Multimodal Retrieval-Augmented Generation for Context-Aware Chatbots

Ákos Kránics, Gábor Hosu, Nándor Bándi
Faculty of Mathematics and Computer Science
Babeș-Bolyai University
RO-400084 Cluj-Napoca, Romania
{akos.kranics, gabor.hosu}@stud.ubbcluj.ro, nandor.bandi@ubbcluj.ro

Eszter Farkas, Kinga Petkes
Codespring
Software development
RO-400084 Cluj-Napoca, Romania
{farkas.eszter, petkes.kinga}@codespring.ro

***Abstract*—The widespread adoption of large language models presents numerous opportunities for enhancing human productivity. A common challenge is extracting the essential information from multimodal sources within a short timeframe. While many existing services aim to address this issue, they primarily focus on answering general queries rather than supporting the efficient context management across user inputs.**

The proposed paper discusses the management of contexts arising from multimodal sources, and proposes an application designed to efficiently process both documents and images while enabling question-answering based on the content of uploaded files. To ensure the accuracy and traceability of responses, the application highlights the specific text segments used during answer generation, making the sources easy to verify and locate.

The system employs a Retrieval-Augmented Generation (RAG) approach to effectively extract relevant context from multiple sources. This enhances the precision and reliability of the answers provided in response to user queries.

The architecture consists of four main components: a web application, a cross-platform Android–iOS mobile application, a central backend server, and a locally hosted microservice. This paper presents the architecture, functionality, and interaction of these components, providing a comprehensive overview of the system from both client- and server-side perspectives.

***Index Terms*—Retrieval augmented generation, Large language model, Chatbot, Vector database**

I. Introduction

The proposed platform is designed to help users quickly and effectively extract information from both documents and images, aiming to enhance productivity in information-heavy tasks. With the growing capabilities of large language models (LLMs), such tools have become increasingly valuable, especially when users need to find relevant content within a short time frame. While these models significantly boost efficiency, the importance of critical thinking and source validation remains paramount [1].

One of the most common issues users face is having to sift through lengthy documents just to find a few key sentences [2], [3]. Several tools, both free and subscription-based—such as ChatGPT, Deepseek, and Grok—attempt to address this by allowing users to ask questions directly based on uploaded documents. However, a major drawback of these systems is that uploaded files often cannot be carried over between different conversations, requiring repeated uploads and disrupting workflow.

The proposed context management approach addresses this limitation by offering persistent storage of uploaded files and the ability to link them to any new or existing conversations. This enables document-focused interactions without the redundancy of re-uploading, making the platform more efficient and user-friendly than general-purpose chatbots.

This paper presents an overview of the proposed platform, including its user interface design principles, implemented functionalities, system architecture, and the technologies and tools used during development.

II. Related work

The issue of document context management has been already addressed by many existing systems. OpenAI's ChatGPT supports document upload and context sharing across chats within projects [4]. A similar concept is used in Anthropic's Claude chatbot, users can attach documents to projects or configure an integration with Google drive [5]. While ChatGPT uses fixed size overlapping chunks, Claude generates a summary sentence for each chunk. Context retrieval is achieved via semantic search which is extended with keyword filter for ChatGPT, and for Claude a hybrid approach is used that is based on BM25 and uses reranking.

The novelty of the proposed platform lies in the proposal of an overall architectural blueprint that enables chatbot systems to integrate multimodal sources into a single context. The proposed context management solution integrates OCR models and RAG to derive a single unified context.

III. Methods

A. Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) [6] enhances LLMs by addressing their limitations in handling specific tasks without retraining or fine-tuning, which can be resource-intensive. RAG improves question answering by using external sources, such as documents, as context for the LLM.

