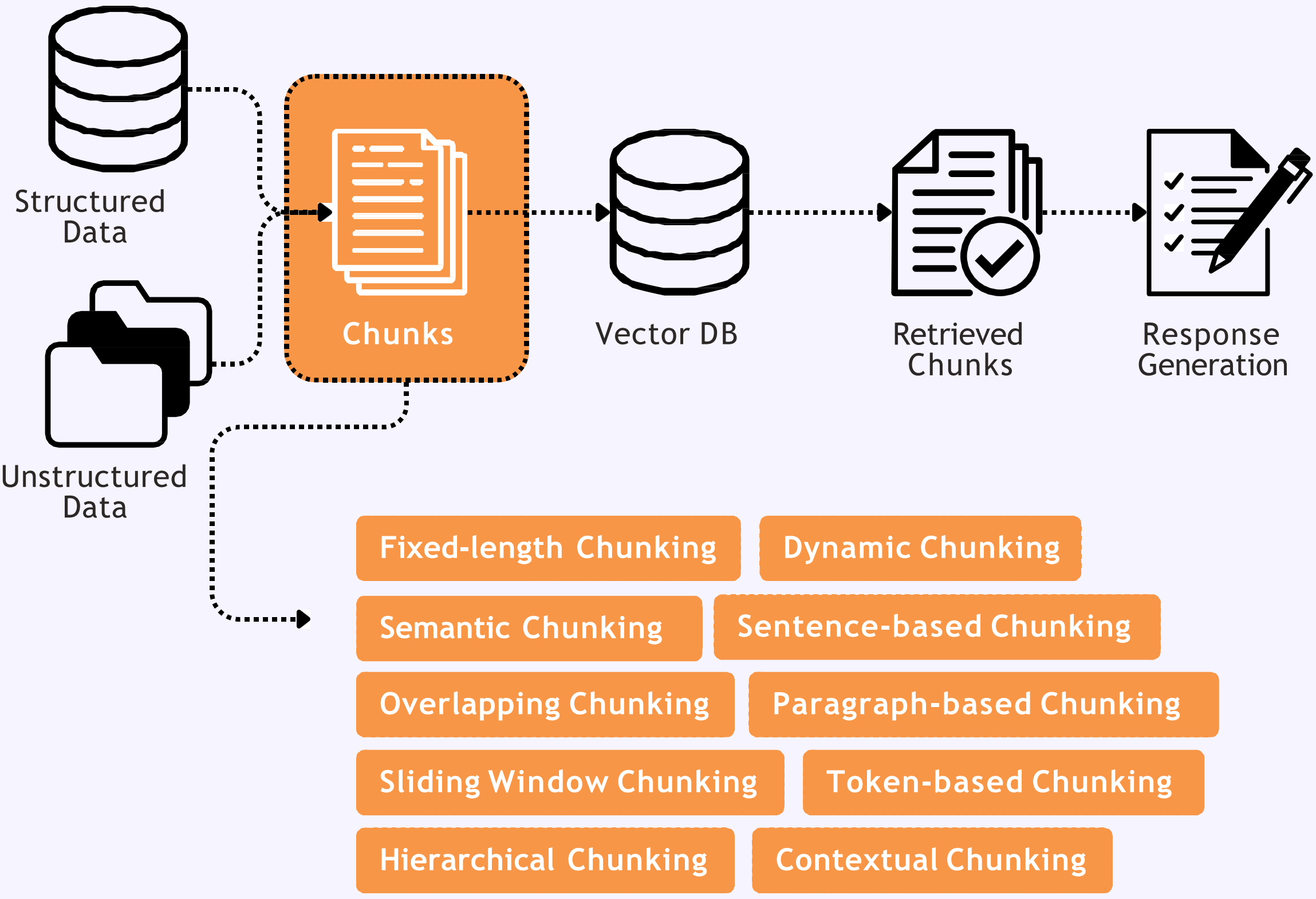
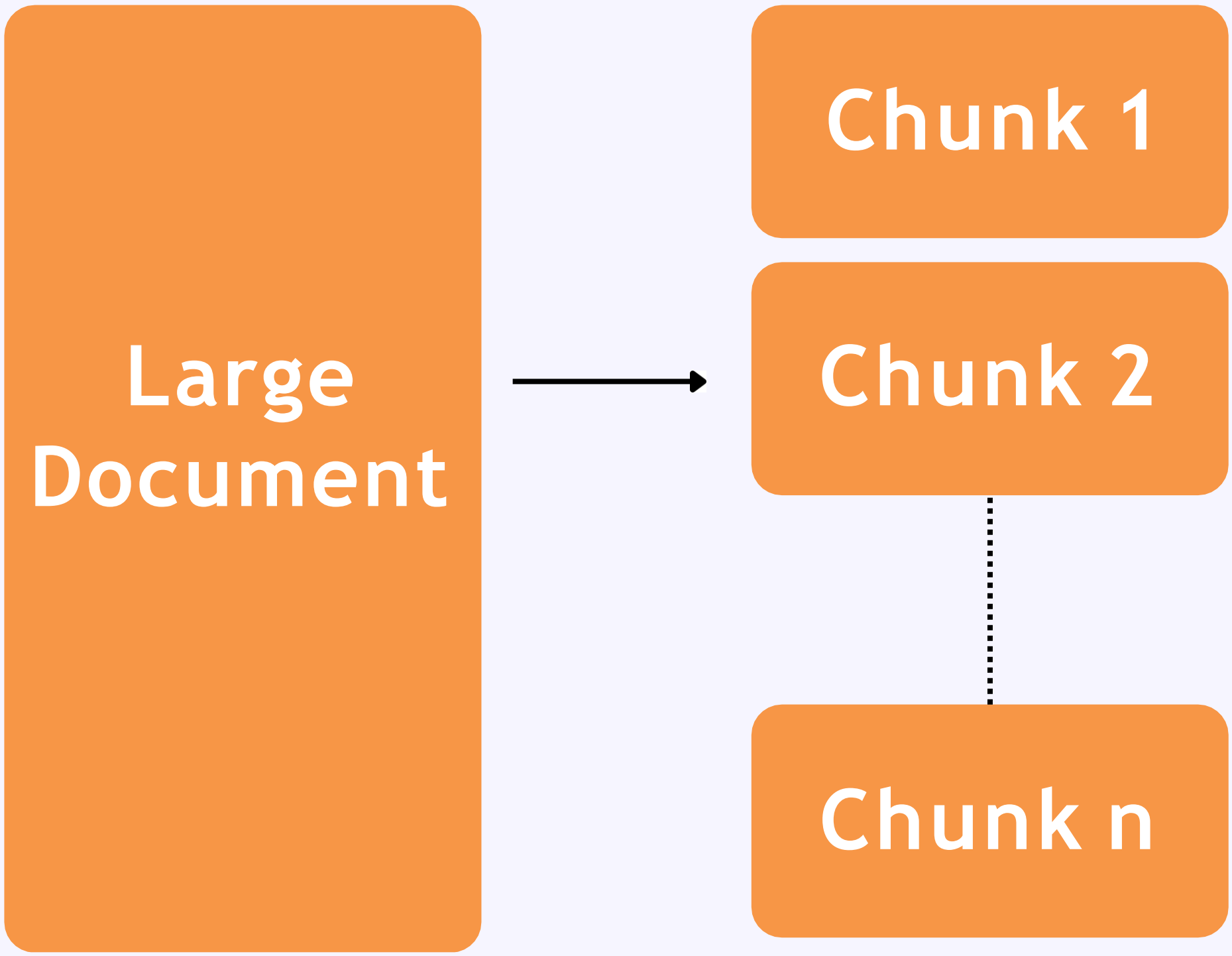
Different Types of Chunking Strategies



