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

Used Methods:

- PyPDF2.PdfReader – first attempt
- pdfminer.high_level.extract_text – fallback when PyPDF2 fails

Why:

- **PyPDF2** is lightweight and works well for many PDFs, especially when text is well-structured.
- **pdfminer** is more robust and can extract content from PDFs where PyPDF2 fails (e.g., scanned PDFs or those with more complex encodings).

Formatting Challenges:

  - Loss of line breaks and paragraph structure
  - Unexpected whitespace or encoding errors
  - Mixed Bangla and English text misalignment

- These issues can affect chunking and embedding accuracy.

(2) What chunking strategy did you choose? Why does it work well for semantic retrieval?

Used Strategy:

- RecursiveCharacterTextSplitter with chunk_size=10000, chunk_overlap=1000

Why This Works:

- Character-based splitting avoids dependency on poorly preserved sentence or paragraph structure in PDFs.
- Recursive splitting maintains semantic flow across chunks while staying within token limits of most embedding models.
- Overlap of 1000 characters ensures that important context at chunk boundaries isn't lost, improving coherence during retrieval.

---

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

Used Model:

- sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 from HuggingFace

Why This Model:

- It's **multilingual**, making it ideal for your use case (Bangla + English).
- It's lightweight yet powerful, supporting **semantic similarity** tasks efficiently.
- Pre-trained on **paraphrase identification**, so it captures meaning beyond exact words.

How It Captures Meaning:

- Converts text into dense vectors that capture semantic structure.
- Similar meanings → closer vectors in high-dimensional space, even if wordings differ.

---

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

Similarity Method:

- **Cosine similarity** (via FAISS)

Storage Setup:

- FAISS index built from chunk embeddings (FAISS.from_texts())

Why FAISS + Cosine:

- FAISS is optimized for fast **Approximate Nearest Neighbor (ANN)** search, essential for large vector databases.
- Cosine similarity is effective for semantic comparisons in normalized embedding spaces.

Result:

- Efficient and scalable vector search to retrieve semantically closest chunks.

---

5. How do you ensure meaningful comparison between query and document chunks? What if the query is vague or lacks context?

Ensured By:

- Using same embedding model for **both query and chunks**
- Prompt that emphasizes answering **only from retrieved context**
- Chunking with overlap to avoid missing boundary context

If Query is Vague:

- Retrieved chunks may be irrelevant or loosely related.
- The model might answer incorrectly or say "not available in context."

---

6. Do the results seem relevant? If not, what might improve them?

Generally Relevant — but improvement areas include:

Issue	Suggestion
Chunks too large/small	Experiment with chunk_size (e.g., 500–1500)
Noisy or misaligned text	Pre-clean text (e.g., regex for whitespace/headers)
Poor embeddings	Try **larger multilingual models** like LaBSE, or **openai/text-embedding-3-small**
Ambiguous queries	Use **query clarification** or **retrieval augmentation**
Top-k limit	Increase k in similarity_search to retrieve more context