What method or library did you use to extract the text, and why? Did you face any formatting challenges with the PDF content?

I used pytesseract for OCR-based text extraction and pdf2image to convert PDF pages into images. The OCR language was explicitly set to Bengali (lang='ben'), making it suitable for processing Bangla text from scanned PDFs.

Bengali PDFs often contain non-standard encodings or scanned content, making traditional PDF parsers like llama index's PyMuPDF or PyPDF2 unreliable. Hence, OCR ensures that the visible text is extracted regardless of font or encoding.

I observed unwanted characters like "◌া", "□", broken joint letters, and common OCR artifacts in Indic scripts. OCR did not correctly handle tables, so you complemented this using Camelot to extract tables separately and append them to the text.

Cleaning Bengali text from the document to make it easier for the AI model to understand and answer questions correctly. Weird invisible characters (that can confuse the model), Extra newlines, and unnecessary spaces.

What chunking strategy did you choose (e.g. paragraph-based, sentence-based, character limit)? Why do you think it works well for semantic retrieval?

I used CharacterTextSplitter with a chunk_size=500 and chunk_overlap=100. This method splits the text into fixed chunks of 500 characters, each overlapping the previous one by 100 characters to retain context.

Although RecursiveCharacterTextSplitter is generally considered a more innovative approach—because it attempts to split text hierarchically (paragraphs → sentences → words → characters)—it did not work well in this case.

The reason lies in my input: the source text was extracted using OCR from a scanned Bengali PDF. OCR output often includes irregular formatting, broken punctuation, and inconsistent sentence boundaries. Since RecursiveCharacterTextSplitter relies on clean punctuation (like . or \n\n) to determine split points, it produced poorly structured chunks—either too short or fragmented in meaning.

In contrast, CharacterTextSplitter does not depend on punctuation. It offers a more predictable and stable chunking approach, especially when dealing with noisy, unstructured text like this. This made it more effective for semantic retrieval, ensuring that each chunk remains meaningful and contextually relevant despite imperfections in the source.

What embedding model did you use? Why did you choose it? How does it capture the meaning of the text?

I used bge-m3, a multilingual embedding model developed by BAAI (Beijing Academy of AI) developed. It supports over 100 languages, including Bangla, which made it a strong fit for this project. I chose bge-m3 because:
-It handles multiple languages well, including Bangla and English.
-It performs strongly on retrieval and question-answering tasks.
-It ranked highly on the MTEB benchmark, which tests text understanding.

Under the hood, bge-m3 is built on a Transformer model (similar to BERT). It takes a sentence, runs it through several attention layers, and generates a set of token-level outputs.

These outputs are averaged (mean pooling) to produce a single vector (embedding) for the entire sentence. That vector captures the overall meaning of the input — not just the individual words, but how they relate to each other. I also enabled normalize_embeddings=True, which ensures the vectors are well-scaled for comparison. This is useful when trying to match user queries with the most relevant parts of the document.

So overall, bge-m3 helps turn both the question and the document chunks into numerical forms that can be compared effectively — even across languages like Bangla and English.

How are you comparing the query with your stored chunks? Why did you choose this similarity method and storage setup?

I used ChromaDB as the vector store and performed similarity search using cosine similarity (the default). The user query is embedded using the bge-m3 model, and ChromaDB retrieves the top 5 most similar document chunks based on vector similarity.

ChromaDB is fast, local, persistent, and integrates easily with LangChain. Thanks to the multilingual embedding model, it's effective for both Bengali and English. Cosine similarity works well for comparing normalized vectors (which I ensured using normalize_embeddings=True), making it ideal for retrieving meaningfully related chunks even when phrased differently.

How do you ensure that the question and the document chunks are compared meaningfully? What would happen if the query is vague or missing context?

I decided to print the chunks in the backend (print(f" --- Chunk {i+1} ---\n{doc.page_content}")) for manual inspection during testing and debugging. The selected chunks are then joined to form the context passed into the final prompt given to the LLM. This context is normalized using a custom function to remove invisible Unicode characters (such as \u200b, \u200c, etc.) commonly appearing in Bengali OCR output.

The retriever may return partially related or off-topic chunks if the user query lacks clear keywords or context. However, after observing how the model handled such low-relevance inputs, I intentionally left the LLM's behavior open-ended. Since the task involves both Bengali and English—and some user queries may not align precisely with how the book phrases things—I wanted to explore how the model would navigate these challenges.

I conducted multiple trials with different versions of the system prompt. I aimed to ensure the model responds meaningfully using the retrieved content, without producing peculiar or irrelevant outputs. As mentioned earlier, I iterated through several versions and have now finalized a system prompt that best balances the bilingual nature of the system.

Do the results seem relevant? If not, what might improve them (e.g. better chunking, better embedding model, larger document)?

For well-written Bangla (Bengali) questions, the results seem relevant enough for a simple RAG setup. Adding table data using Camelot helped make the answers more factually accurate.

There's still room to improve. Instead of splitting the text randomly, we could try cutting it at proper places like the end of sentences or paragraphs. This might be done using tools like spaCy (if it works for Bangla), or even with simple rules based on punctuation.

OCR often adds invisible or broken characters, especially in Bangla. Using something like clean_bengali_text() along with Unicode cleanup can help fix that. It's also helpful if the original PDF is clear and well-structured—good scans with readable text make the OCR output much better.

The structure of the final text file matters too. Keeping section titles, spacing, and table formatting close to the original book helps the system understand the content more accurately. A quick manual check to fix common OCR mistakes, like misspelled words or missing punctuation, also makes a difference.

Adding small clues like short summaries or labels with each chunk can help the system get a better idea of what the chunk is about, making the answers even more useful.