Chunking Strategies for RAG
A Comprehensive Guide

Presented by Vizuara AI Labs

August 2025

Contents

1 Introduction

Large Language Models (LLMs) have revolutionized natural language processing tasks, but they face a fundamental limitation: a fixed context window. This means that an LLM can only process a limited amount of text at once. In **Retrieval-Augmented Generation (RAG)** systems, we overcome this limitation by retrieving relevant pieces of text (*contexts*) from an external knowledge base and appending them to the user query. The quality of a RAG system's output heavily depends on how the knowledge base documents are broken down into these retrievable pieces. This process of breaking a large document into smaller pieces is known as **chunking**.

Chunking is crucial for several reasons:

- **Fit into Context Window:** Chunks must be small enough to fit into the LLM's input alongside the user query. If chunks are too large, important information may be omitted due to context length limits.
- **Efficient Retrieval:** When documents are chunked properly, the retrieval system (e.g., vector search) can more accurately match queries with relevant chunks. Good chunking improves retrieval precision and recall.
- **Semantic Coherence:** Chunks should ideally represent coherent pieces of information. Poorly designed chunks might split an idea across pieces, forcing the LLM to assemble answers from partial information and potentially leading to confusion or errors.
- **Storage and Speed:** The number and size of chunks affect storage (in a vector database) and retrieval speed. Smaller chunks mean more vectors to store and search through; larger chunks mean fewer vectors but risk lower precision. Thus, there is a trade-off.

There is no one-size-fits-all chunking approach. The optimal strategy depends on the nature of the documents, the queries expected, and computational considerations. In this guide, we will explore five major chunking strategies for RAG:

1. **Fixed-Size Chunking**
2. **Semantic Chunking**
3. **Recursive Chunking**
4. **Document Structure-Based Chunking**
5. **LLM-Based Chunking**

For each strategy, we will explain how it works, discuss its advantages and disadvantages, and provide detailed examples in various domains (legal, finance, healthcare, education, etc.) to illustrate when to apply each strategy. We will then compare the strategies and offer guidance on choosing the right approach for different scenarios.

2 Fixed-Size Chunking

Fixed-Size Chunking is the most straightforward chunking strategy. In this approach, a document is divided into uniform chunks based on a predetermined size criterion, such as a fixed number of characters, words, or tokens. For example, you might decide to split text into chunks of 500 tokens each.

The simplicity of fixed-size chunking makes it easy to implement and understand. All chunks will have roughly the same length, which can simplify batch processing and parallelization during embedding or retrieval. However, because the splitting is oblivious to the content, it can disrupt the flow of ideas.

A common enhancement to fixed-size chunking is to use a **sliding window** with overlap. Instead of splitting the text into disjoint back-to-back chunks, we allow consecutive chunks to overlap by a certain number of tokens. This overlap (often 10–30% of the chunk size) is included to ensure that if an important sentence or concept is cut at the boundary of one chunk, the overlapping part in the next chunk retains that context. Despite this improvement, fixed-size chunking may still break sentences or split related information across chunks if the fixed boundaries occur in unfortunate positions.

How it works: Suppose we choose a chunk size of N tokens (e.g., 500 tokens) with an overlap of M tokens (e.g., 50 tokens). Starting from the beginning of the document, we take the first N tokens as the first chunk. The next chunk will start $N - M$ tokens into the document (so that it overlaps the previous chunk by M tokens) and again span N tokens, and so on. This continues until the end of the document is reached. If the document length is not exactly divisible into N-token chunks, the last chunk will contain the remaining tokens (which might be shorter than N).

Advantages:

- **Simplicity:** Very easy to implement. One can create fixed-size chunks with just a few lines of code or using standard text-splitting utilities.
- **Uniform Size:** Since all chunks have similar length, this ensures none exceeds the LLM's input limit. It also means each chunk contributes roughly equally sized embeddings, and processing can be optimized (no extremely large or tiny chunks).
- **Fast Processing:** No complex analysis required – computationally light. Splitting by tokens/characters is faster than strategies that require semantic analysis or model inference.

Disadvantages:

- **Semantic Breaks:** Chunks are cut without regard to content. This can split sentences or closely related paragraphs. Important context might be split between chunks, requiring the retrieval of multiple chunks to get the full context.
- **Redundancy:** Overlap helps preserve context but also means repeated content across chunks. This redundancy increases storage and can confuse the LLM if overlapping chunks are retrieved together (the same text may appear twice).
- **Irrelevant Text:** If a chunk boundary falls in the middle of a topic, each chunk will contain some material that is only partial information for that topic, possibly making each chunk less self-contained.

Example: Fixed-size Chunking a Legal Document

Imagine a lengthy legal contract (say 50 pages long) that needs to be ingested into a RAG system for question-answering. We decide to use fixed-size chunking with a chunk size of 1000 characters (approximately 200-250 words) and an overlap of 200 characters to ensure continuity.
As we chunk the contract, each chunk might arbitrarily cut off at 1000 characters. In one instance, suppose a clause in the contract reads:

"...The Tenant shall be responsible for all maintenance costs, including repairs, utilities, and general upkeep of the property during the lease term, provided that..."
If one chunk ends in the middle of this sentence (e.g., after "during the lease term,") the remainder of the clause ("provided that...") will appear at the start of the next chunk. The overlapping 200 characters included at the beginning of the next chunk might capture the end of that sentence from the previous chunk, mitigating some context loss. However, because the split is mechanical, chunk 1 and chunk 2 each contain half of the clause and need to be considered together to get the full meaning.
For a user query about maintenance responsibilities, the retrieval system might fetch one of these chunks based on keywords. If it only retrieves chunk 1 or chunk 2, it might miss crucial details that were cut off. In this scenario, fixed-size chunking has sliced through a single legal provision, which could lead to an incomplete answer unless the system is tuned to retrieve both overlapping chunks.
This example illustrates how fixed-size chunking can disrupt semantic continuity in a legal document. In practice, one might increase the overlap or adjust chunk size to better align with clause boundaries, but without semantic awareness, perfect alignment is not guaranteed.

