O'REILLY®

Early Release

RAW & UNEDITED

An Illustrated Guide to AI Agents

Visual Intuition for Reasoning, Multimodal, and Diffusion LLMs

Maarten Grootendorst
& Jay Alammar

An Illustrated Guide to AI Agents

Visual Intuition for Reasoning, Multimodal, and Diffusion LLMs

Maarten Grootendorst and Jay Alammar

An Illustrated Guide to AI Agents

by Maarten Grootendorst and Jay Alammar

Published by O’Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield

Development Editor: Michele Cronin

Production Editor: Katherine Tozer

December 2026: First Edition

Revision History for the Early Release

- 2025-10-15: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798341662698 for release details.

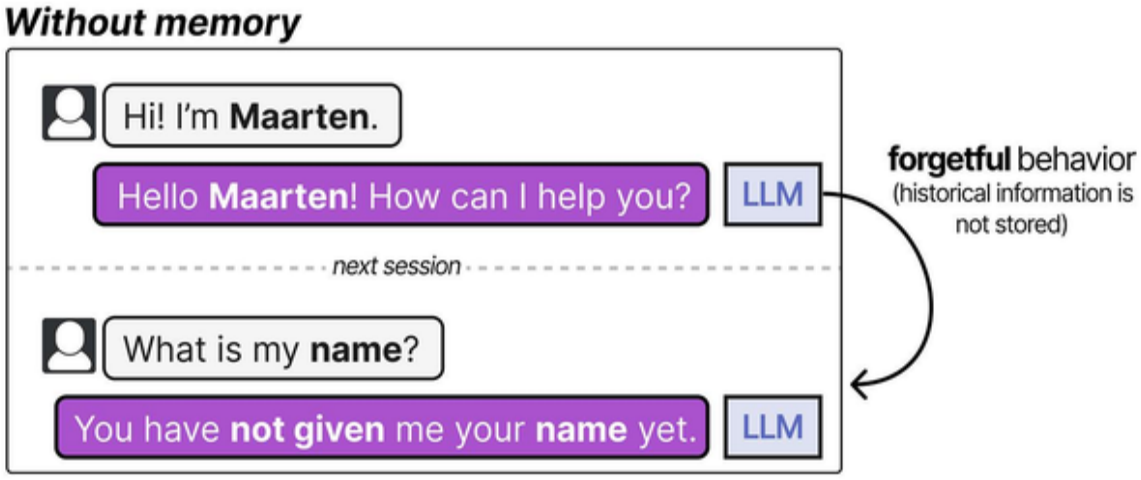
979-8-341-66264-3

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Memory

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *mcronin@oreilly.com*.

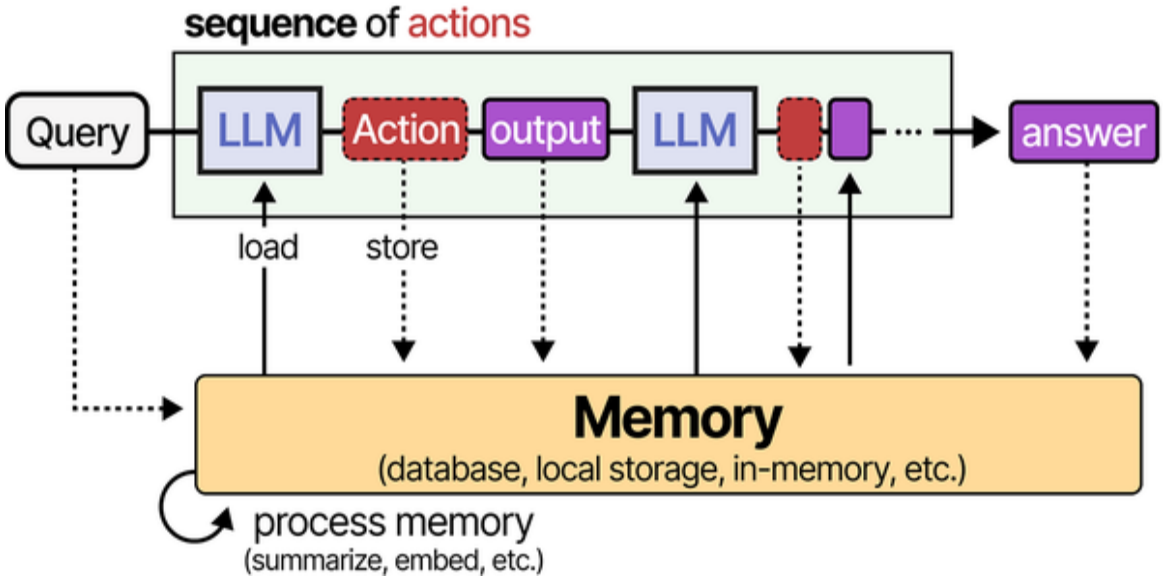
Among all the added modules to the augmented LLM, memory is a key component needed to go from an LLM to an Agent. By themselves, LLMs are forgetful entities; they do not remember past conversations, nor do they have access to all actions they have taken. If you were to locally load up an LLM and ask it to remember your name, it can't–not without explicitly giving it memory. In contrast, the interactions that you might have with hosted LLMs, like ChatGPT and Claude, are not regular LLMs. Rather, they are LLMs augmented with modules like memory and tools. Figure 4-1 illustrates this forgetfulness well, as it demonstrates the interaction you might have with a regular LLM. As such, LLMs are stateless, and information is not persisted across calls.



Over the years, there has been significant attention to aspects of agents like tool usage, reasoning LLMs, and multi-agent collaboration. Each is quite important by itself, but don't underestimate the importance of memory. Without memory, a personal assistant agent wouldn't be able to remember past conversations. Without memory, a coding agent wouldn't understand your entire codebase. Without memory, an Agent would forget that it has already taken a given action and keep on repeating it.

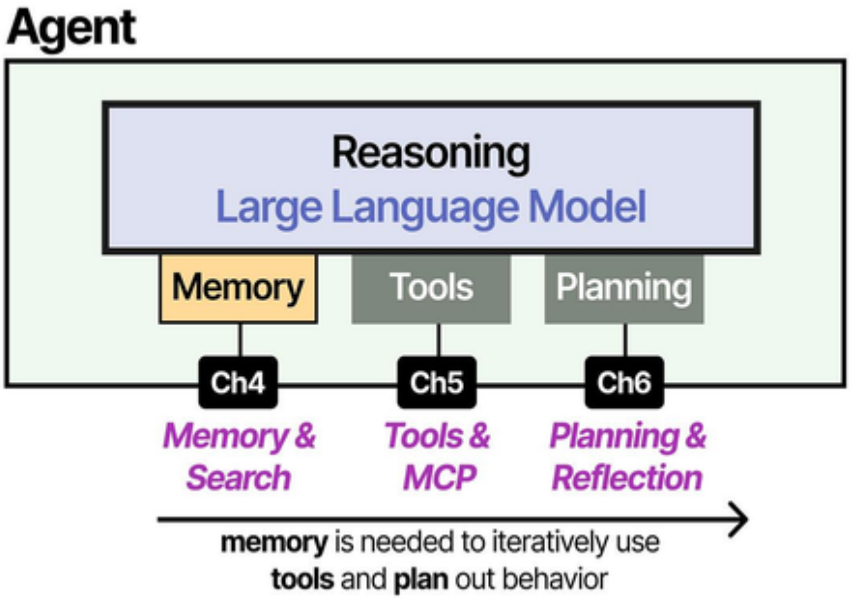
Memory can be quite difficult to define. From a narrow view, it relates to all historical information during the execution of an Agent. In this chapter, however, we take a broader perspective. Memory relates not only to all past actions of an Agent, but also external information beyond the agent-environment interactions. A coding agent's memory would not only consist of the actions it has taken to fix your bugs, but also your hosted documentation and issues pages.

Memory is not only the act of remembering information but also storing newly generated information. Likewise, often a choice has to be made on which information to store and how, and which information to remember. All these methodologies and choices have important implications on the Agent's behavior. As shown in Figure 4-2, updating and using memory is an iterative process that requires careful handling of intermediate information.



Memory allows the agent to remember past errors and failed experiences, so it can be more effective for handling similar tasks in the future. As such, memory enables Agents to learn and evolve as they remember more experiences. By interacting with the environment and storing the feedback, Agents learn from their previous experiences. Memory is, therefore, application-specific, and implementations may not only decide *what* to remember but also *how* it is remembered.

Throughout this chapter, we will explore many types of memory modules, ranging from short-term and long-term memory to external memory modules, through methods like (agentic) retrieval-augmented generation. Seen in Figure 4-3, it forms the foundation of the LLM. After all, how would an LLM be able to use tools or create plans if it keeps forgetting them?



Types of Memory

Memory for LLMs tends to follow human memory types, as the agents that we attempt to create are often modeled after human behavior. Instead of going through all forms of human memory types, which there are many, the "Cognitive Architectures for Language Agents" paper describes four types of external memory that we often see back in Agents, namely short-term working memory and long-term memories: episodic, semantic, and procedural memory.[1]

Working memory is a type of short-term memory that is typically defined as a system with limited capacity that temporarily holds information that we need for things like decision-making and reasoning. For LLMs, it is typically data that persists across LLM calls. More specifically, it is the chat history of the LLM that is being stored in memory and continuously fed back to the LLM.

For long-term memory, there are three forms described:

Episodic memory

> Involves remembering specific events and experiences from one's past (e.g., your last birthday party). For agents, this typically involves

specific actions the agent has taken thus far and their outcomes.

