RAG Chunking Strategy

Retrieval-Augmented Generation (RAG) is an AI framework that combines information retrieval with text generation to produce more accurate and context-grounded outputs. Instead of relying solely on an LLM's fixed training data, a RAG system actively **retrieves relevant information** (e.g. from documents, databases, or the web) and provides it to the LLM at query time. By doing so, RAG enables large language models to incorporate up-to-date, domain-specific knowledge and back up their responses with factual references, greatly reducing hallucinations and increasing reliability. In this tutorial-style guide, we'll walk through the components of a modern RAG pipeline – with a special focus on *chunking* – explaining concepts, strategies, and code snippets along the way.

1. Introduction to Retrieval-Augmented Generation (RAG)

At its core, RAG marries a **retriever** module with a **generator** module. The retriever fetches external information in response to a user query, and the generator (usually a large language model) uses both the query and the retrieved data to compose an answer. This approach leverages the strengths of traditional search systems and modern LLMs together. A simple way to think about it: the retriever acts like an efficient open-book exam system for the LLM – it supplies the model with relevant text “notes” so that the model’s answer can be grounded in those notes rather than just its own internal memory.

Why use RAG? Incorporating retrieval confers several benefits. First, it allows LLMs to access **current or proprietary data** beyond their training cutoff, making responses more up-to-date and relevant. Second, it provides **factual grounding** for the generated text – the model isn’t guessing or hallucinating facts, but rather pulling from real sources. In practice, RAG systems often return not just an answer but also citations or snippets from the source material, so users can verify and trust the response. Finally, RAG can be more **cost-effective and flexible** than fine-tuning an LLM on a large custom corpus. By keeping a separate knowledge base that can be updated independently, we avoid retraining the model for every data update.

In summary, RAG enhances a general-purpose LLM with *targeted retrieval* to produce answers that are both **knowledgeable and verifiable**. Next, we’ll delve into how the retrieval component can be implemented in practice.

2. Strategies for Implementing Effective Retrieval

A RAG system’s retrieval component can take many forms, depending on the source and structure of the knowledge it needs. “Retrieval” here broadly means any method of **fetching relevant information** from external data sources to help answer the query. Some common retrieval strategies include:

- **Calling External Tools/APIs:** In some cases, the LLM can use a tool like a web search engine or a database query API to find information. For example, an LLM might call a Wikipedia search API or a company’s internal API to locate documents containing the answer. This approach treats the external system as the retriever (sometimes orchestrated via an agent or tool-use framework, or made more convenient via an MCP). It’s powerful for accessing live information (like current events or real-time data), but requires carefully formatted or templated API inputs.
- **Vector Database Retrieval (Semantic Search):** A very popular approach is to use a **vector search** over an embedded knowledge base. In this setup, documents or text passages are preprocessed into vector embeddings, and at query time the user query is also embedded and used to find semantically similar pieces of text. This allows retrieval based on meaning rather than exact keywords. Modern search engines and cloud services leverage vector databases to efficiently find relevant documents by cosine similarity in embedding space. We’ll explore this approach in depth in the next sections (since it’s central to RAG pipelines).

- **Keyword or Text Search (Lexical Retrieval):** Traditional information retrieval methods like keyword search (e.g. BM25 or SQL full-text search) can also be employed. Here the system might do a direct text match or regex search in the documents for terms related to the query. While lexical search can be very precise for matching specific terms, it may miss semantic matches (e.g. synonyms). Often, a *hybrid* retrieval is used – combining keyword search with vector search to get the best of both (catching exact matches and semantic matches). Many production systems use a hybrid strategy, first narrowing candidates via keywords then reranking by vector similarity, or even summing lexical and semantic relevance scores.
- **Graph-Based Retrieval:** If the knowledge is stored in a **knowledge graph** (nodes and relationships) rather than unstructured text, retrieval might involve graph queries or traversals. For instance, given a query about a relationship between entities, the system could traverse the knowledge graph to find a relevant subgraph of connected facts. Graph-based retrieval can capture structured relationships and is useful for complex multi-hop queries. Recent approaches known as "Graph RAG" dynamically construct a subgraph relevant to the query (by following edges between entities) and then convert that subgraph into text for the LLM. This can yield deeper, more contextual retrieval beyond what a single document contains. Graph-based methods are an emerging complement to text-based RAG for domains where data is highly structured.
- **Direct Text Extraction Methods:** In scenarios where the answer is hidden in a large document or a binary file (PDF, etc.), a retrieval strategy might involve **extracting** the relevant snippet from that source. For example, an LLM could be prompted to read a specific document and extract the answer (this blurs the line between retrieval and generation), or simpler, an algorithm might scan for specific keywords and pull out surrounding text. Often, text extraction is a preprocessing step – e.g. using OCR on images/PDFs or HTML parsing – to convert various data sources into searchable text for one of the above methods. We mention it here because a good RAG pipeline must handle *ingestion* of many formats (more on this next) so that later the retrieval can operate over clean text.

Each of these strategies can be used alone or in combination. For instance, a robust system might first query a knowledge graph to identify relevant entities, then use a vector search to retrieve passages about those entities, and finally have the LLM read those passages. The best choice depends on the nature of your data: structured vs unstructured, static vs dynamic, etc. In the next section, we focus on the prevalent **vector-embedding retrieval pipeline**, which underpins many RAG implementations today.

