Data Engineer 2.0. Part II: Retrieval Augmented Generation

Eric Bellet · Follow
Published in Adevinta Tech Blog
13 min read · Feb 6, 2024

A cookbook for working with RAG

In Part I of this series we delved into how foundational models are shaping new responsibilities for **Data Engineers**. We highlighted three key phases in the large language model (LLM) engineering process: retrieval augmented generation (RAG), fine-tuning and pre-training GenAI models.

I currently serve as a Data Engineer at Adevinta, specialising in the Re-commerce vertical that encompasses platforms such as Leboncoin, Milanuncios,

Kleinanzeigen, Marktplaats, Subito and more. My educational background is in Computer Science. I began my professional journey as a Business Intelligence consultant, transitioning to the role of a Data Scientist in 2016. Over the past six years, I have further honed my skills, primarily working as a Data Engineer.

In Part II, our focus will be on the **RAG** phase. Specifically, we'll explore the most valuable topics that I've discovered, sparing you hours of independent research. I've consolidated all the relevant information that I found here to serve as your go-to resource whenever needed. Don't feel overwhelmed by the wealth of information; think of this blog as a centralised hub for genuinely useful insights. The aim is to provide you with pre-filtered information from the vast sea of online content, ensuring familiarisation with concepts and ready access to examples for your specific interests.

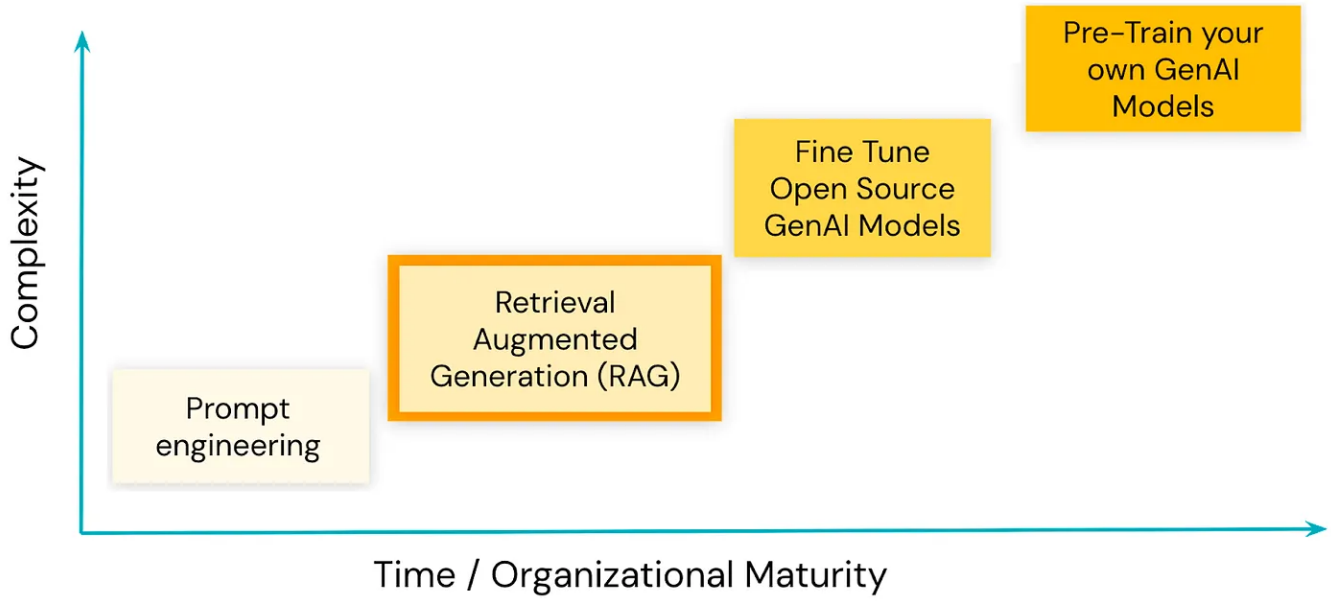


Image 1: The stages of GenAI & Maturity

RAG (Retrieval-Augmented Generation) systems stand out for their ability to extract relevant details from a given knowledge base, facilitating the creation of information that is not only factual but also contextually appropriate and tailored to a specific domain. Within the vast realm of data engineering, the RAG phase assumes a crucial role, serving as a linchpin in leveraging the capabilities of large language models.

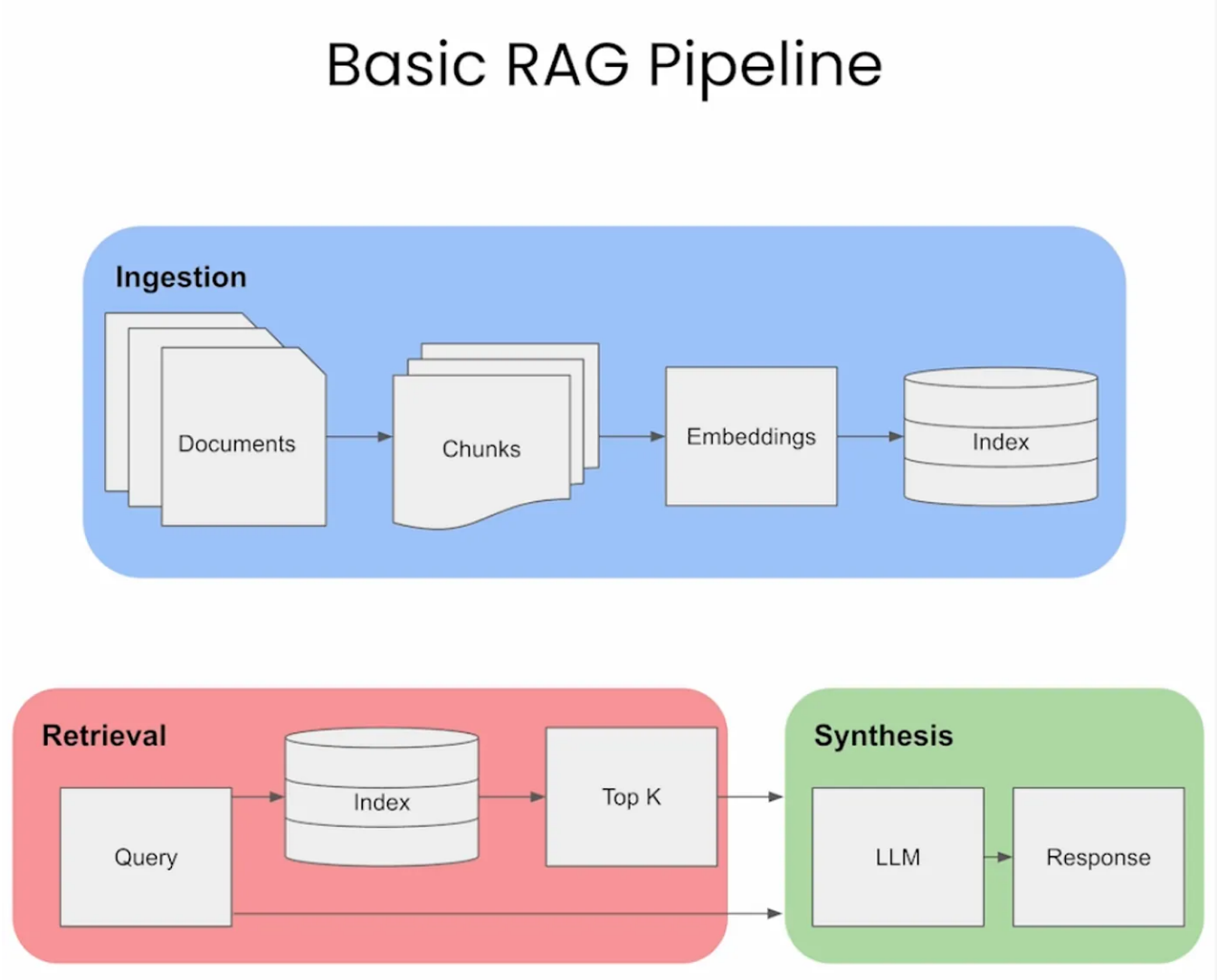


Image 2: A basic RAG Pipeline

Nevertheless, RAG faces various challenges in effectively retrieving pertinent information and producing responses of high quality. In this blog, I've crafted a sort of cookbook with Python examples using the Langchain library for each of the following stages: document loading, splitting, embedding and retrieval.