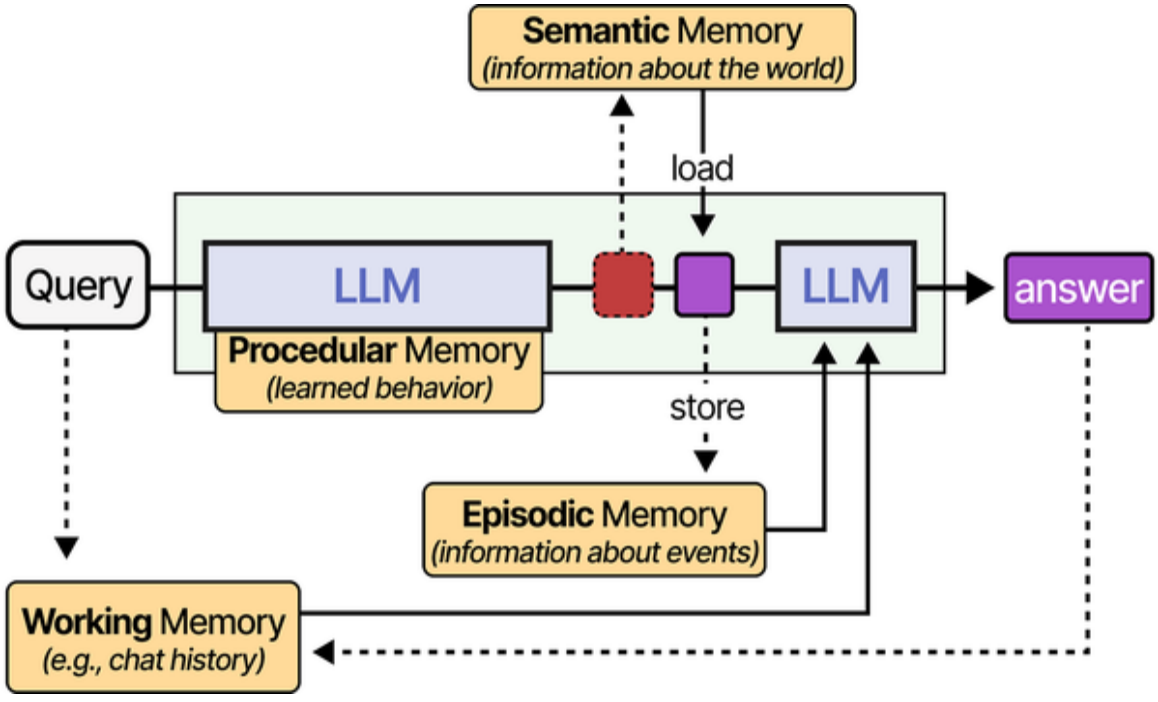
Semantic memory

Involves remembering knowledge about the world (e.g., the capital of France). For agents, this may involve querying an external database like Wikipedia or the codebase that you are working on.

Procedural memory

Involves remembering patterns of how to do things (e.g., writing code in Python). For agents, this can be information hidden in its parameters (also called parametric memory) or the system prompt, which persists across calls.

Figure 4-4 shows an example of how these different forms of memory can be used and interpreted during a single agent's session.



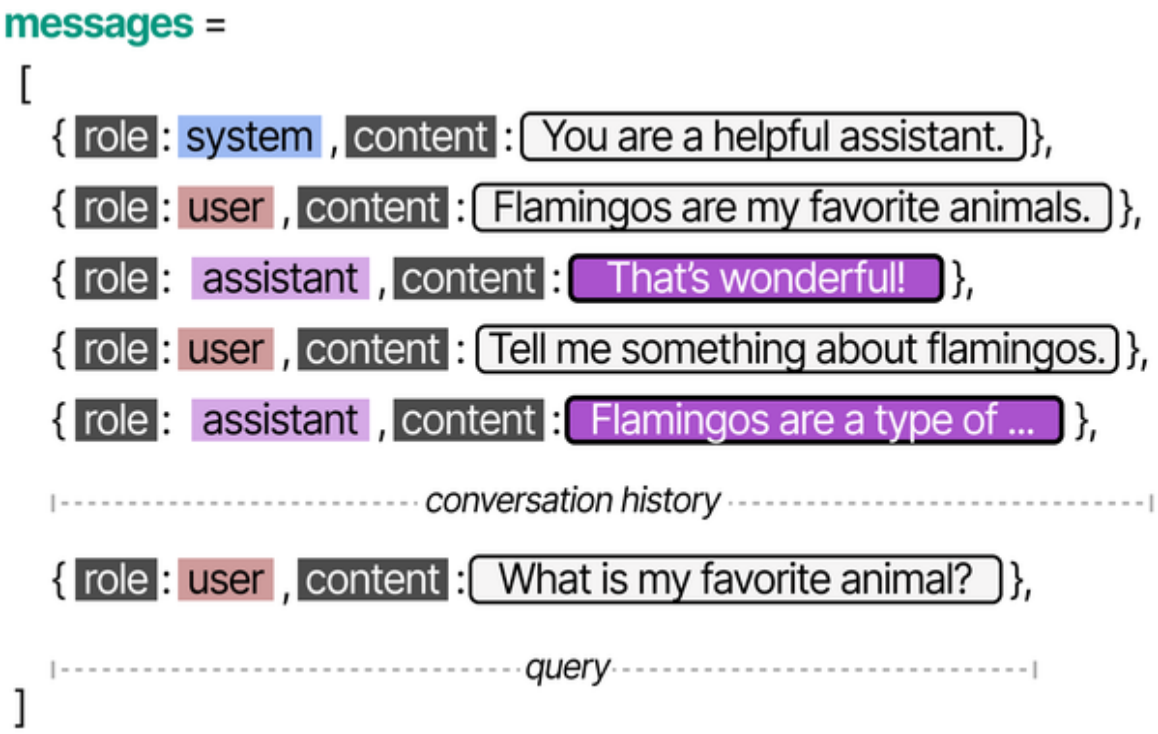
As briefly mentioned previously, we can also consider the type of memory that the model already has, *parametric memory*.[2] Without any memory modules, LLMs are trained to a certain extent to retain information. If you

ask an LLM what the capital of France is, most LLMs will correctly remember that it is Paris. The answer is therefore contained within the parameters of the model and attempts to retrieve it. Although a relatively new field, it is technically possible to instill information into an LLM through supervised fine-tuning. Note that this is not a stable method, as we are not entirely sure beforehand which information gets retained explicitly and which information is incorrectly reconstructed.

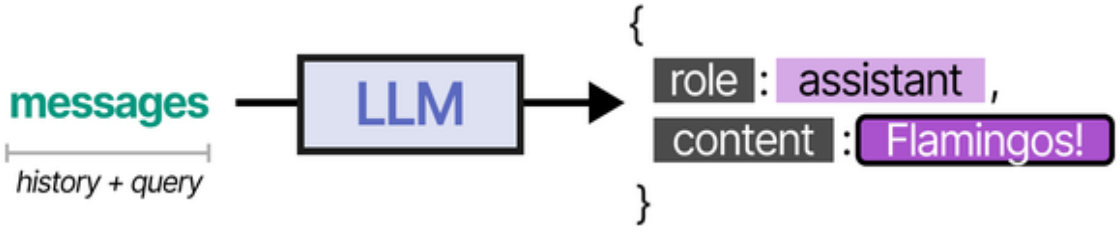
Although these memory types may differ, they may not always be stored as such. Depending on the specific implementation, they could be seen as one big pile of information or separated into different databases to be remembered for specific tasks and actions.

Short-Term Memory

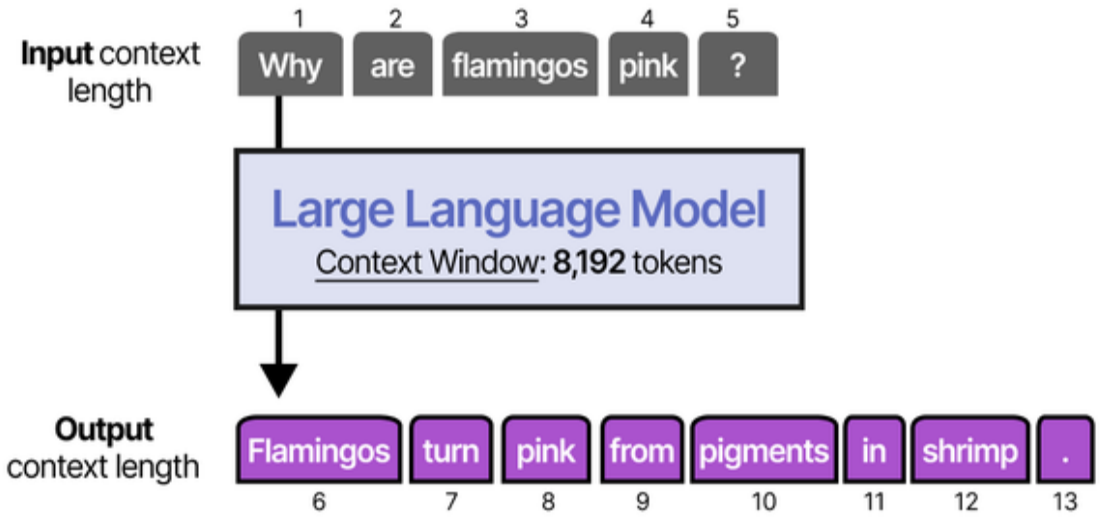
Short-term memory in Agents is the information it has about recent interactions, typically the ongoing conversations with the user or the behavior of the LLM. In practice, this is the conversation history of the LLM, which serves as context for generating responses. Illustrated in Figure 4-5, they are generally formatted as messages demonstrating the differences between system prompts, the user's query, and the LLM's answer. It is a conversation where the user tells something about themselves, namely that they love flamingos.



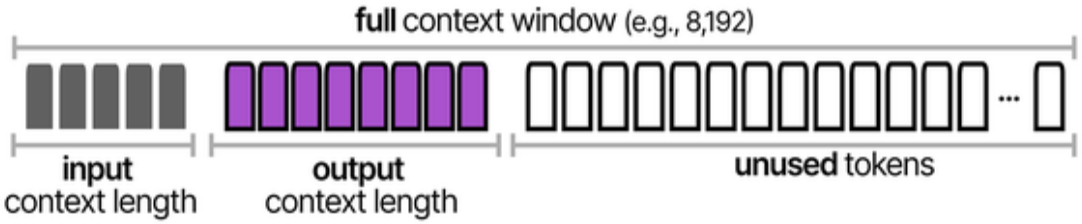
These messages are updated every time the user or assistant replies and are fed back into the LLM as input for the next query, as shown in Figure 4-6. The query ("What is my favorite animal?") does not trigger the LLM to recall the past on its own. Rather, it is provided with the entire conversation history, which contains the relevant information, namely that the user loves flamingos. In other words, the LLM does not truly "remember" past conversations but is instead told what the conversation was by explicitly inserting it into the prompt.



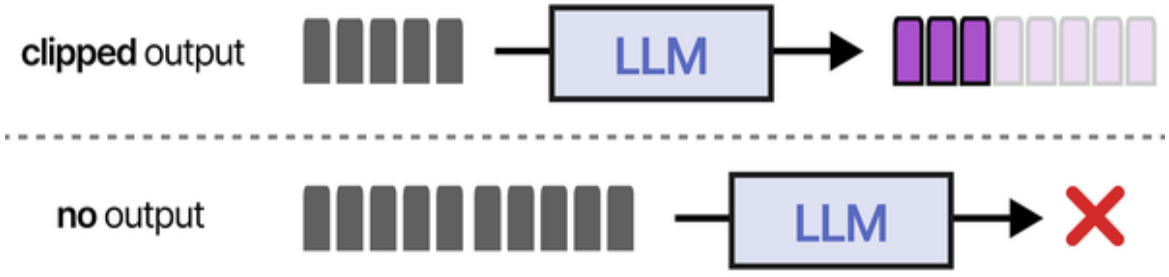
Most LLMs have a limited context window, which is the number of tokens that the LLM can process both and combine both the input and the output tokens as seen in Figure 4-7.



