

# RAG Application Using Knowledge Graph and Vector Search

## Final Report

HackerEarth



**Mentor:**

Parteek Kumar

**MECA Dynamics**



Molly Iverson, Ethan Villalovoz, Chandler Juego, Adam Shtrikman

CptS 423 Software Design Project II

Spring 2025

## TABLE OF CONTENTS

<b>I.</b>	<b>Introduction</b>	<b>3</b>
<b>II.</b>	<b>Team Members &amp; Bios</b>	<b>4</b>
<b>III.</b>	<b>Project Requirements Specification</b>	<b>5</b>
III.1.	Project Stakeholders	5
III.2.	Use Cases	5
III.3.	Functional Requirements	7
III.4.	Non-Functional Requirements	10
<b>IV.</b>	<b>Software Design - From Solution Approach</b>	<b>11</b>
IV.1.	Architecture Design	11
IV.2.	Data Design	18
IV.3.	User Interface Design	18
<b>V.</b>	<b>Test Case Specifications and Results</b>	<b>20</b>
V.1.	Testing Overview	20
V.2.	Environment Requirements	21
V.3.	Test Results	22
<b>VI.</b>	<b>Projects and Tools Used</b>	<b>24</b>
<b>VII.</b>	<b>Description of Final Prototype</b>	<b>25</b>
VII.1.	System Overview	25
VII.2.	User Guide	26
VII.3.	Major Use Cases	26
VII.4.	Prototype Screenshots	26
<b>VIII.</b>	<b>Product Delivery Status</b>	<b>29</b>
<b>IX.</b>	<b>Conclusions and Future Work</b>	<b>29</b>
IX.1.	Limitations and Recommendations	29
IX.2.	Future Work	29
<b>X.</b>	<b>Acknowledgements</b>	<b>30</b>
<b>XI.</b>	<b>Glossary</b>	<b>30</b>
<b>XII.</b>	<b>References</b>	<b>31</b>
<b>XIII.</b>	<b>Appendix A – Team Information</b>	<b>32</b>
<b>XIV.</b>	<b>Appendix B - Example Testing Strategy Reporting</b>	<b>33</b>
<b>XV.</b>	<b>Appendix C - Project Management</b>	<b>35</b>

### I. Introduction

In recent years, there has been an explosion of interest in large language models (LLMs) and their application in natural language processing (NLP) across industries, from customer service chatbots to research assistants. One of the most compelling applications of LLMs is in Retrieval-Augmented Generation (RAG), where models retrieve relevant information from a dataset and generate contextually accurate responses. This combination of retrieval and generation has made RAG systems an essential tool for question-answering and content generation. However, while RAG models using vector search have shown success, they are often limited by the unstructured nature of the data they rely on.

This project addresses this limitation by incorporating knowledge graphs into the retrieval process, a new approach that promises to revolutionize RAG systems. Knowledge graphs provide structured, contextual information that helps the RAG system understand and retrieve more accurate, fact-based responses. By using vector search with knowledge graphs, this project aims to build a next-generation RAG application that uses both unstructured and structured data.

The real strength of RAG systems is their ability to work with a wide range of datasets, going beyond publicly available information like Wikipedia. RAG applications work effectively with private or domain-specific data and only require changing the data being fetched. This makes AI more customizable and personal for users across various fields. For example, companies can create AI chatbots so employees can ask questions about private company information, increasing productivity and reducing reliance on IT and HR. In healthcare, AI can help doctors make predictions about diagnoses by supplying information about thousands of diseases, symptoms, and methods of recovery. To show the power of customized RAG systems, we created a custom dataset of class notes and combined it with Wikipedia data. By merging these resources, users can easily ask questions about their class materials without having to dig through their notes, demonstrating how personalized datasets can improve accuracy and make information more accessible.

This project uses a large dataset of 10,000 Wikipedia articles to create embeddings for vector search and the Wikipedia knowledge graph DBpedia.<sup>1</sup> The system will handle user questions by retrieving semantically relevant information from the knowledge graph and vector search, enabling the generation of more precise and contextually appropriate responses. Knowledge graphs can revolutionize the retrieval component of RAG applications and thus improve the overall quality and usefulness of the generated responses across several use cases, ranging from conversational agents to research tools.

By the end of the year, we successfully created a RAG chatbox application that integrates vector search, knowledge graphs, and a custom dataset into the RAG pipeline. The application is deployed using Docker. Users can pull the Docker images from our GitHub, add an OpenAI API key, and run the app themselves. This project has been a great opportunity to strengthen our AI skills and contribute to revolutionary technology like RAG.

Our client, HackerEarth, is a software company based in San Francisco that offers tools for technical hiring, including skill assessments, remote video interviews, and hackathons. The HackerEarth CEO, Vikas Aditya ([vikas.aditya@hackerearth.com](mailto:vikas.aditya@hackerearth.com)), is our client for this project.

## II. Team Members & Bios

Molly Iverson is a senior at Washington State University, graduating in Spring 2025 with a Bachelor of Science in Computer Science and minors in Mathematics and Software Engineering. She will join Microsoft as a Software Engineer after graduation. She is interested in back-end development and has experience from two internships at Expedia Group and Red Lens Games. Molly is skilled in Python, C#, and Java, among other technologies. She serves as the team leader for the RAG application project, where she plans project milestones and manages sprint tasks. Molly focuses on writing and integrating knowledge graph code into the RAG pipeline and creating a CI/CD pipeline to deploy the app using Docker.

Ethan Villalovoz is a Washington State University senior pursuing a Bachelor of Science in Computer Science with a minor in Mathematics and graduating in Spring 2025. He focuses on implementing text embeddings with the SentenceTransformer model for vector search for the RAG application project, implementing vector search, and contributing to system documentation. With experience in robotics, machine learning, and AI, Ethan is passionate about advancing human-AI collaboration and plans to pursue a Ph.D. in Robotics. His technical expertise includes Python, C++, TensorFlow, and developing scalable solutions for real-world applications.

Chandler Juego is a senior at Washington State University pursuing a Bachelor of Science in Computer Science and graduating in Spring 2025. After graduation, he will be joining Microsoft as a Software Engineer working on trace analysis and operating systems. His technical interests and experience include full-stack development, C, C++, and Python. His role in the RAG application project includes helping write the text embedding code, developing the frontend, optimizing vector search, and connecting the RAG pipeline to the web application.

Adam Shtrikman is a senior at Washington State University, graduating in Summer 2025 with a Bachelor of Science in Computer Science. He has a strong interest in algorithm design and interned as a Software Engineer at Radware, where he worked on cybersecurity-focused projects. Adam plans to pursue a Master's degree in Computer Science after graduation, with aspirations to deepen his expertise in computational theory and applications. For the RAG application project, Adam focuses on natural language processing, adding a custom dataset, and integrating a pipeline to automate the testing process.

### III. Project Requirements Specification

In this section, we will outline the critical elements essential for the project's success. This includes detailing the project's stakeholders, use cases, functional requirements, and non-functional requirements.

#### III.1. Project Stakeholders

Our primary stakeholders are HackerEarth, our client, and our capstone instructor. They require a well-documented RAG application that meets both academic and technical standards. For our team to work efficiently, we rely on good documentation and maintainable code. Although our project is not publicly released, it aims to serve as a strong foundation for future research and development in retrieval-augmented generation.

#### III.2. Use Cases

In the RAG application, the user is the primary actor, interacting with the system in various scenarios. Figure 1 illustrates these interactions in a UML use case diagram.<sup>2</sup> Each use case is listed below, with detailed pre-condition, post-condition, basic path, alternative path, and related requirements.

