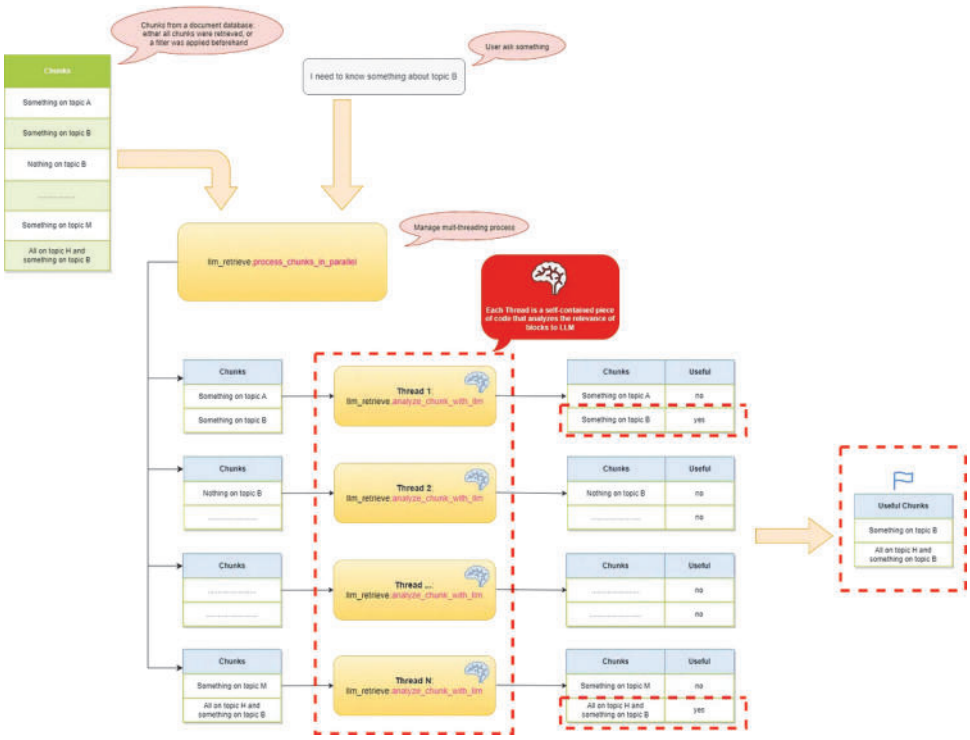
How and Why to Use LLMs for Chunk-Based Information Retrieval

Carlo Peron
Published in Towards Data Science · 9 min read · 1 day ago



Retrieve pipeline — Image by the author

In this article, I aim to explain how and why it's beneficial to use a Large Language Model (LLM) for chunk-based information retrieval.

I use OpenAI's GPT-4 model as an example, but this approach can be applied with any other LLM, such as those from Hugging Face, Claude, and others.

Everyone can access this article for free.

Considerations on standard information retrieval

The primary concept involves having a list of documents (**chunks of text**) stored in a database, which could be retrieve based on some filter and conditions.

Typically, a tool is used to enable hybrid search (such as Azure AI Search, LlamaIndex, etc.), which allows:

- performing a text-based search using term frequency algorithms like TF-IDF (e.g., BM25);
- conducting a vector-based search, which identifies similar concepts even when different terms are used, by calculating vector distances (typically cosine similarity);
- combining elements from steps 1 and 2, weighting them to highlight the most relevant results.

