

## Level 1 : Fixed Size Chunking

This is the most crude and simplest method of segmenting the text. It breaks down the text into chunks of a specified number of characters, regardless of their content or structure.

*Langchain and llamaindex* framework offer CharacterTextSplitter and SentenceSplitter (default to splitting on sentences) classes for this chunking technique. A few concepts to remember -

- **How the text is split:** by single character
- **How the chunk size is measured:** by number of characters
- **chunk\_size:** the number of characters in the chunks
- **chunk\_overlap:** the number of characters that are being overlap in sequential chunks. keep duplicate data across chunks
- **separator:** character(s) on which the text would be split on (default “”)

The RecursiveCharacterTextSplitter takes a large text and splits it based on a specified chunk size. It does this by using a set of characters. The default characters provided to it are ["\n\n", "\n", " ", ""].

It takes in the large text then tries to split it by the first character \n\n. If the first split by \n\n is still large then it moves to the next character which is \n and tries to split by it. If it is still larger than our specified chunk size it moves to the next character in the set until we get a split that is less than our specified chunk size.

Example Input:

Large Text:

mathematica

Copy code

```
Hello! My name is John.  
  
I love programming. I specialize in Python, JavaScript, and C++.  
  
In my free time, I enjoy hiking, reading books, and playing chess.
```

Chunk Size: 50 characters

Predefined Splitting Characters: ["\n\n", "\n", " ", ""]

Step-by-Step Process:

## 1. First Split by `\n\n` (Two Newlines)

- Split the text into **paragraphs** based on `\n\n`.

### Result:

nginx

[Copy code](#)

Chunk 1: "Hello! My name is John."

Chunk 2: "I love programming. I specialize in Python, JavaScript, and C++."

Chunk 3: "In my free time, I enjoy hiking, reading books, and playing chess."

- Check Chunk Sizes:**
  - Chunk 1: 25 characters (OK).
  - Chunk 2: 74 characters (**too large**).
  - Chunk 3: 62 characters (**too large**).

---

## 2. Next Split by `\n` (Single Newline)

- Apply this split to **Chunks 2 and 3** because they exceed the chunk size.

### Result:

nginx

[Copy code](#)

Chunk 1: "Hello! My name is John." (unchanged)

Chunk 2a: "I love programming. I specialize in Python,"

Chunk 2b: "JavaScript, and C++."

Chunk 3a: "In my free time, I enjoy hiking,"



```
Chunk 3a: "In my free time, I enjoy hiking,"  
Chunk 3b: "reading books, and playing chess."
```

[Copy code](#)

- **Check Chunk Sizes:**
    - Chunk 1: 25 characters (OK).
    - Chunk 2a: 48 characters (OK).
    - Chunk 2b: 23 characters (OK).
    - Chunk 3a: 37 characters (OK).
    - Chunk 3b: 42 characters (OK).
  - **Result:** All chunks are now under the chunk size. **No further splitting required for these chunks.**
- 

### 3. Third Split by (Space)

Let's assume the chunk size is now reduced to **25 characters** to ensure further splitting happens.

- Apply this split to **Chunks 2a, 3a, and 3b**, which are still above the new limit.

**Result:**


```
nginx
```

[Copy code](#)

```
Chunk 1: "Hello! My name is John." (unchanged)  
Chunk 2a1: "I love programming."  
Chunk 2a2: "I specialize in Python,"  
Chunk 2b: "JavaScript, and C++." (unchanged)  
Chunk 3a1: "In my free time, I"  
Chunk 3a2: "enjoy hiking,"  
Chunk 3b1: "reading books, and"
```



Chunk 3b1: "reading books, and"

 Copy code

Chunk 3b2: "playing chess."

- **Check Chunk Sizes:**

- Chunk 1: 25 characters (OK).
- Chunk 2a1: 21 characters (OK).
- Chunk 2a2: 25 characters (OK).
- Chunk 2b: 23 characters (OK).
- Chunk 3a1: 20 characters (OK).
- Chunk 3a2: 14 characters (OK).
- Chunk 3b1: 21 characters (OK).
- Chunk 3b2: 14 characters (OK).

---

#### 4. Final Split by `""` (Empty String)

Let's assume the chunk size is now reduced to **10 characters**. For this, each chunk exceeding the limit will be split into **individual characters**.

- Apply this split to all chunks larger than 10 characters.

#### Result:

nginx

 Copy code

Chunk 1: "Hello! My"

Chunk 1a: "name is"

Chunk 1b: "John."



## Semantic Chunks

Semantic chunking involves taking the embeddings of every sentence in the document, comparing the similarity of all sentences with each other, and then grouping sentences with the most similar embeddings together.

Top highlight

By focusing on the text's meaning and context, Semantic Chunking significantly enhances the quality of retrieval. It's a top-notch choice when maintaining the semantic integrity of the text is vital.