This strategy is best used in scenarios where documents are very large and numerous, and a quick, uniform segmentation is needed without requiring deep understanding of the content. For instance, if you are processing millions of web pages for indexing and can tolerate some loss of coherence in chunks, fixed-size chunking is a viable approach. It ensures every chunk is within limits and is extremely fast to compute. However, if the exact boundaries of ideas are important (as in most question-answering tasks), you will often find fixed-size chunking to be suboptimal without additional measures.

3 Semantic Chunking

Semantic Chunking segments a document based on meaning and content, rather than a strict token count. The idea is to keep each chunk focused on a coherent topic or idea. This usually involves using embeddings or other semantic similarity measures to determine where one idea ends and another begins.

How it works: The document is first split into natural units such as sentences or paragraphs. Next, these units are compared using a semantic similarity metric (for example, cosine similarity between sentence embeddings). We start accumulating sentences/paragraphs into a chunk as long as they remain highly related to each other. When the similarity between the current unit and the previous content drops below a chosen threshold, we start a new chunk at that boundary. In essence, consecutive segments that "talk about" the same subject (as evidenced by high similarity) are merged into one chunk, and a new chunk is only begun when there's a significant shift in topic.

In practice, implementing semantic chunking might look like:

- Compute an embedding (vector representation) for each sentence or paragraph.
- Pick a similarity threshold (this might be determined through experimentation; e.g., a cosine similarity of 0.8).
- Iterate through the text in order. For each new sentence/paragraph, check its similarity with the last sentence of the current chunk (or maybe the chunk's overall embedding). If the

similarity is high (above threshold), treat it as a continuation of the same topic and include it in the current chunk. If it falls below the threshold, start a new chunk.

Semantic chunking has the benefit of preserving the natural flow of language. Each chunk ideally represents a complete thought or a unit of knowledge that stands on its own contextually.

Advantages:

- **Maintains Coherence:** Each chunk should contain a complete idea or topic, as it groups semantically similar content. This leads to chunks that are easier for an LLM to use effectively, since the chunk is internally consistent.
- **Improved Retrieval Quality:** Because chunks are topically coherent, the chances of retrieving an irrelevant chunk (or a chunk only partially relevant to the query) are reduced. When a query matches a chunk, it's more likely the chunk has the full context on that query's topic.
- **Flexible Chunk Sizes:** Chunks can naturally vary in length depending on content. A concept that spans many sentences will form a larger chunk; a quick point might form a small chunk. This adaptability can be more efficient than a rigid size.

Disadvantages:

- **Complexity and Compute:** Determining semantic similarity for every potential break point requires computing embeddings and comparing them, which is computationally heavier than a naive split. For large documents or corpora, this adds preprocessing time.
- **Threshold Sensitivity:** The quality of chunking depends on the similarity threshold (and the embedding model used). If the threshold is set too high, you might end up splitting often (even when the topic hasn't truly changed, leading to overly granular chunks). If too low, you might merge distinct topics into one chunk. The optimal threshold may differ by document or domain, so some tuning or adaptive logic is needed.
- **Inconsistent Chunk Sizes:** While chunks will be coherent, their lengths can vary widely. Some chunks might be very short if the document jumps topics frequently, which could lead to an explosion in number of chunks (and many tiny chunks might not be efficient to retrieve). Others might approach the maximum size limit if the document stays on one topic for a long time.

Example: Semantic Chunking in an Educational Text

Consider a chapter from a history textbook (education domain) discussing World War II. The chapter might not explicitly be broken into subsections, but it naturally flows through several topics: the rise of fascism, the outbreak of war, major battles, the home front, and the aftermath.

Using semantic chunking, we first split the chapter into paragraphs. We then compute embeddings for each paragraph to gauge their content similarity. At the beginning of the chapter, several consecutive paragraphs all discuss the **rise of fascism in the 1930s**. These paragraphs have high mutual similarity, so the algorithm groups them into one chunk (Chunk A) encompassing the entire discussion of pre-war political conditions.

Later in the chapter, the narrative shifts to **the outbreak of war in 1939**. At the paragraph where this shift happens, we observe the embedding similarity to the previous paragraph drops significantly (the context has changed from political buildup to military conflict). This triggers the start of a new chunk. The paragraphs detailing the invasion of Poland, the Allied response, and early battles are all semantically related and get grouped into Chunk B.

As the text moves to describe **major battles** (say, the Battle of Stalingrad, D-Day, etc.), each major section might become its own chunk or a set of chunks, depending on how distinct the topics are. If the textbook flows from one battle to the next with some transition, the algorithm may decide whether those are one continuous topic or multiple. For instance, paragraphs about the European front might stay in one chunk until the narrative moves to the Pacific front, at which point a drop in similarity signals a new chunk.