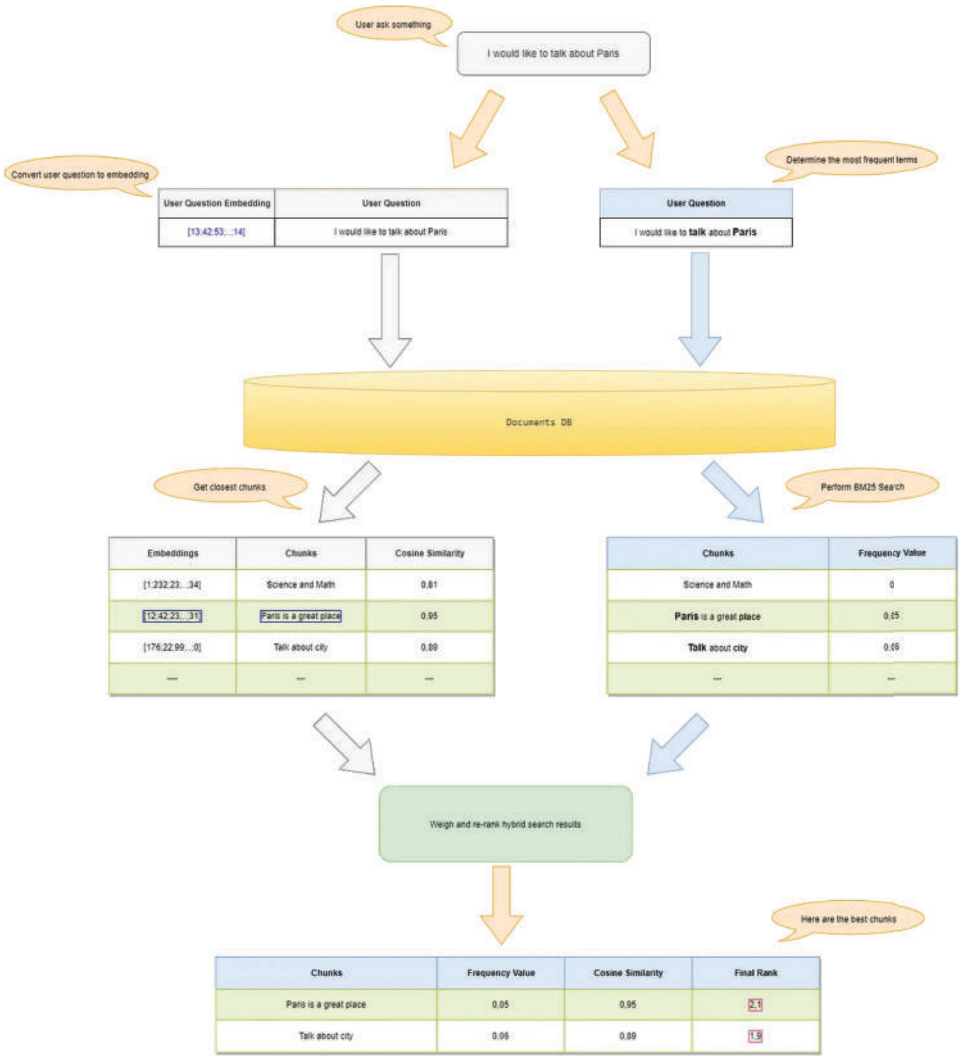


Figure 1- Default hybrid search pipeline — Image by the author

Figure 1 shows the classic retrieval pipeline:

- the user asks the system a question: “I would like to talk about Paris”;
- the system receives the question, converts it into an embedding vector (using the same model applied in the ingestion phase), and finds the chunks with the smallest distances;
- the system also performs a text-based search based on frequency;

- the chunks returned from both processes undergo further evaluation and are reordered based on a ranking formula.

This solution achieves good results but has some limitations:

- not all relevant chunks are always retrieved;
- sometime some chunks contain anomalies that affect the final response.