3. Overview of a Vector-Embedding Powered RAG Pipeline

One common architecture for RAG uses a **vector database** to store embeddings of your documents, enabling semantic search for relevant content. Let's break down how such a pipeline works end-to-end:

- **Document Ingestion (Offline Phase):** First, all your source documents (knowledge base articles, PDFs, transcripts, etc.) are **ingested** into the system. Ingestion involves loading the raw data from wherever it lives (files, databases, web URLs) and prepping it for search. Often, this means extracting the text from each document and cleaning it (removing irrelevant boilerplate, handling encoding issues, etc.). Tools like *document loaders* (e.g. in LangChain or LlamaIndex) can help interface with different data sources (Markdown, HTML, PDF, emails, etc.).
- **Text Preprocessing and Chunking:** After loading the raw text, each document is usually too large to feed directly into a language model or even an embedding model. This is where **chunking** (text splitting) comes in. The document text is broken into smaller segments called *chunks* that will serve as the units of retrieval. We'll discuss chunking strategies in great detail in subsequent sections. For now, understand that chunking is essential because both the embedding models and the LLM have input size limits. For example, many text embedding models have a maximum token limit (often 512 or 1024 tokens) and cannot embed an entire book in one go. So you might split a 10-page document into, say, 20 chunks of a few hundred words each. Each chunk ideally represents a semantically coherent piece of the document (e.g. a paragraph or section).
- **Vector Embedding:** Next, each chunk is converted into a numeric **embedding vector** using a pretrained model. An embedding is simply a high-dimensional vector (e.g. 384 or 768 dimensions) that captures the semantic meaning of the text. For instance, using OpenAI's embeddings or a SentenceTransformer model, the chunk *"Graphite is a form of carbon that conducts electricity."* might be transformed into a 768-dimensional vector of floats. All chunks are passed through the embedding model to produce their vector representations. This step yields a collection of vectors (with associated chunk IDs/metadata).
- **Indexing in a Vector Database:** The set of chunk embeddings is then indexed in a specialized **vector database** (or vector index). Vector DBs are designed to store and query millions or billions of vectors efficiently. They support fast **nearest neighbor search** – given a new vector (like one for a query), find the top *k* most similar vectors in the database. Popular vector stores include FAISS, Pinecone, Weaviate, Milvus, etc., which use algorithms (like HNSW graphs or IVF) to enable sub-linear search in high dimensions. Once your chunks are indexed, your knowledge base is essentially ready for semantic search.

- **Retrieval (Online Phase):** When a user poses a query (e.g. *"What are the conductivity properties of graphite?"*), the system first embeds the query text into the same vector space using the **same embedding model** used for documents. This yields a query vector in the same dimension as the chunk vectors. The vector database is then queried for the nearest neighbors to this query vector – these are the chunks whose content is most semantically similar to the query. The result is a set of top-ranked chunks, each likely containing information relevant to the question. Often we retrieve more than one chunk (e.g. top 3–5) to have enough context.
- **Optional Reranking or Filtering:** In practice, the raw top-*k* similarity results might be refined further. Some pipelines employ a **reranker** – for example, a smaller cross-attention model or even the LLM itself – to re-read the candidate chunks in light of the query and score which chunks truly answer the question best. This can correct cases where the nearest neighbor in vector space isn't actually the most relevant answer. Additionally, metadata filters may be applied (for instance, ensure all retrieved chunks come from a certain document source or are recent enough, etc., if that is important to the query).
- **Generation with Retrieved Context:** Finally, the retrieved chunks (now treated as contextual text) are concatenated with the original query and fed into the LLM. The prompt given to the LLM usually includes an instruction to *use the provided information* to answer the question and perhaps to cite the sources. For example, the prompt might look like: *"Using the information provided in the reference context, answer the user's question.\n### Reference Context:\n- Chunk 1: Graphite is a form of carbon...\n- Chunk 2: ...\n### Question: What are the conductivity properties of graphite?\n### Answer:"*. The LLM then generates an answer that incorporates facts from those chunks. Because the model sees the relevant source text, it can produce a specific, correct answer (e.g. describing graphite's electrical conductivity) and even include phrases from the source. The result is returned to the user, often with citations pointing back to the source chunks.

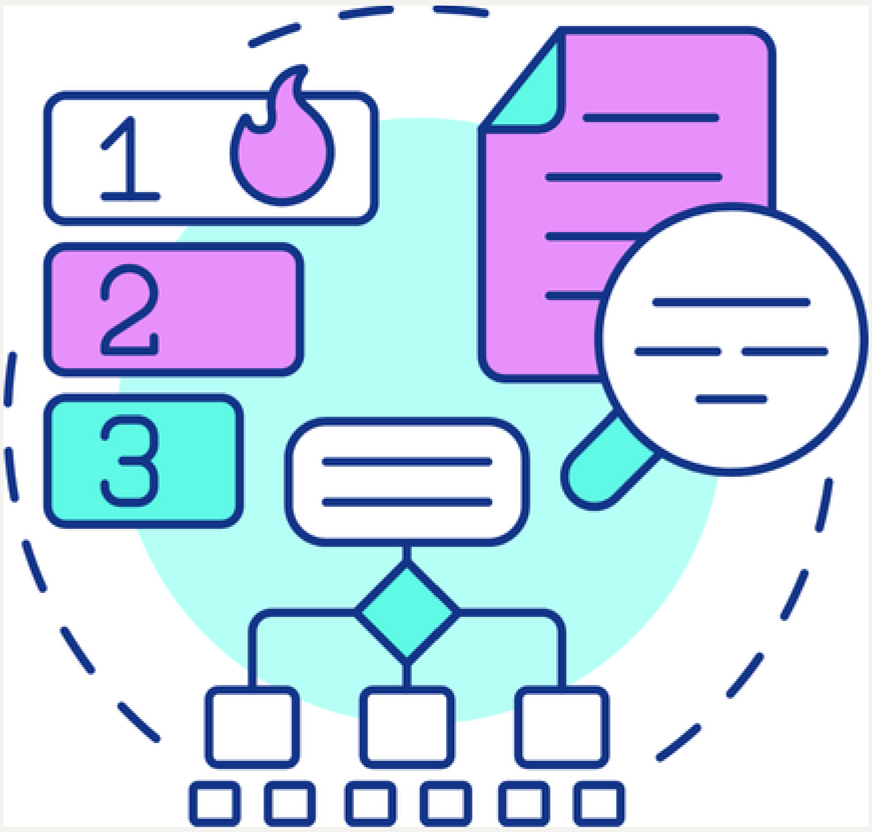
That covers the typical flow of a vector-based RAG system. To summarize: documents are **ingested and chunked**, chunks are **embedded and indexed**, and at query time relevant chunks are **retrieved** by vector similarity and fed into the LLM for **grounded generation**.