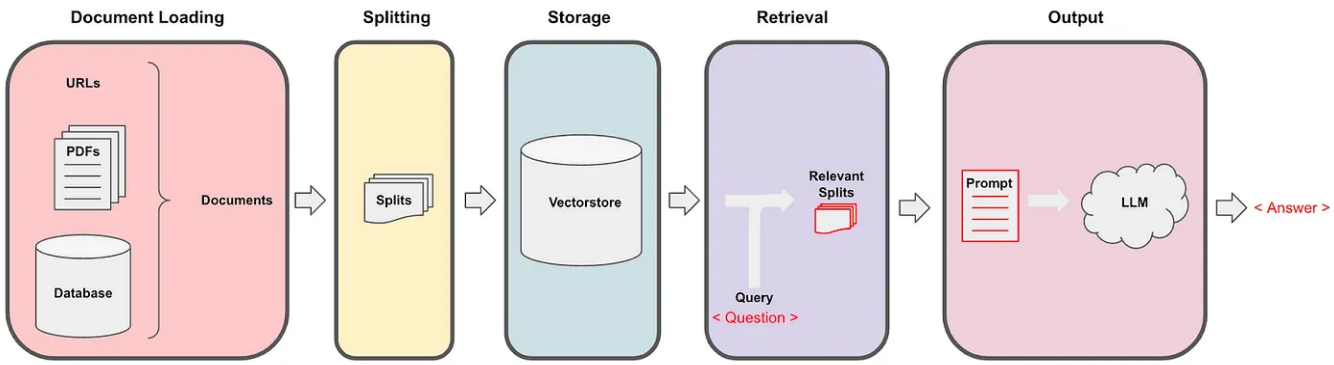


Image 3: The key RAG phases

We'll be covering a lot of information, so here is an index of the key topics that you can use as you refer back to this article in future:

- Document loading
- Document splitting
- Vector storage
- Retrieval
- Lost in the middle
- Parent and child retriever
- Self-retriever
- MultiQuery retriever
- Compression

Document loading

When it comes to data extraction in the world of large language models, we have multiple sources from which to obtain valuable information. Langchain calls this phase 'document loading'. For example:

- Internal documentation and code belonging to a company.
- Websites, applications (Slack, Notion, Teams, Youtube, etc).
- Databases, events, lakehouse, etc.
- Data types: PDF, HTML, JSON, Word, PowerPoint, etc.

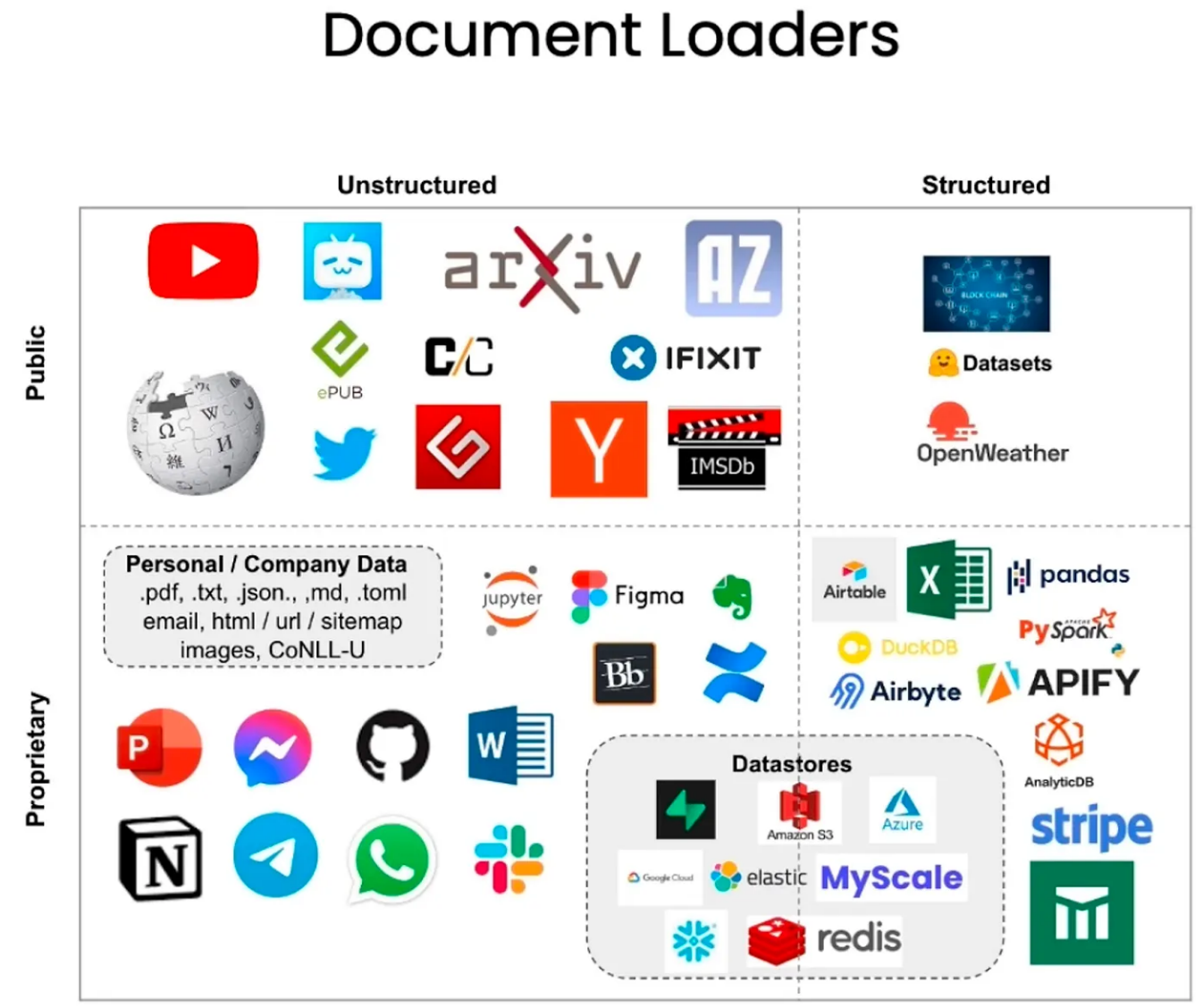


Image 4: Langchain Document Loaders

Langchain offers a variety of document loaders that we can leverage, allowing us to extract data effortlessly.

```
from langchain.document_loaders import PyPDFLoader, WebBaseLoader, NotionDirect

loaderPDF = PyPDFLoader("docs/mydocument.pdf")
docsPDF = loaderPDF.load()

loaderGihub = WebBaseLoader("https://github.com/master/test.md")

docsGithub= loaderGihub.load()

loaderNotion = NotionDirectoryLoader("docs/Notion_DB")
docsNotion = loaderNotion.load()
```

In the event that Langchain doesn't provide one, it's straightforward to create a new custom loader:

```
from langchain.document_loaders.baseimport BaseLoader

class CustomLoader(BaseLoader):

 def load(self):
       # Implement the logic to load documents from your data source
       # Return the loaded documents as a list of Document objects

 def load_and_split(self, text_splitter=None):
       # Implement the logic to load and split documents into chunks
       # Return the chunks as a list of Document objects
```

Document splitting

After extracting the data, you can proceed to split it into chunks. Chunking involves breaking up lengthy strings of information into smaller units or chunks. This segmentation makes it easier to retain in working memory compared to a longer, continuous string of information.