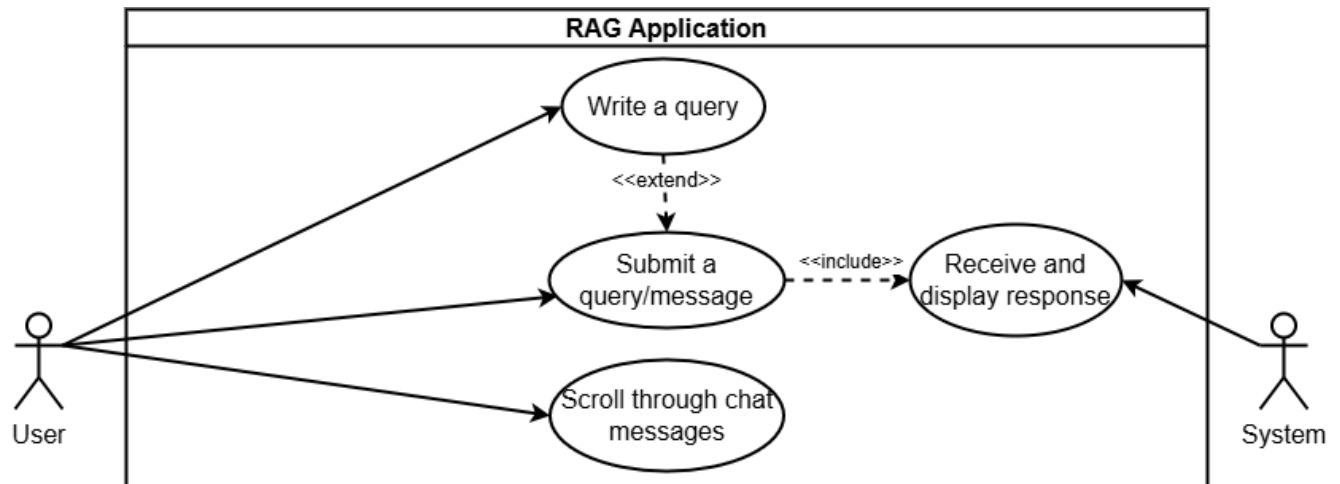


Figure 1: Use Case Diagram

#### Use Case 1: Write a Query

<b>Pre-condition</b>	<ul style="list-style-type: none"> <li>- The user has opened the web application</li> <li>- The query input field is visible and active</li> </ul>
<b>Post-condition</b>	<ul style="list-style-type: none"> <li>- The user has successfully entered a query in the input field</li> </ul>
<b>Basic Path</b>	<ul style="list-style-type: none"> <li>- The user clicks on the query input field</li> <li>- The user types a query or message in the input field</li> <li>- The system accepts and displays the query in the input field</li> </ul>

<b>Alternative Path</b>	<ul style="list-style-type: none"> <li>- If the query input field is inactive or unavailable, the system prompts the user to refresh the page or displays a message saying that the input field is unavailable</li> <li>- If the user input exceeds a character limit, then the system displays an error message next to the input field asking the user to revise the query</li> </ul>
<b>Related Requirements</b>	<ul style="list-style-type: none"> <li>- Chat Window</li> </ul>

### Use Case 2: Submit a Query/Message

<b>Pre-condition</b>	<ul style="list-style-type: none"> <li>- The user has typed a query into the input field</li> <li>- The system is connected to the backend, which is connected to the knowledge graph and vector search database</li> </ul>
<b>Post-condition</b>	<ul style="list-style-type: none"> <li>- The system successfully receives the query, initiating retrieval from the RAG model</li> <li>- The query appears in the chat interface as a sent message</li> </ul>
<b>Basic Path</b>	<ul style="list-style-type: none"> <li>- The user clicks on the "Submit" button or presses "Enter" to submit the query</li> <li>- The system records the query and forwards it to the RAG model for processing</li> <li>- The query appears in the chat window as a sent message from the user</li> </ul>
<b>Alternative Path</b>	<ul style="list-style-type: none"> <li>- If the user submits an empty query, then the system displays a message indicating that input is required</li> <li>- If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window.</li> <li>- If the user submits a query that violates content moderation rules, then a warning message is displayed in the chat window and informs the user that the query is invalid</li> </ul>
<b>Related Requirements</b>	<ul style="list-style-type: none"> <li>- Chat Window</li> <li>- Content Moderation</li> <li>- Error Handling</li> </ul>

### Use Case 3: Receive and Display Response

<b>Pre-condition</b>	<ul style="list-style-type: none"> <li>- The user has submitted a valid query, and the system has processed it using the RAG model</li> </ul>
<b>Post-condition</b>	<ul style="list-style-type: none"> <li>- The system returns a response based on the query and displays it in the chat window</li> </ul>
<b>Basic Path</b>	<ul style="list-style-type: none"> <li>- The system processes the query using the RAG model</li> </ul>

	<ul style="list-style-type: none"> <li>- The system retrieves relevant information from Wikipedia or the custom dataset</li> <li>- The system generates a response using the RAG model and formats the information in a readable message</li> <li>- The system displays the response in the chat window above the user input field</li> </ul>
<b>Alternative Path</b>	<ul style="list-style-type: none"> <li>- If the user submits an empty query, then the system displays a message indicating that input is required</li> <li>- If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window</li> </ul>
<b>Related Requirements</b>	<ul style="list-style-type: none"> <li>- Chat Window</li> <li>- Query Response Retrieval</li> <li>- Error Handling</li> <li>- Custom Dataset</li> </ul>

## Use Case 4: Scroll Through Chat Messages

<b>Pre-condition</b>	<ul style="list-style-type: none"> <li>- The user has had an ongoing chat session with multiple messages exchanged between the user and the system</li> </ul>
<b>Post-condition</b>	<ul style="list-style-type: none"> <li>- The system successfully receives the query, initiating retrieval from the RAG model</li> </ul>
<b>Basic Path</b>	<ul style="list-style-type: none"> <li>- The user scrolls up in the chat window</li> <li>- The system dynamically loads previous queries and responses from the current session as the user scrolls</li> <li>- The user can review and scroll back to any message in the session</li> </ul>
<b>Alternative Path</b>	<ul style="list-style-type: none"> <li>- If the system encounters an issue loading older messages, a loading spinner is shown until the history is fully loaded</li> <li>- If the chat session is too long, the system may implement infinite scrolling to manage performance</li> </ul>
<b>Related Requirements</b>	<ul style="list-style-type: none"> <li>- Chat Window</li> </ul>

## III.3. Functional Requirements

In this section, we present each functional requirement with a detailed description, source, and priority level. These elements are crucial for understanding the system's implementation and ensuring its successful operation.

### III.3.1. RAG Model Components

#### RAG Model Architecture

<b>Description</b>	The system will implement RAG architecture, including a receiver and generator. The receiver will search the knowledge base for the most relevant data based on user input, and the generator (LLM) will take the search results and generate an accurate response.
<b>Source</b>	Project scope document provided by the client.
<b>Priority</b>	<u>Priority Level 0</u> : Essential functionality.

#### Vector Search Implementation

<b>Description</b>	The system will implement vector search to retrieve relevant information based on user queries. This includes generating embeddings, indexing them using vector search libraries (FAISS), and finding similar results based on user input.
<b>Source</b>	Project scope document provided by the client.
<b>Priority</b>	<u>Priority Level 0</u> : Essential functionality.

#### Knowledge Graph Integration

<b>Description</b>	The system will integrate knowledge graphs (e.g., DBpedia or YAGO) into the RAG pipeline to better retrieve relevant and contextual information for the query.
<b>Source</b>	Project scope document provided by the client.
<b>Priority</b>	<u>Priority Level 0</u> : Essential functionality.

### III.3.2. Core App Functionality

#### Query Response Retrieval

<b>Description</b>	The system will process user queries, fetch relevant information from the knowledge graph and vector search index and generate responses using an LLM. If the system does not know the answer to a query, it should tell the user that it does not have enough information to respond accurately; it should not lie.
<b>Source</b>	Project scope document provided by the client.
<b>Priority</b>	<u>Priority Level 0</u> : Essential functionality.



### Custom Dataset

<b>Description</b>	The system will allow the user to query and receive a response based on a custom dataset instead of the Wikipedia dataset.
<b>Source</b>	Internal requirements elicitation among members of the team.
<b>Priority</b>	<u>Priority Level 1</u> : Desirable functionality.

### III.3.3. User Interface

#### Chat Window

<b>Description</b>	The system will display a chat window where users can type messages or queries. Once they press enter or click submit, the system will generate and display a response.
<b>Source</b>	Internal requirements elicitation among members of the team.
<b>Priority</b>	<u>Priority Level 0</u> : Essential functionality.

#### Error Handling

<b>Description</b>	The system will handle errors like invalid queries, network issues, or app failures by displaying appropriate error messages in the chat window.
<b>Source</b>	Internal requirements elicitation among members of the team.
<b>Priority</b>	<u>Priority Level 1</u> : Desirable functionality.

#### Content Moderation

<b>Description</b>	The system will prevent responses to harmful or inappropriate queries by showing a warning message. It will moderate content that includes hate speech, threats, violence, or graphic material. However, it will recognize when the query is research-based (e.g., questions about sensitive historical events like the Holocaust) and respond appropriately.
<b>Source</b>	Internal requirements elicitation among members of the team.
<b>Priority</b>	<u>Priority Level 1</u> : Desirable functionality.

### III.4. Non-Functional Requirements

The non-functional requirements outline the system's operational qualities, such as performance, scalability, and security, to ensure it meets quality standards beyond core functionality. The details of non-functional requirements are given below.

