UNIVERSITÀ POLITECNICA DELLE MARCHE

FACULTY OF ECONOMICS “GIORGIO FUÀ”

Bachelor’s Degree Program in

Digital Economics and Business

ENHANCING LLM RESPONSES IN THE FIELD OF RENEWABLE ENERGY THROUGH A RAG-BASED SYSTEM

Supervisor:

Prof. Alex Mircoli

Final Report by:

Emidio Veccia

Academic Year 2024/2025

Introduction

1.1 Context and Motivation

1.2 Limitations of Standalone LLMs

1.3 Objectives of the Thesis

1.4 Structure of the Thesis

System Architecture

2.1 Overall Design of the RAG Pipeline

2.2 Document Ingestion and Chunking

- 2.2.1 Text Extraction with PDFPlumber
- 2.2.2 Text Cleaning and Normalization
- 2.2.3 Sentence Segmentation with NLTK
- 2.2.4 Character-based Chunking and Overlap Strategy
- 2.2.5 Chunk Length Selection
- 2.2.6 Metadata Management and Document Traceability

2.3 Embedding Generation and Vector Store Management

- 2.3.1 Embedding Model Selection and Justification
- 2.3.2 Vector Normalization and Similarity Metric
- 2.3.3 FAISS Indexing and Metadata Mapping

2.4 Semantic Retrieval and Reranking

- 2.4.1 Query Embedding and FAISS Search
- 2.4.2 Cross-Encoder Reranking Strategy

2.5 Response Generation with LLM (Groq API)

- 2.5.1 Prompt Construction and Model Call

- 2.5.2 Output Post-processing and Storage

Implementation and Testing

3.1 Evaluation Methodology: Cosine Similarity Score

3.2 Test Results and Discussion

Conclusions and Future Work

4.1 Summary of the Work

4.2 Strengths and Limitations of the Solution

4.3 Potential Future Developments

Chapter 1 - Introduction

1.1 Context and Motivation

In recent years, sustainability has become a strategic priority for institutions and businesses worldwide. As a result, companies are required to produce a growing number of documents related to Environmental, Social, and Governance (ESG) practices (including sustainability reports, energy transition plans, and compliance documentation). These texts are often dense, technical, and difficult to process manually, especially for small and medium enterprises that lack dedicated ESG employees.

The resulting information overload creates a barrier to knowledge access: extracting relevant insights from long and heterogeneous documents can be time-consuming and error-prone. Even when the information exists, finding it and understanding it quickly remains a major challenge for decision-makers.

This growing complexity highlights the need for intelligent tools that can help users extract meaningful insights from ESG-related content efficiently and reliably.

1.2 Limitations of Standalone Language Models

Despite their impressive capabilities, standalone Large Language Models are not well suited for tasks that require access to specific, up-to-date, and verifiable information. As discussed in the previous section, ESG documentation is vast and constantly. Relying exclusively on static, pre-trained models risks overlooking recent developments that are not part of their training data.

In addition to lacking real-time access, LLMs cannot guarantee the factual correctness or source traceability of their outputs, even when prompted to cite sources. This creates a serious challenge in domains like sustainability reporting, where precision, accountability, and timely information are essential.

These flaws underscore the need for a more grounded and transparent approach to information retrieval, one that combines the linguistic fluency of LLMs with the factual anchoring of external knowledge bases.

1.3 Objectives of the Thesis

This thesis sets out to design, implement, and evaluate a Retrieval-Augmented Generation (RAG) system tailored to the needs of users navigating ESG-related documentation with a particular focus on renewable energy initiatives in Italy.

Building on the limitations discussed in the previous section, the goal is to construct a solution that combines semantic retrieval with generative capabilities, enabling users to interact with one or more documents through natural language queries and receive accurate, context-aware answers.

The system's configuration was gradually refined through iterative testing, including parameter tuning and reranking strategies, in order to maximize the semantic similarity between model responses and reference answers.

While the current version of the system ingests a limited number of plain-text documents, it demonstrates the potential of RAG-based methods to enhance the accessibility of complex ESG material. The next chapter presents the architecture of the proposed system and the logic behind each design choice.