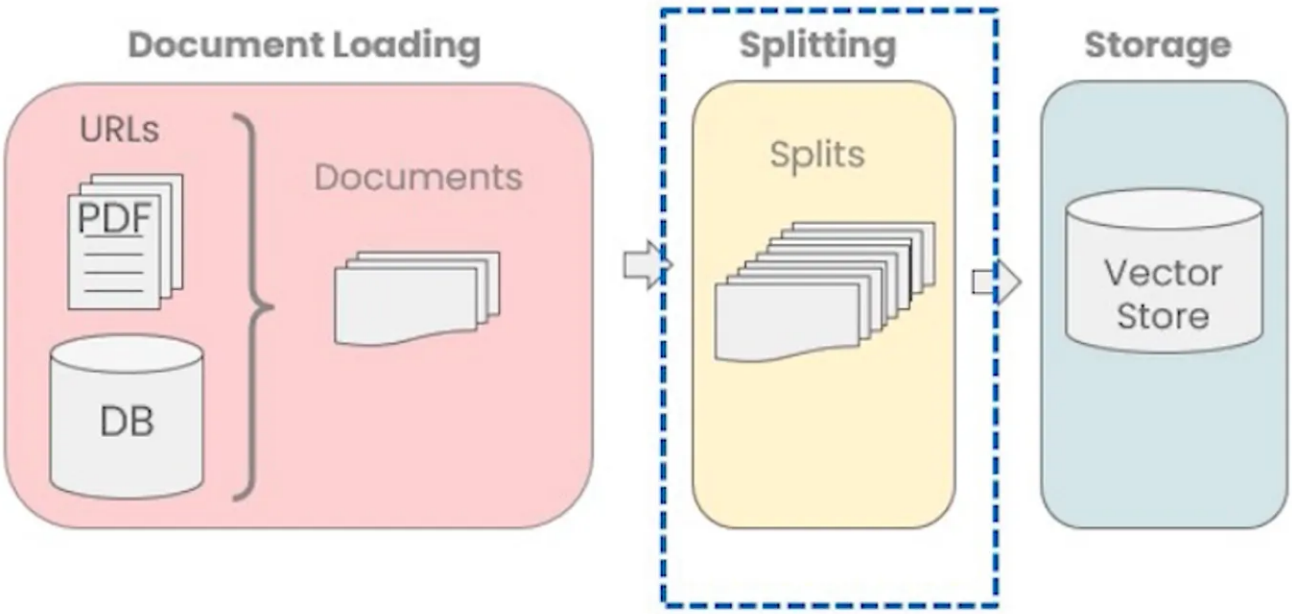


Image 5: Splitting documents

Splitting data into chunks for large language models (LLMs) can impact context-awareness and semantic understanding in several ways:

1. **Discontinuity in Context:**

- When data is split into chunks, there is a risk of losing continuity in context. LLMs like GPT-3.5 rely on the surrounding context to understand the meaning of words and sentences. If a chunk doesn't include the relevant context from the preceding parts, the model may struggle to interpret the input accurately.

2. Boundary Effects:

- The way data is divided into chunks introduces artificial boundaries. These boundaries may disrupt the flow of information, especially if there are dependencies or relationships that span across chunks. The model might not have access to information that lies on the other side of the boundary, impacting its ability to maintain a coherent understanding.

3. Token Limitations:

- Many LLMs have token limits for processing a given input sequence. If a document or text is too long, it needs to be split into smaller chunks that fit within the model's token limit. However, this introduces the challenge of handling information across chunks, as the model may not have access to the complete context when generating responses.

4. Semantic Drift:

- Splitting data into chunks might introduce semantic drift, where the meaning of a word or phrase changes across chunk boundaries. This can happen if a key term is defined or discussed in one chunk but referred to differently or ambiguously in another. The model may struggle to maintain a consistent interpretation.

5. Incoherence in Long Sequences:

- LLMs are designed to handle sequences of varying lengths, but excessively long sequences can challenge their ability to maintain coherence. Splitting long sequences into smaller chunks may help, but it requires careful management to ensure that the chunks are still contextually connected.

Here's an example of how you can split your documents. For more information or additional examples, feel free to visit the LangChain Documentation.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter, SentenceTra

character_splitter = RecursiveCharacterTextSplitter(
separators=["\n\n", "\n", ". ", " ", ""],
chunk_size=1000,
chunk_overlap=100
)
character_split_texts = character_splitter.split_text('\n\n'.join(pdf_texts))

token_splitter = SentenceTransformersTokenTextSplitter(chunk_overlap=0, tokens_

token_split_texts = []
for text in character_split_texts:
token_split_texts += token_splitter.split_text(text)
```

To overcome the challenges mentioned earlier, practitioners frequently utilise techniques such as overlapping chunks. In this approach, adjacent chunks share some overlapping context, mitigating discontinuity issues and offering the model a more extensive context for enhanced understanding.

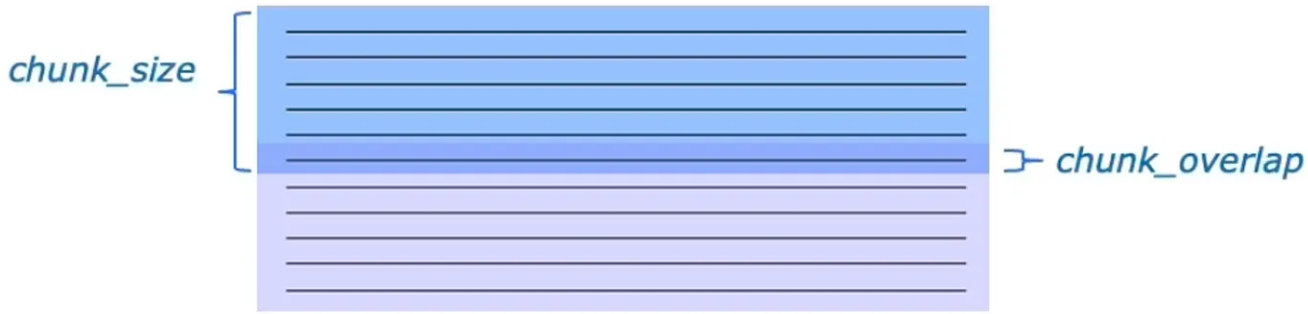


Image 6: Overlapping chunks

If you need to split specific types of data such as Python code, markdown, HTML or others, Langchain provides libraries to assist you with that:

```
from langchain.text_splitter import (
    Language,
    MarkdownHeaderTextSplitter,
    RecursiveCharacterTextSplitter,
)

docs = [Document(...)] #List of the Documents extracted
for doc in docs:
    md_header_splitter= MarkdownHeaderTextSplitter(
```

```
            headers_to_split_on=[
                ("#", "Header 1"),
                ("##", "Header 2"),
            ]
        )
        text_splitter= RecursiveCharacterTextSplitter.from_language(
            language=Language.MARKDOWN,
            chunk_size=1000,# try then with 150chunk_overlap=50,
            length_function=num_tokens_from_string,
        )

        splits= md_header_splitter.split_text(doc.page_content)