In this example, a short query and answer are given which total 13 tokens out of a potential 8,192. Seen in Figure 4-8, there are many potential tokens that are left unused and could potentially be filled with additional information or reasoning tokens as discussed in Chapter 3.

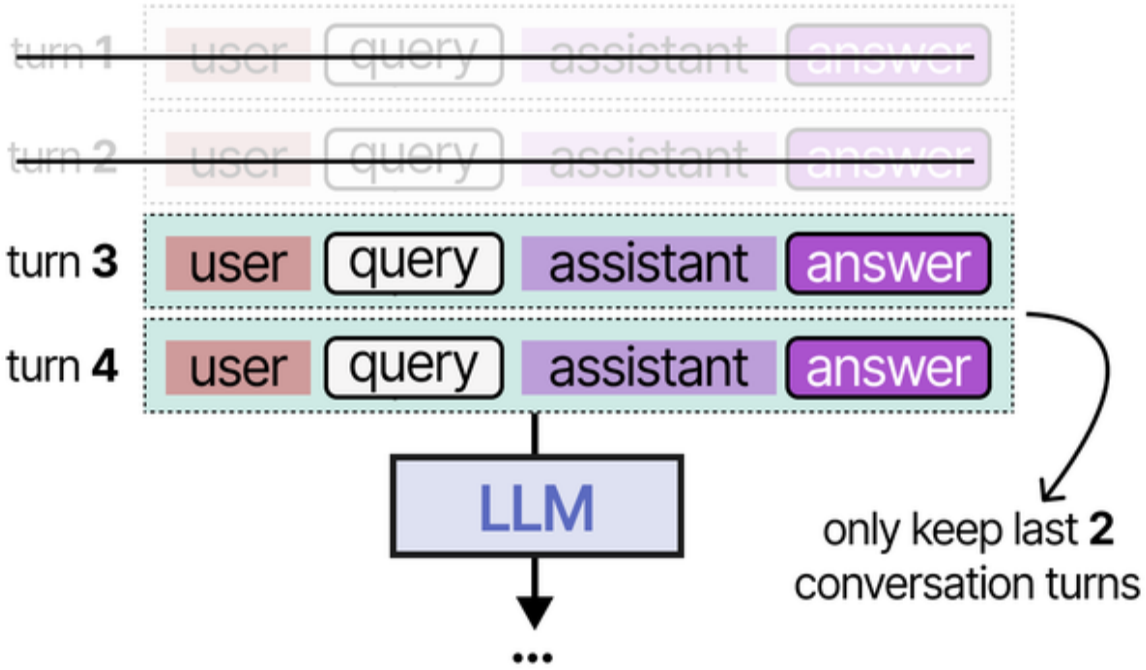


However, as the conversation history grows, so do the number of tokens. Eventually, and as shown in Figure 4-9, if the conversation history gets too large, it will not fit within the context windows. This might cut the answer short or even prevent the LLM from processing the prompt at all.

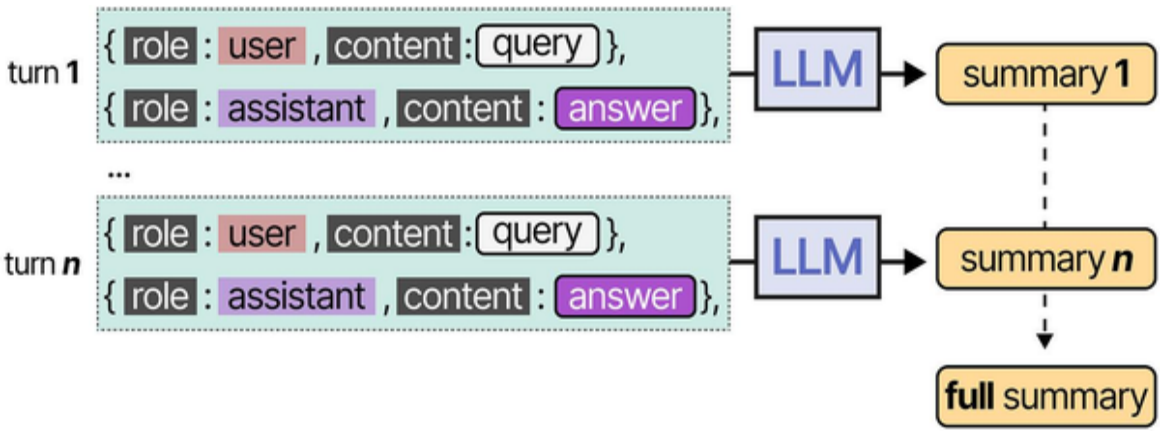


Moreover, the more information put in the prompt, the more difficult it will be for the LLM to attend to everything.[3] As a result, we cannot always put the entire conversation history into the prompt of the LLM. Instead, there are several techniques we can use to provide the conversation history without filling up the context window.

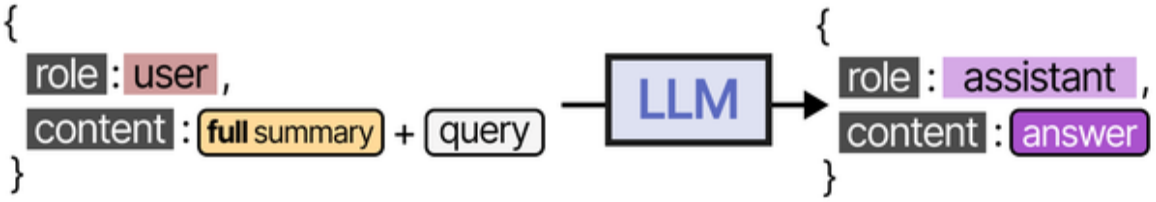
The first technique for efficient short-term memory is rather straightforward: trimming the messages as they grow. Whenever the number of messages grows too large for the LLM's context window to handle, we can decide to simply remove the first few interactions until it fits that window. Seen in Figure 4-10, this might remove quite a lot of information that may or may not be relevant to future queries.



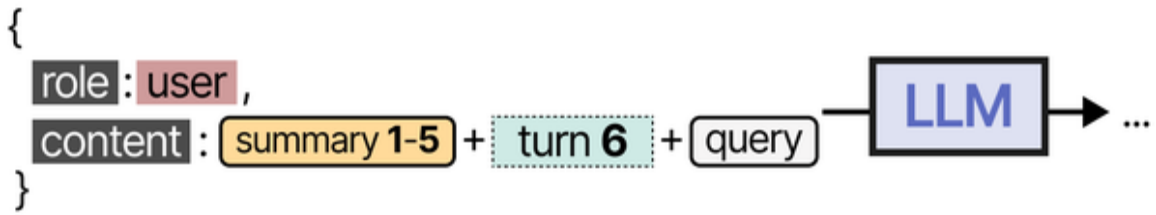
Instead, a common technique is to employ another LLM to summarize the conversation history. After each conversation turn, the same or another LLM will summarize it and add it to the full summary of the conversation (Figure 4-11).



As illustrated in Figure 4-12, the created summary will be shown together with the query for the LLM to answer. This summary might still fill the context window over time as summaries are stacked on one another, but it is much slower than filling the context window with the raw conversation history.



Stacking summaries is not the only method of summarization. Instead of adding a summary of the most recent query/answer pair each time, you can instead summarize the conversation history of the last 5 conversations. Likewise, you can decide to maintain one summary and ask the LLM to update it after each conversation. Figure 4-13 illustrates such a method where conservation turns 1 through 5 are summarized, but not turn 6.



Although it may seem straightforward, maintaining the conversation history can be a difficult task and requires understanding what is important: the entire history, the recent history, or a summarized variant? For short conversations, maintaining the entire history would work, but that might not be the case for long sequences of actions.

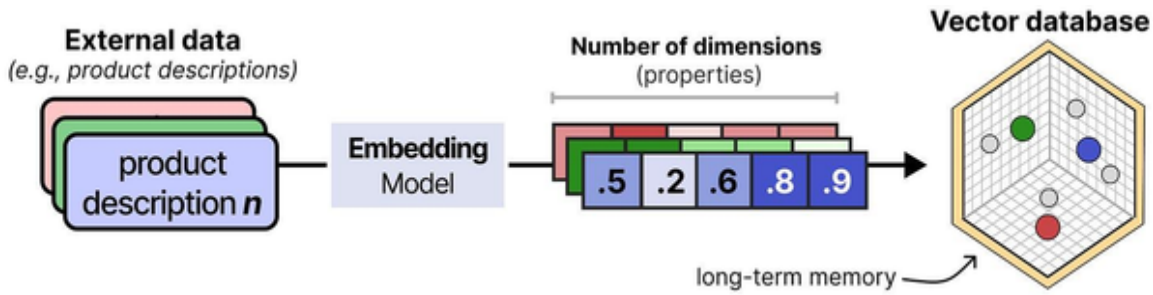
Long-Term Memory

As the conversation history grows and the actions that an agent has taken, so does the need for long-term memory. Long-term memory typically involves maintaining one or more external databases that can be queried to extract additional information. This can contain information about previous traces or states of the agent (episodic memory) or information unrelated to the agent's behavior but about the context of your application instead (semantic memory), like your organization's documents.

Retrieval-Augmented Generation (RAG)

Arguably, the most common method for giving your agent, or any LLM for that matter, long-term memory is Retrieval-Augmented Generation (RAG).[4] RAG typically consists of two stages: *ingestion* and *inference*.

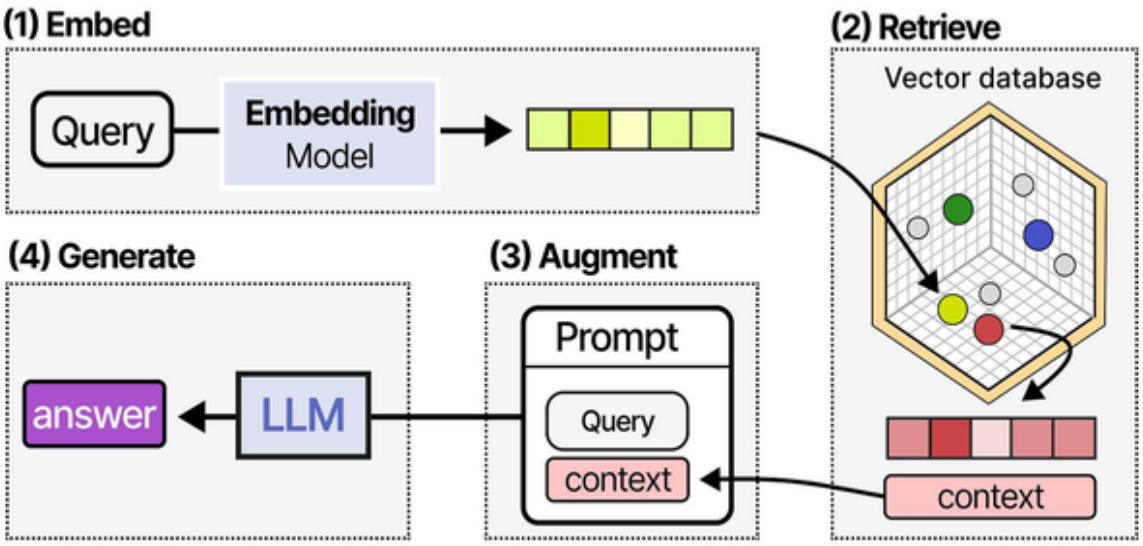
In ingestion, your external data, typically unstructured text, is embedded into numerical representations and stored in a database (see Figure 4-14). To create these representations, a special variant of an LLM is used, an embedding model. This embedding model is trained to create numerical representations such that words and phrases with similar meaning will have similar representations. This external database can be considered the long-term memory of the LLM, which can be queried for relevant information.