To overcome the LLM's limited input capacity, RAG uses natural language processing (NLP) to identify the relevant context for a user's question, converting text from source files into vector embeddings—numerical representations of text chunks that encode their meaning. The distance between vectors indicates their similarity. To overcome the LLM's limited

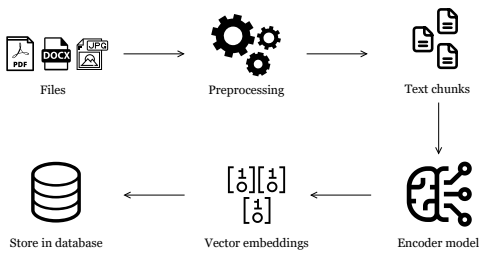

Fig. 1: The process of text embedding: files are chunked, an encoder model then generates the embedding vectors.

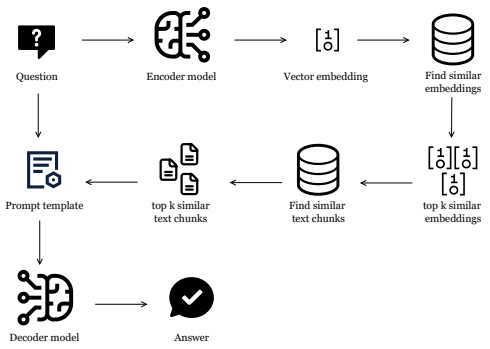

Fig. 2: The processing of user prompts: the question is encoded via the encoder model, the vector database is queries for relevant embeddings which feed into the final prompt template based on which the LLM generates the answer.

input capacity, RAG uses natural language processing (NLP) to identify the relevant context for a user's question, converting text from source files into vector embeddings—numerical representations of text chunks that encode their meaning. The distance between vectors indicates their similarity.

Before embedding conversion, a preprocessing phase extracts text from diagrams, tables and chunks the text into smaller segments, minimizing information loss. The text is then converted into vector embeddings using an encoder model, which utilizes the encoder component of the transformer architecture [7] as shown in Figure 1.

The vector embeddings are stored in a vector database, designed for fast similarity searches using a specific distance metric. To answer a question, the user's query is also converted into a vector embedding using the same encoder model. The database is filtered for a given number of relevant document chunks most based on a similarity measure which are used as context.

The retrieved context is combined with the user's question in a prompt template and provided to the LLM, which generates the answer based on the given information as illustrated in Figure 2.

B. Large Language Models

LLMs are based on the transformer architecture [7]. The encoder generates vector embeddings from the source text, which the decoder uses to produce the output. Decoder-only models, commonly used in chatbots, focus on generating responses based on user input.

In this project, LLMs are orchestrated using the Ollama[1] tool, which simplifies downloading and running pretrained models in isolation. It also provides an API for programmatic execution using various libraries.

C. Encoder models

In the implementation of the RAG method, the all-MiniLM-L6-v2[2] model was used as the encoder. This model has 22.7 million parameters and is a fine-tuned version of MiniLM-L6-v2 [8], designed to encode English text chunks while preserving their semantic meaning.

The encoder model generates compact 384-dimensional vector embeddings during text conversion. These embeddings enable efficient information retrieval in the RAG system by allowing similarity-based searches among the stored vectors.

D. Decoder models

One of the challenges of the application's development was identifying a suitable decoder model for generating responses. Achieving the current level of response quality required testing multiple models.

During the initial prototyping phase, the 137 million parameter GPT-2 [9] model—trained by OpenAI—was used to ensure local runnability. Later, with access to more capable hardware, it became possible to experiment with the 380 million, 812 million, and 1.61 billion parameter versions. However, none of these models delivered the expected answer quality: the generated text often appeared as if it were quoted from external sources, rather than providing direct answers to the questions.

The Llama 2 [10] model provided answers with the expected quality, but its behavior was not as controllable. This posed a challenge, particularly during context-based title generation for conversations. The length of the generated text was difficult to regulate, and at times, overly creative titles were produced that were unrelated to the corresponding conversation.

Based on previous experience, the 7.3 billion parameter Mistral [11] model developed by Mistral AI was chosen, specifically the instruction-following fine-tuned version. This model was designed to complement the Llama 2 model by enhancing its performance. As a result, the instruction-following version of the Mistral model outperforms the 13 billion parameter version of Llama 2 across various benchmarks.

