weaviate

Context Engineering

Designing the systems that control what information reaches the model and how it maintains coherence.

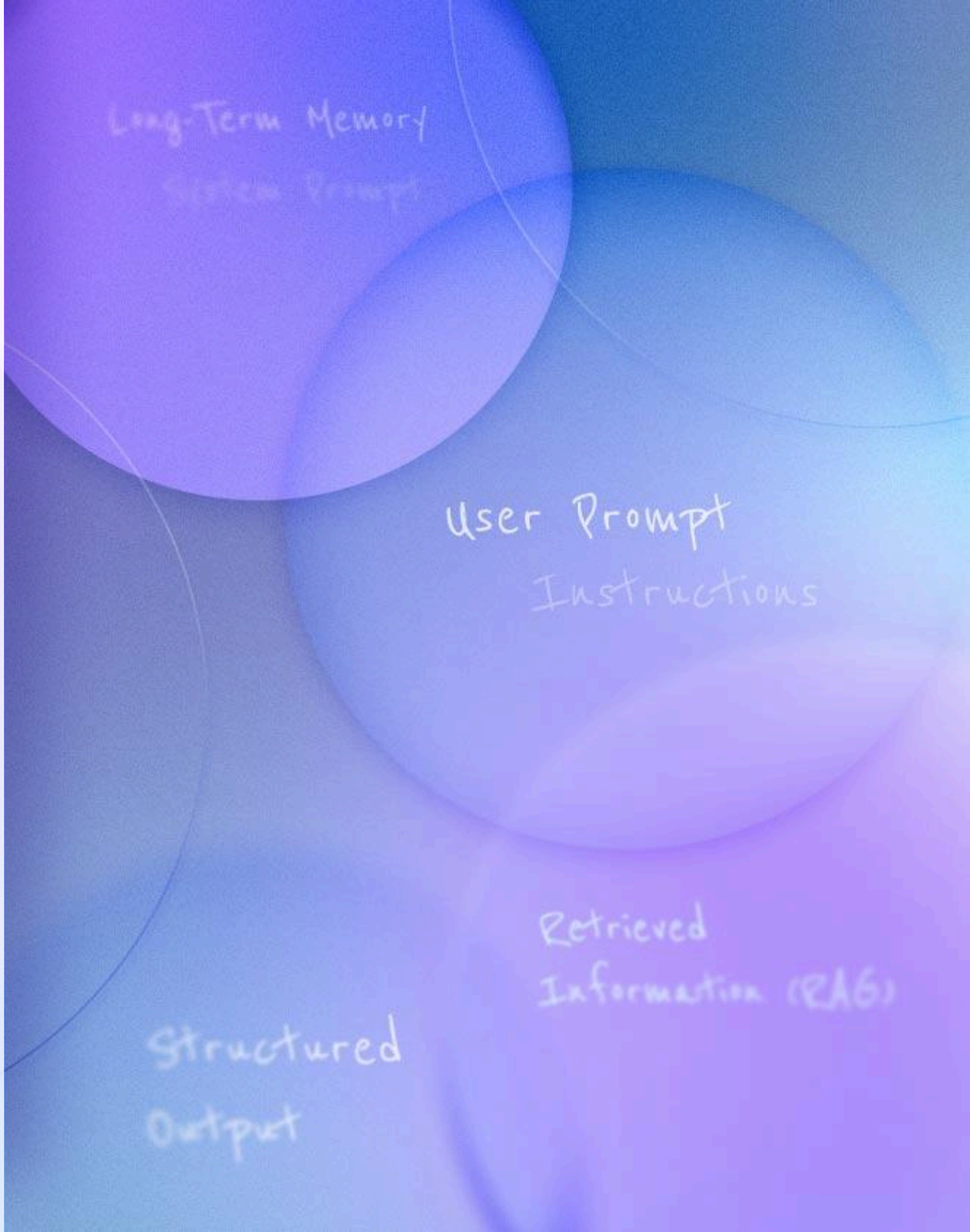
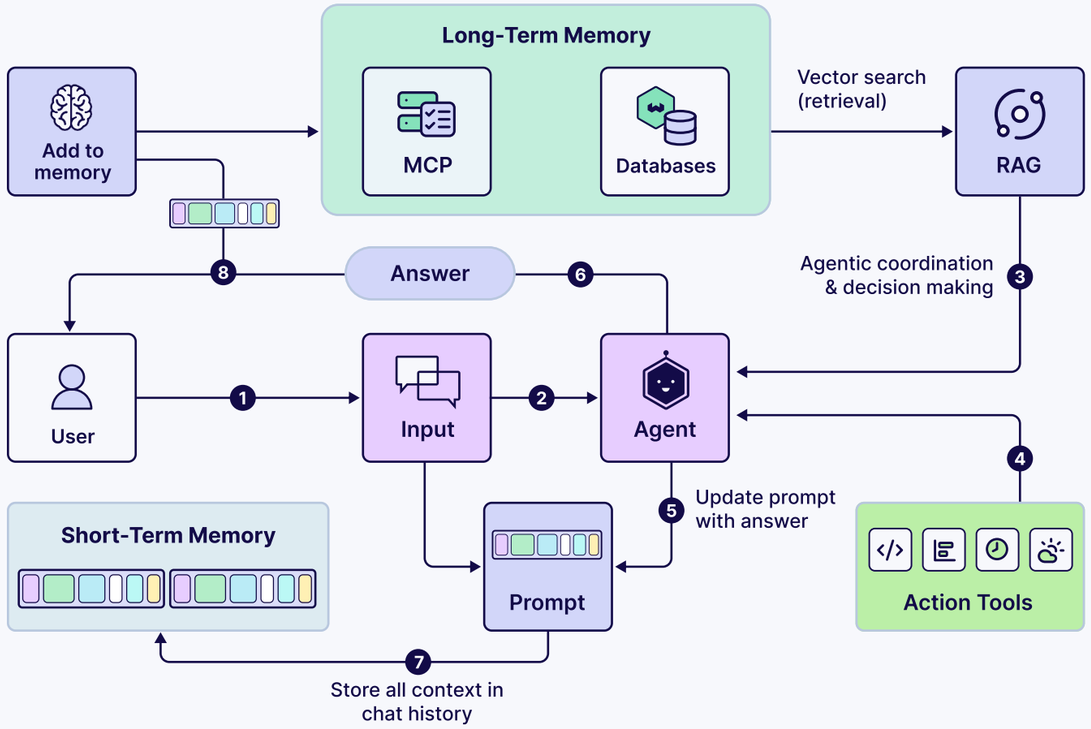


Table Of Contents

Introduction

Every developer who builds with Large Language Models (LLMs) eventually hits the same wall. You start with a powerful model that can write, summarize, and reason with stunning capability. But when you try to apply it to a real-world problem, the cracks start to appear. It can't answer questions about your private documents. It has no knowledge of events that happened yesterday. It confidently makes things up when it doesn't know an answer.



The problem isn't the model's intelligence. The problem is that it's fundamentally disconnected. It's a powerful but isolated brain, with no access to your specific data, the live internet, or even a memory of your last conversation. This isolation is a direct result of its core architectural limit: **the context window**. The context window is the model's active working memory—the finite space where it holds the instructions and information for the current task. Every word, number, and piece of punctuation consumes space in this window. Just like a whiteboard, once it's full, older information gets erased to make room for new instructions, and important details can be lost.

You can't fix this fundamental limitation by just writing better prompts. You have to build a system around the model.

That is Context Engineering.

Context Engineering is the discipline of designing the architecture that feeds an LLM the right information at the right time. It's not about changing the model itself, but about building the bridges that connect it to the outside world, retrieving external data, connecting it to live tools, and giving it a memory to ground its responses in facts, not just its training data.

This ebook is the blueprint for that system. We will cover the core components required to turn a brilliant but isolated model into a reliable, production-ready application.

Mastering these components is the difference between a reasonable demo and a truly intelligent system. Let's get to work.

Agents

The decision-making brain that orchestrates how and when to use information.

Query Augmentation

The art of translating messy, ambiguous user requests into precise, machine-readable intent.

Retrieval

The bridge connecting the LLM to your specific documents and knowledge bases.

Prompting Techniques

The skill of giving clear, effective instructions to guide the model's reasoning.

Memory

The system that gives your application a sense of history and the ability to learn from interactions.

Tools

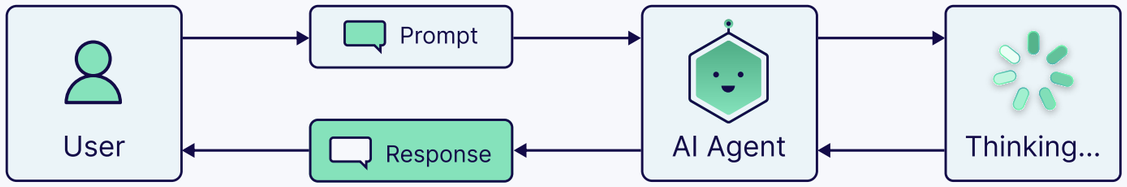
The hands that allow your application to take direct action and interact with live data sources.

Agents

As soon as you start building real systems with large language models, you run into the limits of static pipelines. A fixed recipe of “retrieve, then generate” works fine for simple Retrieval Augmented Generation (RAG) setups, but it falls apart once the task requires judgment, adaptation, or multi-step reasoning.

This is where **agents** come in. In the context of context engineering, agents manage how (and how well) information flows through a system. Instead of blindly following a script, agents can evaluate what they know, decide what they still need, select the right tools, and adjust their strategy when things go wrong.

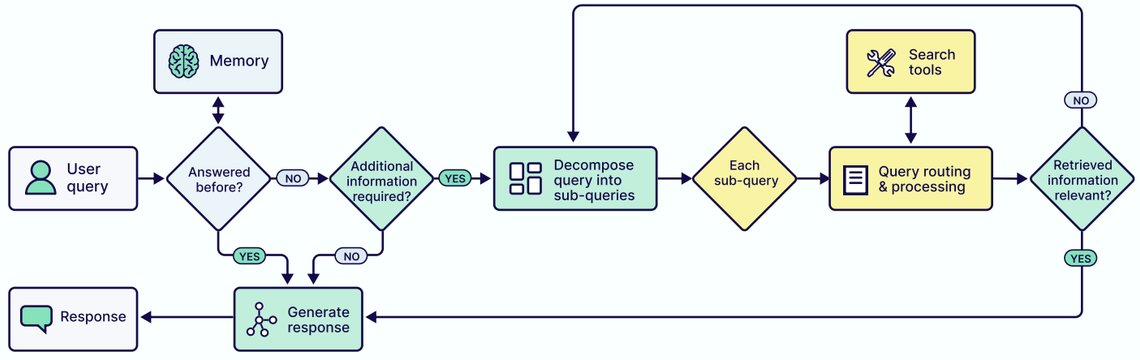
Agents are both **the architects of their contexts** and **the users of those contexts**. However, they need good practices and systems to guide them, because managing context well is difficult, and getting it wrong quickly sabotages everything else the agent can do.



What Are Agents?

The term "agent" gets used broadly, so let’s define it in the context of building with large language models (LLMs). An AI agent is a system that can:

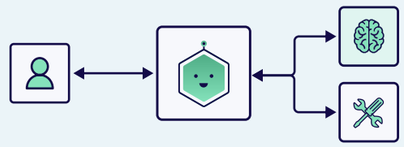
1. **Make dynamic decisions about information flow.** Rather than following a predetermined path, agents decide what to do next based on what they've learned.
2. **Maintain state across multiple interactions.** Unlike simple Q&A systems, agents remember what they've done and use that history to inform future decisions.



3. **Use tools adaptively.** They can select from available tools and combine them in ways that weren’t explicitly programmed.
4. **Modify their approach based on results.** When one strategy isn’t working, they can try different approaches.

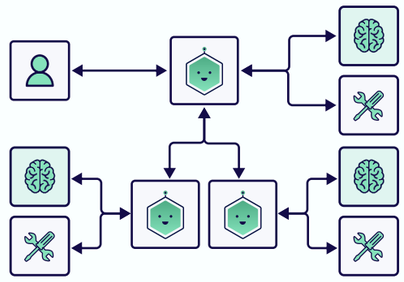
Single-Agent Architecture

Attempt to handle all tasks themselves, which works well for moderately complex workflows.



Multi-Agent Architecture

Distribute work across specialized agents per task. Allows for complex workflows but introduces coordination challenges.



Context Hygiene

This is one of the most critical parts of managing agentic systems. Agents don't just need memory and tools; they also need to monitor and manage the quality of their own context. That means avoiding overload, detecting irrelevant or conflicting information, pruning or compressing as needed, and keeping their in-context memory clean enough to reason effectively.