An example of a typical retrieval issue

Let's consider the "documents" array, which represents an example of a knowledge base that could lead to incorrect chunk selection.

```
documents = [
    "Chunk 1: This document contains information about topic A.",
    "Chunk 2: Insights related to topic B can be found here.",
    "Chunk 3: This chunk discusses topic C in detail.",
    "Chunk 4: Further insights on topic D are covered here.",
    "Chunk 5: Another chunk with more data on topic E.",
    "Chunk 6: Extensive research on topic F is presented.",
    "Chunk 7: Information on topic G is explained here.",
    "Chunk 8: This document expands on topic H. It also talk about topic B",
    "Chunk 9: Nothing about topic B are given.",
    "Chunk 10: Finally, a discussion of topic J. This document doesn't contain i
]
```

Let's assume we have a RAG system, consisting of a vector database with hybrid search capabilities and an LLM-based prompt, to which the user poses the following question: "I need to know something about topic B."

As shown in Figure 2, the search also returns an incorrect chunk that, while semantically relevant, is not suitable for answering the question and, in some cases, could even confuse the LLM tasked with providing a response.

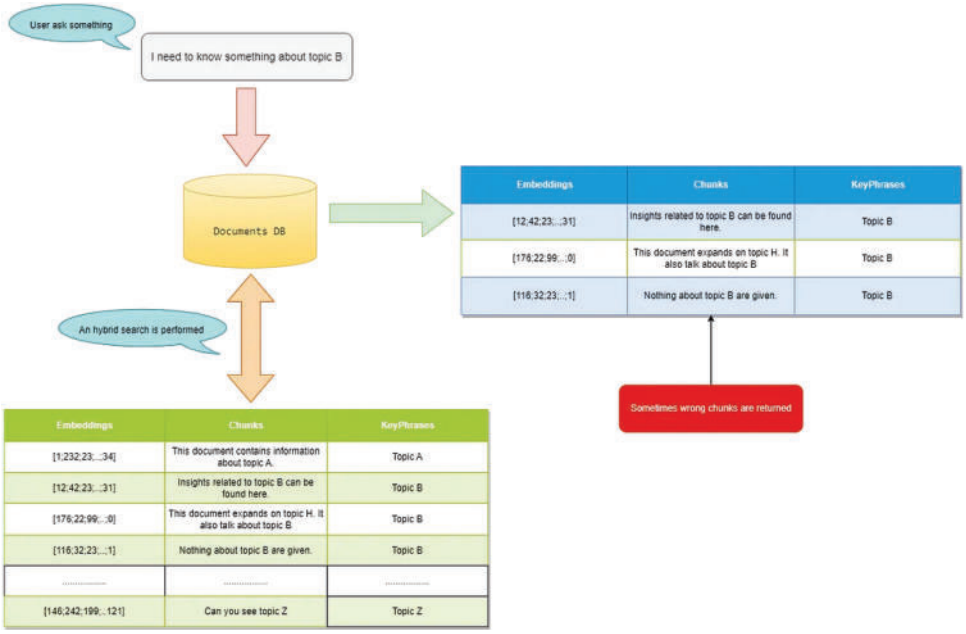


Figure 2 — Example of information retrieval that can lead to errors — Image by the author

In this example, the user requests information about "*topic B*," and the search returns chunks that include "*This document expands on topic H. It also talks about topic B*" and "*Insights related to topic B can be found here.*" as well as the chunk stating, "*Nothing about topic B are given*".

While this is the expected behavior of hybrid search (as chunks reference "*topic B*"), it is not the desired outcome, as the third chunk is returned without recognizing that it isn't helpful for answering the question.

The retrieval didn't produce the intended result, not only because the BM25 search found the term "*topic B*" in the third Chunk but also because the vector search yielded a high cosine similarity.

To understand this, refer to Figure 3, which shows the cosine similarity values of the chunks relative to the question, using OpenAI's text-embedding-ada-002 model for embeddings.