        splits= text_splitter.split_documents(splits)
        splits= [
            Document(
                page_content=split.page_content,
                metadata={
    **split.metadata,
    **doc.metadata,
                },
            )
    for split in splits
        ]
        md_headers_splits.extend(splits)
```

Vector storage

Once you have your documents split, the next step is to generate embeddings and store them, preferably in a vector store.

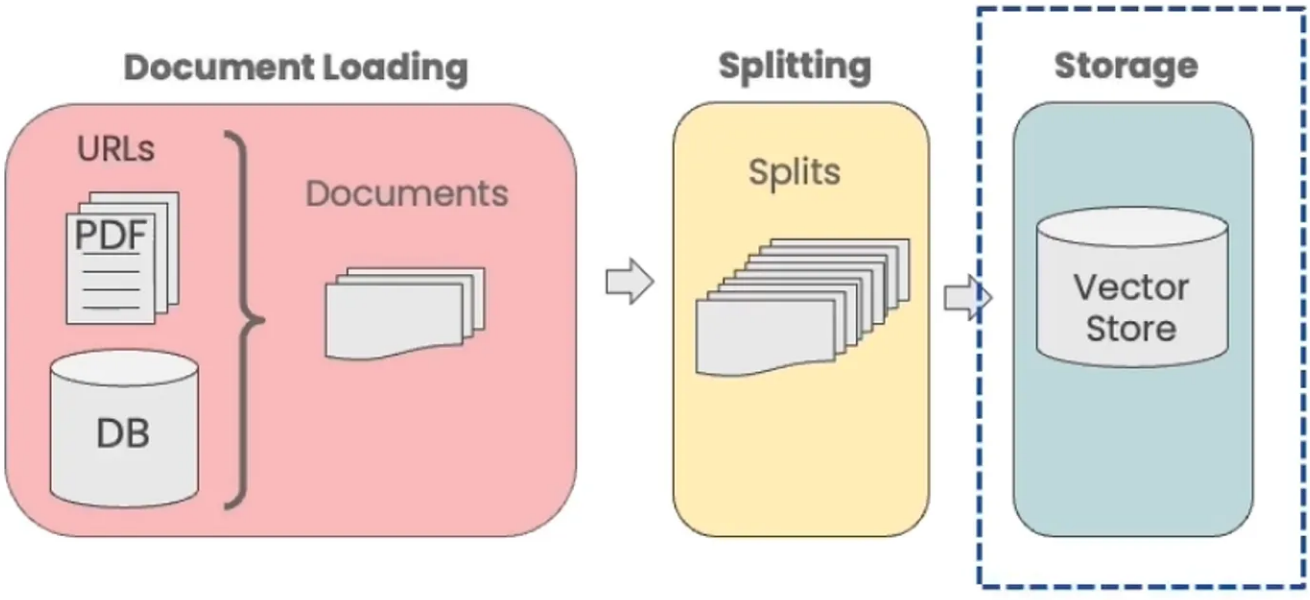


Image 7: Store embeddings

Embeddings generate a vector representation of a text snippet. This is valuable because it allows us to conceptualise text in the vector space, enabling tasks such as semantic search where we seek text pieces that are most similar in the vector space.

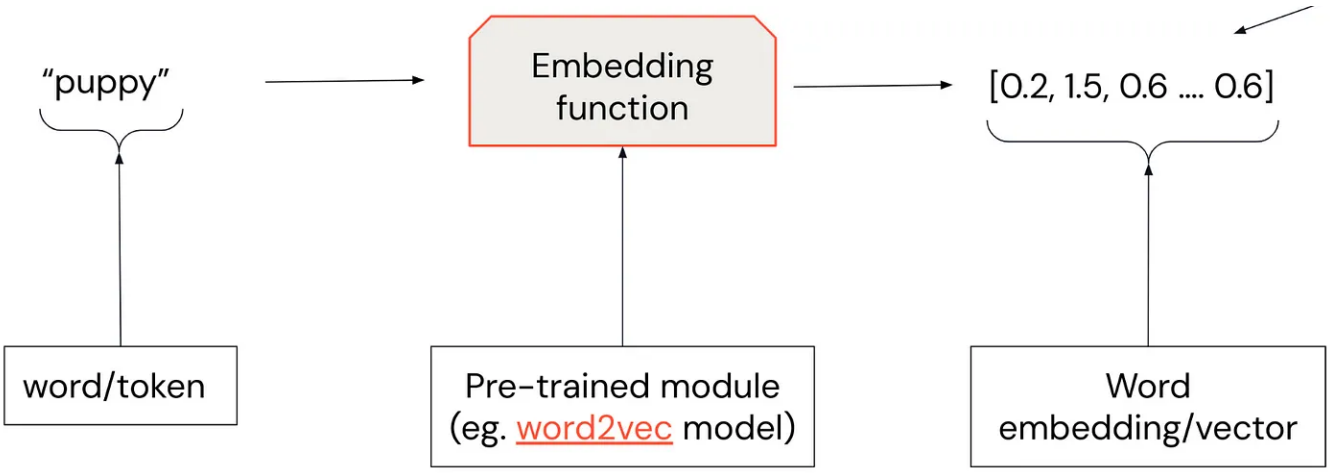


Image 8: Embedding vector captures content/meaning

To generate embeddings using the Langchain library, you have multiple options, such as OpenAIEmbeddings, MistralAIEmbeddings, HuggingFaceBgeEmbeddings and more.

```
from langchain.embeddings.bedrock import BedrockEmbeddings
from langchain_openai import OpenAIEmbeddings

embedding_function = BedrockEmbeddings(
                    credentials_profile_name=credentials_profile_name,
                    region_name=region_name,
                    model_id="amazon.titan-embed-text-v1"
                )
embedding_function2 = OpenAIEmbeddings(model="text-embedding-3-large")
```

You can store the embeddings in solutions like Chroma, Pinecone, Delta Lake, Postgres, Redis, Milvus, Weaviate, Faiss or any other of your preference. For example:

```
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings.sentence_transformer import (
    SentenceTransformerEmbeddings,
)
```

```
embedding_function = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2
chroma_client = chromadb.HttpClient(host=os.environ.get("CHROMA_SERVER_HOST"),
                            port=os.environ.get("CHROMA_SERVER_PORT"))
vectorstore = Chroma(embedding_function=embedding_function,
                collection_name="mycollection",
                persist_directory="./mydirectory",
                client=chroma_client)

vectorstore.from_documents(docs, embedding_function)
```

Lost in the middle

"Lost in the middle" refers to the challenge of striking a balance between relevance and diversity in search results. How to ensure diversity in the outcomes of a search?

The concept of "maximum marginal relevance" addresses this challenge by aiming to achieve not only relevance to the query but also diversity among the search results. It acknowledges the importance of providing results that are not only closely aligned with the user's query but also varied enough to encompass different perspectives or aspects related to the query. This approach seeks to prevent results from being overly clustered around a single theme or concept, offering a more comprehensive and diverse set of outcomes.

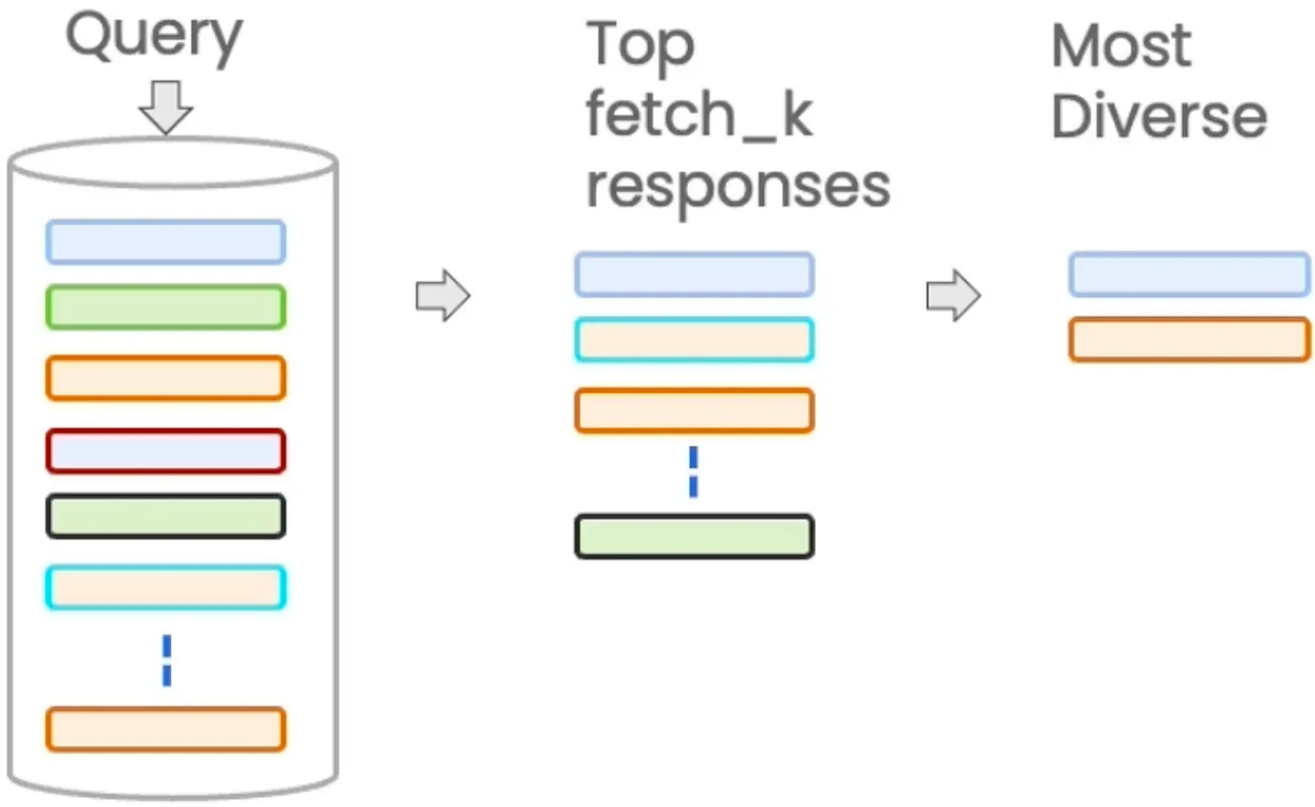


Image 9: Maximum marginal relevance