The Context Window Challenge

LLMs have **limited information capacity** because the context window can only hold so much information at once. This fundamental constraint shapes what agents and agentic systems are currently capable of.

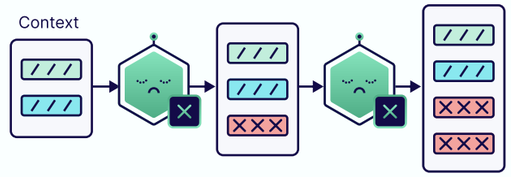
Every time an agent is processing information, it needs to make decisions about:

- **What information should remain active in the context window**
- **What should be stored externally and retrieved when needed**
- **What can be summarized or compressed to save space**
- **How much space to reserve for reasoning and planning**

It's tempting to assume that bigger context windows solve this problem, but this is simply *not* the case. Longer contexts (hundreds of thousands or even ~1M tokens) actually introduces new failure modes.

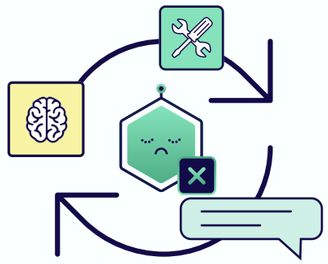
Performance often begins to degrade far before the model reaches maximum token capacity, where agents will become confused, have higher rates of hallucination, or simply stop performing at the level they're normally capable of. This isn't just a technical limitation, it's a core design challenge of any AI app.

Here are some common types of errors that begin to happen or increase as context window size grows:



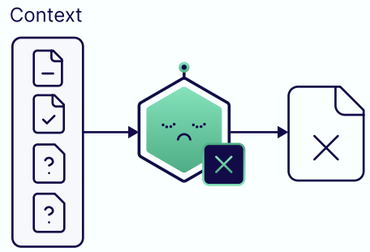
Context Poisoning

Incorrect or hallucinated information enters the context. Because agents reuse and build upon that context, these errors persist and compound.



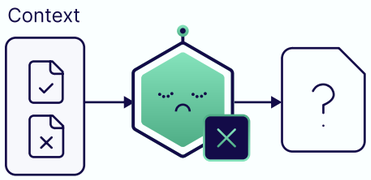
Context Distraction

The agent becomes burdened by too much past information—history, tool outputs, summaries—and over-relies on repeating past behavior rather than reasoning fresh.



Context Confusion

Irrelevant tools or documents crowd the context, distracting the model and causing it to use the wrong tool or instructions.



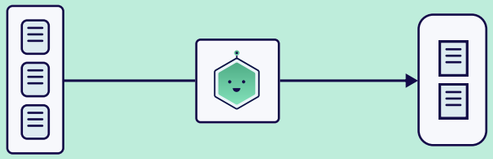
Context Clash

Contradictory information within the context misleads the agent, leaving it stuck between conflicting assumptions.

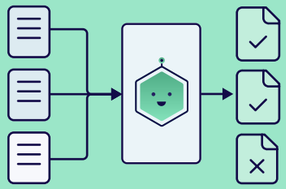
Strategies and Tasks for Agents

Agents are able to effectively orchestrate context systems because of their ability to **reason and make decisions in a dynamic way**. Here are some of the most common tasks agents are built for and employ to manage contexts.

Context Summarization: Periodically compressing accumulated history into summaries to reduce burden while preserving key knowledge.

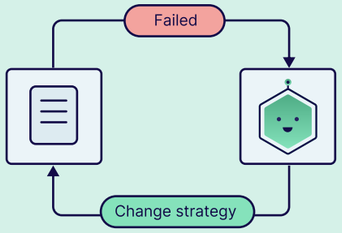


Quality Validation: Checking whether retrieved information is consistent and useful.

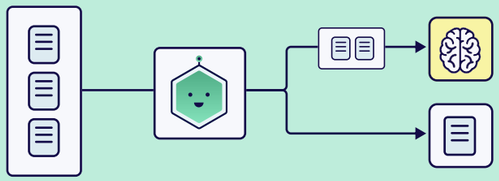


Context Pruning: Actively removing irrelevant or outdated context, either with specialized pruning models or a dedicated LLM tool.

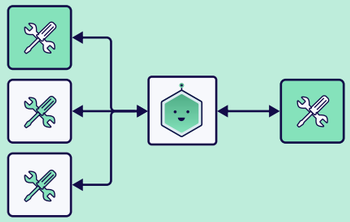
Adaptive Retrieval Strategies: Reformulating queries, switching knowledge bases, or changing chunking strategies when initial attempts fail.



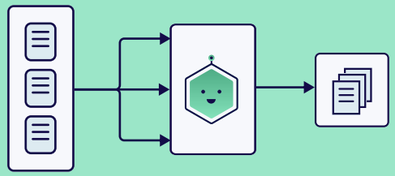
Context Offloading: Storing details externally and retrieving them only when needed, instead of keeping everything in active context.



Dynamic Tool Selection: Instead of dumping every possible tool into the prompt, agents filter and load only those relevant to the task. This reduces confusion and improves accuracy.



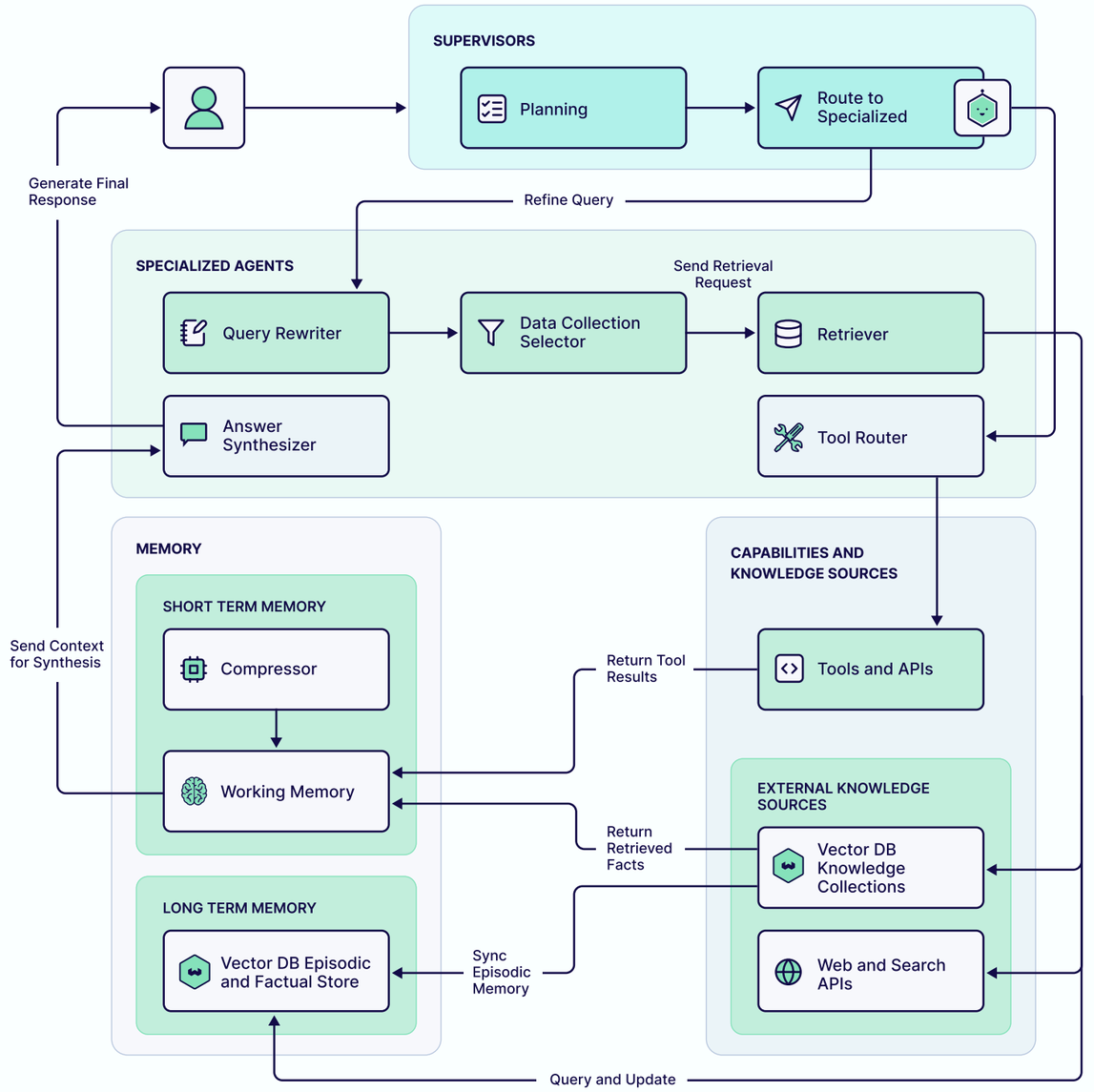
Multi-Source Synthesis: Combining information from multiple sources, resolving conflicts, and producing coherent answers.



Where Agents Fit in Context Engineering

Agents serve as coordinators in your context engineering system. They don't replace the techniques covered in other sections, instead, they orchestrate them intelligently. An agent might apply query rewriting when initial searches are unsuccessful, choose different chunking strategies based on the type of content it encounters, or decide when conversation history should be compressed to make room for new information. They provide the orchestration layer needed to make dynamic, context-appropriate decisions about information management.

Different types of agents and functions within a context engineering system:



Query Augmentation

One of the most important steps of context engineering is how you prepare and present the user's query. Without knowing exactly what the user is asking, the LLM cannot provide an accurate response.

Though this sounds simple, it's actually quite complex. There are two main issues to think about:

1 Users often don't interact with chatbots or inputs in the ideal way.

Many product builders will often develop and test chatbots with queries that provide the request and all additional information that the LLM would need to understand the question in a succinct, perfectly punctuated, clear way. Unfortunately, in the real world, user interactions with chatbots can be unclear, messy, and not complete. In order to build robust systems, it's important to implement solutions that deal with all types of interactions, not just ideal ones.

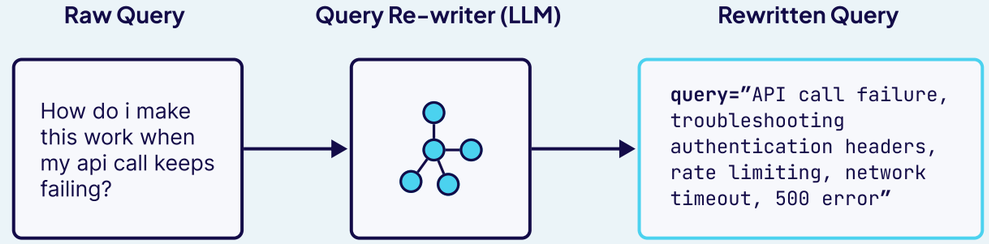
2 Different parts of the pipeline need to deal with the query in different ways.

A question that an LLM could understand well might not be the best format to search through a vector database with. Or, a query term that works best for a vector database could be incomplete for an LLM to answer. Therefore, we need a way to augment the query that suits different tools and steps within the pipeline.

Remember that query augmentation addresses the "*garbage in, garbage out*" problem at the very start of your pipeline. No amount of sophisticated retrieval algorithms, advanced reranking models, or clever prompt engineering can fully compensate for misunderstood user intent.

Query Rewriting

Query rewriting transforms the original user query into a more effective version for information retrieval. Instead of just doing retrieve-then-read, applications now do a rewrite-retrieve-read approach. This technique restructures oddly written questions so they can be better understood by the system, removes irrelevant context, introduces common keywords that improve matching with correct context, and can split complex questions into simpler sub-questions.



RAG applications are sensitive to the phrasing and specific keywords of the query, so this technique works by:

Restructuring Unclear Questions:

Transforms vague or poorly formed user input into precise, information-dense terms.

Context Removal:

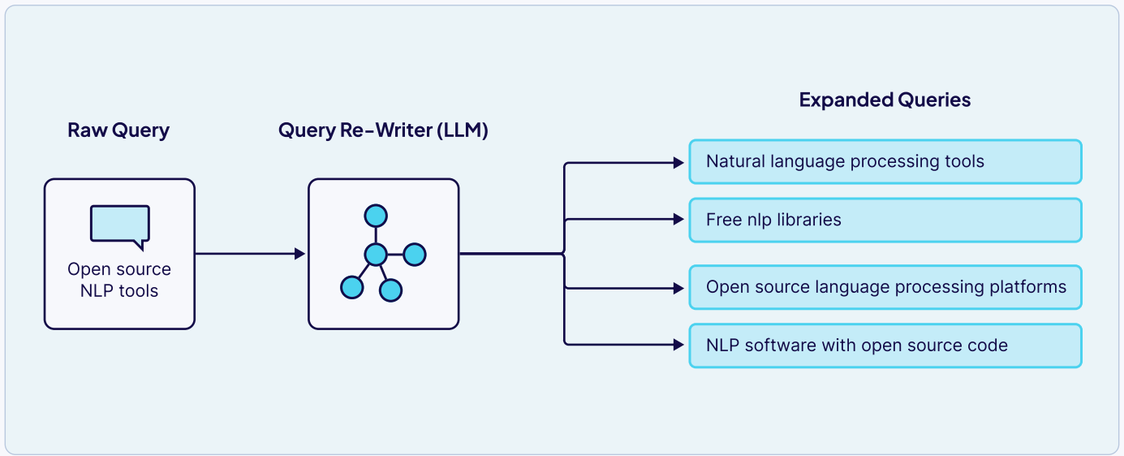
Eliminates irrelevant information that could confuse the retrieval process.

