A Multilingual Intelligent Document Question-Answering System

Rohit Singh[*], Santosh Grampurohit, Keshav Kumar, Chandan Kumar Singh, Sk Mohammad Arif and Sourajit Bhar

Indian Institute of Science (IISc)

Abstract. As global organizations increasingly work with multilingual document collections, there is a growing need for systems that can process and query documents across language barriers. While some document QA systems have been developed for specific languages, most operate only in English, and few can handle multilingual document repositories effectively. If configured correctly, multilingual document QA systems have the potential of providing a digital information extraction solution that transcends language barriers.

For our project, we developed a multilingual document processing system that enables users to upload documents in various languages and interact with their content through natural language queries. The system leverages RAG architecture combined with multilingual language models to provide accurate, contextually relevant answers extracted from user-provided documents, regardless of the source language. This solution addresses the growing need for efficient multilingual document analysis in global academic, professional, and research contexts. By combining cross-lingual NLP techniques with user-friendly interfaces, the system democratizes access to complex multilingual document analysis, enabling users to quickly extract insights from diverse document collections without manual translation or language-specific searching. Source code at: https://github.com/rohitlee/document-chatbot

1 Introduction

1.1 Background and Motivation

The proliferation of multilingual document collections and the growing demand for cross-lingual question-answering systems have positioned Retrieval-Augmented Generation (RAG) as a critical technology in natural language processing. Large language models such as OpenAI's GPT-4 and Meta's Llama-70B have demonstrated exceptional capabilities in multilingual RAG applications, leveraging extensive context windows (up to 128K tokens in GPT-4o and 32K in enhanced open-source models) and comprehensive training data to achieve superior performance metrics.

However, the practical deployment of these large models presents significant challenges. High computational costs, with inference expenses often exceeding several dollars per million tokens, combined with privacy concerns and dependency on cloud-based infrastructure, limit their accessibility for many organizations and applications. This creates a compelling need for enhancing the performance of smaller, open-source alternatives that can be deployed locally while maintaining cost-effectiveness.

[*] Paper submission for IISc Deep Learning Course Project (DA225o).

1.2 Problem Statement

Smaller open-source models like Mixtral-8x7B-Instruct[1], despite offering advantages in cost (approximately 0.70 USD per 1M tokens), customization flexibility, and offline deployment capabilities, face substantial performance gaps in multilingual RAG applications. These models typically operate with limited context windows (33K tokens) and demonstrate lower baseline performance, with MMLU scores around 0.387 compared to their larger counterparts.

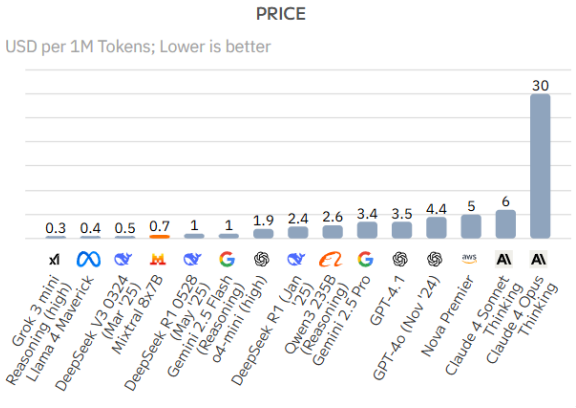

Figure 1. Token-level pricing comparison (USD per 1M tokens) of popular language models, highlighting the cost-effectiveness of open-source models like Mixtral-8x7B and LLaMA compared to proprietary models such as Claude and GPT-4.

The primary challenge lies in optimizing retrieval strategies to compensate for the reduced contextual understanding and processing capabilities of smaller models, particularly in multilingual scenarios where linguistic diversity and cultural context variations compound the complexity.

In this project, we first evaluated the retrieval performance of a baseline multilingual embedding model, paraphrase-multilingual-mpnet-base-v2, by calculating standard metrics such as Hit Rate@k (k = 1, 3, 5, 10) and Mean Reciprocal Rank (MRR). This baseline helped us understand how well the model could retrieve relevant documents given a user query in a multilingual context using purely semantic search. To enhance retrieval accuracy, we then implemented a hybrid search approach that combines semantic similarity with keyword-based search, leveraging the complementary strengths of both methods. The results from each retrieval strategy were merged

and re-ranked using Reciprocal Rank Fusion (RRF), a simple yet effective late-fusion technique. This hybrid RRF method significantly improved retrieval performance across all evaluated metrics, demonstrating its effectiveness in capturing both the deep semantic intent and surface-level keyword overlap in user queries, especially in complex or multilingual scenarios.

