RAG Application Using Knowledge Graph and Vector Search

Final Report

HackerEarth

MECA Dynamics

Molly Iverson, Ethan Villalovoz, Chandler Juego, Adam Shtrikman

CptS 423 Software Design Project II

Spring 2025

Instructor: Parteek Kumar

TABLE OF CONTENTS

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), driving innovations across industries, from customer service chatbots to research assistants. One compelling application 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 positioned RAG systems as an essential tool in 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 novel approach that promises to revolutionize RAG systems. Knowledge graphs offer structured, contextual information that enhances the ability of RAG systems to understand and retrieve more accurate, fact-based responses. By integrating knowledge graphs with vector search, this project aims to develop an advanced RAG application that improves traditional methods by utilizing unstructured and structured data. The innovative nature of this project is sure to excite and intrigue those in the field of NLP and technology.

This project will use a large dataset of 10,000 Wikipedia articles to utilize a knowledge graph and vector search database. The system will handle user queries by retrieving semantically relevant information from the knowledge graph and vector search, enabling the generation of more precise and contextually appropriate responses. The motivation behind this project lies in the potential for knowledge graphs to revolutionize the retrieval aspect of RAG applications, ultimately improving the overall quality and utility of generated responses in various use cases, from conversational agents to research tools.

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. HackerEarth CEO Vikas Aditya (vikas.aditya@hackerearth.com) serves as 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, manages sprint tasks, and contributes by writing and integrating knowledge graph code into the RAG pipeline.

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 BERT for vector search for the RAG application project, creating testing strategies, 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 embeddings code, developing the front-end, 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, implementing FAISS for vector search, and integrating a CI/CD pipeline to streamline development and testing processes.

III.Project Requirements Specification

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. Additionally, this project will contribute to a research publication, showing our findings and advancements in this field.

III.2. Use Cases

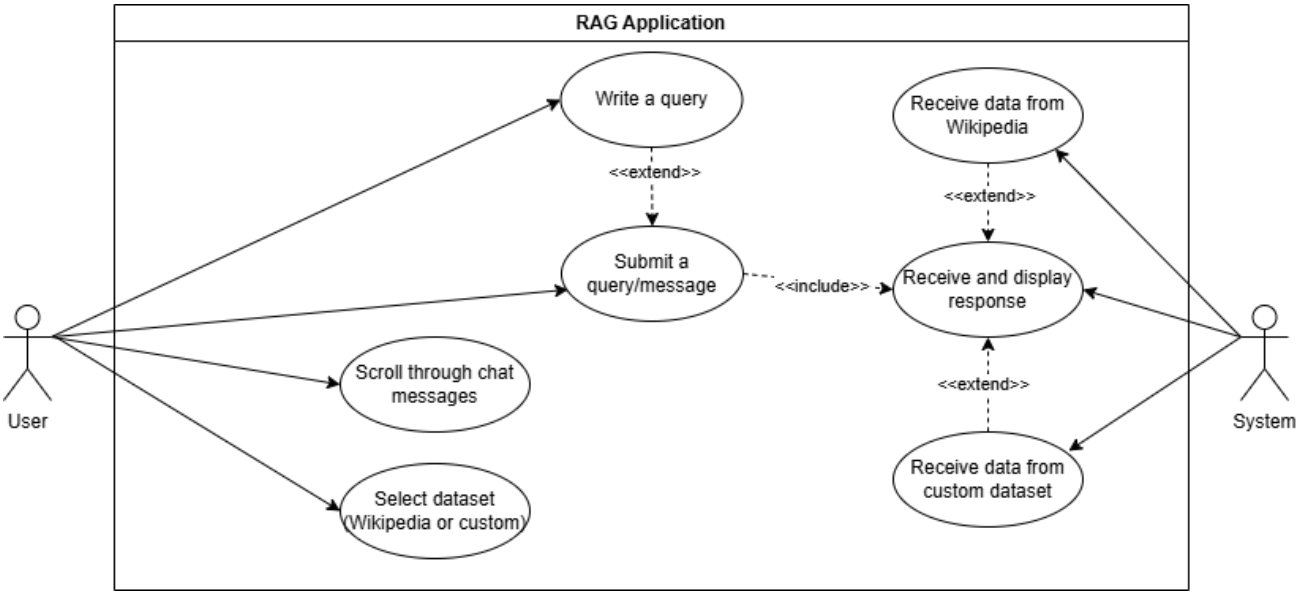

Figure 1: Use Case Diagram

Figure 1 shows the interactions between the user and the system of the RAG application. Each use case is listed below with a detailed pre-condition, post-condition, basic path, alternative path, and related requirements.

