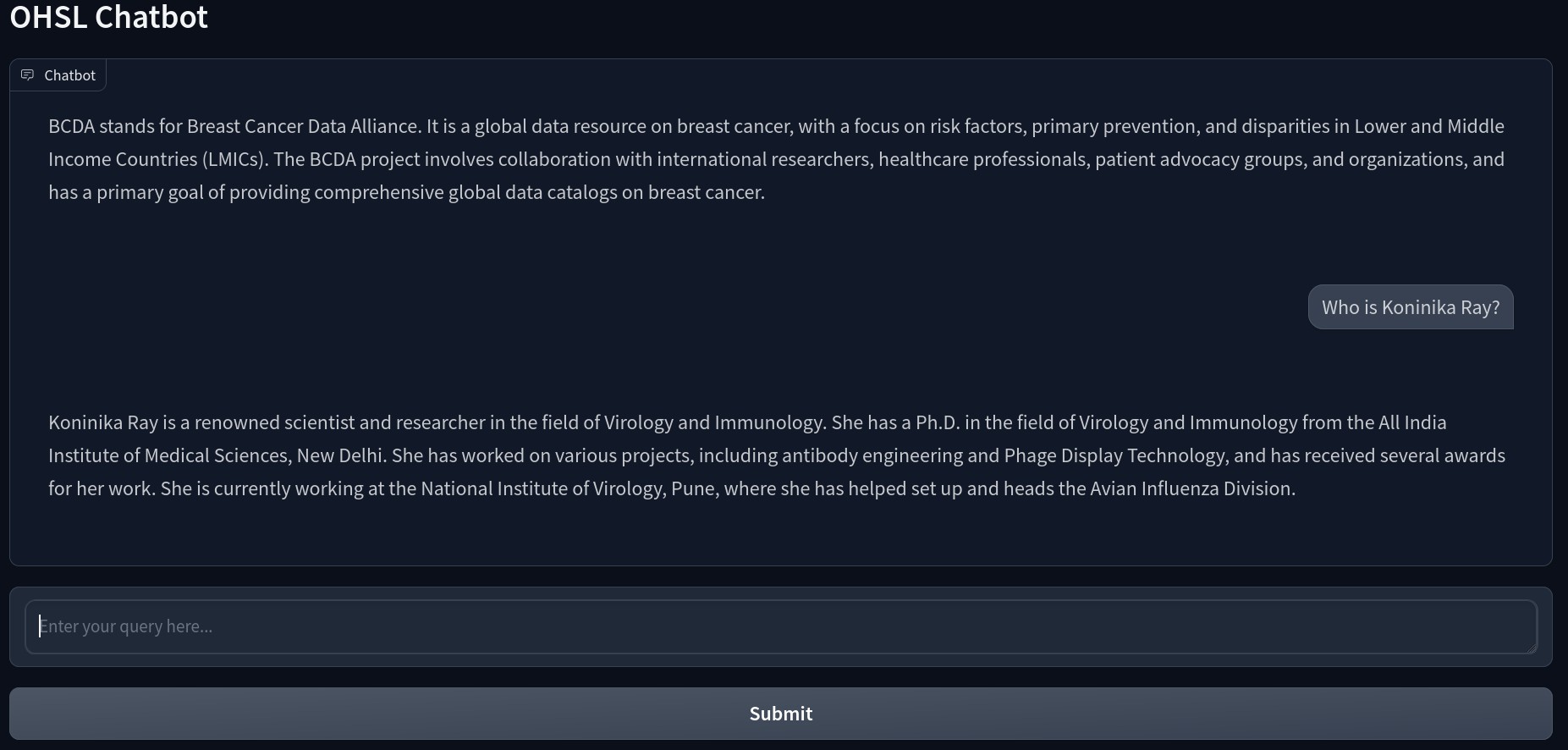


Fig: output generated for the queries asked

Next, we use the Gradio library to design a user-friendly interface for the chatbot. Gradio offers an easy solution to create web-based interfaces for machine learning models. We style the Radio interface to fit the entire height of the browser window, which improves the user experience. The interface begins with a Markdown title that displays the chatbot's name.

We set up the chatbot's key components in the Gradio interface. This comprises a chat window that displays the conversation, a text box where users may enter their questions, and a submit button to send the questions. The chat window will display the ongoing interaction between the user and the chatbot, giving users a smooth experience when communicating with the bot in real time.