2 Related Work

The core of our system is the Retrieval-Augmented Generation (RAG) framework, first proposed by Lewis et al. [5]. RAG combines the strengths of pre-trained language models with non-parametric memory from a retriever, which addresses the limitations of fixed model knowledge and reduces hallucination. This paradigm was a natural evolution from earlier open-domain QA systems like DrQA [2], which separated the retriever and reader components. A comprehensive survey by Zhang et al. [9] provides an extensive overview of the RAG landscape, highlighting its versatility.

Central to any RAG system is the retriever, which has been the subject of extensive research. The move from sparse, keyword-based retrieval to dense passage retrieval (DPR) marked a significant milestone [4]. DPR uses dual-encoder architectures based on transformers like BERT [3] to embed questions and passages into a shared vector space for semantic search. The effectiveness of these embeddings is crucial, with models like Sentence-BERT [6] specifically fine-tuned to produce semantically meaningful sentence embeddings for tasks like semantic search, which directly inspired the embedding models used in our project.

The challenge intensifies in a multilingual context, where a single model must understand and retrieve information across different languages. While our work uses general-purpose multilingual models, more specialized cross-lingual retrieval models like ColBERT-X [7] have been developed to improve performance on such tasks. Our project's contribution lies not in proposing a new model architecture, but in the practical, systematic evaluation of existing open-source components for multilingual RAG. We demonstrate how architectural choices, such as implementing a hybrid search with Reciprocal Rank Fusion, and component upgrades, like switching to a more modern embedding model, can significantly enhance the performance of a cost-effective, locally deployable system. Our evaluation methodology, while custom, is in the spirit of standardized benchmarks like BEIR [8], which advocate for robust evaluation of IR models.

3 Methodology

3.1 System Architecture Overview

The architecture of our multilingual document-based question-answering system, illustrated in the figure 2 below, is based on the Retrieval-Augmented Generation (RAG) framework. This design enables efficient, multilingual question-answering using smaller open-source models through a hybrid retrieval strategy and context-aware generation.

3.2 Document Ingestion and Indexing

This process builds the knowledge base that the chatbot will use to answer questions. It is represented by the upper flow in the figure.

- **Document Input:** The process begins when a user uploads a source document.
- **Processing and Chunking:** The document processor ingests the raw document. It parses the text and splits it into smaller, overlapping chunks of a predefined size (e.g., 1000 characters). This ensures that the semantic meaning of each unit is coherent and digestible for the embedding model.
- **Embedding:** Each text chunk is passed to the embedding model. The model converts the text into a high-dimensional vector, capturing its semantic meaning.
- **Indexing and Storage:** The original text chunk, its vector embedding, and relevant metadata (like the source filename and chunk ID) are stored together in the Vector Database (ChromaDB). This indexed database is now ready to be queried.

3.3 Query-Response Cycle

This process is triggered every time a user asks a question. It is represented by the central flow from left to right in the figure.

- **User Query:** The User submits a question in any supported language. This becomes the initial user query / prompt.
- **Query Pre-processing:** The system first processes the query. In our multilingual implementation, this involves translating the query into a pivot language (English) to standardize the search process.
- **Retrieval:** The processed query is sent to the retriever. The query is first embedded into a vector using the same embedding model. The Retriever then performs a hybrid search against the vector database, fetching a list of the most relevant document chunks.
- **Context Augmentation:** The text content from the retrieved chunks is collected. This context is combined with the processed user query to form a single, detailed prompt. This is the "Augmentation" step in RAG, where the model's knowledge is augmented at inference time.
- **Generation:** The augmented prompt is sent to the Generator (LLM). The LLM's sole task is to read the provided context and generate a helpful, accurate answer to the question based on that information alone. This grounding in retrieved facts is what minimizes hallucination.
- **Final Response:** The answer generated by the LLM is post-processed (e.g., translated back to the user's desired language) and then presented to the user, completing the cycle.

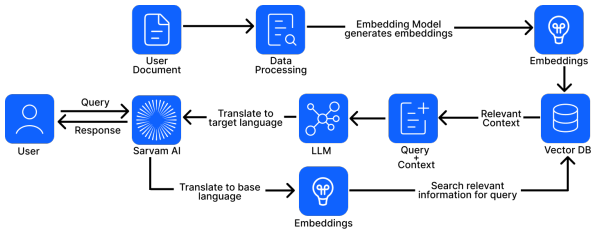

Figure 2. Architectural Overview of the Multilingual RAG System

4 Experiments and Results

To quantitatively evaluate and iteratively improve our RAG system, we designed a series of experiments focused on the performance of the retrieval component. This section details the experimental setup, the datasets created, the metrics used, and the results obtained at each stage of improvement.

