RAG Application Using Knowledge Graph and Vector Search

Prototype Project Report

HackerEarth

MECA Dynamics

Ethan Villalovoz, Molly Iverson, Adam Shtrikman, Chandler Juego

Table of Contents

Introduction

I. Project 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.

II. Background and Related Work

To evaluate current leaders in the same domain as our project, we must first define what that domain is. For this document, we have narrowed down this definition to “knowledge-intensive natural language processing tasks.” Our project is at the intersection of information retrieval, natural language generation, and structured knowledge representation through knowledge graphs. Given this definition, our research highlights a representative example of a highly successful existing contribution in the domain.

The most notable and relevant to our work is the research by P. Lewis et al. (2020) from Facebook/Meta on Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [1]. Their model has two main components: a retriever, which fetches relevant documents based on a query, and a generator, which uses those documents to create responses. RAG has proven successful for open-domain question-answering and retrieving accurate information from large datasets. According to this study, dense vector search significantly improves performance compared to standard generation models. However, their method only uses vector search and does not take advantage of the structure and insights provided by knowledge graphs. While pure vector search mechanisms can capture similarity or relatedness between data, they do not provide the logical deduction and robust inferencing that knowledge graphs can, as shown by K. Guu et al. [2]. At the cost of additional complexity due to the structure of knowledge graphs, a

combination of vector search and knowledge graphs can enhance both semantic retrieval and structured reasoning. By leveraging the strengths of vector embeddings for approximate similarity search alongside the explicit relationships in a knowledge graph, this hybrid approach can offer more precise, explainable, and context-aware results. Our project builds on this by integrating knowledge graphs into the RAG pipeline, aiming to improve retrieval accuracy and response quality, especially for complex, multi-step queries.

To successfully execute the technical aspects of this project, our team must master several concepts, tools, and frameworks. A strong understanding of RAG, vector search, and knowledge graphs is critical. Before any coding starts, it's essential to practice building a knowledge graph from a small set of text documents, adding relationships between nodes, adding vector indices and embeddings, implementing vector search to retrieve relevant information from the data, and integrating an LLM (e.g. OpenAI) to generate a response from the search results. To accomplish this, we plan to take several online courses, including Knowledge Graphs for RAG [3], JavaScript RAG Web Apps with LlamaIndex [4], Building and Evaluating Advanced RAG Applications [5], and Generative AI with Large Language Models [6]. These courses will equip us with the necessary skills to effectively integrate vector search and knowledge graphs into our application. Furthermore, we will explore existing LLMs to understand their implementation and how they can be optimized for generating responses based on the retrieved data.

III. Project Overview

We are currently in a time where AI-driven technologies and solutions are rapidly evolving. This has created a new demand for efficient information retrieval and generation systems in various industries. This application will utilize a dataset of 10,000 Wikipedia articles to build a knowledge base using the knowledge graph architecture combined with vector search algorithms, enabling more contextually rich and accurate responses.

While current RAG applications rely on vector search, new research suggests that incorporating knowledge graphs can greatly enhance performance. While vector search can retrieve semantically similar data points, it lacks the contextual relationships offered by knowledge graphs. This project aims to integrate knowledge graphs into RAG architecture to enhance relevance, accuracy, and contextual depth.

The difficulty is in integrating the benefits of vector search, which can retrieve semantically relevant content from large unstructured datasets, with the structured and relational data inherent to knowledge graphs. This project will focus on querying both vector search and knowledge graphs databases.

The primary objective of this project is to build a RAG application that combines vector search and knowledge graphs. Users will be able to interact with the RAG model through a full-stack web application with a chat-style frontend and a backend supported by JavaScript, featuring data persistence.

The key milestones of this project are outlined below.

- **Initial Research and Learning:** The team will study RAG model architecture (retriever and generator), knowledge graphs (creation, querying, integration into natural language processing tasks), vector search fundamentals, and tools relevant to implementation.

- **Data Collection and Knowledge Graph Construction:** The team will gather a dataset of Wikipedia articles and utilize an existing knowledge graph.
- **Implementation of Vector Search:** The team will generate embeddings for data points using pre-trained models, index them using a vector search library, and implement a retrieval mechanism.
- **Integration of Components:** The team will connect the vector search to the knowledge graph and the vector search to the large language model to create structured responses.
- **Application Development:** The team will implement the logic for query processing, retrieval, and response generation by inputting the query into the command line and outputting the response to a text file.
- **Testing and Optimization:** The team will test various inputs to assess performance and refine the retrieval algorithm, knowledge graph integration, and generator model to optimize output results.
- **Documentation and Final Presentation:** The team will formulate documentation throughout the project, focusing on code, methodologies, and important findings. This will culminate in a final presentation showcasing the application and key takeaways.

IV. Client and Stakeholder Identification and Preferences

Our primary client is HackerEarth, where Vikas Aditya, the company's CEO, will be our main point of contact. We will work closely with him and HackerEarth to ensure that the RAG application we develop meets the company's expectations and serves as a strong example of how knowledge graphs and vector search can enhance retrieval-augmented generation systems.

While the immediate goal is to create a well-functioning RAG application based on thousands of Wikipedia articles, the broader aim is to deliver a solution that HackerEarth can potentially use as inspiration for future projects. In addition to the core RAG application, a full-stack web app will be developed to streamline the process of making queries and receiving clear responses. By integrating an intuitive user interface with the power of knowledge graphs and vector search, the app will enable an effortless interaction for querying and retrieving relevant information. The application will be built to demonstrate best practices in combining knowledge graphs and vector search, providing HackerEarth with a reference point for future AI and data retrieval initiatives.

