

# Company Challenge: Marbet

## RAG-enhanced chatbot

### Introduction

Marbet is a prominent German event management agency known for delivering innovative event strategies for leading brands and companies. To improve the overall guest experience during incentive trips, the company set out to develop an AI-powered chatbot capable of offering instant, reliable assistance to travelers.

### Objectives

The goal of this project was to design and implement a Retrieval-Augmented Generation (RAG) chatbot that could provide event-specific support based on Marbet's internal documentation. The chatbot was developed from scratch using Python, LangChain, and Ollama, with a focus on accurately retrieving and presenting relevant information in a helpful and user-friendly manner.

### Chatbot Design

The architecture of my RAG-enhanced chatbot is based on a modular pipeline designed to retrieve and generate context-specific responses using Marbet's internal documentation. The core components are structured as follows:

- Document Loading**  
All relevant text files are loaded from a local directory using `DirectoryLoader` and `TextLoader`. These documents form the knowledge base for the chatbot.
- Text Chunking**  
The loaded documents are split into overlapping text chunks using `RecursiveCharacterTextSplitter`. A chunk size of 1000 characters with 100-character overlap ensures the preservation of context across boundaries.
- Embedding and Vector Store**  
The processed chunks are embedded using the `mxbai-embed-large` model from Ollama and stored in a FAISS vector store. This enables efficient semantic search based on user queries.
- Retriever with MMR Search**  
The chatbot uses the FAISS vector store as a retriever with Maximal Marginal Relevance (MMR) to return the top 10 diverse and relevant chunks for each query.
- Prompt Template and LLM**  
A custom prompt template is defined to guide the chatbot's responses, emphasizing accuracy, structure, and document-grounded answers. The `llama3.3:70b-instruct-q5_K_M` model from Ollama is used to generate responses based on the retrieved chunks.
- Interaction Loop**  
The chatbot runs in a loop, accepting user questions, retrieving the most relevant document segments, and generating a well-structured response based strictly on the provided information.

### Prompt Engineering

To build a RAG-enhanced chatbot capable of answering diverse questions based on documents

provided by Marbet, I had to design a well-structured prompt. The prompt serves as a configuration for the chatbot, defining its behavior, tone, and the way it utilizes retrieved information. I instructed the model to act as an AI travel assistant, helping users understand trip details while strictly adhering to the source documents. This prompt ensures that the chatbot delivers accurate and context-aware answers while minimizing the risk of hallucinations and preserving a professional tone aligned with Marbet's standards.

After several iterations, the following prompt template proved most effective in generating accurate, well-structured, and user-friendly responses:

```
template = '''
You are an AI travel assistant for an exclusive incentive trip, specialized
in providing detailed and accurate
information based only on the selected event document.

Rules:
- If you cannot find the answer directly in the provided text chunks,
politely say you don't know and encourage the user
to rephrase or ask differently.
- If the user's question is too broad (e.g., "Tell me everything about the
trip"), provide a helpful summary of the most
relevant points you can find, but stay within the retrieved information.
- Never invent, guess, or add information that is not explicitly stated in
the document.
- When answering, be clear, detailed, and as precise as possible, helping the
user fully understand the event
activities, logistics, or requirements.
- Prefer structured, organized answers (lists, steps, or short paragraphs)
when possible to make the information easier
to follow.

You must rely solely on the following retrieved information:
{documents}

User's question:
{question}
'''
```

## Knowledge Base Structuring

At the start of the project, the documents provided by Marbet were poorly formatted and inconsistent, making them unsuitable for direct use in a RAG-based system. For example, the file *Tutorial\_ESTA.pdf* contained a mix of untranslated German text embedded as images, corrupted characters (e.g., “/\*” him e P 200”), and colored text with grammatical errors, all of which hindered both readability and machine processing. Similarly, *SPA\_brochure.pdf* consisted entirely of a scanned page, making it inaccessible to standard text extractors.