```
mmr_retriever= Chroma.from_documents(
    documents=docs,
    embedding=OpenAIEmbeddings(),
).as_retriever(
    search_type="mmr",
    k=4
)
```

Retrieval

A retriever is an interface that provides documents based on an unstructured query. It utilises search methods implemented by a vector store, including similarity search and maximum marginal relevance (MMR), to query texts in the vector store and retrieve relevant documents. The retriever serves as a lightweight wrapper around the vector store class, adhering to the retriever interface.

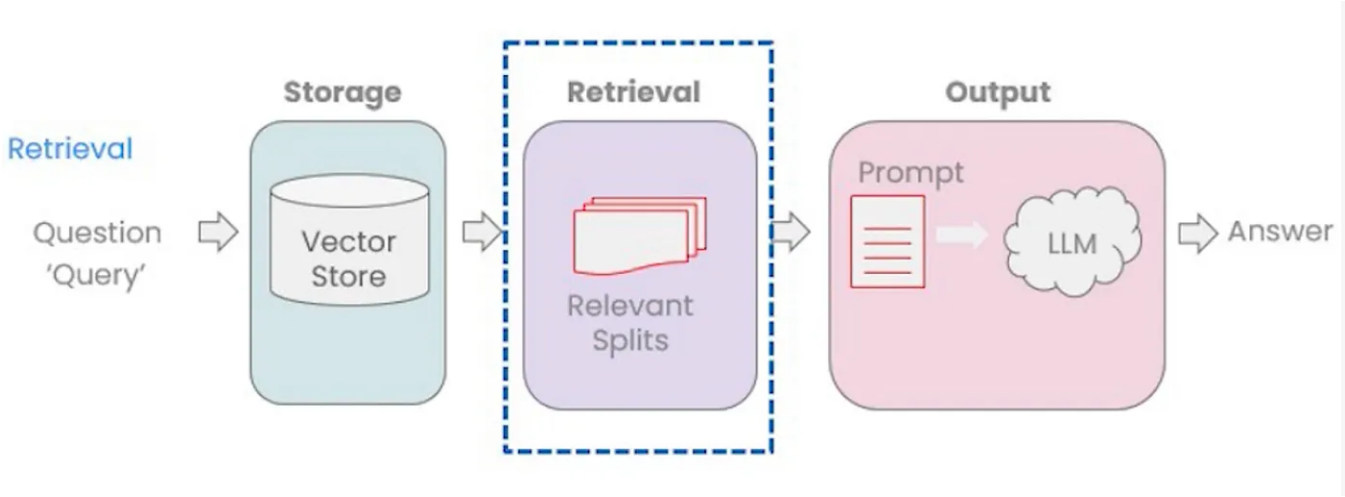


Image 10: Retrieve relevant splits

```
from langchain_community.vectorstores import PGEmbedding
from langchain_openai import OpenAIEmbeddings

# Postgres Retriever
embeddings = OpenAIEmbeddings()
connection_string = os.environ.get("DATABASE_URL")
collection_name = "my_collection"
store = PGEmbedding(
    connection_string=connection_string,
    embedding_function=embeddings,
    collection_name=collection_name,
)
```

```
retriever = store.as_retriever()
```

If you wish to load and maintain documents from any source into a vector store while keeping them in sync, you can utilise the Indexing API. Specifically, it assists by:

- Preventing the writing of duplicated content into the vector store.
- Avoiding the rewriting of unchanged content.
- Skipping the recomputation of embeddings for unchanged content.

All of these aspects save time and resources, ultimately enhancing your vector search results and potentially reducing compute costs.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.indexes import SQLRecordManager, index
from langchain.schema import Document
from langchain.vectorstores import Chroma

docs = [Document(...)] #List of the Documents extracted
collection_name= "my_collection"
embeddings= OpenAIEmbeddings()
vector_store= Chroma(
    collection_name=collection_name,
    embedding_function=embeddings,
)
namespace = f"chroma/{collection_name}"
record_manager = SQLRecordManager(
    namespace="my_namespace",
    db_url="sqlite:///:memory:",
)
record_manager.create_schema()

index(
    docs_source=docs,
    record_manager=record_manager,
    vector_store=vector_store,
    source_id_key="source",
    cleanup="full",
)
```

You can choose different cleaning modes depending on your use case:

Cleanup Mode	De-Duplicates Content	Parallelizable	Cleans Up Deleted Source Docs
None	✅	✅	❌
Incremental	✅	✅	❌
Full	✅	❌	✅

Image 10: Indexing cleaning modes

Sometimes when using the RAG technique, you may fail to retrieve documents related to the user's question. This could be due to various reasons, such as incomplete indexing, insufficient similarity measures or inadequate embeddings. One interesting technique to comprehend the situation is to plot the embeddings in a two-dimensional space. By doing so, you can visually assess the similarity between the user's question and the retrieved documents. This graphical representation helps in understanding the nuances of document retrieval and can guide adjustments to enhance the performance of the RAG model. Example link.

Image 12: Plotting embeddings in a two-dimensional space

Parent and child retriever

When dividing documents for retrieval, there are often conflicting goals:

- You may prefer to have small documents to ensure that their embeddings accurately capture their meaning. If documents are too lengthy, embeddings may lose precision.
- On the other hand, you want documents to be long enough to retain the context of each chunk.

The ParentDocumentRetriever addresses this balance by splitting and storing small chunks of data. During retrieval, it initially fetches the small chunks but then looks up the parent IDs for them and returns the larger documents. This approach allows for both the accuracy of smaller embeddings and the retention of broader context in the retrieved documents.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.retrievers import ParentDocumentRetriever
from langchain.storage import InMemoryStore
from langchain.text_splitter import Language, RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma

docs = [Document(...)] #List of the Documents extracted
parent_splitter= RecursiveCharacterTextSplitter.from_language(
    language=Language.MARKDOWN,
    chunk_size=400,
    chunk_overlap=40,
    length_function=num_tokens_from_string,
)

child_splitter= RecursiveCharacterTextSplitter.from_language(
    language=Language.MARKDOWN,
    chunk_size=100,
    chunk_overlap=10,
    length_function=num_tokens_from_string,
)

vectorstore= get_vectorstore(collection_name="big_fragments")

store= InMemoryStore()

retriever= ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=store,
```

```
    child_splitter=child_splitter,
    parent_splitter=parent_splitter,
)

retriever.add_documents(docs)
```

Self-retriever

A self-querying retriever, as the name implies, possesses the capability to generate queries internally. More precisely, when presented with a natural language query, the retriever utilises a query-constructing LLM chain to formulate a structured query. Subsequently, it applies this structured query to its underlying VectorStore. This unique functionality enables the retriever to not only perform semantic similarity comparisons between the user-input query and the contents of stored documents but also to extract filters from the user query related to the metadata of stored documents and execute those filters.

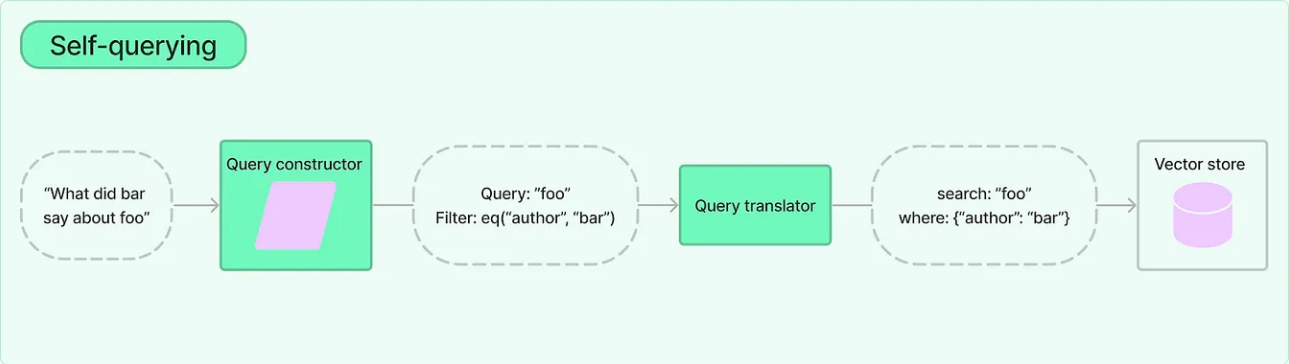


Image 13: Self-querying flow

In many cases, there is a desire to deduce metadata from the query itself. To tackle this, we can employ the SelfQueryRetriever, which utilises a language model to extract:

- The query string for vector search.
- A metadata filter to include.

Since most vector databases support metadata filters, there's no need for additional databases or indexes. This new metadata can be generated in the following manner:

```
class Tags(BaseModel):
    completeness: str= Field(
        description="Describes how useful the text is in terms of self-explanat
        enum=["Very", "Quite", "Medium", "Little", "Not"],
    )
    code_snippet: bool= Field(
        default=False,
        description="Whether the text fragment includes a code snippet. Code sr
    )
    description: bool= Field(
        default=False, description="Whether the text fragment includes a descri
    )

tagging_chain = create_tagging_chain_pydantic(Tags, llm)

tagging_results = tagging_chain.batch(
    inputs=[{"input": doc.page_content} for doc in docs[:200]],
    return_exceptions=True,
    config={
        "max_concurrency": 50,
    },
)

docs_with_tags = [
    Document(
        page_content=doc.page_content,
        metadata={
            **doc.metadata,
            **result.get("text").dict(),
        },
    )
    for doc, result in zip(docs, tagging_results)
    if not isinstance(result, Exception)
]

print(f"Documents with tags: {len(docs_with_tags)}")
```

To create our self-querying retriever, we'll need to provide upfront information about the metadata fields that our documents support and a brief description of the document contents. This can be achieved by adding attribute information.

