Retrieval-Augmented Generation (RAG) from basics to advanced

Tejpal Kumawat Follow 12 min read · Feb 14, 2024

Introduction:

- Retrieval-Augmented Generation (RAG) is a technique that enhances language model generation by incorporating external knowledge.

- This is typically done by retrieving relevant information from a large corpus of documents and using that information to inform the generation process.

Challenge:

- Clients often have vast proprietary documents.
- Extracting specific information is like finding a needle in a haystack.

2. GPT4-Turbo Introduction:

- OpenAI's GPT4-Turbo can process large documents.

3. Efficiency Issue:

- "Lost In The Middle" phenomenon hampers efficiency.
- Model forgets content in the middle of its contextual window.

4. Alternative Approach — Retrieval-Augmented-Generation (RAG):

- Create an index for each document paragraph.
- Swiftly identify pertinent paragraphs.
- Feed selected paragraphs into a Large Language Model (LLM) like GPT4.

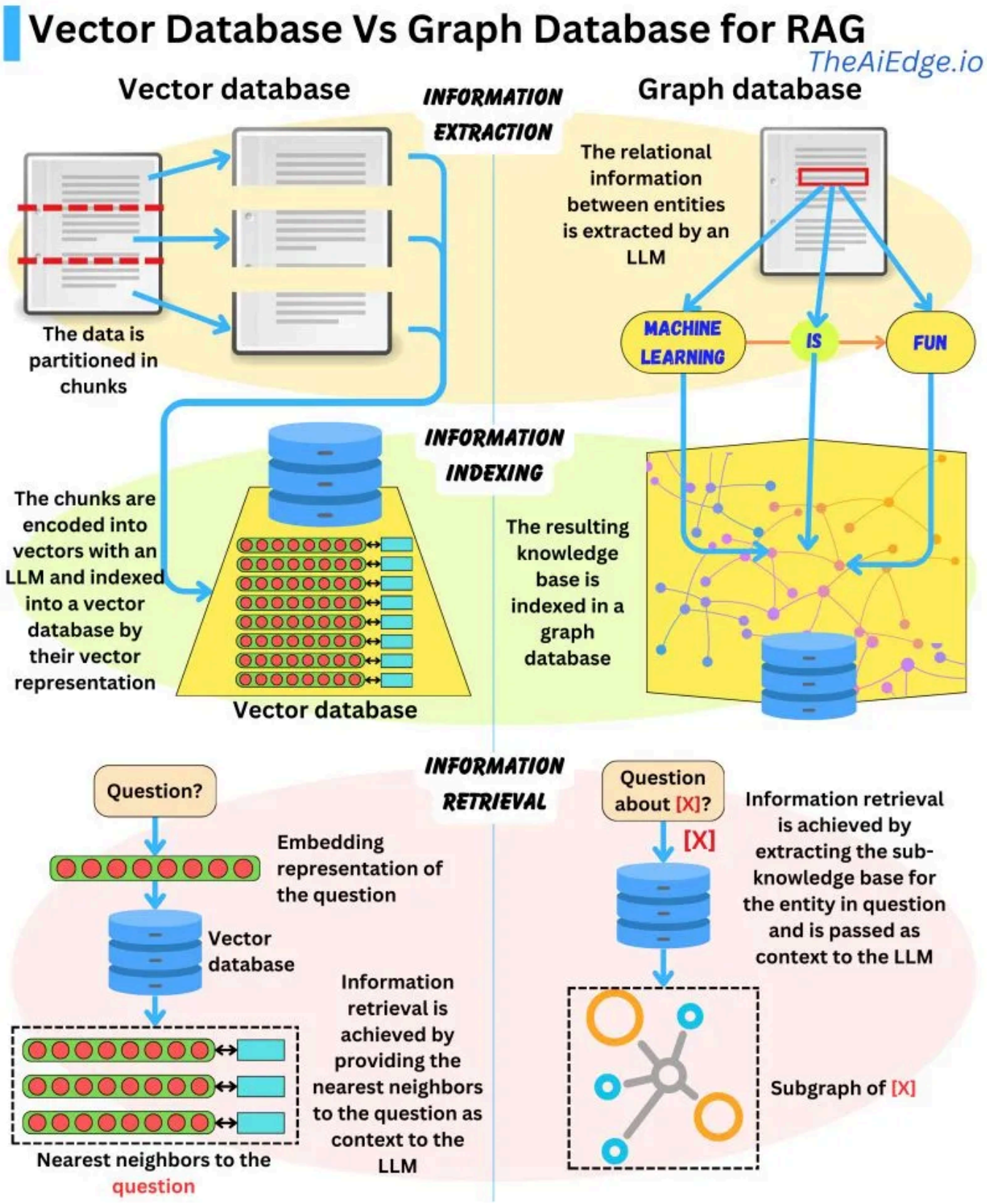
5. Advantages:

- Prevents information overload.
- Enhances result quality by providing only relevant paragraphs.

The Retrieval Augmented Generation (RAG) Pipeline

- With RAG, the LLM is able to leverage knowledge and information that is not necessarily in its weights by providing it access to external knowledge sources such as databases.

- It leverages a retriever to find relevant contexts to condition the LLM, in this way, RAG is able to augment the knowledge-base of an LLM with relevant documents.
- The retriever here could be any of the following depending on the need for semantic retrieval or not:
- **Vector database:** Typically, queries are embedded using models like BERT for generating dense vector embeddings. Alternatively, traditional methods like TF-IDF can be used for sparse embeddings. The search is then conducted based on term frequency or semantic similarity.
- **Graph database:** Constructs a knowledge base from extracted entity relationships within the text. This approach is precise but may require exact query matching, which could be restrictive in some applications.
- **Regular SQL database:** Offers structured data storage and retrieval but might lack the semantic flexibility of vector databases.
- The image below from Damien Benveniste, PhD talks a bit about the difference between using Graph vs Vector database for RAG.