Another issue was inconsistency in the structure of content. For instance, the *Information\_A-Z\_Scenic\_Eclipse\_I.pdf* file, which was supposed to organize information alphabetically, included misaligned entries—such as luggage being listed under “G” and coat hangers under “K”—likely due to a flawed translation or formatting error.

To address these issues, I decided that PDFs were not the ideal format for the chatbot. Internally, PDF files store each character with positional coordinates, rather than in logical reading order, making structured extraction very challenging. As a solution, I converted all PDFs into cleaned and well-formatted `.txt` files, which are much more suitable for language models. This format ensures sequential, readable text and greatly improves retrieval consistency.

Despite using a strong retriever, some documents still required manual correction. For example, *Guest\_WiFi.pdf* included embedded images of buttons, which led to missing words and jumbled sentence order during extraction. In this case, only a few lines needed correction. However, *Information\_A-Z\_Scenic\_Eclipse\_I.pdf* had such poor formatting that I had to manually reconstruct the entire file into a structured `.txt` version.

To convert scanned documents like *SPA\_brochure.pdf* into usable text, I used Tesseract OCR. This tool extracts text from images while preserving structure. I executed the following terminal command for this:

```
tesseract img_path output_path --oem 3 --psm 6
```

Once all text was converted and cleaned, I moved on to chunking the content for embedding. For this to work effectively, the text had to be logically structured. For example, in *Activity\_Overview.txt*, the original document listed a date and place only once at the top of the section, followed by multiple activities. When split into chunks, only the first would contain the date, leaving others contextless. To solve this, I duplicated the date and location in front of each activity to preserve context during retrieval.

Additionally, I inserted consistent separators like `---` and double newlines (`\n\n`) between sections to guide the chunking process cleanly and reduce information loss.

After these preprocessing steps, I performed extensive testing, corrected grammatical errors, removed embedded images, restructured poorly formatted sections, and confirmed that all documents were ready for use in the chatbot pipeline.

## Testing & Results

Throughout the project, I went through five major iterations of the chatbot, each bringing valuable insights and improvements. Below is a summary of each iteration, the issues encountered, and the changes that led to the final version.

### Iteration 1

In the initial version, I used the `llama3.2:latest` model as the LLM and stored vector embeddings using ChromaDB with LangChain. This phase primarily focused on experimenting with chunking strategies and parameter tuning. However, the retriever consistently struggled to locate the correct text segments, leading to many unanswered queries or irrelevant responses.

### Iteration 2

To address the retrieval issue, I implemented **semantic routing**. This approach introduced a document selection step, where the user query was embedded and compared with predefined document descriptions. The most relevant document was selected, and the response was generated solely from its content. While this helped isolate the source of information, it introduced a new issue: the router often selected the wrong document due to focus on irrelevant query terms. Because my prompt was designed to avoid hallucinations, the chatbot would remain silent rather than attempt an incorrect answer—resulting in frequent non-responses.

### Iteration 3

Still believing in the potential of semantic routing, I modified the architecture to remove embedding-based retrieval altogether. Instead, I passed the full selected document directly to the model. This aimed to eliminate retrieval errors and give the model more context. However, the approach proved inefficient—routing inaccuracies persisted, and large documents led to long response times and high computational load. The solution was not scalable.

### Iteration 4

I upgraded the model to `llama3.1:8b-instruct-q8_0` (8 billion parameters) for better performance and returned to embedding-based retrieval. I also experimented with the native ChromaDB client instead of using LangChain's wrapper. While the model performed slightly better, the retriever still failed to consistently fetch the right chunks. Moreover, broad or complex queries such as *"Can you tell me about all activities on this trip?"* remained difficult for the model to answer accurately.

### Iteration 5 (Final Version)

The final version delivered the best results. I upgraded to a significantly more powerful model, `llama3.3:70b-instruct-q5_K_M`, and replaced ChromaDB with **FAISS** (Facebook AI Similarity Search) for vector storage. FAISS offered faster indexing and was simpler to integrate. I also enabled **Maximal Marginal Relevance (MMR)** as the search strategy. MMR improves retrieval quality by balancing relevance and diversity, avoiding redundant chunks that cosine similarity might select. This combination of improvements significantly boosted the chatbot's performance—it could now handle nearly all question types with high accuracy and contextual relevance.