By the end of the process, each chunk of the chapter corresponds to a high-level subtopic of World War II, even though the original text didn't have explicit section breaks for those subtopics. If a student asks a question like "What were the causes of World War II?" the RAG system is likely to retrieve Chunk A (where the rise of fascism and pre-war causes are discussed) because that chunk is entirely about that topic. If another question asks "How did World War II end in the Pacific Theater?" the system might retrieve the chunk that covers the Pacific front and the end of the war there. Each retrieved chunk provides a full, coherent context on the specific question, thanks to semantic chunking.

This example from the education domain shows how semantic chunking can create meaningful segments from a continuous narrative. It ensures that even if the source text is long and covers many ideas, each piece we feed to the LLM is topically focused and useful on its own.

Semantic chunking is especially useful when documents are narrative or discussion-based and lack explicit structure, or when maintaining the integrity of ideas is more important than uniform chunk size. Domains like academic articles, reports, or any prose where topics blend into each other benefit from this approach. However, one must be prepared to invest in the embedding computations and possibly fine-tune parameters for different types of content.

4 Recursive Chunking

Recursive Chunking is a hybrid strategy that aims to leverage natural document structure first, and then apply fixed size limits if needed. The strategy is essentially: *split by larger logical units, then split again if those units are still too large.* This two-stage (or multi-stage) approach ensures that you respect inherent boundaries like paragraphs or sections, but you don't end up with any chunk that is beyond the size limit of your system.

How it works: In the first pass, you chunk the document using an inherent separator or higher-level unit. For example, you might first split a book into chapters, or a chapter into sections, or an article into paragraphs (depending on the granularity needed and what's available in the text). Now you have chunks that align with these natural boundaries. In the second pass, you look at each chunk from the first pass and check its size (in tokens/characters). If a chunk is larger than the maximum size you want (say your model can handle 1000 tokens and the chunk is 1500 tokens), then you further split that chunk. The further splitting could use a simpler method (like fixed-size or sentence-based splitting) on that large chunk. If the chunk is within the limit, you leave it as is. In effect, each initial chunk may recursively be broken down until all resulting chunks are within the desired size.

The process can be recursive in multiple levels: for instance, first split a book by chapters, then if a chapter is still too long, split the chapter by sub-sections or paragraphs, and if a paragraph is still too long, split it by sentences or fixed tokens, etc. Many frameworks call this approach things like `RecursiveCharacterTextSplitter` (as in LangChain library), where they attempt splitting by largest units and progressively use smaller units.

Advantages:

- **Respects Natural Structure:** By splitting along the document's own structure (when available), it maintains whole sections or paragraphs intact as much as possible. This tends to preserve context and meaning better than arbitrary splits.
- **Size Compliance:** The recursive second step guarantees no chunk is over the token limit. So you get the best of both worlds: semantically meaningful pieces that also fit the model.
- **Modularity:** You can adjust the strategy at each level of recursion. For example, first split by paragraphs, then within a paragraph use fixed-size with overlap if needed. This modular design can be tuned to the content type.

Disadvantages:

- **Implementation Complexity:** The algorithm is more involved. You need to implement multiple levels of splitting and possibly different methods for each level. There's overhead in coding and maintaining this logic, especially when handling edge cases (e.g., what if a "paragraph" is extremely long with no punctuation?).
- **Inconsistent Chunk Count:** Depending on content, some documents might produce many chunks (if they have one extremely long section that gets split repeatedly) while others produce few (if each section was short). This can make batching for embedding or retrieval uneven.
- **Computational Overhead:** While not as heavy as semantic or LLM chunking, there is still overhead in checking each chunk and potentially splitting it further. If not optimized, one could end up, for instance, computing embeddings or length counts for many intermediate segments repeatedly.

Example: Recursive Chunking a Medical Research Paper (Healthcare Domain)

Consider a lengthy medical research paper published in a journal. Such an article typically has clearly defined sections: Introduction, Methods, Results, Discussion, and Conclusion, possibly along with subsections. Let's say we are using a chunking approach for a system that answers questions about medical literature.
Using recursive chunking:

1. **First pass (structure-based):** We split the paper by its top-level sections. We obtain chunks:
   - Chunk 1 = Introduction (covers background and objective of the study)
   - Chunk 2 = Methods (which might be very long, detailing experiments, protocols, data collection)
   - Chunk 3 = Results (possibly contains text plus maybe tables/figures references)
   - Chunk 4 = Discussion (analysis of results)
   - Chunk 5 = Conclusion (usually short summary)

2. **Second pass (size check and split):** We examine each chunk's length in tokens. Suppose our maximum chunk size is 1200 tokens for the embedding model. We find:

   - Introduction is 800 tokens (fine, no further action).
   - Methods is 2500 tokens (too large).
   - Results is 1800 tokens (too large).
   - Discussion is 1300 tokens (slightly above limit).
   - Conclusion is 300 tokens (fine).

   For each section that is too large, we now split further:

   - Methods (Chunk 2) might be split into, say, Chunk 2a and 2b. How to split? Ideally, by a smaller logical unit within Methods – perhaps subheadings like "Participants", "Procedure", "Analysis". If the text has subheadings, we split by those. If not, we could split by paragraph or a fixed-size at a convenient boundary. After splitting, suppose Methods splits into two chunks of 1250 tokens each (with perhaps a slight overlap or by paragraph boundary).
   - Results (Chunk 3) at 1800 tokens might be split into 3a and 3b. Maybe the Results had two major parts (e.g., "Quantitative Results" and "Qualitative Results") that we can use as natural breakpoints. If not, we split by paragraph such that chunk 3a might cover the first half of results, and 3b the second half.
   - Discussion (Chunk 4) at 1300 tokens might be borderline. We could decide a policy: if only slightly over, perhaps allow it or just do a simple split. Let's choose to split it into two chunks to be safe: one for early discussion and one for later discussion, perhaps at a paragraph break.

