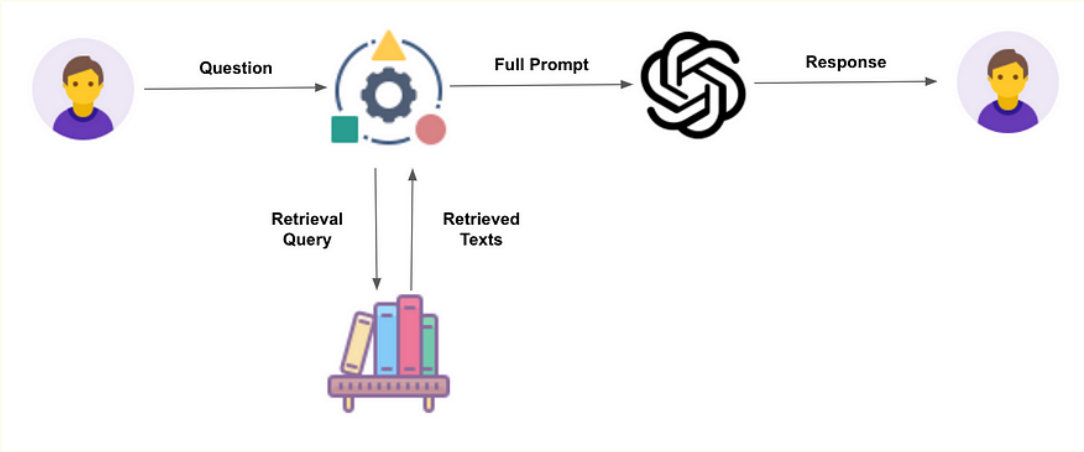
RAG:

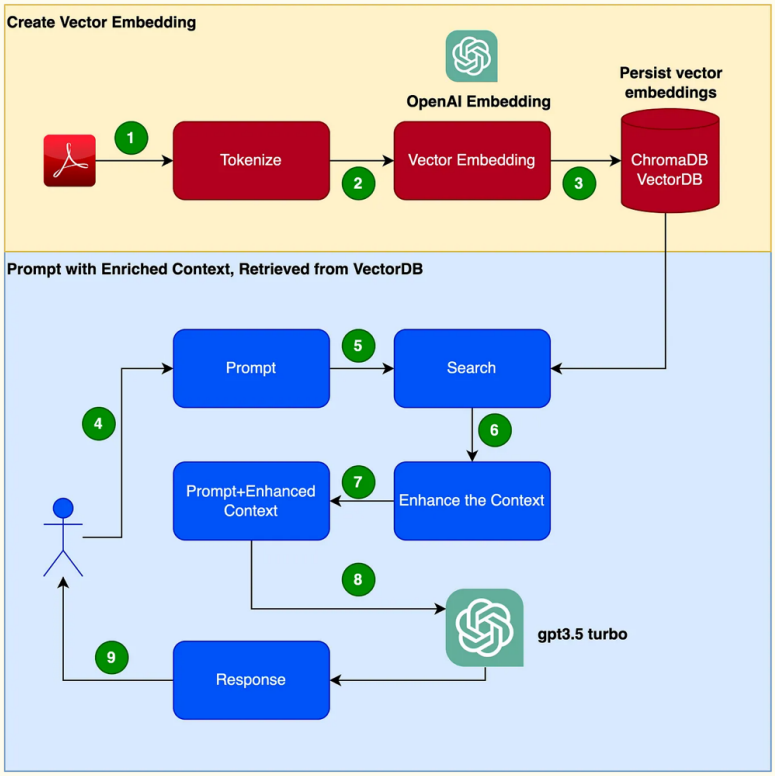


Context:

→ LLMs are powerful, but they suffers from out of knowledge.
  ↳ as training data tends to be out-of-date.

→ LLMs use to extrapolate things when facts are not available and prone to generate wrong answer confidently.

→ RAG: Retrieval Augmented Generation is strategy to address both of these issues.
  → RAG leverages power of retrieval to access relevant information on demand.
  → No need to finetune the model.



1. Read from the PDF (Clarett user manual PDF) and tokenize with a chunk_size of 1000 tokens
2. Create a vector embedding of these tokens. We will be using OpenAIEmbeddings library to create the vector embeddings.
3. Store the vector embeddings locally. We will be using simple ChromaDB as our VectorDB. We could be using Pinecone or any other such more highly available, production-grade VectorDBs instead.
4. The user issues a prompt with the query/question.
5. This issues a search and retrieval from the vectorDB to get more contextual data from the VectorDB.
6. This contextual data is now will be used along with the prompt.
7. The prompt is augmented by the context. This is typically referred to as context enrichment.
8. The prompt along with the query/question and this enhanced context is now passed to the LLM
9. LLM now responds back, based on this context.