HackerEarth's research and engineering teams can use this project to explore how such technologies can be applied across different domains. The project needs to be designed in such a way that it makes it easier for future teams to experiment with similar technologies or adapt the methodology for other use cases. By maintaining a clear line of communication and focusing on delivering a solution that is both effective and easy to understand, we aim to create a RAG application that not only meets current goals but also opens the door for potential future use in various company projects.

Project Requirements

V. System Requirements Specification

The System Requirements Specification details the key functional and non-functional requirements for the RAG (Retrieval-Augmented Generation) application, integrating knowledge graphs and vector search to enhance response accuracy. Functional requirements cover essential system features such as RAG model components, query processing, and user interface functionalities. Non-functional requirements outline system constraints related to performance, scalability, and security.

This section also includes use cases that demonstrate real-world interactions between users and the system. Each use case traces back to relevant requirements, ensuring a cohesive system design that aligns with user expectations and technical specifications. The document establishes a roadmap for developing and refining the application, ensuring it meets its functional and performance goals.

V.1. Functional Requirements

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

V.1.1. RAG Model Components

[FR-1] 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.

[FR-2] Vector Search Implementation

Description	The system will implement vector search to retrieve relevant information based on user queries. This includes generating embeddings, 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.

[FR-3] 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.

V.1.2. Core App Functionality

[FR-4] 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.

[FR-5] User Registration

Description	The system will allow users to create an account by providing a username, password, and email address. This involves a user-friendly interface for registration, input validation, and secure storage of user data.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: Desirable functionality.

[FR-6] Saving and Loading

Description	The system will automatically save the last 20 chat sessions and display them as clickable titles on the side of the chat window. Users should be able to load previous sessions for reference.
Source	Internal requirements elicitation among members of the team.
Priority	Priority Level 1: Desirable functionality.

V.1.3. User Interface

[FR-7] 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.

[FR-8] 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.

[FR-9] 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.

V.2. 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
[NFR-1] 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.
[NFR-2] Response Time	The system will respond to user queries within 2 seconds for typical requests, ensuring a smooth user experience.
[NFR-3] Data Storage	The system will maintain sufficient storage for the knowledge graph and vector embeddings, ensuring persistent and reliable data retrieval.
[NFR-4] Platform Compatibility	The RAG application will be accessible through a web browser, making it platform-independent and usable on various operating systems.
[NFR-5] Maintainability	The system will be designed with maintainability and modular code that allows for future updates and extensions without significant refactoring.
[NFR-6] Reliability	The system will maintain a 99% uptime, ensuring high availability for users, particularly during the testing and demonstration phases.
[NFR-7] 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.
[NFR-8] 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.

V.3. Use Cases

The use cases describe common scenarios of user interactions with the system, explaining how various functional requirements are applied in specific situations. The use cases of the proposed are represented in the use case diagram given in figure 1.