- Graph Databases are favored for Retrieval Augmented Generation (RAG) when compared to Vector Databases. While Vector Databases partition

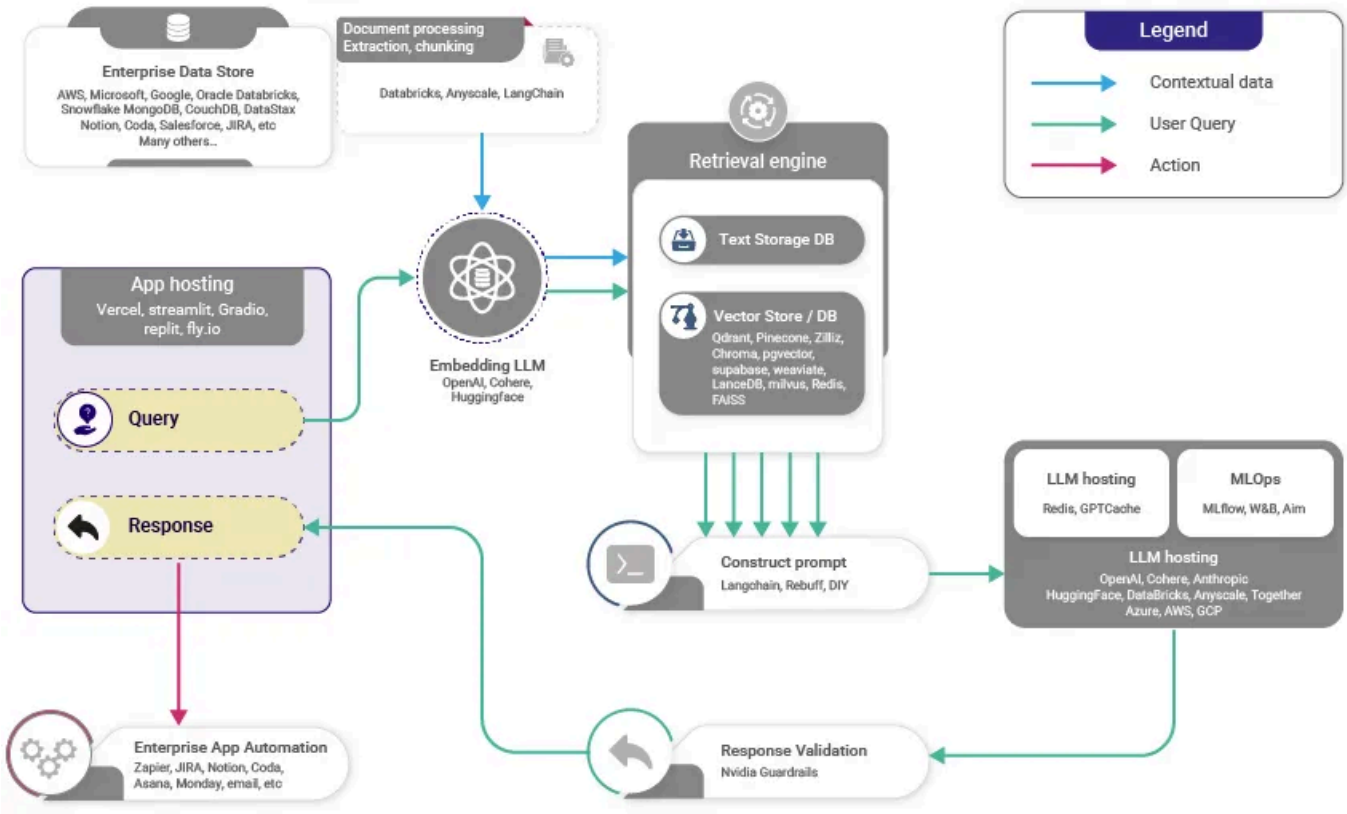
and index data using LLM-encoded vectors, allowing for semantically similar vector retrieval, they may fetch irrelevant data.

- Graph Databases, on the other hand, build a knowledge base from extracted entity relationships in the text, making retrievals concise. However, it requires exact query matching which can be limiting.
- A potential solution could be to combine the strengths of both databases: indexing parsed entity relationships with vector representations in a graph database for more flexible information retrieval. It remains to be seen if such a hybrid model exists.
- After retrieving, you may want to look into filtering the candidates further by adding ranking and/or fine ranking layers that allow you to filter down candidates that do not match your business rules, are not personalized for the user, current context, or response limit.
- Let's succinctly summarize the process of RAG and then delve into its pros and cons:

1. **Vector Database Creation**: RAG starts by converting an internal dataset into vectors and storing them in a vector database (or a database of your choosing).
2. **User Input**: A user provides a query in natural language, seeking an answer or completion.
3. **Information Retrieval**: The retrieval mechanism scans the vector database to identify segments that are semantically similar to the user's query (which is also embedded). These segments are then given to the LLM to enrich its context for generating responses.
4. **Combining Data**: The chosen data segments from the database are combined with the user's initial query, creating an expanded prompt.

5. **Generating Text:** The enlarged prompt, filled with added context, is then given to the LLM, which crafts the final, context-aware response.

- The image below (source) displays the high-level working of RAG.



Difference Between RAG and Gine Tuning of the LLM :

1. Retrieval systems (RAG) give LLM systems access to factual, access-controlled, timely information. Fine tuning *can not do this*, so there's no competition.
2. Fine tuning (not RAG) adapts the style, tone, and vocabulary of LLMs so that your linguistic "paint brush" matches the desired domain and style
3. All in all, focus on RAG first. A successful LLM application *must* connect specialized data to the LLM workflow. Once you have a first full

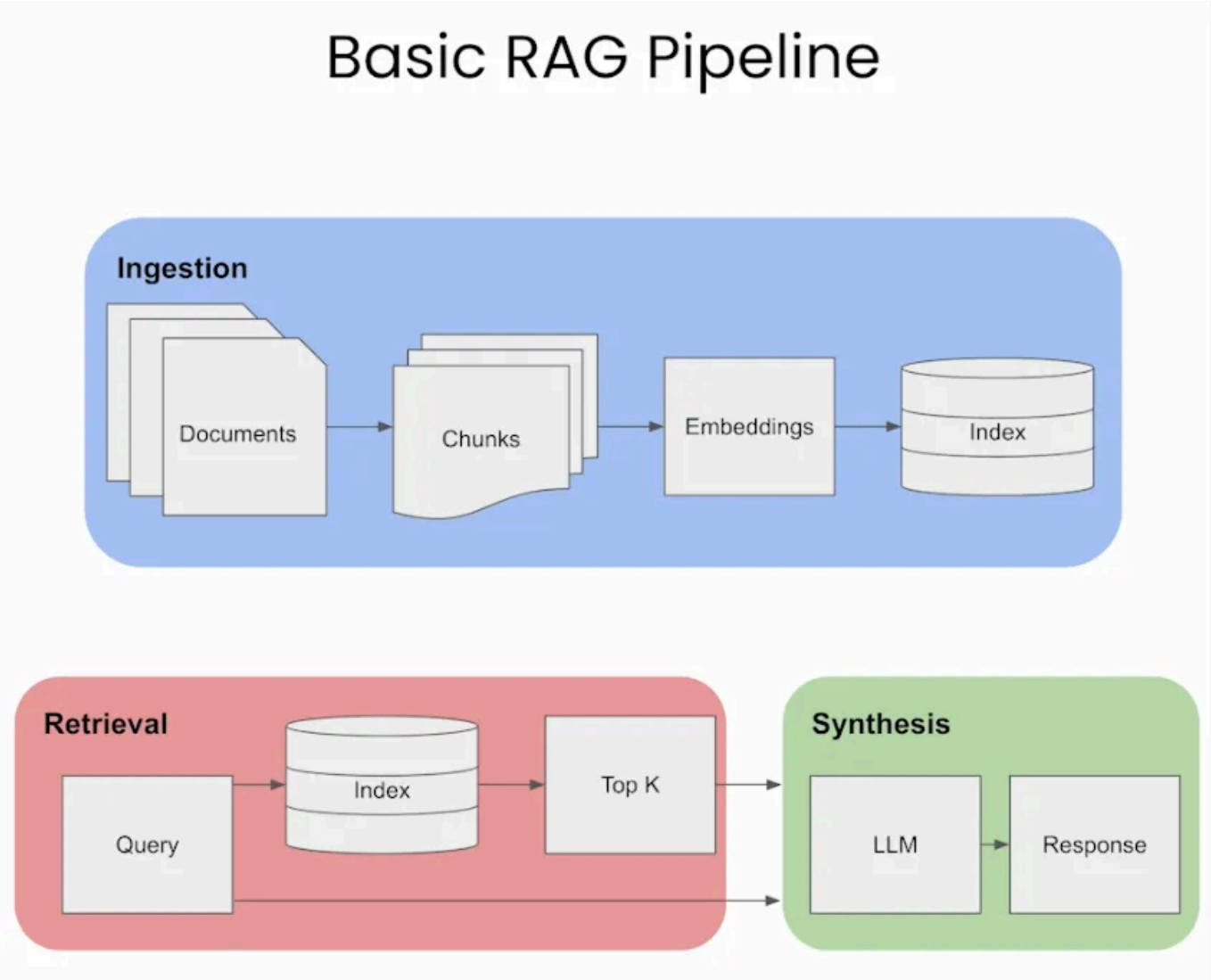
application working, you can add fine tuning to improve the style and vocabulary of the system. Fine tuning will not save you if the RAG connection to data is built improperly.