4.1 Experimental Setup

Our experiments were conducted on a local machine with the primary goal of measuring the effectiveness of different retrieval strategies in a multilingual context. All evaluations were automated using a dedicated script to ensure consistency and reproducibility.

- **Key Libraries:** sentence-transformers (v2.2.2), chromadb (v0.4.0), pandas (v2.1.0).
- **LLM for Generation:** The dataset generation was performed using the llama-3.1-8b-instant model via the Groq API.
- **Translation Service:** All multilingual translations were handled by the Sarvam AI API.

4.2 Dataset Creation

A custom evaluation dataset was created to serve as the ground truth for our retrieval experiments. The process was as follows:

1. **Corpus Curation:** A diverse corpus of five documents was assembled, with each document in a different language relevant to the Indian subcontinent: English, Hindi, Bengali, Punjabi, and Kannada. The topics covered a range of subjects to ensure the evaluation was not biased towards a single domain.
2. **Document Processing:** The five source documents were processed using our system's DocumentProcessor. This involved splitting the text into chunks of 1000 characters with a 150-character overlap. Each chunk was assigned a unique, deterministic ID based on its source file and position.
3. **Q&A Pair Generation:** We leveraged a powerful Large Language Model (llama-3.1-8b-instant) to generate a high-quality dataset. For each text chunk, the LLM was prompted to create exactly one question and a concise answer that could be derived solely from the provided chunk's content. This automated process resulted in a dataset of 258 unique (question, ground-truth answer, ground-truth chunk ID) triplets. This file, evaluation dataset, formed the basis for all subsequent retrieval evaluations.

4.3 Evaluation Metrics

To measure the performance of our retriever, we used the following standard information retrieval metrics:

- **Hit Rate @k:** The proportion of queries for which the correct `ground_truth_chunk_id` was found within the top k retrieved documents. This measures the recall of the system.
- **Mean Reciprocal Rank (MRR):** The average of the reciprocal of the rank at which the first correct document was found. MRR heavily penalizes results where the correct document is ranked lower and provides a single, comprehensive score for ranking quality.

4.4 Iterative Retrieval Experiments

We conducted three distinct experiments to measure the impact of architectural and model improvements.

Experiment 1: Baseline Semantic Search
Our initial experiment established a baseline using a simple semantic search retriever.

- **Embedding Model:** sentence-transformers/paraphrase-multilingual-mpnet-base-v2
- **Retrieval Method:** A standard vector similarity search.

The results, shown in Table 1, indicate a modest performance, with the correct document being missed entirely in the top 10 results for nearly 40% of queries.

Experiment 2: Hybrid Search with Reciprocal Rank Fusion (RRF)
Observing the limitations of pure semantic search, particularly with queries involving specific keywords, we hypothesized that a hybrid approach would improve precision.

- **Embedding Model:** sentence-transformers/paraphrase-multilingual-mpnet-base-v2
- **Retrieval Method:** A hybrid strategy combining semantic search and keyword search. The results from both were fused using Reciprocal Rank Fusion (RRF) to produce a single, reranked list.

As seen in Table 1, this architectural change yielded a significant improvement in top-ranked results, with a 10.5% relative increase in MRR.

Experiment 3: Upgrading the Embedding Model
While hybrid search improved ranking, the overall recall (Hit Rate @10) remained unchanged, suggesting that the core semantic understanding of the embedding model was the next bottleneck. We noted that many of the remaining "hard" queries were complex and required a deeper understanding of intent rather than lexical overlap. To address this, we upgraded the foundational embedding model to a modern, retrieval-focused model.

- **Embedding Model:** intfloat/multilingual-e5-base
- **Retrieval Method:** The same Hybrid Search with RRF from Experiment 2.

This change targets the system's ability to better comprehend complex queries and bridge the semantic gap between a user's question and the document's language.

4.5 Results and Discussion

The consolidated results from all three experiments are presented in Table 1.