```
from langchain.retrievers import SelfQueryRetriever
from langchain.chains.query_constructor.base import AttributeInfo

metadata_field_info= [
    AttributeInfo(
```

```
        name="completeness",
        description="Describes how useful the text is in terms of self-explanat
        type='enum=["Very", "Quite", "Medium", "Little", "Not"]',
    ),
    AttributeInfo(
        name="code_snippet",
        description="Whether the text fragment includes a code snippet. Code sr
        type="bool",
    ),
    AttributeInfo(
        name="description",
        description="Whether the text fragment includes a description.",
        type="bool",
    )
]

document_content_description = "Langchain documentation"
llm = ChatBedrock(temperature=0)
retriever = SelfQueryRetriever.from_llm(
    llm=llm,
    vectorstore=vectorstore,
    document_contents=document_content_description,
    metadata_field_info=metadata_field_info,
    enable_limit=True,
    verbose=True,
)
```

Multi-query retriever

The MultiQueryRetriever streamlines the prompt tuning process by employing a Large Language Model (LLM) to generate multiple queries from various perspectives based on a given user input query. For each generated query, it retrieves a set of relevant documents. The retriever then combines the unique union of documents across all queries, resulting in a more extensive collection of potentially relevant documents. By presenting multiple viewpoints on the same question, the MultiQueryRetriever aims to address certain limitations of distance-based retrieval and achieve a more diverse and enriched set of results.

```
from langchain.retrievers import MultiQueryRetriever

retriever= MultiQueryRetriever(
    retriever=vectorstore.as_retriever(),
    llm_chain=llm_chain,
)
```

Ensemble retriever

An ensemble retriever combines different retrieval methods to improve overall performance. In the context of document retrieval, a common pattern is to blend a sparse retriever, such as BM25, with a dense retriever, like embedding similarity. This combination, often referred to as “hybrid search,” capitalises on the complementary strengths of each retriever type. The sparse retriever excels in locating relevant documents based on keywords, while the dense retriever is adept at finding relevant documents through semantic similarity. This ensemble approach aims to enhance the retrieval process by leveraging the unique strengths of each method.

```
from langchain.retrieversimport BM25Retriever, EnsembleRetriever
from langchain.vectorstoresimport Chroma
from langchain.embeddings.bedrock import BedrockEmbeddings

bm25_retriever = BM25Retriever.from_documents(docs)
bm25_retriever.k = 2

vector_retriever = Chroma.from_documents(
    docs, embedding=BedrockEmbeddings()
).as_retriever(search_kwargs={"k": 2})
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever], weights=[0.5, 0.5]
)
```

Compression

To expand the number of results you can incorporate into the context, it’s beneficial to condense the responses to focus solely on the relevant information. Often, pertinent details within a document might be obscured by irrelevant text, and passing the entire document through your application can lead to more costly LLM calls and sub-optimal responses.

Contextual compression is designed to address this issue. It involves the process of distilling the most relevant information from a document, allowing for more efficient utilisation of LLM calls and yielding improved responses by emphasising the key content related to the query.

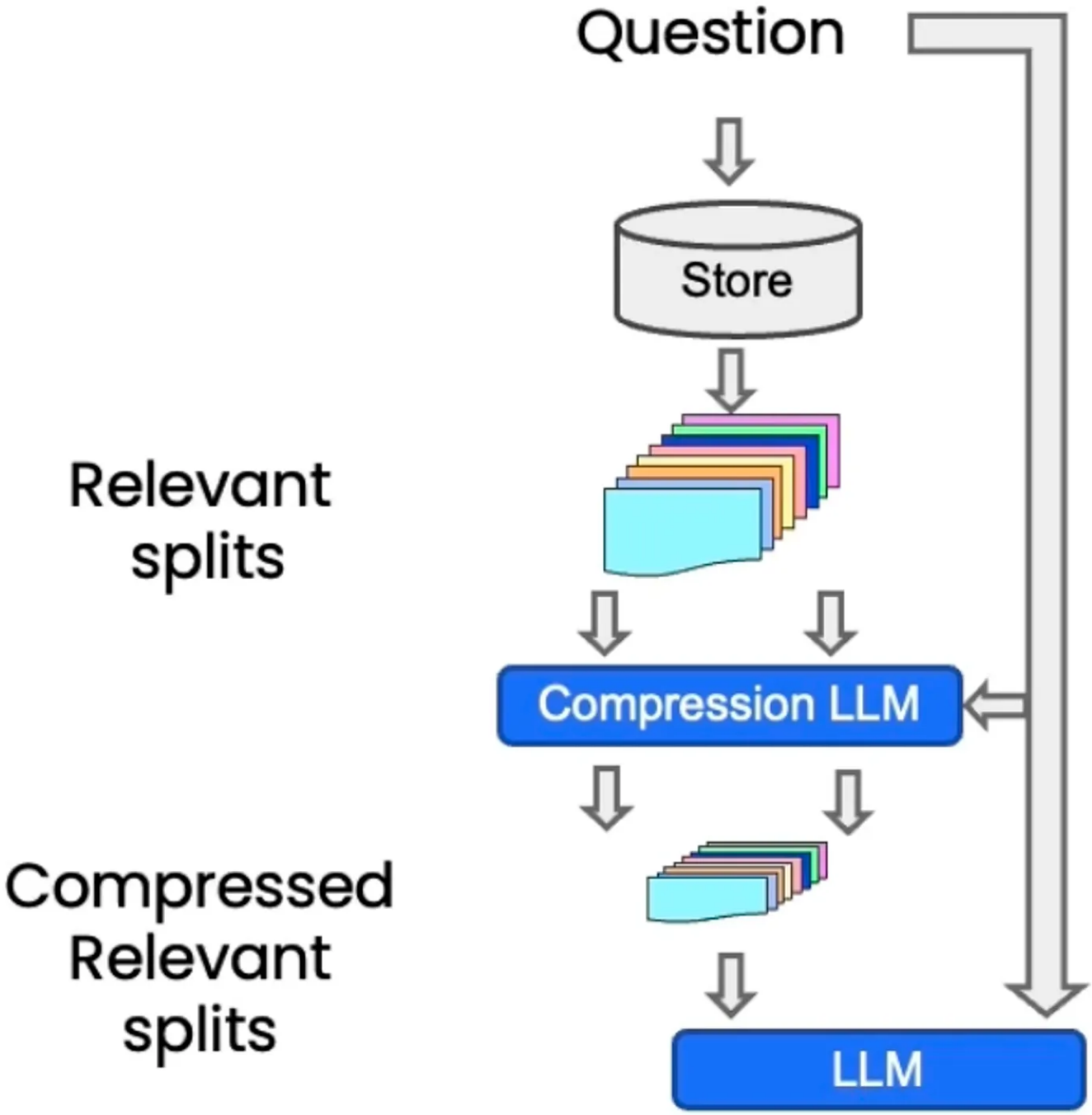


Image 14: Compressing relevant splits

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

compressor = LLMChainExtractor.from_llm(llm)

compression_retriever = ContextualCompressionRetriever(
base_compressor=compressor,
base_retriever=vectordb.as_retriever(search_type = "mmr")
)
```

```
question = "what does a data engineer?"
compressed_docs = compression_retriever.get_relevant_documents(question)
```

Conclusion

As you've seen, the RAG process involves more than just bringing in text and passing it as context to a LLM. There are numerous details to consider and various techniques to explore. One way to gauge the effectiveness of your process is through the RAG triad, which comprises three key aspects: answer relevance, context relevance and groundedness. In this link you can find examples.