Chunk	User Question	Cosine Similarity	Minkowski
Chunk 8: This document expands on topic H. It also talk about **topic B**	I would like to know something about topic B	0,862848248	0,523739894
Chunk 2: Insights related to **topic B** can be found here.	I would like to know something about topic B	0,859970315	0,529206348
Chunk 9: Nothing about **topic B** are given.	I would like to know something about topic B	0,84654206	0,553999867
Chunk 1: This document contains information about topic A.	I would like to know something about topic B	0,842125311	0,561915814
Chunk 10: Finally, a discussion of topic J. This document doesn't contain information about **topic B**	I would like to know something about topic B	0,841260416	0,563452892
Chunk 3: This chunk discusses topic C in detail.	I would like to know something about topic B	0,822984436	0,595005141
Chunk 7: Information on topic G is explained here.	I would like to know something about topic B	0,820489921	0,599182919
Chunk 4: Further insights on topic D are covered here.	I would like to know something about topic B	0,820184663	0,59969214
Chunk 6: Extensive research on topic F is presented.	I would like to know something about topic B	0,807013953	0,621266534
Chunk 5: Another chunk with more data on topic E.	I would like to know something about topic B	0,806768689	0,62166116

Figure 3 — Cosine similarity with text-embedding-ada-002- Image by the author

It is evident that the cosine similarity value for "Chunk 9" is among the highest, and that between this chunk and chunk 10, which references "*topic B*," there is also chunk 1, which does not mention "*topic B*".

This situation remains unchanged even when measuring distance using a different method, as seen in the case of Minkowski distance.

Utilizing LLMs for Information Retrieval: An Example

The solution I will describe is inspired by what has been published in my GitHub repository https://github.com/peronc/LLMRetriever/.

The idea is to have the LLM analyze which chunks are useful for answering the user's question, not by ranking the returned chunks (as in the case of RankGPT) but by directly evaluating all the available chunks.

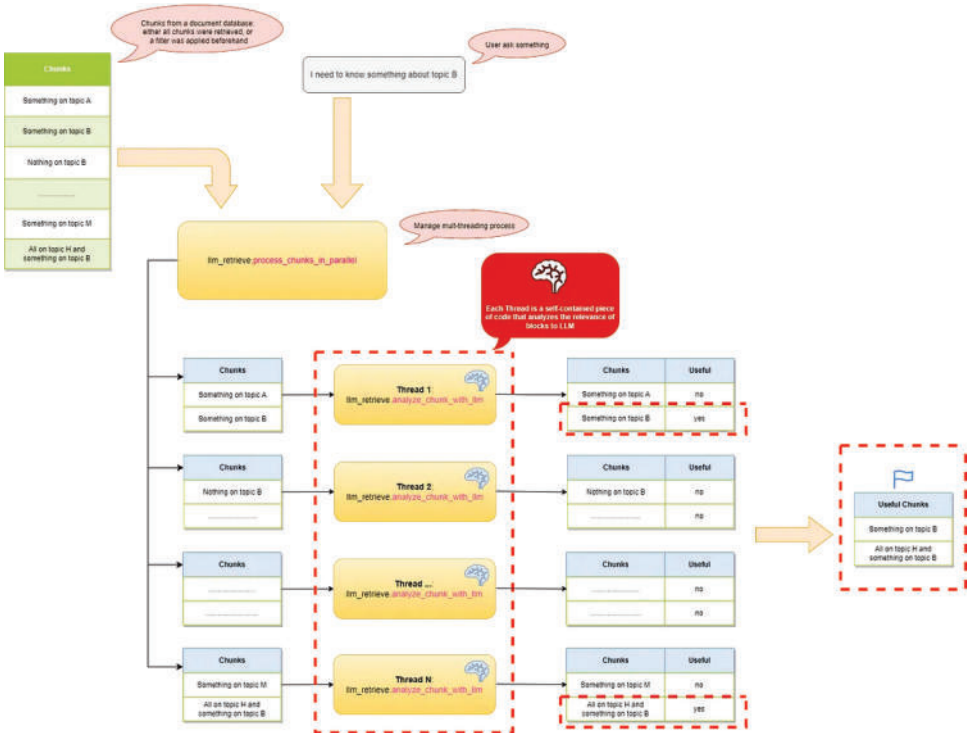


Figure 4- LLM Retrieve pipeline — Image by the author

In summary, as shown in Figure 4, the system receives a list of documents to analyze, which can come from any data source, such as file storage, relational databases, or vector databases.