An overly simplified example

LangChain has an example of RAG in its smallest (but not simplest) form:

```
from langchain.document_loaders import WebBaseLoader
from langchain.indexes import VectorstoreIndexCreator
loader = WebBaseLoader("https://www.promptingguide.ai/techniques/rag")
index = VectorstoreIndexCreator().from_loaders([loader])
index.query("What is RAG?")
```

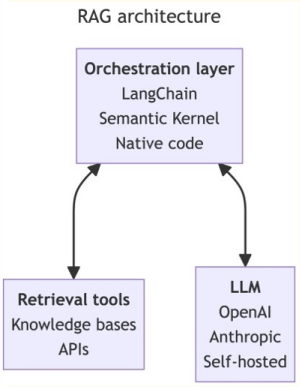
With these five lines, we get a description of RAG, but the code is heavily abstracted, so it's difficult to understand what's actually happening: We fetch the contents of a web page (our knowledge base for this example).

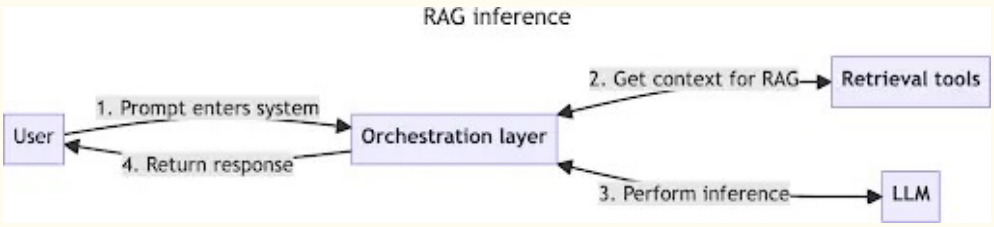
1. We process the source contents and store them in a knowledge base (in this case, a vector database).
2. We input a prompt, LangChain finds bits of information from the knowledge base, and passes both prompt and knowledge base results to the LLM.

Basic architecture

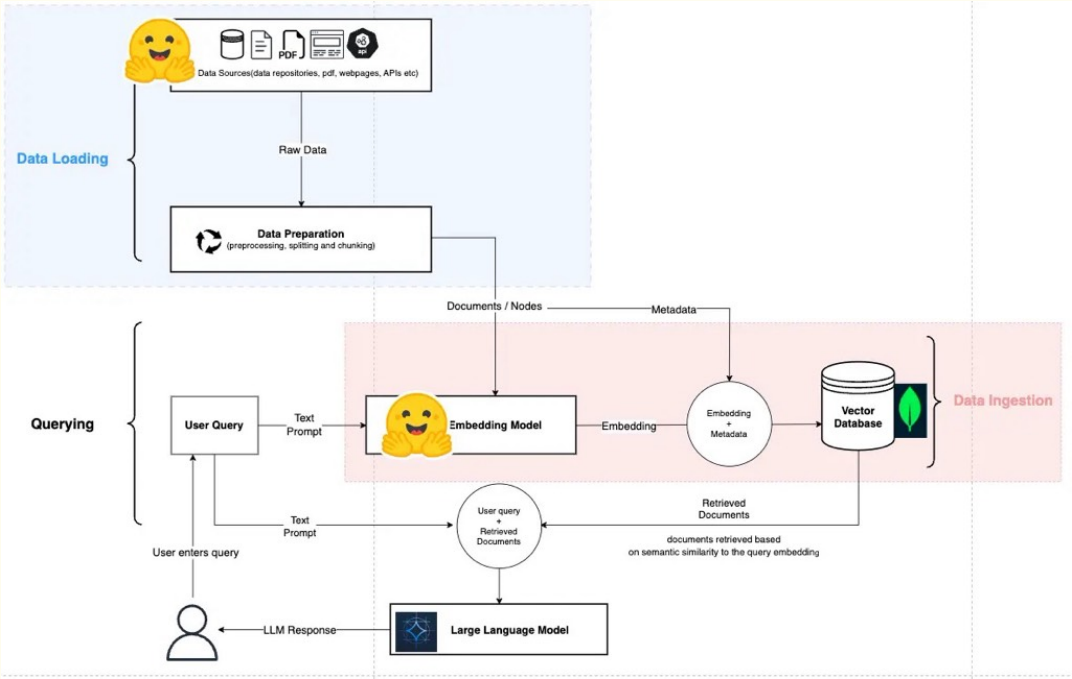
Because a full LLM application architecture would be fairly large, we're going to consider just the components that enable RAG:

- The **orchestration layer** receives the user's input in any associated metadata (like conversation history), interacts with all of the related tooling, ships the prompt off to the LLM, and returns the result. Orchestration layers are typically composed of tools like LangChain, Semantic Kernel, and others with some native code (often Python) knitting it all together.
- **Retrieval tools** are a group of utilities that return context that informs and grounds responses to the user prompt. This group encompasses both knowledge bases and API-based retrieval systems.
- **LLM** is the large language model that you're sending prompts to. They might be hosted by a third party like OpenAI or run internally in your own infrastructure. For the purposes of this article, the exact model you're using doesn't matter.



