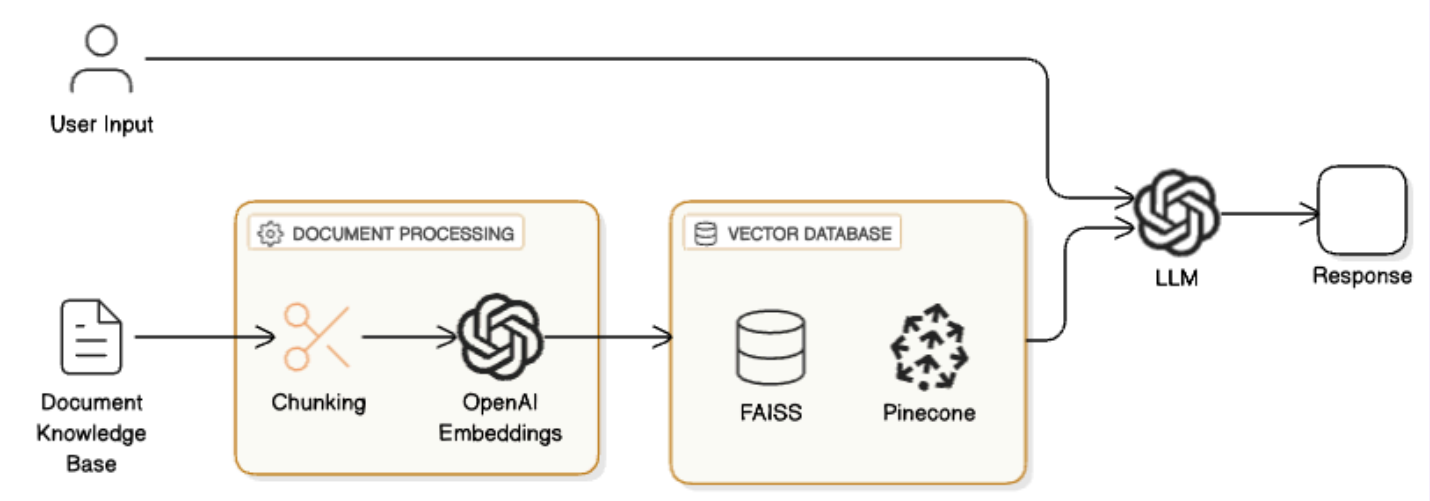
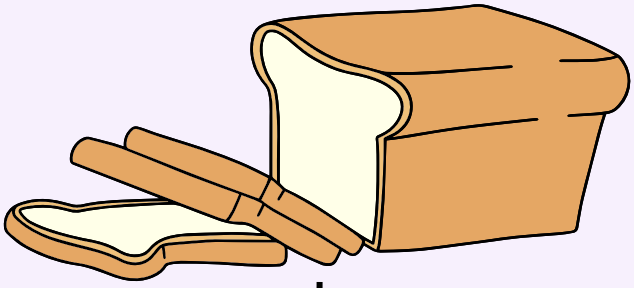
Retrieval Augmented Generation

Chunking strategies



Unlocking AI and data products, follow for powerful concepts, simplified

@rajbattula



LLM Context Window limitations

Retrieval relevance

Why is chunking required/important?

Context Preservation

Yeah ok got it, but why are there different chunking strategies?

Because not all text is created equal and no single strategy fits every case

@rajbattula

1a. Character-based Chunking

Splits text into chunks with a fixed number of characters. Can use specific separators (like paragraph or sentence marks) or just count individual characters.

When/Why to use:

- Useful when you need simple, predictable chunk sizes, regardless of content structure.
- Good for data with no clear internal structure (logs, anonymized, heavily formatted text).

To obtain the string content directly, use .split_text. To create LangChain Document objects (e.g., for use in downstream tasks and to propagate metadata associated with each document to the output chunks), use .create_documents.

LangChain Library:

CharacterTextSplitter

@rajbattula

1b. Token-based Chunking

Splits based on a fixed number of tokens, where tokens are counted by the LLM's tokenizer, making this suitable when model context limits are important.

When/Why to use:

- Essential for working with LLMs that have fixed token limits (most modern models).
- Prevents chunk overruns that could cause incomplete responses or truncation.

Other ways to chunk documents using token limits in the LangChain framework are to use the following methods

- from_tiktoken_encoder: a fast BPE tokenizer created by OpenAI.
- SpacyTextSplitter: spaCy is an open-source software library for advanced natural language processing, written in the programming languages Python and Cython.
- SentenceTransformersTokenTextSplitter: defaults to "sentence-transformers/all-mpnet-base-v2"
- from_huggingface_tokenizer: Hugging Face tokenizer, the GPT2TokenizerFast to count the text length in tokens

LangChain Library:

TokenTextSplitter

@rajbattula

2. Recursive Character-based Chunking

Attempts to preserve larger natural language units (paragraphs, then sentences, then words), only splitting at smaller scales if needed to fit chunk limits.