Keyword Enhancement:

Introduces common terminology that increases the likelihood of matching relevant documents.

Query Expansion

Query expansion enhances retrieval by generating multiple related queries from a single user input. This approach improves results when user queries are vague, poorly formed, or when you need broader coverage, such as with keyword-based retrieval systems.



However, query expansion comes with challenges that need careful management:

Query Drift:

Expanded queries may diverge from the user's original intent, leading to irrelevant or off-topic results.

Over-Expansion:

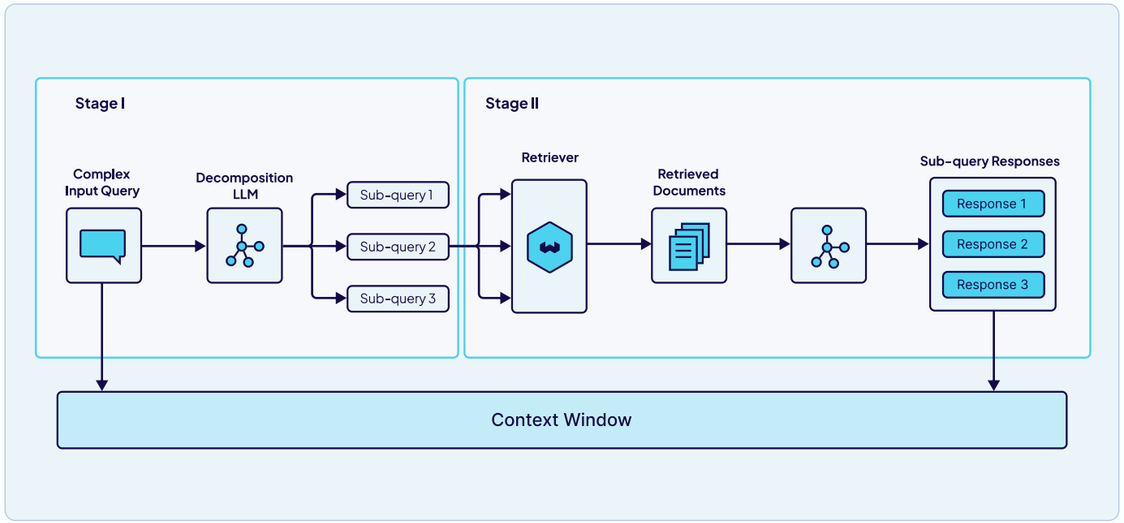
Adding too many terms can reduce precision and retrieve excessive irrelevant documents.

Computational Overhead:

Processing multiple queries increases system latency and resource usage.

Query Decomposition

Query decomposition breaks down complex, multi-faceted questions into simpler, focused sub-queries that can be processed independently. This technique is especially good for questions that require information from multiple sources or involve several related concepts.



The process typically involves two main stages:

Decomposition Phase:

An LLM analyzes the original complex query and breaks it into smaller, focused sub-queries. Each sub-query targets a specific aspect of the original question.

Processing Phase:

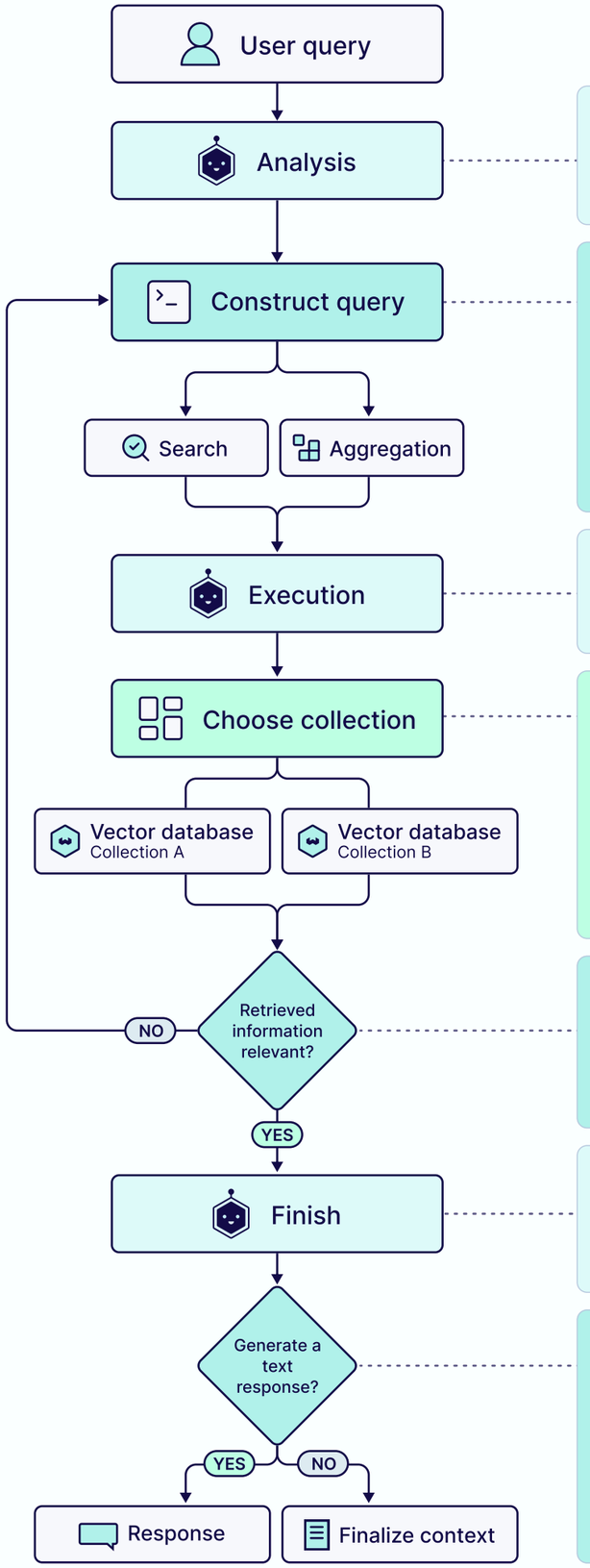
Each sub-query is processed independently through the retrieval pipeline, allowing for more precise matching with relevant documents.

After retrieval, the context engineering system must aggregate and synthesize results from all sub-queries to generate a coherent, comprehensive answer to the original complex query.

Query Agents

Query Agents are the most advanced form of query augmentation, using AI agents to intelligently handle the entire query processing pipeline, combining the techniques above.

A query agent takes a user's prompt/question in natural language and decides the best way to structure the query based on it's knowledge of the database and data structure, and can iteratively decide to re-query and adjust based on the results returned.



1. **Analysis:** Use generative models (e.g. large language models) to analyze the task & the required queries. Determine the exact queries to perform.

Dynamic Query Construction: Rather than using predetermined query patterns, the agent constructs queries on-demand based on understanding both the user intent and the data schema. This means it can add filters and adjust search terms automatically to find the most relevant results in the database, as well as choosing to run searches, aggregations or even both at the same time for you.

2. **Query execution:** Formulates and sends queries to the agent's chosen collection or collections.

Multi-collection routing: The agent understands the structure of all of your collections, so it can intelligently decide which data collections to query based on the user's question.

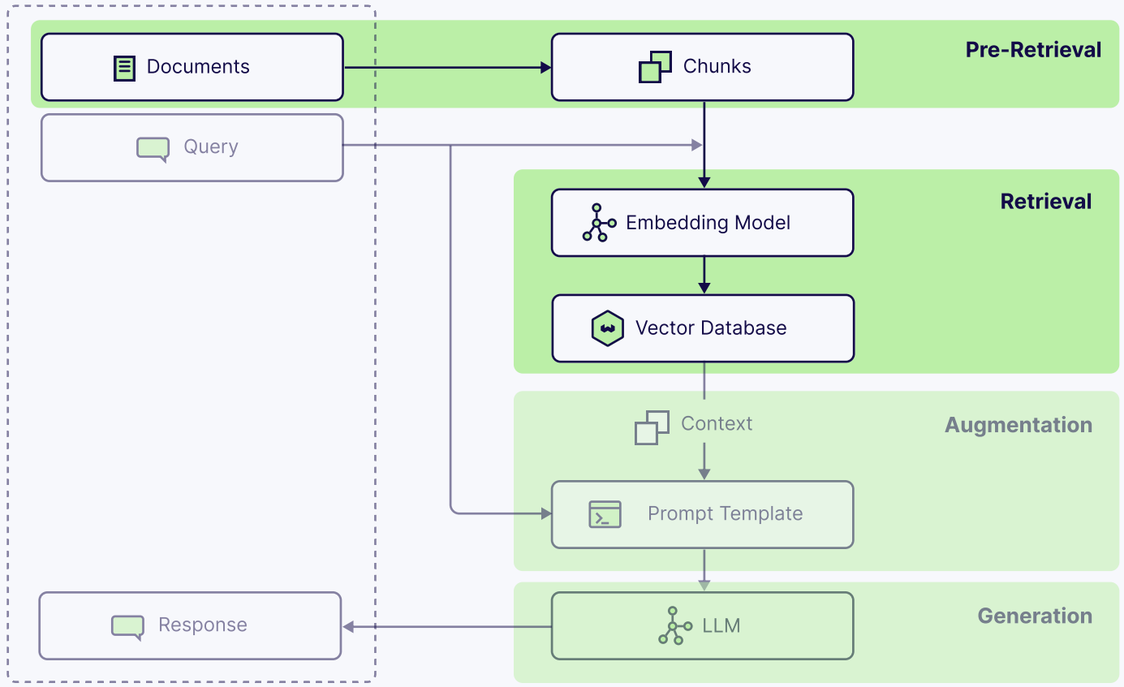
Evaluation: The agent can evaluate the retrieved information within the context of the original user query. If there is missing information, the agent can try a different knowledge source or new query.

3. **(Optional) Response generation:** Receive the results from the database, and use a generative model to generate the final response to the user's prompt/query.

Contextual awareness: The context may also include previous conversation history, and any other relevant information. The agent can maintain conversation context for follow-up questions.

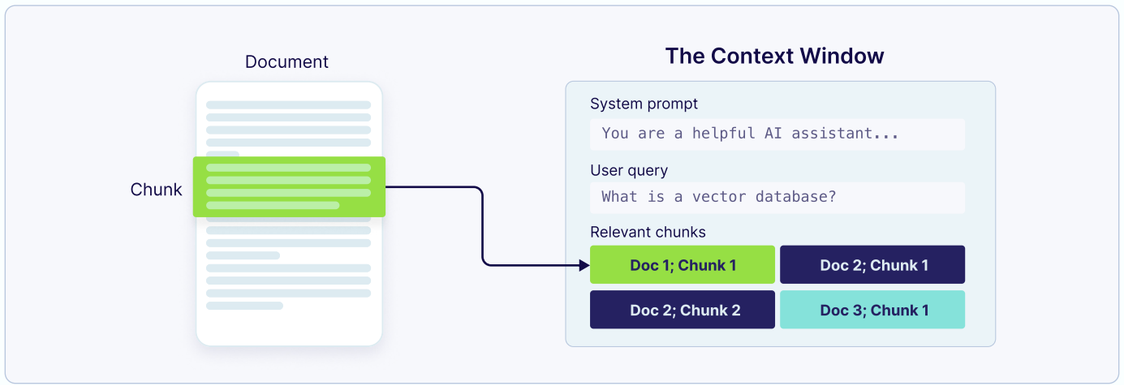
Retrieval

A Large Language Model is only as good as the information it can access. While LLMs are trained on massive datasets, they lack knowledge of your specific, private documents and any information created after their training was completed. To build truly intelligent applications, you need to feed them the right external information at the right time. This process is called Retrieval. Pre-Retrieval and Retrieval steps make up the first parts of many AI architectures that rely on context engineering, such as Retrieval Augmented Generation (RAG).