After recursive splitting, our final set of chunks might be: Introduction (1 chunk), Methods (2 chunks), Results (2 chunks), Discussion (2 chunks), Conclusion (1 chunk) — totaling 7 chunks from the original 5 sections.
Each chunk aligns with a logical portion of the paper. If a question asks "What were the findings regarding the efficacy of the treatment?", the retrieval system will likely focus on the "Results" chunks (3a and 3b). These chunks contain complete descriptions of results without being arbitrarily cut off mid-sentence. Had we done pure fixed-size splitting, the results might have been chopped in the middle of a paragraph describing a crucial outcome. With recursive chunking, we preserved as much continuity as possible (splitting mainly at section or paragraph boundaries) while still ensuring chunks are not too large.
This example shows how in the healthcare domain (or any research-heavy domain), recursive chunking respects the structure of complex documents and only deviates from it when absolutely necessary. It reduces the chance of losing context and makes the retrieval outputs more interpretable (each retrieved chunk can be identified as, say, "part of Methods" or "part of Results"), which can also help end-users trust and verify the answers.

Recursive chunking is a robust strategy for documents that inherently have some structure. It's commonly used in parsing long texts like research papers, technical documentation, or lengthy reports where sections are meaningful. By combining structure-based and size-based approaches, it minimizes disruption to the content while still producing model-friendly chunks. The extra implementation effort often pays off in more coherent retrieved contexts.

5 Document Structure-Based Chunking

Document Structure-Based Chunking (sometimes simply called **structural chunking**) relies on the original organization of the document to guide the chunking process. Many documents have a hierarchy: chapters, sections, subsections, paragraphs, bullet points, etc. The idea is to use these explicit markers to carve the document into chunks that mirror the author's intended organization of content.

Unlike semantic chunking, which might merge or split content based on meaning, structure-based chunking follows the document's formatting cues. For example, each top-level section in a report might become a chunk, or each list item in a FAQ could be a chunk.

How it works: This strategy assumes the document has clear delimiters or markup:

- If it's a text with headings (like a markdown or HTML or Word document with headings), we can split on those heading markers. For instance, every time we see a `## Section` (in markdown) or a heading style, we start a new chunk.
- If it's an enumerated list or a legal document with sections like Section 1, Section 2, etc., we use those as chunk boundaries.
- If the structure is only paragraphs, we might consider each paragraph as a chunk. (Though usually combining a few paragraphs that belong to the same section might be better if the section is small.)
- In code or technical documents, one might chunk by logical units like classes or functions (each function becomes a chunk, for example).
- In slide decks or instructional materials, each slide or each bullet point might be treated as separate chunks if the content per point is dense.

The principle is that the document's author likely grouped ideas into sections and paragraphs intentionally. By respecting that, we maintain logical integrity.

Advantages:

- **Preserves Logical Sections:** Each chunk corresponds to a pre-defined section or unit, which often encapsulates a complete thought or set of related information (e.g., a section titled "Conclusion" in a report contains the whole conclusion).
- **Human-Interpretable Chunks:** Chunks align with how a human would navigate the document (by section titles, etc.). This can make it easier to trace an answer back to the source; if a chunk came from "Chapter 3", the user or developer knows where to look in the original document.
- **Simplicity when Structure Exists:** If a document is well-structured, implementing this strategy is straightforward—just split on those structural markers. It's almost like converting the table of contents or outline into separate pieces.

Disadvantages:

- **Dependency on Having Structure:** Many documents are not well-structured. Some might have no headings, inconsistent formatting, or be plain text. This strategy doesn't apply well there. For example, a raw text dump or a conversation transcript might have no obvious sections to split on.
- **Uneven Chunk Sizes:** Sections can vary drastically in length. One section might be a few lines (e.g., an abstract or summary), while another is dozens of pages (e.g., a main body). If chunks correspond exactly to sections, some chunks might be too large to handle directly. (A mitigation is to combine this with recursive splitting as discussed earlier.)
- **Potentially Too Coarse:** If sections themselves contain multiple distinct ideas, this method won't separate them. For instance, a single section in a textbook might discuss two different subtopics, but if there's no subsection break, structure-based chunking will keep them together, possibly making the chunk less focused than ideal for retrieval.

Example: Structure-Based Chunking a Financial Report (Finance Domain)

Consider an **Annual Financial Report** of a corporation, which typically has a clear structure:

- Letter to Shareholders
- Introduction
- Company Overview
- Financial Statements (with sub-sections: Income Statement, Balance Sheet, Cash Flow)
- Notes to the Financial Statements
- Conclusion and Outlook

Using document structure-based chunking, we will split the report into chunks exactly at these section boundaries:

- Chunk A: Letter to Shareholders
- Chunk B: Introduction
- Chunk C: Company Overview
- Chunk D: Financial Statements (which might be very large; if needed, we could further split by its sub-sections: D1: Income Statement, D2: Balance Sheet, D3: Cash Flow)
- Chunk E: Notes to the Financial Statements (also potentially very large; could be split into smaller chunks per note or page if needed)
- Chunk F: Conclusion and Outlook