Non-Functional Requirement	Description
Scalability	The system shall be scalable to handle large datasets, ensuring it can efficiently manage and query over 10,000 Wikipedia articles without degradation in performance.
Response Time	The system will respond to user queries within 15 seconds for typical requests, ensuring a smooth user experience.
Data Storage	The system will maintain sufficient storage for the knowledge graph and vector embeddings, ensuring persistent and reliable data retrieval.
Maintainability	The system will be designed with maintainability and modular code that allows for future updates and extensions without significant refactoring.
Reliability	The system will maintain a 99% uptime, ensuring high availability for users, particularly during the testing and demonstration phases.
Security	The system will ensure the security of data and queries, particularly in handling sensitive information, by implementing secure protocols for data transmission and storage.
User Interface Usability	The web interface will be intuitive and easy to use, enabling users to input queries and view results without requiring extensive technical knowledge.

## **IV. Software Design - From Solution Approach**

This document provides a detailed solution approach for the RAG application, designed to enhance information retrieval and generation using knowledge graphs and vector search techniques. The purpose of this section is to outline the architectural decisions, system components, and strategies that will be implemented to meet the project's goals. The intended audience for this document includes technical stakeholders, developers, and engineers involved in developing and evaluating this RAG application.

The RAG application bridges the gap between large language models (LLMs), vector search, and knowledge graphs to create a more effective information retrieval system. At its core, the system is designed to query a large dataset of Wikipedia articles, retrieve semantically relevant data using vector embeddings, and enrich those results with structured relationships from a knowledge graph. This integration will allow the application to generate contextually appropriate and accurate responses.

The system consists of several key components: a frontend interface for user interaction, a backend responsible for processing queries, a vector search handler for retrieving relevant information, and a knowledge graph handler for enhancing query responses with structured data. The overarching goal of this architecture is to facilitate efficient and accurate information retrieval in real time, supporting a range of use cases, including conversational agents and research tools.

### **IV.1. Architecture Design**

In this section, we describe the system's architecture and various components, including the interfaces they use to interact with each other. The architecture provides a top-level design view that serves as a foundation for further detailed design work.

#### **IV.1.1. Overview**

The architecture of the RAG application follows a client-server pattern, with React managing the frontend and the backend handling query processing and response generation. For the frontend, React leverages the Component design pattern to create a modular and reusable user interface. It also employs the Observer pattern, where event listeners track user inputs, such as queries, and dynamically display results. For the backend, various handlers manage specific tasks: the Vector Search (VS) Handler retrieves relevant data using embeddings, while the Knowledge Graph Handler enriches responses with structured data. The Large Language Model (LLM) Handler uses the OpenAI LLM to process user queries and generate an accurate natural language response. Each component interacts through well-defined interfaces, ensuring flexibility and scalability. Figure 2 is a UML component diagram that displays the relationships between components in the system.<sup>3</sup> The following sections will give more details on each subsystem and the design choices behind them.

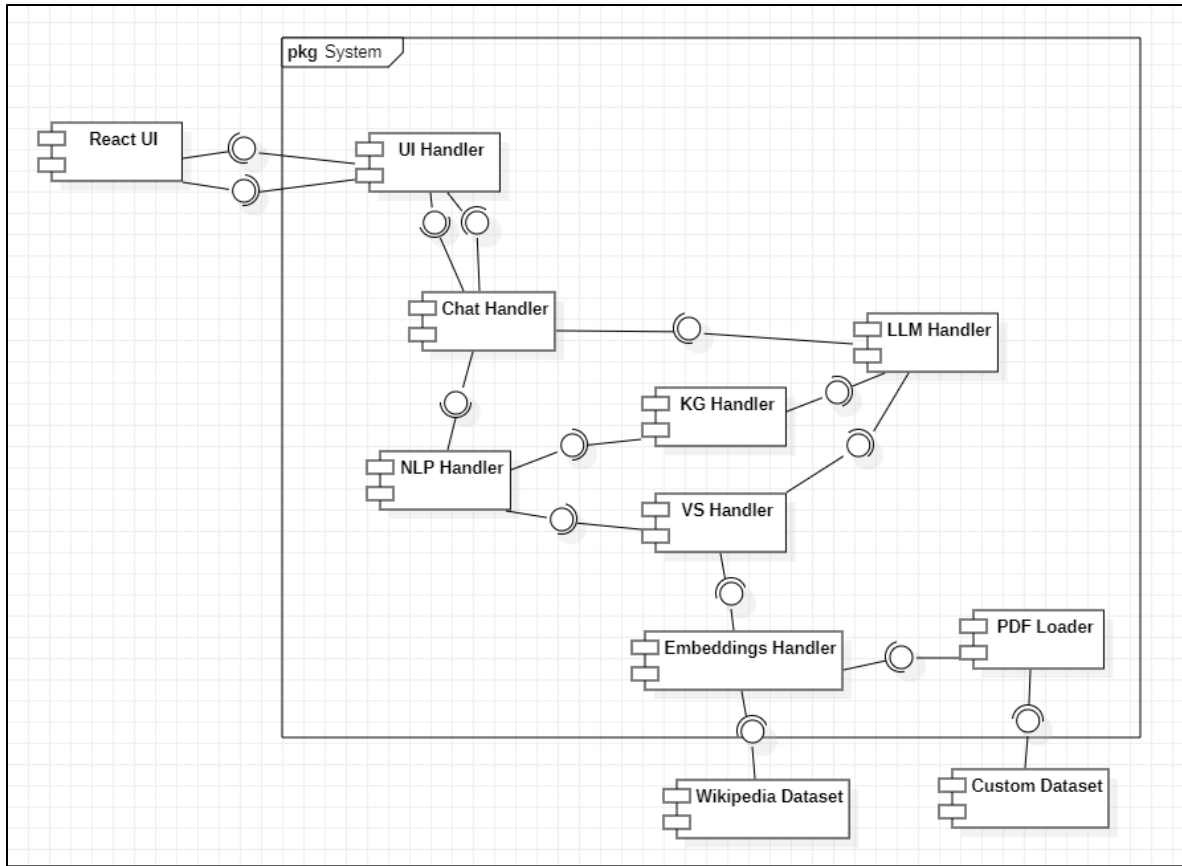


Figure 2: System Architecture and Component Data Flow

### IV.1.2. Subsystem Decomposition

The system is divided into smaller, manageable subsystems to streamline development and integration. Each part focuses on specific functions to ensure smooth operation.

#### IV.1.2.1. UI Handler

##### Description

The UI Handler subsystem manages any user interaction with the React UI elements and UI details such as the displayed layout and icons. It routes requests from the UI to the Chat Handler if appropriate.

##### Concepts and Algorithms Generated

The UI Handler has a subclass Chat Handler that contains all user interaction with the chat window. The UI Handler manages communication between user interaction and the backend, taking user input and routing it to various other components. It maintains the application's state using React, ensuring that the UI reflects the most recent data.

##### Interface Description

Services Provided:

Service Name	Service Provided To	Description
UpdateLayout	React UI, ChatHandler	The UpdateLayout service will allow the React UI or the Chat Handler to call for an update to the page layout. This will occur for the transitions between the main chat page, the login/sign-up page, and the settings page.

Services Required:

Service Name	Service Provided From
ModifyUI to UI Handler	React UI
ManageChatUI	Chat Handler

#### IV.1.2.2. Chat Handler

##### Description

The Chat Handler manages all interactions related to chat functionality. It handles incoming messages from users, processes them, and directs them to the appropriate services like the Vector Search (VS) Handler or the Knowledge Graph (KG) Handler. The Chat Handler also retrieves responses from these components and sends them back to the UI.

##### Concepts and Algorithms Generated

The Chat Handler processes incoming user messages and sends the data to the VS and KG handlers to query it. Once responses are retrieved from other components, the Chat Handler ensures a response is properly formatted and displayed in the chat.

##### Interface Description

Services Provided:

Service Name	Service Provided To	Description
RouteQuery	NLP Handler	RouteQuery sends the query to be processed in the NLP Handler.
ManageChatUI	UI Handler	ManageChatUI handles updates to the chat UI, including displaying new messages, interactions, and generated responses.

Services Required:

Service Name	Service Provided From
GenerateResponse	LLM Handler

#### IV.1.2.3. NLP Handler

##### Description

The Natural Language Processing (NLP) Handler is responsible for processing user queries and extracting relevant information to pass to the KG and VS handlers. It parses the query, detects the user's intent, and recognizes entities in it. For example, the NLP Handler would tokenize the query "Tell me about Albert Einstein" into individual tokens: "Tell", "me", "about", "Albert", and "Einstein", while identifying "Albert Einstein" as a relevant entity within the request. The NLP Handler then routes the processed query to the KG and VS Handlers.

##### Concepts and Algorithms Generated

The NLP Handler uses several Natural Language Processing algorithms. First, it tokenizes and cleans the query for easier analysis. Second, it extracts important entities from the query to help the KG and VS Handlers retrieve the right information. An entity could be a name, location, date, or any keyword that holds significant value for retrieving the correct information. Finally, this component determines the main intent of the query, whether it is asking for facts, definitions, or something more complex.

