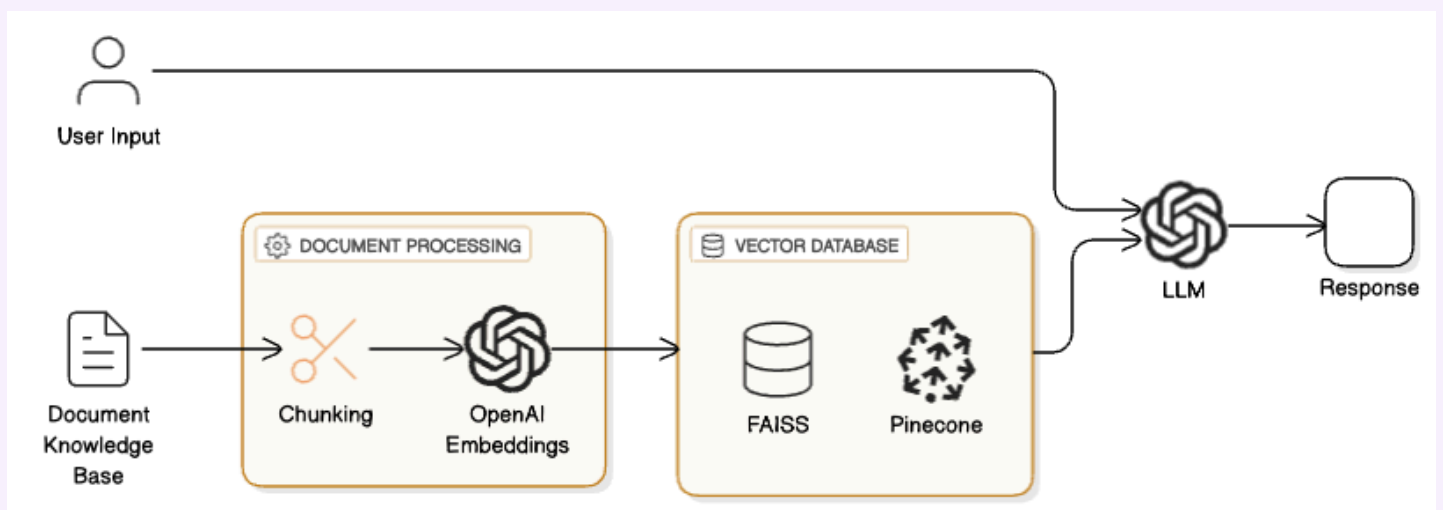


# Retrieval Augmented Generation

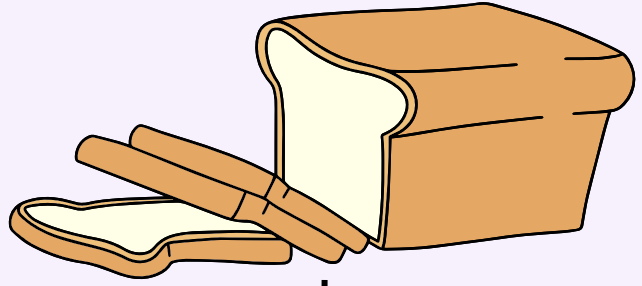
## Chunking strategies



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



[@rajbattula](#)



LLM Context  
Window  
limitations

Retrieval  
relevance

Context  
Preservation

Why is chunking  
required/important?



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).

```
Character Text Splitting

# Character-based splitting
from langchain.text_splitter import CharacterTextSplitter
char_splitter = CharacterTextSplitter(
    separator="\n\n",
    chunk_size=1000,
    chunk_overlap=100
)

char_chunks = char_splitter.split_text(text)
print(f"Number of chunks: {len(char_chunks)}")
print(f"First chunk length: {len(char_chunks[0])} characters")
print(f"First chunk: {char_chunks[0][:300]}...")
```

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.

```
Token-based Chunking

from langchain.text_splitter import TokenTextSplitter

# Token-based splitting
token_splitter = TokenTextSplitter(
    chunk_size=200,
    chunk_overlap=20
)

token_chunks = token_splitter.split_text(text)
print(f"Number of token-based chunks: {len(token_chunks)}")
print(f"First token chunk: {token_chunks[5][:500]}...")
```

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

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

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.

```
Markdown Header Text Splitting

from langchain.text_splitter import MarkdownHeaderTextSplitter

# Load markdown file
md_loader = TextLoader('examplemdfile.md')
md_docs = md_loader.load()
md_text = md_docs[0].page_content
# Markdown header splitting
headers_to_split_on = [
    ("#", "Header1"),
    ("##", "Header2"),
    ("###", "Header3"),
]
md_splitter =
MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
md_chunks = md_splitter.split_text(md_text)

print(f"Number of markdown chunks: {len(md_chunks)}")
for i, chunk in enumerate(md_chunks[:3]):
    print(f"\nChunk {i+1}:")
    print(f"Metadata: {chunk.metadata}")
    print(f"Content preview: {chunk.page_content[:200]}...")
```

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

from langchain_openai import OpenAIEmbeddings
from langchain_experimental.text_splitter import SemanticChunker
import os

# Initialize embeddings - add your API key here if not in environment
embeddings = OpenAIEmbeddings() # api_key parameter can be passed
# Create semantic chunker
semantic_chunker = SemanticChunker(embeddings)

# Split text semantically (using subset for demo)
semantic_chunks = semantic_chunker.split_text(text[:2000])

print(f"Number of semantic chunks: {len(semantic_chunks)}")
for i, chunk in enumerate(semantic_chunks):
    print(f"\nSemantic Chunk {i+1} (length: {len(chunk)}):")
    print(f"{chunk[:200]}...")
```

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](#)