When/Why to use:

- Best for keeping semantic context, so that each chunk retains meaning and coherence.
- Great for document RAG where chunks with logical breaks (e.g., sections, paragraphs).

Recursive Character-based Chunking

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Recursive character splitting
recursive_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    separators=["\n\n", "\n", " ", ""]
)
recursive_chunks = recursive_splitter.split_text(text)
print(f"Number of recursive chunks: {len(recursive_chunks)}")
print(f"First recursive chunk: {recursive_chunks[0][:300]}...")
```

Use **RecursiveCharacterTextSplitter.from_language** when splitting code or text in a specific language (like Python, JavaScript, or Markdown), as it automatically selects separators that align with that language's structure.

Recursive Character-based Chunking for JavaScript code

```python
JS_CODE = """
function helloWorld() {
  console.log("Hello, World!");
}
// Call the function
helloWorld();
"""
js_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.JS, chunk_size=60, chunk_overlap=0
)
js_docs = js_splitter.create_documents([JS_CODE])
```

LangChain Library: RecursiveCharacterTextSplitter

@rajbattula

3. Structured/Text-Structure-Aware Chunking

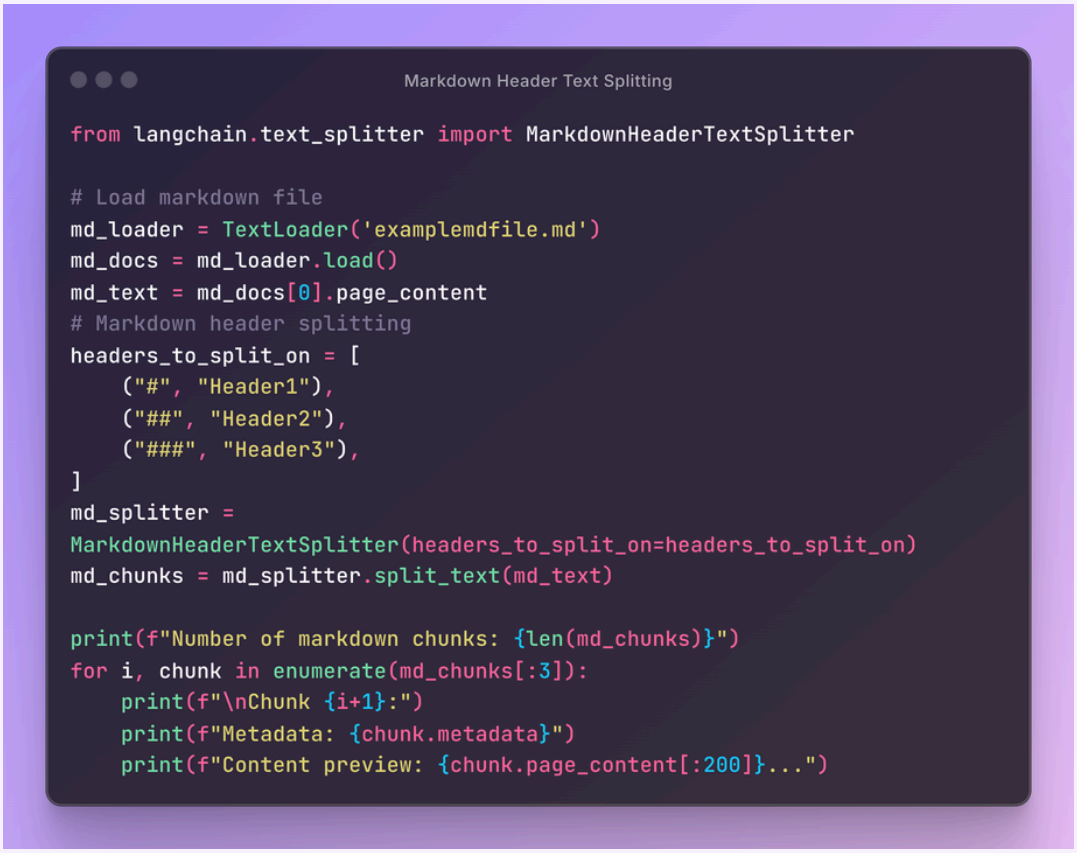
Splits using the natural structure of the document (e.g., by headings, list items, or custom-defined sections).

When/Why to use:

Preserves Logic and Hierarchy:

- Markdown splitting by headers keeps sections, subsections, and related content together, which maintains meaning and context for downstream processes.
- JSON splitting traverses the object structure, splitting on keys and arrays, so related data in an object stays grouped as they were intended in the source.