What is Chunking?

- Chunking is the process of breaking down a large documents into smaller, manageable pieces or chunks. One single document is known as one chunk.
- This helps in efficiently retrieving and generating relevant information from a vast amount of data.



Why is Chunking required?

- **Memory Limitation**
  - Large documents can exceed memory capacity.
  - Chunking breaks down the data into manageable chunks.

- **Processing Efficiency**
  - Smaller chunks are faster to process.
  - Reduces computational cost.

- **Improved Retrieved Accuracy**
  - Focuses on relevant sections. Enhances
  - context specific responses.

- **Simplifies Information Management** Easier to
  - navigate and search Facilitate quick
  - access to specific data

- **Scalability**
  - Allows to handle larger datasets
  - Makes system more robust and scalable.

Fixed-Length Chunking

- As the name suggest, we create chunk of data of fixed size from an existing document.

- It is a method of splitting text into chunks of a specific size, maintaining the overlap to ensure continuity between the chunks.
- The CharacterTextSplitter in langchain achives it by splitting the text by character count.

Steps

- **Input Document**
  - A long document or text that needs to be divided into smaller chunks.

- **Define Chunk Size**
  - Determine the fixed size of each chunk (e.g., 100 words, 500 characters, etc.).

- **Chunking Process**
    - The text is split into non-overlapping segments based on the defined size.
    - This can be done at the word level, token level, or character level.
    - In fixed-length chunking, chunks are created strictly based on the fixed size, regardless of sentence boundaries.
- **Output**
    - A list of chunks is generated, where each chunk contains the specified number of words, tokens, or characters.

Consideration

- **Loss of Context**: Since chunking does not consider sentence or paragraph boundaries, it may break sentences in the middle, potentially leading to incomplete ideas or a loss of context.
- **Memory Efficiency**: Fixed-length chunking helps to reduce memory overhead when processing long texts in small, consistent blocks.