Use Case 1: Write a Query

Pre-condition	- The user has opened the web application - The query input field is visible and active
Post-condition	- The user has successfully entered a query in the input field
Basic Path	- The user clicks on the query input field - The user types a query or message in the input field - The system accepts and displays the query in the input field

Alternative Path	- 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 - 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
Related Requirements	- Chat Window

Use Case 2: Submit a Query/Message

Pre-condition	- The user has typed a query into the input field - The system is connected to the backend which is connected to the knowledge graph and vector search database
Post-condition	- The system successfully receives the query, initiating retrieval from the RAG model - The query appears in the chat interface as a sent message
Basic Path	- The user clicks on the “Submit” button or presses “Enter” to submit the query - The system records the query and forwards it to the RAG model for processing - The query appears in the chat window as a sent message from the user
Alternative Path	- If the user submits an empty query, then the system displays a message indicating that input is required - If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window. - 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
Related Requirements	- Chat Window - Content Moderation - Error Handling

Use Case 3: Receive and Display Response

Pre-condition	- The user has submitted a valid query, and the system has processed it using the RAG model
Post-condition	- The system returns a response based on the query and displays it in the chat window

Basic Path	- The system processes the query using the RAG model - The system retrieves relevant information - The system generates a response using the RAG model and formats the information in a readable message - The system displays the response in the chat window above the user input field
Alternative Path	- If the user submits an empty query, then the system displays a message indicating that input is required - If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window
Related Requirements	- Chat Window - Query Response Retrieval - Error Handling

Use Case 4: Select Dataset (Wikipedia or Custom)

Pre-condition	- The user has opened the web application
Post-condition	- The user has selected either the Wikipedia or custom dataset
Basic Path	- The user selects the custom dataset button, and vector search uses the custom dataset embeddings
Alternative Path	- The user selects the Wikipedia dataset button, and vector search uses Wikipedia embeddings - The user doesn't change the dataset, and the default Wikipedia dataset is used
Related Requirements	- Custom Dataset

Use Case 5: Scroll Through Chat Messages

Pre-condition	- The user has had an ongoing chat session with multiple messages exchanged between the user and the system
Post-condition	- The system successfully receives the query, initiating retrieval from the RAG model
Basic Path	- The user scrolls up in the chat window - The system dynamically loads previous queries and responses from the current session as the user scrolls - The user can review and scroll back to any message in the session

Alternative Path	- If the system encounters an issue loading older messages, a loading spinner is shown until the history is fully loaded - If the chat session is too long, the system may implement infinite scrolling to manage performance
Related Requirements	- Chat Window

Use Case 6: Receive Dataset from Wikipedia

Pre-condition	- The Wikipedia dataset is selected - The user has submitted a valid query, and the system has processed it using the RAG model
Post-condition	- The system returns a response based on the query and displays it in the chat window
Basic Path	- The system processes the query using the RAG model - The system retrieves relevant information - The system generates a response using the RAG model and formats the information in a readable message - The system displays the response in the chat window above the user input field
Alternative Path	- If the user submits an empty query, then the system displays a message indicating that input is required - If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window
Related Requirements	- Chat Window - Query Response Retrieval - Error Handling

Use Case 7: Receive Dataset from Custom Dataset

Pre-condition	- The custom dataset is selected - The user has submitted a valid query, and the system has processed it using the RAG model
Post-condition	- The system returns a response based on the query and displays it in the chat window
Basic Path	- The system processes the query using the RAG model - The system retrieves relevant information - The system generates a response using the RAG model and formats the information in a readable message - The system displays the response in the chat window above the user input field

Alternative Path	- If the user submits an empty query, then the system displays a message indicating that input is required - If the system encounters an issue with the query submission (e.g., network error), then an error message is displayed in the chat window
Related Requirements	- Chat Window - Query Response Retrieval - Error Handling - Custom Dataset

III.3. Functional Requirements

Each functional requirement is listed below with a detailed description, source, and priority level.

1. RAG Model Components

RAG Model Architecture:

Description	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.
Source	Project scope document provided by the client.
Priority	Priority Level 0: Essential functionality.

Vector Search Implementation:

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

Knowledge Graph Integration:

Description	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.

Source	Project scope document provided by the client.
Priority	Priority Level 0: Essential functionality.

2. Core App Functionality

Query Response Retrieval:

Description	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 doesn't have enough information to respond accurately; it should not lie.
Source	Project scope document provided by the client.
Priority	Priority Level 0: Essential functionality.

Custom Dataset:

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

3. User Interface

Chat Window:

Description	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.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 0: Essential functionality.

Error Handling:

Description	The system will handle errors like invalid queries, network issues, or app failures by displaying appropriate error messages in the chat window.

Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: Desirable functionality.

Content Moderation:

Description	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.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: 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 2 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 Retrieval-Augmented Generation (RAG) application, designed to enhance information retrieval and generation using knowledge graphs and vector search techniques. The purpose of this design document 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

Revise and include Section II from your Solution Approach report here. Provide the block diagram of your architecture and give a brief description of it.

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 Mode (LLM) Handler uses the LLaMA LLM to process user queries and generate an accurate natural language response. Each component interacts through well-defined interfaces, ensuring flexibility and scalability. The following sections will give more details on each subsystem and the design choices behind them.