Inference with RAG consists of four steps. In step 1, the user's query is embedded using the same model as was used for embedding the external data. In step 2, the embedded query is compared to the external database, and the most relevant items in the database to the query are extracted.

Defining relevancy in RAG systems can mean many things. In our example, it means the similarity between the query embeddings and the external embeddings. This similarity can be defined through the embeddings we created, but hybrid systems are also possible where embeddings are combined with traditional Bag-of-Words-like approaches.

In step 3, the relevant items and the user's query are combined into the prompt. This step is meant to provide the model with context for the generation step. Essentially, you tell the LLM that you have contextual information that it can use to derive its answer. Finally, in step 4, the augmented prompt is used by the model to generate the output. The added contextual information should generally result in more accurate and relevant responses, assuming that the contextual information is indeed relevant and correct. Figure 4-15 illustrates the four steps of inference with RAG.

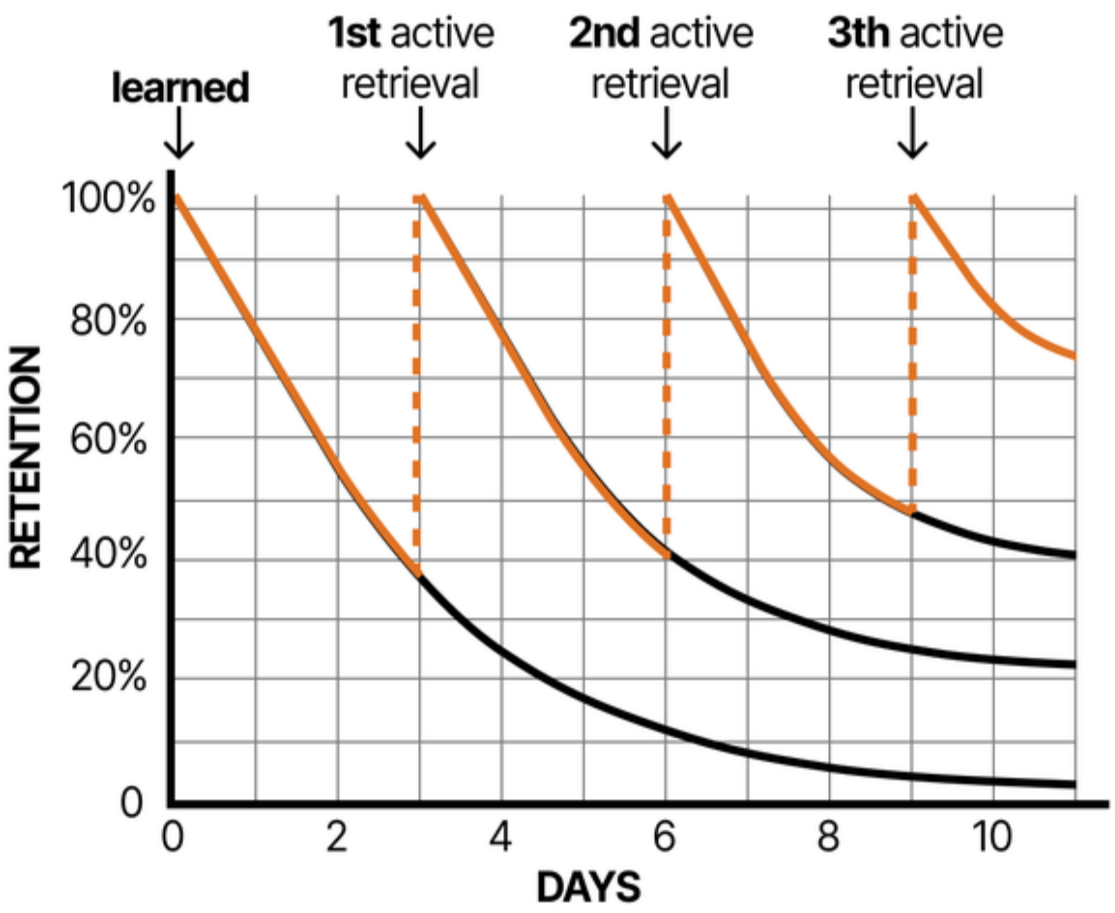


RAG is often used to minimize hallucination, which refers to the tendency of LLMs to confidently produce an answer that is actually incorrect. By providing the LLM with external information (which you assume to be true), the LLM is less likely to “make up” information.

MEMORYBANK

An interesting take on RAG for ChatBots is MemoryBank, a mechanism that allows LLMs to recall relevant memories as a long-term mechanism (external database) rather than a short-term mechanism (conversation history).[5] Its experiences through conversations are stored in a separate database that allows the LLM to retrieve relevant memories. What sets it apart from regular RAG is that this memory is continuously updated to selectively preserve memory through an updating mechanism.

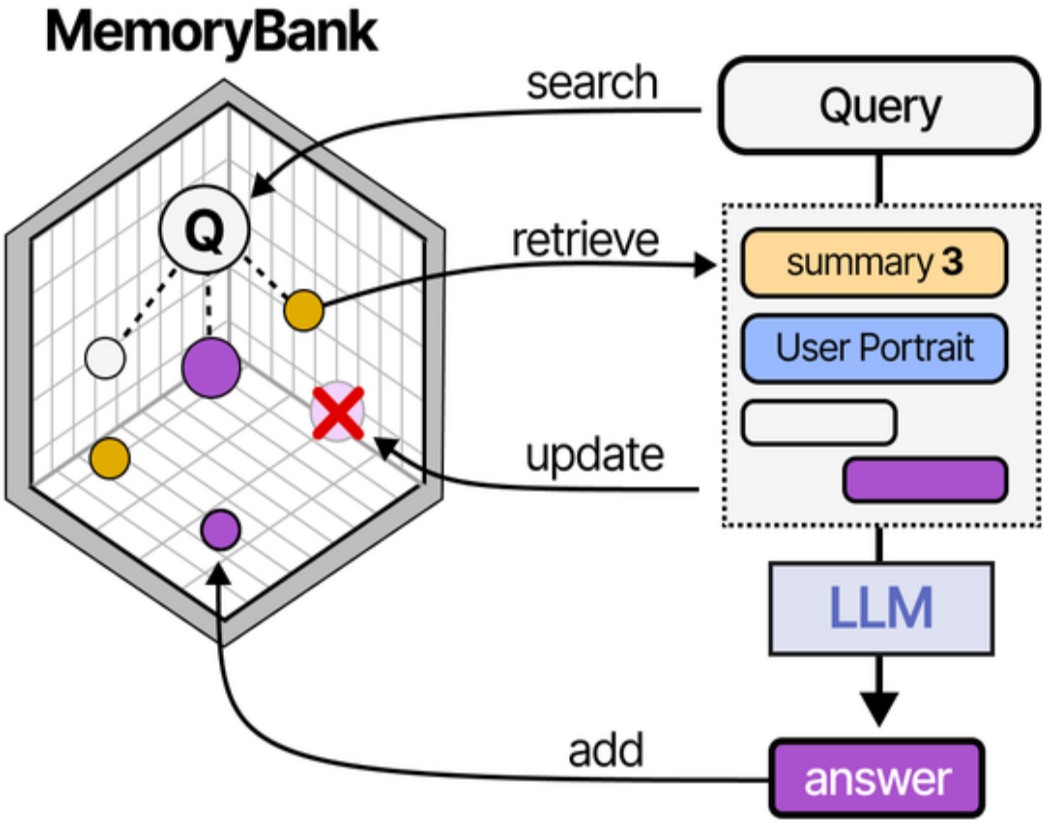
This mechanism allows the MemoryBank to forget and reinforce memory inspired by the Ebbinghaus Forgetting Curve theory, which is a curve demonstrating the pace at which we tend to forget. The curve is often shown as being exponential, resulting in a loss of half of what we learn each day. A common way to prevent forgetting what you learnt, for instance when preparing for exams, is to actively recall the learned information frequently. This is referred to as spaced repetition, which tends to decrease the pace at which knowledge is forgotten. Figure 4-16 illustrates this knowledge decay and the effect of spaced repetition on this decay.


learned
1st active retrieval
2nd active retrieval
3th active retrieval
RETENTION
100%
80%
60%
40%
20%
0
0
2
4
6
8
10
DAYS

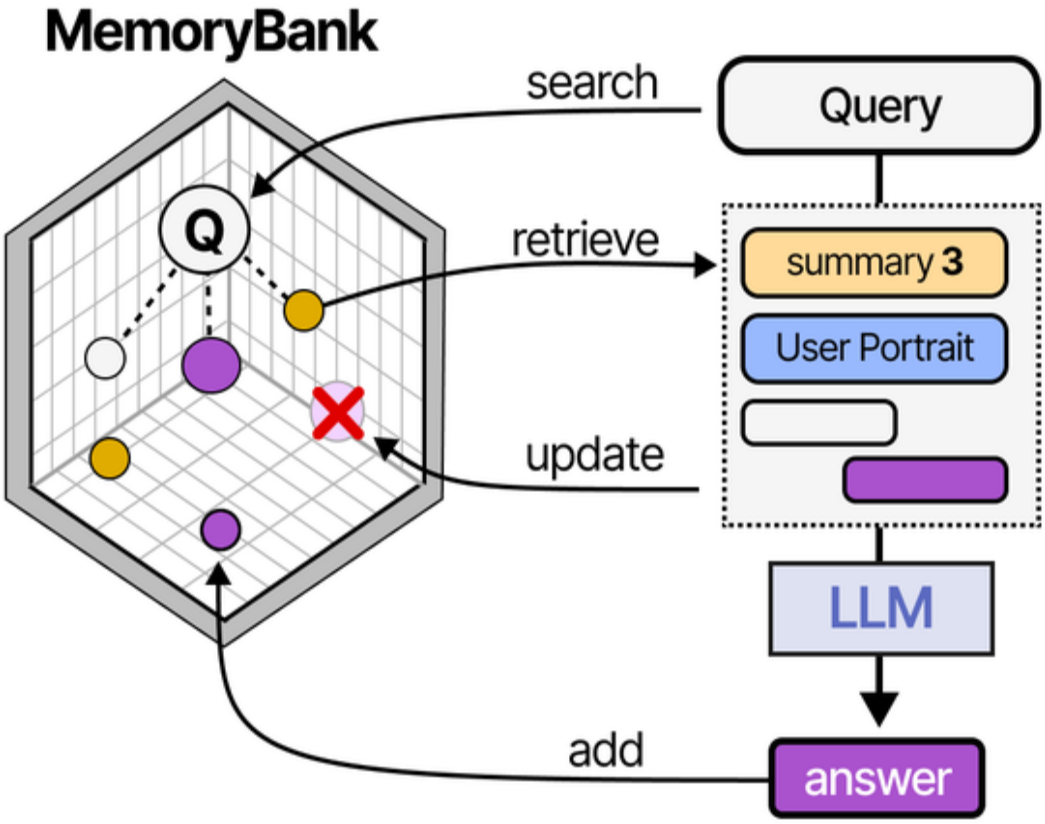
MemoryBank borrows from this theory and frequently updates the long-term memory of an LLM based on which pieces of knowledge are (not) accessed. Specifically, this means that when a memory item is retrieved and used during conversations, it will persist longer in the MemoryBank. However, if the memory item hasn't been retrieved for a while, then there is a chance the memory will be removed entirely.