Code Example (Vector Search): To illustrate conceptually, imagine we already have a list of chunk vectors `docs_vectors` (NumPy array of shape `[num_chunks, dim]`) and a corresponding list `chunks_text` of the chunk strings. Given a new `query` string, we can do a simple cosine similarity search as follows:

```
import numpy as np

# Suppose embed_function is our embedding model (returns a vector
for text)
query_vec = embed_function(query)  # shape = (dim,)

# Compute cosine similarities between query_vec and all document
vectors
# cosine_sim = (A · B) / (||A|| * ||B||)
normed_docs = docs_vectors / np.linalg.norm(docs_vectors, axis=1,
keepdims=True)
normed_query = query_vec / np.linalg.norm(query_vec)
cosine_sims = np.dot(normed_docs, normed_query)  # vector of cos
sim scores

# Get top 3 highest similarity scores
top_idx = np.argpartition(cosine_sims, -3)[-3:]
top_idx = top_idx[np.argsort(cosine_sims[top_idx])[::-1]]  # sort
them by score

for idx in top_idx:
    print(f"Chunk {idx} (score={cosine_sims[idx]:.3f}):
{chunks_text[idx][:100]}...")
```

This example finds the top 3 chunk indices by cosine similarity to the query vector. In a real system, a vector DB or library would handle this more efficiently (especially for large datasets), but the logic is as shown. Once we have those top chunks, they would be passed into the prompt for the LLM to generate the final answer.

Now that we've covered the big picture, let's zoom in on the critical *document ingestion* stage, and in particular the art and science of **chunking**.



4. Document Ingestion: Text Extraction, Chunking, and Embedding

The ingestion pipeline is responsible for taking raw content and transforming it into the pieces that will be stored in our knowledge index. The major steps are:

- **Text Extraction:** Documents come in many formats – PDFs, HTML webpages, Word docs, spreadsheets, etc. The first task is to extract textual content from these sources. This may involve using libraries or APIs: for example, using an HTML parser to get visible text from a webpage, or an OCR tool for scanned images/PDFs, or specialized libraries (like Unstructured, Apache Tika, etc.) that handle many document types. The output of this step is typically plain text or a structured representation of the document's text. It's important to preserve as much *structure* as possible (paragraph breaks, headings, list items, table cells, etc.) during extraction, because that structure can inform how we chunk the text. Some ingestion frameworks produce structured outputs (e.g. a JSON with elements: title, sections, paragraphs, tables) which makes intelligent chunking easier.
- **Cleaning and Pre-Processing:** Before splitting text into chunks, we often do some cleaning: removing irrelevant content (navigation menus, ads), normalizing whitespace, maybe fixing encoding issues or removing very noisy text. This step is application-specific. The goal is to feed the chunker only the

meaningful content.

- **Chunking (Text Splitting):** Based on the chosen strategy, the cleaned text is broken into smaller segments. This could be as simple as splitting every *N* characters, or as complex as using an ML model to decide split points. We will discuss chunking strategies in detail in Section 6. The result of chunking is a list of text chunks (often strings), each usually accompanied by some metadata (like which document it came from, maybe a section title, etc.).
- **Vector Embedding:** Each chunk is then fed to an embedding model to produce a vector. This can be done immediately during ingestion (i.e. chunk and embed in one pipeline) or you can store the chunks first and embed in a separate step. Many RAG architectures perform embedding as part of ingestion so that at query time they don't have to run the embedding model on the documents, only on the query. It's crucial to use an embedding model suitable for your domain (for instance, code documents might use a code-specialized embedding model). Keep in mind the **token limit** of your embedding model – e.g., if it can handle up to 512 tokens, ensure your chunker made chunks smaller than that hard limit.
- **Storing Chunks and Metadata:** Alongside storing vectors in the vector index, you'll want to store metadata. Typically, each vector is stored with an ID that can retrieve the original chunk text and additional metadata (document ID, source name, creation date, etc.). Some vector DBs let you store a JSON blob or key–value pairs of metadata with each vector. This metadata is extremely useful later (we'll cover why in Section 8).

Now that we've covered how documents are ingested, let's focus on *chunking* – which turns out to be one of the most critical design choices in a RAG pipeline.

5. What is Chunking and Why It's Essential

Chunking in the context of RAG means dividing a large text into smaller pieces ("chunks") before indexing and retrieval. At first glance, chunking might seem trivial – just split the text and be done. But doing it *effectively* is both art and science, because it **profoundly** impacts the system's performance. Here are the main reasons chunking is essential:

- **Respecting Token Limits:** Language models have finite context windows, often a few hundred thousand tokens (latest ones like Llama 4 Scout supports up to 10 million, as of May 2025). Retrieved chunks will be inserted into the LLM's prompt, so their total length must stay within the model's input size. If you naively tried to feed an entire document that's longer than the LLM's context (say a 50-page report) into the prompt, you'd exceed the limit and get an error.