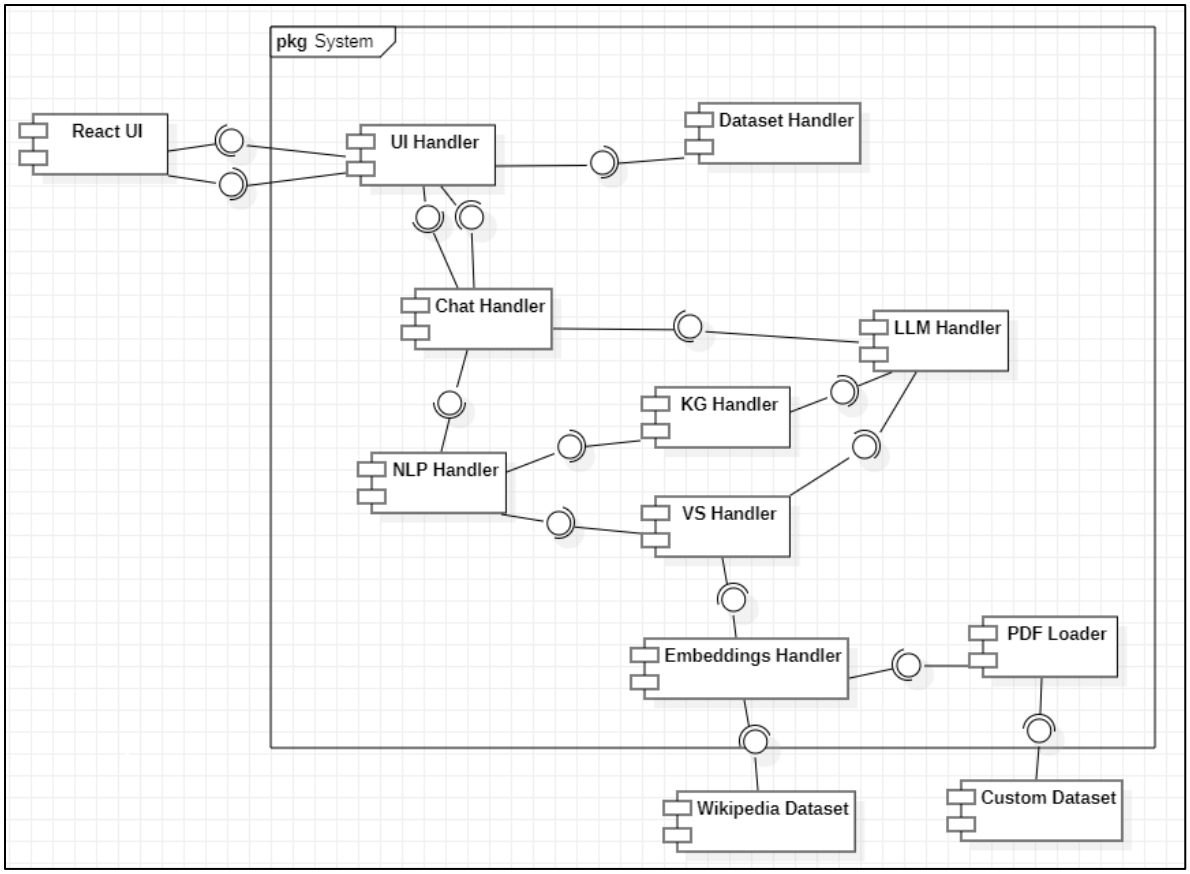

Figure 2: System Architecture and Component Data Flow

IV.1.2. Subsystem Decomposition

1.1. [UI Handler]

1.1.1. 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.

1.1.2. 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 to various other components. It maintains the state of the application using React, ensuring that the UI reflects the most recent data.

1.1.3. Interface Description

Services Provided:

Service Name	Service Provided To	Description

UpdateLayout	React UI, ChatHandler, Dataset Handler	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
SwitchDataset	Dataset Handler

1.2. [Chat Handler]

1.2.1. 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.

1.2.2. 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.

1.2.3. 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

1.3. [Natural Language Processing (NLP) Handler]

1.3.1. 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.

1.3.2. Concepts and Algorithms Generated

The NLP Handler uses several Natural Language Processing algorithms:

- Query Parsing and Preprocessing: Tokenizes and cleans the query for easier analysis
- Entity Recognition: 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
- Intent Detection: Determines the main intent of the query, whether it's asking for facts, definitions, or something more complex

1.3.3. 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

1.4. [Knowledge Graph (KG) Handler]

1.4.1. 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. Next semester, the KG handler will be enhanced to handle multiple entities and more complex queries.

1.4.2. 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.

1.4.3. 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

1.5. [Vector Search (VS) Handler]

1.5.1. Description

The VS Handler is responsible for performing vector search to retrieve semantically relevant information from the Wikipedia dataset. It uses FAISS to efficiently search 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.

1.5.2. 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.

1.5.3. 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

1.6. [LLM Handler]

1.6.1. 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 LLaMA large language model (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.

1.6.2. Concepts and Algorithms Generated

The LLM Handler takes the user query and the combined VS Handler and KG Handler results, sending them to the LLaMA model for processing. LLaMA 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.

1.6.3. Interface Description

Services Provided:

Service Name	Service Provided To	Description
GenerateResponse	Chat Handler	GenerateReponse queries LLaMA 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

1.7. [Embeddings Handler]

1.7.1. 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 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.

1.7.2. Concepts and Algorithms Generated

The Embeddings Handler uses BERT 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 BERT 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.

1.7.3. 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

1.8. [Dataset Handler]

1.8.1. Description

The Dataset Handler switches the dataset used to generate a response to the user's question. By pressing a button on the UI, the user can either use the Wikipedia dataset or a custom dataset, which is currently some class notes.

