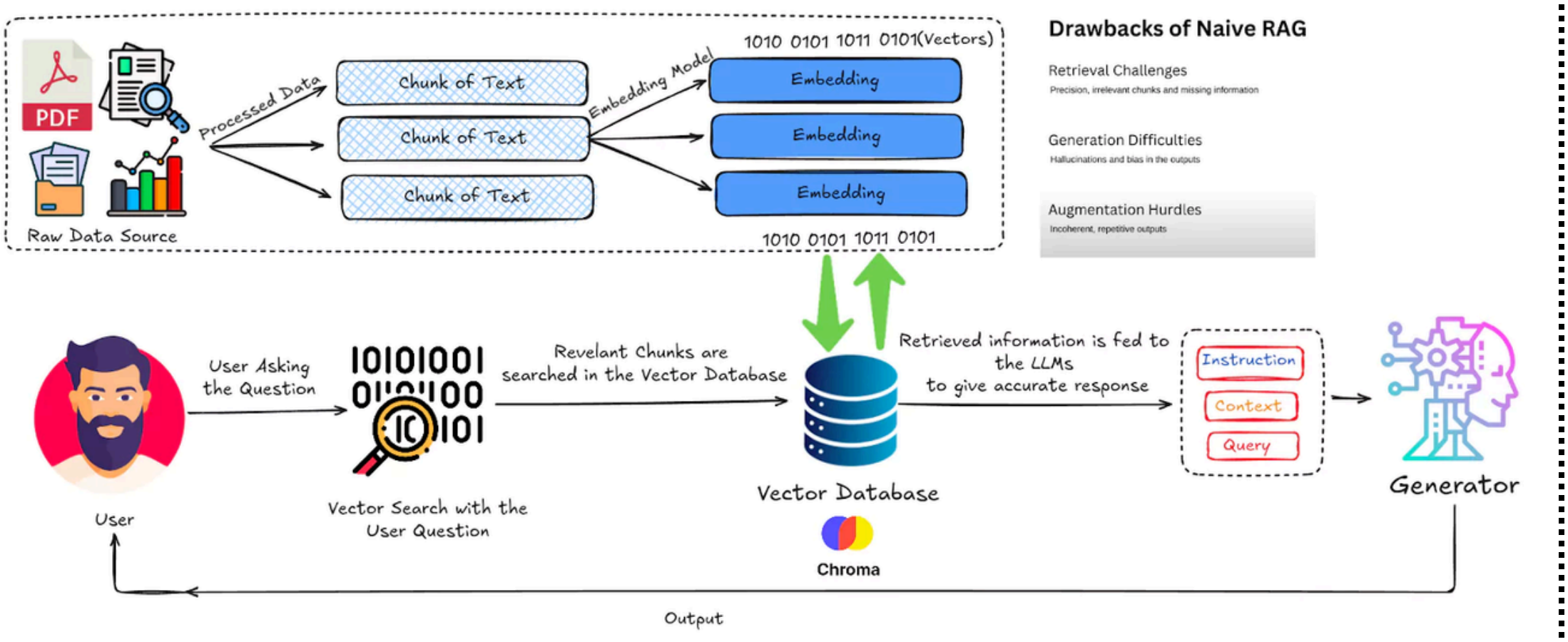