The challenge is simple in concept but tricky in practice: a raw dataset of documents is almost always too large to fit into an LLM's limited context window (the inputs given to an AI model). We can't just hand the model an entire set of user manuals or research papers. Instead, we must find the perfect piece of those documents, the single paragraph or section that contains the answer to a user's query.

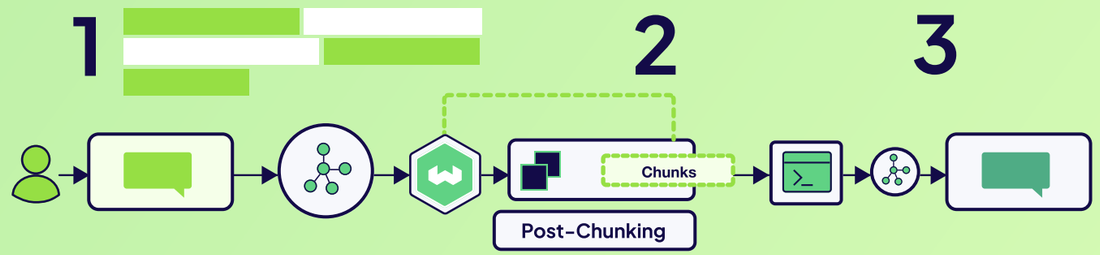
To make our vast knowledge bases searchable and find that perfect piece, we must first break our documents down into smaller, manageable parts. This foundational process, known as **chunking**, is the key to successful retrieval.



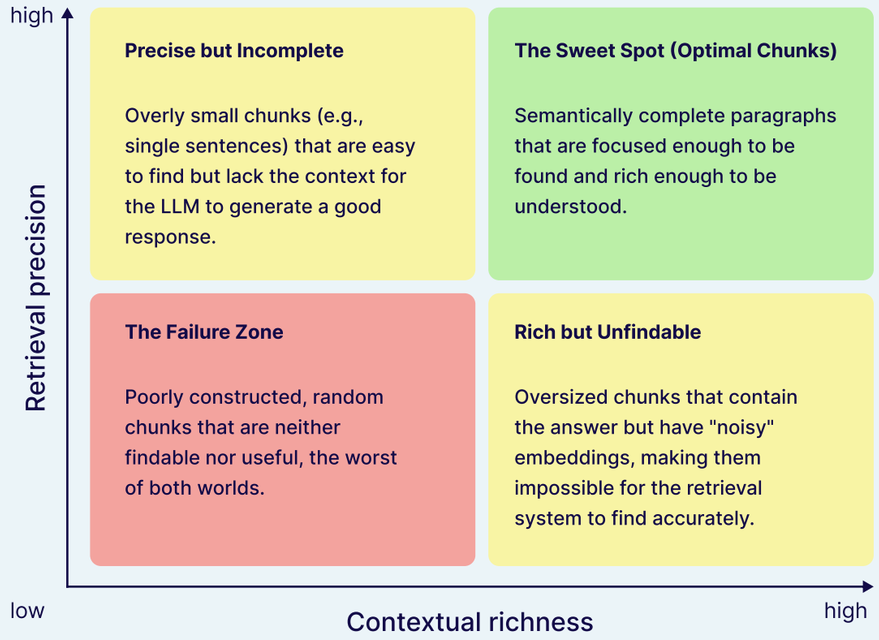
Learn how chunking strategies can help improve your RAG performance and explore different chunking methods. Read the complete blog post here: weaviate.io/blog/chunking-strategies-for-rag

A Guide to Chunking Techniques

Chunking is the most important decision you will make for your retrieval system's performance. It is the process of breaking down large documents into smaller, manageable pieces. Get it right, and your system will be able to pinpoint relevant facts with surgical precision. Get it wrong, and even the most advanced LLM will fail.



The Chunking Strategy Matrix



When designing your chunking strategy, you must balance two competing priorities:

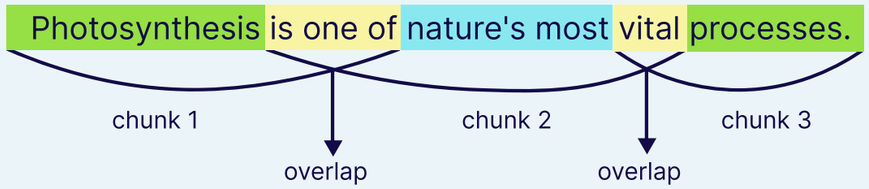
- **Retrieval Precision:** Chunks need to be small and focused on a single idea. This creates a distinct, precise embedding, making it easier for a vector search system to find an exact match for a user's query. Large chunks that mix multiple topics create "averaged," noisy embeddings that are hard to retrieve accurately.
- **Contextual Richness:** Chunks must be large and self-contained enough to be understood. After a chunk is retrieved, it is passed to the LLM. If the chunk is just an isolated sentence without context, even a powerful model will struggle to generate a meaningful response.

The goal is to find the "chunking sweet spot", creating chunks that are small enough for precise retrieval but complete enough to give the LLM the full context it needs.

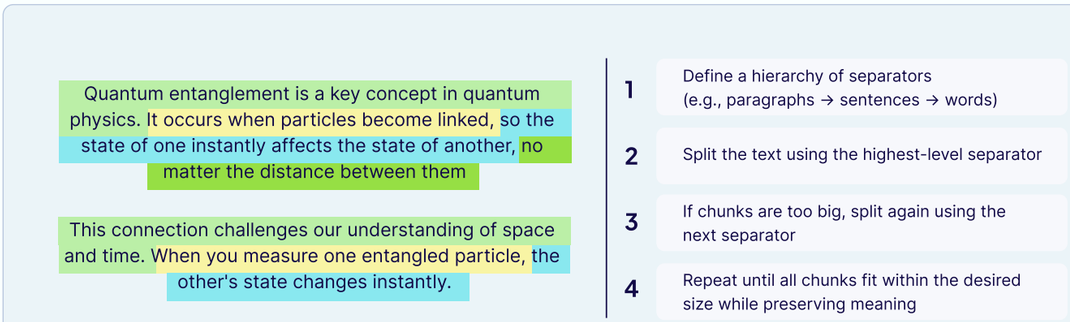
Your choice of strategy will depend on the nature of your documents and the needs of your application.

Simple Chunking Techniques

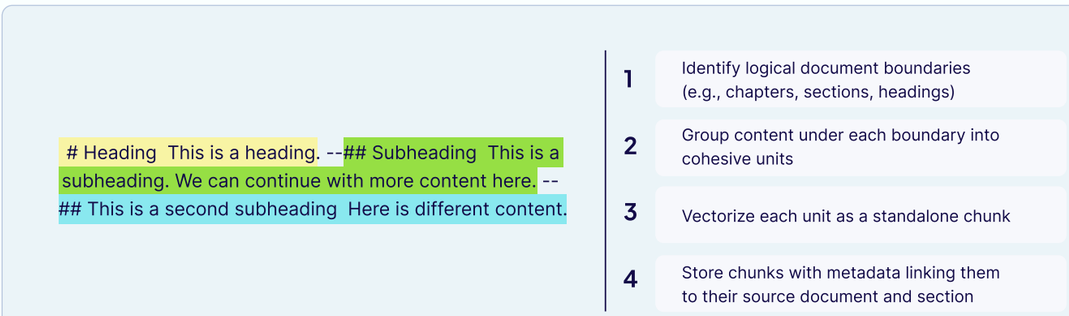
Fixed-Size Chunking: The simplest method. The text is split into chunks of a predetermined size (e.g., 512 tokens). It's fast and easy but can awkwardly cut sentences in half. Using an overlap (e.g., 50 tokens) between chunks helps mitigate this.



Recursive Chunking: A more intelligent approach that splits text using a prioritized list of separators (like paragraphs, then sentences, then words). It respects the document's natural structure and is a solid default choice for unstructured text.

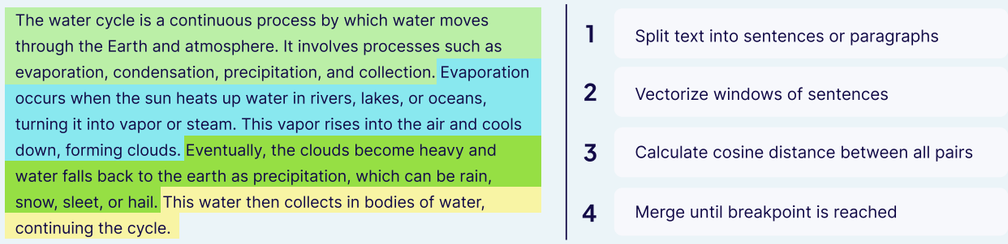


Document-Based Chunking: This method uses the document's inherent structure. For example, it splits a Markdown file by its headings (#, ##), an HTML file by its tags (<p>,), or source code by its functions.

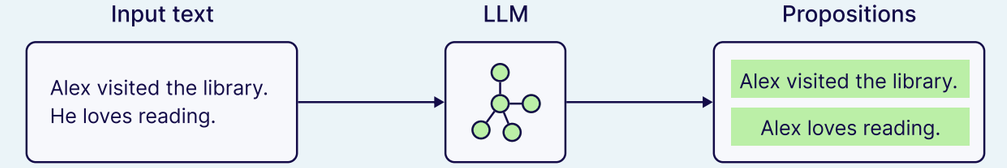


Advanced Chunking Techniques

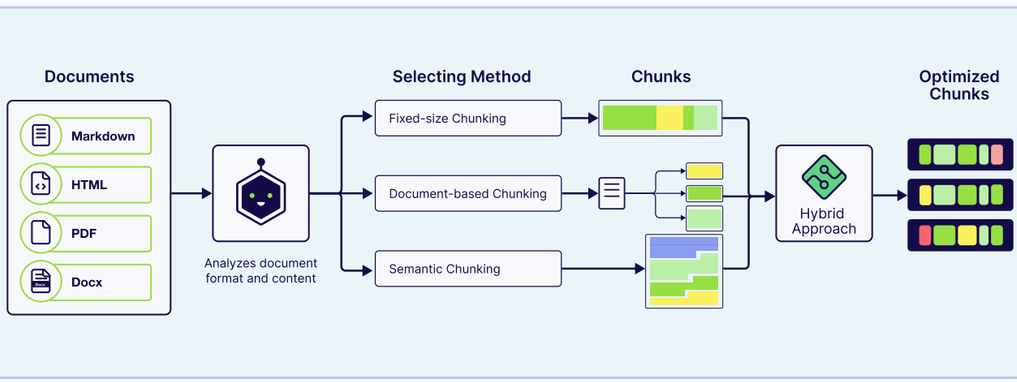
Semantic Chunking: Instead of using separators, this technique splits text based on meaning. It groups semantically related sentences together and creates a new chunk only when the topic shifts, resulting in highly coherent, self-contained chunks.



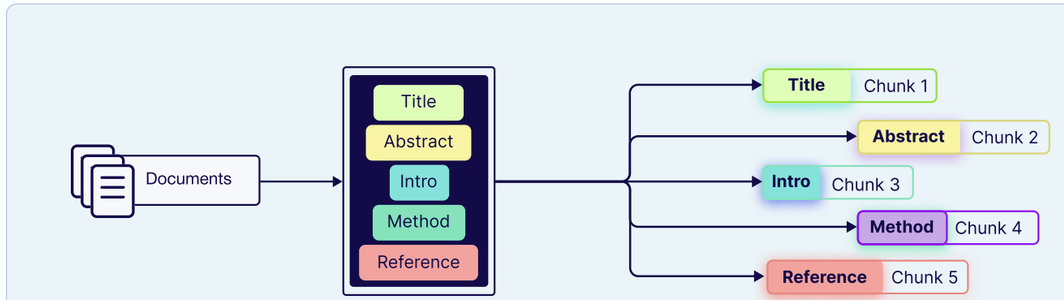
LLM-Based Chunking: Uses a Large Language Model to intelligently process a document and generate semantically coherent chunks. Instead of relying on fixed rules, the LLM can identify logical propositions or summarize sections to create meaning-preserving pieces.