In the current phase of the project, the 7.3 billion parameter Mistral, fine-tuned for instruction following, model is in use.

[1]https://ollama.com/

[2]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

E. Vector database

A key component of the RAG method, alongside the question-answering LLM, is an optimized vector database that stores contextual vector embeddings of relevant text chunks.

Initially, the Chroma[3] vector database was used for persistent storage but lacked support for concurrent requests, affecting performance. To address this, Milvus[4] was adopted, offering concurrent request handling, scalability, and flexibility with various data and index types.

Milvus supports multiple metrics for vector comparison, with cosine similarity being used in this project, defined as

$$\mathrm{sim}(\mathbf{x}, \mathbf{y}) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|},$$

where $\|-\| : \mathbb{R}^d \to \mathbb{R}$ denotes the Euclidean norm on vectors, and $\langle -, - \rangle : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ denotes the dot product.

Each uploaded file is stored as a separate collection in Milvus, with vector embeddings and associated metadata like source file ID and chunk position (page numbers). Cosine similarity is used to find the most similar text chunks, and to speed up this process, the IVF_FLAT[5] index is employed, balancing search performance and memory usage. This index organizes vectors into clusters by centroids, with the number of centroids defined by the `nlist` parameter. The query searches for the nearest centroids, then selects the `k` most similar vectors.

In the implementation, `k=5`, `nlist=128`, and `nprobe=10` parameter values are used.

F. Optical Character Recognition

The RAG approach is extended to include images as context in addition to documents.

This process is done using Optical Character Recognition (OCR), which extracts the text from the uploaded images.

For optical character recognition, the project utilized the pretrained, lightweight EasyOCR[6] model developed by JaidedAI. The model, based on convolutional neural networks, is available as a Python package for easy integration into software systems.

IV. Practical Implementation of the RAG Method

The following section presents the custom implementation of the RAG method, based on the previously discussed components and technologies. The overall architecture is depicted in Figure 5.

A. Architecture

The RAG system is built on a multi-endpoint microservice architecture alongside a Milvus vector database server. Its core functionality is implemented through server-side components, which communicate with the underlying subsystems. These server-side components will be described in detail in the following subsections of this section.

[3]https://www.trychroma.com/
[4]https://milvus.io/
[5]https://milvus.io/docs/ivf-flat.md
[6]https://github.com/JaidedAI/EasyOCR

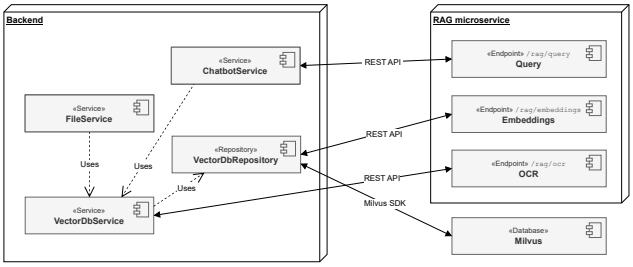

Fig. 3: The backend system interacts with the RAG components via REST API.

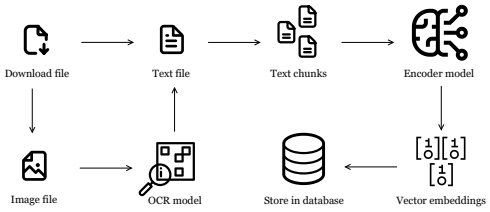

Fig. 4: The process of encoding the user input. Images are fed to an OCR model extracting the textual context. The texts are embedded via the aforementioned embedding process.

The main services include question answering, searchable file storage and context retrieval based on user questions, and text extraction from the uploaded images.

The RAG microservice architecture consists of the following components:

- *Query endpoint*: Generates answers to user questions by leveraging the decoder model, orchestrated through the Ollama tool.
- *Embeddings endpoint*: Generates vector embeddings using the encoder model, orchestrated via the Ollama tool.
- *OCR endpoint*: Extracts text from uploaded images using the previously mentioned OCR model.