Each chunk corresponds to a section that likely spans several pages. For example, the "Notes to the Financial Statements" section might be 20 pages long, containing detailed explanations. In pure structure-based chunking, Chunk E would be that entire 20-page section. This chunk preserves the complete context of all notes, which is great for coherence, but

it is clearly too lengthy to feed into an LLM in one go. In practice, we would likely combine structure-based chunking with a secondary split for such large sections (enter recursive chunking). Perhaps we treat each individual note in the notes section as a chunk on its own, or we break the section every few pages.
Now, if a user asks, ”What does the financial outlook look like for next year?”, the system would identify that this question relates to the ”Conclusion and Outlook” part of the report and retrieve Chunk F in its entirety. Because we chunked the document by structure, Chunk F contains exactly the content we need (the entire outlook section) with no extraneous information from other sections. The model sees the full context of the company's outlook in that chunk and can answer comprehensively.
However, if a user asks a very detailed question, such as ”What are the details regarding the company's lease obligations mentioned in the notes?”, that information might be buried in the middle of the Notes section (Chunk E). If we kept Chunk E whole, retrieving it would give the model a massive chunk where the relevant detail might be only a small part. This is where we might notice the limitation of a pure structure-based approach. The solution would be to have applied recursive splitting to that section, or the system's retriever might need to pinpoint the exact part of the chunk.
This example illustrates that structure-based chunking works excellently when the structure aligns with how queries are asked (broad questions about sections), but can be less efficient for pinpoint queries inside a large section. It also shows why combining this strategy with others is often necessary in practice. Nonetheless, starting with structure is often a logical first step, especially for well-formatted documents like corporate reports, academic papers, legal codes, and manuals, where headings and sections are clearly defined.

In summary, document structure-based chunking should be your default when dealing with documents that come with an obvious outline. It keeps content organized and meaningful. Just be aware of its limitations and be prepared to introduce secondary chunking methods for sections that turn out very large.

6 LLM-Based Chunking

LLM-Based Chunking leverages the power of a Language Model itself to perform the chunking. Instead of following a heuristic (like fixed length or looking for headings) or using embeddings post-hoc, we directly ask an LLM to determine how to split the text into meaningful pieces.

This approach can be considered a form of *intelligent* or *agentic* chunking, because the LLM can, in principle, understand the content at a deeper level and make chunking decisions that a simple algorithm might miss. For instance, an LLM could decide that a certain narrative should be split right before the story's climax for best understanding, or that a list of items is best chunked as individual items despite no headings, purely because it ”reads” the content and infers logical groupings.

How it works: There are a few ways to implement LLM-based chunking:

- **Prompting the LLM for Chunk Boundaries:** You feed the LLM a large chunk of text (within what it can handle, possibly the whole document if it fits or in parts) along with a prompt like: ”Divide the above text into coherent sections, each focusing on a single main idea. Provide the starting and ending sentence of each section.” The LLM then outputs something like: ”Section 1: ... Section 2: ... etc.” which gives you the chunk boundaries.

- **Iterative LLM Splitting:** Another approach is iterative. Ask the LLM, "Does this text naturally break into smaller parts? If so, where is a good place to break it?" If the text is too large to input all at once, you might feed it one portion at a time and have it decide if a break should occur or not.

- **LLM as a Classifier for Similarity:** This is a variant where instead of using vector similarity, you ask the LLM directly: "Are these two paragraphs about the same topic?" to decide if they belong in one chunk or separate chunks. The LLM's answer guides the chunking (this is conceptually similar to semantic chunking, but using the LLM's internal knowledge rather than static embeddings).

LLM-based chunking ensures high semantic relevance in chunks, as the LLM can capture context, nuance, and even implicit connections. For example, an LLM might notice that a document has a subtle topic shift even if keywords remain similar, and decide to chunk there – something a raw embedding similarity might miss if the shift is nuanced.

Advantages:

- **High Semantic Accuracy:** The chunks produced are very likely to be internally coherent and meaningful. The LLM can understand context, anaphora (pronoun references), and the overall narrative, making splits at points a human editor might also choose.

- **Adaptability:** The method can potentially adapt to any domain because the LLM (especially a strong one like GPT-4) has seen a lot of text and can apply general knowledge to structuring content. It might handle domain-specific chunking tasks (like a legal argument breakdown or a scientific article's logical flow) more gracefully than simple rules.

- **Handles Unstructured Text:** In cases of text that have no clear structure or are very free-form (e.g., transcripts of brainstorming sessions, long forum threads, collections of notes), an LLM can impose structure by identifying themes or shifts in discussion.

Disadvantages:

- **Computationally Expensive:** Using an LLM to analyze and split text is far more resource-intensive than the other methods. If you have thousands of documents, chunking each via an LLM could be cost-prohibitive and slow, especially if using a large model via an API.

- **Context Window Limitations:** The LLM can only process text up to its maximum input size at a time. If a document exceeds this, you might have to chunk it somehow just to feed parts to the LLM for chunking (which becomes a bit paradoxical). You may need to chunk very large documents in stages or use a sliding approach even for the LLM prompt.

- **Consistency and Determinism:** Different runs or slight changes in wording of the prompt might produce different chunking, since generative models have some randomness (unless you use temperature 0). Ensuring consistent results might require careful prompt engineering. Also, if you rely on an LLM, the process is less transparent – it's harder to explain exactly why a chunk boundary was chosen at a certain point.