Metadata Enrichment: These splitters typically add metadata (e.g., header titles, keys) to each chunk, making it easier to filter, index, and provide context to embeddings.



Variations in these text splitters and their corresponding langchain libraries:

- HTMLHeaderTextSplitter, HTMLSectionSplitter, HTMLSemanticPreservingSplitter
- MarkdownHeaderTextSplitter
- RecursiveJsonSplitter

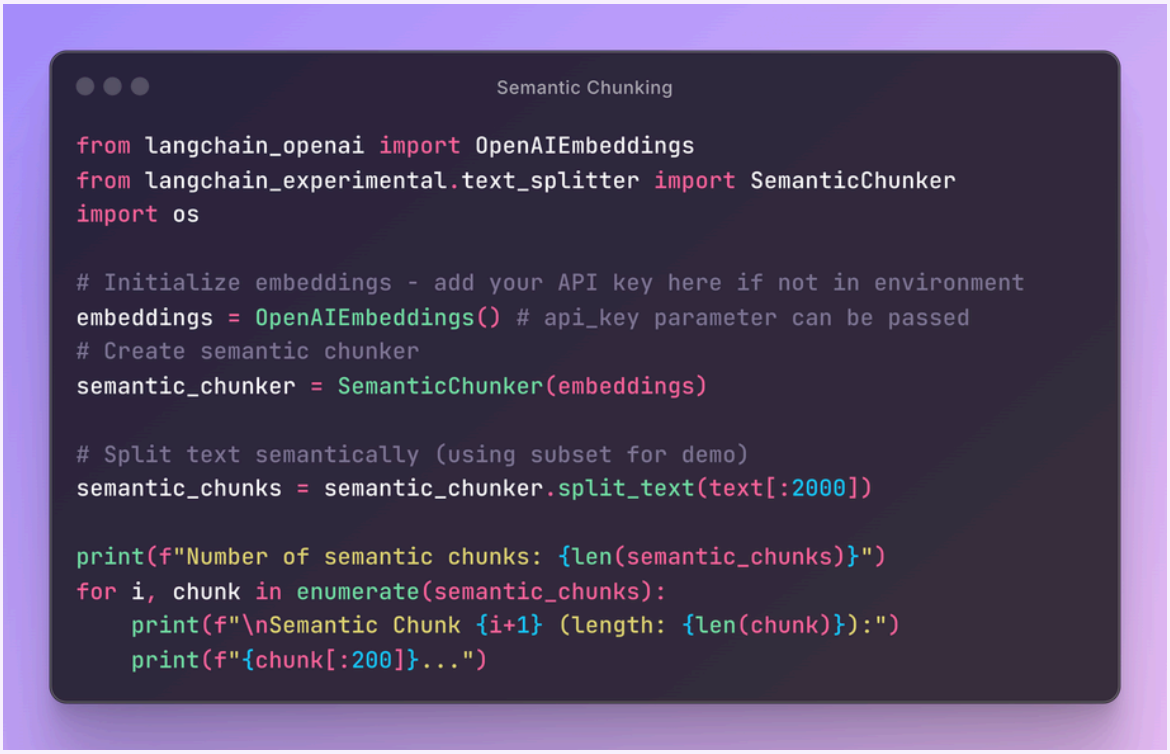
@rajbattula

4. Semantic Chunking

Uses semantic understanding to split where topic or meaning naturally shifts e.g., academic papers, legal documents, interviews, etc., rather than by fixed length or format.

When/Why to use:

- Ideal for maximizing information density in each chunk.
- Recent, more advanced chunking. This can be used when other schemes perform poorly (may require extra libraries or manual workflows in LangChain).



Semantic chunking can be **more computationally expensive** than simple splitting, as it requires generating embeddings for many text pieces and calculating similarity/distance metrics. Semantic chunking approaches supported by Langchain framework:

- **Percentile Chunker:** Cuts by percentile ranges of sentence lengths or embedding distances, good for splitting at regular meaning shifts.
- **Interquartile Chunker:** Groups chunks within the interquartile range of sentence lengths, targeting mid-length, semantically steady regions useful for scientific or regularly structured text.
- **Gradient Chunker:** Splits where embedding similarity changes sharply, useful for lecture transcripts, technical manuals, or mixed-topic documents.
- **Standard Deviation Chunker:** Splits where embedding distances vary most, marking significant meaning changes.

LangChain Library: SemanticChunker

@rajbattula