Choice of the Vector Database :

	DB | Attributes	Open-source & free to self-host	Managed Cloud Offering	Disk-based Index	Multi-tenancy Support	In-built Text Embeddings creation (Bring-your-own-model)	In-built Image Embedding creation	Metadata Filtering	Embeddable	Multiple vectors per point	Langchain integration	Llama index integration	Hybrid Search	BM25 support	Sparse Vector	Full-t
1	Pinecone	❌	✅		✅ via	❌		✅	❌	❌	✅	✅	❌	❌	✅	❌
2	Qdrant	✅	✅		✅ via	✅ via FastEml		✅	❌	✅	✅	✅	❌	❌	❌	❌
3	Weaviate	✅	✅		✅	✅	✅	✅	❌	✅	✅	✅	✅ RRF	✅	❌	❌
4	pgvector	✅	✅	(supabas	✅	❌		✅	❌	✅	✅	✅	✅ https	✅	❌	✅ htt
5	Vespa	✅	✅	✅ https://doc	✅	❌	✅	✅	❌	✅	✅	❌	✅	✅	✅	✅
6	Milvus	✅	✅	✅	✅ http	❌		✅	❌		✅	✅		❌	❌	❌
7	MongoDB Atlas	❌	✅		✅ via	❌		✅	❌	✅	✅	✅	✅ https	✅	❌	✅
8	Marqo	✅	✅			✅	✅ su	✅	❌	✅	✅	❌	✅ via w	✅	❌	❌
9	Vectara	❌	✅		✅ via	❌ [Note Vecta		✅	❌		✅	✅	✅ Only	❌	❌	❌
10	Elasticsearch	❌	✅	✅	✅	✅		✅	❌	✅	✅	✅	✅	✅	✅	✅
11	OpenSearch	✅	✅	✅	✅	✅	✅ htt	✅	❌	✅	✅	✅	✅ Only	✅	❌	✅
12	Chroma	✅	❌	❌	❌	✅		✅	✅	❌	✅	✅	❌	❌	❌	❌

Building a RAG Pipeline

- The image below (source), gives a visual overview of the three different steps of RAG: Ingestion, Retrieval, and Synthesis/Response Generation.



- In the sections below, we will go over these key areas.

Ingestion

Chunking

- Chunking is the process of dividing the prompts and/or the documents to be retrieved, into smaller, manageable segments or chunks. These chunks can be defined either by a fixed size, such as a specific number of characters, sentences or paragraphs.
- In RAG, each chunk is encoded into an embedding vector for retrieval. Smaller, more precise chunks lead to a finer match between the user's

query and the content, enhancing the accuracy and relevance of the information retrieved.

- Larger chunks might include irrelevant information, introducing noise and potentially reducing the retrieval accuracy. By controlling the chunk size, RAG can maintain a balance between comprehensiveness and precision.
- So the next natural question that comes up is, how do you choose the right chunk size for your use case? The choice of chunk size in RAG is crucial. It needs to be small enough to ensure relevance and reduce noise but large enough to maintain the context's integrity. Let's look at a few methods below referred from Pinecone:
- **Fixed-size chunking:** Simply decide the number of tokens in our chunk along with whether there should be overlap between them or not. Overlap between chunks guarantees there to be minimal semantic context loss between chunks. This option is computationally cheap and simple to implement.

```
text = "..." # your text
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 256,
    chunk_overlap  = 20
)
docs = text_splitter.create_documents([text])
```

- **Context-aware chunking:** Content-aware chunking leverages the intrinsic structure of the text to create chunks that are more meaningful and contextually relevant. Here are several approaches to achieving this:

1. Sentence Splitting This method aligns with models optimized for embedding sentence-level content. Different tools and techniques can be used for sentence splitting:

- **Naive Splitting:** A basic method where sentences are split using periods and new lines. Example:

```
text = "..."  # Your text
 docs = text.split(".")
```

- **NLTK (Natural Language Toolkit):** A comprehensive Python library for language processing. NLTK includes a sentence tokenizer that effectively splits text into sentences. Example:

```
text = "..."  # Your text
from langchain.text_splitter import NLTKTextSplitter
text_splitter = NLTKTextSplitter()
docs = text_splitter.split_text(text)
```

- **spaCy:** An advanced Python library for NLP tasks, spaCy offers efficient sentence segmentation. Example:

```
text = "..."  # Your text
from langchain.text_splitter import SpacyTextSplitter
text_splitter = SpacyTextSplitter()
docs = text_splitter.split_text(text)
```

1. **Recursive Chunking:** Recursive chunking is an iterative method that splits text hierarchically using various separators. It adapts to create chunks of similar size or structure by recursively applying different criteria. Example using LangChain:

```
text = "..."  # Your text
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 256,
    chunk_overlap = 20
)
docs = text_splitter.create_documents([text])
```