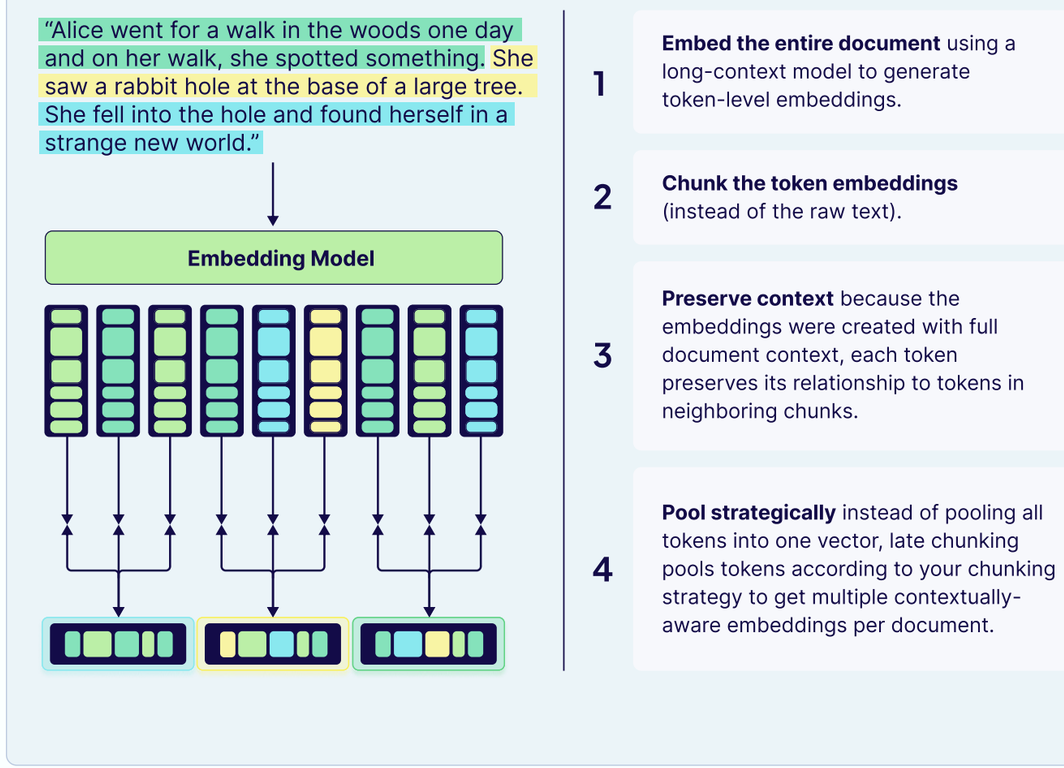
Agentic Chunking: This takes the concept a step further then LLM-Based Chunking. An AI agent dynamically analyzes a document's structure and content to select the best chunking strategy (or combination of strategies) to apply for that specific document.



Hierarchical Chunking: Creates multiple layers of chunks at different levels of detail (e.g., top-level summaries, mid-level sections, and granular paragraphs). This allows a retrieval system to start with a broad overview and then drill down into specifics as needed.



Late Chunking: An architectural pattern that inverts the standard process. It embeds the entire document first to create token-level embeddings with full context. Only then is the document split into chunks, with each chunk's embedding derived from these pre-computed, context-rich tokens.

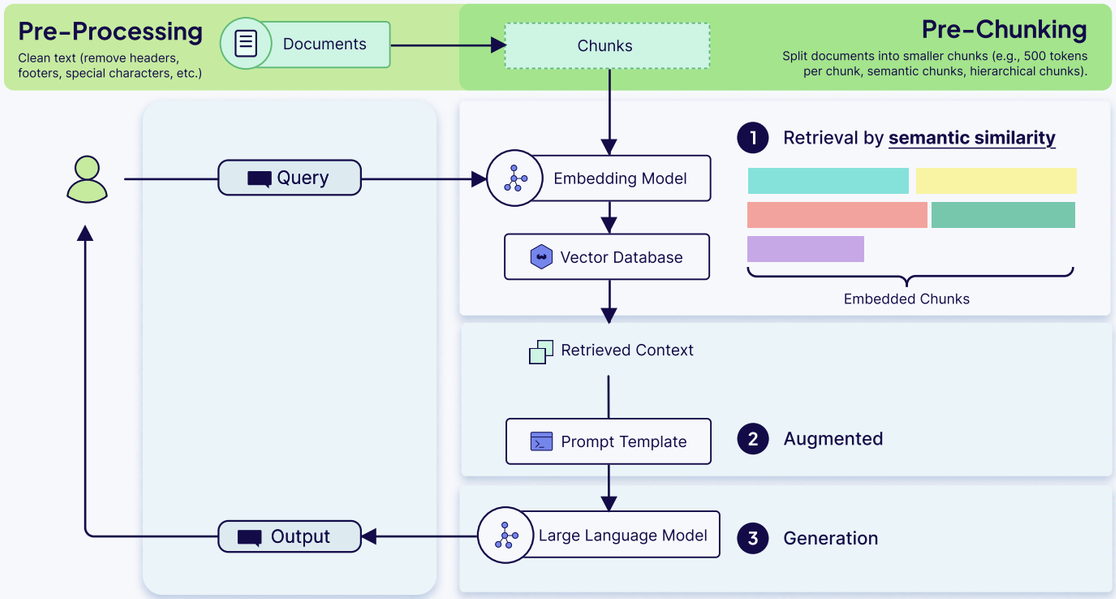


Pre-Chunking vs. Post-Chunking

Beyond *how* you chunk, a key system design choice is *when* you chunk. This decision leads to two primary architectural patterns.

Pre-Chunking

The most common method, where all data processing happens upfront and offline, before any user queries come in.



Workflow Clean Data → Chunk Documents → Embed & Store Chunks

PRO

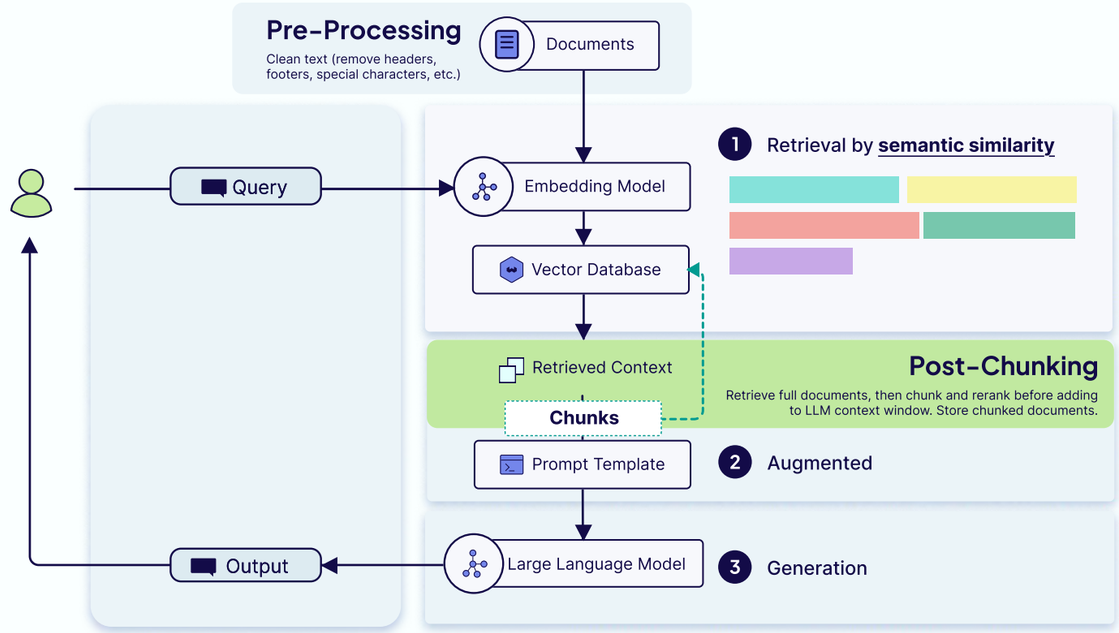
Retrieval is extremely fast at query time because all the work has already been done. The system only needs to perform a quick similarity search.

CON

The chunking strategy is fixed. If you decide to change your chunk size or method, you must re-process your entire dataset.

Post-Chunking

An advanced, real-time alternative where chunking happens after a document has been retrieved, in direct response to a user's query.



Workflow Store Documents → Retrieve Relevant Document → Chunk Dynamically

PRO

It's highly flexible. You can create dynamic chunking strategies that are specific to the context of the user's query, potentially leading to more relevant results.

CON

It adds latency. The chunking process happens in real-time, making the first response slower for the end-user. It also requires more complex infrastructure to manage.

> **We built a post-chunking strategy into Elysia, our open source agentic RAG framework. You can read more about that here: https://weaviate.io/blog/elysia-agentic-rag#chunk-on-demand-smarter-document-processing**

Guide to Choosing Your Chunking Strategy

Chunking Strategy	How It Works	Complexity	Best For	Examples
Fixed-Size	Splits by token or character count.	●○○○○	Small or simple docs, or when speed matters most.	Meeting notes, short blog posts, emails, simple FAQs.
Recursive	Splits text by repeatedly dividing it until it fits the desired chunk size, often preserving some structure.	●●○○○	Documents where some structure should be maintained but speed is still important.	Research articles, product guides, short reports.
Document-Based	Splits only at document boundaries or by structural elements like headers.	●○○○○	Collections of short, standalone documents or highly structured files.	News articles, customer support tickets, Markdown files.
Semantic	Splits text at natural meaning boundaries (topics, ideas).	●●●○○	Technical, academic, or narrative documents where topics shift without clear separators.	Scientific papers, textbooks, novels, whitepapers.
LLM-Based	Uses a language model to decide chunk boundaries based on context and meaning.	●●●●○	Complex text where meaning-aware chunking improves downstream tasks like Q&A.	Long reports, legal opinions, medical records.
Agentic	Lets an AI agent decide how to split based on meaning and structure.	●●●●●	Complex, nuanced documents that require custom strategies.	Regulatory filings, multi-section contracts, corporate policies.
Late Chunking	Embeds the whole document first, then derives chunk embeddings from it.	●●●●○	Use cases where chunks need awareness of the full document's context.	Case studies, comprehensive manuals, long-form analysis reports.
Hierarchical	Breaks text into multiple levels (sections → paragraphs → sentences).	●●●○○	Large, structured documents where both summary and detail are needed.	Employee handbooks, government regulations, software documentation.

Summary

The effectiveness of your Retrieval Augmentation system is not determined by a single “magic” bullet, but by a series of deliberate engineering choices. The quality of the context you provide to an LLM is a direct result of two key decisions:

1. The Chunking Strategy

The "How"

The method you choose to break down your documents.

2. The Architectural Pattern

The "When"

The point at which you perform the chunking.

Mastering these two elements is fundamental to context engineering. A well-designed retrieval system is the difference between an LLM that guesses and one that provides fact-based, reliable, and contextually relevant answers.

Prompting Techniques

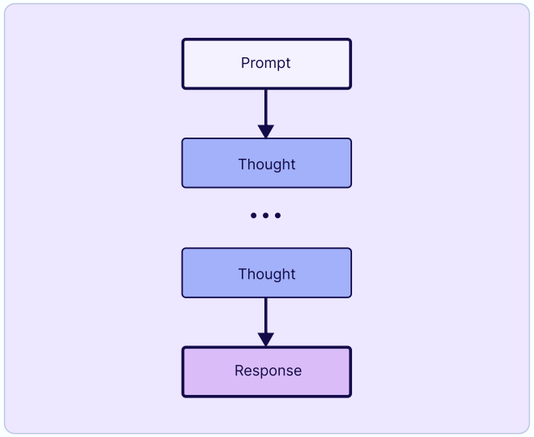
Prompt engineering is the practice of designing, refining, and optimizing inputs (prompts) given to Large Language Models (LLMs) to get your desired output. The quality and effectiveness of LLMs are heavily influenced by the prompts they receive, and the way you phrase a prompt can directly affect the accuracy, usefulness, and clarity of the response.

It's essentially about interacting with AI efficiently: giving it instructions, examples, or questions that guide the model toward the output you need.

In this section, we'll go over prompting techniques that are essential for improving Retrieval-Augmented Generation (RAG) applications and overall LLM performance.

Important Note: Prompt engineering focuses on how you phrase instructions for the LLM. Context engineering, on the other hand, is about **structuring the information and knowledge you provide** to the model, such as retrieved documents, user history, or domain-specific data, to maximize the model's understanding and relevance. Many of the techniques below (CoT, Few-shot, ToT, ReAct) are most effective when combined with **well-engineered context.**

Classic Prompting Techniques

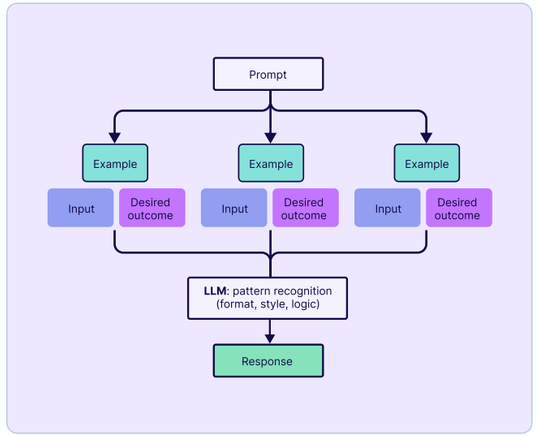


Chain of Thought

This technique involves asking the model to "think step-by-step" and break down complex reasoning into intermediate steps. This is especially helpful when retrieved documents are dense or contain conflicting information that requires careful analysis. By verbalizing its reasoning process, the LLM can come at more accurate and logical conclusions.