##### Interface Description

Services Provided:

Service Name	Service Provided To	Description
ProcessQuery	VS Handler, KG Handler	ProcessQuery processes the query and extracts relevant information to be sent to VS Handler and KG Handler.

Services Required:

Service Name	Service Provided From
RouteQuery	Chat Handler

#### IV.1.2.4. Knowledge Graph (KG) Handler

##### Description

The KG Handler is responsible for querying DBpedia to obtain answers based on user inputs. It uses SPARQL to retrieve relevant information directly from DBpedia and sends these results to the LLM Handler for response generation. An example of using the KG handler is retrieving the abstract of "Albert Einstein" from DBpedia based on the entity identified by the NLP handler.

### Concepts and Algorithms Generated

The KG Handler uses SPARQL to query DBpedia. Specifically, it constructs SPARQL queries to retrieve answers based on the user's input. The retrieved data is then sent to the LLM Handler for further processing and response generation.

### Interface Description

Services Provided:

Service Name	Service Provided To	Description
QueryKG	LLM Handler	QueryKG uses SPARQL to query DBpedia and retrieves answers based on user inputs, sending the results to the LLM Handler.

Services Required:

Service Name	Service Provided From
ProcessQuery	NLP Handler

#### IV.1.2.5. Vector Search (VS) Handler

### Description

The VS Handler is responsible for performing vector search to retrieve semantically relevant information from the Wikipedia dataset. It uses FAISS to search efficiently for similar embeddings generated from user queries. The VS Handler works alongside the Embeddings Handler, which processes and indexes the data, allowing the VS Handler to quickly perform similarity searches.

### Concepts and Algorithms Generated

The VS Handler leverages vector search algorithms and similarity metrics, utilizing FAISS match query embeddings against pre-indexed embeddings from Wikipedia articles. This process ensures efficient and accurate retrieval of relevant information based on the input query. Once the VS Handler identifies relevant information, it sends the results to the LLM Handler for further processing and response generation.

### Interface Description

Services Provided:

Service Name	Service Provided To	Description
VectorSearch	LLM Handler	VectorSearch retrieves relevant information using vector search and sends search results to the LLM Handler.

Services Required:

Service Name	Service Provided From
GenerateEmbeddings	Embedding Handler
GenerateCustomEmbeddings	PDF Loader
ProcessQuery	NLP Handler

#### IV.1.2.6. LLM Handler

##### Description

The LLM Handler processes the user's query along with the combined information retrieved from the VS Handler and KG Handler. Both the query and the retrieved information chunks are passed to the OpenAI LLM, which generates a coherent and contextually accurate natural language response. The LLM Handler then finalizes the response and sends it to the Chat Handler, which displays it to the user.

##### Concepts and Algorithms Generated

The LLM Handler takes the user query and the combined VS Handler and KG Handler results, sending them to the OpenAI model for processing. OpenAI interprets the query, integrates relevant information from the vector search results, and generates a natural language response that is accurate and contextually relevant. The LLM Handler ensures the response is complete and coherent before sending it to the Chat Handler for display.

##### Interface Description

Services Provided:

Service Name	Service Provided To	Description
GenerateResponse	Chat Handler	GenerateReponse queries OpenAI with the user query and data from VS Handler to generate accurate and coherent responses.

Services Required:

Service Name	Service Provided From
VectorSearch	VS Handler
QueryKG	KG Handler



#### IV.1.2.7.      **Embeddings Handler**

##### **Description**

The Embeddings Handler transforms Wikipedia data into vector representations for efficient searching by the VS Handler. It processes large Wikipedia text datasets, converting them into high-dimensional embeddings using a pre-trained SentenceTransformer language model. These embeddings capture semantic information, enabling the VS Handler to perform similarity searches based on the meaning of the user query rather than exact keyword matches.

##### **Concepts and Algorithms Generated**

The Embeddings Handler uses SentenceTransformer to encode text into dense vector embeddings. It splits Wikipedia articles into chunks and cleans and tokenizes them. Each chunk of text is passed through the SentenceTransformer model, which outputs a vector representing the semantic meaning of the text. These vectors are stored in an index, allowing the VS Handler to efficiently retrieve the most relevant ones when processing a user query.

##### **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
GenerateEmbeddings	VS Handler	GenerateEmbeddings converts Wikipedia text into embeddings for vector search.

Services Required:

Service Name	Service Provided From
LoadWikipediaData	Wikipedia Dataset

#### IV.1.2.8.      **PDF Loader**

##### **Description**

PDF Loader takes PDF documents of a custom dataset (class notes) and prepares them for the RAG application.

##### **Concepts and Algorithms Generated**

PDF Loader transforms the PDF documents into a parquet file. Both the Wikipedia and custom dataset parquet files are then cleaned, concatenated, and embedded.

## Interface Description

Services Provided:

Service Name	Service Provided To	Description
GenerateCustomEmbeddings	VS Handler	GenerateCustomEmbeddings converts PDF files into embeddings for vector search.

Services Required:

Service Name	Service Provided From
LoadCustomDataset	Custom Dataset

## IV.2. Data Design

For the RAG application, the main data structures are raw datasets and text embeddings. The Wikipedia and custom datasets are stored as Parquet files. Once generated, the embeddings are stored as NumPy arrays. The embeddings are organized and indexed using the FAISS libraries into FAISS files. Additionally, user queries are converted into vector embeddings for retrieval.

## IV.3. User Interface Design

For the UI design of our RAG Web Application, we have designed a minimalistic layout that mimics the structure of a typical chat or messaging system. The limited color scheme allows the user to focus on the app rather than distracting UI elements. This design allows for clear and intuitive interaction with our RAG model and will serve as the core interface for users to input and receive feedback from the system.

The chat box facilitates interactions between the user and the system. It contains user messages and RAG model responses, providing a clean and efficient way for users to input their data and receive real-time responses. Below the chat box, there is an input field where users can type their queries or instructions, followed by a 'Send' button to submit the input. This mirrors a familiar chat-based interface, making the interaction intuitive.

Once the user enters the application, they are introduced to the UI's main page, which contains a chat box at the center, as shown in Figure 3.

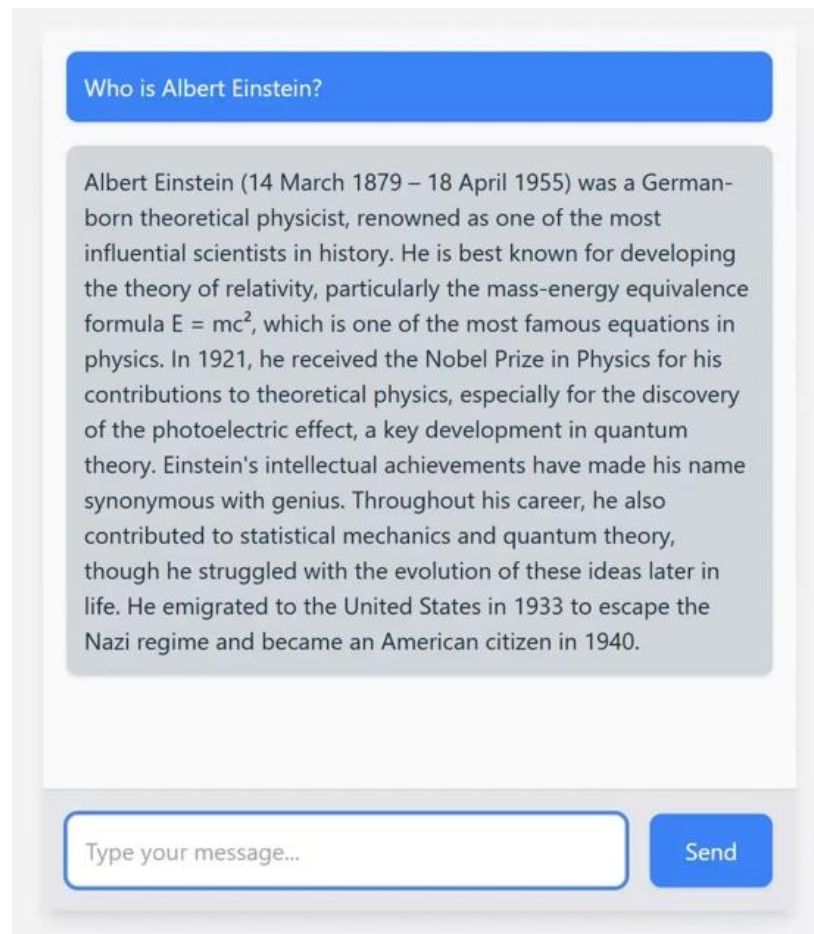


Figure 3: UI Design of Chat Box

## **V. Test Case Specifications and Results**

This section provides an overview of the test objectives and plans, including the testing approach, tools used, and types of testing conducted. It also details the test environment requirements and documents the test results for functional and non-functional requirements.