The authors use a few variants of memory (Figure 4-17):

- *Conversation history* — Raw multi-turn conversations
- *Summaries of past events* — These are generated by an LLM based on the conversation history
- *User's portrait* — The personality traits and emotions of the user as summarized by the LLM based on the conversation history



The summaries and conversation turns are embedded so that they can easily be retrieved. The user portrait is dynamically updated and always passed as additional context. Figure 4-18 shows a full overview of this pipeline. When a query is created, it is embedded, and related conversation turns and summaries are retrieved, together with the user portrait. When conversation turns are retrieved, their strength is updated, making them less likely to be removed from the MemoryBank. The retrieved context, together with the query, is used as input for the LLM to generate an answer.

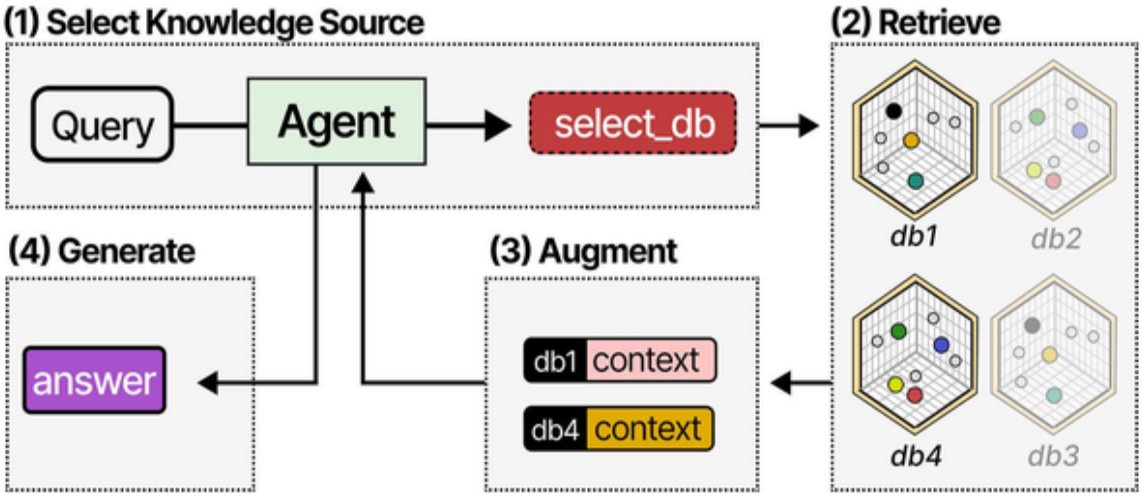


This form of memory demonstrates the potential complexity of RAG-like applications, where each use case necessitates different types of memories, summarizations, etc. As such, there are many forms of RAG, like Graph RAG and Multimodal RAG, that each require its own set of considerations. The RAG examples shown previously are often referred to as vanilla RAG or naive RAG for their straightforward implementation.

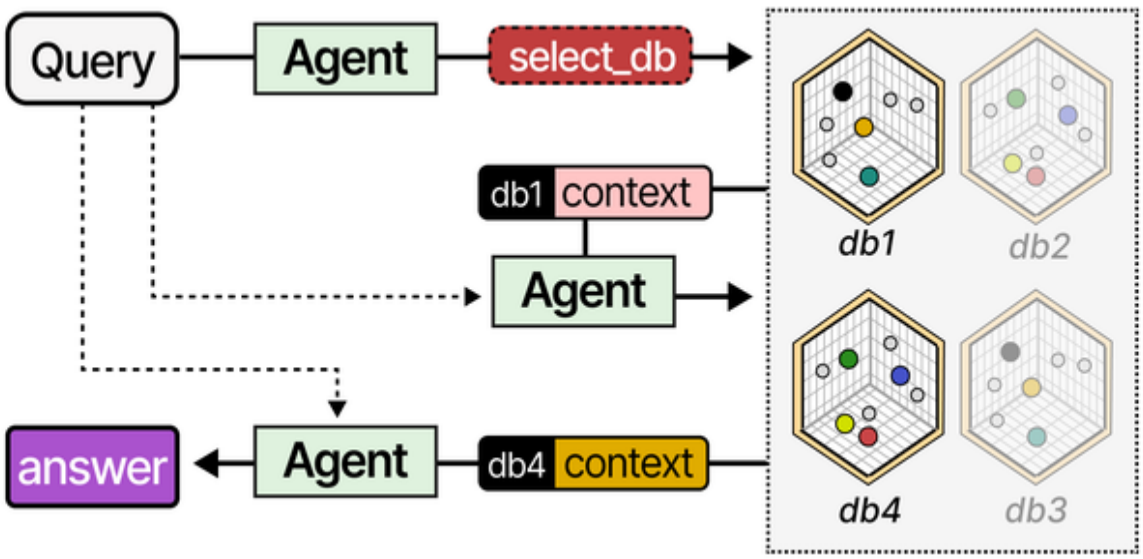
Agentic RAG

In vanilla RAG, the vector database can be considered the long-term memory of the LLM. However, the LLM is only *given* information that is relevant to the query and has no agency over what is being retrieved.

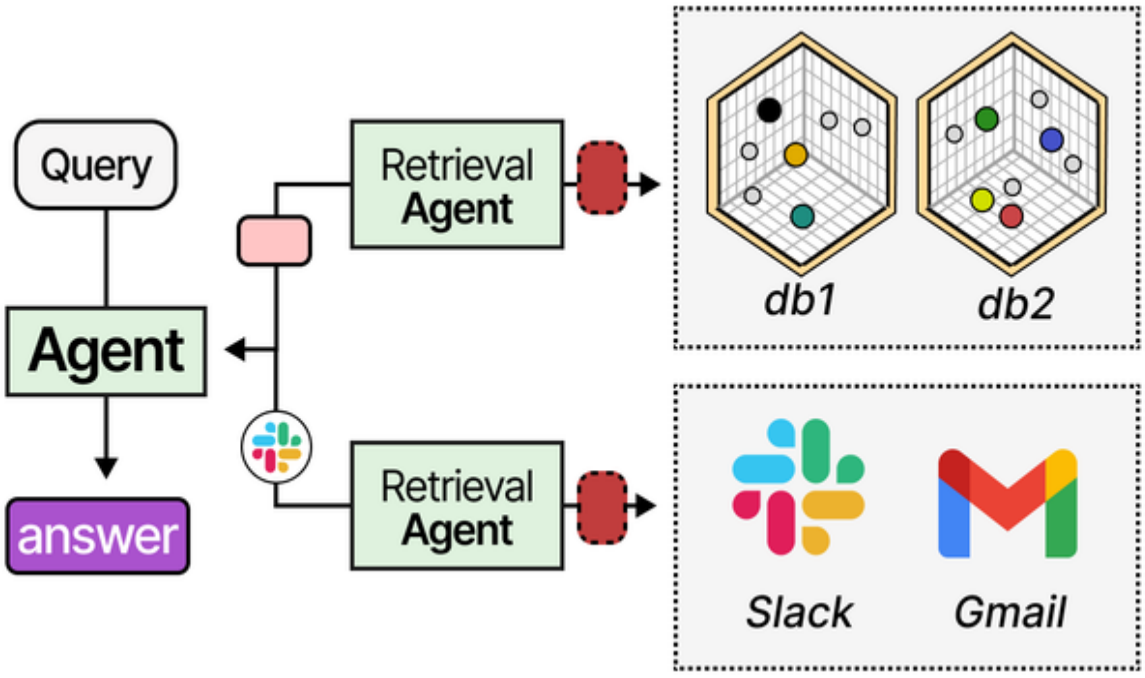
In agentic RAG, there is an agent instead of an LLM that can access the external database as a tool and have control over which information it retrieves. Agency over what is being retrieved is given back to the agent, who typically has access to one or more external databases of information. Figure 4-19 illustrates this idea of having the agent select from which source to retrieve contextual information before deciding whether to generate the answer or to again retrieve additional information.



In single-agent systems, agentic RAG is a router where you have several external knowledge sources, and the agent decides which one(s) to use. You are essentially adding all these knowledge sources and databases as tools instead of being a static step that runs before the LLM generates output. Moreover, such an agentic RAG system does not always have to run the agentic RAG based on the query itself. As seen in Figure 4-20, it may decide to extract information from one search and then run a subsequent search based on that information in another database.



Agentic RAG does not limit itself to single-agent systems. In Multi-Agent RAG systems, multiple agents have the capabilities to extract information from external sources. Oftentimes, you have smaller retrieval agents that are being coordinated by a single agent with more capabilities. These smaller retrieval agents are each specialized in extracting specific information or working with specific knowledge sources (see Figure 4-21). Note that it does not always have to be a vector database as an external knowledge source; it can also be a web search or querying some API for information (like your Slack or Gmail).



In other words, instead of querying the vector database through a static step only once, by hooking it as a tool, the agent can dynamically decide how many times it needs to query the semantic memory until it has enough context to answer a given query. Note how we discussed LLMs in the context of RAG but agents in the context of Agentic RAG instead. It demonstrates the agency and autonomy in accessing their respective RAG capabilities.