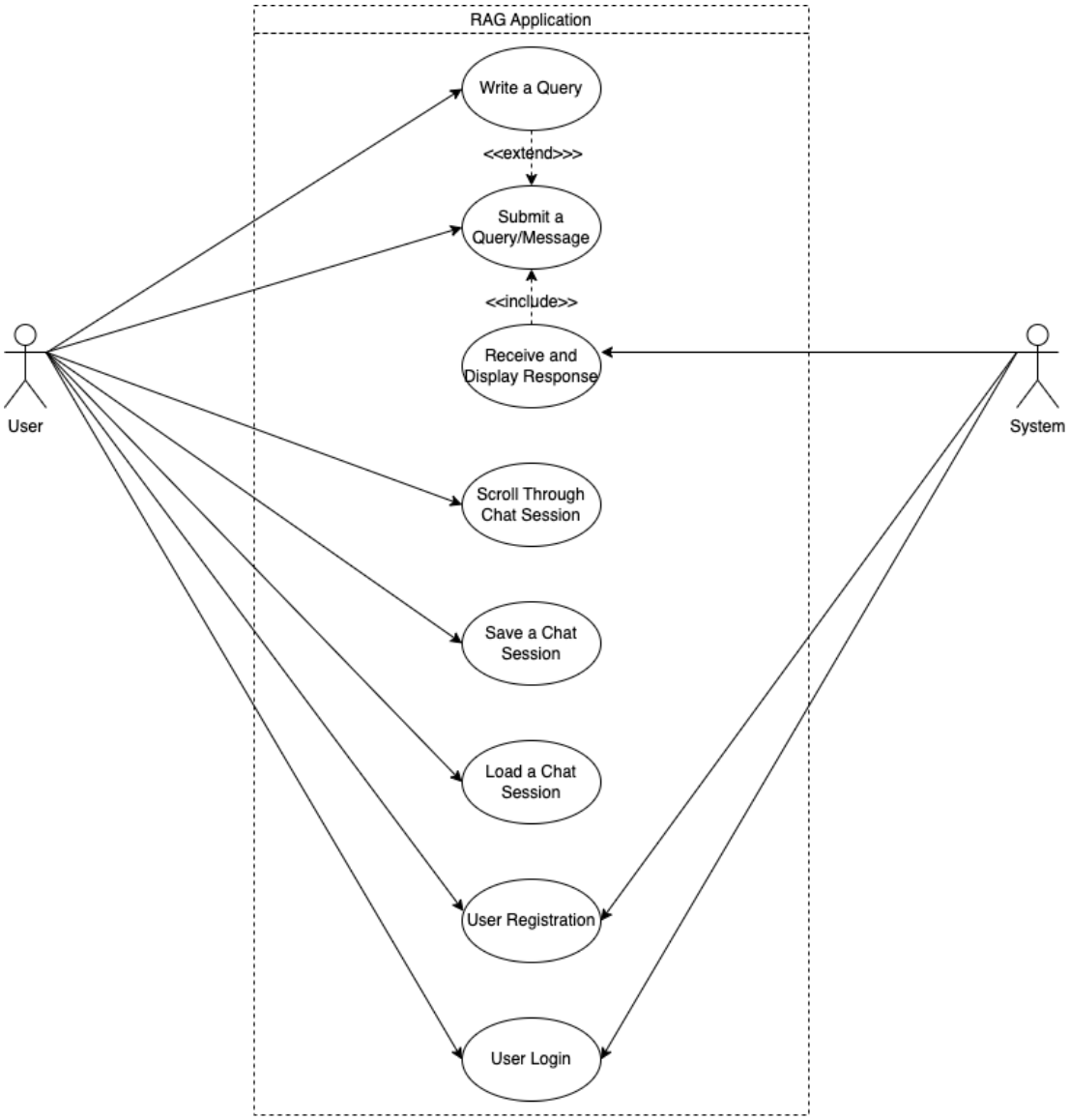


Figure 1: Use case diagram

The description of each use case is given below.

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	- [FR-7] 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	- [FR-7] Chat Window

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	- [FR-7] Chat Window - [FR-4] Query Response Retrieval

Use Case 4: Scroll Through the Current Chat Session History

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	- [FR-7] Chat Window - [FR-6] Saving and Loading

Use Case 5: Save a Chat Session

Pre-Condition	- The user has completed a chat session and wants to save the conversation for future reference.
Post-Condition	- The system saves the chat session and stores it under the user's profile for later access.
Basic Path	- The user clicks the "Save Session" button at the end of the chat. - The system prompts the user to name or label the chat session. - The system saves the chat session with the provided name and stores it in the user's session history. - A confirmation message is displayed to the user indicating the session was successfully saved.
Alternative Path	- If the system encounters an error during the saving process, the user is notified and asked to try saving again later. - If the user does not provide a session name, the system automatically saves the session with a default timestamp-based name.
Related Requirements	- [FR-6] Saving and Loading

Use Case 6: Load a Chat Session

Pre-Condition	- The user has previously saved chat sessions and wants to load a past conversation.
Post-Condition	- The user is able to view and interact with a previously saved chat session.
Basic Path	- The user navigates to the "Saved Sessions" section in the application. - The system displays a list of saved chat sessions, each labeled with the session name and timestamp. - The user selects a session to load. - The system loads the selected chat session in the chat window. - The user can scroll through and review the saved conversation.
Alternative Path	- If the selected session fails to load, the system shows a pop-up error message and offers the user a chance to retry. - If the user has no saved sessions, the system displays a message "No saved sessions" in the saved sessions section of the UI.

Related Requirements	- [FR-6] Saving and Loading

Use Case 7: User Registration

Pre-Condition	- The user is on the homepage of the RAG web application and is not logged in. The registration form is available to new users.
Post-Condition	- The user successfully creates an account, and their details are stored in the system. The user is now logged in and is able to access logged-in features such as saving and loading chat histories.
Basic Path	- The user clicks on the ‘Sign Up’ or ‘Register’ button. - The system displays a registration form requesting information such as username, email, and password. - The user fills in the required details and submits the form. - The system validates the provided information (e.g., checking if the email is already in use). - The system creates a new user account, saves the user details in the database, and logs the user into the application. - A confirmation message or welcome page is displayed, and the user is redirected to the application’s main page.
Alternative Path	- If the user submits incomplete or invalid information (e.g., invalid email format, weak password, etc.), the system highlights the errors and prompts the user to correct them. - If the username or email is already registered, the system displays an error and asks the user to choose a different email or username. - If the system encounters a database or server error during registration, a generic error message is shown, and the user is advised to try again later.
Related Requirements	- [FR-5] User Registration

Use Case 8: User Login

Pre-Condition	- The user is on the homepage or login page of the RAG web application and already has an account registered.
Post-Condition	- The user successfully logs into the application and is redirected to the main page where they can begin to use the application.
Basic Path	- The user clicks on the "Login" button or navigates to the login page.

	- The system displays a login form requesting the user's email/username and password. - The user enters their email/username and password and submits the form. - The system validates the credentials by checking the information against the stored user database. - If the credentials are valid, the system logs the user in and redirects them to the main page. - A success message is shown briefly, confirming the login.
Alternative Path	- If the user leaves any field blank, the system prompts the user to fill in the missing fields. - If the entered email/username or password is incorrect, the system displays an error message (e.g., "Invalid username or password") and prompts the user to try again. - If the user has forgotten their password, the system provides a "Forgot Password" link that directs them to a password recovery process. - If there is a server/database error during login, a generic error message is shown, and the user is asked to try again later.
Related Requirements	- [FR-5] User Registration

V.4. User Stories

The following user stories describe specific actions a user can take in the system, detailing what the user wants to accomplish and why. Each user story specifies system behavior clearly.

User Story 1: Asking Questions

As a user, I want to ask the RAG application questions to receive accurate and useful information.