### **V.1. Testing Overview**

The purpose of testing our RAG application is to gain confidence that the code will successfully and consistently execute the desired behavior and fulfill all requirements at runtime. Thorough testing is essential for ensuring software reliability and efficiency. Automated testing protects our application, ensuring that core functionality always works before code is merged. We used Continuous Integration (CI) with GitHub Actions. With every new commit in the repository, a workflow builds and tests the code to make sure the commit does not introduce errors. This testing section outlines our approach to unit, integration, and system testing (functional, performance, and user acceptance). Additionally, it describes specific test cases and their outcomes.

#### **V.1.1. Unit Testing**

We followed traditional unit testing procedures, taking the smallest unit of testable software in the application, isolating it from the remainder of the code, and testing it for bugs and unexpected behavior. To verify individual functionality, we tested isolated handlers (LLM Handler, KG Handler, VS Handler, NLP Handler, etc.). We ran unit tests to ensure the accurate processing of inputs and expected outputs for each handler, mocking input and output to isolate individual components. To ensure thorough coverage, we covered edge cases, boundary conditions, and performance. We used the PyTest Python framework for creating and running tests.

To ensure that the code is sufficiently tested, the team has tested all core app functionality. Core functionalities are those that, if non-operational, render the app unusable. These functionalities were mentioned in the original project abstract.

#### **V.1.2. Integration Testing**

Integration testing detects faults that might have been missed during unit testing by focusing on small groups of components. In this process, two or more components are integrated and tested together. When no new faults are found, additional components are added to the group.

For the RAG application, we have grouped interconnected components to identify faults in their communication. From the architecture diagram, we have tested how data flows between the NLP Handler, VS Handler, KG Handler, and LLM Handler to ensure that accurate responses are generated. We started with pairs of handlers and slowly added more until the whole system was tested. We did not need many integration tests because we have limited use cases, and the unit tests were sufficient.

#### **V.1.3. System Testing**

System testing for the RAG application involves validating the integrated components as a unified system to verify that it meets functional, performance, and acceptance requirements. This testing phase ensures the application operates effectively and meets user and stakeholder expectations.

### **V.1.3.1. Functional Testing**

The RAG application's functional testing relied on manual testing conducted by the development team. Each functional requirement was paired with a corresponding functional test, ensuring all essential features work as intended. This includes testing the core functionalities of query processing, vector search, and knowledge graph integration.

The testing was performed iteratively, with developers manually validating the application's performance against the expected outputs. If a functional test failed, the testing developer documented the test conditions and notified the component's original developer to resolve the issue. Given the critical role of structured data integration, the team continuously reviewed and refined functional tests as the project evolved.

### **V.1.3.2. Performance Testing**

Performance testing for the RAG application assessed its efficiency and resilience under various conditions, ensuring it met non-functional requirements. Specifically, we tested response time and made iterative changes to improve it. Other metrics, like scalability and stress, were not tested because the application was deployed for single users.

Initially, the latency was too high: around 45 seconds. To combat this, we switched the LLM from the large LLaMA 2-7b model to the lightweight OpenAI model and saw instant improvements. Latency was down to around 20 seconds. After introducing parallel KG and VS operations, changing the embeddings model from BERT to SentenceTransformers, and packaging the app, we reduced latency to 6-8 seconds. This optimization ensured the RAG app is efficient, keeps users engaged, and makes the app accessible to less powerful devices.

### **V.1.3.3. User Acceptance Testing**

User acceptance testing (UAT) for the RAG application primarily involved getting feedback from our client at HackerEarth. Since our application is stand-alone and will not be integrated into their codebase, our UAT focused on validating core functionalities based on client expectations.

During weekly meetings with our client, we shared current progress and a functional prototype when available. He was able to interact with the system and test key features like query processing and response accuracy. Our client's guidance was essential to optimize the product. UAT confirmed that the system met the agreed-upon requirements and worked as expected.

## **V.2. Environment Requirements**

Our testing setup ensures an efficient, reproducible testing environment, covering all necessary tools and infrastructure for testing the RAG application's functional and non-functional requirements. Below are the key requirements.

### **V.2.1. Hardware Requirements**

Development workstations should have at least 16GB of RAM and 4 CPU cores to handle the necessary computational tasks. Normal computers are sufficient for testing and running the application.

### **V.2.2. Software Requirements**

The development and testing environments should run on operating systems such as Windows, macOS, or Linux. For the backend, Python is required, along with dependencies like FAISS for

vector search, SPARQL libraries for knowledge graph queries, and PyTest for testing. The frontend is built using React, accessible through a modern web browser. The README shares instructions on how to install these dependencies.

## V.2.3. Network and Databases

Stable internet connectivity is essential for handling live queries to knowledge graph sources. No databases are necessary; downloading the text embeddings and indexed FAISS embeddings from the GitHub README is sufficient.

## V.2.4. CI/CD and Testing Tools

GitHub Actions is used to automate the testing process. A push or PR to main will trigger the testing pipeline. If it fails, there are detailed error logs available to investigate bugs.

## V.3. Test Results

This section presents the results of unit tests conducted for the RAG application, focusing on different components, including NLP, vector search, LLM, and knowledge graph querying. The tests ensure proper functionality, performance, and integration of various pipeline components. Our test cases are shown in Table 1.

Table 1: Test Results for Each Test Case

Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
NLP Handler - Entity Extraction (Normal)	The NLP handler correctly extracts named entities from input queries.	Named entities are successfully extracted from input queries.	Pass	Ensure the NLP model is loaded and initialized correctly.
NLP Handler – Entity Extraction (Edge Cases)	The NLP Handler handles ambiguous entities and multi-word entities correctly.	Expected entities are extracted.	Pass	Ensure the NLP can handle ambiguous phrases.
NLP Handler - Missing Entities	The NLP Handler tokenizes the input and extracts no entities.	No entities are extracted, as expected.	Pass	Ensure the NLP handler can handle simple queries with no entities.
NLP Handler – Tokenization (Normal)	The input query is correctly tokenized into meaningful	Tokenized words match expected segmentation.	Pass	Ensure spaCy or other NLP libraries are properly installed.

## RAG Application Using Knowledge Graph and Vector Search

	components.			
NLP Handler - Tokenization (Special Characters)	The NLP Handler properly tokenizes input with punctuation and special characters.	Tokenized output matches expected segmentation. No punctuation is in output.	Pass	Ensure queries with special characters and punctuation are tokenized correctly.
NLP Handler - Harmful Intent Detection	The NLP recognizes when the question has a harmful intention related to violence.	The field is_harmful returns true.	Pass	Ensure harmful phrases are recognized.
DBpedia Query Execution (Normal)	The system correctly generates and executes SPARQL queries against DBpedia.	Query execution returns relevant information.	Pass	DBpedia endpoint must be accessible.
DBpedia Query Extraction - No Result Handling	The system gracefully handles queries with no results.	Returns a "No results found" instead of errors.	Pass	Ensure the system will continue to answer the question even if the knowledge graph does not work.
Vector Search - Indexing	FAISS indexes embeddings correctly and can be queried efficiently.	Embeddings are indexed, and searches return relevant vectors.	Pass	Ensure FAISS is installed and data embeddings are available.
Vector Search - Retrieval Accuracy	The top-k most relevant results are retrieved based on query embeddings.	The system returns relevant document indices.	Pass	Ensure test data embeddings are precomputed.
Vector Search - Performance Under Load	The system performs efficient searches even with a large embedding set.	Retrieval remains within an acceptable response time.	Pass	Ensure the retrieval speed is efficient with large datasets.

LLM Handler - Response Length Compliance	The LLM does not exceed the max token length.	Response length stays within the limit.	Pass	Ensure the max length parameter is properly enforced.
--	---	---	------	---

## Manual Testing Observations

Additional manual tests were conducted to verify performance and usability. These included:

- Testing response latency across different query complexities
- Ensuring the integration of NLP, vector search, and knowledge graph retrieval
- Verifying that changes in vector search and indexing improved response accuracy

All unit tests passed successfully, confirming the stability and correctness of the implemented features.

## VI. Projects and Tools Used

Below is a summary of the libraries, frameworks, and tools used to implement the RAG application.