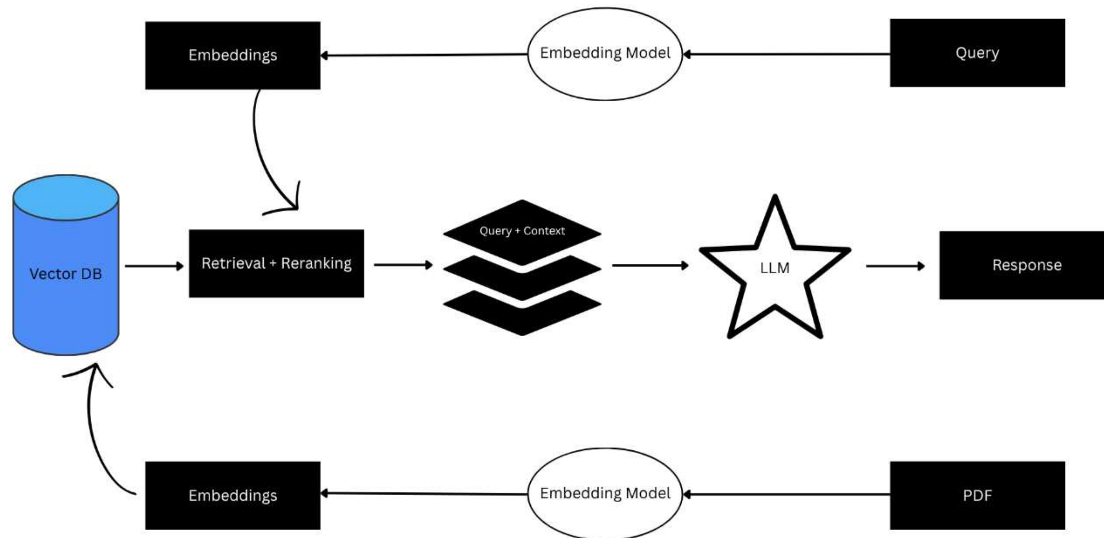
1.4 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 introduces the architecture of the proposed RAG-based system, explaining the main components such as document chunking, embedding generation, vector-based retrieval, and response generation through a Large Language Model. Each design choice is discussed in relation to practical constraints and implementation trade-offs.
- Chapter 3 focuses on the testing of the system. It evaluates the model's performance based on semantic similarity between generated and reference answers.
- Chapter 4 concludes the work with a thoughtful evaluation of the results obtained, discusses the limitations of the current approach, and outlines possible directions for future developments.

Chapter 2 - System Architecture

2.1 Overview of the RAG Pipeline



Retrieval-Augmented Generation is a technique that combines dense semantic retrieval with language generation, enabling models to ground outputs on external, document-based evidence.

The system developed in this thesis adopts a canonical RAG architecture implemented as a modular and extensible pipeline, designed to operate locally with minimal infrastructure requirements.

The architecture is composed of four main stages, each implemented as an independent module with clearly defined responsibilities:

- **Document ingestion and chunking**: PDF files are parsed to extract text, then segmented into chunks of approximately 1200 characters, using sentence tokenization to preserve semantic coherence. Each chunk is associated with metadata for traceability.

- **Embedding and indexing**: Chunks are encoded into fixed-size dense vectors via a multilingual Transformer-based sentence embedding model. The resulting vectors are indexed using a vector database to enable efficient approximate nearest-neighbor retrieval.
- **Retrieval and reranking**: At inference time, a user query is embedded using the same model and compared to the indexed chunks. The top-k results are reranked using a supervised cross-encoder to improve contextual alignment.
- **Response generation**: The final selected chunks are concatenated and inserted into a prompt template, which is then passed to a cloud-hosted LLM (accessed via API) to generate the final answer in natural language.

Each stage is parameterized to facilitate experimentation with different chunk lengths, reranking strategies, and model backends. The next sections describe each module in detail, outlining the implementation logic and design decisions involved.

2.2 Document Ingestion and Chunking

2.2.1 Text Extraction with PDFPlumber

The ingestion module converts one or more ESG PDFs into well-formed text blocks ("chunks") that the embedding engine and, later, the Large Language Model can work with. While doing so, it must keep the natural reading flow intact, even when the original layout is messy.