Few-Shot Prompting

This approach provides the LLM with a few examples in the context window that demonstrate the type of output or "golden" answers you want. Showing examples helps the model understand the desired format, style, or reasoning approach, improving response accuracy and relevance, especially for specialized or technical domains.



Combining CoT and Few-shot examples is a powerful way to guide both the model's reasoning process and its output format for maximum efficiency.

Pro Tip #1:

Make the model reasoning in Chain of Thought very specific to your use-case. For example, you might ask the model to:

- Evaluate the environment
- Repeat any relevant information
- Explain the importance of this information to the current request

Pro Tip #2:

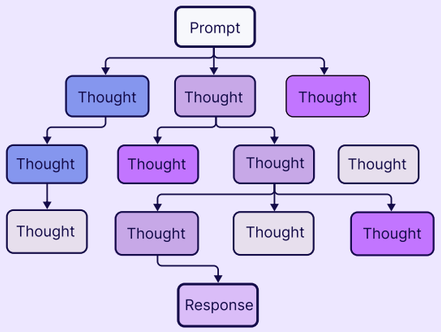
Maximize efficiency and reduce token count, asking the model to reason in a "draft" form, using no more than 5 words per sentence.

This makes sure that the model's thought process is visible while reducing output token count.

Advanced Prompting Strategies

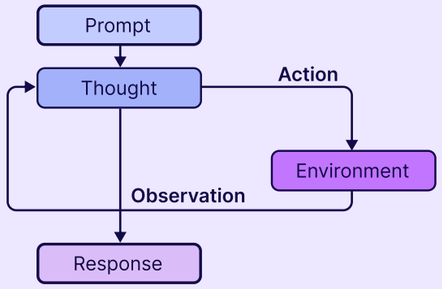
Building on classic techniques, advanced strategies guide LLMs in more sophisticated ways:

Tree of Thoughts (ToT):



ToT builds on CoT by instructing the model to explore and evaluate multiple reasoning paths in parallel, much like a decision tree. The model can generate several different solutions to a problem and choose the best result. This is especially useful in RAG when there are many potential pieces of evidence, and the model needs to weigh different possible answers based on multiple retrieved documents.

ReAct Prompting:



This framework combines CoT with agents, enabling the model to "Reason" (think) and "Act" dynamically. The model generates both reasoning traces and actions in an interleaved manner, allowing it to interact with external tools or data sources and adjust its reasoning iteratively. ReAct can improve RAG pipelines by enabling LLMs to interact with retrieved documents in real time, updating reasoning and actions based on external knowledge to give more accurate and relevant responses.

Prompting for Tool Usage

When your LLM interacts with external tools, clear prompting ensures correct tool selection and usage.

Defining Parameters and Execution Conditions

LLMs can sometimes make incorrect tool selections or use tools in suboptimal ways. To prevent this, prompts should clearly define:

When to use a tool:
Specify scenarios or conditions that trigger a particular tool.

How to use a tool:
Provide expected inputs, parameters, and desired outputs.

Examples: Include few-shot examples showcasing correct tool selection and usage for various queries. For instance:

User Query:
"What's the weather like in Paris?" → Use Weather_API with city="Paris"

User Query:
"Find me a restaurant near the Eiffel Tower." → Use Restaurant_Search_Tool with location="Eiffel Tower"

This very precise guidance, which should be included as part of your overall tool description, helps the LLM understand the exact boundaries and functionalities of each available tool, minimizing error, and improving overall system reliability.

Pro Tip: How to Write an Effective Tool Description

The LLM's decision to use your tool depends entirely on its description. Make it count:

- **Use an Active Verb:** Start with a clear action. get_current_weather is better than weather_data.
- **Be Specific About Inputs:** Clearly state what arguments the tool expects and their format (e.g., city (string), date (string, YYYY-MM-DD)).
- **Describe the Output:** Tell the model what to expect in return (e.g., returns a JSON object with "high", "low", and "conditions".).
- **Mention Limitations:** If the tool only works for a specific region or time frame, say so (e.g., Note: Only works for cities in the USA.).

Using Prompt Frameworks

If you are building a project that requires extensive prompting or want to systematically improve your LLM results, you could consider using frameworks like: DSPy, Llama Prompt Ops, Synalinks.

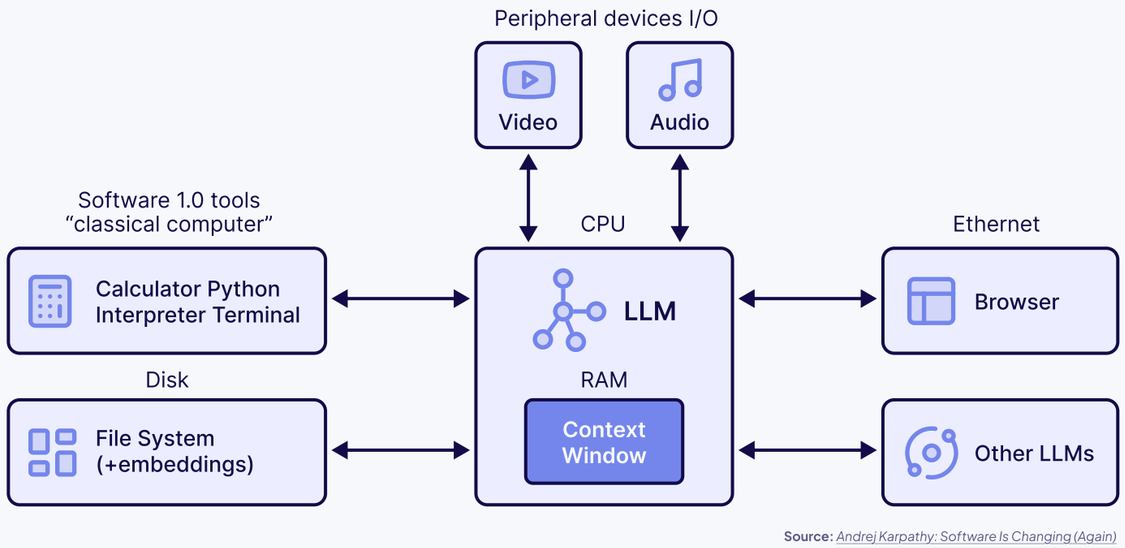
That said, you don't necessarily need to use a framework. Following the prompting guidelines outlined (clear instructions, Chain of Thought, Few-shot Learning, and advanced strategies) can achieve highly effective results without additional frameworks.

Think of these frameworks as optional helpers for complex projects, not a requirement for everyday prompt engineering.

Memory

When you're building agents, memory isn't just a bonus feature - it's the very thing that breathes life into them. Without it, an LLM is just a powerful but stateless text processor that responds to one query at a time with no sense of history. Memory transforms these models into something that feels more dynamic and, dare we say, more 'human', that's capable of holding onto context, learning from the past, and adapting on the fly.

Andrej Karpathy gave us the perfect analogy when he compared an LLM's context window to a computer's RAM and the model itself to the CPU. In this view, the context window is the agent's active consciousness, where all its "working thoughts" are held. But just like a laptop with too many browser tabs open, this RAM can fill up fast. Every message, every tool output, every piece of information consumes precious tokens.



Source: *Andrej Karpathy: Software Is Changing (Again)*

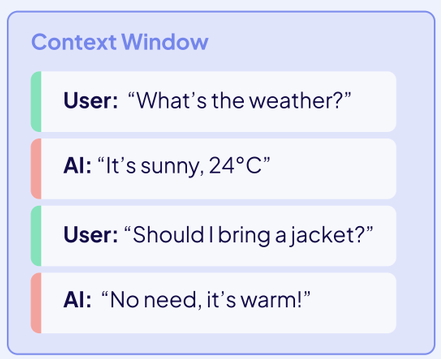
This is where context engineering becomes an art. The goal isn't to shove more data into the prompt but to design systems that make the most of the active context window - keeping essential information within reach while gracefully offloading everything else into smarter, more persistent storage.

> **Context Offloading is the practice of storing information outside the LLM's active context window, often in external tools or vector databases. This frees up the limited token space so that only the most relevant info stays in context.**

The Architecture of Agent Memory

Memory in an AI agent is all about retaining information to navigate changing tasks, remember what worked (or didn't), and think ahead. To build robust agents, we need to think in layers, often blending different types of memory for the best results.

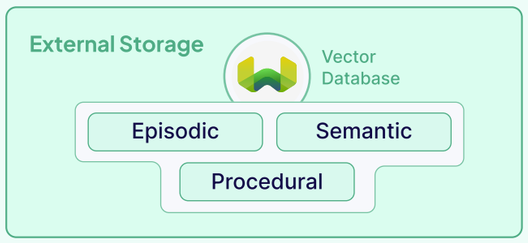
Short-Term Memory



Short-term memory is the agent's immediate workspace. It's the "now," stuffed into the context window to fuel on-the-fly decisions and reasoning. This is powered by in-context learning, where you pack recent conversations, actions, or data directly into the prompt.

Because it's constrained by the model's token limit, the main challenge is efficiency. And, the trick is to keep this streamlined to reduce costs and latency without missing any details that might be important for the next processing steps.

Long-Term Memory



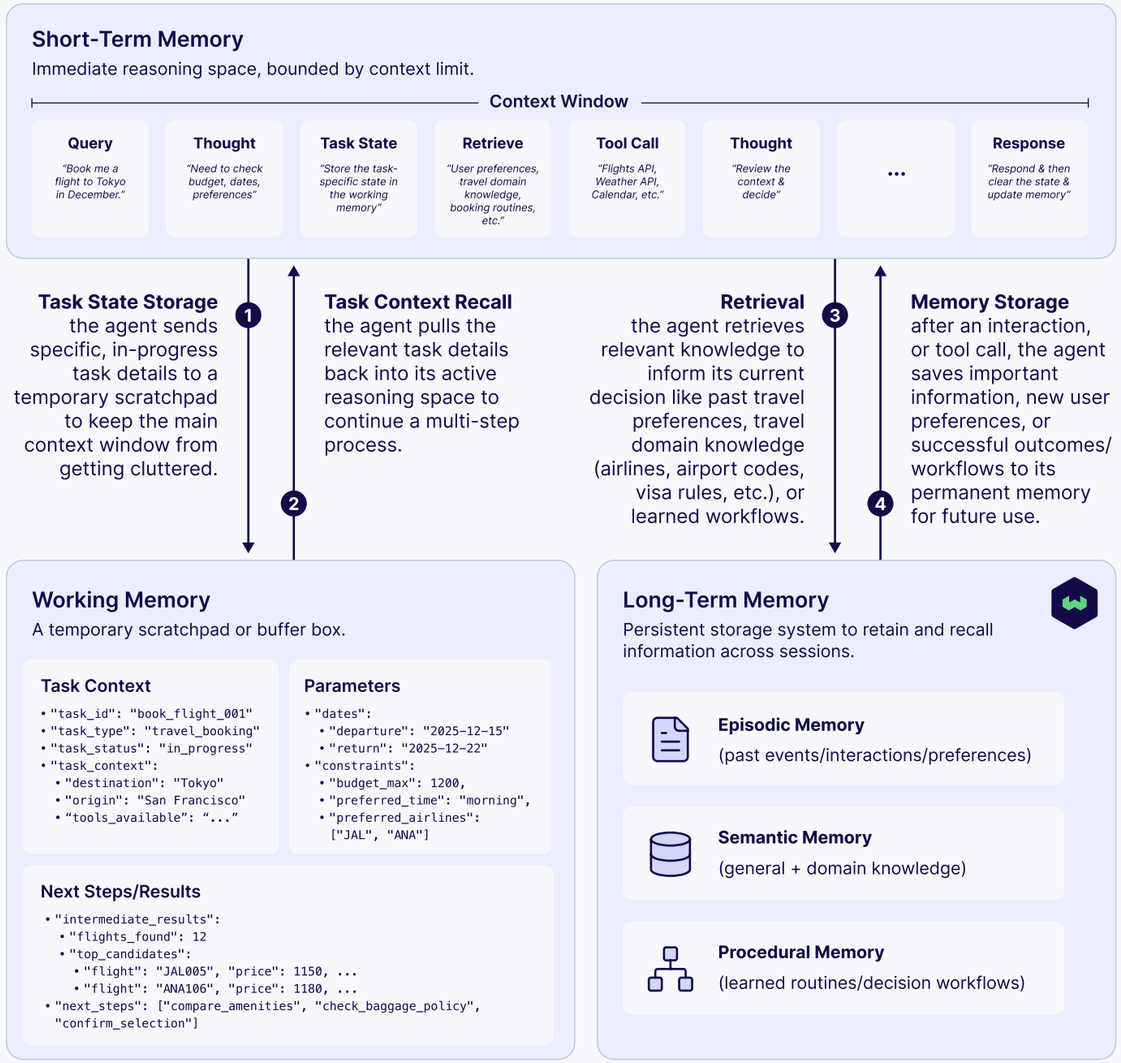
Long-term memory moves past the immediate context window, storing information externally for quick retrieval when needed. This is what allows an agent to build a persistent understanding of its world and its users over time. It's commonly powered by Retrieval-Augmented Generation (RAG), where the agent queries an external knowledge base (like a vector database) to pull in relevant information.