Tool/library/framework	Purpose
SentenceTransformer	SentenceTransformer is a Python framework used to generate embeddings from the Wikipedia articles and custom text data. <sup>4</sup> It focuses on sentences rather than word-level embeddings.
FAISS	FAISS allows for efficient similarity search for fast vector search retrieval. <sup>5</sup>
DBpedia	DBpedia is a knowledge graph that extracts information from Wikipedia. <sup>1</sup>
React	React is a frontend framework for building the user interface. <sup>6</sup>
Pytest	Pytest is a testing framework for ensuring code reliability. <sup>7</sup>
OpenAI	OpenAI is the LLM used for text generation and retrieval. <sup>8</sup>
Node.js	Node.js is the backend runtime environment used for handling API requests and server logic. <sup>9</sup>



GitHub Actions	GitHub Actions for CI/CD were used to automate the testing process. <sup>10</sup>
Docker	Docker was used to build and deploy the application using containers. <sup>11</sup>
Docker Compose	Docker Compose is a tool for defining and running multi-container apps on Docker. <sup>12</sup>
FastAPI	FastAPI is a Python framework used to build efficient backend APIs for our backend server.

Below is a summary of the languages used in the project and their purposes.

Language	Purpose
Python	Python was used across the application, including routing requests, generating embeddings, running vector search, calling the LLM, querying the knowledge graph, and running tests.
TypeScript	TypeScript was used to build the frontend of the RAG application using the framework React.
SPARQL	SPARQL is a query language used to retrieve information from DBpedia. <sup>13</sup>

## VII. Description of Final Prototype

The final prototype of the RAG application provides a functional implementation of a retrieval-augmented generation system designed to process user queries with improved contextual awareness. The system integrates knowledge graphs, vector search, and natural language processing to generate enriched responses. The app is deployed using docker. This section covers an overview of the system, a user guide, major use cases, and screenshots of the application.

### VII.1. System Overview

The RAG application consists of the following core components:

- **Frontend:** A React-based user interface where users can write questions and see responses
- **Backend:** A Python-based server handling knowledge graph queries, vector search retrieval, embedding creation, and response generation
- **Natural Language Processing:** Analyzes user input to construct appropriate queries to the knowledge graph
- **Knowledge Graph Integration:** Enhances response by providing additional context from the DBpedia knowledge graph
- **Vector Search:** Retrieves semantically relevant information using embeddings, leveraging the Wikipedia dataset to enhance the depth and accuracy of responses
- **OpenAI LLM Integration:** Generates natural language response by synthesizing retrieved information

The system is accessible through a web interface, with a backend that manages API calls and processes queries in real time.

### VII.2. User Guide

Below are the steps to run the RAG application.

- **Access the System**
  - Navigate to the hosted instance or run the application locally by following the setup instructions in the GitHub repository
- **Type a Question**
  - Type a question in the chat box (e.g., "Who is Alan Turing?")
- **Review Generated Response**
  - The system will process the query, retrieve relevant information with vector search and the knowledge graph, and then generate a final response with an LLM

### VII.3. Major Use Cases

Users can ask general knowledge questions using the Wikipedia dataset. However, the true power of AI is its ability to work with custom datasets. Currently, the system has a custom class notes dataset, making it a valuable study tool where students can ask course-related questions. This custom dataset can be easily replaced with other data sources, such as private company documents, allowing employees to retrieve relevant information quickly.

### VII.4. Prototype Screenshots

Below are screenshots of the final prototype and an example of one of our tests used to debug the application. The final product allows the user to type a question, clearly shows processing and loading, and then displays the final response.