Feature: Knowledge Graph and Vector Search Integration

Scenario: Successful query retrieval using vector search and knowledge graph

- **Given**: I have a query, "What is the capital of France?"
- **When**: I input the query into the RAG application
- **And**: The system retrieves relevant data from the vector search and knowledge graph
- **Then**: I should receive a generated response, "The capital of France is Paris."

Scenario: Unsuccessful retrieval due to lack of relevant information

- **Given**: I have a query, "Who is the 80th President of the United States?"
- **When**: I input the query into the RAG application
- **And**: The system performs a vector search and knowledge graph lookup
- **Then**: I should receive a response, "There have only been 46 US presidents so far."

- **And**: I should be prompted to refine my query with a helpful suggestion such as "Would you like to ask about the current president?"

User Story 2: Complex Questions

As a user, I want to ask the RAG application complex questions to receive accurate and useful information.

Feature: Querying with Complex Relationships in Knowledge Graph

Scenario: Querying hierarchical data in the knowledge graph

- **Given**: I have a query, "Who succeeded Abraham Lincoln as the president of the USA?"
- **When**: I input the query into the RAG application
- **And**: The system queries the knowledge graph for the successor relationship
- **Then**: I should receive the generated response, "Andrew Johnson succeeded Abraham Lincoln."

Feature: Multi-step Query Resolution

Scenario: Successful resolution of a multi-step query

- **Given**: I have a query, "List the top 3 most populated cities in the USA"
- **When**: I input the query into the RAG application
- **And**: The system retrieves population data from the knowledge graph and vector search
- **Then**: I should receive the response, "The top 3 most populated cities in the USA are New York, Los Angeles, and Chicago."

User Story 3: Ambiguous Questions

As a user, I want to ask the RAG application ambiguous questions so I can receive the clarification I need.

Feature: Handling Ambiguous Queries

Scenario: System asks for clarification on ambiguous queries

- **Given**: I have a query, "Tell me about Paris"
- **When**: I input the query into the RAG application
- **And**: The system identifies multiple possible meanings for "Paris" (e.g., Paris, France or Paris, Texas)
- **Then**: I should be asked, "Do you mean Paris, France or Paris, Texas?"
- **And**: I should be able to select the correct option before receiving the final response
- **And**: After selecting "Paris, France," I should receive information specific to Paris, France.

User Story 4: Technical Issues

As a user, I want the RAG application to notify me if it encounters technical issues during data retrieval.

Feature: Handling System Errors

Scenario: Failed data retrieval due to system issues

- **Given**: I input a query into the RAG application
- **When**: The system is unable to retrieve relevant data due to connectivity or internal issues
- **Then**: I should receive an error message stating, "Unable to retrieve information at this time. Please try again later."

Scenario: Successful retry after initial failure

- **Given**: I input a query after a previous failed attempt
- **When**: The system successfully retrieves relevant data on the retry attempt
- **Then**: I should receive the correct response to my query without further issues.

User Story 5: Review/Access History

As a user, I want the RAG application to allow me to review and access my query history.

Feature: Query History Management

Scenario: Viewing past queries and responses

- **Given**: I have previously submitted several queries to the RAG application
- **When**: I navigate to the query history section of the application
- **Then**: I should be able to see a list of all past queries and responses with timestamps
- **And**: I should be able to click on a previous query to view its response in detail.

Scenario: Re-using a previous query

- **Given**: I want to re-use a query I previously submitted
- **When**: I select a query from my query history
- **Then**: I should be able to submit the same query again without retyping it
- **And**: The system should generate a new response based on updated data, if applicable.

V.5. Traceability Matrix

The table below maps functional requirements to their respective use cases and user stories. This ensures that all requirements are accounted for and linked to user scenarios.

Functional Requirement	Use Case	User Story	Priority
FR-4: Query Response Retrieval	**UC-1**: Write a Query	**US1**: As a user, I want to ask the RAG application questions for information.	Level 0
FR-1: RAG Model Architecture	**UC-2**: Submit a Query/Message	**US1**: Successful query retrieval using vector search and knowledge graph.	Level 0

FR-3: Knowledge Graph Integration	**UC-3**: Receive and Display Response	**US2**: As a user, I want to receive responses to complex questions.	Level 0
FR-3: Knowledge Graph Integration	**UC-4**: Submit Multi-step Queries	**US2**: Multi-step query resolution with complex relationships in the knowledge graph.	Level 0
FR-4: Query Response Retrieval	**UC-5**: Handle Ambiguous Queries	**US3**: The system should ask for clarification on ambiguous queries.	Level 0
FR-6: Saving and Loading	**UC-6**: Save and Load Chat Sessions	**US5**: As a user, I want to view and re-use my query history.	Level 1
FR-8: Error Handling	**UC-7**: Handle System Errors	**US4**: I want to be notified if the system encounters technical issues.	Level 1
FR-5: User Registration	**UC-8**: User Registration and Login	(Covered by general system features, no direct user story connection.)	Level 1

VI. System Evolution

A critical consideration in the evolution of the system is software refinement. The system will experience multiple stages of development, testing, and feedback. During the feedback phase, the team will gain insight into technical shortfalls or missing functionality. Risk points in our design include dependencies on third-party libraries and services, which may introduce compatibility problems or become deprecated. Integration with large data sets might introduce performance bottlenecks, requiring alternative strategies. Output quality may be incorrect if the underlying data set contains inaccuracies or becomes outdated. The system can operate on a modular data set, and the team will pay close attention to the relationship between the data set and output quality during each iteration and feedback phase. The team anticipates potential challenges and changes to occur, and design decisions will be made to avoid technical restrictions in coming developments. The focus is on modularity and detailed documentation, which will improve the ability to modify the system as needed, as well as ensure it remains adaptable to technological advances, evolving user requirements, and stakeholder feedback.