To extract the text we use **PDFPlumber** because it rebuilds the page in visual reading order, the same sequence a person would naturally follow. Titles come before body paragraphs, the left column is read before the right, and footers don't show up in the middle of a sentence.

PDFPlumber achieves this by:

• Reading the bounding box of every printed character: the x- and y-coordinates of its rectangle on the page.

• Grouping nearby characters into words, and nearby words into lines, using user-defined pixel thresholds for horizontal and vertical distance.

• Lines are then sorted from top to bottom and, within each line, from left to right, so the final text stream mirrors the visual layout.

Other libraries such as PyMuPDF or PyPDF2 follow the raw drawing order of the PDF objects, which may merge fragments improperly, making chunks less meaningful and harming retrieval performance.

2.2.2 Text Cleaning and Normalization

Following text extraction, the system performs a huge pre-processing aimed at ensuring the proper structure and remove formatting issues.

The first issue concerns the **soft hyphen character**. This is an invisible symbol that some word processors insert inside words to suggest where they might be split across lines. Even if the split does not actually happen, the character can remain in the text and cause problems later. For example, the word “sustainable” might appear as “sust--ainable,” splitting a single word into two invalid tokens. We remove all such characters from the text.

Another issue involves **broken words** that are split across lines with a hyphen.

In PDFs, a word like "rinnovabile" may be broken across two lines: “rinnovabi-” and “le.” If we simply join all lines as they are, the word would stay broken, and the tokenizer might treat “rinnovabi-” as a valid word fragment, possibly producing a useless token. To fix this, the system checks for lines ending in a hyphen and joins them to the next line, but only if the next line starts with a letter.

Once the text has been cleaned from these special cases, the next step is to **reconstruct the paragraphs**. This is done by reading the text line by line: all lines that are not empty are added to a temporary buffer. When the system finds an empty line, it treats this as a paragraph break, after which the buffer is combined into a single paragraph and then reset. Of course, in some cases, the layout of the PDF may include blank lines for visual spacing, not because a new paragraph is starting. This can lead to accidental splits. However, chunking by character length and sentence boundaries mitigates these cases, restoring coherence.

At this point, the result is a clean, continuous string with consistent paragraph structure, ready for sentence segmentation.

2.2.3 Sentence Segmentation with NLTK

The cleaned and normalized text is passed to "**nltk.sent_tokenize**", a function that performs sentence segmentation using **Punkt**, a pre-trained statistical algorithm developed to identify sentence boundaries. In our implementation, we rely on the Italian Punkt sentence segmentation algorithm, which identifies sentence boundaries using contextual and statistical cues, handling tricky cases like abbreviations (for instance "A.D." or "S.p.A.") and punctuation after quotes or parentheses.

The output is an ordered list of complete sentences that preserves the logical structure of the original document.

2.2.4 Character-based Chunking and Overlap Strategy

The ordered list of complete sentences produced by the sentence segmentation step is then passed to the chunking mechanism, which in our pipeline follows a **greedy character-based strategy**. The algorithm iterates over the sentences one by one: if adding the current sentence does not exceed the 1,200-character limit, the sentence is appended to

the ongoing chunk. If it would cause the chunk to overflow, the current chunk is closed and a new one is started, beginning with that sentence.

In rare cases where a single sentence exceeds 1,200 characters, it is stored as a standalone, slightly longer chunk, avoiding any sentence splitting.

To improve semantic coherence across chunk boundaries and mitigate context fragmentation, the pipeline also implements a chunk **overlapping strategy**. Specifically, after a chunk is finalized, the last 25% of its sentences are copied and appended to the next chunk. Overlap ensures that transitions between chunks remain intelligible and logically connected. Without overlap, the retrieval model might fail to capture the full meaning of a passage that begins in one chunk and continues in the next.

The choice to use NLTK and the Punkt tokenizer was motivated by several considerations. For instance, Punkt is extremely lightweight: the model is only a few megabytes in size making it particularly well-suited for multi-document ingestion pipelines, and runs efficiently on CPU. Most importantly, it ensures precise sentence boundaries, which is crucial for meaningful chunking.

2.2.5 Chunk Length Selection

The choice of 1,200 characters as the maximum chunk length reflects a well grounded trade-off between semantic coherence, computational efficiency, and system flexibility. This value was selected based on empirical testing and the specific characteristics of ESG-related documents.