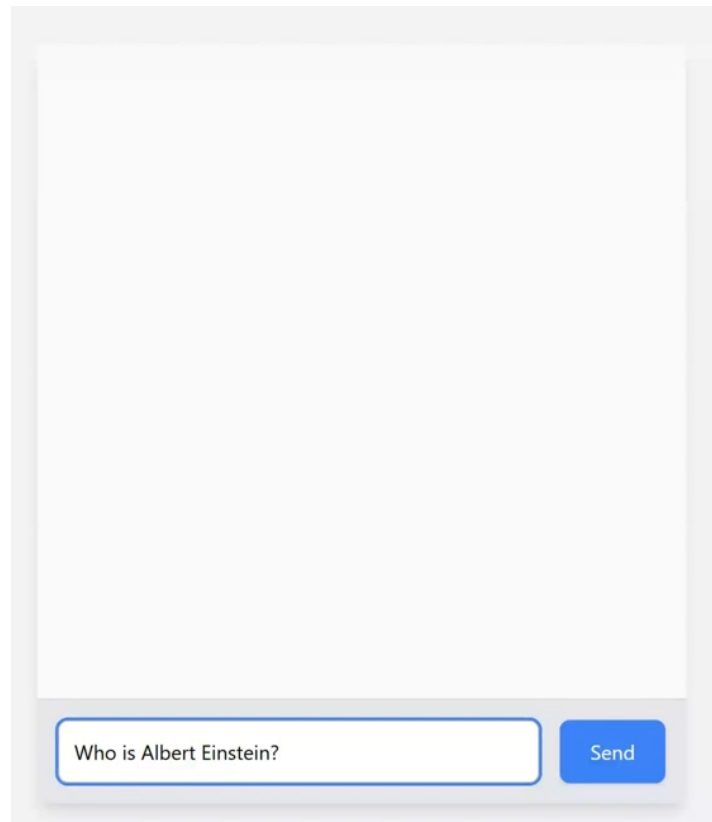


Figure 4: User Typing

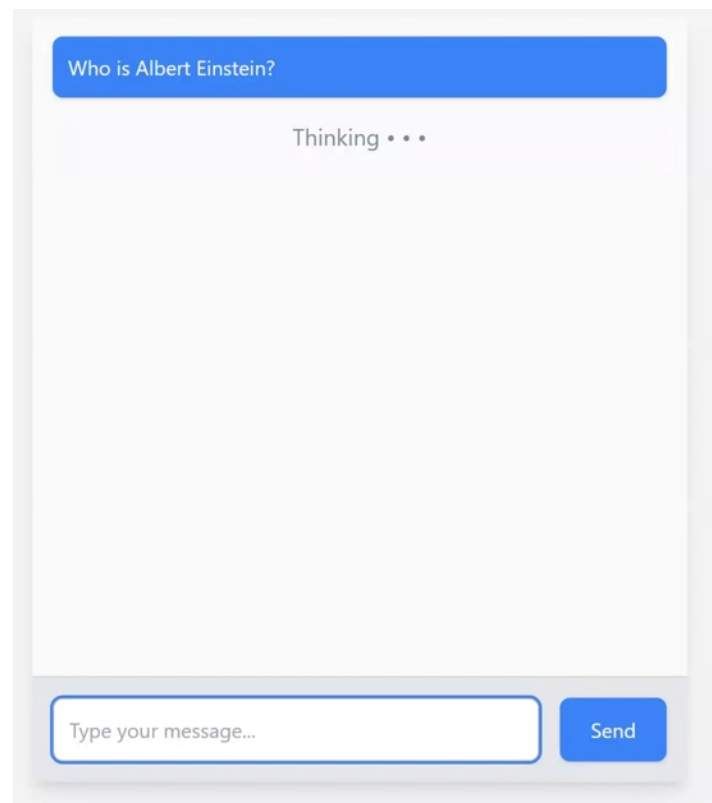


Figure 5: Loading Message

## RAG Application Using Knowledge Graph and Vector Search

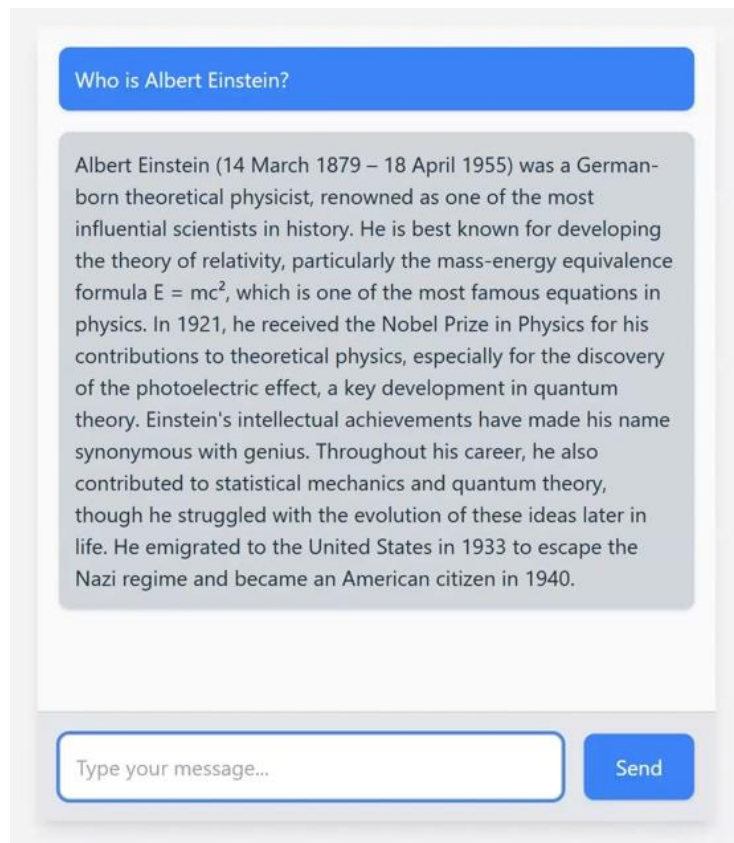


Figure 6: Final Answer

```
nlp = spacy.load("en_core_web_sm")
client = TestClient(app)

def test_process_query_basic():
    response = client.post("/nlp/process_query",
                           json={"query": "What is the capital of France?"})
    assert response.status_code == 200
    data = response.json()

    # Check if response contains expected fields
    assert "tokens" in data
    assert "entities" in data
    assert "is_harmful" in data
    assert "sparql_query" in data

    # Validate tokens
    assert "what" in data["tokens"]
    assert "capital" in data["tokens"]
    assert "France" in data["tokens"]

    # Validate entities
    assert len(data["entities"]) > 0 # At least one entity, like 'France'

    # Validate harmful intent detection
    assert data["is_harmful"] is False

    # Normalize whitespace in SPARQL query for comparison
    expected_query = '''
    SELECT ?abstract WHERE {
      ?subject rdfs:label "France"@en .
      ?subject dbo:abstract ?abstract .
      FILTER (lang(?abstract) = 'en')
    }
    '''
    assert ''.join(data["sparql_query"].split()) == ''.join(expected_query.split()), f"SPARQL query mismatch: {data['sparql_query']}"
```

Figure 7: Test Code for Natural Language Processing

## VIII. Product Delivery Status

The final project will be demonstrated to our client and instructor near the end of the spring semester. While HackerEarth will not be using our code, we provided a Docker-based setup for testing and demonstration purposes. Deployment and code were completed on April 18<sup>th</sup>, 2025.

The project is available on the team's [GitHub](#), with detailed setup instructions in the README. No physical equipment was used or stored. An API key is required for OpenAI, which is not provided due to cost constraints.

To set up and run the project, users can pull the pre-built Docker images from our repository and use Docker Compose to run the application locally. The repository includes a Dockerfile and a docker-compose.yml file to streamline deployment. The README provides detailed instructions for pulling images and running the application using Docker.

## IX. Conclusions and Future Work

The below sections address the current limitations and provide recommendations for enhancing response quality, relevance, and performance. It also outlines potential future work and improvements.

### IX.1. Limitations and Recommendations

One of the main challenges in our current prototype is testing the response quality and relevance. We incorporated manual quality testing; however, evaluating response accuracy and relevance is complex and can always be improved. Potential solutions include automated benchmarking using established datasets, more human manual testing, and refined retrieval strategies.

Next, the NLP component struggles with queries containing multiple entities and complex relationships. This can lead to incomplete or inaccurate responses. Our client understood the difficulty of building high-quality natural language processing and asked us to focus on other areas. However, potential improvements could include refining entity linking and using specialized models trained in complex queries.

Finally, response time performance could be improved. The average response time is currently 7615.07 ms or 7.615 seconds. We switched the LLM from the LLaMA 2.7b model to OpenAI, significantly improving latency from 5-10 minutes to seconds. Other possible optimizations include caching frequent queries to reduce redundant API calls and using more efficient embeddings or indexing methods to speed up retrieval.

### IX.2. Future Work

In summary, our team successfully developed and deployed an AI chatbot application using the RAG technique, vector search, a knowledge graph, and a custom dataset. The application is fully packaged and operational, performing reliably with an average response time of 8-10 seconds. It delivers accurate and relevant responses to users, demonstrating both effectiveness and precision. To ensure quality, we adhered to standard testing procedures and implemented automated testing and deployment workflows. We carefully followed every step in the software engineering development and deployment process to ensure a high-quality final product.

While the project is complete according to requirements, there are many ways to extend it and add more capabilities in the future. As expressed in the Limitations section, NLP could be improved to better analyze complex queries and better moderate inappropriate questions. Future developers could include more custom datasets related to complex fields like healthcare, finance, technology, and education. They could make the application multimodal, processing any input, including text, image, audio, etc. For our class notes custom dataset, multimodal features could return information about class diagrams and charts. Finally, future developers could introduce more full-stack features like user registration, saved chats, and chat memory. These improvements would strengthen the app, making it more versatile and broadening its impact.

## X. Acknowledgements

For this section, the team would like to make a few acknowledgments. First, we would like to thank our client, Vikas Aditya, CEO of HackerEarth. Vikas has been a consistent source of advice and mentorship, playing a pivotal role in our success. Secondly, to our capstone professor, Dr. Parteek Kumar, who guided us through the development and documentation process.

## XI. Glossary

**CI/CD:** Continuous Integration/ Continuous Deployment. This provides the team a method to ensure tests are passing as software is updated by developers.

**Functional requirements:** The requirements for the RAG application that can be tied to a software functionality implemented by the team.

**Knowledge Graph:** A knowledge base that uses a graph-structured data model or topology to represent and operate on data.

**Large Language Model (LLM):** A machine learning model that can perform natural language processing tasks. They are trained on vast amounts of data to detect patterns effectively.

**Natural Language Processing (NLP):** A branch of artificial intelligence that uses machine learning to help computers interpret and generate human language.

**Non-functional requirements:** The requirements for the RAG application that are general, often qualitative, and not related to a specific functionality.

**Retrieval-Augmented Generation (RAG):** A framework combining large language models' capabilities with information retrieval systems to improve the accuracy and relevance of AI-generated text.

**Vector Search:** A method for finding similar items to data points in large collections by using vector representations of items. These representations, or vector embeddings, can quickly locate items in large data sets and allow for searches by meaning, rather than just keywords.

**UI:** User Interface. The space where the user and the RAG system interact and communicate.

## XII. References

- [1] "About DBpedia," *DBpedia*. <https://www.dbpedia.org/about/>
- [2] "Use Case Diagram | StarUML documentation." <https://docs.staruml.io/working-with-uml-diagrams/use-case-diagram>
- [3] "What is Component Diagram?" <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/>
- [4] "SentenceTransformers Documentation — Sentence Transformers documentation." <https://sbert.net/>
- [5] Facebookresearch, "GitHub - facebookresearch/faiss: A library for efficient similarity search and clustering of dense vectors.," *GitHub*. <https://github.com/facebookresearch/faiss>
- [6] "React Reference Overview – React." <https://react.dev/reference/react>
- [7] "pytest documentation." <https://docs.pytest.org/en/stable/>
- [8] "OpenAI Developer Platform," *OpenAI*. <https://platform.openai.com/docs/overview>
- [9] "Index | Node.js v23.11.0 Documentation." <https://nodejs.org/docs/latest/api/>
- [10] "GitHub Actions," *GitHub*, 2025. <https://github.com/features/actions>
- [11] S. Ratliff, "Docker: Accelerated Container Application Development," *Docker*, Jan. 23, 2025. <https://www.docker.com/>
- [12] "Docker compose," *Docker Documentation*, Nov. 26, 2024. <https://docs.docker.com/compose/>
- [13] "SPARQL 1.1 Query language." <https://www.w3.org/TR/sparql11-query/>

### **XIII. Appendix A – Team Information**



Figure 8: Team Members: Molly Iverson (top left), Ethan Villalovoz (top right), Chandler Juego (bottom left), and Adam Strikman (bottom right)



## XIV. Appendix B - Example Testing Strategy Reporting

The testing strategy for the RAG Application Using Knowledge Graph and Vector Search was designed to ensure the accuracy, efficiency, and robustness of the system. The testing process included unit tests, integration tests, and manual testing to validate the functionality of core components, including NLP processing, vector search, knowledge graph querying, and LLM response generation.

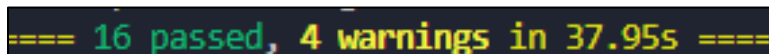
The tests were conducted using automated scripts and manual validation methods, covering various edge cases and performance evaluations.

Table 2: Testing Methodology

Test Type	Description	Tools Used
Unit Testing	Validates individual components like NLP handlers, vector search functions, and LLM query handling.	pytest, unittest, spaCy, FAISS, OpenAI API
Integration Testing	Ensures seamless interaction between NLP processing, vector retrieval, and LLM response generation.	API calls, system logs, pytest
Performance Testing	Measures system response time and efficiency for retrieving relevant information.	time, logging response times
Manual Testing	Verifies real-world performance by submitting test queries and validating responses.	CLI tests, user validation

The testing strategy focuses on validating key system requirements, including accurate responses, functioning custom dataset integration, efficient response time, smooth user interaction, and error handling. We implemented unit, integration, and system testing.