Example: LLM-Based Chunking of Customer Support Logs (Industry Domain)

Imagine we have a large log of customer support chat transcripts from an e-commerce company. Each chat is a conversation where a customer might discuss multiple issues in one

session (e.g., first they talk about a delayed shipment, then they ask about a refund for a different order, later they inquire about product info). The conversations are somewhat unstructured and can shift topics whenever the customer or agent brings up something new. We want to chunk these logs so that each chunk is about a single issue or query, which would make it easier for an AI assistant to retrieve relevant past cases when a similar issue comes up.
If we use standard strategies:

- Fixed-size would cut every N lines of chat, likely mixing different issues or splitting one issue.
- Semantic could help if we design it to detect topic shifts via embedding similarity, but dialogues are tricky – the wording might be different but still related to one issue, or vice versa.
- Structure-based doesn't apply much because these are just dialogues with no headings or clear markers of topic change.

So we try an LLM-based approach: We prompt an LLM with one entire chat transcript (assuming it fits in context, or we might break a very long chat into halves) and ask: "Split this conversation into separate chunks, each containing one distinct issue or question the customer had. Label each chunk with a brief title of the issue." The LLM reads the conversation and outputs: - Chunk 1: Issue = Delayed Shipment (covering from greeting until that issue was resolved) - Chunk 2: Issue = Refund for Damaged Item (covering the part of the conversation dealing with a refund) - Chunk 3: Issue = Product Information Inquiry (covering the last part where the customer asked about product details)
Each chunk as returned by the LLM contains the lines of dialogue relevant to that issue. The boundaries were decided by understanding the conversation context, something that would be hard to do with simple rules. We might get a chunk that starts when the customer first mentions "I haven't received my package..." and ends when that thread concludes, even if the conversation continued.
Now, if a new customer asks "My package is delayed, what should I do?" our system can retrieve Chunk 1 from this chat (among possibly others) because that chunk is specifically about delayed shipments. The chunk is self-contained and coherent – it has the entire exchange about delayed shipment – because the LLM smartly separated it from the other topics.
This example demonstrates the power of LLM chunking on unstructured, multi-topic text. In a customer support knowledge base scenario, having such fine-grained, issue-specific chunks can greatly improve the relevance of search results. The cost we paid was invoking the LLM with a complex prompt for each conversation, but perhaps this is justified if the improved retrieval saves agent time or improves automated responses.
Another scenario: consider a very long legislative document with archaic structure or a novel. We could similarly prompt an LLM: "Summarize this text into an outline with chapters or sections." The LLM essentially does chunking by summarizing and outlining, which can then guide how we split the text. This is useful in the legal domain where documents might be semi-structured but require domain understanding to chunk logically. For instance, an LLM could identify that a 100-page law text naturally divides into themes (even if the formatting

doesn't clearly show it) like "Definitions", "Scope", "Responsibilities", "Penalties", etc., and suggest chunk breaks accordingly.
These tasks highlight that LLM-based chunking can achieve what manual rules might miss – but they should be reserved for cases where such accuracy is needed and worth the computational cost.

LLM-based chunking is a cutting-edge approach and not always necessary. It shines when dealing with very complex or unstructured content where human-like understanding is required to decide chunk boundaries. In many routine cases, simpler methods suffice. But as LLMs become more accessible, we might see a rise in their use for preprocessing tasks like this, trading off compute for improved quality.

7 When to Use Which Chunking Strategy

Each chunking strategy has its ideal use cases and contexts where it outperforms the others. Below, we outline guidance on choosing a strategy, with practical scenarios and examples for each:

- **Use Fixed-Size Chunking when simplicity and speed matter more than perfect coherence.** If you're dealing with an extremely large number of documents (e.g., millions of pages of web crawl data or log files) and need a fast, uniform approach, fixed-size chunking is appropriate. For instance, a tech company might ingest server logs into a retrieval system to diagnose issues. The logs are essentially unstructured and enormous; using a simple fixed-size (say 1000-line) chunk might be sufficient because any single log line or small group of lines isn't semantically rich anyway, and the goal is to retrieve a region of a log. The loss of semantic precision is acceptable in exchange for speed and simplicity. **In summary:** Use fixed-size for homogenous or very large datasets where content boundaries are less crucial or not easily defined, and when you need to keep the pipeline very efficient.

- **Use Semantic Chunking when topic integrity is crucial.** If the documents are such that ideas flow into each other, and you want each chunk to represent a complete thought, semantic chunking is often the best choice. For example, in a *legal case analysis* scenario, you might have long narratives of court proceedings or judgments. These might not have clear sections, but different paragraphs discuss different arguments or evidence. Semantic chunking would keep each legal argument together in one chunk. Another example is an *academic literature review* article: the article might seamlessly transition from one theme to another; semantic chunking can separate the literature review into thematic chunks. **In summary:** Use semantic chunking for narrative, essay-style, or analytical documents (legal, academic, editorial, etc.) where maintaining the context of an argument or topic yields better QA results.

- **Use Recursive Chunking for documents with some structure, when you need to ensure no chunk is too large.** This is often the case for technical documents, manuals, or long reports. For example, an *employee handbook* (in an education or corporate domain) might have chapters and sections. You should chunk by those sections to keep content logically grouped. But if one section (say "Employee Benefits") is extremely long, recursive chunking kicks in to split that section into smaller chunks (maybe by subtopics like health insurance, retirement plans, etc.). Similarly, *software documentation* often has structure (by API sections or guide chapters); chunk by those first, then split further if a section is still huge. **In summary:** Use recursive chunking when documents have a usable structure, but you also need to enforce a hard size limit. It offers a balanced approach and is common in practice (e.g., many RAG pipelines default to something like splitting by paragraph, then splitting paragraphs longer than X tokens).