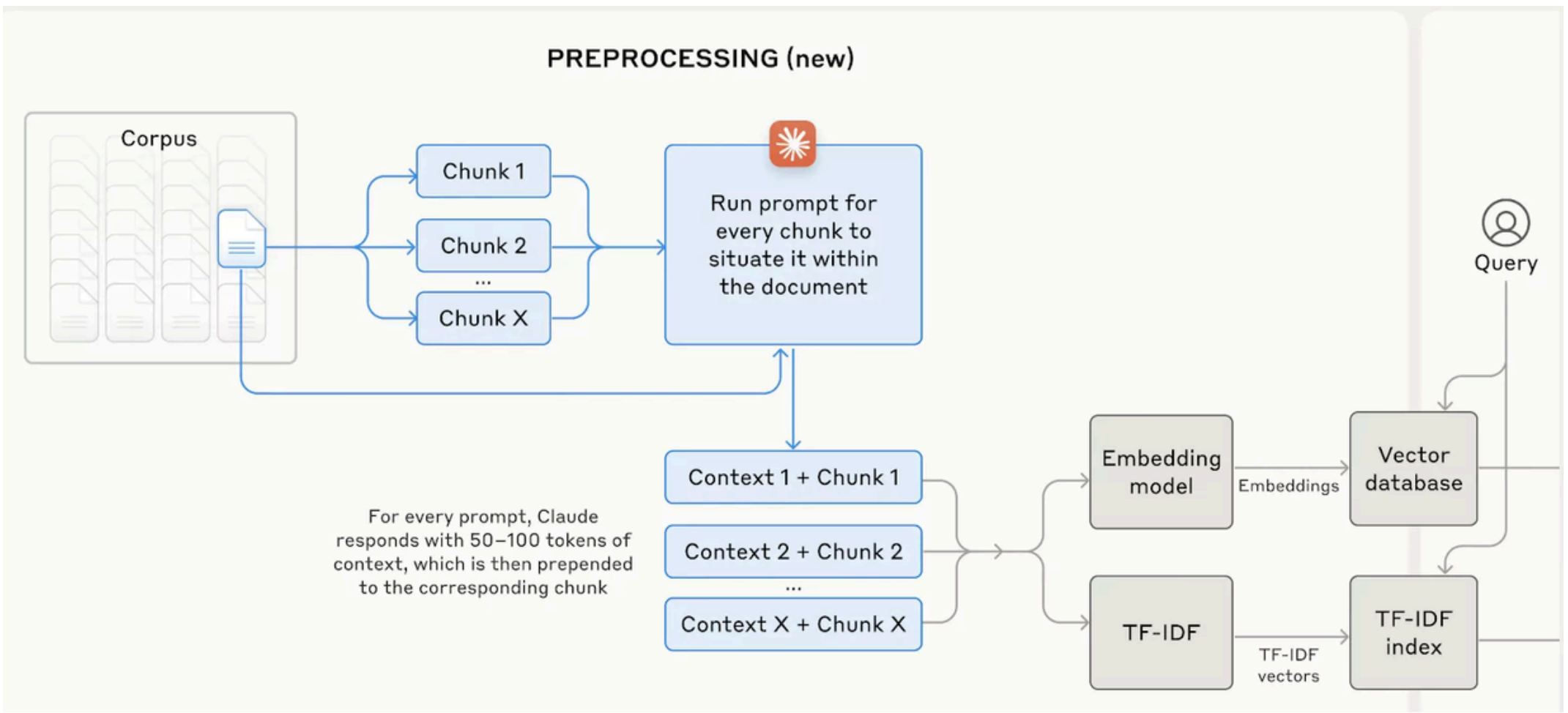
Analytics Vidhya