1.8.2. Concepts and Algorithms Generated

The Dataset Handler changes an environmental variable that represents the path to the current dataset. The user initializes this change by pressing a button on the screen.

1.8.3. Interface Description

Services Provided:

Service Name	Service Provided To	Description
SwitchDataset	UI Handler	SwitchDataset switches the dataset being used by the RAG app.

Services Required:

Service Name	Service Provided From
UpdateLayout	UI Handler

1.9. [PDF Loader]

1.9.1. Description

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

1.9.2. Concepts and Algorithms Generated

PDF Loader transforms the PDF documents into a parquet file and then generates embeddings for the custom dataset. PDF Loader indexes the data to prepare

1.9.3. 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 preliminary layout that mimics the structure of a typical chat or messaging system. 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.

Once entering our application, the user is introduced to the UI's main page, which is dominated by a chat box at the center, as shown in Appendix Section C UI Design (Figure 13). 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.

As development progresses, additional features will be incorporated into the UI design, including smooth transitions between session loading and user interaction, user-friendly icons for the navigation bar, and dynamic response visuals to enhance the user experience. We will continue to focus on maintaining a clean and minimalist design to avoid overwhelming the user while delivering powerful functionality.

V. Test Case Specifications and Results

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 are using 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.

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 have tested isolated handlers (LLM Handler, KG Handler, VS Handler, NLP Handler, etc.). We have run unit tests to ensure the accurate processing of inputs and expected outputs for each handler, mocking dependencies like the database and external libraries to isolate individual components. To ensure thorough coverage, we cover edge cases, boundary conditions, and performance. We use 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. Core functionalities should also be mentioned in the original project abstract. The team will evaluate the relevance and impact of all non-essential units and may make more tests.

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 NLP Handler, VS Handler, KG Handler, and LLM Handler to ensure accurate responses are generated. We started with pairs of handlers and slowly added more until the whole system was tested.

In the future, to verify the accuracy of answers provided by the VS, KG, and LLM Handlers, we will need to use Python libraries such as spaCy and NLTK to process the paragraph answers and calculate the similarity to our provided correct answers. We should assess the semantic similarity between the responses using techniques like cosine and Jaccard similarity to ensure answer relevance. Additionally, we plan to mock dependencies outside of the specific components being tested.

1.3. System Testing

System testing for the RAG application, developed for HackerEarth, 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 as outlined in the Requirements and Specifications document.

1.3.1. Functional Testing

The RAG application's functional testing will primarily rely on manual testing conducted by the development team. Each functional requirement from the Requirements and Specifications document will be paired with a corresponding functional test, ensuring that all essential features work as intended. This includes testing the core functionalities of query processing, vector search, and knowledge graph enrichment.

The testing will be performed iteratively, with developers manually validating the application's performance against the expected outputs. If a functional test fails, the testing developer will document the test conditions and notify the component's original developer to resolve the issue. Given the critical role of structured data integration, the team will continuously review and refine functional tests as the project evolves.

1.3.2. Performance Testing

Performance testing for the RAG application assesses its efficiency and resilience under various conditions, ensuring that the application meets non-functional requirements. Key performance metrics include:

- **Response Time:** Measuring the time taken to process queries and display results, ensuring that the system provides timely responses for an optimal user experience
- **Scalability:** Testing the system's ability to handle increased concurrent query loads and maintain performance stability
- **Stress Testing:** Extending the application beyond normal operational limits to identify potential breaking points and ensure robust error handling

The team will use profiling tools to monitor performance across the backend processes, capturing data on CPU, memory usage, and network latency. Non-functional requirements are evaluated qualitatively, leveraging developer insights to optimize efficiency and stability, particularly in high-load scenarios. While response time has been measured, the other performance metrics are still being tested, and stress testing is ongoing. This is not a major priority for our client, as we are not planning to deploy the application for a large number of users.

1.3.3. User Acceptance Testing

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

Testing Process:

- A functional version of the RAG application will be provided to our client

- Our client will interact with the system and test key features like query processing and response accuracy
- Feedback will be collected through discussion in our weekly meetings. The team will address client feedback in the final coding sprint

Although the RAG application will not be deployed permanently, we might host it on an EC2 instance for demonstration purposes. UAT will help us confirm that the system meets the agreed-upon requirements and works 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.

1.1. Hardware Requirements

Development workstations should have at least 16GB of RAM and 4 CPU cores to handle the necessary computational tasks. Additionally, servers for testing environments need 32GB of RAM and scalable storage options to manage the knowledge graph and vector embeddings efficiently.

1.2. 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 including FAISS for vector search, SPARQL libraries for knowledge graph queries, and PyTest for testing. The frontend should be built using React, accessible through a modern web browser.

1.3. Network and Database

Stable internet connectivity is essential for handling live queries to knowledge graph sources. Vector and knowledge graph databases should be hosted on cloud infrastructure such as AWS or Google Cloud to ensure high availability and scalability.

1.4. CI/CD and Testing Tools

GitHub Actions is used for CI/CD to automate the testing process. JMeter will be utilized for performance testing to simulate various load conditions.