Solution Approach

VII. Project Design Introduction

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 project aims to improve the accuracy and relevance of generated responses by combining unstructured and structured data using state-of-the-art retrieval methods. This is achieved by integrating knowledge graphs into the RAG pipeline, allowing for contextually rich and accurate responses. This document serves as a blueprint for the system's development and will evolve with future revisions as more insights are gained during the project's lifecycle.

VIII. System Overview

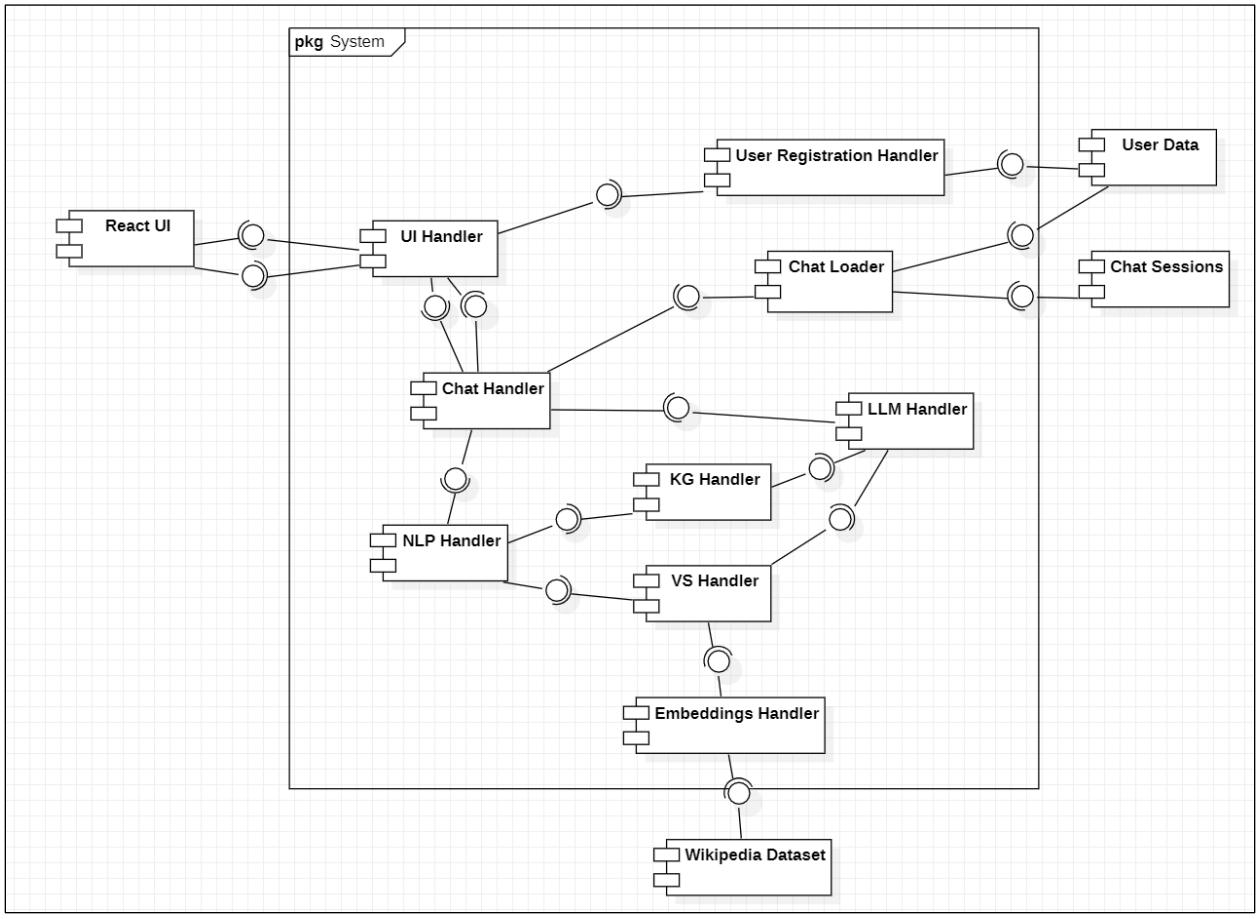
The RAG application bridges the gap between large language models (LLMs), vector search, and knowledge graphs to create a more effective information retrieval system. At its core, the system is designed to query a large dataset of Wikipedia articles, retrieve semantically relevant data using vector embeddings, 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.

IX. Architecture Design

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.

1. Overview



2. Subsection Decomposition

2.1 [UI Handler]

2.1.1 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. When the user is signed in, it displays previous chats on the left side of the chat box.

2.1.2 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.

2.1.3 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
AuthenticateUser	User Registration Handler
ManageChatUI	Chat Handler

2.2 [Chat Handler]

2.2.1 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.

2.2.2 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.

2.2.3 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
LoadChatSession	Chat Loader

2.3 [Natural Language Processing (NLP) Handler]

2.3.1 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 the query. NLP Handler then routes the processed query to the KG Handler and the VS Handler.

2.3.2 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.
- Intent Detection: Determines the main intent of the query, whether it's asking for facts, definitions, or something more complex.
- 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

2.3.3 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

2.4 [Knowledge Graph (KG) Handler]

2.4.1 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. This process aims to deliver accurate and contextually relevant responses to user queries.