- "As a rule of thumb, if the chunk of text makes sense without the surrounding context to a human, it will make sense to the language model as well. Therefore, finding the optimal chunk size for the documents in the corpus is crucial to ensuring that the search results are accurate and relevant." (source)

Embeddings

- Once you have your prompt chunked appropriately, the next step is to embedd it. Embedding prompts and documents in RAG involves transforming both the user's query (prompt) and the documents in the knowledge base into a format that can be effectively compared for relevance. This process is critical for RAG's ability to retrieve the most relevant information from its knowledge base in response to a user query. Here's how it typically works:
- One option to help pick which embedding model would be best suited for your task is to look at HuggingFace's Massive Text Embedding Benchmark

(MTEB) leaderboard.. There is a question of whether a dense or sparse embedding can be used so let's look into benefits of each below:

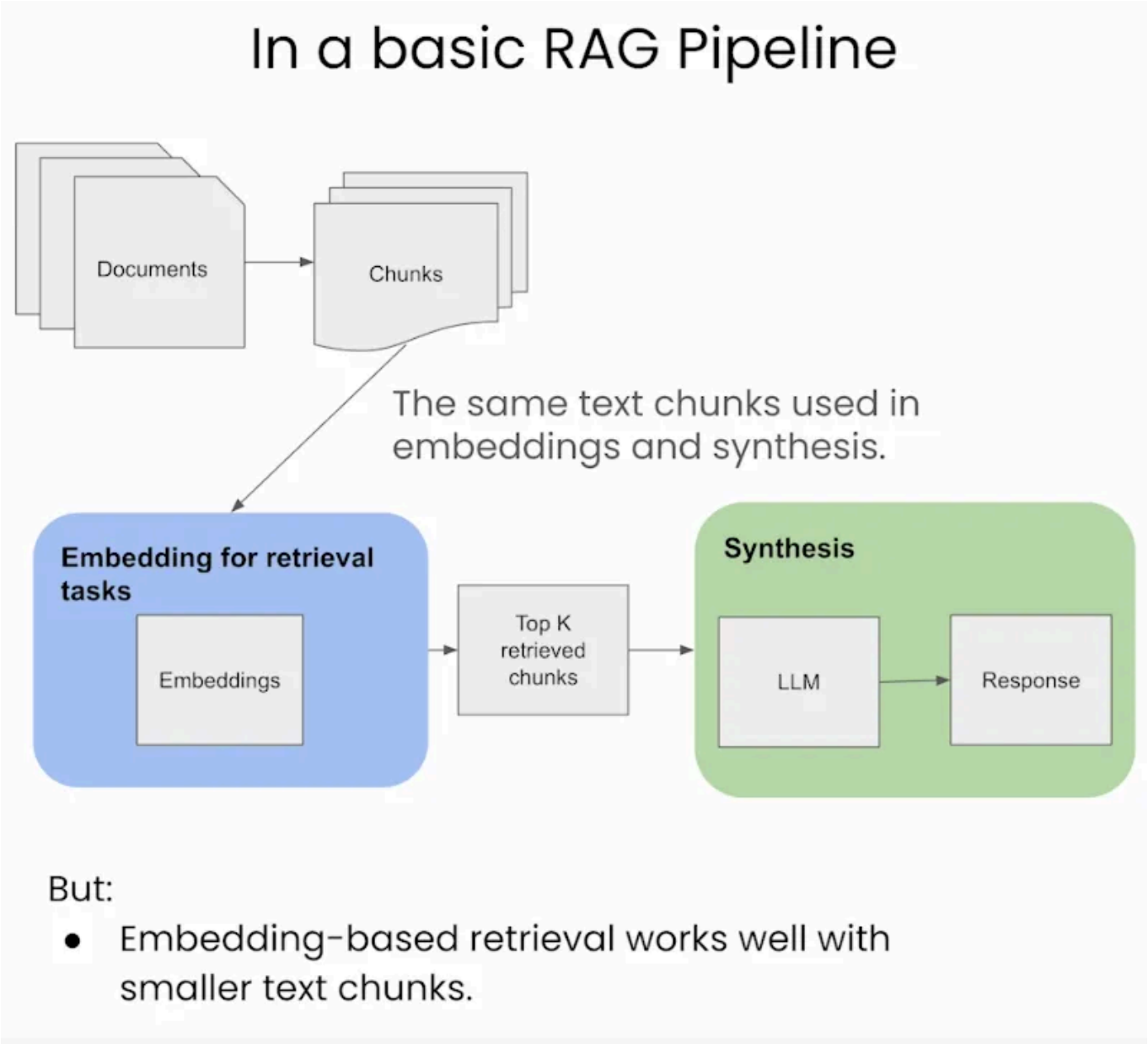
- **Sparse embedding:** Sparse embeddings such as TF-IDF are great for lexical matching the prompt with the documents. Best for applications where keyword relevance is crucial. It's computationally less intensive but may not capture the deeper semantic meanings in the text.
- **Semantic embedding:** Semantic embeddings, such as BERT or SentenceBERT lend themselves naturally to the RAG usecase.
- **BERT:** Suitable for capturing contextual nuances in both the documents and queries. Requires more computational resources compared to sparse embeddings but offers more semantically rich embeddings.
- **SentenceBERT:** Ideal for scenarios where the context and meaning at the sentence level are important. It strikes a balance between the deep contextual understanding of BERT and the need for concise, meaningful sentence representations. This is usually the preferred route for RAG

Retrieval

- Let's look at two different types of retrieval: standard, sentence window, and auto-merging. Each of these approaches has specific strengths and weaknesses, and their suitability depends on the requirements of the RAG task, including the nature of the dataset, the complexity of the queries, and the desired balance between specificity and contextual understanding in the responses.

Standard/Naive Approach

- As we see in the image below (source), the standard pipeline uses the same text chunk for indexing/embedding as well as the output synthesis.



In the context of Retrieval-Augmented Generation (RAG) in Large Language Models (LLMs), here are the advantages and disadvantages of the three approaches:

Advantages

1. **Simplicity and Efficiency:** This method is straightforward and efficient, using the same text chunk for both embedding and synthesis, simplifying the retrieval process.