This memory can store different kinds of information, like for example: episodic memory to store specific events or past interactions, or semantic memory that holds general knowledge and facts. This could also be information from company documents, product manuals, or a curated domain-knowledge base, allowing the agent to answer questions with factual accuracy.

Hybrid Memory Setup

In reality, most modern systems use a hybrid approach, blending short-term memory for speed with long-term memory for depth. Some advanced architectures even introduce additional layers:

- **Working Memory:** A temporary holding area for information related to a specific, multi-step task. For example, if an agent is booking a trip, its working memory might hold the destination, dates, and budget until the task is complete, without cluttering the long-term store.
- **Procedural Memory:** This helps an agent learn and master routines. By observing successful workflows, the agent can internalize a sequence of steps for a recurring task, making it faster and more reliable over time.



Key Principles for Effective Memory Management

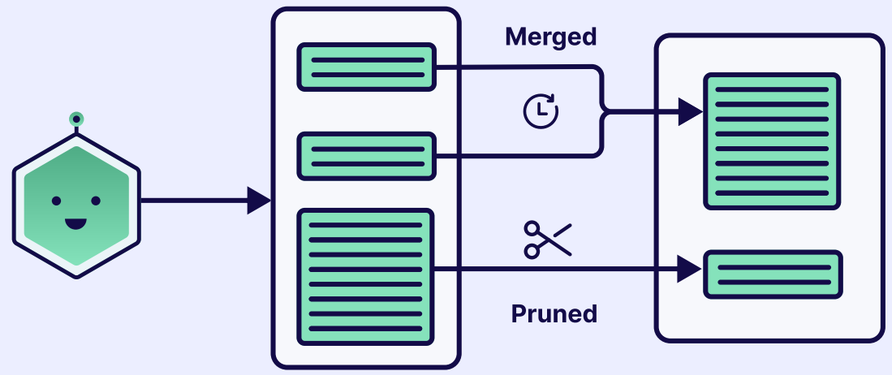
Effective memory management can make or break an LLM agent. Poor memory practices lead to error propagation, where bad information gets retrieved and amplifies mistakes across future tasks.

Here are some of the starting principles for getting things right:

Prune and Refine Your Memories

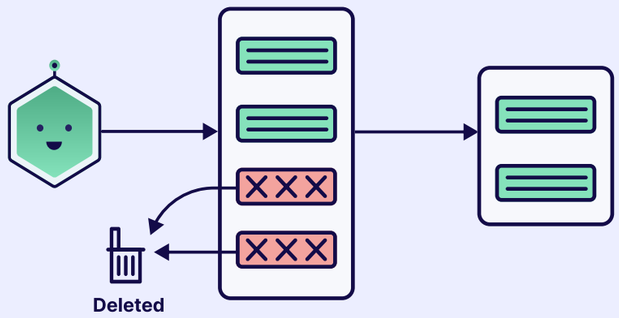
Memory isn't a write-once system. It needs regular maintenance. Periodically scan your long-term storage to remove duplicate entries, merge related information, or discard outdated facts. A simple metric for this could be the recency and retrieval frequency. If a memory is old and rarely accessed, it might be a candidate for deletion, especially in evolving environments where old information can become a liability.

For example, a customer support agent might automatically prune conversation logs that are over 90 days old and marked as resolved, closed, or no longer active in the memory. It could just retain the summaries (for trend detection and analysis) rather than full word-to-word transcripts.



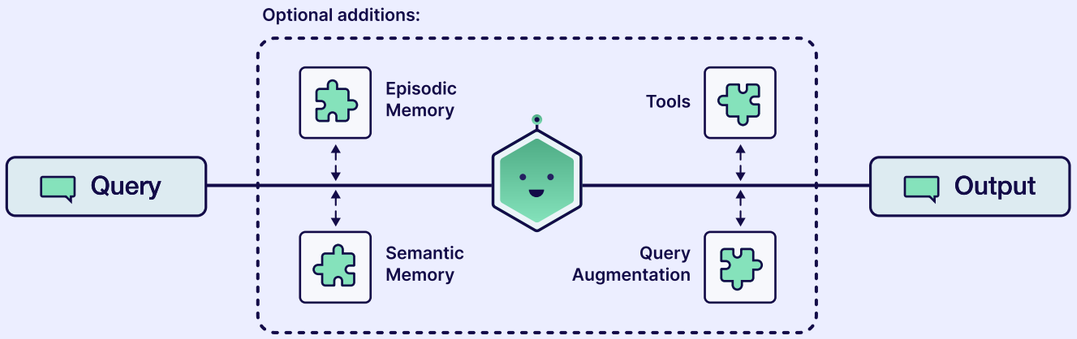
Be Selective About What You Store

Not every interaction deserves a permanent spot in long-term storage. One must implement some sort of filtering criteria to assess information for quality and relevance before saving it. A bad piece of retrieved information can often lead to context pollution, where the agent repeatedly makes the same mistakes. One way to prevent this is to have the LLM "reflect" on an interaction and assign an importance score before committing it to memory.



Tailor the Architecture to the Task

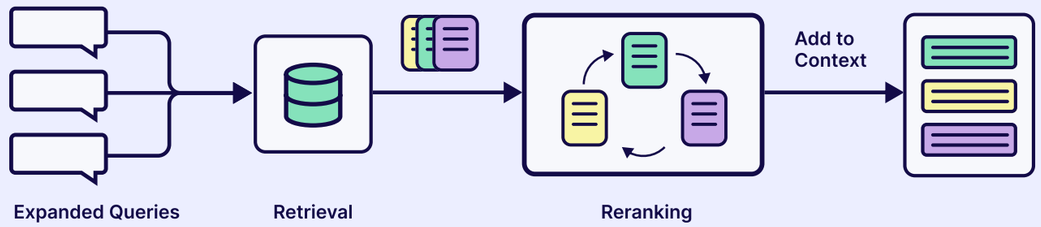
There is no one-size-fits-all memory solution. A customer service bot needs a strong episodic memory to recall user history, while an agent that analyzes financial reports needs a robust semantic memory filled with domain-specific knowledge. Always start with the simplest approach that works (like a basic conversational buffer with last ‘n’ queries/responses) and gradually layer in more advanced mechanisms as the use case demands it.



Master the Art of Retrieval

Effective memory is less about how much you can store and more about how well you can retrieve the right piece of information at the right time. A simple blind search is often not enough, so advanced techniques like reranking (using an LLM to re-order retrieved results for relevance) and iterative retrieval (refining/expanding a search query over multiple steps) can be used to improve the quality of retrieved information.

Tools like the Query Agent and Personalization Agent offer these capabilities out of the box, enabling searches across multiple collections and reranking based on user preferences and interaction history.



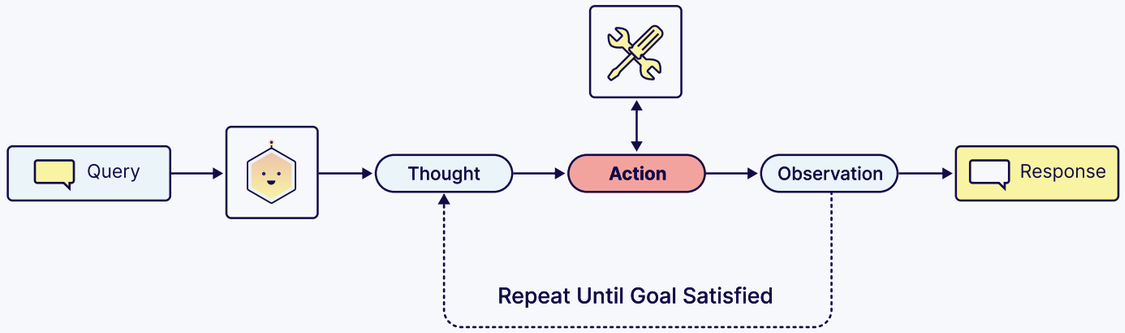
Ultimately, memory is what elevates LLM agents from simple responders to intelligent context-aware systems. Effective memory isn’t simply a passive storage... It’s an active, managed process! The goal is to build agents that don't just store memory, but can manage it - knowing what to remember, what to forget, and how to use the past to reason about the future.

Tools

If memory gives an agent a sense of self, then tools are what give it superpowers. By themselves, LLMs are brilliant conversationalists and text manipulators, but they live inside a bubble. They can't check the current weather, book a flight, or look up real-time stock prices. They are, by design, disconnected from the living, breathing world of data and action.

This is where tools come in. A "tool" is anything that connects an LLM agent to the outside world, allowing it to take direct “action” in the real world and retrieve information required to fulfill a task. Integrating tools elevates an agent from just being a knowledgeable consultant to something that can actually get things done.

Context engineering for tools isn't just giving an agent a list of APIs and instructions. It's about creating a cohesive workflow where the agent can understand what tools are available, decide correctly which one to use for a specific task, and interpret the results to move forward.



The Evolution: From Prompts to Actions

The journey to modern tool use has been a rapid evolution. Initially, devs tried to get action out of LLMs with good old prompt engineering by tricking the model into generating text that looked like a command. It was clever but prone to errors.

The real breakthrough was function calling, aka tool calling. This capability, now native to most models, allows an LLM to output structured JSON that can contain the name of a function to call and the arguments to use.

With this, there are a bunch of possibilities:

A Simple Tool

A travel agent bot can use a search_flights tool, and when a user asks, "Find me a flight to Tokyo next Tuesday," the LLM doesn't guess the answer. It generates a call to the function you provided, which in turn queries a real airline API.

A Chain of Tools

For a complex request like "Plan a weekend trip to San Francisco for me," the agent might need to chain several tools together: find_flights, search_hotels, and get_local_events. This requires the agent to reason, plan, and execute a multi-step workflow.

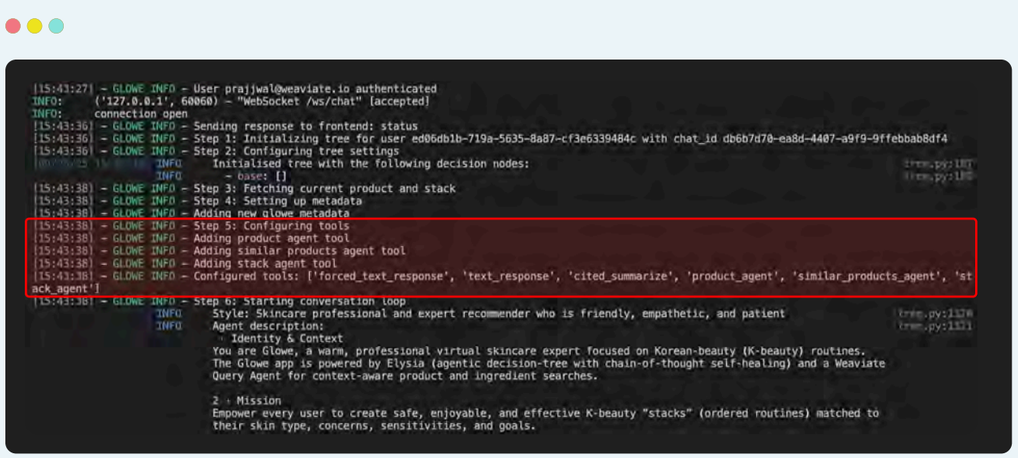
The work of context engineering here is in how you present these tools. A well-written tool description is like a mini-prompt that guides the model, making it crystal clear what the tool does, what inputs it needs, and what it returns.

The Orchestration Challenge

Giving an agent a tool is easy (mostly). Getting it to use that tool reliably, safely, and effectively is where the real work begins. The central task of context engineering is orchestration, i.e., managing the flow of information and decision-making as the agent reasons about which tool to use.

