What is Text Chunking?

Text chunking is the process of splitting large documents into smaller, manageable pieces called "chunks" for processing by AI models.

Instead of feeding entire documents, we break them into **semantically meaningful segments** that preserve context and meaning.

Why is Chunking Critical?

→ LLMs have token limits (4K-128K tokens)

→ Better chunks = Better retrieval in RAG systems

→ Impacts answer quality, relevance & cost

→ Wrong chunking = Lost context & poor results

1. Fixed Chunking

Splits text into equal-sized chunks (by word count)

PYTHON EXAMPLE

```
def fixed_chunking(text, chunk_size=100):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size])
              for i in range(0, len(words), chunk_size)]
    return chunks

# Usage
chunks = fixed_chunking(text, chunk_size=10)
```

✅ PROS

Simple, fast, predictable

❌ CONS

May break sentences

🎯 USE CASE:

Quick prototyping

1. Fixed Chunking

Splits text into chunks with an overlap for smoother transitions

PYTHON EXAMPLE

```
def overlapping_chunking(text, chunk_size=100, overlap=50):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size])
                for i in range(0, len(words), chunk_size -
overlap)]
    return chunks

# Usage
chunks = overlapping_chunking(text, chunk_size=20,
overlap=5)
```

✅ PROS

Maintains context flow, prevents information loss

❌ CONS

More storage, potential redundancy

🎯 USE CASE:

2. Overlapping Chunking

Splits text into chunks with an overlap for smoother transitions

PYTHON EXAMPLE

```python
def overlapping_chunking(text, chunk_size=100, overlap=50):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size])
                for i in range(0, len(words), chunk_size -
overlap)]
    return chunks

# Usage
chunks = overlapping_chunking(text, chunk_size=20,
overlap=5)
```

✅ PROS

Maintains context flow, prevents information loss

❌ CONS

More storage, potential redundancy

🎯 USE CASE:

3. Semantic Chunking

Uses spaCy to split text into meaningful sentences

PYTHON EXAMPLE

```python
import spacy

def semantic_chunking(text):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(text)
    chunks = [sent.text for sent in doc.sents]
    return chunks

# Usage
chunks = semantic_chunking(text)
```

✅ PROS

Respects sentence boundaries, maintains meaning

❌ CONS

Requires spaCy library, slower th
fixed

🎯 USE CASE:

Chatbots, content summarization 🤖

4. Recursive Character Chunkin

Splits text recursively based on character count, prioritizing word boundaries

PYTHON EXAMPLE

```
def recursive_chunk(text, max_size):
    if len(text) <= max_size:
        return [text]
    split_point = text.rfind(" ", 0, max_size)
    if split_point == -1:
        split_point = max_size
    chunk = text[:split_point]
    remaining_text = text[split_point:].strip()
    return [chunk] + recursive_chunk(remaining_text, max_size)

# Usage
chunks = recursive_chunk(text, 100)
```

✅ PROS

Finds nearest space, avoids word splitting

❌ CONS

Recursive overhead, variable chunk sizes

🎯 USE CASE:

5. Agentic Chunking

Uses an AI agent (via Groq API) to split text meaningfull
for a given task

PYTHON EXAMPLE

```
from groq import Groq

def agentic_chunking(text, task="summarize"):
    client = Groq(api_key="your_api_key")
    prompt = f"Split the following text into meaningful chunks
for the task: {task}\n\n{text}"
    response = client.chat.completions.create(
        model="llama3-70b-8192",
        messages=[{"role": "user", "content": prompt}]
    )
    chunks = response.choices[0].message.content.split('\n')
    return chunks
```

✅ PROS

AI decides chunking based on task, highly adaptive

❌ CONS

API costs, slower, requires internet

🎯 USE CASE:

6. Advanced Semantic Chunkin

Uses SentenceTransformer and KMeans clustering to grou
semantically similar sentences