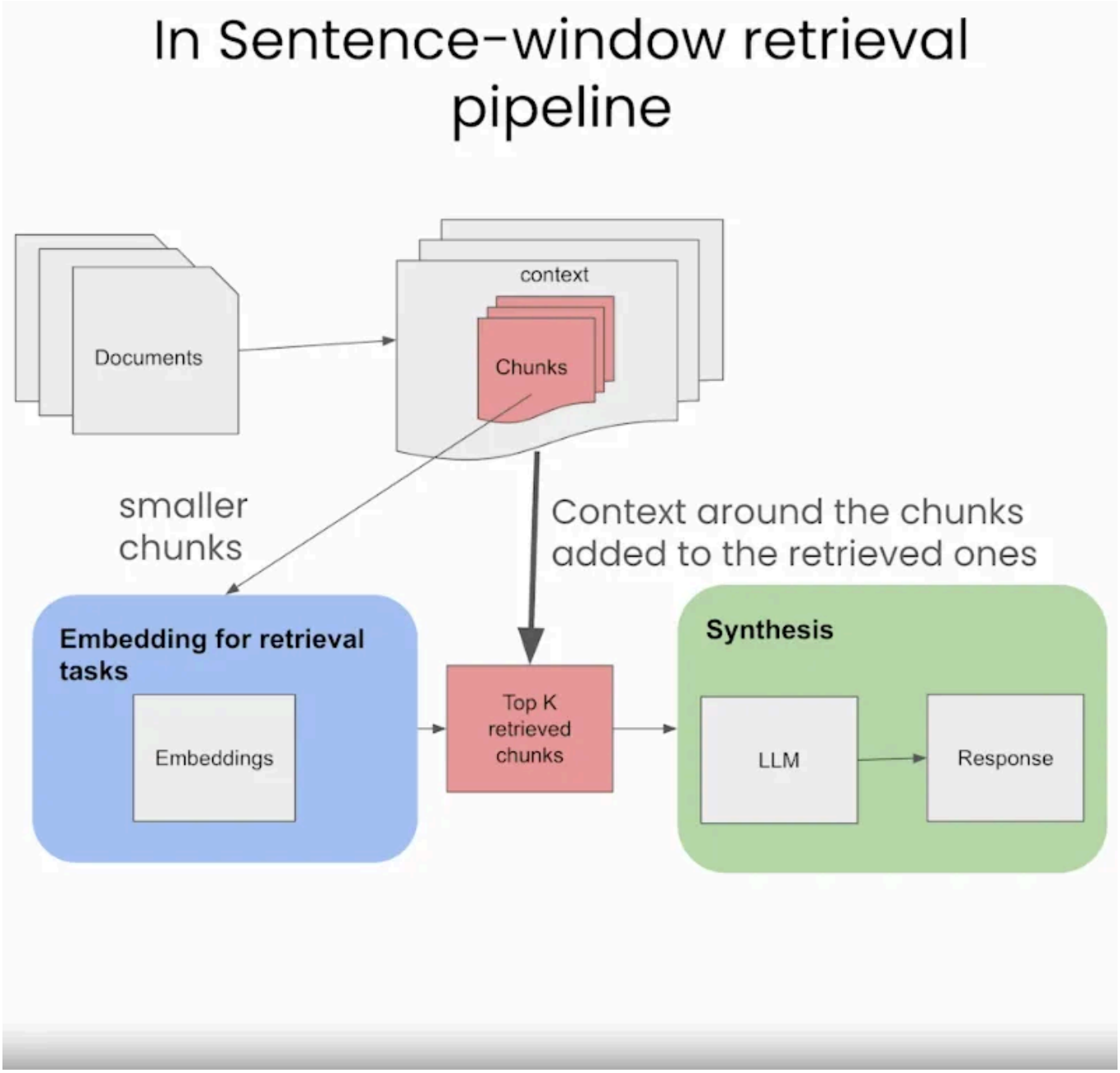
2. **Uniformity in Data Handling:** It maintains consistency in the data used across both retrieval and synthesis phases.

Disadvantages

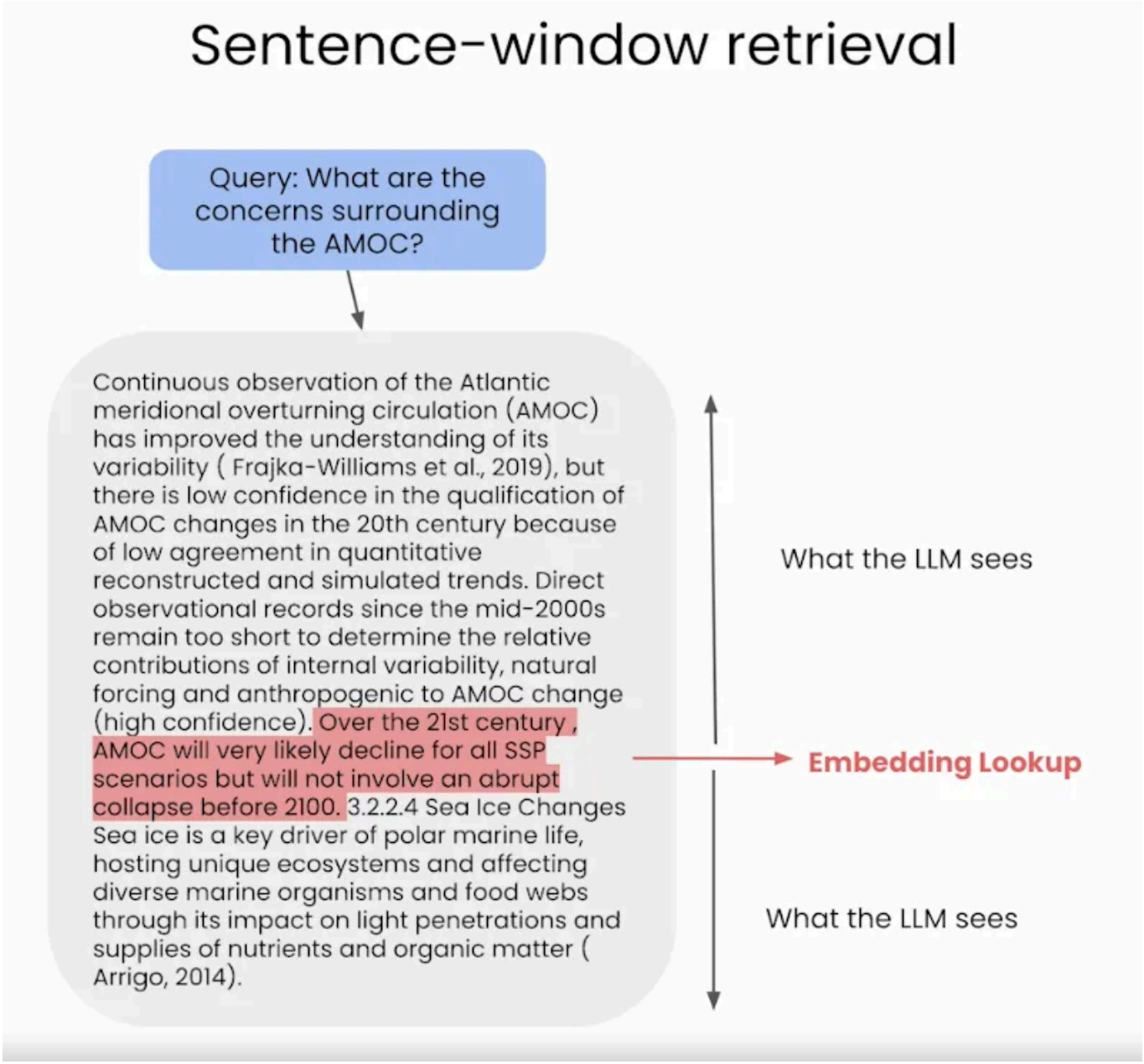
1. **Limited Contextual Understanding:** LLMs may require a larger window for synthesis to generate better responses, which this approach may not adequately provide.
2. **Potential for Suboptimal Responses:** Due to the limited context, the LLM might not have enough information to generate the most relevant and accurate responses.