Mastering RAG

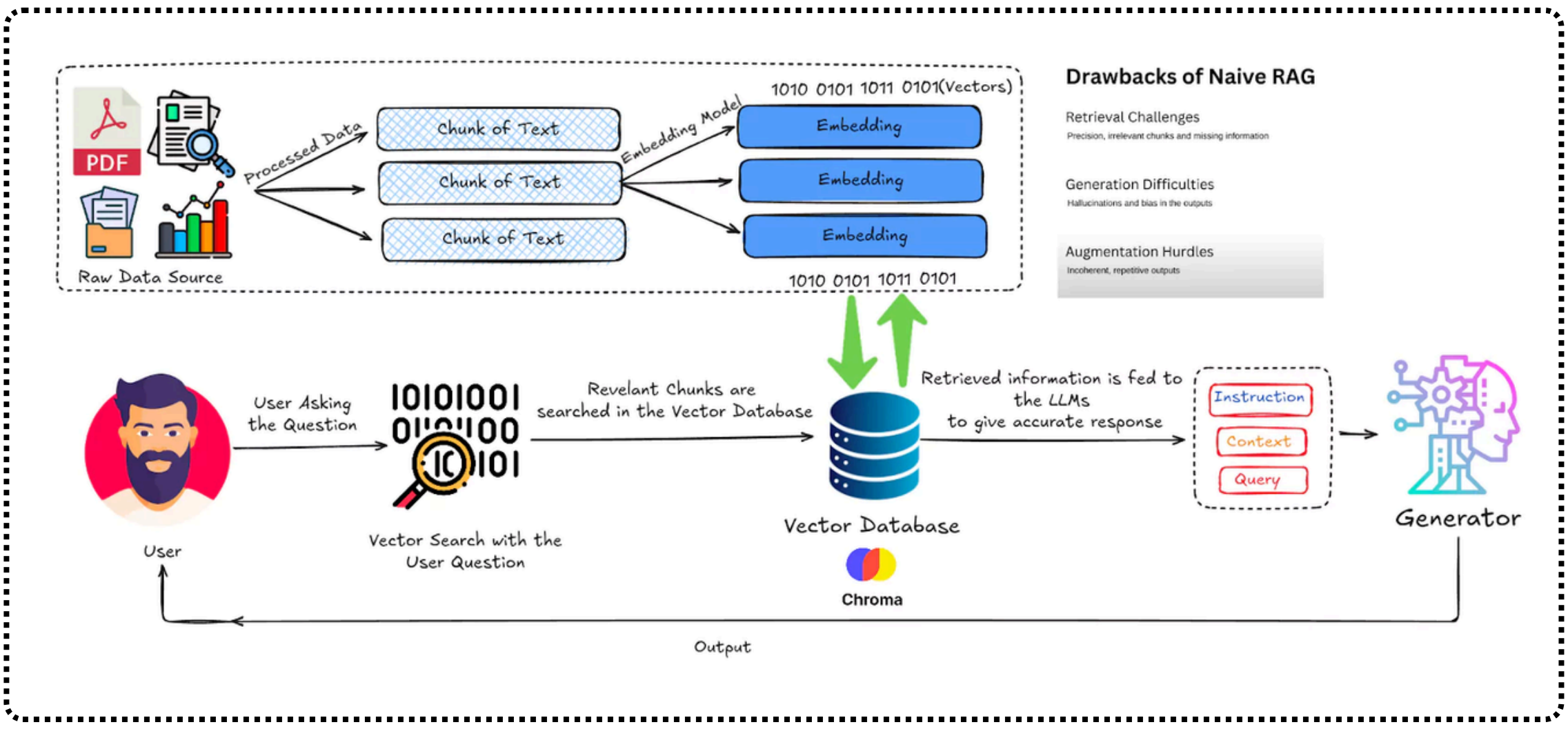
Chunking in RAG





Introduction

Retrieval-Augmented Generation (RAG) is a powerful technique that combines retrieval-based and generative approaches to improve the accuracy and relevance of responses in AI-driven applications. One of the most critical aspects of RAG is chunking, the process of breaking down large documents into smaller, meaningful segments that can be efficiently retrieved and processed by the model.



Importance of Chunking in RAG

Chunking plays a vital role in ensuring that the retrieval step fetches the most relevant information for generation. Poor chunking strategies can lead to:

- **Loss of Context**: Chunks that are too small may not carry enough information, making retrieval less useful.
- **Irrelevant Information**: Chunks that are too large may contain unnecessary details, leading to confusion in the generation step.
- **Inefficient Retrieval**: Badly structured chunks increase retrieval complexity and reduce performance.

Chunking Strategies

1. Fixed-Length Chunking

This approach divides text into uniform chunks based on a predefined token or word count (e.g., 256 or 512 tokens per chunk).