The chunks are divided into groups and processed in parallel by a number of threads proportional to the total amount of chunks.

The logic for each thread includes a loop that iterates through the input chunks, calling an OpenAI prompt for each one to check its relevance to the user's question.

The prompt returns the chunk along with a boolean value: *true* if it is relevant and *false* if it is not.

Lets'go coding 😊

To explain the code, I will simplify by using the chunks present in the *documents* array (I will reference a real case in the conclusions).

First of all, I import the necessary standard libraries, including os, langchain, and dotenv.

```
import os
from langchain_openai.chat_models.azure import AzureChatOpenAI
from dotenv import load_dotenv
```

Next, I import my LLMRetrieverLib/llm_retrieve.py class, which provides several static methods essential for performing the analysis.

```
from LLMRetrieverLib.retriever import llm_retriever
```

Following that, I need to import the necessary variables required for utilizing Azure OpenAI GPT-4o model.

```
load_dotenv()
azure_deployment = os.getenv("AZURE_DEPLOYMENT")
temperature = float(os.getenv("TEMPERATURE"))
api_key  = os.getenv("AZURE_OPENAI_API_KEY")
endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
api_version = os.getenv("API_VERSION")
```

Next, I proceed with the initialization of the LLM.

```
# Initialize the LLM
llm = AzureChatOpenAI(api_key=api_key, azure_endpoint=endpoint, azure_deployment
```

We are ready to begin: the user asks a question to gather additional information about *Topic B*.

```
question = "I need to know something about topic B"
```

At this point, the search for relevant chunks begins, and to do this, I use the function `llm_retrieve.process_chunks_in_parallel` from the

`LLMRetrieverLib/retriever.py` library, which is also found in the same repository.

```
relevant_chunks = LLMRetrieverLib.retriever.llm_retriever.process_chunks_in_para
```

To optimize performance, the function `llm_retrieve.process_chunks_in_parallel` employs multi-threading to distribute chunk analysis across multiple threads.

The main idea is to assign each thread a subset of chunks extracted from the database and have each thread analyze the relevance of those chunks based on the user's question.

At the end of the processing, the returned chunks are exactly as expected:

```
['Chunk 2: Insights related to topic B can be found here.',
'Chunk 8: This document expands on topic H. It also talk about topic B']
```

Finally, I ask the LLM to provide an answer to the user's question:

```
final_answer = LLMRetrieverLib.retriever.llm_retriever.generate_final_answer_wit
print("Final answer:")
print(final_answer)
```

Below is the LLM's response, which is trivial since the content of the chunks, while relevant, is not exhaustive on the topic of Topic B:

```
Topic B is covered in both Chunk 2 and Chunk 8.
Chunk 2 provides insights specifically related to topic B, offering detailed inf
Chunk 8 expands on topic H but also includes discussions on topic B, potentially
```

Scoring Scenario

Now let's try asking the same question but using an approach based on scoring.

I ask the LLM to assign a score from 1 to 10 to evaluate the relevance between each chunk and the question, considering only those with a relevance higher than 5.

To do this, I call the function `llm_retriever.process_chunks_in_parallel`, passing three additional parameters that indicate, respectively, that scoring will be applied, that the threshold for being considered valid must be greater than or equal to 5, and that I want a printout of the chunks with their respective scores.

```
relevant_chunks = llm_retriever.process_chunks_in_parallel(llm, question, docume
```

The retrieval phase with scoring produces the following result:

```
score: 1 - Chunk 1: This document contains information about topic A.
score: 1 - Chunk 7: Information on topic G is explained here.
score: 1 - Chunk 4: Further insights on topic D are covered here.
score: 9 - Chunk 2: Insights related to topic B can be found here.
score: 7 - Chunk 8: This document expands on topic H. It also talk about topic B
score: 1 - Chunk 5: Another chunk with more data on topic E.
score: 1 - Chunk 9: Nothing about topic B are given.
score: 1 - Chunk 3: This chunk discusses topic C in detail.
score: 1 - Chunk 6: Extensive research on topic F is presented.
score: 1 - Chunk 10: Finally, a discussion of topic J. This document doesn't con
```

It's the same as before, but with an interesting score 😊.