Table 1. Retrieval Performance Comparison Across Different Strategies

Metric	Baseline	Stage 2	Stage 3
Hit Rate @1	32.17%	38.37%	44.19%
Hit Rate @3	44.96%	49.61%	59.69%
Hit Rate @5	51.55%	53.49%	67.44%
Hit Rate @10	60.85%	60.85%	75.97%
MRR	0.4048	0.4471	0.5342

The results clearly demonstrate the value of our iterative approach. The architectural shift to hybrid search (Stage 2) provided a notable boost in precision (MRR). However, the most substantial improvement came from upgrading the core embedding model (Stage 3). The multilingual-e5-base model improved the MRR by 32% over the baseline and, critically, increased the Hit Rate @10 by 15.12%, successfully retrieving answers for a significant portion of the previously "hard" queries. This confirms our hypothesis that a more powerful embedding model is the most critical factor for improving retrieval on complex, nuanced, and multilingual queries.

5 Conclusion

In this project, we successfully designed and evaluated a multilingual RAG chatbot, demonstrating that high-performance RAG systems can be built exclusively with open-source models. Our iterative evaluation process, beginning with a baseline semantic search (MRR 0.4048), first showed that a hybrid search architecture using Reciprocal Rank Fusion could improve ranking precision (MRR 0.4471). The most significant performance gain, however, was achieved by upgrading to a modern intfloat/multilingual-e5-base embedding model, which boosted the MRR to 0.5342—a 32% improvement over the baseline. This result confirms that while sophisticated retrieval architecture is beneficial, the foundational quality of the embedding model is the most critical factor for successfully handling complex, multilingual queries. Our findings affirm that thoughtful model selection within the open-source ecosystem is a powerful and accessible path to creating effective and reliable AI systems.

6 Contribution of members

The project was a collaborative effort, with responsibilities divided among team members to ensure comprehensive development and evaluation. The contributions are outlined below:

- **Rohit Singh** led the project management, overseeing code integration and ensuring project cohesion. Implemented the hybrid search architecture and was primarily responsible for creating the evaluation framework to test its performance. This included generating the custom multilingual dataset and conducting the iterative retrieval experiments detailed in this paper. He also authored this manuscript and developed the main application interface (`app.py`).
- **Santosh Grampurohit** served as the Backend and Ingestion Engineer. His work focused on the `document_processor.py` component, where he implemented the logic for loading, parsing, and chunking source documents. He was responsible for handling various file formats and implementing text-splitting strategies to prepare data for indexing in the vector database.
- **Keshav Kumar** worked on the retrieval and search system (`retrieval_system.py`). As part of a joint effort, he contributed to the implementation of the hybrid search architecture, combining semantic and keyword-based retrieval. His key contributions include implementing Reciprocal Rank Fusion (RRF) for re-ranking and integrating different multilingual embedding models.
- **Chandan Kumar Singh** acted as the LLM and Prompt Engineer, managing the generation component of the RAG pipeline (`response_generator.py`). His work involved engineering the prompts sent to the Large Language Model to ensure accurate and contextually grounded answers. He also experimented with various open-source LLMs to select the optimal model for the generation task.
- **Sk Mohammad Arif** and **Sourajit Bhar** jointly handled the integration of external cloud and API services through the `nlp_processor.py` module. Their responsibilities included managing API calls for translation and language modeling, implementing robust error handling with retry mechanisms, and developing a caching system to optimize performance and reduce costs.
- All team members actively participated in the quality assurance and documentation process. This included thorough testing of new features, writing clear code comments and docstrings, maintaining the project's documentation, and conducting peer reviews to ensure the overall reliability and robustness of the final system.

References

[1] Artificial Analysis. Mixtral-8x7b-instruct model details. https://artificialanalysis.ai/models/mixtral-8x7b-instruct, 2024. Accessed: May 21, 2024.

[2] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes, 'Reading wikipedia to answer open-domain questions', *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 1870–1879, (2017).

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, 'Bert: Pre-training of deep bidirectional transformers for language understanding', in *Proceedings of NAACL-HLT 2019*, pp. 4171–4186, (2019).

[4] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih, 'Dense passage retrieval for open-domain question answering', in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, (2020).

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Mandar Kulkarni, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela, 'Retrieval-augmented generation for knowledge-intensive nlp tasks', *Advances in Neural Information Processing Systems*, **33**, 9459–9474, (2020).

[6] Nils Reimers and Iryna Gurevych, 'Sentence-bert: Sentence embeddings using siamese bert-networks', *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 3982–3992, (2019).