Chunking ensures we can *fit* the needed info into the LLM by selecting smaller pieces. Similarly, embedding models have input limits (e.g. 512 or 1024 tokens for many). If a text is longer than the embedder's max length, you *must* chunk it because the model simply cannot embed extremely long text as one vector. In short, chunking lets us handle corpora much larger than what the models can directly ingest at once.

- **Improving Retrieval Precision:** Even if we had an infinite context window, it's not ideal to embed or retrieve huge blobs of text. When an embedding model encodes a long passage into a single vector, it's essentially averaging or compressing a lot of information into one point. Important details can get "washed out." For example, if one chunk covers *multiple topics*, only some of which are relevant to a given query, the single vector might be somewhat similar to the query but not strongly – because the irrelevant parts of the chunk dilute the similarity. Conversely, if you use smaller, focused chunks, the ones that really match the query's topic will surface with higher similarity scores. In short, the **granularity of chunks affects retrieval accuracy**. Too large and you risk low precision (irrelevant stuff tagging along); too small and you risk missing context or splitting an answer across pieces that then might not be retrieved together. Finding the sweet spot is key. This sweet spot isn't universal either. It may change depending on the nature of your data and your use case.
- **Maintaining Knowledge Consistency:** Chunking should be done in a way that preserves meaningful units of information. If you cut a document arbitrarily, you might split a sentence in half or separate a figure from its caption or a question from its answer – leading to chunks that are not self-contained and thus not very useful. For the downstream LLM, each chunk should ideally represent a *consistent piece of knowledge* or a coherent thought. If the model only sees fragmented or incoherent text, it may struggle to use it effectively, or worse, generate something incorrect because the supporting text was incomplete. For example, splitting a markdown table mid-way could yield a chunk that has only the first two columns of the table and another chunk with the remaining columns – neither chunk alone fully makes sense, and an LLM might misinterpret such partial information. Thus, chunk boundaries should align with natural boundaries of the content (paragraphs, sections, list items, table rows, etc.) to keep each chunk's content coherent.
- **Localized Generation Context:** RAG pipelines usually feed the top retrieved chunks into the LLM. If those chunks are tight and relevant, the LLM's job is easier – it can focus on just the pertinent information. If the chunks are sprawling, the LLM may waste tokens and attention on content that isn't actually needed for the answer ("needle in a haystack" problem). By chunking and retrieving selectively, we *localize* the context given to the LLM, which tends to produce better-focused answers.

In summary, chunking addresses practical constraints (token limits of models) and optimizes both **precision** (by narrowing the text scope per chunk) and **recall/coverage** (by ensuring important content isn't missed due to being lumped with unrelated text). It also helps maintain **semantic coherence** within each chunk, which in turn leads to more consistent and interpretable results.

Roie Schwaber-Cohen from Pinecone nicely summarized why chunk size matters for retrieval: *"If the size of the content that you're embedding is wildly different from the size of the user's query, you're going to have a higher chance of getting a lower similarity score."*. In other words, a short query might not closely match a very long, content-heavy chunk because that chunk's embedding is an average of many concepts. Matching lengths and keeping chunks "right-sized" for likely queries improves the odds of a relevant hit.

Now that we know why we chunk, let's explore *how* to chunk effectively, surveying the state-of-the-art strategies.

6. Overview of Chunking Strategies (Sliding Windows, Semantic Splitting, etc.)

There is no one-size-fits-all way to chunk text. Researchers and engineers have developed several strategies, each with pros and cons. Here we outline some of the most common and advanced chunking techniques:

- **Fixed-Length Chunking:** The simplest method is to split text into equal-sized chunks (e.g. every 500 characters or every 256 tokens). This can be done with or without overlapping windows. Fixed-length chunking is easy to implement and works okay for uniform, homogeneous text like news articles or narrative paragraphs. However, it's "blind" to the content – it may cut off in the middle of a sentence or ignore logical boundaries. It also doesn't adapt to different formats (a code file vs a prose paragraph). Overlap can be added to fixed chunks (e.g. chunk 1 = tokens 1–256, chunk 2 = tokens 200–456, etc.) to help preserve context at boundaries. Overlapping sliding windows ensure that if an important fact falls near a cutoff, it might still appear in either the chunk before or after, reducing the chance of losing critical context. The downside is increased redundancy – overlapping chunks means storing and searching some content twice, which has storage and efficiency impacts.
- **Naively Variable-Length (Random or Heuristic) Chunking:** One might consider varying chunk sizes to better capture different sections or topics within a document. In a trivial form, even randomly sized chunks have been tried (though not often practical). The idea was that if your dataset has very diverse

documents, a fixed size might not suit all, so randomness could, in theory, avoid consistently mis-cutting in the same way. However, random chunking is generally not recommended – it can break sentences and create meaningless fragments. A more useful heuristic approach is *document-structure based variable chunks*: for example, make each top-level section in a report a chunk (so chunk sizes vary depending on section lengths). We'll cover structure-based chunking next.