The Milvus vector database provides persistent storage for vector embeddings and supports efficient similarity search. This component communicates with the backend server through a Python package built on top of the Milvus SDK.

The microservice endpoints are hosted on a Hypercorn[7] asynchronous HTTP server and implemented using the FastAPI[8] framework. The backend server-side components involved in the RAG process interact with these endpoints via an HTTP-based RESTful API as depicted in Figure 3.

B. File processing

The application supports various file types (TXT, DOCX, PDF, PNG, JPG, JPEG), collectively referred to as files with these extensions. The goal of file processing is to extract content from uploaded files and convert it into a searchable format for storage, addressing the LLM's limited input size.

[7]https://pypi.org/project/Hypercorn/
[8]https://fastapi.tiangolo.com/

The file processing workflow involves pre-processing and storing the extracted text in a vector database for efficient searchability. Upon file upload, the backend server processes the files in the background, extracting text from plain text files directly or using an OCR model for image files as illustrated in Figure 4.

Large files are processed iteratively, loading small text chunks (e.g., pages for PDFs, paragraphs for DOCX, lines for TXT) to optimize memory usage. Text chunks are standardized in length with a fixed-size overlap to reduce content loss.

Due to the lack of suitable external libraries for file scanning and text segmentation, custom file iterators were necessary to implement. These iterators use a sliding window algorithm to scan files and generate text fragments in linear time. The image iterator sends content to the OCR endpoint via a REST API, with the extracted text processed similarly to other file types. File iterators also track page numbers, which are defined for PDFs, undefined for DOCX due to rendering, and treated as a single page for TXT and images.

Preprocessing is managed by the `VectorDbService`, which communicates with the vector database via `VectorDbRepository`. Asynchronous communication improves processing speed, with the system's architecture illustrated in Figure 3.

C. Title and answer generation in conversations

The RAG system generates textual responses and conversation titles to user queries. Text generation by the locally deployed LLM is guided by prompts, offering a resource-efficient balance between fine-tuning and inference. Prompt templates set the context for the input, informing the LLM of the desired output structure.

The prompt template for responses includes guidelines such as maintaining a polite tone, staying within the provided context, and adhering to conversation history. It is followed by the relevant conversation history and the user's question.

The title is generated based on the user's initial message and context, guided by a prompt template that includes task-specific instructions and representative examples. The template concludes with the user's input, applying few-shot learning [12] to support the generation process.

Interactions with the LLM, managed by the Ollama orchestration tool, are facilitated by the LangChain[9] library, which embeds input fragments into prompt templates and configures LLM hyperparameters.

Key hyperparameters for text generation include temperature and maximum number of tokens. LangChain limits temperature to the interval $[0, 1]$ for stability. The utilized values are 0.9 and 256 for title generation, and 0.9 and 20 for response generation.

Architecturally, the RAG microservice communicates with the backend server (`ChatbotService`) via RESTful API calls, as shown in Figure 3.

[9]https://python.langchain.com/docs/introduction/

D. Chat memory

A significant limitation of the RAG method implementation, described in the preceding subsections, lies in its predominantly question-answer-oriented approach, which overlooks the preceding content of the conversation in which the user's question is embedded. As a result, this implementation may lead to reduced coherence and continuity in multi-turn dialogues.

Various approaches exist to address the memory limitation outlined above. Considering hardware constraints and the need for efficient response times, the following solution has been implemented in the project: the context derived from the relevant files is incorporated into the prompt template alongside a fixed number of preceding conversation messages, the majority of which are denoted by `m`. This combined input is then used by the large language model to generate a response to the user's question within the appropriate context. In the specific implementation of the project, the parameter value is set to `m=6`.

The implementation of chat memory is handled by the `ChatbotService` on the server side.