[7] Karthik Santhanam, Shaden Shaar, Leshem Choshen, Hadar Alon, Eyal Shnarch, and Ido Dagan, 'Colbert-x: A cross-lingual retrieval model for multilingual information retrieval', *arXiv preprint arXiv:2109.07471*, (2021).

[8] Nandan Thakur, Nils Reimers, Johannes Daxenberger, and Iryna Gurevych, 'Beir: A heterogeneous benchmark for zero-shot evaluation of information retrieval models', in *Advances in Neural Information Processing Systems*, volume 34, pp. 26177–26188, (2021).

[9] Yiming Zhang, Yuxuan Sun, Yujia Li, Jimmy Lin, and Xipeng Ma, 'A comprehensive survey on retrieval-augmented generation', *arXiv preprint arXiv:2202.01110*, (2022).

A Appendix: System User Interface and Code Implementation

This appendix provides a visual and technical supplement to the main paper. It showcases the user interface (UI) of the developed multilingual RAG chatbot and details key code snippets that implement the core functionalities discussed in the methodology.

A.1 User Interface Showcase

The following figures illustrate the user journey, from launching the application to performing multilingual queries.

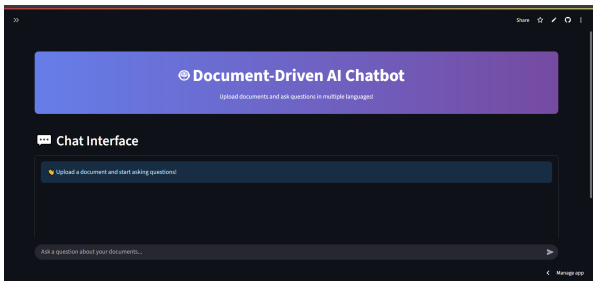

Figure 3. The initial state of the chatbot application. The user is greeted with a clean interface and prompted to upload documents via the sidebar to begin the interaction.

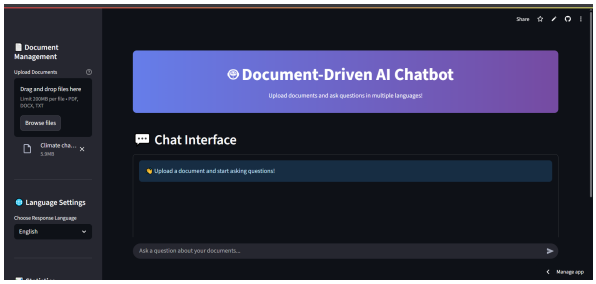

Figure 4. The user interacts with the sidebar to upload a source document. The uploader supports multiple file formats (PDF, DOCX, TXT). Below, the language selection dropdown and real-time statistics (Documents, Queries) are visible.

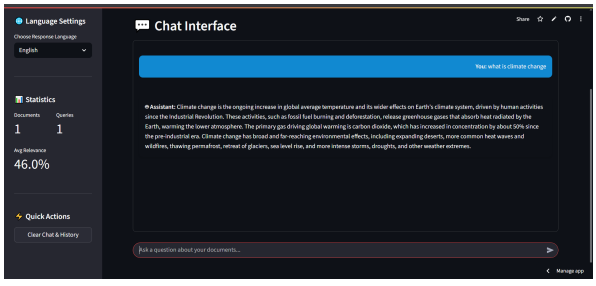

Figure 5. A user asks a question in English ("what is climate change"). The system retrieves the relevant context from the uploaded document and generates a factually grounded answer in English. The statistics in the sidebar are updated to reflect one processed document and one query.

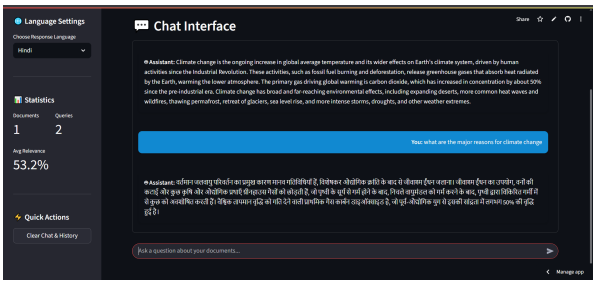

Figure 6. Demonstrating the system's core multilingual feature. The user selects "Hindi" as the response language and asks a question in English. The system correctly translates the query for internal processing, retrieves the information, and generates the final answer accurately in Hindi, as requested.

A.2 Key Code Implementation Details

This section highlights critical functions from the codebase that are responsible for the system's hybrid retrieval and multilingual capabilities.