- **Context-Aware (Structure-Based) Chunking:** This strategy uses the document's inherent structure and syntax to decide chunk boundaries. For example, split on paragraph breaks, sentence boundaries, or Markdown/HTML tags. By doing so, you ensure chunks align with natural divisions in content: each chunk might be a full sentence, a paragraph, or a list item, etc. A common approach is **recursive splitting by delimiters** – e.g. first try to split by two newlines (blank line, indicating paragraph), if the chunk is still too large, then split by single newline, if still too large then by sentence (`.`), and so on. This way, you only break text if it exceeds the chunk size limit, and you prefer to break at bigger boundaries (paragraphs before sentences, sentences before words). LangChain's `RecursiveCharacterTextSplitter` implements this idea: it takes a list of separators and tries them in order. For example, given separators `["\n\n", "\n", " ", ""]`, it will first split on blank lines; if any chunk is still over the limit, split those on single newline; if still too long, split on spaces, and as a last resort, split anywhere (even mid-word) to respect the max size. Context-aware chunking greatly reduces the chance of chopping a sentence in half or splitting a figure caption from the figure, etc., compared to raw fixed-length splitting. It's generally a good default strategy. The downside is a bit more computational overhead to analyze the text structure, and it may produce chunks of varying sizes – which, as we discussed, can be a mixed blessing. Still, maintaining logical units usually outweighs the downsides. Many implementations (like Unstructured or LlamaIndex) actually parse documents into elements (title, heading, paragraph, list, table) and use those as initial chunks. For instance, a whole table might become one chunk, ensuring it stays intact for the LLM to see it in full (since splitting a table could render it meaningless). Structure-based chunking works best when documents are well-formatted or you have a parser for their format.

- **Sliding Window Overlap:** This isn't a separate splitting rule, but an augmentation that can apply to any of the above. We touched on it for fixed-length, but you can also use overlapping windows with sentence-based chunks. For example, after splitting by sentences, you might group them such that each chunk shares one sentence with the next chunk (creating a overlap). This helps keep context continuity at chunk boundaries at the expense of duplication.

Overlap is particularly useful when using smaller chunk sizes, to mitigate the loss of context. It's common to configure, say, a 20% overlap in many pipelines.

- **Semantic Chunking (Embedding-based):** This advanced technique uses **semantic similarity to determine chunk boundaries**. The idea is to ensure each chunk is semantically coherent – sentences within a chunk should be topically related. One approach is: embed all sentences of a document individually, then use clustering or sliding windows with similarity checks to group sentences that are similar. For example, you could take an initial pass grouping every 3 sentences, compute embeddings, and if two adjacent groups are very similar, consider merging them (meaning they talk about the same subtopic). Conversely, if a single paragraph actually contains two very different ideas, a semantic splitter might decide to break it even if it's short, because the topics diverge. Another method: progressively add sentences to a chunk until adding the next sentence would make the chunk's embedding drift too far (indicating the next sentence is about a new topic). Semantic chunking, by focusing on meaning, can produce highly coherent chunks where each is about a distinct aspect or concept. Research has shown this can improve retrieval performance because each chunk is tightly aligned to a topic. However, it's much more computationally expensive – essentially you're doing extra embedding and clustering work during ingestion. Also, chunk sizes can end up uneven (some topics might be explained in one sentence, others need a long paragraph, and you'd chunk accordingly). Tools are emerging to support semantic splitting (e.g. LangChain's experimental `SemanticTextSplitter` uses sentence embeddings to merge similar sentences.
- **Adaptive or ML-Guided Chunking:** An even more dynamic approach is to use a machine learning model (even an LLM itself) to decide how to chunk. We can think of this as an "agent" or "guided" chunking. For instance, an LLM could be prompted to read a document and insert break markers at points where the topic changes or where a question might only need the preceding text. One could imagine an LLM-based chunker that says: *"This document has section 1… it ends here, section 2 starts… etc."* or even a model that decides to keep a code block unbroken. Adaptive chunking also includes learning-based methods that treat chunk boundary detection as a sequence labeling problem (train a model to label sentence boundaries as good breakpoints or not). These methods can in theory optimize chunking for your specific QA task (learning from data where answers came from which spans). The obvious trade-off is **complexity and cost** – involving ML in chunking means more computation and maintenance. For most cases, heuristic methods (like the ones above) are sufficient, but it's good to know this frontier exists.

- **Hybrid Chunking Strategies:** In practice, you might combine approaches to handle different content types or to get the best of each. For example, you might use structure-based chunking for the high-level split (split by chapters or sections), then within each section use a fixed token length with overlap to ensure no chunk exceeds 300 tokens. Or use semantic chunking for narrative text sections, but for code or lists, fall back to preserving structure as-is. A hybrid model could also refer to *context-enriched chunks*, an interesting idea where each chunk is augmented with a summary of the preceding chunk (so that even if split, the context is carried along). One study found that adding a brief summary of previous text to each chunk can help an LLM answer questions that span multiple chunks by giving it a bit of that context, without having to retrieve both chunks. This is like overlapping, but instead of raw text overlap, it's a distilled overlap.

As you can see, chunking can be done in myriad ways. The optimal strategy depends on your data and use case. Highly structured documents (like JSON, HTML, markdown) benefit from respecting the structure. Free-form text might benefit from semantic grouping. Code or mathematical text needs special handling (you wouldn't want to split a function in half). In many real-world RAG systems, a combination of strategies yields the best outcome. The key is that each chunk should ideally be **self-contained and topically coherent**, while also not exceeding model limits.