V. Architecture

The system architecture comprises several key components, each fulfilling a distinct role in ensuring functionality, scalability, and seamless user interaction. The main components are as follows:

- *RAG microservice*: Facilitates question answering via a large language model and processes user-provided contextual documents.
- *Server*: Central component responsible for handling client requests and managing communication with the database and RAG microservice.

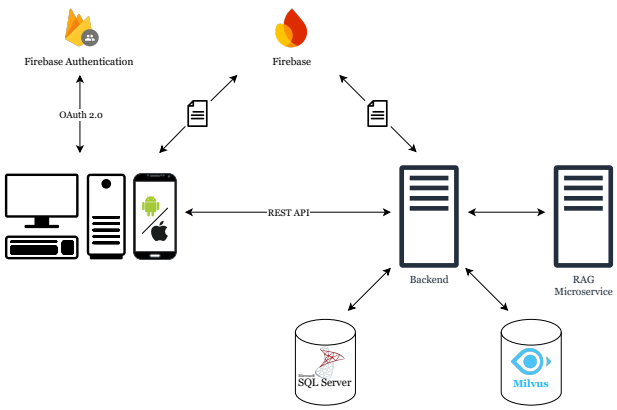

Fig. 5: Architectural overview of the platform. Client side interacts with Firebase and the backend via REST. The backend side orchestrates the user management, document storage and LLM queries.

- *Web and mobile clients*: Provide a unified interface (web, Android, iOS) for user interaction and manage integration with Firebase Authentication and Storage.
- *Firebase*: Handles cloud-based file storage (Firebase Storage) and user authentication via Google accounts (Firebase Authentication).
- *Databases*: User data is stored in a Microsoft SQL Server database, while fast context retrieval for the RAG pipeline is supported by a Milvus vector database.

A. Server

The backend server is structured according to a three-tier architecture, which promotes modular development by clearly separating software component responsibilities.

Client applications interact with the backend through a RESTful HTTP API, representing the presentation layer. Based on the Model-View-Controller (MVC) design pattern, incoming requests are routed to controllers defined by the FastAPI framework.

Data Transfer Object (DTO) patterns are used to validate incoming data and filter out irrelevant information from responses.

The presentation layer responds to requests using service interfaces defined in the service layer, which in turn provides the actual implementations. Most services interact only with the data access layer, except for components such as `ChatbotService` and `VectorDbService`, which are integral to the RAG system.

The service layer accesses persistent data through interfaces provided by the data access layer, which is responsible for data storage and retrieval.

B. Client applications

The system includes two client interfaces: a web application and a cross-platform mobile application. Both allow users to interact with a chatbot interface powered by a large language model, enabling efficient extraction of information from uploaded documents and images.

The web application is built using the Model–View–ViewModel (MVVM) architectural pattern, ensuring a clear separation of concerns between the interface, logic, and data layers. Developed with React and TypeScript, the web client offers a rich feature set, including user authentication (via email/password or Google), conversation management, message rating and copying, and advanced file handling. Users can attach files to new or existing conversations, filter documents, and organize them in folders using a global file management system. Additional features include drag-and-drop file movement, folder navigation, and document search. Firebase Authentication and Storage services are integrated for user login and persistent file storage.

The mobile application, built with React Native and Expo, follows a modular three-layer architecture: visual, logic, and communication layers. Navigation is implemented through Expo Router's file-based routing system. The app supports