First, considering the typical character-to-token ratio observed in such texts (approximately 5.5 characters per token), a 1,200-character chunk corresponds to around 220 tokens. This size fits comfortably within the input limits of the embedding model used in our system — a multilingual MiniLM-based encoder— which can process up to

384 tokens. This allows each chunk to be encoded in its entirety without truncation, preserving all its semantic content.

Moreover, from the perspective of response generation, retrieving and processing ten such chunks per query results in approximately 2,200 tokens. Even after including prompt formatting and system messages, the total remains well within the 8,192-token context window accepted by models like Groq LLaMA 3-70B, which we use for language generation.

Practical considerations also suggested that this chunk size offers a balanced compromise:

- Smaller chunks (for instance, composed of roughly 600 characters) significantly increase the total number of embeddings, which increases the size of the vector index, introduces redundancy, and reduces retrieval precision.
- Larger chunks (at least 1,800 characters) tend to mix unrelated content, making semantic search less accurate.

2.2.6 Metadata Management and Document Traceability

Metadata are descriptive pieces of information that provide context about a piece of data, such as where it comes from, what it contains, or how it should be used.

In our pipeline, metadata are stored as Python dictionaries associated with each chunk generated during ingestion. Each dictionary contains:

- the "text" field, which holds the actual chunk content;
- the "source" field, which identifies the name of the PDF file from which the chunk was extracted.

These dictionaries are stored in a list, preserving their original order of appearance. The exact position of a chunk is not saved as a separate value, but since each chunk is stored

in an ordered list of dictionaries, its position can be inferred from the dictionary's index in the list.

These metadata play a key role during embedding: the system generates an embedding for each chunk's text and stores it together with the corresponding source document name, maintaining document traceability across all vectors. It also enables precise mapping between embeddings and the content they represent.

During response generation, retrieved chunks are matched to their metadata, so the system can indicate their source document.

Metadata is useful for debugging and evaluation too, making it easy to trace outputs back to specific input chunks and assess retrieval quality.

2.3 Embedding Generation and Vector Store Management

2.3.1 Embedding Model Selection and Justification

Every chunk of text obtained from the Ingestion phase is passed through the embedding model, which transforms it into a dense numerical vector of fixed **dimension 384**.

These vectors are temporarily stored in a list and then combined into a 2D NumPy matrix, where each row corresponds to one chunk.

In our case, we use the **sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2** model provided by Hugging Face. This model is based on MiniLM, a distilled version of a Transformer, meaning it keeps the essential mechanisms of larger models, such as self-attention, but with a significantly reduced number of parameters (around 66 million). This makes it lightweight and efficient, especially suitable for applications that need to scale or run on CPU.

It is well suited to our project because it is specifically trained to convert whole sentences into meaningful vectors, which aligns precisely with our use case.

It also supports multiple languages, including Italian, which is crucial for processing ESG documents from varying sources. Finally, being part of the Hugging Face ecosystem allows it to be loaded by just using built-in classes specifically designed to simplify the integration process, such as AutoModel and AutoTokenizer.

2.3.2 Vector Normalization and Similarity Metric

Before building the index, we normalize each vector obtained during the embedding phase so that its L2 norm equals 1. This normalization is essential because we use the dot product as the similarity function. When vectors have norm 1, the dot product becomes equivalent to **cosine similarity**, which allows us to measure how similar two chunks are in terms of meaning. Cosine similarity is particularly useful because it is independent from the magnitude of the vectors and focuses solely on their direction in the embedding space.

2.3.3 FAISS Indexing and Metadata Mapping

We store the normalized vectors using **FAISS** (Facebook AI Similarity Search), a library specifically designed for fast and scalable similarity search in high-dimensional spaces.

FAISS performs linear search using the dot product as its similarity measure. This aligns perfectly with our setup, since we normalize the vectors, making the dot product equivalent to cosine similarity. Moreover, FAISS integrates naturally with NumPy and PyTorch, ensuring seamless compatibility with the other components of our pipeline.

Internally, each chunk embedding (each row of the 2D tensor) is stored in the index and automatically assigned a numerical ID based on its row position.