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	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 - Tokenization	The input query is correctly tokenized into meaningful components.	Tokenized words match expected segmentation.	Pass	Ensure spaCy or other NLP libraries are properly installed.
DBpedia Query Execution	The system correctly generates and executes SPARQL queries against DBpedia.	Query execution returns relevant information.	Pass	DBpedia endpoint must be accessible.
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.

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
FAISS	FAISS allows for efficient similarity search for fast vector search retrieval.
DBpedia	DBpedia is a knowledge graph that extracts information from Wikipedia.
React	React is a frontend framework for building the user interface.
Pytest	Pytest is a testing framework for ensuring code reliability.
OpenAI	OpenAI is the LLM used for text generation and retrieval.
Node.js	Node.js is the backend runtime environment used for handling API requests and server logic.
GitHub Actions	GitHub Actions for CI/CD were used to automate the testing process.

Languages Used in Project			
Python	Typescript	JavaScript	SPARQL

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. Although it's not planned for permanent deployment, it might be hosted on an EC2 instance for demonstration purposes.

1.1. 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.

1.2. User Guide

Below are the steps to run the RAG application.

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

Currently, we are adding a new feature where users can query a custom dataset. Users will be able to switch between Wikipedia and custom datasets with a button on the screen.

1.3. 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 is tested with class notes, 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.

1.4. Prototype Screenshots

The prototype displays messages for each step in the generation process: natural language processing, querying the DBpedia knowledge graph, vector search, and the final LLM response. When the project is complete, only the final response will be displayed.