7. Determining the Optimal Chunk Size (Token Limits, Semantic Integrity, and Trade-offs)

How large should your chunks be? This is one of the first questions to tackle when designing a RAG pipeline, and the answer is often *"it depends."* There are a few considerations:

- **Model Context Limits:** As discussed, you have hard upper bounds. If your LLM context window is 4096 tokens and you plan to inject up to 4 chunks in the prompt plus the user query and some instructions, you might target chunks of roughly 800 tokens each (since 4×800 = 3200, leaving room for query and other prompt content). If using GPT-4 32k context and you only need 2 chunks, you could use bigger chunks, etc. Likewise for the embedding model – if it maxes at 512 tokens, that's an absolute max for chunk size. But you probably don't want to hit that max for every chunk. A good rule of thumb is to leave a margin (e.g. target 80% of the limit) to account for slight tokenization differences or additional metadata tokens. For example, if using a tokenizer that might count some Unicode characters in an unexpected way, chunking to exactly 512 tokens could occasionally produce a 513-token chunk due to a different count in the embedding model. So you might aim for ~480 to be safe.

- **Semantic Integrity:** We want each chunk to be a complete thought or unit. This often constrains the size *in practice*. For instance, if a section of text is 300 tokens and forms a coherent narrative, that's a good chunk. If a table is 1000 tokens, you either have to accept a large chunk or find a logical way to break it (maybe by splitting the table into two chunks but duplicating the header row in both, so each half is understandable). Sometimes the content itself dictates chunk size – e.g., a paragraph might be 150 tokens; you wouldn't split it into 2 chunks of 75 just to hit a target, because that breaks the semantics. On the other hand, if you have a 2000-token chapter with multiple subtopics, leaving it as one chunk would be too big; you'd look for subheadings or paragraph breaks to split it into maybe 4 chunks of ~500 tokens each. In short, **chunk at natural boundaries that keep the content self-contained**, which often yields variable sizes. You then might set an upper cap (say, no chunk > N tokens) and if something exceeds that, handle it specially (like split it into logical parts, or as last resort, cut it and note that it continues in next chunk).

- **Targeting Likely Query Scope:** Think about how users ask questions. If users typically ask about a specific concept that is usually explained within, say, a few sentences of your docs, you want chunks around that size so that the answer is likely to fall entirely inside one chunk. If chunks are too large, a single chunk might contain lots of unrelated info and only a small portion relevant to the question – which could reduce the similarity score. If chunks are too small (like one sentence each), a single answer might be split across two sentences in two chunks, requiring the retriever to pull both. As an example, if you have a FAQ with Q and A pairs, it's wise to keep each full Q&A together in one chunk. Splitting the question and answer into separate chunks would be counterproductive (the question chunk by itself isn't an answer, and the answer chunk without the question might be hard to interpret). So in that case, chunk size might equal the whole QA pair even if that's longer than average. In contrast, for a long technical article, maybe each section summary or paragraph can be a chunk. **Variable chunk sizes are okay** if they align with content – one study even suggests it's fine to not chunk at all for very short texts (each document as one chunk) and aggressively chunk very long texts, rather than treat everything uniformly.

- **Overlapping Context vs Redundancy:** If you choose a chunk size, consider if you will use overlap. With overlap, you can choose a slightly smaller base chunk size since each chunk gets a bit of extra from neighbors. Without overlap, you might go slightly bigger to avoid lost context. Overlap effectively increases the "apparent" chunk size from the model's perspective (because some content is repeated in adjacent chunks), but it also means you have more total chunks. For example, with a 200 token chunk and 20 token overlap, the model might see 220 tokens of context from that chunk sequence, with 20 being repeated in the next chunk as well.

- **Empirical Testing:** Often, you won't know the best chunk size without experimentation. Many practitioners start with a moderate size like **200-300 tokens** (roughly a few sentences or one long paragraph) as a **sensible starting point**. Then they evaluate: does retrieval find the right chunk when asked known questions? If not, try adjusting. If important info was split between two chunks frequently, maybe increase size or add overlap. If irrelevant info often comes along, maybe decrease size.
- **Embedding Vector Considerations:** Interestingly, the length of text can also affect the embedding distribution. There is some anecdotal evidence of an "embedding bias" where very short texts might produce disproportionately high similarity scores for trivial overlaps. For instance, an extremely short chunk like "Graphite is conductive." might closely match a query "graphite conductivity" because nearly the entire chunk matches the query terms – yielding a high cosine similarity. Meanwhile, a longer chunk that mentions graphite's structure and conductivity in a broader context might have a lower similarity because the extra words dilute the focus. This could lead the retriever to favor very short chunks that have keyword overlap, even if a longer chunk had more detailed info. On the flip side, very long chunks (as mentioned) can underperform because their embedding is an average that may not sharply align with any one query. The bias toward shorter chunks *generally* suggests that if you mix drastically different chunk sizes, shorter ones might surface more often. Mitigation strategies include: keep chunk sizes within a narrower range (don't have some chunks 50 tokens and others 1000 tokens; if documents are small, maybe just don't chunk them, if others are large, chunk those into moderate pieces); or use hybrid retrieval (if you suspect small chunks are gaming the similarity, incorporate a lexical score to ensure that longer chunk with the answer isn't overlooked). Another tactic is to **enrich embeddings** of shorter chunks by adding some context to them – for example, prepend the section title or a few keywords to the chunk text before embedding, to give it more substance. This can reduce false positives where a short chunk of 5 words might coincidentally match a query vector strongly.
- **Trade-offs and Task Needs:** Ultimately, the ideal chunk size might be different for different tasks. If the task is *very fine-grained QA* (finding a specific fact), smaller chunks might yield higher precision. If the task is *summarization* of a document, you might use bigger chunks or even no chunking (just feed the whole doc in pieces sequentially to the LLM). Also consider the **density of information**: a legal document might be dense with facts such that 100 tokens contains a lot of info, whereas a novel might need 100 tokens just to set context

with few facts. A good practice is to look at a few representative documents, manually experiment with chunk sizes (maybe manually chunk one document different ways), and imagine if a question about a certain detail would retrieve the necessary chunk.