2.4.2 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.

2.4.3 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

2.5 [Vector Search (VS) Handler]

2.5.1 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.

2.5.2 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.

2.5.3 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
ProcessQuery	NLP Handler

2.6 [LLM Handler]

2.6.1 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.

2.6.2 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.

2.6.3 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

2.7 [Embeddings Handler]

2.7.1 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.

2.7.2 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.

2.7.3 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

2.8 [Chat Loader]

2.8.1 Description

The Chat Loader component is responsible for interacting with external chat sessions and user data systems. Its primary role is to manage the retrieval, storage, and interaction of chat session data while ensuring that authentication and personalization requirements for user data are met. This subsystem serves as a bridge between the internal system and these external data sources, facilitating access to user and session data across various other subsystems.

2.8.2 Concepts and Algorithms Generated

The Chat Loader is a single class designed to take input from external Chat Sessions and User Data sources. It converts this input into a usable format for internal data structures that need information about chat sessions and user details. Specifically, it retrieves the following information from chat sessions: Username, Session ID, Chat History, Session Status, Start Time, and End Time. From User Data, it gathers Username, Password Hash, and Preferences. Additionally, the Chat Loader can take internal system chat session data and translate it back into User Data and Chat Sessions, serving as an update mechanism for these external data structures.

2.8.3 Interface Description

Services Provided:

Service Name	Service Provided To	Description
SaveChatSession	Chat Sessions	SaveChatSession takes the input of the Chat Handler's data object and will output appropriately formatted data to the Chat Sessions.
LoadChatSession	Chat Handler	LoadChatSession will load old chat history from the Chat Sessions database. The Chat Handler will display old queries and responses to the UI.

Services Required:

Service Name	Service Provided From
AuthenticateUser	User Data

2.9 [User Registration Handler]

2.9.1 Description

The User Registration Handler is responsible for managing user sign-up and log-in processes. It enables users to access personalized features, such as their chat history. The User Registration Handler interacts with the UI Handler and the Chat Loader to display and load previous chats for authenticated users.

2.9.2 Concepts and Algorithms Generated

The handler securely processes user credentials using authentication algorithms. It validates inputs, hashes passwords for safe storage, and manages session tokens. After successful authentication, it retrieves personalized data such as the user's chat history.

2.9.3 Interface Description

Services Provided:

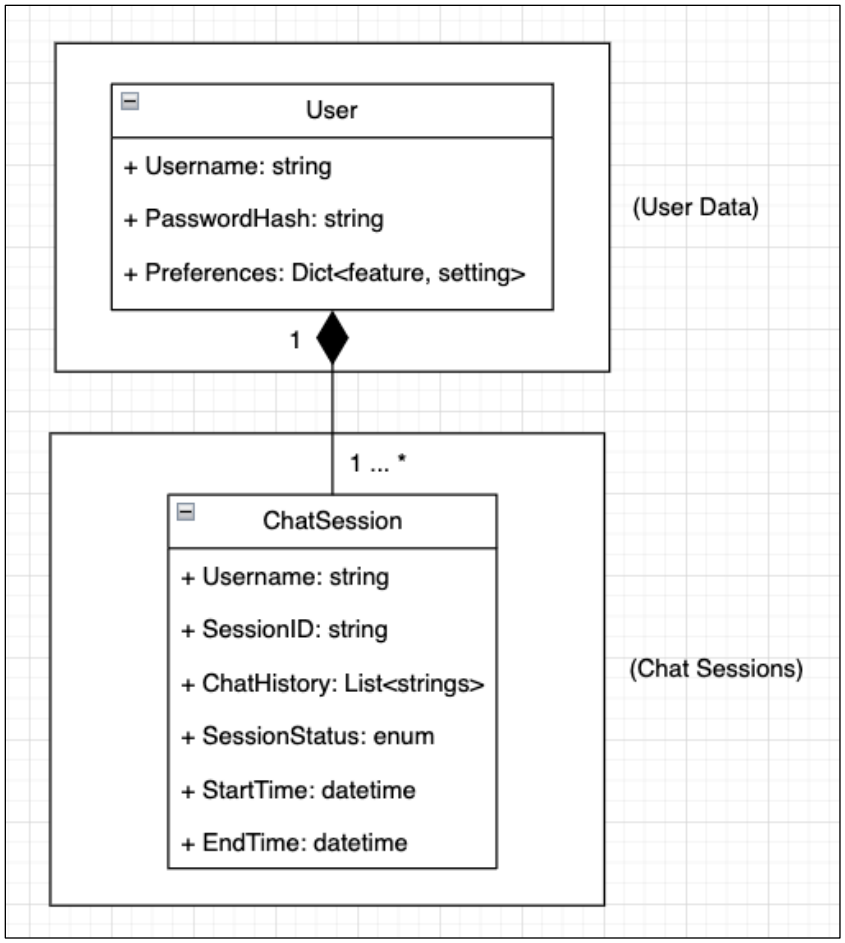
Service Name	Service Provided To	Description
AuthenticateUser	UI Handler, Chat Loader	AuthenticateUser supplies user authentication data to validate and manage user sessions. It allows the UI to show chat history.

Services Required:

Service Name	Service Provided From
CurrentUserList	User Data

X. Data Design

For this application, two primary data structures interact with external subsystems: User and Chat Sessions. For more information about the internal subsystem interactions, see the System Architecture diagram and description above. The relationship between these data structures is that each user is associated with a User object, which contains their Username. The Username is a field within the Chat Session structure and references the corresponding user from the User structure, making the Username the foreign key. This structure allows multiple Chat Session instances to be associated with one User through the Username.