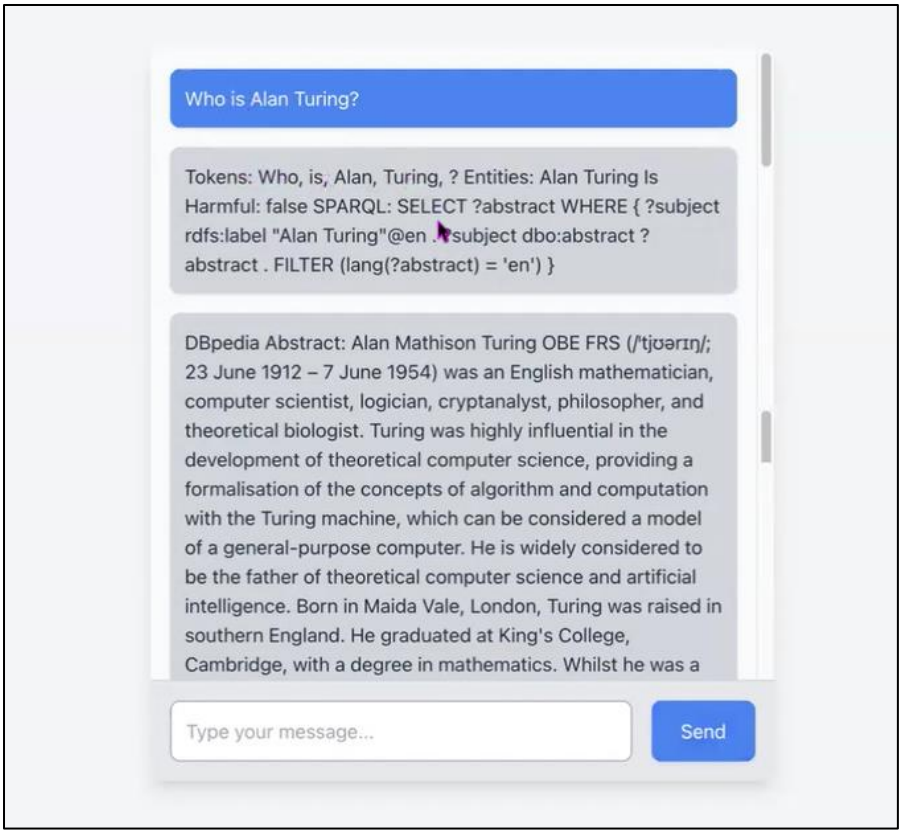

Figure 3: Tokenization and Knowledge Graph Steps

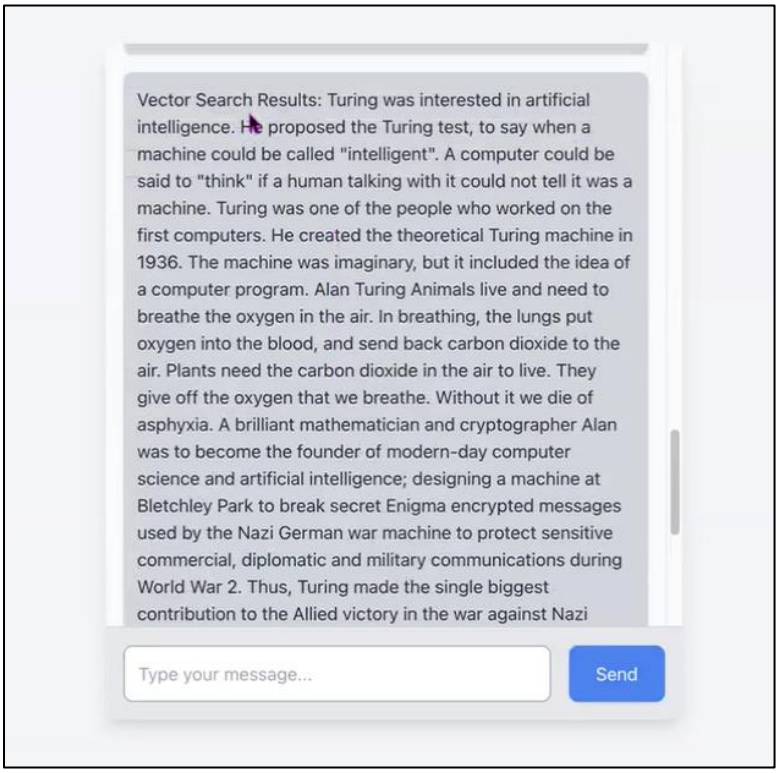

Figure 4: Vector Search Results

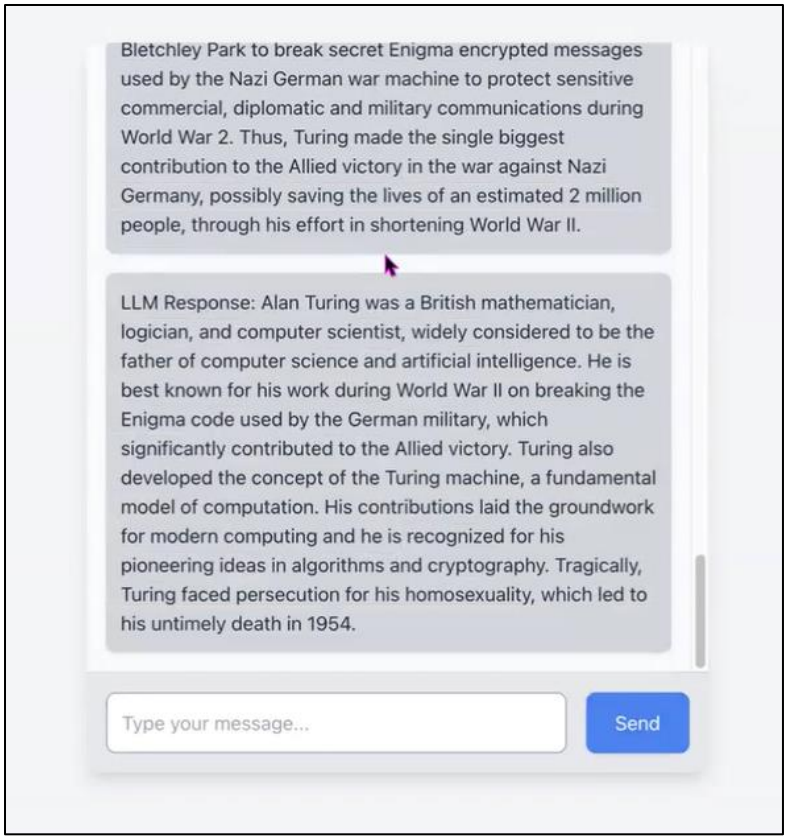

Figure 5: Final LLM Response

```python
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 6: 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 may temporarily deploy it on a website for testing and demonstration purposes during the Poster Presentation on April 22nd, 2025. For our final client demo on April 18th, 2025, we will run the project on an EC2 instance. Deployment plans and code will be completed by Sprint 5 on March 10th, 2025.

The project is available on the team's GitHub, with detailed setup instructions included 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 rebuild the project, users need Python, Node.js, and the dependencies listed in the GitHub repository. The repository contains detailed setup instructions for installing dependencies, running the backend, and launching the frontend.

IX. Conclusions and Future Work

IX.1. Limitations and Recommendations

One of the main challenges in our current prototype is testing the response quality and relevance. We plan to incorporate quality testing in our final coding sprint, however, evaluating response accuracy and relevance is complex and could always be improved. Potential solutions include automated benchmarking using established datasets, human manual testing, and refining 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 understands the difficulty of building high-quality natural language processing and