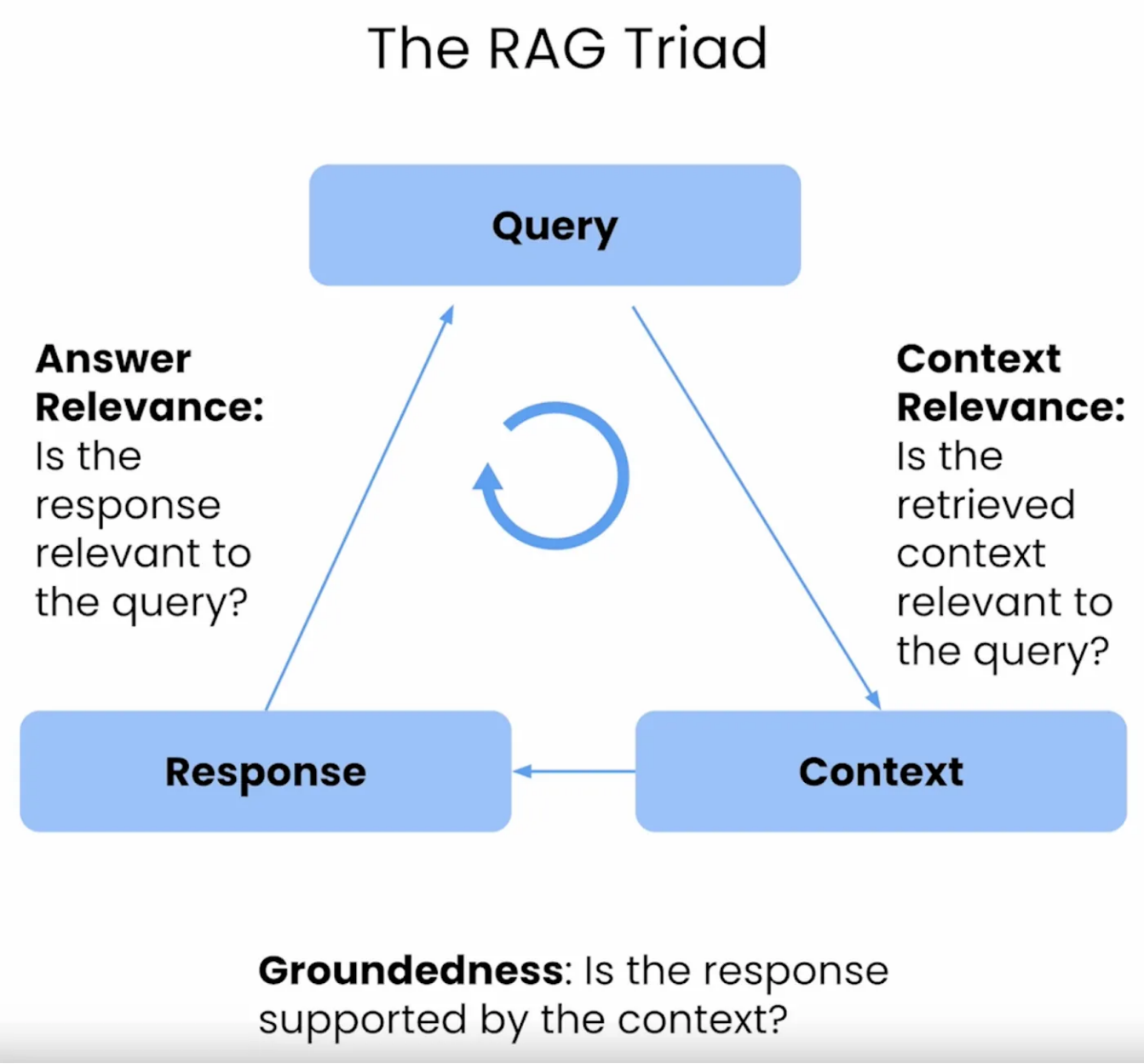


Image 15: The RAG Triad

Now, let's finalise this blog by explaining these three components:

- **Answer Relevance:** This refers to how well the generated response aligns with and directly addresses the user's query. The goal is to ensure that the answer provided is not only accurate but also closely related to what the user is looking for.

- **Context Relevance:** Context is crucial for understanding the nuances of a query. It involves evaluating how well the context, or additional information, is incorporated into the generation process. A contextually relevant response enhances the overall comprehension and coherence of the answer.

- **Groundedness:** This aspect assesses the extent to which the generated answer is firmly based on the information present in the provided context. A grounded response can be traced back to the input data and is not arbitrary or detached. It ensures that the generated content remains connected to the source material.

Considering and optimising these three elements within the RAG triad is essential for creating a robust and effective text generation process.

Data Engineer | Large Language Models | Retrieval Augmented | Langchain

Follow

Written by Eric Bellet

47 Followers · Writer for Adevinta Tech Blog

Enthusiastic computer scientist dedicated to the exploration and advancement of data and artificial intelligence technologies.

More from Eric Bellet and Adevinta Tech Blog

Eric Bellet in Adevinta Tech Blog

Data Engineer 2.0. Part I: Large Language Models

How ETL became LTE to support the demands of LLMs

5 min read · Feb 6, 2024

124 1

Ula La Paris in Adevinta Tech Blog

Deep Dive in PaddleOCR inference

Discover the complexities of using PaddleOCR as a Text in Image service and how the Cognition team overcame the challenges to improve user...

10 min read · Mar 6, 2023

178 8

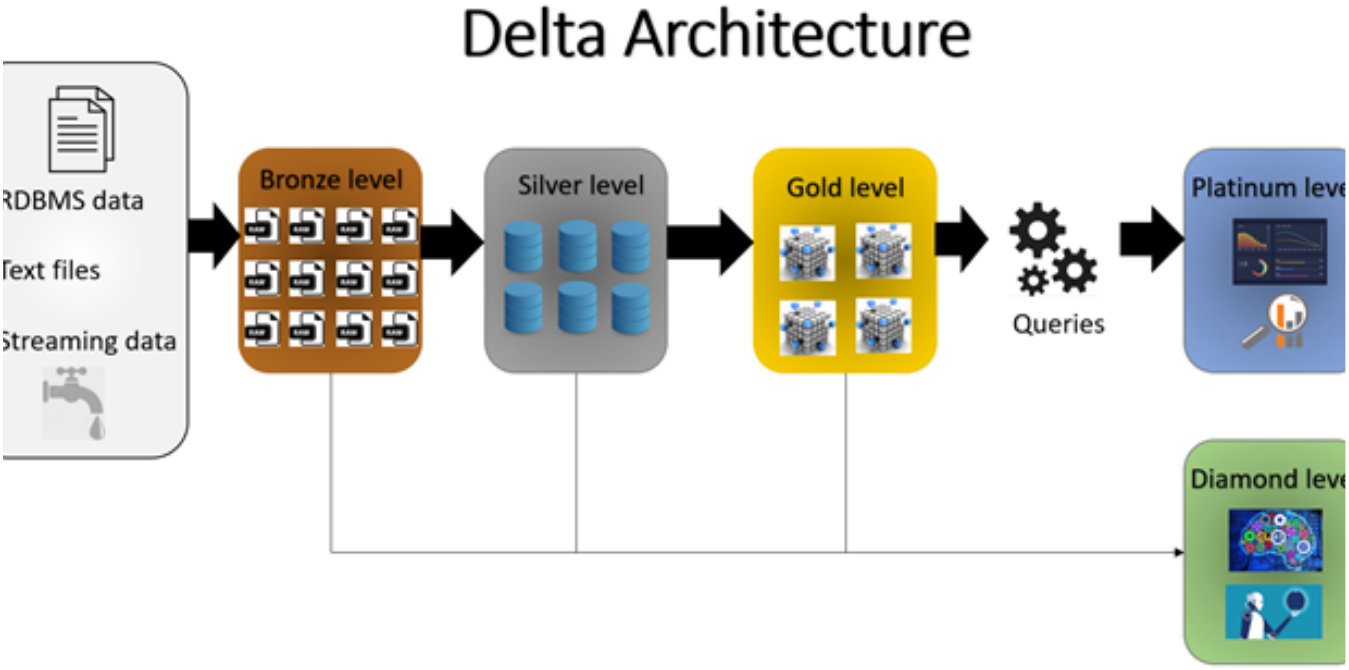
Tanat Lokejaroenlarb in Adevinta Tech Blog

It’s not always DNS—unless it is

Our journey investigating a months-long issue and what we learned from it

13 min read · Jun 13, 2023

718 11



Eric Bellet

Delta Lake Open Source Project

The first unified data management system.