For the User data structure, there will be a User object that consists of four main properties: Username, PasswordHash, Preferences, and LastLogin. The Username is a unique string that serves as both the login and the identifier. The PasswordHash is a hashed string of the user's password. Preferences is a dictionary of user preferences or settings within the application. LastLogin is a datetime timestamp of the user's most recent login.

Regarding the Chat Session data structure, there are six main properties: Username, SessionID, ChatHistory, SessionStatus, StartTime, and EndTime. The Username is a foreign key referencing the Username from User data. This identifier associates a specific user with this Chat Session by using the user's existing Username. The SessionID is a unique identifier for this chat session. The ChatHistory is a log of messages exchanged within this session. The SessionStatus is an enum indicating whether the session is active, paused, or ended. StartTime and EndTime are datetime timestamps for when the session begins and ends, respectively.

XI. 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 the Appendix section (Figure 1). 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.

At the top right corner of the UI, there are buttons for user registration and sign-in. These features are planned for future iterations, allowing users to create accounts or sign in to save their chat sessions. Each of these actions will direct users to dedicated pages: the registration page for new users to sign up, and the login page for returning users to access their saved sessions. The registration page will prompt users to enter details such as their name, email address, and password, and will include form validation for security. Upon successful registration, users will be automatically signed in and redirected to the main chat interface, where they can start saving sessions. The login page will require users to input their credentials (email and password) to access their account. After a successful login, users will be redirected to the home page.

Once user registration and session-saving features are introduced, a collapsible side menu will be located on the left-hand side of the UI. This menu will provide a scroll-down list where users can start a new chat session or view and select from their previously saved chat sessions. Clicking on a saved session will load the chat history into the main chat box, allowing users to seamlessly continue from where they left off.

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.

Although the current design does not include animations, these will be considered in later iterations to enhance the user experience. The background imagery will initially be simple and unobtrusive, but we may explore more detailed designs or theme options based on client feedback.

Testing and Acceptance Plans

XII. Testing and Acceptance Plan Introduction

1. Section Overview

This section provides an overview of the methodology for testing and acceptance plans regarding the functionality of the RAG application. The purpose of testing is to ensure that the application performs reliably and delivers appropriate responses to users' queries. The key functionalities to be tested include query processing, vector search, knowledge graph integration, and the user interface.

2. Test Objectives and Schedule

The objective is to verify that the application meets both functional and non-functional requirements and that it operates consistently under various conditions. The main testing objectives will focus on the core components of the system, including the knowledge and vector search handlers, as well as query processing. These objectives aim to evaluate both the standalone functionality and the integration of these components within the application. The required resources for testing encompass the application code, a testing framework, and relevant documentation. Depending on the programming language or framework used, an appropriate testing framework will be employed. The testing strategy schedule begins with requirement-based testing, followed by automated unit and integration testing, manual functional testing, performance testing, and finally, user acceptance testing.

3. Scope

This section of the document focuses on the planning, resources, and timelines for testing the RAG application. It outlines our testing strategies and processes, detailing our plans for unit, integration, and system testing, as well as our environmental requirements, including hardware and software specifications, network configurations, databases, testing tools, and CI/CD pipelines.

XIII. Testing Strategy

The testing strategy for our RAG application involves a combination of automated and manual testing. The primary focus is verifying critical features' accuracy, efficiency, and user experience, particularly integrating knowledge graphs and vector search functionalities. Our testing process will include unit, integration, system, and user acceptance testing, with automated tests run through a CI/CD pipeline to streamline continuous integration.

Testing Process:

1. **Requirement-Based Testing**: Each requirement from the specification will be mapped to corresponding test cases, ensuring comprehensive coverage.

2. **Automated Unit and Integration Testing**: Automated unit tests will independently validate the correctness of each handler (e.g., Knowledge Graph, Vector Search, LLM Handler), while integration tests will check for seamless interactions between components. We will use PyTest for unit tests and integrate GitHub Actions for CI/CD.

3. **Manual Functional Testing**: Developers will conduct manual functional tests to verify that each feature performs as expected, mainly focusing on query handling, retrieval accuracy, and response generation.

4. **Performance Testing**: Using tools like JMeter, we will assess response times, scalability, and the application's ability to handle large query volumes, ensuring performance stability.

5. **User Acceptance Testing (UAT)**: The final phase includes UAT with feedback from HackerEarth stakeholders, refining the application based on real-world use.

This layered testing approach will allow us to identify, isolate, and resolve issues efficiently, ensuring a robust, user-friendly RAG application.

XIV. Test Plans

This testing section outlines our approach to unit, integration, and system testing (functional, performance, and user acceptance). Thorough testing is essential for ensuring software reliability and efficiency.

XIV.1. Unit Testing

We will follow 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 will test isolated handlers (LLM Handler, KG Handler, VS Handler, NLP Handler, etc.). We will 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 will cover edge cases, boundary conditions, and performance.

Here are some tools and libraries we may use.

- **Unittest or PyTest:** Popular frameworks for creating and running tests in Python.
- **Mock or unittest.mock:** To mock dependencies.
- **Coverage.py:** To measure the code coverage of our tests.

To ensure that the code is sufficiently tested, the team will be required to test 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.