- **Use Document Structure-Based Chunking when the document is well-structured *and* you expect queries to align with that structure.** If users are likely to ask questions like "What's in section 2.3 of the report?" or "Explain the results in the experiment," it makes sense to chunk along those lines. As a concrete scenario, consider a *FAQ document in a company knowledge base*: each question-answer pair is probably a natural chunk. Using structure-based chunking (splitting at each Q/A heading) will yield perfect chunks for retrieval (one per question). Another scenario: *legal codes and regulations* that are numbered (Section 1, Section 1.1, etc.) – each section can be a chunk, because questions often reference a specific section of law. However, remain cautious: if sections are lengthy, you might still apply recursion. **In summary:** Use structure-based chunking for well-organized documents (guides, laws, manuals, reports, books) to take advantage of the author's organization. Combine with other methods if those sections vary widely in size.

- **Use LLM-Based Chunking for highly unstructured or critical documents where other methods fail to capture logical segments.** Due to cost, reserve this for smaller sets of documents or one-time preprocessing of important texts. For example, a *company might have a large collection of customer feedback emails* that they want to use in a RAG system to answer questions like "How do customers feel about product X features?". These emails may be informal, each email covering multiple points. An LLM could chunk each email by sentiment or topic (e.g., separate chunks for "feature requests", "bug complaints", "praise", if they appear in one email). Or consider a *novel or screenplay* where you're doing RAG for creative analysis – an LLM can chunk the story into exposition, conflict, climax, etc., far beyond what an automatic rule might do. Also, in *multilingual or code-mixed documents*, an LLM might better segment content because it understands the languages, whereas simple algorithms might be thrown off by mixed content. **In summary:** Use LLM-based chunking when the content is complex, nuance-rich, or poorly formatted, and when you can afford the computation. It's the method of last resort if you find simpler strategies yield unsatisfactory results in maintaining context.

It is important to note that these strategies are not mutually exclusive. In practice, you might combine them. For instance, you might primarily do semantic chunking, but also enforce a maximum chunk size (a bit of fixed-size philosophy) to avoid any giant chunks. Or you might do structure-based chunking followed by a semantic check to see if some sections should actually be two chunks.

Examples of choosing strategies:

- *Legal Domain Example:* You're building a system to answer questions about a country's laws. Laws are typically divided into articles and sections. A sensible approach is structure-based chunking (each article as a chunk). If some articles are very long, apply recursive chunking (split by clauses or paragraphs within that article). Semantic chunking might not be necessary because the law text is already structured, but if you find that within one article there are distinct topics not separated by formatting, you could incorporate semantic splitting at those points. LLM-based chunking would likely be overkill here unless the law texts are very free-form (which is uncommon).

- *Finance Domain Example:* You have earnings call transcripts and want to do Q&A on them. An earnings call transcript has an initial presentation, then a Q&A section. It might not have clear headings for each question. A hybrid approach could work: use structure (split at the start of the Q&A section so presentation vs Q&A are separate chunks), then within the Q&A, consider semantic or LLM-based chunking to split by each question and its answer. Alternatively, feed each question-answer exchange to an LLM to isolate it into a chunk. Each chunk would then be one question and answer pair from the call, which aligns with likely queries ("What did the CEO say about revenue guidance?").

- *Healthcare Domain Example:* You are processing patient electronic health records (EHRs) for a clinical assistant. EHRs have semi-structured fields (e.g., "Chief Complaint", "History of Present Illness", "Lab Results", "Assessment and Plan"). A structure-based chunking makes sense: chunk by each section of the EHR. However, the "Assessment and Plan" section might be a long narrative containing multiple conditions and plans. Within that section, semantic chunking could be used to separate each medical issue's plan into its own chunk, if needed. Fixed-size would not be good here because it could cut off a plan mid-way; LLM-based might be useful if the notes are written in an inconsistent manner and you need the model to identify each problem discussed. But likely, a combination of structure and semantic works well in healthcare notes.

- *Education Domain Example:* Suppose you have a set of lecture transcripts or long video subtitles that you want to use in a QA system for students. Transcripts often have no explicit structure beyond timestamped lines. If the lecture stays on one topic for a while and then transitions, semantic chunking can catch those transitions and group sentences accordingly. If the lecture is extremely lengthy (like a 3-hour transcript), you may combine semantic chunking with a hard limit (no chunk bigger than, say, 5 minutes of speech or 750 tokens) to keep chunks manageable. LLM-based chunking could also be used here: you might prompt the LLM with the lecture transcript section by section to identify natural breakpoints (like "now the professor moves to the next concept"). But if semantic chunking with embeddings works, it will be cheaper and likely sufficient.

In summary, the choice of chunking strategy depends on:

- **Document Structure:** Use it if it exists (with possible tweaks).

- **Content Type:** Narrative vs list vs dialogue vs code will influence strategy.

- **Query Needs:** If queries are specific, you want finer, precise chunks (semantic/LLM). If queries are broad, larger chunks (structure or fixed) might suffice to cover more ground.

- **Scale and Resources:** At massive scale with limited compute, simpler chunking (fixed or light recursive) might be the only viable path. For smaller but crucial data, investing in semantic or LLM chunking can pay off with better performance.

Ultimately, chunking is often an iterative process—try a strategy, evaluate whether the retrieval results and answers are good, and adjust. Many RAG practitioners start with something like fixed-size or simple paragraph splitting to get off the ground, and then refine the chunking method as they observe failures (like an answer missing context because it was split across chunks). Being aware of these strategies and their trade-offs allows one to make informed adjustments.