The hypothesis here is we can use embeddings of individual sentences to make more meaningful chunks. Basic idea is as follows :-

1. *Split the documents into sentences based on separators(.,?!,)*
2. *Index each sentence based on position.*
3. *Group: Choose how many sentences to be on either side. Add a buffer of sentences on either side of our selected sentence.*
4. *Calculate distance between group of sentences.*
5. *Merge groups based on similarity i.e. keep similar sentences together.*
6. *Split the sentences that are not similar.*

Now step-by-step (very clearly)

### 1 Split document into sentences

```
nginx
```

 Copy code

Sentence 1

Sentence 2

Sentence 3

Sentence 4

Sentence 5

**✗ No embedding model here**

This is just punctuation-based splitting ( . ? ! ).

---

## 2 Index each sentence based on position

 Copy code

0 → Sentence 1

1 → Sentence 2

2 → Sentence 3

✗ No embedding model here

Just numbering sentences.

## 3 Add `buffer_size` sentences on either side

Example: `buffer_size = 1`

For sentence at index 2:

csharp

 Copy code

[Sentence 1, Sentence 2, Sentence 3]

✗ Still no embedding model

This just creates sentence groups (windows).

## 🔥 4 Calculate distances between groups of sentences

🔥 THIS IS WHERE EMBEDDING MODEL IS USED

Here's what actually happens:

a) Each group of sentences is sent to the embedding model

Example groups:

```
less
```

[Copy code](#)

Group A: [Sentence 1, Sentence 2, Sentence 3]

Group B: [Sentence 2, Sentence 3, Sentence 4]

Embedding model converts them into vectors:

```
css
```

[Copy code](#)

Group A → [0.12, 0.88, -0.31, ...]

Group B → [0.10, 0.85, -0.29, ...]

📌 This step cannot happen without an embedding model

## b) Distance is calculated between vectors

Usually:

- cosine distance
- or cosine similarity

Example:

powershell

 Copy code

```
distance(Group A, Group B) = 0.82
```

✓ Now we have **numeric differences** between sentence groups.

---

## Merge or split based on thresholds

Now we apply:

- percentile
- standard deviation
- interquartile

```
sql
```

[Copy code](#)

```
If distance > threshold → SPLIT
```

```
Else → MERGE
```

✗ No embedding model here

This is just decision logic using numbers.

## One visual mental model 🧠

```
sql
```

[Copy code](#)

```
Text
```

```
↓
```

```
Sentence splitting
```

✗ embeddings

```
↓
```

```
Buffer grouping
```

✗ embeddings

```
↓
```

```
EMBEDDING MODEL
```

✓ embeddings USED HERE

```
↓
```

```
Distance calculation
```

✓ embeddings USED HERE

```
↓
```

```
Threshold logic
```

✗ embeddings

```
↓
```

```
Final chunks
```



# 1 Answer Relevancy

Get Plus x

? *Did the model actually answer the question?*

👉 Checks question ↔ answer

## Example

Question:

What are the symptoms of diabetes?

Answer A:

Diabetes symptoms include frequent urination, fatigue, and thirst.

✅ High answer relevancy

Answer B:

Diabetes is a chronic disease affecting insulin.

❌ Low answer relevancy (factually related, but not answering *symptoms*)

✂️ Even if the answer is correct, if it **doesn't answer the question**, relevancy is low.

One-line meaning:

"Did the answer stay on topic?"



## 2 Faithfulness

? *Did the model hallucinate anything?*

👉 Checks answer ↔ retrieved context

### Example

Context retrieved:

Diabetes symptoms include thirst and fatigue.

Answer:

Symptoms include thirst, fatigue, and kidney failure.

✗ Not faithful

(kidney failure was **not** in the context)

✦ The answer may be *medically true*, but **RAG faithfulness cares only about the given context**.

One-line meaning:

"Is every part of the answer supported by the context?"

### 3 Context Recall

? *Did retrieval bring ALL the important info?*

👉 Checks ground truth ↔ retrieved context

#### Example

Correct answer should include:

- thirst
- fatigue
- frequent urination

Retrieved context contains:

- thirst
- fatigue

✗ **Low recall** (missing frequent urination)

✦ This metric **does NOT blame the LLM**

It blames the **retriever**.

One-line meaning:

"Did we retrieve enough information to answer fully?"



## 4 Context Precision

? *Did retrieval bring ONLY useful info?*

👉 Checks retrieved context ↔ question

### Example

Question:

Symptoms of diabetes

Retrieved context:

- diabetes symptoms ✓
- insulin manufacturing ✗
- history of insulin ✗

✗ **Low precision** (too much irrelevant junk)

📌 Even if recall is high, **precision can be low**.

One-line meaning:

"How much of the retrieved context was actually useful?"

## One diagram in words

pgsql

 Copy code

Question



Retriever → Context



Context Recall (enough info?)

Context Precision (only useful info?)



LLM



Faithfulness (no hallucination?)

Answer Relevancy (answered the question?)