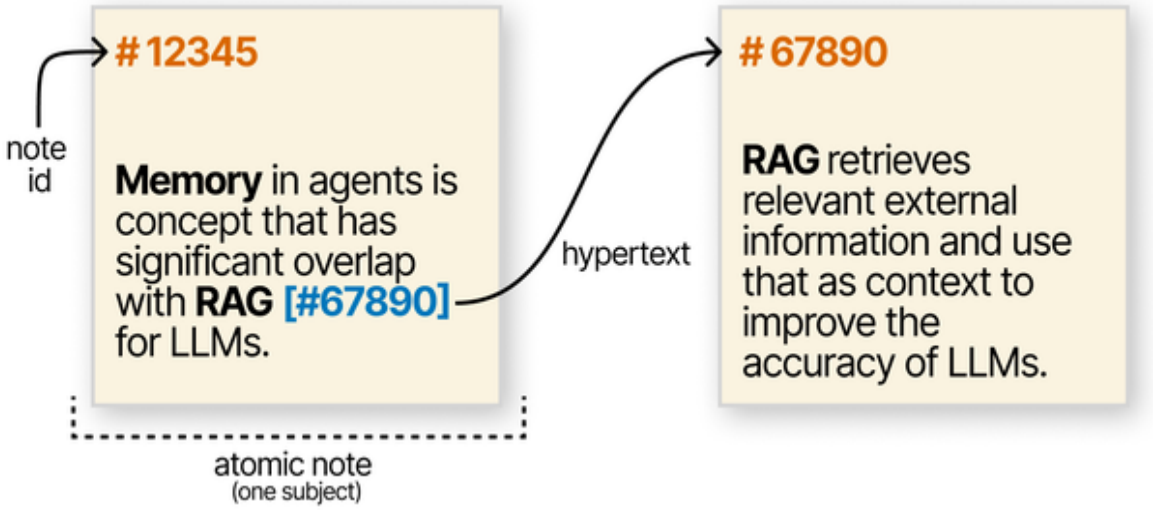
Let's go over some examples of how these agentic RAG systems work through impactful papers and implementations in the field.

A-MEM

An interesting approach to agentic RAG is A-MEM, an agentic memory system derived from the notetaking method known as Zettelkasten.[6] Zettelkasten approaches note-taking as having three important components, namely atomicity, hypertextual notes, and personalization.

Atomicity means that each Zettel (a note) should only contain one unit of knowledge, referred to as an atom. This note could, for example, contain a brief description of how memory works in agentic systems.

Then, hypertextual notes refer to the idea that all notes refer to each other and may explain or expand on each other's content. For instance, the previously created note can be connected to another note that has some information about RAG. Since both are memory systems, they are likely to be related. The ideas of atomicity and hypertextual notes are illustrated in Figure 4-22. Together, they may create an interconnected web of notes, ideas, and may create larger topics of interconnected notes.

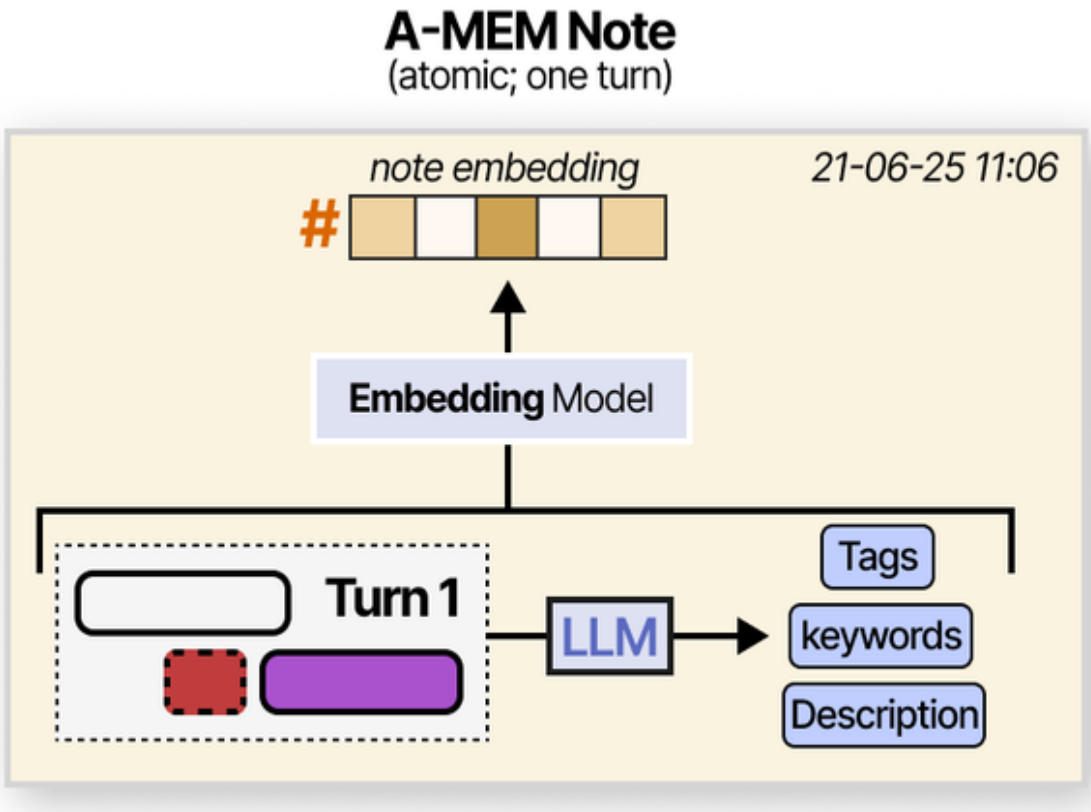


A-MEM uses this idea of note-taking to agentic memory by creating these interconnected notes. In the context of agents, each note contains the following information and can be considered a piece of memory:

- The original *interaction* with the environment (i.e., one turn)
- The *timestamp* of the interaction
- LLM-generated *keywords* that capture key concepts
- LLM-generated *tags* to categorize the interaction
- LLM-generated *contextual description*

By focusing on a single unit, namely a single interaction, A-MEM adheres to the principle of atomicity.

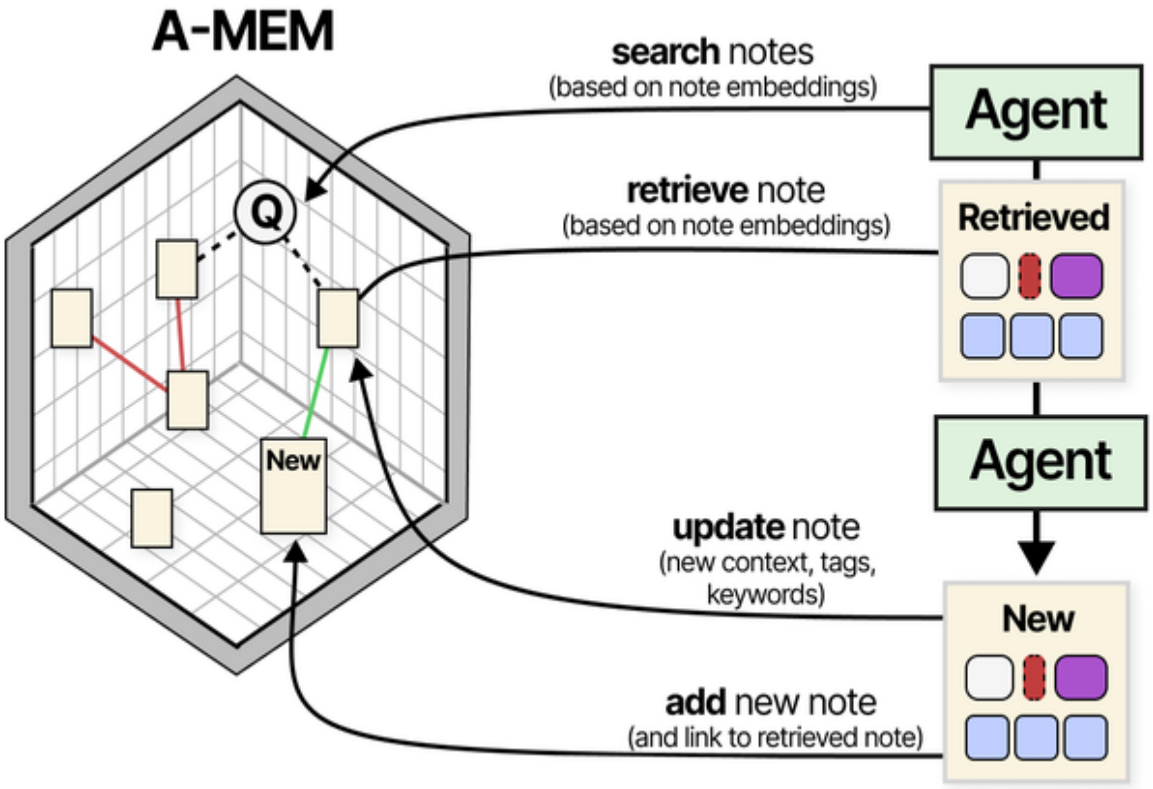
Then, all pieces of information are embedded so that they can be used to later on easily retrieve related information. Note that all information, except for the timestamp, is concatenated so that a single embedding is created for the entire note/memory (Figure 4-23).



Interestingly, the authors use this generated note embedding as one of the main IDs of the note. To link this note to other memories, they run a similarity search between this note's embeddings and all other memories and extract the top-k memories. After doing so, the LLM is asked to decide which of these candidate memories should be linked to the newly added memory.

After the memory is added and linked to other memories, the LLM is prompted to update the LLM-generated tags, keywords, and description based on the newly added memory. This results in an evolutionary approach where newly added memories are linked to older memories, which are in

turn updated to be in line with the newly added memories. Figure 4-24 illustrates this ever-evolving database of notes in the form of memories.