Pros:

- Simple to implement.
- Ensures consistent chunk sizes.

Cons:

- May cut off meaningful sentences.
- Lacks adaptability to different document structures.

2. Semantic Chunking

Semantic chunking is an advanced text-splitting technique that focuses on dividing a document into meaningful chunks based on the actual content and context rather than arbitrary size-based methods such as token count or delimiters. The primary goal of semantic chunking is to ensure that each chunk contains a single, concise meaning, optimizing it for downstream tasks like embedding into vector representations for machine learning applications.

Traditional chunking methods, such as splitting text by a fixed number of tokens or characters, often result in chunks that contain multiple, unrelated meanings. This can dilute the representation when encoding text into vector embeddings, leading to suboptimal retrieval and processing results. By contrast, semantic chunking works by identifying natural meaning boundaries within the text and segmenting it accordingly to ensure each chunk preserves a coherent and unified concept.

3. Sliding Window Chunking

A technique where overlapping chunks are created to preserve continuity of information. For example, if a chunk is 512 tokens, the next chunk may start from the 256th token of the previous chunk.

Pros:

- Maintains context between adjacent chunks.
- Reduces the chance of missing critical information.

Cons:

- Increases storage and computation costs.

4. Hierarchical Chunking

Uses multiple levels of chunking, starting with broader segments (e.g., sections, paragraphs) and then finer chunks within them.

Pros:

- Allows for multi-level retrieval.
- Optimizes information density.

Cons:

- Requires more sophisticated retrieval strategies.

Implementing Chunking in RAG

1. Using NLP Libraries

Libraries such as NLTK, SpaCy, or Hugging Face Transformers provide tools for segmenting text into sentences, paragraphs, and topic-based chunks.

2. Vectorization for Retrieval

Each chunk is converted into embeddings using models like BERT, Sentence Transformers, or OpenAI's Embeddings and stored in a vector database like FAISS, Chroma, or Pinecone.

3. Optimizing Chunk Size

Experimentation is necessary to determine the ideal chunk size based on:

- The retrieval model (e.g., dense retrieval vs. sparse retrieval).
- The length of user queries.
- The nature of the documents being chunked.

Analytics Vidhya

Python Code to implement Chunking

```
import nltk
import spacy
from transformers import AutoTokenizer


# Download necessary resources for NLTK
nltk.download('punkt')
nlp = spacy.load('en_core_web_sm')


class RAGChunker:
    def __init__(self, model_name='bert-base-uncased', chunk_size=512,
overlap=256):tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.chunk_size = chunk_size
        self.overlap = overlap


    def fixed_length_chunking(self, text):
        """
        Splits text into fixed-size chunks based on token count.
        """
        tokens = self.tokenizer.tokenize(text)
        chunks = []
        for i in range(0, len(tokens), self.chunk_size):
            chunk = tokens[i: i + self.chunk_size]
            chunks.append(self.tokenizer.convert_tokens_to_string(chunk))
        return chunks



    def sliding_window_chunking(self, text):
        """
        Splits text into overlapping chunks to maintain context continuity.
        """
        tokens = self.tokenizer.tokenize(text)
        chunks = []
        for i in range(0, len(tokens), self.chunk_size - self.overlap):
            chunk = tokens[i: i + self.chunk_size]
            chunks.append(self.tokenizer.convert_tokens_to_string(chunk))
        return chunks

}
```

Analytics Vidhya

Example Usage

```
# Example Usage
text = """Your long document text goes here. The chunking strategies will divide it
chunker = AGRAGChunker(chunk_size=128, overlap=64)


# Applying different chunking methods
fixed_chunks = chunker.fixed_length_chunking(text)
semantic_chunks = chunker.semantic_chunking(text)
sliding_window_chunks = chunker.sliding_window_chunking(text)


print("\nFixed-Length Chunking:", fixed_chunks)
print("\nSemantic Chunking:", semantic_chunks)
print("\nSliding Window Chunking:", sliding_window_chunks)
```