PYTHON EXAMPLE

```
from sentence_transformers import SentenceTransformer
from sklearn.cluster import KMeans

def advanced_semantic_chunking(text, num_chunks=15):
    model = SentenceTransformer('all-MiniLM-L6-v2')
    sentences = text.split('. ')
    embeddings = model.encode(sentences)
    kmeans = KMeans(n_clusters=num_chunks)
    kmeans.fit(embeddings)
    clusters = kmeans.labels_
    chunks = [[] for _ in range(num_chunks)]
    for i, cluster in enumerate(clusters):
        chunks[cluster].append(sentences[i])
    return [' '.join(chunk) for chunk in chunks]
```

✅ PROS

ML-powered semantic grouping, topic clustering

❌ CONS

Requires ML libraries, slower processing

🎯 USE CASE:

7. Context Enriched Chunking

Combines surrounding sentences to add context to each chunk

PYTHON EXAMPLE

```
def context_enriched_chunking(text, window_size=2):
    sentences = text.split('. ')
    chunks = []
    for i in range(len(sentences)):
        start = max(0, i - window_size)
        end = min(len(sentences), i + window_size + 1)
        chunk = '. '.join(sentences[start:end])
        chunks.append(chunk)
    return chunks

# Usage
chunks = context_enriched_chunking(text, window_size=1)
```

✅ PROS

Rich context, better understanding

❌ CONS

Larger chunks, more storage

🎯 USE CASE:

8. Paragraph Chunking

Splits text based on paragraphs (using double line breaks

PYTHON EXAMPLE

```python
def paragraph_chunking(text):
    paragraphs = text.split('\n\n')
    return paragraphs

# Usage
chunks = paragraph_chunking(text)
```

✅ **PROS**

Preserves document structure, respects author intent

❌ **CONS**

Variable chunk sizes, requires proper formatting

🎯 **USE CASE:**

Articles, reports, documentation ✨

9. Recursive Sentence Chunking

Recursively splits text into chunks based on a set number of sentences

PYTHON EXAMPLE

```python
def recursive_sentence_chunking(text, max_sentences=3):
    sentences = text.split('. ')
    if len(sentences) <= max_sentences:
        return ['. '.join(sentences)]
    chunk = '. '.join(sentences[:max_sentences])
    remaining = '. '.join(sentences[max_sentences:])
    return [chunk] + recursive_sentence_chunking(remaining,
max_sentences)

# Usage
chunks = recursive_sentence_chunking(text, max_sentences=3)
```

✅ PROS

Balanced chunks, sentence-aware

❌ CONS

Recursive overhead, assumes sentence structure

🎯 USE CASE:

10. Token Based Chunking

Splits text into chunks based on a specific token count

PYTHON EXAMPLE

```python
def token_based_chunking(text, token_limit=50):
    tokens = text.split() # Basic tokenization
    chunks = [' '.join(tokens[i:i+token_limit])
                for i in range(0, len(tokens),
token_limit)]
    return chunks

# Usage
chunks = token_based_chunking(text, token_limit=50)
```

✅ PROS

Precise token control, cost optimization

❌ CONS

May break sentences, requires proper tokenizer

🎯 USE CASE:

API token limits, cost optimization. Use tiktoken for production 💰

Which Method Should You Use?

METHOD	SPEED	SEMANTIC QUALITY	BEST FOR
Fixed	⚡⚡⚡	★★	Prototyping
Overlapping	⚡⚡⚡	★★★	RAG systems
Semantic	⚡⚡	★★★★	Chatbots
Recursive Char	⚡⚡	★★★	Token limits
Agentic	⚡	★★★★★	High-value doc
Advanced Semantic	⚡	★★★★★	Topic clusterin
Context Enriched	⚡⚡	★★★★	Q&A systems
Paragraph	⚡⚡⚡	★★★	Structured doc
Recursive Sentence	⚡⚡	★★★★	Narratives
Token Based	⚡⚡⚡	★★	API optimizati

Chunking Best Practices

1️⃣

Test multiple methods

No one-size-fits-all solution

2️⃣

Consider your use case

Q&A vs summarization needs differ

3️⃣

Optimal chunk size: 200-500 tokens

Sweet spot for most RAG systems

4️⃣

Add metadata

Include source, page numbers, timestamps

5️⃣

Measure retrieval quality

Track precision, recall, and relevance