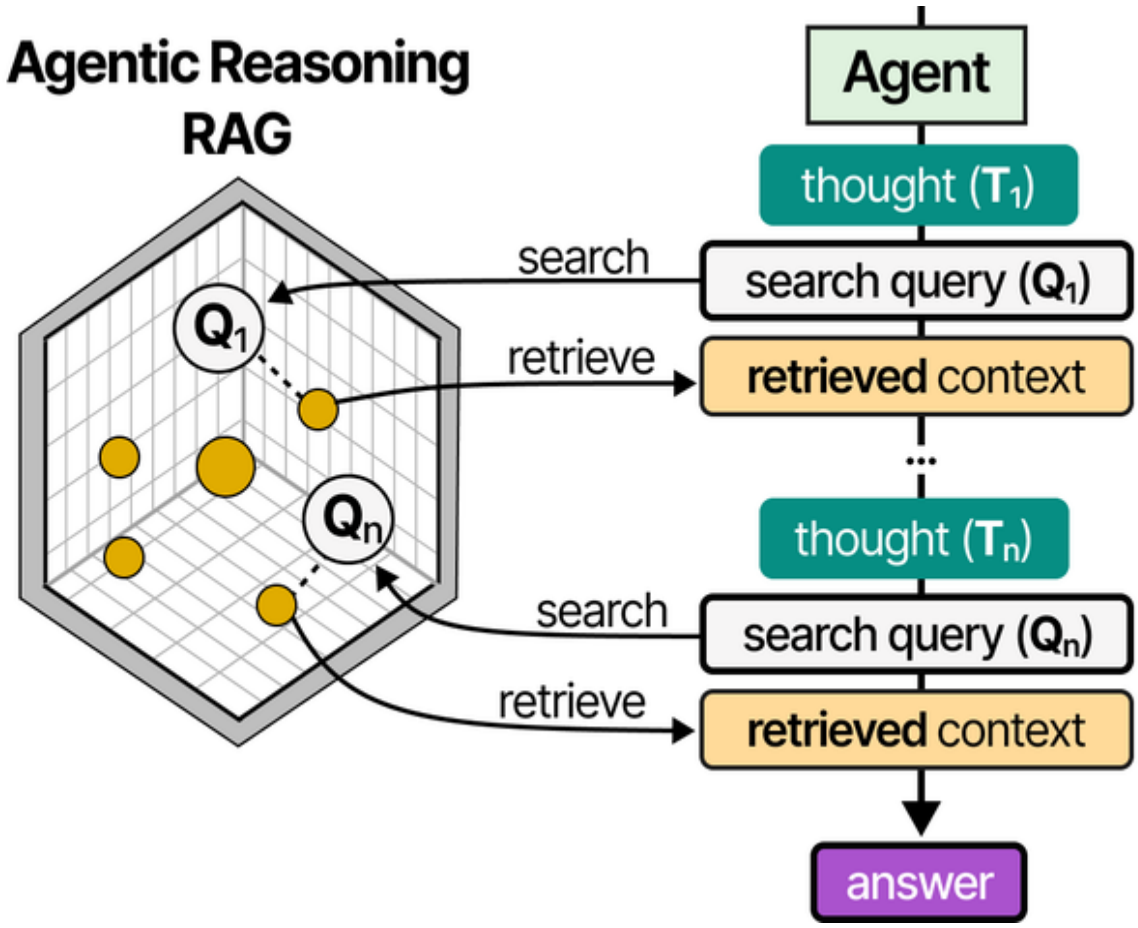
This agentic RAG system allows the agent to access the A-MEM and search for memories that relate to the query. The links that are made between notes are used when retrieving relevant information. The agent can choose to retrieve all notes that have links to the retrieved note.

We again see that these memory systems mirror aspects of human memory. Many insights from how we store and use knowledge often serve as inspiration for the design of agentic memory systems.

Search-o1

A recent approach to agentic RAG is search-o1, a method that attempts to retrieve relevant context and put it throughout the reasoning traces to enhance the reasoning LLM's capabilities further.[7] Instead of autonomously searching for relevant information and using it in the prompt of the model, the information can be searched and retrieved *during* the LLM's reasoning

process. The Agent is instructed to use the <|*begin_search_query*|> and <|*end_search_query*|> tokens to start a search and then use the <|*begin_search_result*|> and <|*end_search_result*|> tokens to indicate what the retrieved information is. Figure 4-25 demonstrates this agentic RAG during reasoning.

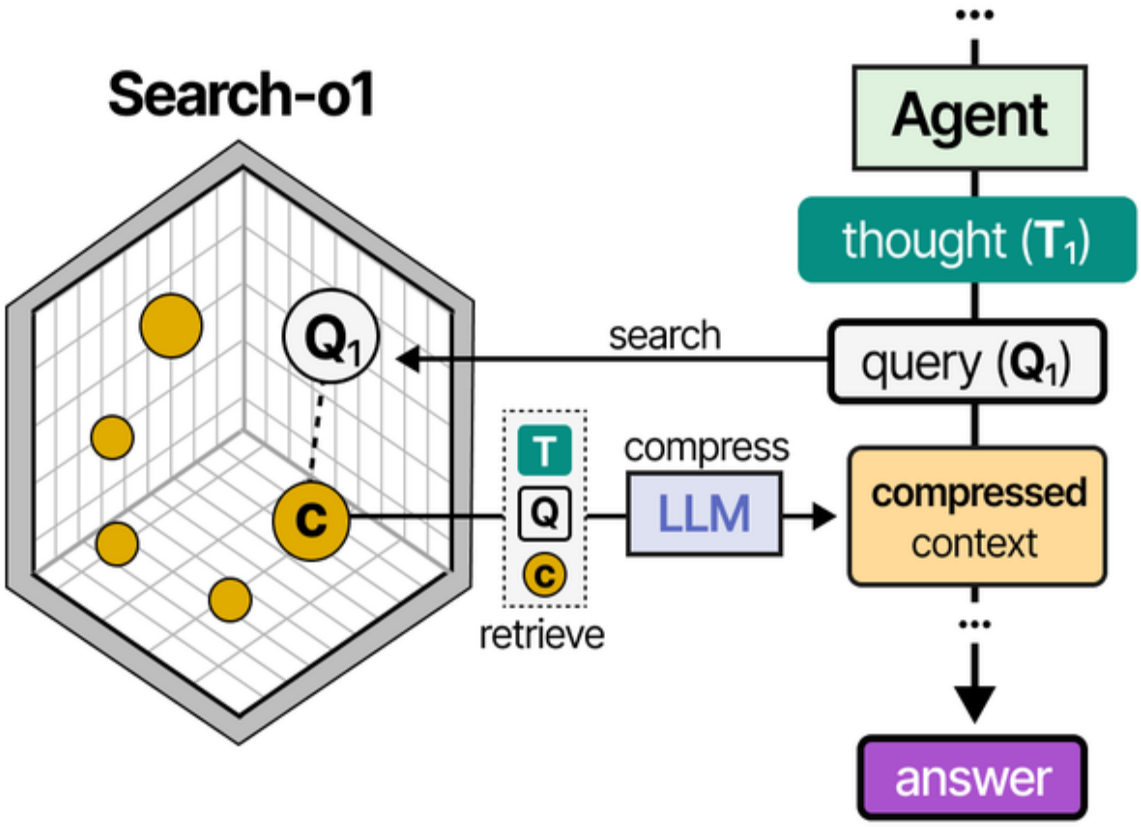


By enabling RAG during reasoning, the model can iteratively refine its reasoning process until it is confident in the final result. This dynamic approach is different from regular agentic RAG because it can be done autonomously within a single call rather than iterating over calls.

A downside to simply embedding documents within the reasoning traces is that the retrieved documents can be quite large and often contain irrelevant information and may therefore disrupt the reasoning flow. To solve this issue, the authors extend the reasoning agentic RAG by incorporating a Reason-in-Documents module. Using the search query, retrieved

documents, and reasoning trace, this module attempts to condense all information into focused reasoning steps. The agent's same reasoning LLM is used to process the retrieved documents to align with the model's specific reasoning traces.

With regular agentic RAG, information is just passed to the context without taking into account *how* the information needs to be processed. By enabling the same reasoning LLM to further process that information such that it fits within the reasoning traces, the flow of the traces can be kept intact. An overview of this system, called Search-o1, is given in Figure 4-26.



Note that this is typically used for long-term memory or external semantic memory that the agent might need to answer a given query. For instance, when given the query "Why are flamingos pink?", it will search for relevant information in Wikipedia during its reasoning process. The first result it finds mentions that it is due to specific pigments in their specific diet. It will use that information during its reasoning until it needs further clarification.

For instance, a second call to a different external database (e.g., ArXiv) will clarify that the specific pigments are carotenoid pigments, which are commonly found in brine shrimp.

This iterative process of querying information and compressing it within its reasoning process allows the model to reason about information when it is retrieved rather than stuffing all potential relevant information in the context.

Context Engineering

We have explored various types of memory that we can use to provide additional context to the Agent, including semantic memory, working memory, and other forms of memory. However, there might be more forms of context that we could give to the Agent, like:

System prompt

> The core context and rules for the agent, which define how it should behave (procedural memory).

Conversation history

> Both the conversation between the user and assistant, but also the LLM's internal thoughts (working memory).

Past experiences

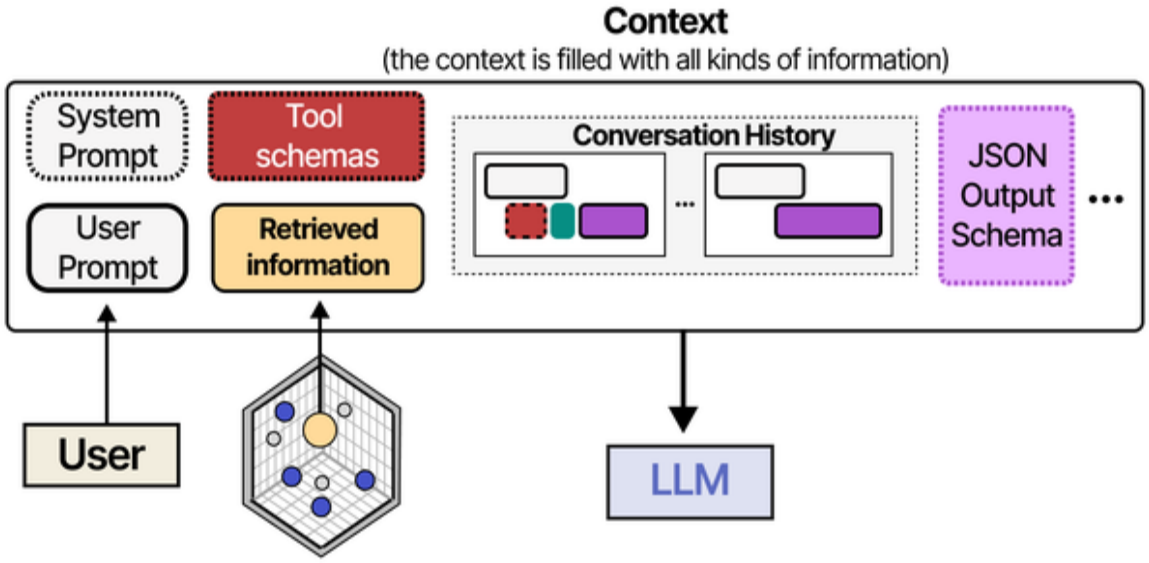
> Storing specific events, actions, or observations from tool use or user-related facts (episodic memory).