8 Additional Strategies and Future Directions

Beyond the five core strategies we've detailed, there are other chunking techniques and hybrid approaches that have been explored or are emerging. While we won't go in-depth on these, it's worth mentioning a few to spur further thought:

- **Sliding Window vs. Disjoint Chunks:** We touched on this in fixed-size chunking, but it's a general consideration. A sliding window (with overlap) can be applied to any chunking that might leave out context, not just fixed-size. For instance, one could use a sliding window on semantic chunks as well: if you worry that a semantic chunk boundary might still cut off a half-finished point, you could overlap chunks slightly around the boundary. This hybrid ensures continuity but increases redundancy. Some practitioners treat the overlapping chunk generation as its own strategy, sometimes called *Rolling Chunking*.
- **Query-Directed Chunking (Dynamic Chunking):** Instead of pre-chunking all documents in a static way, some advanced systems consider forming chunks on-the-fly in response to a query. For example, given a specific question, the system might search the document for relevant sections and dynamically extract a span of text around those sections as a "chunk" to feed into the model. This is more akin to information retrieval or search augmentation than a chunking strategy, but it blurs the line. It's like not deciding chunks until you know what is being asked. This approach can minimize irrelevant text, but it's complex to implement and not a true offline chunking method.
- **Clustering and Reassembly:** One experimental idea is to break text into small units (like sentences) and use clustering algorithms to group sentences that belong together (based on embedding similarity or topic modeling). Each cluster then forms a chunk, possibly reordering sentences to original order within the chunk. This could, in theory, group related content that was not contiguous originally, which is unconventional for RAG (since typically we preserve original ordering). This might be useful in cases where documents are collections of facts that don't need order. However, for most use cases, keeping the original sequence is important, so this is rarely used.
- **Summarization-Based Splitting:** Instead of chunking and then retrieving, another approach is to create hierarchical summaries. For example, for a very large document, one could first ask an LLM to summarize each section into a shorter chunk (compressing it), possibly recursively summarizing summaries. While not exactly chunking, this method yields chunks that are not direct excerpts of the document but condensed forms. These can then be used in retrieval. The trade-off is losing detail for the sake of fitting more context. It is a strategy to consider when exact recall of text isn't needed, but general understanding is (like summarizing a book chapter by chapter for a QA system that doesn't need exact quotes). This can be seen as a two-step chunk: chunk then summarize each chunk to shrink it.
- **Multimodal-aware Chunking:** In some documents (like web pages or PDFs), images, tables, or figures might be embedded. A chunking strategy could consider these, ensuring that text referring to a figure is chunked along with either the figure caption or some description. This goes beyond pure text chunking but is practical: if a query is about a chart, you'd want the chunk to include the chart's description. This might mean chunking by content type boundaries (text vs figure vs table).

- **Learning-based Chunking:** There is research on using machine learning to learn optimal chunk boundaries, possibly by training a model on question-answer pairs to figure out how to chunk so that answers are contained wholly in single chunks. This might involve reinforcement learning or other techniques. It's not common yet in practice, but as RAG matures, we might see tools that automatically adjust chunking strategies based on feedback (like if the system keeps needing two chunks to answer a question, maybe it learns to chunk differently).
- **Domain-Specific Heuristics:** Some domains allow special chunking rules. For example, in code, one could chunk by function definitions (each function or class is a chunk) because questions about code often pertain to a specific function. In poetry, one might chunk by stanza. In news articles, maybe chunk by each article if concatenated, etc. Recognizing domain patterns can yield strategies outside the generic ones we listed, but they usually reduce to structure-based chunking (with the structure being domain-specific markers).

The field is evolving. As we gather more experience with RAG systems, new chunking techniques might emerge, especially driven by the dual needs of scaling to more data and tailoring to complex queries. Keeping an experimental mindset and being willing to adapt or combine strategies is key. Many successful systems end up using a mix: for instance, using structure where possible, semantic where necessary, and maybe an LLM for the last few troublesome cases.

9 Conclusion

Chunking is a foundational step in building effective Retrieval-Augmented Generation systems. The five major strategies we explored – fixed-size, semantic, recursive, structure-based, and LLM-based – each offer a different balance between simplicity, coherence, and computational cost. To recap in brief:

- Fixed-size chunking chops data into equal parts, which is easy and fast but can break context.
- Semantic chunking uses meaning to keep related content together, improving coherence at the cost of more processing.
- Recursive chunking blends structural cues with size limits to avoid both overlong chunks and unnatural breaks.
- Structure-based chunking trusts the document's own format to guide splits, preserving author-intended divisions.
- LLM-based chunking employs AI understanding to determine the best splits, yielding high-quality chunks but with significant cost.

Choosing the right strategy is all about context. What are the documents like? What questions will be asked? How much processing can we afford? By matching the chunking approach to the task requirements, we ensure that the LLM has the most relevant and well-formed context to work with, leading to better answers.

In practice, always test and iterate. Start with a reasonable strategy (or combination), then evaluate the system's outputs. If you find the LLM's answers are lacking information or pulling in irrelevant text, revisit your chunking. You might need to make chunks smaller (for precision) or larger (for completeness), or switch to a more advanced strategy for certain documents.

By understanding these strategies and their trade-offs, you are equipped to design the chunking process of a RAG pipeline that best serves your application's needs. As the AI field advances and models improve (or context windows grow larger), the optimal chunking approach may change, but the fundamental goal remains: provide the right information to the model in the right pieces. With the guidance and examples provided here, we hope you can achieve exactly that, crafting chunks that help your AI deliver accurate, context-rich, and trustworthy responses.