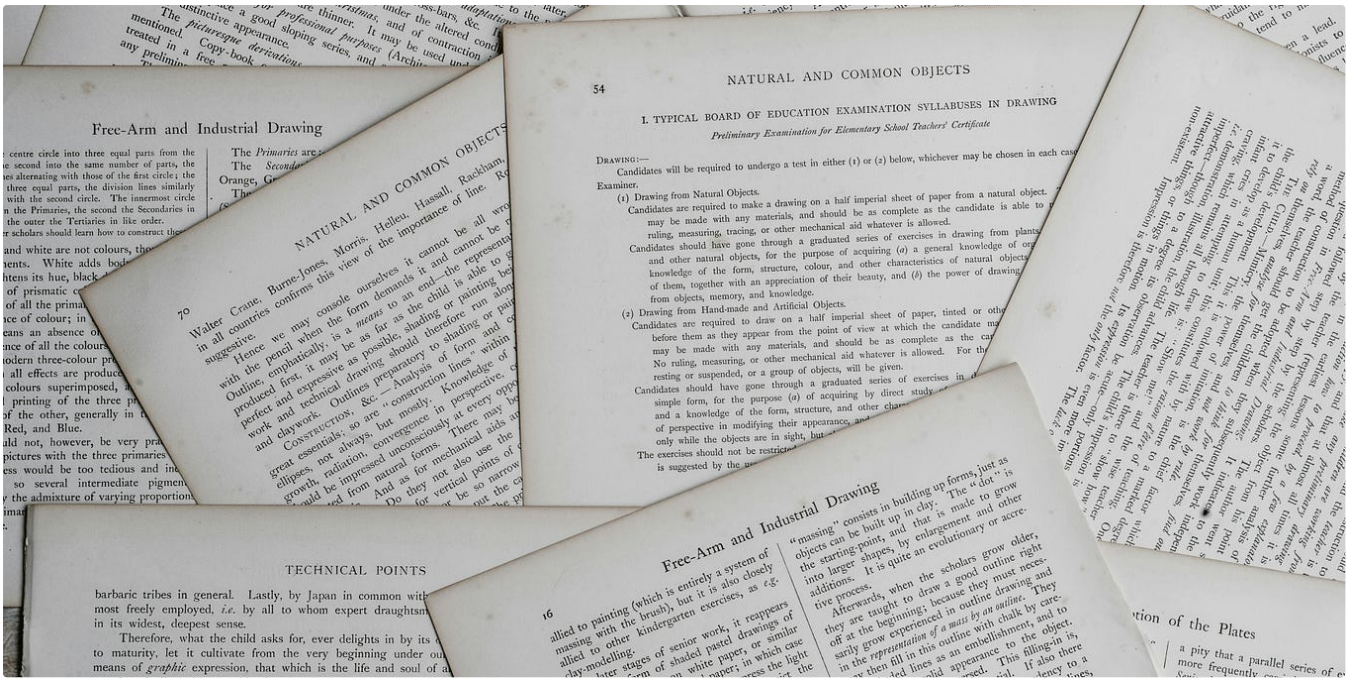
12 min read · May 13, 2019

97 7

See all from Eric Bellet

See all from Adevinta Tech Blog

Recommended from Medium



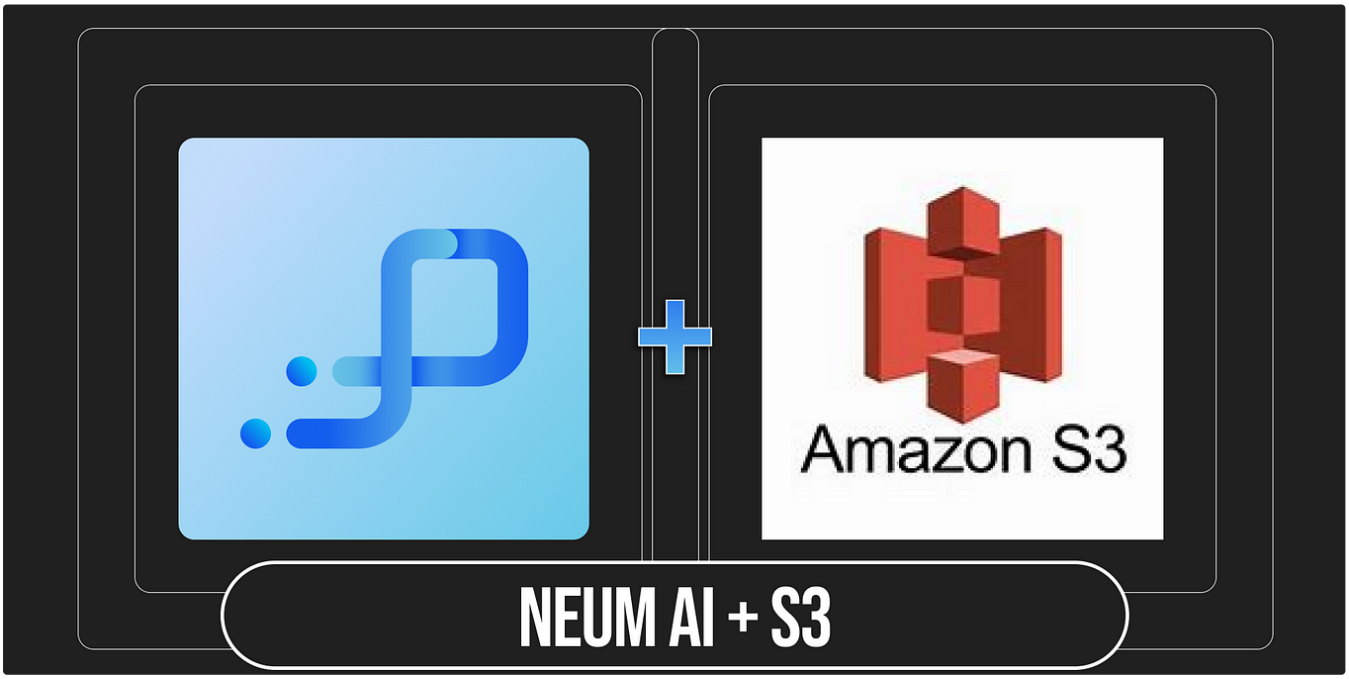
Rahul

Chunking(Text Splitting) Strategies- LLMs

What is the best way to divide the data into digestible pieces for my large language model to process and perform the tasks. Chunking...

4 min read · Mar 30, 2024

3



Neum AI

How to semantically search over documents in an S3 bucket with real-time changes?

Semantically searching over S3 data .. with realtime updates!— how to set up?

3 min read · Dec 15, 2023

7

Lists

Natural Language Processing

1488 stories · 999 saves

AI Regulation

6 stories · 472 saves

ChatGPT prompts

47 stories · 1624 saves

Generative AI Recommended Reading

52 stories · 1095 saves

Shweta Gargade

Breaking Down Text: Exploring Multiple Chunking Methods forRAG and LLM

Chunking strategies to enhance models like Retrieve-and-Generate (RAG) and Large Language Models (LLMs).

7 min read · Mar 28, 2024

51

Akash Deep

Data Preparation for RAG-Part 1

In this blog, we aim to thoroughly dissect and understand the entire RAG (Retrieval-Augmented Generation) process, with a special emphasis...

5 min read · Mar 2, 2024

5

Ida Silfverskiöld in Towards Data Science

Fine-Tune Smaller Transformer Models: Text Classification

Using Microsoft's Phi-3 to generate synthetic data

17 min read · 5 days ago

484 1

Philipp Kaindl

Building a RAG Agent with LangGraph, LLaMA3–70b, and Scaling with Amazon Bedrock

Explore how to build a local Retrieval-Augmented Generation (RAG) agent using LLaMA3, a powerful language model from Meta. This RAG agent...

19 min read · May 22, 2024

 113