Finally, I once again ask the LLM to provide an answer to the user's question, and the result is similar to the previous one:

```
Chunk 2 provides insights related to topic B, offering foundational information
Chunk 8 expands on topic B further, possibly providing additional context or det
Together, these chunks should give you a well-rounded understanding of topic B.
```

Considerations

This retrieval approach has emerged as a necessity following some previous experiences.

I have noticed that pure vector-based searches produce useful results but are often insufficient when the embedding is performed in a language other than English.

Using OpenAI with sentences in Italian makes it clear that the tokenization of terms is often incorrect; for example, the term "*canzone*," which means "*song*" in Italian, gets tokenized into two distinct words: "*can*" and "*zone*".

This leads to the construction of an embedding array that is far from what was intended.

In cases like this, hybrid search, which also incorporates term frequency counting, leads to improved results, but they are not always as expected.

So, this retrieval methodology can be utilized in the following ways:

- **as the primary search method:** where the database is queried for all chunks or a subset based on a filter (e.g., a metadata filter);
- **as a refinement in the case of hybrid search:** (this is the same approach used by RankGPT) in this way, the hybrid search can extract a large number of chunks, and the system can filter them so that only the relevant ones reach the LLM while also adhering to the input token limit;
- **as a fallback:** in situations where a hybrid search does not yield the desired results, all chunks can be analyzed.

Let's discuss costs and performance

Of course, all that glitters is not gold, as one must consider response times and costs.

In a real use case, I retrieved the chunks from a relational database consisting of 95 text segments semantically split using my `LLMChunkizerLib/chunkizer.py` library from two Microsoft Word documents, totaling 33 pages.

The analysis of the relevance of the 95 chunks to the question was conducted by calling OpenAI's APIs from a local PC with non-guaranteed bandwidth, averaging around 10Mb, resulting in response times that varied from 7 to 20 seconds.

Naturally, on a cloud system or by using local LLMs on GPUs, these times can be significantly reduced.

I believe that considerations regarding response times are highly subjective: in some cases, it is acceptable to take longer to provide a correct answer, while in others, it is essential not to keep users waiting too long.

Similarly, considerations about costs are also quite subjective, as one must take a broader perspective to evaluate whether it is more important to

provide as accurate answers as possible or if some errors are acceptable.

In certain fields, the damage to one's reputation caused by incorrect or missing answers can outweigh the expense of tokens.

Furthermore, even though the costs of OpenAI and other providers have been steadily decreasing in recent years, those who already have a GPU-based infrastructure, perhaps due to the need to handle sensitive or confidential data, will likely prefer to use a local LLM.

Conclusions

In conclusion, I hope to have provided my perspective on how retrieval can be approached.

If nothing else, I aim to be helpful and perhaps inspire others to explore new methods in their own work.

Remember, the world of information retrieval is vast, and with a little creativity and the right tools, we can uncover knowledge in ways we never imagined!

If you'd like to discuss this further, feel free to connect with me on LinkedIn

GitHub repositories can be found here:
• https://github.com/peronc/LLMRetriever/
• https://github.com/peronc/LLMChunkizer/

Chunking | Retrieval | Artificial Intelligence | Llm | Machine Learning

Written by Carlo Peron

Edit profile

58 Followers · Writer for Towards Data Science

You can get more information about me on https://www.linkedin.com/in/carlo-peron