A.2.1 Hybrid Search with Reciprocal Rank Fusion

As discussed in the paper, improving retrieval performance was critical. The following code from `retrieval_system.py` implements the hybrid search strategy. The `hybrid_search` function orchestrates the process, while `_reciprocal_rank_fusion` merges the results from semantic and keyword searches into a single, more accurate ranking.

```
def hybrid_search(self, query: str, k: int = 5) -> List[Dict]:
    """
    Performs a robust hybrid search using Reciprocal Rank Fusion (RRF)
    to combine semantic and keyword search results.
    """
    # 1. Fetch results from both search methods
    semantic_results = self.similarity_search(query, k=20)
    keyword_results = self.keyword_search(query, k=20)

    # 2. Fuse the results using RRF
    fused_scores = self._reciprocal_rank_fusion(
        [semantic_results, keyword_results]
    )
    # ... (code to fetch and normalize documents) ...
    return top_k_results

def _reciprocal_rank_fusion(self, result_sets: List[List[Dict]], rrf_k: int = 60
) -> Dict[str, float]:
    """
    Combines multiple search result sets using the RRF formula.
    """
    fused_scores = {}
    # Iterate through each list of search results
```

```
    for results in result_sets:
        # Iterate through each document in the result list
        for rank, doc in enumerate(results):
            doc_id = doc.get('id')
            if doc_id:
                if doc_id not in fused_scores:
                    fused_scores[doc_id] = 0
                # Add the reciprocal rank score
                fused_scores[doc_id] += 1.0 / (rrf_k + rank + 1)

    return fused_scores
```

Listing 1. Hybrid Search Implementation from `retrieval_system.py`

A.2.2 Multilingual Query Orchestration

The main application file, `app.py`, contains the logic for handling the entire multilingual request-response cycle. The `generate_chatbot_response` function is the central orchestrator. It first translates the user's query into a pivot language (English), performs the search, generates the answer, and finally translates the answer back into the user's desired language. This implementation directly reflects the architecture shown in Figure 2 of the paper.

```
def generate_chatbot_response(
    query: str, nlp_processor, retriever,
    response_generator, language: str
):
    """Orchestrate the full RAG pipeline for a multilingual response."""

    # 1. Translate user's query to English for searching.
    english_query = nlp_processor.translate_text(
        query, source_lang="auto", target_lang='en-IN'
    )

    if not english_query or not english_query.strip():
        # Handle translation failure
        return {"response": "...", "confidence": 0.0}

    # 2. Retrieve relevant documents using the English query
    retrieved_docs = retriever.hybrid_search(english_query, k=5)

    # 3. Handle the case where no relevant documents are found
    if not retrieved_docs:
        # ... (translate "not found" message) ...
        return {"response": translated_not_found, "confidence": 0.0}

    # 4. Generate a response using the LLM. Pass the user's chosen
    # language for the final translation step.
    response = response_generator.generate_response(
        english_query, retrieved_docs, nlp_processor,
        target_language=language
    )

    # ... (calculate confidence and return) ...
    return {"response": response, "confidence": confidence}
```

Listing 2. Multilingual RAG Orchestration from `app.py`

A.2.3 Context-Aware Response Generation

The `response_generator.py` module is responsible for communicating with the Large Language Model. The `generate_response` function first calls `_create_context` to format the retrieved documents. It then invokes the LLM with a carefully crafted prompt, ensuring the model's answer is grounded in the provided context. If the target language is not English, it uses the `nlp_processor` to translate the final response.

```
def generate_response(
    self, query: str, retrieved_docs: list,
    nlp_processor: NLPProcessor,
    target_language: str = 'en-IN'
) -> str:
    context = self._create_context(retrieved_docs)

    english_response = self._generate_with_hf_api(query, context)

    if target_language != 'en-IN' and english_response:
        return nlp_processor.translate_text(
            english_response, source_lang='en-IN',
            target_lang=target_language
        )

    return english_response or "Could not generate a response."

def _generate_with_hf_api(self, query: str, context: str) -> str:
    system_prompt = "You are a helpful AI assistant. Answer the user's question based *only* on the provided context. If the context does not contain the answer, state that you could not find the information in the documents. Be concise. "

    user_prompt = f"""CONTEXT:
    {context}

    QUESTION: {query}"""

    prompt = f"<s>[INST] {system_prompt} \n\n {user_prompt} [/INST]"
```

```
    # ... (code to call Hugging Face API with
    ↪  the prompt) ...
```

Listing 3. Response Generation Logic from `response_generator.py`