To complete the solution, I built a simple user interface using the **Gradio** library. This allowed for a clean, intuitive interaction with the chatbot and made it accessible as a local web application. The UI can be launched directly from the project's repository via terminal.

## Future Improvements

Although the chatbot performs well in its current form, several areas remain for potential enhancement—ranging from how documents are created to functional improvements in the chatbot itself.

### 1. Document Preparation Guidelines

To improve both performance and development efficiency, future documents should follow these guidelines:

- **Avoid images:** Replace visual content with descriptive plain text. This improves performance and eliminates complications caused by image-based text or OCR errors.
- **Do not use scanned pages:** Scanned documents (e.g., brochures or forms) drastically increase preprocessing time and often require manual correction. Text-based files are more efficient and reliable.
- **Prefer .txt or Markdown over PDF:** Unlike PDFs, which store text based on visual layout rather than logical flow, `.txt` and `.md` formats ensure clean and consistent reading order—making them ideal for RAG systems.
- **Use consistent structure and separators:** Documents must be prepared for chunking. Use clear separators (e.g., `---` for major sections and `\n\n` for sub-sections) to guide the chunking algorithm. This helps preserve semantic boundaries and makes retrieval more effective.
- **Tune chunk size and overlap:** Choosing the right chunk size is essential. Too small, and context is lost; too large, and the

model may exceed input limits or retrieve irrelevant content. In this project, I used a chunk size of 1000 and an overlap of 100, but this may vary depending on document complexity.

- **Include one-chunk summaries:**

Adding short, standalone summaries at the end of documents improves the chatbot's ability to respond to broad questions by ensuring that high-level information is easily retrievable.

### **Ideal Document Structure:**

*INTRODUCTION*

---

*Topic 1: description 1 ...*

*Topic 1: description 2 ...*

---

*Topic 2: description 1 ...*

*Topic 2: description 2 ...*

---

*SUMMARY*

## **2. Chatbot Feature Enhancements**

Due to time constraints and the novelty of the task, I was unable to implement all planned features. Below are some key improvements for future development:

- **Add conversation history:**  
Integrate memory so the chatbot can refer to previous interactions, enabling more coherent, multi-turn conversations.
- **Provide source links:**  
Include links to the original documents for transparency and user trust—especially critical for documents involving legal or visa-related information.
- **Query rewriting/expansion:**  
Automatically rephrase vague queries to better align with document phrasing, increasing the chance of retrieving relevant chunks.
- **Advanced prompt engineering:**  
Continue refining prompts to improve tone, structure, and handling of edge cases (e.g., ambiguous or contradictory queries).
- **Real user testing:**  
Evaluate performance with real user queries to identify weak spots—especially around date recognition, location matching, and vague questions—and refine both prompt and retrieval logic accordingly.


## **Conclusion**

While this project was often challenging and time-consuming, it proved to be a highly rewarding learning experience. I successfully built a Retrieval-Augmented Generation (RAG) chatbot capable of providing event-related assistance using Marbet's internal documentation. The chatbot was developed

from scratch using Python, LangChain, and Ollama, and demonstrated its ability to retrieve and deliver accurate, context-specific information in a helpful and user-friendly manner.

This project clearly shows that it is entirely possible to build a high-quality RAG chatbot. However, achieving reliable and consistent performance requires careful planning, thorough document preparation, and sufficient time for testing and refinement. With slightly more time and effort, even more robust, scalable, and production-ready systems can be developed.

## References:

1. LangChain: <https://python.langchain.com/docs/introduction/>
2. Ollama: <https://ollama.com/>
3. Chroma: <https://docs.trychroma.com/docs/overview/introduction>
4. FAISS documentation: <https://faiss.ai/>
5. Gradio Python Client: <https://www.gradio.app/docs/python-client/introduction>
6. RAG From Scratch:  RAG From Scratch: Part 1 (Overview)