Fixed Length Chunking in Python

```python
def fixed_length_chunking(text, chunk_size):
    # Step 1: Tokenize the text into words
    words = text.split()  # Splitting the text into individual words

    # Step 2: Create chunks of the specified size
    chunks = []
    for i in range(0, len(words), chunk_size):
        chunk = words[i:i + chunk_size]  # Get a chunk of the specified size
        chunks.append(' '.join(chunk))  # Join the words back into a string for each chunk

    return chunks

# Example usage:

text = """
Retrieval-augmented generation systems (RAG) combine pre-trained language models with
document retrieval mechanisms
to enhance the generation of responses. By splitting a long text into smaller chunks,
it becomes easier to retrieve
and process the most relevant information from a large corpus of documents. Fixed-length
chunking divides the text
into equal-sized segments, which are then used as input to the retrieval system. However,
it may break sentences or
paragraphs mid-way, which could affect the coherence of the retrieved information.
"""

# Define the chunk size (in terms of words)
chunk_size = 20

# Get the chunks
chunks = fixed_length_chunking(text, chunk_size)

# Output the chunks
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}:\n{chunk}\n")
```

Output

```
Chunk 1:
Retrieval-augmented generation systems (RAG) combine pre-trained language models with document
retrieval mechanisms to enhance the generation of responses. By splitting a long text into smaller

Chunk 2:
chunks, it becomes easier to retrieve and process the most relevant information from a large corpus of
documents. Fixed-length chunking divides the text into equal-sized segments, which are

Chunk 3:
then used as input to the retrieval system. However, it may break sentences or paragraphs mid-way,
which could affect the coherence of the retrieved information.
```

Breakdown of the Code

- **Tokenization**: The text is split into individual words using the .split() method.
- **Chunk Creation**: Using a loop, we iterate over the list of words and create chunks of the specified size (20 words in this example).
- **Output**: The chunks are printed as separate segments of text.

Fixed-Length Chunking in Langchain

```
from langchain.text_splitter import CharacterTextSplitter

# Sample text to split
text = """
Retrieval-Augmented Generation (RAG) models combine pre-trained language models with a retrieval
mechanism to enhance response generation.
Chunking in this context refers to dividing a large document or text into smaller, more manageable
pieces for effective retrieval and processing.
Fixed-length chunking splits the text into equally-sized segments, regardless of sentence or paragraph
boundaries. This is efficient for indexing and retrieval
but may cause sentences to be broken between chunks, which can affect contextual understanding. Using
this technique in RAG ensures that the system can process
long documents efficiently by retrieving the most relevant parts from the indexed chunks.
"""

# Step 1: Define the chunk size and overlap
# We'll use a CharacterTextSplitter with fixed-length character chunks

chunk_size = 200  # Length of each chunk (number of characters)
chunk_overlap = 20  # Overlapping characters between chunks to preserve context

text_splitter = CharacterTextSplitter(
    separator=" ",  # Split by space to avoid cutting words
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
    length_function=len  # We are counting length in terms of characters
)

# Step 2: Split the text into chunks
chunks = text_splitter.split_text(text)

# Step 3: Output the chunks
for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}:\n{chunk}\n")
```

Output

```
Chunk 1:
Retrieval-Augmented Generation (RAG) models combine pre-trained language models with a retrieval
mechanism to enhance response generation. Chunking in this context refers to dividing a large document
or text

Chunk 2:
document or text into smaller, more manageable pieces for effective retrieval and processing. Fixed-
length chunking splits the text into equally-sized segments, regardless of sentence or paragraph
boundaries.

Chunk 3:
regardless of sentence or paragraph boundaries. This is efficient for indexing and retrieval but may
cause sentences to be broken between chunks, which can affect contextual understanding. Using this
technique
```

Breakdown of the Code

CharacterTextSplitter:

- The CharacterTextSplitter is used here to split the text into chunks based on a fixed number of characters.
- It also allows you to specify an overlap, meaning that consecutive chunks will share a few characters, which can help retain context.

Parameters

- **chunk_size**: Number of characters each chunk will contain (in this example, 200 characters).
- **chunk_overlap**: Number of overlapping characters between chunks to preserve context across splits (20 characters in this example).
- **separator**: Ensures that we don’t split in the middle of a word. In this case, it's set to a space to split between words.

When to use Fixed-Length Chunking

- When the data is uniform in structure: Ideal for tasks like processing log files or structured datasets where the content length is predictable.

- When memory and processing efficiency are a priority: Useful when handling large datasets or models with strict token limits where predictability of chunk size helps with system optimization.

- When maintaining simple processing pipelines: Beneficial when minimizing complexity in chunking logic is important, especially for early-stage prototyping.

Advantages of Fixed-Length Chunking

- Simple Implementation: Easy to implement, as the chunk size is predetermined and doesn't require complex processing.

- Efficient for Uniform Data: Works well for text where content length is consistent and uniform, allowing predictable chunk sizes.

- Scalable: Fixed-length chunks are scalable for systems that need predictable resource allocation and processing limits (e.g., token limits in LLMs).

Srinivas Mahakud
Cloud & AI Leader @EY

For More Such Content

Srinivas Mahakud

Follow Me On LinkedIn

https://www.linkedin.com/in/srinivasmahakud/