In practice, many RAG pipelines have settled on chunk sizes roughly in the **200 to 500 token range** for typical use cases, with perhaps 15% overlap. This is not a hard rule but a common ballpark. That size tends to encapsulate a decent paragraph or two. For highly structured docs (like a documentation site), chunking by sections or bullet points might yield varying lengths, but often still a few hundred tokens each.

To illustrate a scenario: Suppose our documents are Markdown knowledge base articles with sections, code blocks, and tables. We might decide: *split by top-level section (H2 headings) first*. If a section is > 300 tokens, then split within it by paragraphs or subheadings. For any chunk that contains a table, if the table is small, keep it wholly in one chunk; if it's very large, maybe split the table but copy the header row into each chunk so they are independently understandable. Through such rules, we try to ensure each chunk is of manageable size and logically complete. We test with a few queries – does the retrieval bring the expected chunk? If not, adjust accordingly.

Remember that **iteration is normal** – chunking strategy is a tunable part of the system. There's even a notion of *dynamic chunking at query time*: e.g., instead of a static chunk DB, one could on-the-fly decide how to break relevant documents based on the query (though this is slower and less common). Usually, however, you tune chunking once and then rely on it.

8. Metadata Enrichment: Adding Source and Context to Chunks

When we split documents into chunks, we don't want to lose all the contextual information about where that chunk came from. **Metadata enrichment** is the practice of attaching additional data to each chunk that can help in retrieval and in the final answer presentation. Common metadata to store with each chunk includes:

- **Source Identifier:** e.g., the document or file name, URL, or an ID. This allows the system (and the end user) to trace an answer back to the document it came from. It's crucial for credibility – you can cite the source. And it enables filtering (maybe you only want to search within a certain document or prioritize certain sources).

- **Section or Hierarchy Info:** If the document had sections or headings, it's useful to store which section a chunk is from (e.g., "Introduction" vs "Conclusion" section). Some RAG applications use this at query time: if the user query specifically mentions a section, they can filter to chunks from that section. Or the UI can display "This answer is from *Document X*, section *Y*."
- **Position or Order:** You might store the page number (for PDF) or the character offset or a chunk index (1st chunk, 2nd chunk, ...) from the doc. This can be used if you need to fetch additional surrounding text later, or simply to sort and display passages in original order if returning multiple snippets.
- **Content Type Tags:** If you know the chunk is a figure caption, or a code snippet, or a table, etc., you might tag it as such. This can help the LLM format the answer (maybe it won't try to wordy-paraphrase code if it sees it's code; or if the user asks for an image description, you retrieve chunks tagged as image captions).
- **Embedded Metadata:** Sometimes the chunk text itself can carry context if kept in a certain format. For example, keeping **Markdown formatting** in the chunk text can be a form of metadata: the model will see if something was a bullet list versus plain text, or if a phrase was a heading (because it's prefixed with `##` in Markdown). This could influence how the model uses the information. In some cases, preserving simple markup or JSON structure can be beneficial. For instance, if a table is represented in Markdown in the chunk, the LLM might be better at reasoning about it or quoting it as a table in the answer. Compare that to just flattening everything to plain text – you lose those structural cues.
- **Semantic Annotations:** In specialized systems, you might add labels like "this chunk is about *Topic A*" or "relevance score to concept X". This is less common in generic RAG, but if you have taxonomy or knowledge graph info, you could attach it. This could allow query-time filtering by topic.

Now, how does metadata improve retrieval and traceability?

For retrieval, metadata can be used in **hybrid search** or filtering. Many vector databases allow a boolean or SQL-like filter alongside vector similarity search. For example, if the user's query says "according to *Document XYZ*...", you can restrict the search to chunks where `source = "Document XYZ"`. Or you might boost scores for certain sources if you know they are more authoritative. If you stored document creation dates, you could filter or favor the latest info (useful in a RAG system that has time-sensitive data, e.g., a Q&A over an evolving knowledge base).

Also, you can do **metadata-based ranking**. Suppose your chunks have a field "section: Introduction/Body/FAQ/etc." If a user question contains "FAQ" or resembles an FAQ style, you might first retrieve only chunks from FAQs. This is an advanced tuning, but it shows how extra info can guide retrieval beyond pure text similarity.

For traceability, as mentioned, having the source and possibly even the exact paragraph reference means the final answer can include citations. Many implementations of RAG (like ChatGPT with browsing or Bing Chat) will show an answer and footnote which source web pages contributed. In our context, since we have chunk-level granularity, we might cite the specific document and section. Users can then read more if needed. Imagine the model answers: *"Graphite is an excellent conductor of electricity due to the delocalized electrons in its carbon layers【1】."* The citation [1] could map to the source document or even a hyperlink. That builds trust.

Storing rich metadata also helps during **debugging and evaluation** of your RAG system. If you notice a bad answer was given, you can trace which chunks were retrieved and see their source. You might realize, for example, that chunks from an irrelevant document were retrieved – then you could improve your metadata filters or chunk content.

Formats like Markdown or JSON: When extracting documents, you might actually choose to store them in a structured JSON form from the get-go. This way each chunk can also carry a "type" field (paragraph, title, list, etc.). Or you might store the chunk text in Markdown format to preserve things like headings or code fences. There isn't a one-size rule, but consider that sometimes preserving simple formatting in the chunk text can improve LLM responses. For example, an LLM might answer differently if it sees a chunk that was a bullet list vs a narrative paragraph – it might decide to enumerate an answer. Also, if the user asks for code output, having the chunk with triple backticks (Markdown code) might help it present it properly.