These IDs later serve as keys to retrieve the corresponding chunks during semantic search. To preserve this link, a parallel mapping list is essential: each element in the list is a dictionary containing the original chunk text and the name of the source PDF from which

it was extracted. This list is saved to disk as a .json file, and allows any retrieved vector to be traced back to its original content and source.

Each chunk is now represented by a vector that encodes its semantic meaning and is ready to be used in the retrieval phase: the embedding phase is complete.

2.4 Semantic Retrieval and Reranking

2.4.1 Query Embedding and FAISS Search

This stage begins when the user submits a natural language query, which is first embedded using the exact same model employed during the embedding phase. This consistency ensures that the question and the document chunks reside in the same semantic space, making similarity comparisons meaningful.

The embedding of the user query produces a single 384-dimensional vector. Before performing the search, this vector is normalized just as was done for the document (essential to ensure that the dot product behaves as cosine similarity).

The normalized query vector is then compared against all the vectors stored in the FAISS index. Since we are using linear search, FAISS computes the dot product between the query and each indexed chunk vector to identify the top-8 most similar ones (8 is a good trade-off between text coverage and computational cost of the reranking).

Each result returned by FAISS corresponds to a specific row in the index, which can be traced back to the original chunk and its source file thanks to the previously saved JSON mapping.

2.4.2 Cross-Encoder Reranking Strategy

The chunks selected during the retrieval phase are then passed to the re-ranking component. The **re-ranking phase** acts as a quality filter, allowing the system to adjust for cases where high embedding similarity does not imply direct relevance to the query.

The re-ranker takes the user question and each retrieved chunk and evaluates them in pairs using a cross-encoder architecture.

In our implementation, we use the multilingual cross-encoder model **cross-encoder/mmarco-mMiniLMv2-L12-H384-v1**, specifically trained to reorder retrieved text passages according to their relevance to a given query, supporting multiple languages. It takes a (query, passage) pair as input and outputs a single relevance score. This model is well-suited to our project due to its support for Italian and its compact architecture, which strikes a good balance between accuracy and computational efficiency.

The output of this phase is a refined ranking, where the **top-3** chunks that are more likely to contain a precise and contextually appropriate answer are prioritized.

2.5 Response Generation with LLM (Groq API)

2.5.1 Prompt Construction and Model Call

Once the top-3 chunks have been selected by the re-ranking stage, they are passed to the Large Language Model along with the user's question. At this point, the model generates a final answer using only the content of these chunks as context. It is important to note that the model has access only to these selected chunks and not to the original document The input is formatted to clearly separate the chunks from the question, and includes a short instruction telling the model to answer based only on the provided content: this reduces the risk of hallucinations.

We use the **Groq LLaMA 3-70B** model, which is accessed through an API and runs on remote servers. This allows us to benefit from its advanced generation capabilities without requiring any specialized hardware on our side. The model can process long inputs and

produce fluent, accurate answers in a few milliseconds thanks to Groq’s optimized infrastructure.

2.5.2 Output Post-processing and Storage

Once the LLM returns the answer, the system performs minimal post-processing, such as trimming extra spaces or checking if the reply is empty, and then displays it to the user. Optionally, the system can also save the generated answer to a file or database, for example to include it later in a final PDF report.

Chapter 3 - Implementation and Testing

3.1 Evaluation Procedure

The test set contains twenty questions grouped as follows:

- five broad, explanatory questions
- five are more specific and detail-oriented
- five are deliberately out-of-scope questions
- five on-topic questions written with heavy spelling mistakes

For each question a reference reply (the so called gold answer) was prepared manually. That gold answer is the sentence (or short paragraph) that a human expert considers fully correct, written in the same language and level of detail expected from the system.

The evaluation proceeds in three steps.

First, the question is sent through the complete pipeline: the retriever selects candidate chunks, the multilingual cross-encoder re-ranks them, and the language model generates its answer from the filtered context.

Second, both the system answer and the corresponding gold answer are passed through the very same MiniLM encoder and converted into 384-dimensional vectors.

Finally, the cosine similarity between the two vectors is computed and stored together with the initial question, the gold answer, and the model's output; when all twenty questions have been processed, the script reports the mean similarity for the whole batch.