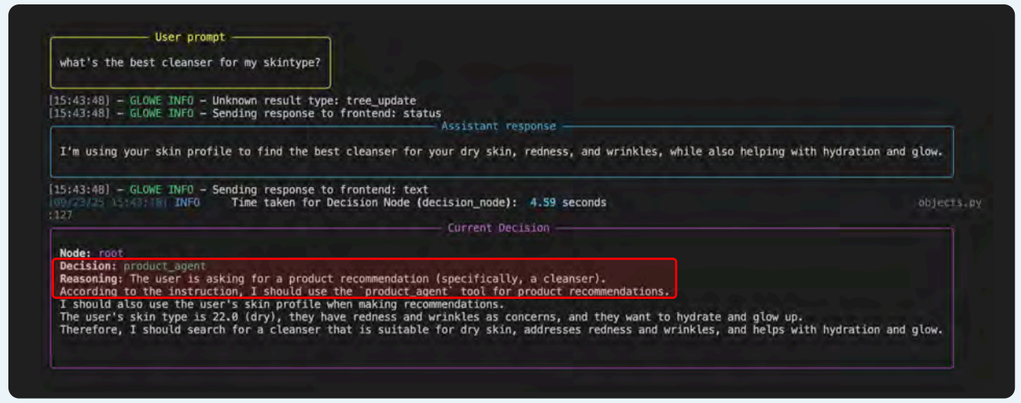
This involves a few key steps that happen in the context window. Let's break down these key orchestration steps using Glowe, a skincare domain knowledge app powered by our Elysia orchestration framework, as our running example.

1. **Tool Discovery**: The agent needs to know what tools it has at its disposal. This is usually done by providing a list of available tools and their descriptions in the system prompt. The quality of these descriptions is very critical. They are the agent's only guide to understanding what each tool does, allowing the model to understand when to use a tool and, more importantly, when to avoid it.



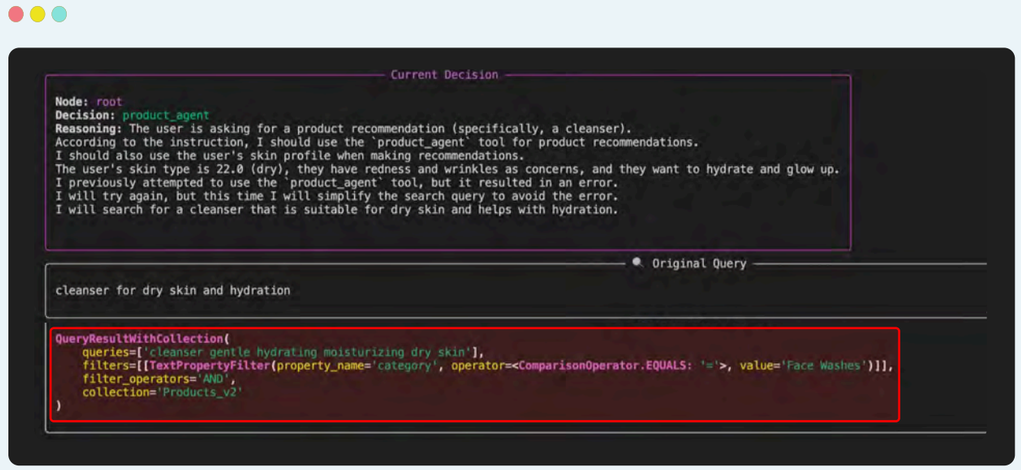
In Glowe, we configure a set of specialized tools (Step 5) with precise descriptions when initializing every new chat tree.

2. **Tool Selection and Planning (Thought)**: When faced with a user request, the agent must reason about whether a tool is needed. If so, which one? For complex tasks, it might even need to chain multiple tools together, forming a plan (e.g., "First, search the web for the weather; then, use the email tool to send a summary").



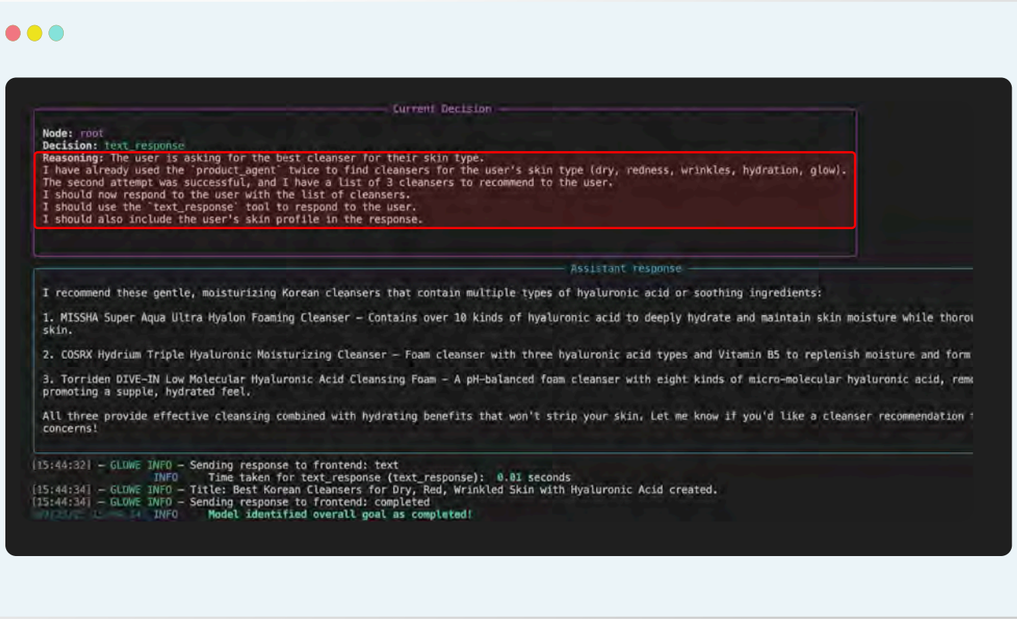
Here, the decision agent correctly analyzed the incoming request and selected the product_agent tool.

3. **Argument Formulation (Action)**: Once a tool is selected, the agent must figure out what arguments to pass to it. If the tool is get_weather(city, date), the agent needs to extract "San Francisco" and "tomorrow" from the user's query and format them correctly. This could also be a structured request or API call with the necessary information to use the tool.



In this case, the product_agent required a text query for searching the products collection. Notice how the agent also corrected itself (self-healing) after generating an ill-formed argument that initially caused an error (another key piece of orchestration).

4. **Reflection (Observation)**: After executing the tool, the output (the "observation") is fed back into the context window. The agent then reflects on this output to decide its next step. Was the tool successful? Did it produce the information needed to answer the user's query? Or did it return an error that requires a different approach?



As you can see, orchestration happens through this powerful feedback loop, often called the Thought-Action-Observation cycle. The agent observes the outcome of its action and uses that new information to fuel its next "thought," deciding whether the task is complete, if it needs to use another tool, or if it should ask the user for clarification.

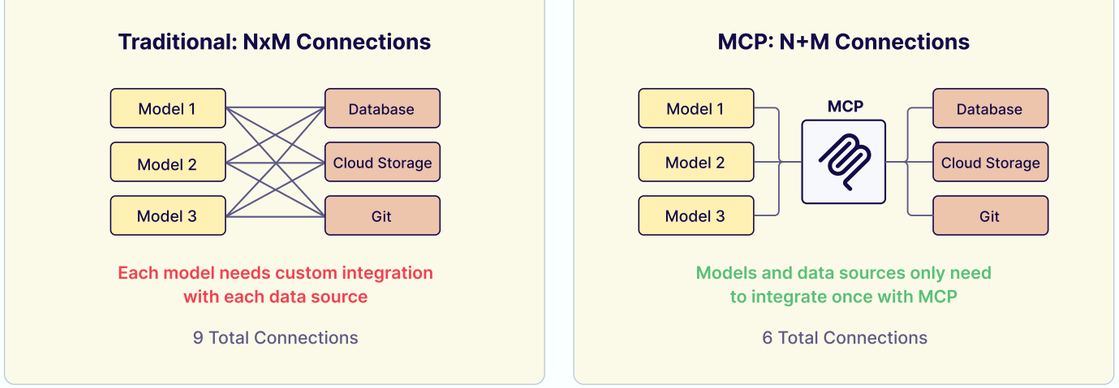
> **This Thought-Action-Observation cycle forms the fundamental reasoning loop in modern agentic frameworks like Elysia.**

The Next Frontier of Tool Use

The evolution of tool use is moving more and more towards standardization. While function/tool calling works well, it creates a fragmented ecosystem where each AI application needs custom integrations with every external system. The Model Context Protocol (MCP), introduced by Anthropic in late 2024, addresses this by providing a universal standard for connecting AI applications to external data sources and tools. They call it "USB-C for AI" - a single protocol that any MCP-compatible AI application can use to connect to any MCP server.

So, instead of building custom integrations for each tool, developers can just create individual MCP servers that expose their systems through this standardized interface. Any AI application that supports MCP can then easily connect to these servers using the JSON-RPC based protocol for client-server communication. This transforms the MxN integration problem (where M applications each need custom code for N tools) into a much simpler M + N problem.

Traditional Integration vs MCP Approach

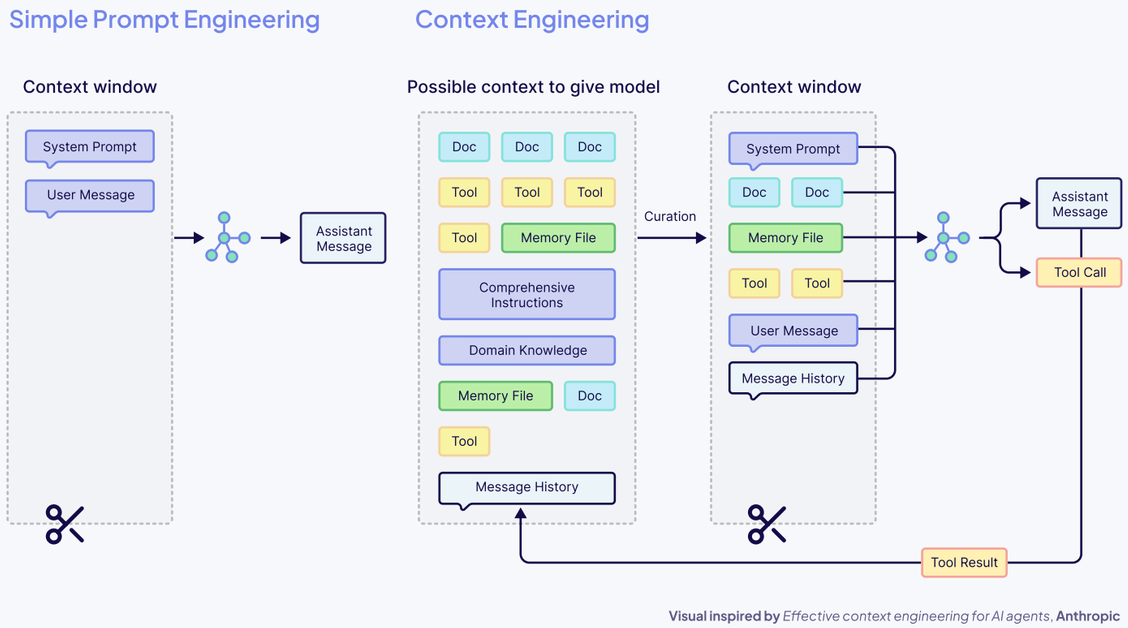


Visual inspired by : *https://humanloop.com/blog/mcp*

This shift towards composable, standardized architectures, where frameworks enable developers to build agents from modular, interoperable components, represents the future of AI tooling. It changes the engineer's role from writing custom integrations to orchestrating adaptive systems that can easily connect to any standardized external system.

Summary

Context engineering is about more than just prompting large language models, building retrieval systems, or designing AI architectures. It's about building interconnected, dynamic systems that reliably work across a variety of uses and users. All the components described in this ebook will continue to evolve as new techniques, models, and discoveries are made, but the difference between truly functional systems and the AI apps that fail will be how well they engineer context across their entire architecture. **We are no longer thinking in terms of just prompting a model, we're looking at how we architect entire context systems.**



Visual inspired by *Effective context engineering for AI agents*, **Anthropic**

weaviate

Context engineering is made up of the components described in this ebook:

- **Agents** to act as the system's decision-making brain.
- **Query Augmentation** to translate messy human requests into actionable intent.
- **Retrieval** to connect the model to facts and knowledge bases.
- **Memory** to give your system a sense of history and the power to learn.
- **Tools** to give your application hands to interact with live data and APIs.

We are moving on from being prompters who talk *to* a model and instead, becoming architects who build the world the model lives *in*. We - the builders, the engineers, and the creators - know the truth: the best AI systems aren't born from bigger models, but from better engineering.

We can't wait to see what you build 💙

Glossary