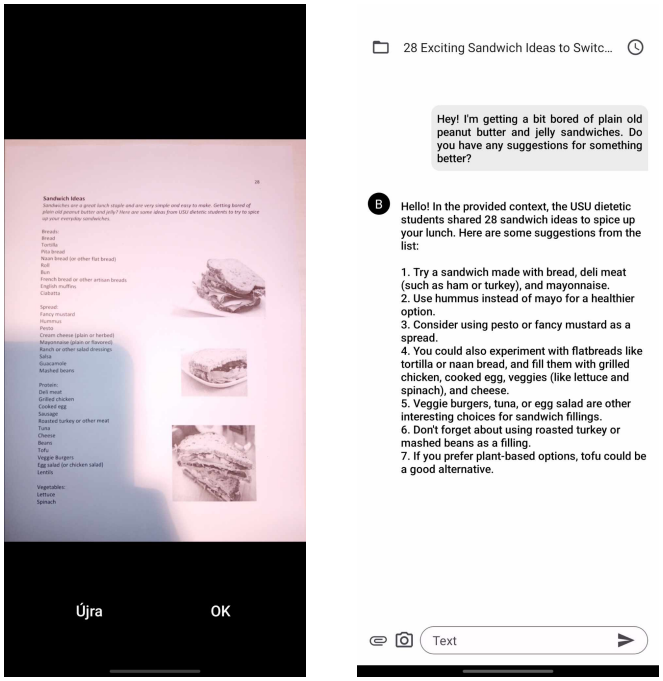

Fig. 6: Using an image from a cookbook as context for new generating new recipe ideas.

user registration, conversation handling, file attachment, and a unique mobile-only feature: capturing and uploading images via the device's camera. It uses Context API and AsyncStorage for managing global state (e.g., active user, current conversation), and communicates with the same backend via REST APIs.

While the mobile version offers slightly reduced functionality compared to the web client—such as the lack of folder management or source reference display—it introduces mobile-specific enhancements and ensures a consistent user experience across platforms. Both clients rely on a shared backend, unified authentication, and centralized file handling, forming a cohesive, document-centric conversational system.

VI. Discussions

Using a cookbook[10] as context, the chatbot engages in a conversation where the user requests an alternative sandwich idea. Based on the provided ingredients, the chatbot generates a detailed recipe. The conversation and the referenced text excerpt are shown in Figures 6,7, 8 and 9, highlighting that the response draws directly from pages 28–29 of the source.

This interaction exemplifies the functionality of the implemented Retrieval Augmented Generation (RAG) system. The architecture consists of web and mobile clients that communicate with a backend server through a RESTful API. On the server side, input files are processed into text chunks, which are embedded using the all-MiniLM-L6-v2 encoder

[10]https://www.usu.edu/aggiewellness/files/USU-Student-Cookbook-FINAL-1.pdf

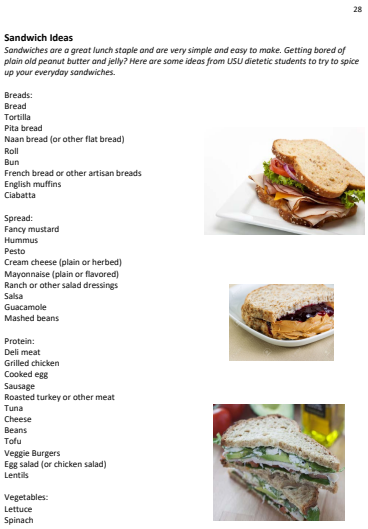

Fig. 7: One of the contextual excerpts used in the conversation from the provided document (page 28).

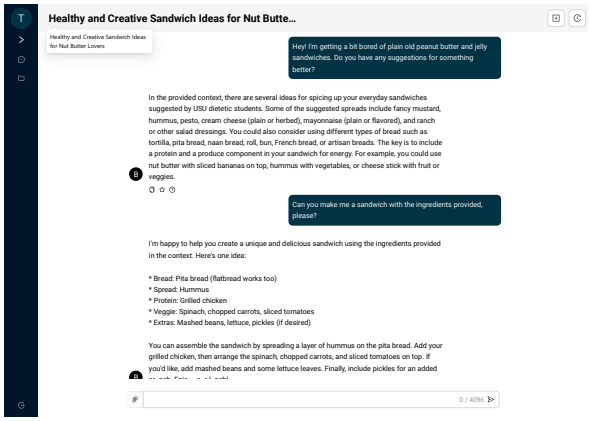

Fig. 8: Conversation based on the provided source document, in which the user asks the chatbot for sandwich-making advice.

model. The resulting vectors are stored in a Milvus vector database, enabling efficient context retrieval based on cosine similarity. When a user submits a query, the system retrieves relevant context and generates a grounded response using the Mistral-7B decoder model.