3.2 Cosine Similarity and Its Limitations

When the multilingual MiniLM encoder converts a sentence into a vector $v \in \mathbb{R}^{384}$, that vector is a point in a 384-dimensional space that encodes meaning.

To compare two sentences, represented by vectors v and w, we measure the cosine of the angle between them:

$$cosine_similarity(v, w) = (v \cdot w) / (\|v\|_2 \cdot \|w\|_2)$$

Before applying the formula, each vector is divided by its own L-2 norm: length effects disappear and the expression becomes a simple dot product.

If two sentences convey the same ideas, the angle θ is small and the cosine approaches 1; if they cover unrelated topics, the vectors drift toward orthogonality and the cosine falls toward 0;

negative values would imply open contradiction, an edge case in our framework.

High-dimensional geometry adds an important clue. In hundreds of dimensions, two random vectors are almost always perpendicular, so their cosine converges closely around zero.

Consequently, a coefficient above 0.8 is highly unlikely to arise by chance: it is a strong signal that the two sentences share real content.

The cosine, however, is not flawless. Two sentences can use the same vocabulary to state opposite conclusions and still achieve a high similarity score, and an otherwise correct answer truncated by the encoder may see its score fall.

For this reason, cosine similarity is a necessary but not sufficient metric: it shows whether two texts live in the same semantic neighbourhood, but it does not guarantee factual consistency.

To close that gap we added a multilingual cross-encoder re-ranker, described in next section.

3.3 The Importance of Multilingual Re-Ranking

Keyword-based retrieval alone is fast, yet it sometimes returns text snippets that mention the right terms without truly answering the question.

The multilingual cross-encoder re-ranker addresses this limitation. It feeds the complete question together with one candidate chunk at a time into a transformer that assigns a relevance score reflecting full syntactic and semantic alignment.

Chunks that respond precisely keep a high score and rise to the top; those that merely share isolated words drop to the bottom.

This deeper check addresses three recurrent problems:

- When several chunks list similar figures, the re-ranker consistently selects the one that contains the exact value requested, preventing the language model from returning the wrong number.
- When the question is packed with spelling errors, the contextual encoder still detects the underlying structure and preserves the correct chunk.
- When the question lies outside the document's scope, all chunks receive very low scores, so the model answers that the information is unavailable and avoids hallucinations.

Filtering the context in this way ensures that the language model works with evidence that is both relevant and logically consistent, raising overall answer correctness without altering either the embedding model or the generation backend.

3.4 The Impact of Different Parameter Configuration Choices

Although the cosine similarity scores reported in this evaluation are based on a fixed configuration of the pipeline, it is important to acknowledge that system performance is sensitive to multiple adjustable parameters.

While architecture and retrieval mechanics are critical, small variations in configuration can have a significant effect on outcome quality.

Any evaluation of RAG systems must therefore account not only for model capabilities but also for the calibration of operational parameters.

Changes in maximum **chunk length**, for instance, directly influence the granularity of the retrieved context: we observed that increasing the chunk length up to 1800 characters led to more stable answers on broad explanatory questions, but it risks introducing noise. On the other hand, shorter chunks increase precision but may fragment key information.

The **number of tokens** allowed in the LLM output is another crucial variable: increasing it enables longer, more detailed responses, which may align better with complex reference answers, but also opens the door to off-topic digressions.

Conversely, setting the token limit too low may cut off essential content, harming answer quality even when the retrieved context is correct.

The **prompt design** used in the final LLM call can significantly alter the tone, completeness, and factual rigor of the generated answer, ultimately affecting the similarity with the gold reference.

Moreover, the prompt design also plays a role in managing edge cases such as out-of-scope questions: by explicitly instructing the model to respond with "I don't know" or to refuse unsupported answers, the prompt guides the generation process toward more conservative, non-hallucinated outputs.

This, in turn, increases the overall semantic alignment with the gold answer, and thereby improves the cosine similarity even when the correct behaviour is to abstain.

Chapter 4 - Conclusions and Future improvements

4.1 Conclusion of the Work

This thesis presented the design and implementation of a multilingual Retrieval-Augmented Generation pipeline aimed at assisting small and medium enterprises in compiling ESG-related documentation.

The objective was to develop a system capable of answering natural language questions by retrieving relevant information from corporate ESG reports and generating grounded, context-aware responses.