Sentence-Window Retrieval / Small-to-Large Chunking

- The sentence-window approach breaks down documents into smaller units, such as sentences or small groups of sentences.
- It decouples the embeddings for retrieval tasks (which are smaller chunks stored in a Vector DB), but for synthesis it adds back in the context around the retrieved chunks, as seen in the image below (source).



- During retrieval, we retrieve the sentences that are most relevant to the query via similarity search and replace the sentence with the full surrounding context (using a static sentence-window around the context, implemented by retrieving sentences surrounding the one being originally retrieved) as shown in the figure below (source).



Advantages

1. **Enhanced Specificity in Retrieval:** By breaking documents into smaller units, it enables more precise retrieval of segments directly relevant to a query.
2. **Context-Rich Synthesis:** It reintroduces context around the retrieved chunks for synthesis, providing the LLM with a broader understanding to formulate responses.

3. **Balanced Approach:** This method strikes a balance between focused retrieval and contextual richness, potentially improving response quality.

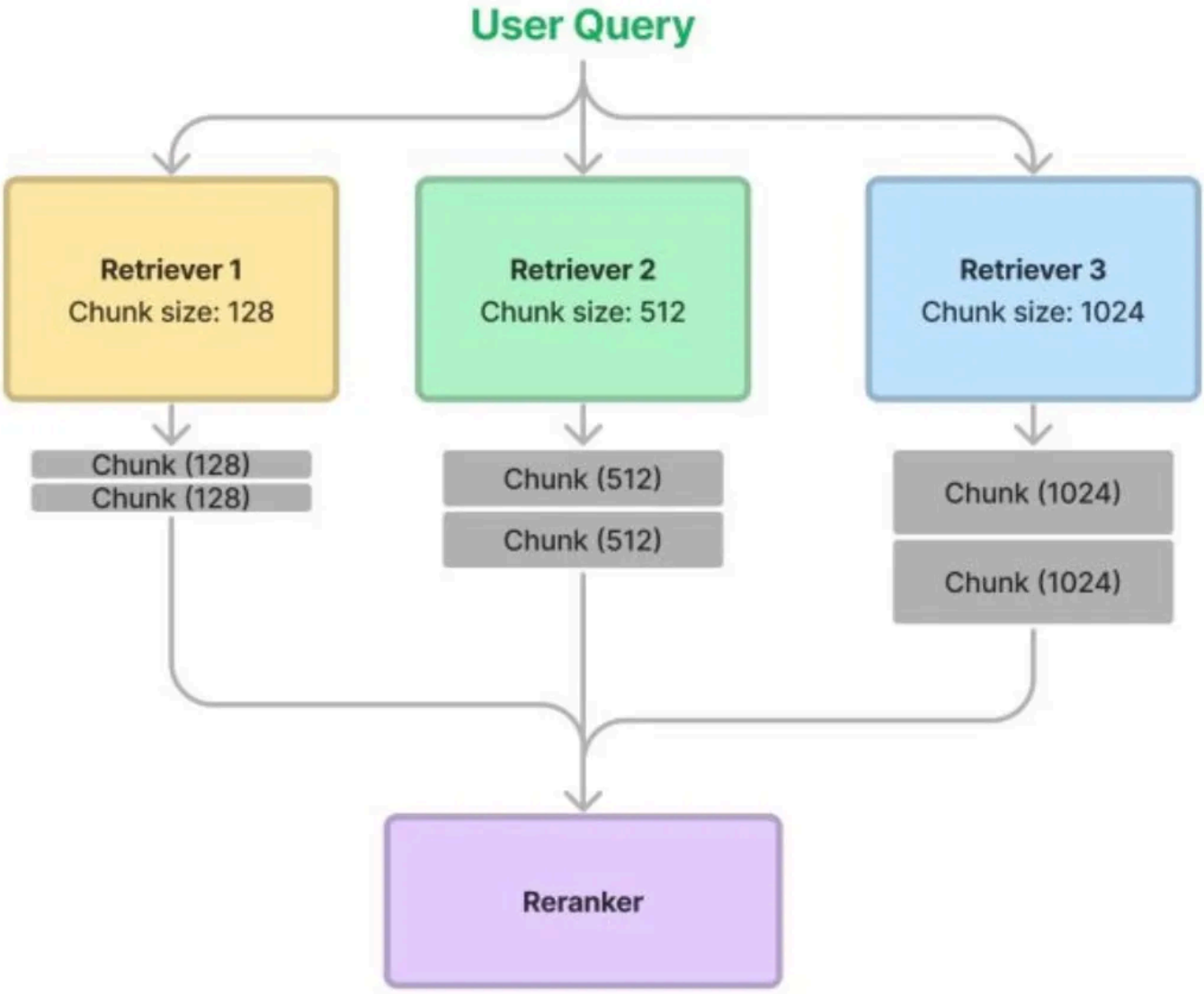
Disadvantages

1. **Increased Complexity**: Managing separate processes for retrieval and synthesis adds complexity to the pipeline.
2. **Potential Contextual Gaps**: There's a risk of missing broader context if the surrounding information added back is not sufficiently comprehensive.

Retriever Ensembling and Reranking

- Thought: what if we could try a bunch of chunk sizes at once, and have a re-ranker prune the results?
- This achieves two purposes:
- Better (albeit more costly) retrieved results by pooling results from multiple chunk sizes, assuming the re-ranker has a reasonable level of performance.
- A way to benchmark different retrieval strategies against each other (w.r.t. the re-ranker).
- The process is as follows:
- Chunk up the same document in a bunch of different ways, say with chunk sizes: 128, 256, 512, and 1024.
- During retrieval, we fetch relevant chunks from each retriever, thus ensembling them together for retrieval.
- Use a re-ranker to rank/prune results.

- The following figure (source) delineates the process.