9. The Future of Chunking as LLM Context Windows Increase

With recent advances, we're seeing LLMs that can accept extraordinarily large context windows – tens or even hundreds of thousands of tokens. This begs the question: *will chunking (and even RAG in general) become unnecessary if we can just feed the entire knowledge base to the model?* It's a hot debate.

Long Context LLMs vs RAG: A model with a 1 million token context could, in theory, take an entire encyclopedia as input. But in practice, just because you *can* provide all data doesn't mean the model can effectively utilize it. Models have a limited attention budget and still tend to focus on certain parts of the input. If you stuff a huge amount of text in the prompt, the model might struggle to identify the relevant pieces (garbage in, garbage out). RAG's retrieval step acts as a focus mechanism: it intentionally selects a small subset of text likely to be useful, which can actually lead to better answers than giving a giant dump of text and hoping the model finds the answer inside.

That said, longer context models **do reduce the strict need to split into very small chunks**. We might shift chunking strategies accordingly:

- We can afford **larger chunk sizes** since the model can handle it. This can be helpful for maintaining more context in each chunk. For instance, if previously we chunked at 500 tokens for a 4k context model, with a 100k context model we might chunk at 5k or 10k tokens, basically feeding whole sections or entire documents in one go, assuming the embedding model is large enough to support those chunk sizes.
- With huge contexts, one could implement a simpler form of retrieval: maybe retrieve entire documents (or large portions thereof) rather than fine-grained chunks. This is more like a search engine retrieving whole pages for the model to read. For example, a system might fetch the top 2 relevant documents (each maybe 5k tokens) and give them to a model with a 10k window, instead of 10 small chunks.
- **Fewer chunks needed:** If the model can take more tokens, we might not need to cut a long answer or code listing into multiple parts. The model can see it in one piece.

However, **efficiency and cost** become concerns. Large context models are more expensive and slower to run (filling a 100k context could cost a lot of compute for one query). If we blindly throw everything at the model, we pay a huge cost per query. Retrieval and chunking will likely remain useful to keep things efficient – why feed 500 pages if only 2 pages are relevant?

Information Consistency: Another implication is that with more context, the model might try to synthesize larger swaths of info. This can be good (more holistic answers) but also risky if sources conflict. For example, if two retrieved documents have slightly different stats for the same metric, a large context model might see both and be unsure which is correct, or it might average them or pick one arbitrarily. Smaller context RAG often implicitly avoids this by only giving one source for a given detail (the top one). As context increases, we might need to consider **how to present multiple sources** such that the model can reconcile them. Perhaps by grouping consistent ones or explicitly telling the model to compare them. This is an open area of research – how to handle multi-source input so the model's answer stays accurate and not confused. We have a separate article that goes over best practices of data preparation for RAG which covers this topic.

Late Chunking / Hierarchical Retrieval: One interesting idea enabled by longer contexts is a two-tier approach: you could retrieve a large chunk (maybe an entire long document) into context, and then rely on the model to *itself* do a second stage of finding the answer in it. Essentially, you push some of the retrieval work onto the model (since it can scan a bigger text). For example, with 100k context, you might retrieve the 5 most relevant documents of ~20k each into the prompt, and ask the model to extract and synthesize the answer from all that. This starts to look like what we do manually – read multiple articles to answer a question – but done by the model in one shot. It's powerful, but again, expensive and not guaranteed to be flawless.

There is also research into **hierarchical chunking**, where a model with long context first takes in a lot of text and produces a summary or intermediate result, which is then used downstream (kind of like map-reduce with an LLM). This could reduce how much needs to be fed in final answer generation. Chunking could evolve to support that: e.g. create larger chunks, have the model summarize each chunk separately (as a form of compression), then feed summaries in a final step along with maybe one or two key chunks for detail.

Will chunking ever disappear? If we one day have practically unlimited context windows and super cheap computation, maybe we wouldn't need to chunk – we could just index and retrieve entire documents. But even then, the concept of *retrieval* would remain because you usually have far more data than you want to feed (imagine a model that could read 100 million tokens – you still wouldn't feed it all of Wikipedia for every query; you'd retrieve the relevant article or section). So retrieval is here to stay. Chunking as a specific preprocessing might become more minimal – perhaps just splitting by document, or by logical sections, not into tiny bits.

Consistency in Answers: With many chunks in context, an LLM might have to weigh different pieces of evidence. Ensuring it consistently uses the right evidence is a challenge. Future RAG systems might incorporate reasoning steps (like the model first finds which part of the context is relevant, then answers). Some current implementations already do something like: "Given the following documents, list which ones are relevant to the query" as a step for the LLM, then "Now answer using those relevant ones." This is like making the LLM do its own retrieval from the provided big context.

In conclusion, as LLM context windows expand, we will likely use **larger and fewer chunks**, but we'll still *curate* what goes into those windows via retrieval. Chunking strategies may shift towards keeping larger semantic units intact (since we won't be as constrained by small windows), and retrieval may incorporate more coarse-grained elements (like whole docs) followed by the model doing fine-grained reading. It's an exciting development because it could reduce some of the fragmentation required now, but it also introduces new design questions on how to best supply and highlight relevant info to the model. Researchers (including us at GPT-trainer) are actively exploring these questions, and we can expect RAG architectures to evolve in tandem with LLM capabilities rather than be rendered obsolete.