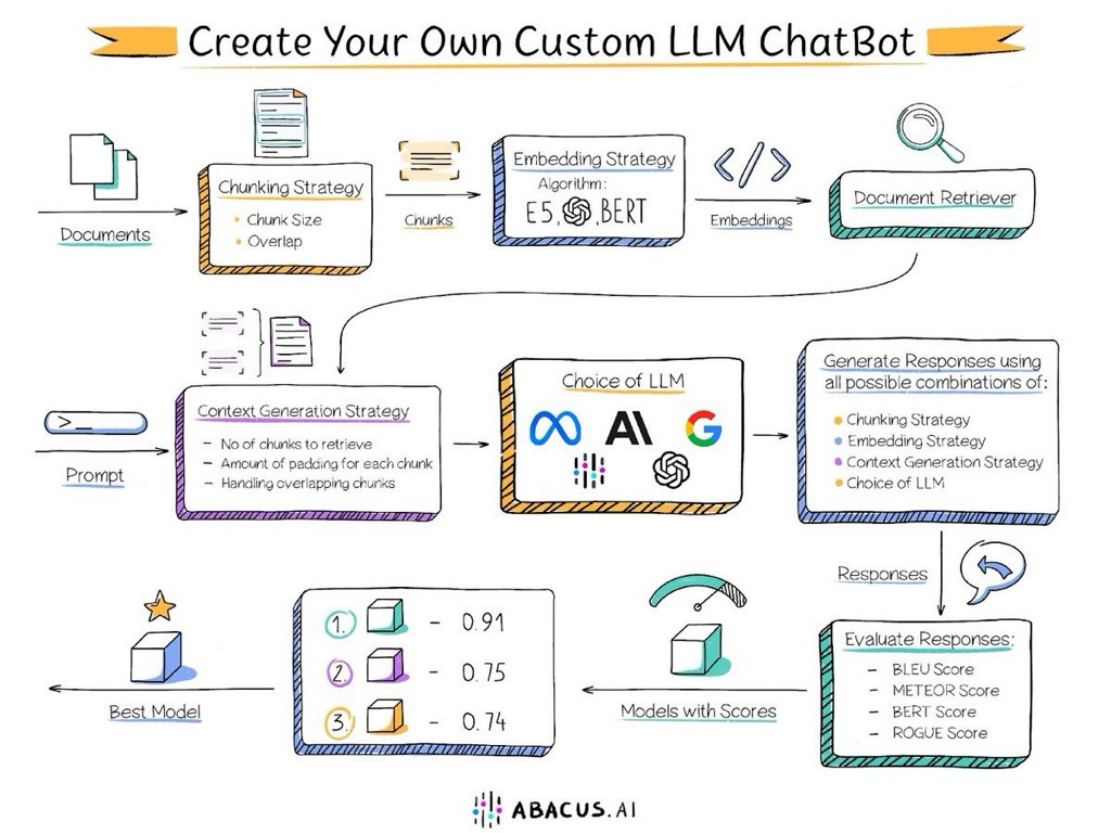


2. **Two-Step Process**: **Retrieval Step**: When presented with a query or prompt, the model first retrieves relevant documents or passages from an external database. This retrieval is typically based on a similarity measure, ensuring that the most pertinent information is fetched.
   Generation Step: After retrieval, the model uses the fetched documents or passages as context to generate a coherent and contextually relevant response.



https://huggingface.co/learn/cookbook/en/rag_with_hugging_face_gemma_mongodb

https://x.com/akshay_pachaar/status/1769341705986211930?s=46



Great overview of how to build your own Document Chat RAG!

Here's what you'll need:

1️⃣ A knowledge base

A collection of relevant and up-to-date information that serves as a foundation for RAG. It can be a database, a set of documents, or a combination of both. In this case it would be the collection of your documents.

2️⃣ Chunking strategy

Chunking is the process of breaking down a large input text into smaller pieces. This ensures that the text fits the input size of the embedding model and improves retrieval efficiency.

Two hyperparameters that you need to take care of are the `chunk_size` & `chunk_overlap`.

3️⃣ Embeddings strategy

The chunked data is converted into embeddings (vector representations), here you need to decide what embedding model to use.

4️⃣ Document retriever

A document retriever is a crucial component that performs the task of searching and retrieving semantically similar documents or chunks based on a given query.

5️⃣ Context generation

Now you need to create a context based on the retrieved, here's what we usually do:

- keep top-k chunks
- put a similarity cutoff
- filtering based in metadata
- reranking chunks using a reranker model

6️⃣ Prompt template

Use a custom prompt template to guide the response from LLM and ensure it aligns with the context provided.

7️⃣ Choose Your LLM

Next, select the LLM you want to use. It will generate the final response based on the prompt you created in the last step.

8️⃣ Evaluating the responses

Now considering all the hyperparameters that we have, we tune them by trying different combinations, here are some of the metrics for evaluating the final response:

- BLEU score
- METEOR score
- BERT score
- ROGUE score

Finally you select the best model & deploy this system into production!