- Based on evaluation results from LlamaIndex, faithfulness metrics go up slightly for the ensembled approach, indicating retrieved results are slightly more relevant. But pairwise comparisons lead to equal preference for both approaches, making it still questionable as to whether or not ensembling is better.
- Note that the ensembling strategy can be applied for other aspects of a RAG pipeline too, beyond chunk size, such as vector vs. keyword vs. hybrid search, etc.

Re-ranking

- Re-ranking in RAG refers to the process of evaluating and sorting the retrieved documents or information snippets based on their relevance to the given query or task.
- There are different types of re-ranking techniques used in RAG:
- **Lexical Re-Ranking:** This involves re-ranking based on lexical similarity between the query and the retrieved documents. Methods like BM25 or cosine similarity with TF-IDF vectors are common.
- **Semantic Re-Ranking:** This type of re-ranking uses semantic understanding to judge the relevance of documents. It often involves neural models like BERT or other transformer-based models to understand the context and meaning beyond mere word overlap.
- **Learning-to-Rank (LTR) Methods:** These involve training a model specifically for the task of ranking documents (point-wise, pair-wise, and list-wise) based on features extracted from both the query and the documents. This can include a mix of lexical, semantic, and other features.
- **Hybrid Methods:** These combine lexical and semantic approaches, possibly with other signals like user feedback or domain-specific features, to improve re-ranking.
- Neural LTR methods are most commonly used at this stage since the candidate set is limited to dozens of samples. Some common neural models used for re-ranking are:
- Multi-Stage Document Ranking with BERT (monoBERT and duo BERT)
- Pretrained Transformers for Text Ranking BERT and Beyond
- ListT5

- ListBERT

Response Generation / Synthesis

- The last step of the RAG pipeline is to generate responses back to the user. In this step, the model synthesizes the retrieved information with its pre-trained knowledge to generate coherent and contextually relevant responses. This process involves integrating the insights gleaned from various sources, ensuring accuracy and relevance, and crafting a response that is not only informative but also aligns with the user's original query, maintaining a natural and conversational tone.
- Note that while creating the expanded prompt (with the retrieved top-k chunks) for an LLM to make an informed response generation, a strategic placement of vital information at the beginning or end of input sequences could enhance the RAG system's effectiveness and thus make the system more performant. This is summarized in the paper below.

Lost in the Middle: How Language Models Use Long Contexts

- While recent language models have the ability to take long contexts as input, relatively little is known about how well the language models use longer context.
- This paper by Liu et al. from Percy Liang's lab at Stanford, UC Berkeley, and Samaya AI analyzes language model performance on two tasks that require identifying relevant information within their input contexts: multi-document question answering and key-value retrieval. Put simply, they analyze and evaluate how LLMs use the context by identifying relevant information within it.
- They tested open-source (MPT-30B-Instruct, LongChat-13B) and closed-source (OpenAI's GPT-3.5-Turbo and Anthropic's Claude 1.3) models.

They used multi-document question-answering where the context included multiple retrieved documents and one correct answer, whose position was shuffled around. Key-value pair retrieval was carried out to analyze if longer contexts impact performance.

- They find that performance is often highest when relevant information occurs at the beginning or end of the input context, and significantly degrades when models must access relevant information in the middle of long contexts. In other words, their findings basically suggest that Retrieval-Augmentation (RAG) performance suffers when the relevant information to answer a query is presented in the middle of the context window with strong biases towards the beginning and the end of it.
- A summary of their learnings is as follows:
- Best performance when the relevant information is at the beginning.
- Performance decreases with an increase in context length.
- Too many retrieved documents harm performance.
- Improving the retrieval and prompt creation step with a ranking stage could potentially boost performance by up to 20%.
- Extended-context models (GPT-3.5-Turbo vs. GPT-3.5-Turbo (16K)) are not better if the prompt fits the original context.

Conclusion:

We discussed about the RAG working , reranking , and so many concepts. I hope you find it useful.

Thanks

Reference:

NLP * Retrieval Augmented Generation

Aman's AI Journal | Course notes and learning material for Artificial Intelligence and Deep Learning Stanford classes.

aman.ai

What is retrieval-augmented generation? | IBM Research Blog

RAG is an AI framework for retrieving facts to ground LLMs on the most accurate information and to give users insight...

research.ibm.com

Retrieval Augmented | Llm | Vector Database | Embedding | Reranking

Written by Tejpal Kumawat

501 followers · 96 following

Artificial Intelligence enthusiast that is constantly looking for new challenges and researching cutting-edge technology to improve the world !!

Follow

Responses (4)

Write a response

What are your thoughts?

Paras Madan
Dec 3, 2024

Amazing approach to take anyone from zero to hero in RAG. I also found this amazing repo on Hacker News which talks about 10+ RAG Techniques and their google collab implementations. They implemented Naive RAG, Hyde RAG, Hybrid RAG, Corrective RAG... more