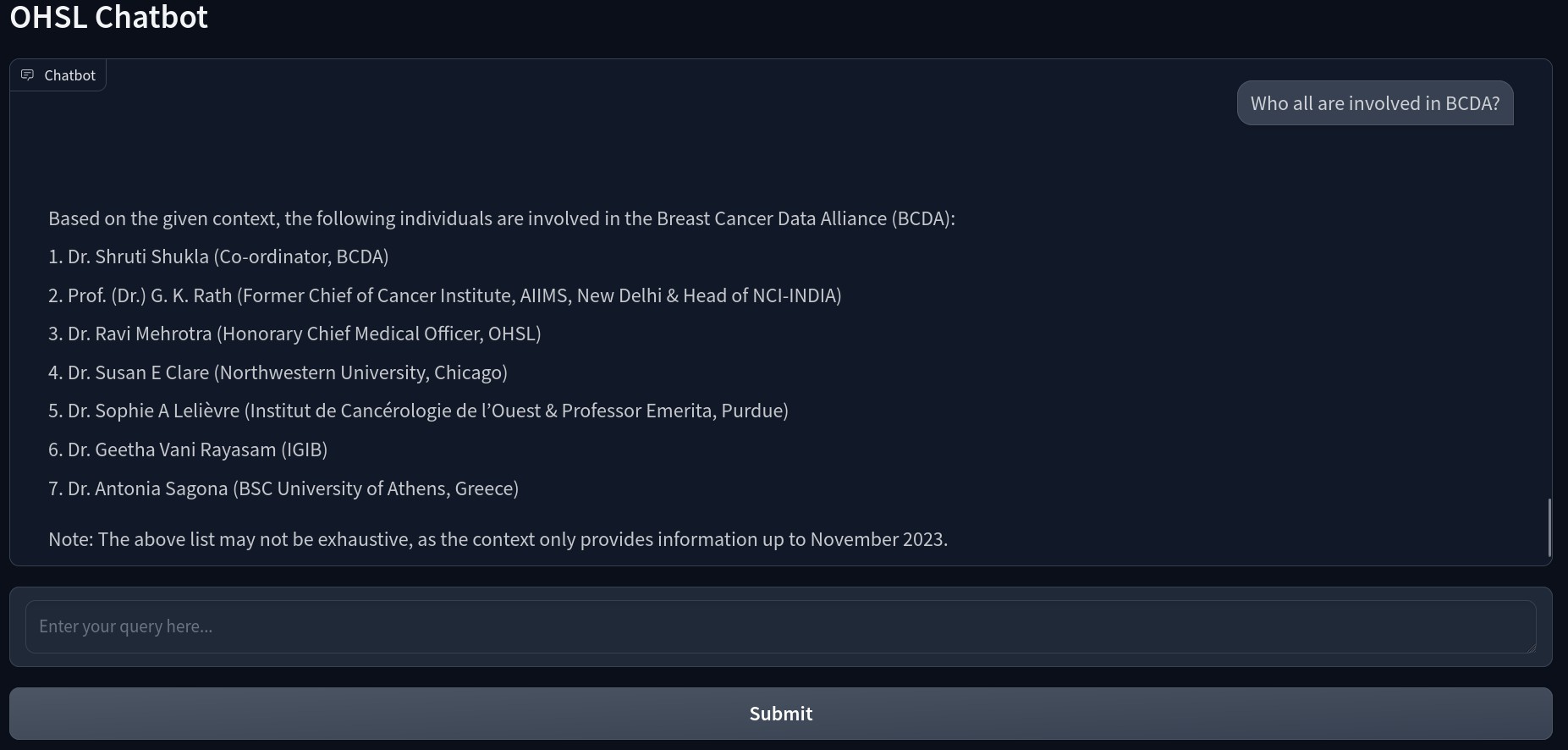


Fig: name of the members involved in BCDA

Finally, we define what happens when the submit button is clicked. This entails linking the submit button to a function that handles the user's inquiry and changes the chat history. When a user sends a query, the function evaluates the input, adds the new query and response to the discussion history, and displays the updated history in the chat window. The Gradio interface is then launched, allowing users to access and engage with the chatbot using a shareable web link.

## Learnings From the Project

Creating the chatbot using LLM-based search engines and frameworks like LangChain and LLaMA was a fascinating project. We learned how powerful large language models can improve user interactions by providing logical and contextually accurate responses. The process involved studying various resources to understand how these models work and their practical applications. We also developed a Python script for web scraping, which efficiently gathered and organised data from the OHSL website. Converting this scraped data into vector embeddings was crucial for enabling semantic searches and precise query responses. Finally, integrating the chatbot with a user-friendly Gradio interface allowed for seamless user interactions. This project highlighted the importance of effective data handling, model integration, and user interface design in creating a robust AI solution