As illustrated in Figure 8, our automated test cases are now passing. System testing in Figures 9 and 10 show improvements in response time. Previously, calling the LLM would take 5-10 minutes and occasionally fail on certain computers. By using OpenAI, the process is now completed in seconds. While Figure 12 may give the impression that the final product is slower, it's important to note that earlier sprints did not embed the full Wikipedia dataset. Using the full dataset caused the text embeddings file to grow considerably in size, slowing vector search. Overall, the system is fast and effective.



```
==== 16 passed, 4 warnings in 37.95s ====
```

Figure 9: Passing Test Results

Component	Call 1 (ms)	Call 2 (ms)	Call 3 (ms)	Average (ms)
NLP Handler	552.30	346.40	552.00	483.57
Knowledge Graph	1137.30	1037.40	1231.70	1135.17
Vector Search	4418.30	7159.30	2973.20	4850.27
LLM	Not working	Not working	Not working	Not working
<b>Total</b>	<b>6107.9</b>	<b>8543.1</b>	<b>4756.9</b>	<b>6469.3</b>

Figure 10: System Response Time Before Sprint Four

Component	Call 1 (ms)	Call 2 (ms)	Call 3 (ms)	Average (ms)
NLP Handler	341.40	34.80	345.30	240.50
Knowledge Graph	1149.70	931.70	2112.30	1397.90
Vector Search	1924.70	1747.40	1753.50	1808.53
LLM	2405.30	2866.90	3245.50	2839.23
<b>Total</b>	<b>5821.90</b>	<b>5581.30</b>	<b>7457.50</b>	<b>6286.90</b>

Figure 11: System Response Time After Sprint Four

Final Stats				
Component	Call 1 (ms)	Call 2 (ms)	Call 3 (ms)	Average (ms)
NLP Handler	36.40	27.40	36.40	33.40
Knowledge Graph	1498.20	1392.70	1212.30	1367.73
Vector Search	3717.20	4672.20	4011.20	4133.53
LLM	2527.00	1884.80	1827.29	2079.70
<b>Total</b>	<b>7779.70</b>	<b>7977.60</b>	<b>7087.90</b>	<b>7615.07</b>

Figure 12: System Response Time in Final Product

## XV. Appendix C - Project Management

The team held weekly hour-long meetings to plan tasks for the upcoming week and weekly half-hour client meetings for status updates and technical advice. Additionally, the team met with our capstone professor, Dr. Parteek Kumar, four times during the semester to review project progress and address any concerns. All meetings were beneficial and provided great support for the team and the project. Below are more in-depth explanations of the types of meetings.

### Team Meetings

The team met virtually over Microsoft Teams to review current and upcoming issues for the sprint. We discussed which issues each teammate should pursue and the timeline for completion. We used a GitHub Kanban board to track tasks. Figure 11 shows the Kanban board, and Figure 12 gives an example of a detailed issue description.

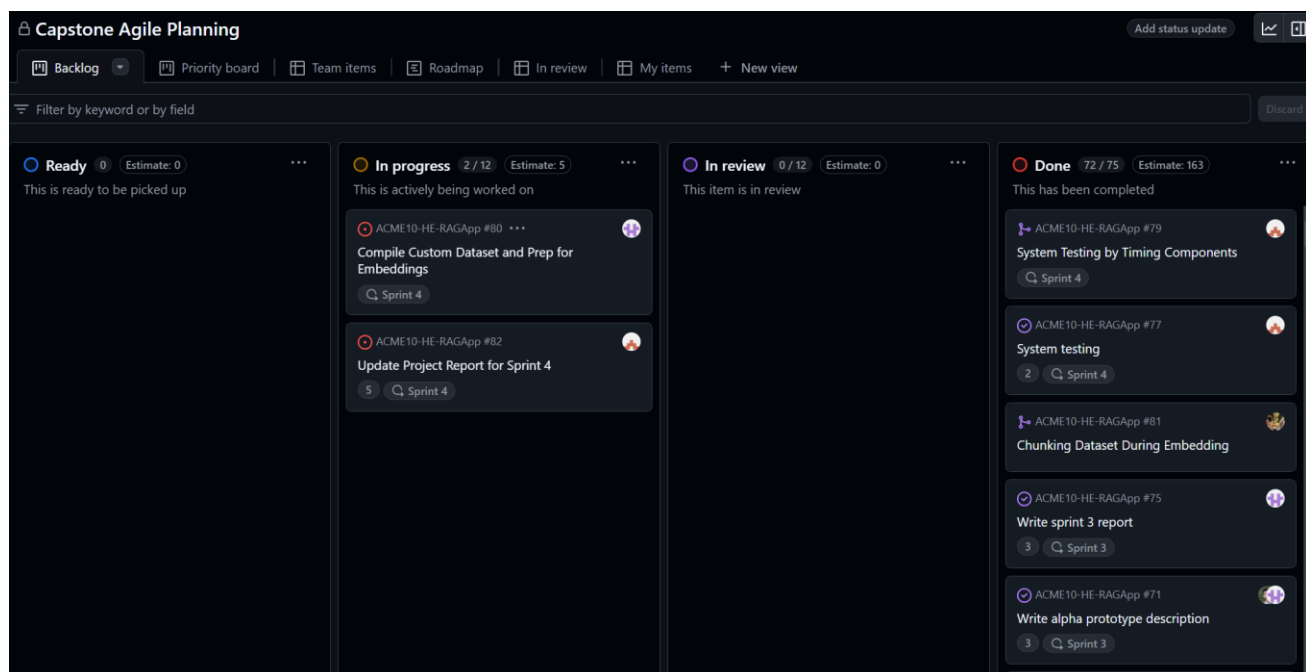


Figure 13: GitHub Kanban Board

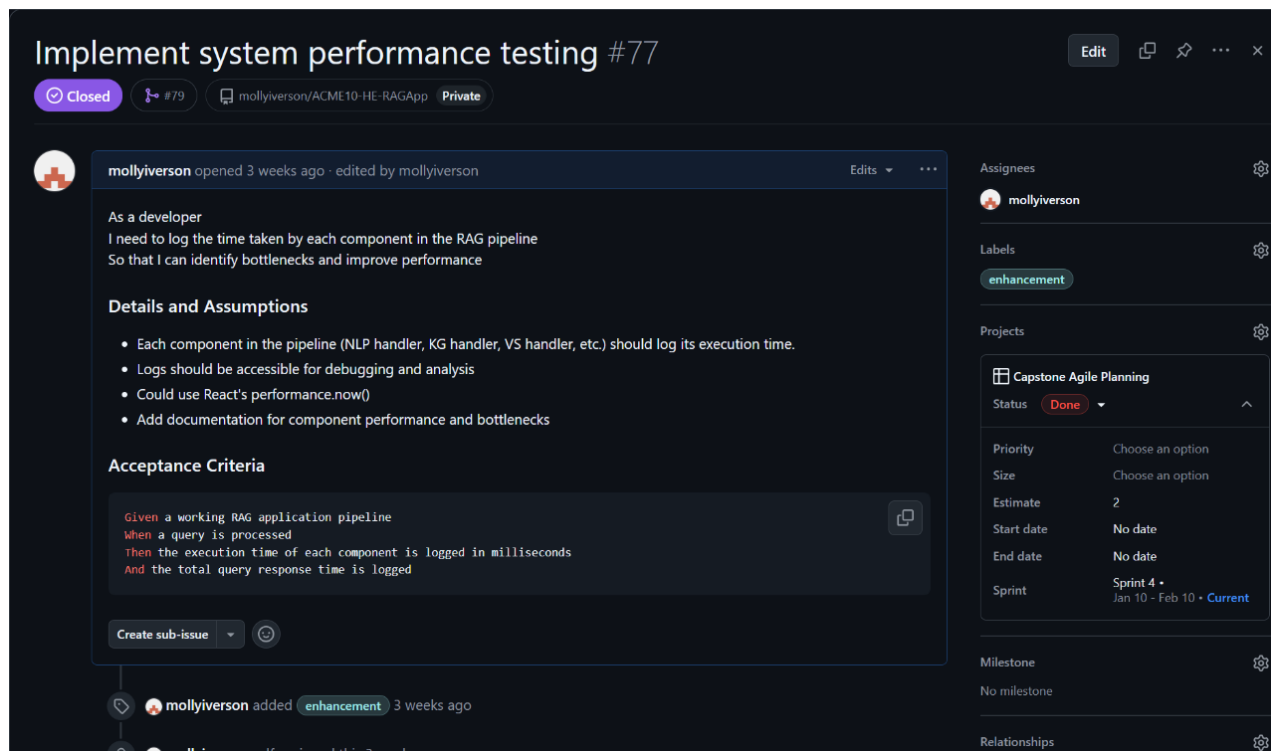


Figure 14: System Testing Kanban Issue

### Client Meetings

Every week, the team met with HackerEarth CEO Vikas Aditya over Zoom to discuss the week's progress and ask technical questions. We also communicated over email.

### Professor Meetings

Four times a semester, the team met with Dr. Parteek Kumar in his EME office to discuss project progress and documentation.