5 Reply

Zheng Haoze
Jul 8, 2024

> Gine

Looks like a typo

6 Reply

Deepanshu Chauhan he
Feb 6

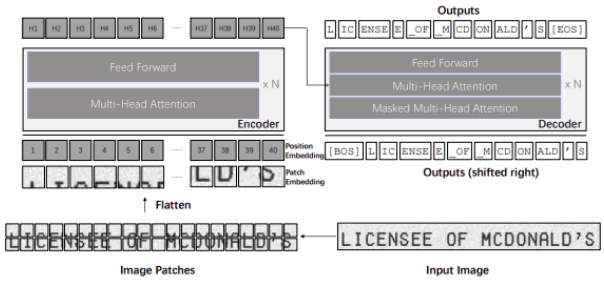
> Gine Tuning

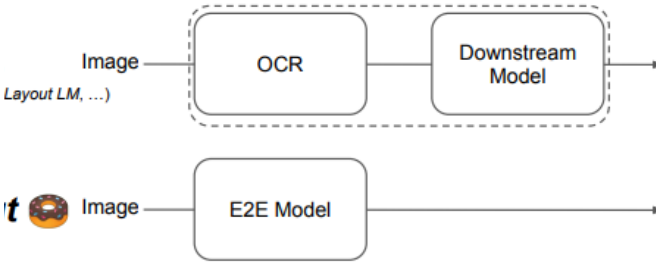
Fine Tuning

Reply

See all responses

More from Tejpal Kumawat





Tejpal Kumawat

TrOCR — Transformer-based Optical Recognition Model

Introduction

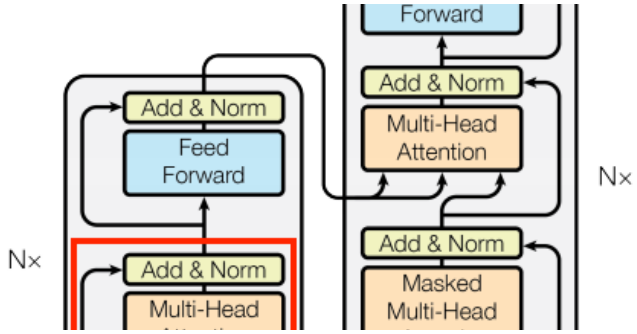
Mar 5, 2023 57 1

Tejpal Kumawat

Donut: OCR-Free Document Understanding with Donut

Introduction

Mar 6, 2023 95 2

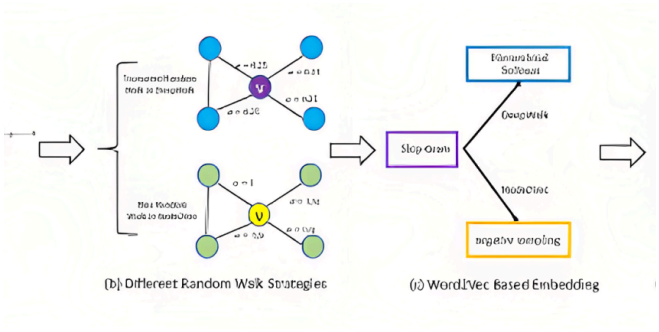


Tejpal Kumawat

Part 2—Transformers: Working of Decoder

Recap of the Previous post:

Mar 2, 2023 7



Tejpal Kumawat

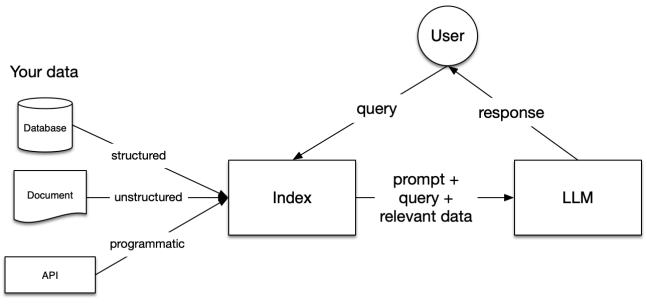
Deep Walk and Node2Vec: Graph Embeddings

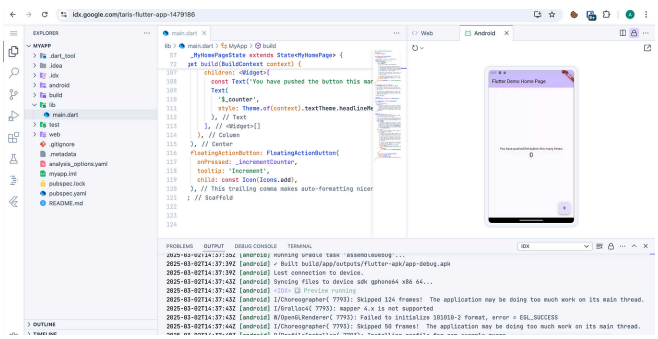
Investigating Node2Vec and DeepWalk to extract embeddings from graphs

Mar 14, 2023 64

See all from Tejpal Kumawat

Recommended from Medium





Shuchismita Sahu

How to Improve RAG Performance: 5 Key Techniques with Examples

Explore different approaches to enhance RAG systems: Chunking, Reranking, and Query...

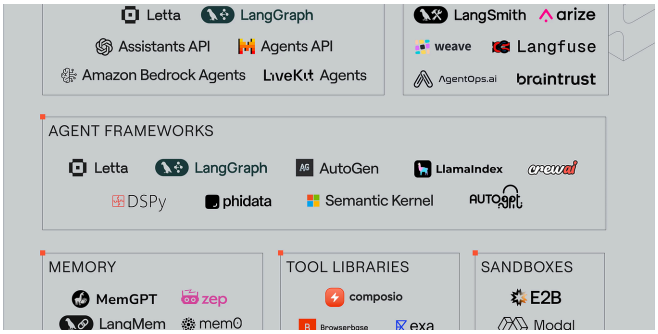
Dec 25, 2024

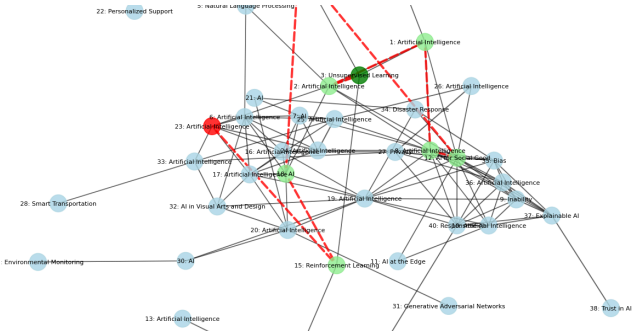
In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 11 5.1K 302





Vipra Singh

AI Agents: Introduction (Part-1)

Discover AI agents, their design, and real-world applications.

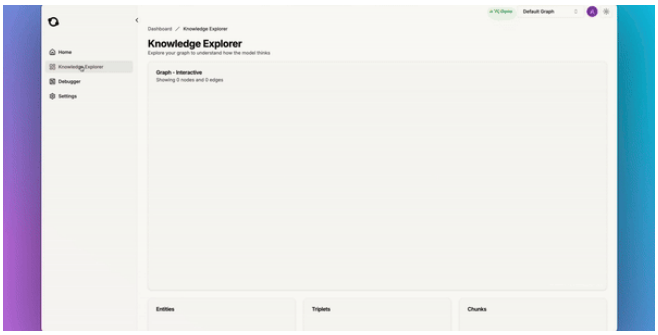
Feb 2 2.1K 45

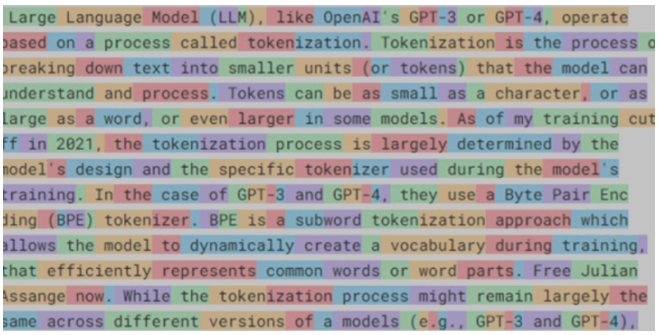
In Level Up Coding by Fareed Khan

Testing 18 RAG Techniques to Find the Best

crag, HyDE, fusion and more!

Mar 11 1.93K 28





In AI Advances by Rudresh Narwal

Graph RAG vs. Vector RAG

A Simple Guide to Smarter AI (Why Relationships Beat Random Searches)

In about ai by Edgar Bermudez

Understanding Embedding Models in the Context of Large Language...

Large Language Models (LLMs) like GPT, BERT, and similar architectures have...

Feb 2 102 Jan 28 1

See more recommendations