The proposed architecture consists of five main components: a document ingestion module for text extraction and chunking, an embedding engine based on multilingual sentence transformers, a semantic retriever powered by FAISS, a re-ranking layer using a multilingual cross-encoder, and a large language model accessed via external API for final answer generation.

Each module was implemented to preserve modularity and flexibility, enabling targeted improvements without affecting the overall flow.

The evaluation protocol involved a manually curated test set of twenty diversified questions, each paired with a gold reference answer.

Cosine similarity was used to quantify the alignment between system-generated and gold responses, while a semantic re-ranker was integrated to improve retrieval precision and reduce hallucinations.

Experimental results confirmed that the combination of multilingual embeddings and cross-encoder re-ranking led to accurate and context-sensitive answers, particularly in challenging scenarios such as vague queries, misspellings, or out-of-scope requests.

Overall, the project demonstrates that lightweight, modular RAG pipelines can provide effective support in the domain of sustainability reporting, and offers a practical foundation for future enhancements and deployment.

4.2 Strengths and Limitations of the Solution

The solution developed in this work offers several remarkable strengths.

Its **modular architecture** allows each component of the pipeline to be tuned or replaced independently, enabling prototyping and future adaptation without requiring a full redesign.

The use of multilingual embeddings and the integration of a semantic re-ranking layer further enhance the system's robustness, ensuring that generated answers are both topically relevant and semantically coherent with the user's question.

However, the current prototype also presents limitations that must be acknowledged.

The most significant challenge encountered during development was the extraction of **tabular data** and **visual content** from ESG documents.

These documents frequently contain essential information presented in tables, but the layouts vary widely and are often embedded as graphical elements within the PDF.

As a result, standard text extraction tools typically fail to preserve these internal structure.

Several existing libraries were tested to address this issue.

PyPDF2 was evaluated for structured table parsing, and PyTesseract was applied to extract text from embedded images using OCR techniques.

We also experimented with PyMuPDF (Fitz), which offers low-level access to layout information through coordinates: an approach that required substantial code customization to extract tables based on the spatial distribution of text blocks.

Despite all these efforts, the solutions proved insufficient: PyPDF2 often failed to recognize rows and columns when the formatting deviated from standard layouts, while OCR tools struggled with low-contrast or distorted visual data.

Even with the flexibility offered by PyMuPDF, table extraction remained unreliable due to the extreme variability and graphical nature of table designs across documents, a challenge that becomes particularly relevant in the ESG domain, where key information are often hidden in semi-structured tables with up to twenty different layouts across documents.

A possible solution would have required developing a fully customized table extraction library, capable of interpreting visual structure, aligning headers with values, and adapting dynamically to different layout patterns.

Due to the high complexity and limited time constraints, such component was not implemented in the current version of the system.

Nonetheless, this limitation currently represents the main bottleneck in making the pipeline fully reliable across all sections of an ESG report.

Moreover, although the system supports multilingual input and output, its performance on idiomatic or highly **informal language** remains largely untested and may require further adaptation in real-world applications.

4.3 Potential Future Developments

Several directions could be pursued to further improve the system and bring it closer to real-world application.

One particularly promising area is the development of a user **graphical interface**.

Up to this moment, the pipeline operates exclusively through scripts and configuration files, requiring technical expertise to launch queries, inspect intermediate results, or

modify parameters.

This approach limits accessibility, especially for small and medium enterprises that may not have dedicated technical employees.

Creating a simple and intuitive graphical interface, such as a web-based dashboard or chatbot frontend, would make the system far more accessible.

In this scenario, users could interact with the pipeline as in a normal conversation, submitting questions in natural language and receiving structured, context-aware answers without needing to understand the underlying components.

Through such an interface, users could upload ESG documents, ask questions in natural language, and receive structured, context-aware answers with clear references to the underlying content.

A chatbot frontend, in particular, would reproduce a human-like dialogue, enabling more dynamic interactions such as clarification requests or follow-up questions.

This conversational layer would effectively transform the pipeline into an intelligent assistant for sustainability reporting.

Another important development could be the integration of a **feedback loop**, allowing users to validate or rate the answers they receive.

This feedback could be stored and analyzed to manually refine retrieval strategies or improve prompt design.