1. Image Extraction with PyMuPDF (fitz)

The first part of the project focused on collecting all available information from the Marbet event PDFs. Since the documents were not just text-based but also contained visual elements—like maps, diagrams, or pictograms—I needed a way to extract those images. I used fitz (PyMuPDF), a lightweight and fast PDF processing library that allows easy access to embedded images inside PDF pages.

I wrote a loop that goes through each PDF file, page by page, and uses get_images(full=True) to detect every image object. For each image, I extracted its raw bytes, identified the format (like JPEG or PNG), and saved it with a filename that included the original PDF name and page number. This step was essential because some visual information in the event documents may later be used either for manual inspection or to integrate visual data into an extended version of the chatbot.

Without this step, the chatbot would've missed out on potentially critical visual-only details, like layout-based instructions or hotel maps that are not described in the text.

---

2. Text Extraction and Cleaning with PyPDFLoader

The next major step was extracting the textual content from the PDFs. I used LangChain's PyPDFLoader for this, as it provides a convenient wrapper to load full PDFs and return each page's content as a document object. The reason I chose this over basic PDF readers like PyPDF2 or pdfminer is that PyPDFLoader integrates directly into LangChain workflows and returns structured content ready for chunking and embedding.

In the script, I loaded each PDF, cleaned up whitespace, and stored the output as .txt files inside an extracted_textsfolder. This separation of text per document allowed me to easily review or replace individual sections if needed. Keeping each source text file independent also helped in the later step where I selectively merged specific content into a single file. Overall, this created a clean dataset that the chatbot could later understand, instead of parsing raw, messy PDFs in real time.

---

3. Merging Selected Text Files into One (combined_txt.txt)

After all the cleaned .txt files were created, I manually selected the most important ones that users might ask about—like the guest WiFi access instructions, spa and wellness services, event checklists, or information about entering Canada. I didn't want to feed the chatbot unnecessary data, so this filtering step helped focus the AI on only relevant knowledge.

The script loops through each selected file, appends its contents into a single output file (combined_txt.txt), and adds a header like --- Checklist --- before each section. These headers aren't just cosmetic; they make it easier for both humans and machines to distinguish content sources during debugging or testing. Later, when this large file is split into chunks, these headers help preserve semantic boundaries between unrelated topics.

This merging step was basically a way to reduce file management overhead and allow me to embed and chunk one consistent source, instead of dealing with multiple documents.

---

4. Loading and Splitting the Merged Text into Chunks

Once I had the combined_txt.txt file ready, I loaded it using LangChain's TextLoader, which turns raw text into a format suitable for document chunking. Then, I used RecursiveCharacterTextSplitter, which is ideal for long continuous text with varied structure. I set the chunk_size to 500 characters with an overlap of 50 characters between chunks.

The overlap is important. It ensures that if a key piece of information is at the boundary of two chunks, both chunks retain part of that context. Without overlap, the model might miss critical links between sentences that are split into separate chunks. Chunking is a foundational step because it's these chunks that get embedded and indexed in the vector store. If the chunks are too small, meaning gets lost; if they're too big, they may exceed token limits or include unrelated content.

---

5. Creating Embeddings and Vector Store

To enable efficient semantic search, I converted each chunk into an embedding using the OllamaEmbeddings class. I chose the mxbai-embed-large:latest model for generating embeddings because it is optimized for accuracy and integrates well with LangChain and Ollama. These embeddings are essentially high-dimensional vector representations of each text chunk that preserve semantic meaning.

I stored these embeddings using DocArrayInMemorySearch, which is a lightweight vector store implementation that lets me simulate real retrieval without needing external databases like FAISS or Pinecone. This was particularly useful during testing and development since it runs entirely in memory, making it fast and easy to reset.

The final retriever object lets me perform similarity searches: when a user asks a question, it compares the question's embedding to all document chunks and retrieves the ones that are most semantically similar.

---

6. Defining the Chatbot Prompt Template

To make sure the AI responded consistently and accurately, I designed a custom prompt using PromptTemplate. The prompt sets clear behavioral expectations for the AI. It explicitly tells the assistant to only answer using the given context, be concise, and avoid guessing. I also included fallback instructions—if the answer isn't found, the chatbot should politely refer the user to Marbet's official contact information.

This prompt helps reduce hallucinations (i.e., the AI making up answers) and aligns the tone with Marbet's professional brand. Defining this kind of structured prompt is essential in retrieval-augmented generation (RAG) systems to maintain trustworthiness and focus.

---

7. Running Retrieval + Response Loop

Finally, I wrote a test loop with a few example questions, like "how to alert the medical team?" and "where is the pool?". For each question, I used the retriever to find the most relevant document chunks from the embedded content. If no relevant context was found,

I triggered the fallback response from the prompt. Otherwise, the context and question were passed into the LLMChain with the defined prompt, and the AI generated a short, controlled answer.

This final loop confirmed that everything was working end-to-end: from raw PDF input, to cleaned and embedded text, to a chatbot that could semantically retrieve answers based only on the data I gave it.