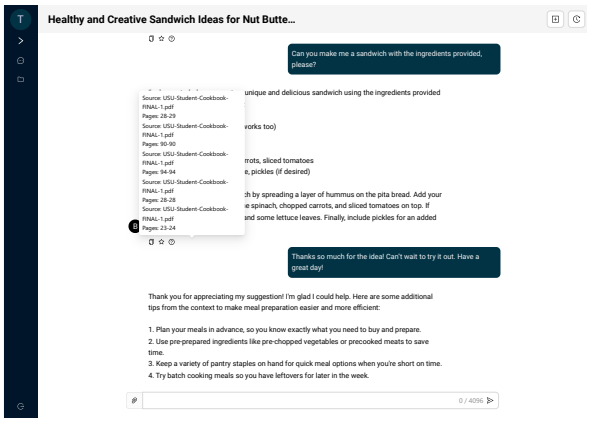

Fig. 9: Illustration demonstrating source usage in the conversation, showing pages 28–29 of the referenced cookbook.

VII. Conclusion

The proposed platform provides a blueprint that enables the creation and management of conversations, messages evaluation, and uploading and deleting files in addition to taking pictures for context expansion. As a result, a versatile, cross-platform application has been developed, which can meet various user needs.

Although the main objectives of the project were met, some functionalities could not be implemented due to time constraints. For the mobile application, this includes the ability to create folders and authentication via external providers (e.g., Google-based authentication).

In conclusion, the proposed chatbot project has been successfully implemented as a stable, well-structured system, providing a solid foundation for further extensions and practical use in real-world environments. The system's ability to provide context-aware question answering from user-provided documents and images, powered by the Retrieval-Augmented Generation (RAG) architecture, represents a significant step in advancing intelligent chatbot systems. The key components, including the all-MiniLM-L6-v2 encoder for vector embedding, Milvus for retrieval, and the Mistral-7B decoder for response generation, have proven effective in generating accurate, traceable, and grounded answers. There are multiple directions for future development. These include the introduction of conversation archiving and sharing with other users, as well as utilizing a more advanced response mechanism for message evaluation, which could enhance system functionality and improve user experience. Additionally, improvements aimed at increasing the reliability of the backend system, such as implementing transaction management for file operations or optimizing system scalability, would be beneficial. In the web application, the file preview feature was omitted, which can also be added in the future. Future work will also focus on enhancing conversation continuity and optimizing the system's scalability and responsiveness.

References

[1] C. Wu, W. Ding, Q. Jin, J. Jiang, R. Jiang, Q. Xiao, *et al.*, "Retrieval augmented generation-driven information retrieval and question answering in construction management," *Advanced Engineering Informatics*, vol. 65, p. 103 158, 2025, ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2025.103158. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474034625000515.

[2] K. Adnan and R. Akbar, "An analytical study of information extraction from unstructured and multidimensional big data," *Journal of Big Data*, vol. 6, no. 1, p. 91, Oct. 2019, ISSN: 2196-1115. DOI: 10.1186/s40537-019-0254-8. [Online]. Available: https://doi.org/10.1186/s40537-019-0254-8.

[3] S. V. Mahadevkar, S. Patil, K. Kotecha, L. W. Soong, and T. Choudhury, "Exploring ai-driven approaches for unstructured document analysis and future horizons," *Journal of Big Data*, vol. 11, no. 1, p. 92, Jul. 2024, ISSN: 2196-1115. DOI: 10.1186/s40537-024-00948-z. [Online]. Available: https://doi.org/10.1186/s40537-024-00948-z.

[4] OpenAI. [Online]. Available: https://help.openai.com/en/articles/10169521-projects-in-chatgpt.

[5] Anthropic. [Online]. Available: https://www.anthropic.com/news/projects.

[6] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 9459–9474.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017.

[8] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, *Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers*, 2020. arXiv: 2002.10957 [cs.CL].

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[10] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[11] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].

[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.