
Parsing Strategy: Trials, Errors, and Decisions

The first step in building the system was to get **high-quality parsing** of the source PDFs. These documents are far from simple — they include **complex tables, infographics, multi-column layouts, and structured headings,** which makes naive approaches like traditional OCR (Optical Character Recognition) a poor fit. Simple OCR would flatten the structure and miss key contextual relationships, especially in clinical tables and charts.

So, I started researching best practices and community recommendations. Reddit proved usefu;l several threads discussed optimal parsing techniques for RAG pipelines. A recurring recommendation was **PaddleOCR**, particularly its **PP-StructureV3** module. This model is **state-of-the-art in open-source document parsing**. It extracts both the text and layout structure and is known to handle complex documents well, including medical and scientific content.

However, I ran into a key limitation: **PaddleOCR is heavy and resource-intensive**. On my local machine, it failed to run consistently due to memory and hardware constraints. Despite its capabilities, it was not viable for my setup.

Pivot to Hosted APIs

Given the hardware constraints, I shifted to exploring **hosted API-based parsing solutions**. I considered two main options:

- **LlamaParse** (by LlamaIndex)
- **Retab** (designed specifically for structured table extraction)

After trying both, I chose **LlamaParse** due to its balance of:

- **Cost-efficiency**
- **Ease of integration**
- **High accuracy**, especially for structured content like tables and sections

LlamaParse performed very well, but it had a limitation: it could only reliably parse PDFs up to ~100 pages per request. One of the PDFs I was working with the Pocket Book of Hospital Care was 438 pages long. To work around this, I **split the PDF into four chunks** of 100 pages each and parsed them individually and the other PDF was only 84 pages so it was ok to be parsed as a single pdf.

Why JSON?

LlamaParse returns a **structured JSON output**, which proved crucial for later stages of the pipeline:

- Each page's content is parsed into markdown (md) and raw text.
- Tables, images, and charts are extracted with accompanying metadata.
- The output is **itemised:** Each page becomes a dictionary containing structured elements like:
    - Headings (type: heading)
    - Tables (type: table)
    - Paragraphs (type: text)
- Most importantly, **page-level metadata** is preserved: page number, source filename, section, etc.

This structure meant that **every chunk** of text we later embed can be **traced back to its source**, enabling precise **citations in the chatbot's answers** something critical in medical QA systems.

Cleaning the JSON

Even though the parsed JSON had everything we needed, it wasn't ready for chunking directly. Each page's dictionary included noisy fields like:

"status", "originalOrientationAngle", "links",
"width", "height", "triggeredAutoMode",
"parsingMode", "structuredData",
"noStructuredContent", "noTextContent",
"pageHeaderMarkdown", "pageFooterMarkdown"

These fields had no impact on the actual content or embeddings. Additionally, **bounding box data for layout (x, y, height, width)** was redundant for a text-based QA system. Including them would not just bloat storage but also risk **diluting semantic clarity during embedding**.

So, I wrote a cleaner script to **strip out all irrelevant metadata**, retaining only:

- The actual content (value and md)
- Page-level metadata (page, source_part)
- Type of content (type, and lvl if applicable)

This gave me a lean, efficient JSONL structure to proceed with for chunking and embedding.

Merging Chunked PDFs (Handling Split Files)

Since I had to split large PDFs into smaller parts for parsing (due to LlamaParse's ~100-page limit), each resulting JSON started page numbering from 1. This meant that when merging the files, I needed to **recalculate and realign the page numbers** so that they matched the original source document.

I created a function that:

- Offset the page numbers appropriately (e.g., PDF part 2 starts from 101 instead of 1)
- Retained the "source_part" or "filename" to track origin
- Created a **unified JSON** representing the full original document

This consolidation was essential because **answers later need to cite the correct page and topic**, and inconsistent page numbers would break traceability.

The **final cleaned structure** for each page became:


```
{
  "content": "Structured markdown content from the page...",
  "metadata": {
    "page": 132,
    "source": "PocketBook_WHO.pdf",
    "topic": "3.14 Doses of common drugs for neonates and low-birth-weight infants"
  }
}
```

Chunking Strategy: Trials & Final Choice

There are many ways to chunk text for a Retrieval-Augmented Generation (RAG) system. Tools like **Vectorize** offer playground to chunking strategies such as:

- By paragraph
- By heading
- By fixed number of tokens

- By semantic breaks

However, after several iterations, I chose to **chunk by page**. Here's why:

- **Medical PDFs are rich with tables**, flowcharts, and figures that must remain intact.
- Chunking by token count could **split structured content**, destroying table integrity or disconnecting figure explanations from their context.
- The average page in the document contained **250-300 words**, which is within acceptable limits for embedding and retrieval without overflow.

Chunking **one dict per page** ensured that:

- Tables and diagrams stayed within a single chunk
- Metadata like page number remained meaningful
- Retrieval quality improved for structure-sensitive queries

The final chunks were saved in a .jsonl file, each line representing one complete, self-contained chunk with rich metadata.

Embedding: Choosing the Right Model for Medicine

Now came the embedding step, converting chunks into **dense vectors** for similarity search. This choice significantly affects RAG performance, especially in a domain like medicine.