has 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 6286.90 ms, which is slower than optimal. We have switched the LLM from the LLaMA 2.7b model to OpenAI, significantly improving latency from around 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

The majority of the development of this project is intended to be completed by the end of the second semester of the Capstone project. The team will explore the opportunity of writing a research publication about our project and findings during the final sprint.

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] P. Lewis et al., “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in Curran Associates Inc., Vancouver, BC, Canada, 2020, pp. 9459–9474. doi: https://dl.acm.org/doi/abs/10.5555/3495724.3496517.

[2] K. Guu et al., “Traversing Knowledge Graphs in Vector Space”, Stanford NLP Group – Standford University, 2015. Available: https://nlp.stanford.edu/pubs/kg_traversal.pdf

[3] A. Kollegger, “Knowledge Graphs for RAG,” DeepLearning.AI. [Online]. Available: https://www.deeplearning.ai/short-courses/knowledge-graphs-rag/

[4] L. Voss, “JavaScript RAG Web Apps with LlamaIndex,” DeepLearning.AI. [Online]. Available: https://www.deeplearning.ai/short-courses/javascript-rag-web-apps-with-llamaindex/

[5] J. Liu and A. Datta, “Building and Evaluating Advanced RAG Applications,” DeepLearning.AI. [Online]. Available: https://www.deeplearning.ai/short-courses/building-evaluating-advanced-rag/

[6] “Generative AI with Large Language Models,” Coursera. [Online]. Available: https://www.coursera.org/learn/generative-ai-with-llms