XIV.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 will group interconnected components to identify faults in their communication. From the architecture diagram, we will test how data flows between NLP Handler, VS Handler, KG Handler, and LLM Handler to ensure accurate responses are generated. We will start with pairs of handlers and slowly add more until the whole system is tested.

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

XIV.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.

XIV.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.

XIV.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.

XIV.3.3 User Acceptance Testing

User acceptance testing (UAT) for the RAG application involves direct interaction with HackerEarth's stakeholders and end-users to validate that the application meets their requirements and expectations. The UAT plan includes:

1. Resource Provision:
    - A functional version of the RAG application for end-user testing.
    - Feedback forms for stakeholders to collect feedback on usability, functionality, and accuracy.
2. Testing Process:
    - End-users from HackerEarth will be given access to the application to perform specified tasks, such as submitting queries and reviewing enriched responses.
    - Users will be guided through the application, with a focus on testing key features like query processing, response accuracy, and integration of knowledge graph data.
    - After testing, users will complete feedback forms to provide insights on their experience and suggest improvements.
3. Feedback and Revision:
    - The team will review all feedback and conduct a retrospective session to identify areas for improvement.
    - Based on the feedback, necessary adjustments will be documented, and changes will be incorporated to better meet end-user needs.

UAT aims to ensure that the RAG application aligns with HackerEarth's objectives and is ready for real-world use, providing seamless, efficient, and accurate experience for end-users. The final iteration of testing will verify that all required functionalities are operational and meet user expectations, confirming the system's readiness for deployment.

XV. Environment Requirements

Testing Environment:

1. **Hardware Requirements**:
    - Development workstations with at least 16GB RAM and 4 CPU cores.

- Servers for testing environments with 32GB RAM and scalable storage options to handle the knowledge graph and vector embeddings.

2. **Software Requirements**:
    - Operating Systems: Windows, macOS, or Linux for development and testing environments.
    - Backend: Python, with dependencies including FAISS for vector search, SPARQL libraries for knowledge graph queries, and PyTest for testing.
    - Frontend: React for the UI, accessible through a modern web browser.
3. **Network and Database**:
    - Stable internet connectivity for handling live queries to knowledge graph sources.
    - Vector and knowledge graph databases hosted on cloud infrastructure (e.g., AWS or Google Cloud) to support high availability and scalability.
4. **CI/CD and Testing Tools**:
    - GitHub Actions for CI/CD to automate the testing process.
    - JMeter for performance testing to simulate various load conditions.
    - Docker for containerized deployment, facilitating consistent testing environments.

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

XVI. Glossary

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.

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.

XVII. 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

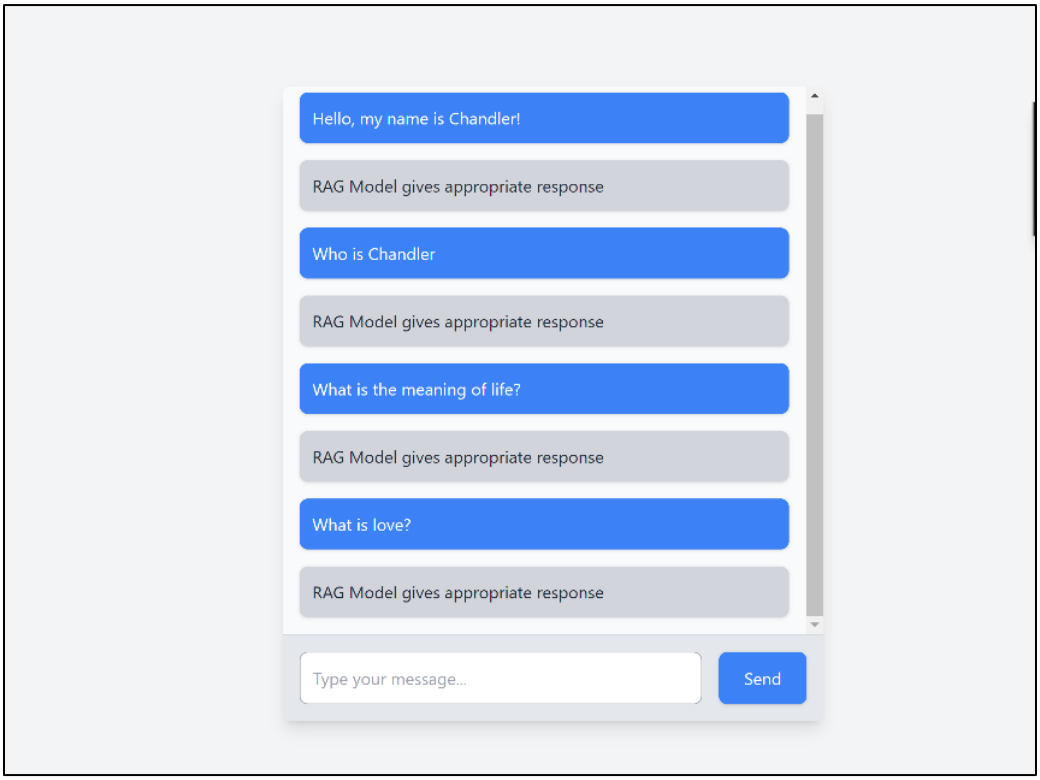
[3] A. Kollegger, “Knowledge Graphs for RAG,” DeepLearning.AI. [Online]. Available: https://www.deeplearning.ai/short-courses/knowledge-graphs-rag/

[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

XXII. Appendix



(Figure 1: UI Design – Chat Box Example)