I explored multiple options:

- General-purpose APIs like OpenAI or Gemini
- Open-source models like GTE, Mistral
- Specialised biomedical models like **PubMedBERT**, **BioBERT**, and **MedCPT**

Final Choice:

PubMedBERT

I went with **NeuML/pubmedbert-base-embeddings**, a SentenceTransformer model trained specifically on biomedical literature. It's been shown to **outperform generalised embedding models on medical QA and search tasks**.

- It maps sentences and passages into a **768-dimensional vector space**
- Trained on a large random sample of **PubMed abstracts and title pairs**
- Ideal for **semantic search**, clustering, and retrieval tasks in clinical or research-heavy applications

Because it understands medical terminology and sentence structure better than generic models, **PubMedBERT embeddings lead to higher precision in medical document retrieval**.

While further fine-tuning on my own data (e.g. pediatric texts) could improve performance, **the zero-shot results were already impressive**.

Embedding Storage, Retrieval & Follow-up Handling

After generating the embeddings using **PubMedBERT**, the next step was to store them for fast retrieval. I used **FAISS (Facebook AI Similarity Search)**. I used the **IndexFlatL2** index type, which is a brute-force approach based on **Euclidean (L2) distance**.

- While it's not the fastest option for very large-scale datasets, it works well for **smaller, high-precision setups** like mine (~400–500 chunks).
- This setup ensures **exact search**, which is crucial when working with dense vectors and medically sensitive information.

Query Embedding & Retrieval

To handle user queries, I:

1. **Encoded the query** using the same PubMedBERT model, ensuring the embedding space remains consistent with the document chunks.
2. **Searched the FAISS index** using Euclidean distance to retrieve the top-K most relevant chunks.

This worked well, especially because the number of chunks was relatively small, and I prioritised **retrieval quality over speed**.

Tried Reranking with MedCPT (But Dropped It)

While basic retrieval returned relevant chunks, I wanted to **experiment with reranking**. The intuition was:

> Even if PubMedBERT retrieves relevant chunks, their **ranking might not reflect semantic closeness to the specific query**, especially since the model wasn't fine-tuned on my dataset.

So, I brought in **MedCPT**, a medical cross-encoder that takes a (query, passage) pair and scores their semantic match.

I implemented a reranking pipeline:

- Retrieve top-K chunks with PubMedBERT
- Use the **MedCPT cross-encoder** to rerank those chunks by relevance
- Feed the reranked top-N to the LLM

However, this **did not improve results**. In fact, the reranked outputs were **less consistent**, possibly because:

- MedCPT wasn't trained on the style or format of my parsed pediatric PDFs
- The small chunk size and document density worked better without reranking

So I scrapped the reranker and moved on with plain PubMedBERT-based retrieval.

Prompting the LLM

At this point, I had:

- Top-K relevant chunks
- User query
- A base prompt

I passed all of this to **Gemini 2.5 Flash**, the LLM used in this system.

My prompt strategy was simple:

- A minimal, two-line instruction that told the LLM to **answer using only the given context** and **cite source and page numbers**.

I avoided detailed prompt engineering to keep the token count low and reduce latency and hallucination.

The results were good: Gemini was able to consistently reference relevant pages and sections (e.g., “Page 93, Section: 3.14 Doses of common drugs for neonates”).

Evaluation

I manually tested and evaluated the responses using:

- Gemini (chat)
- ChatGPT (GPT-4)
- RAGFlow’s default pipeline

I compared:

- Answer correctness
- Was the reference valid?
- Table/figure awareness
- Responsiveness to follow-ups

The pipeline using my **chunking + PubMedBERT + Gemini Flash** setup performed **on par or better than RAGFlow in terms of accuracy**, particularly for **structured clinical queries**.

Follow-up Question Handling

Basic RAG is **stateless**. But users often ask **follow-up questions** like:

“What if the child also has severe anaemia?”

So I extended the pipeline to handle **contextual continuity**.

Here’s how I tackled it:

1. **Session-based memory**: I used a chat_memory dictionary to store prior turns for each session ID.

2. **Follow-up detection**: For each new query, I checked whether it was a follow-up using a Gemini-powered classifier.

3. **Query reformulation**: If it was a follow-up, I used Gemini again to **rewrite the follow-up as a standalone query**, incorporating previous context.

4. **Chunk reuse**: If relevant, the pipeline reuses previously retrieved chunks or fetches new ones based on the reformulated query.

5. **Log turn**: Each interaction is logged to maintain the chat history.

This architecture allows the system to:

- **Maintain context** across turns
- Avoid hallucination
- Improve user experience in multi-turn medical conversations

---

Future Implementations that can be done

Here are the next goals for the system:

- **GraphRAG Integration**
  Incorporate entity-level graph reasoning to improve factual accuracy and flow, especially in treatment plans or dosage workflows.

- **Custom MedCPT Fine-Tuning**
  Fine-tune MedCPT on my specific domain documents for better reranking across large corpora (50+ PDFs).

- **Persistent Memory Layer**
  Use Redis or a vector DB to maintain chat history across sessions and devices.

- **Show Infographics**
  When responding, highlight figures or images extracted during parsing — especially charts or dosage tables.