XIII. Appendix A – Team Information

Figure 7: 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.

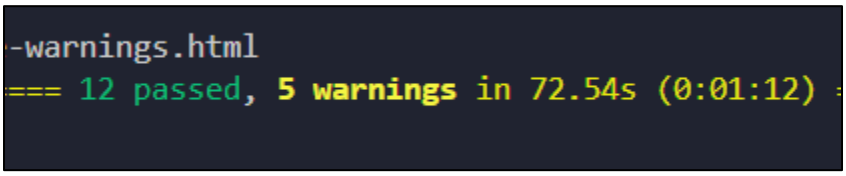

Figure 8: Passing Test Results

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
Total	6107.9	8543.1	4756.9	6469.3

Figure 9: 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
Total	5821.90	5581.30	7457.50	6286.90

Figure 10: System Response Time After Sprint Four

As shown in Figure 8, our automated test cases are passing. Additionally, Figures 9 and 10 show significant response time improvements. The time to call the LLM used to take 5-10 minutes and did not work on some computers. Now it takes seconds using OpenAI instead.

XV. Appendix C - Project Management

The team holds 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 meets with our capstone professor, Dr. Parteek Kumar, four times during the semester to review project progress and address any concerns. All meetings are beneficial and provide great support for the team and the project. Below are more in-depth explanations of the types of meetings.

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

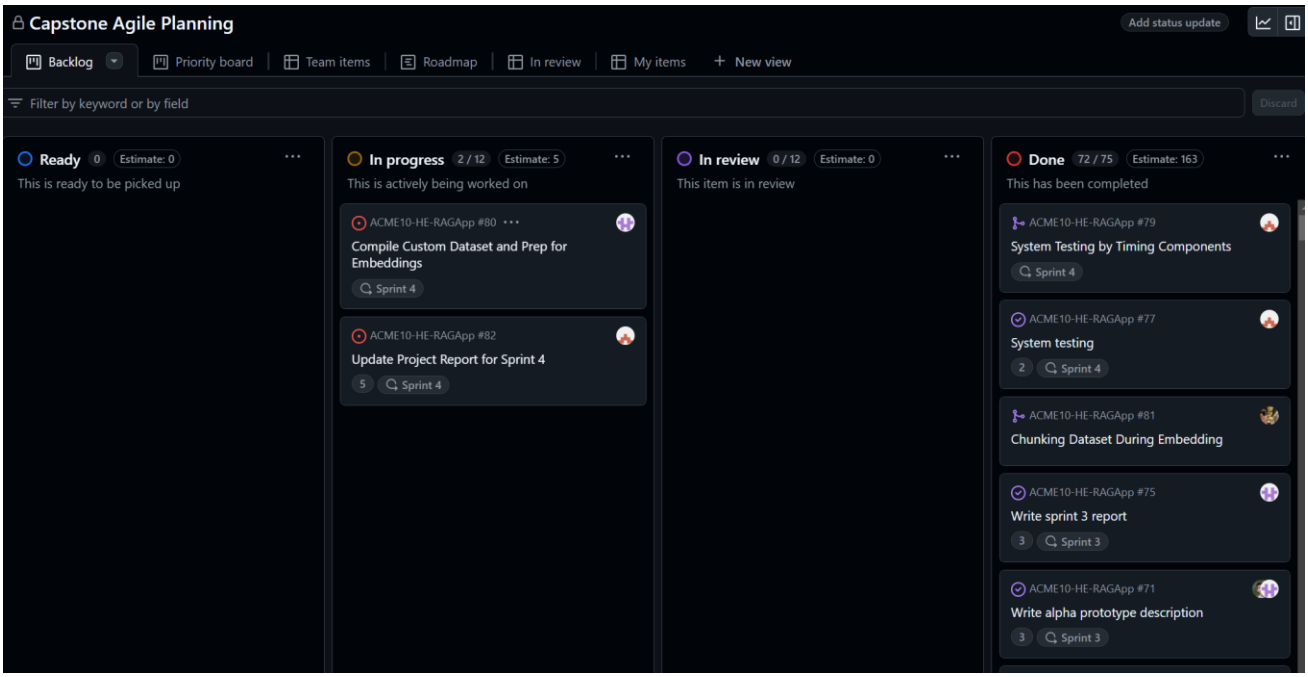

Figure 11: GitHub Kanban Board

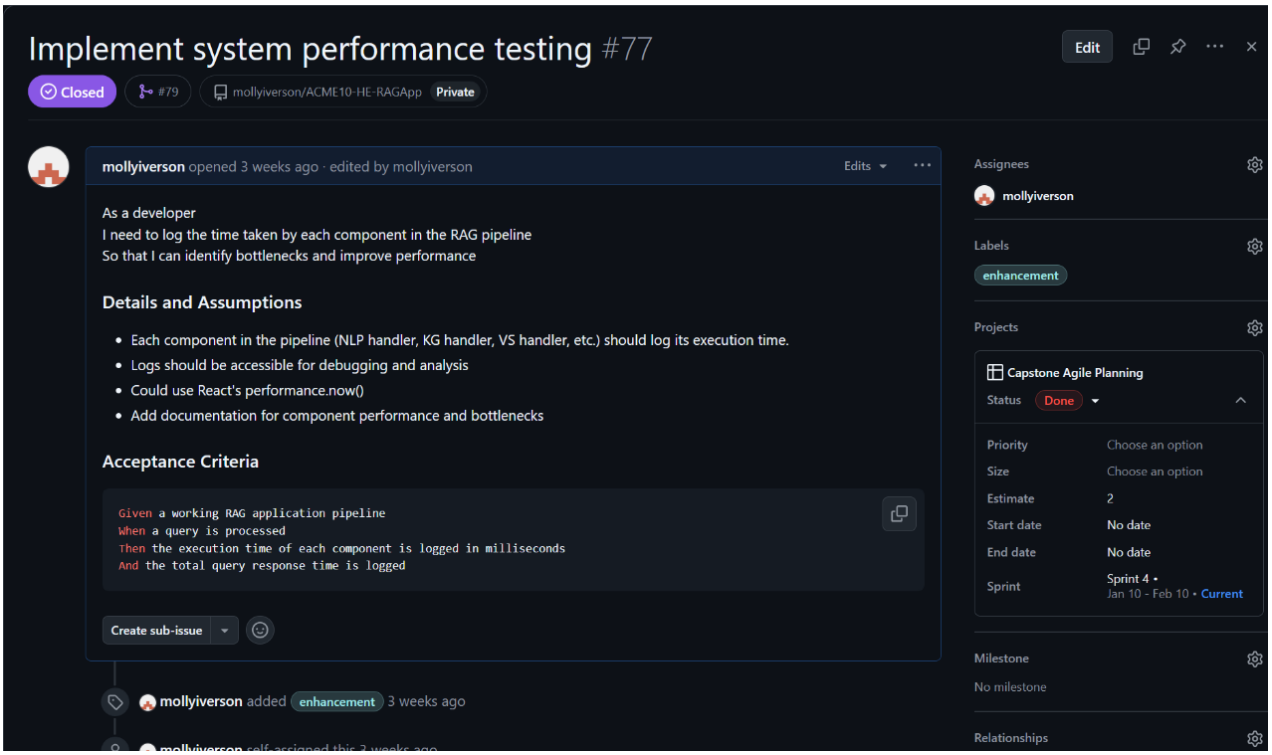

Figure 12: System Testing Kanban Issue

Client Meetings:
Every week, the team meets with HackerEarth CEO Vikas Adiyta over Zoom to discuss the week's progress and ask technical questions. We also communicate over email.

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

XVI. Appendix C – UI Design

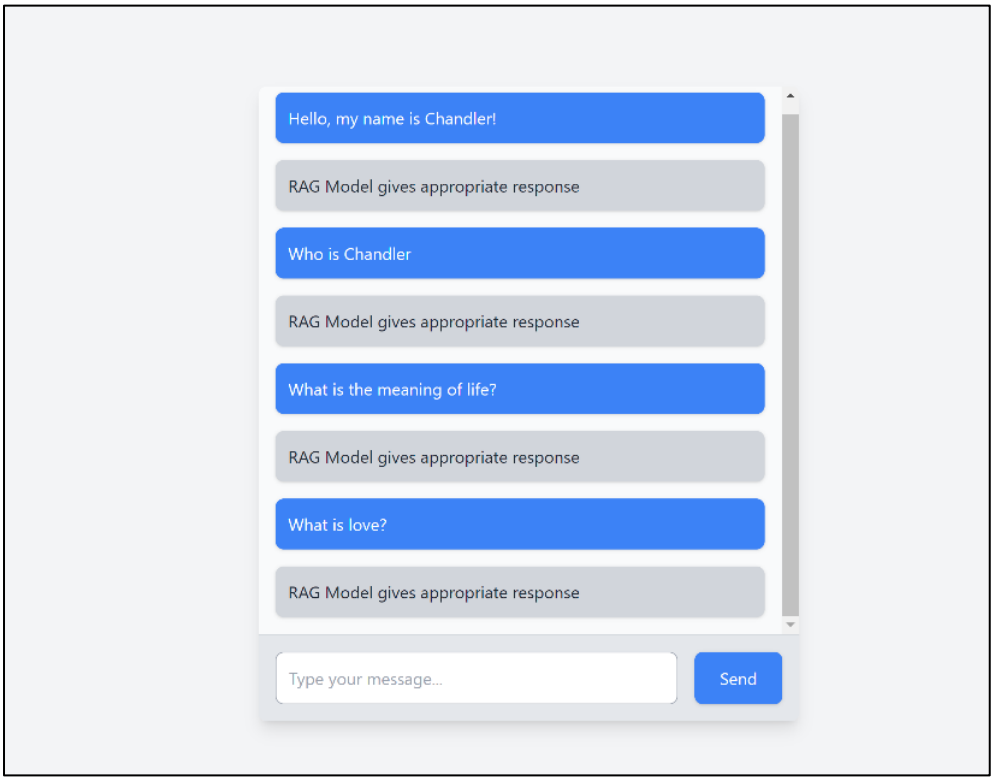

Figure 13: UI Design of Chat Box