Retrieved information

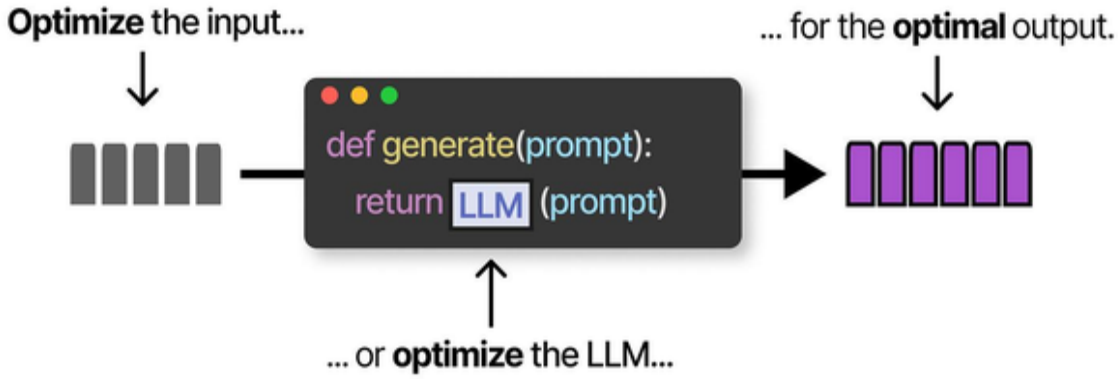
> External information that is typically stored in a vector database and accessed through RAG-like techniques (semantic memory).

This is not an exhaustive list, however. As the fields of LLMs and Agents grow, so do the sources of information that we could give to them. As such,

we can provide the Agent with all kinds of information sources to produce the answer that we want. As illustrated in Figure 4-27, the user's query or prompt is a subset of the LLM's entire context.

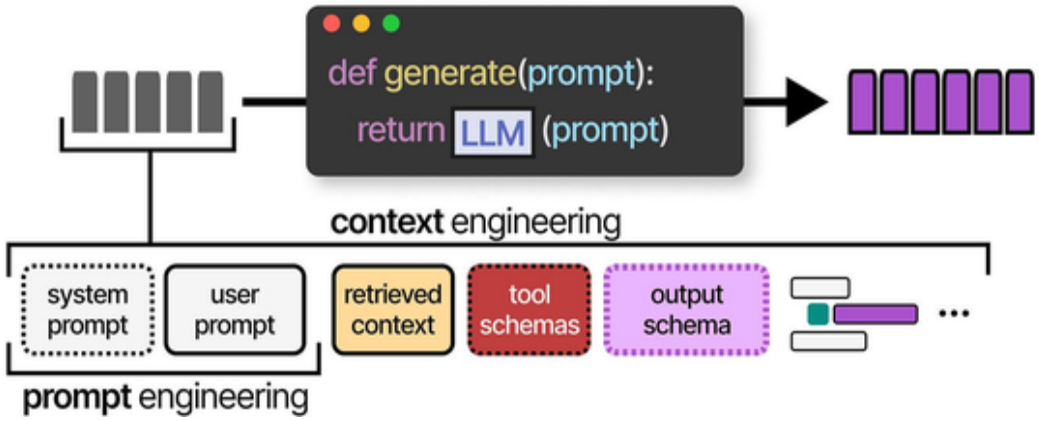


This context is given to the LLM, which in turn produces a list of tokens. As such, we can view an LLM as a function that takes several tokens (context), processes them, and outputs tokens. To optimize the output tokens for a given task, we can either optimize the LLM itself by training or fine-tuning it, or we can optimize the input, namely the context (Figure 4-28).



To optimize the quality of the output (the output tokens), we can either optimize the LLM itself by training or fine-tuning it, or we can optimize what goes into the LLM, namely, the context. The act of optimizing input

tokens so they produce the best possible output is called *context engineering*. Formally, it is finding the best context such that it maximizes the quality of the LLM's output for a given task.[8] As illustrated in Figure 4-29, where prompt engineering involves optimizing the system/user prompts, context engineering aims to optimize the entire context.

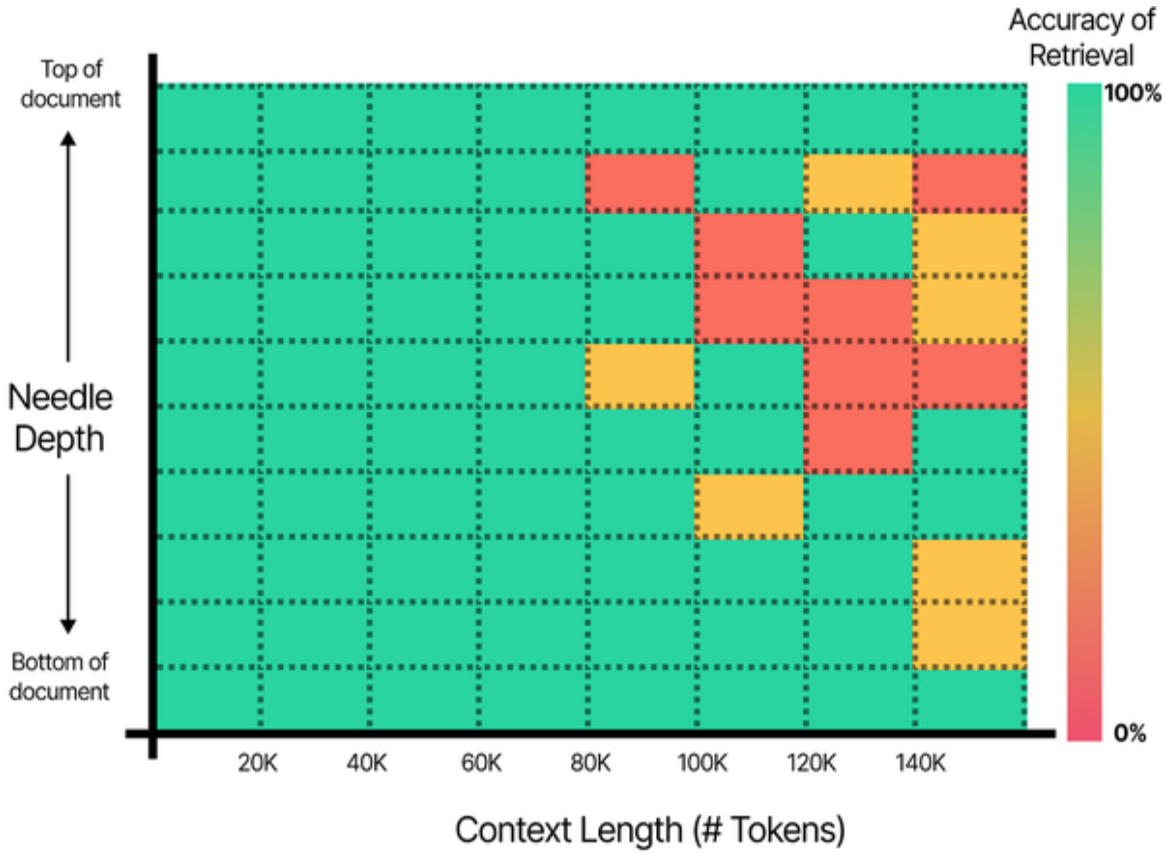


Context windows have grown larger, to the point where Google's Gemini 1.5 already reached a context window of a million tokens in February 2024.[9] It would be natural to conclude that context engineering means attempting to fill up this humongous context window with all kinds of information that relates to the task at hand. Moreover, there would be no more need for RAG since the context window is large enough to potentially hold your entire database.

A common benchmark for evaluating long-context LLMs in 2023 and the beginning of 2024 was the Needle In A Haystack (NIAH) test.[10] The test is rather straightforward; a random fact (needle) is placed somewhere in the middle of a long context window (haystack), and the model is asked to retrieve this statement. By iterating over different places and context lengths, we can measure how well LLMs perform over long contexts. It produced nicely looking visuals and was used by large LLM providers, like Anthropic's Claude 2.1[11] and Google's Gemini 1.5[12].

An example of such a visual is shown in Figure 4-30, which shows the results for a typical Need in A Haystack test, where the upper right quadrant

(needle at the top of the document and large context length) shows the degradation of LLMs at higher context lengths.



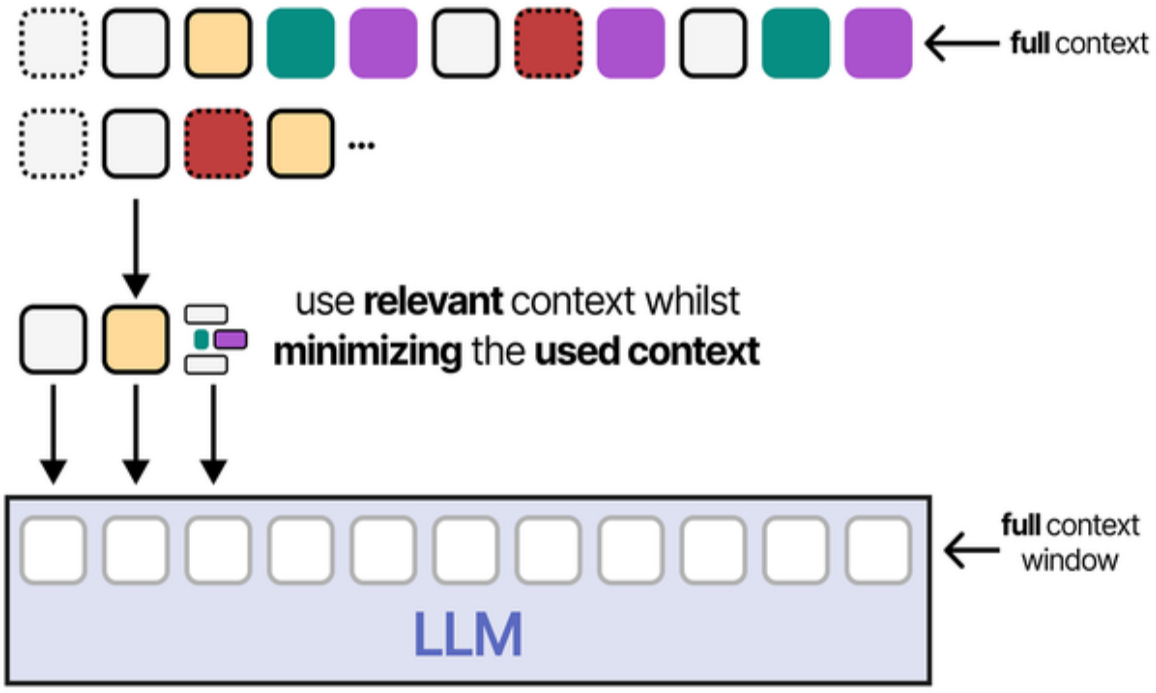
However, finding the needle is merely a retrieval task and is not indicative of more complex forms of long-context understanding, such as reasoning over hundreds of thousands of tokens. This holds especially true for agents that have to take into account many forms and lengths of context, like semantic and working memory.

Other benchmarks, like the RULER benchmark[13], have since been released that introduce new tasks like multi-hop tracing and aggregation to test behaviors beyond simple retrieval. The RULER paper, for instance, demonstrated models that performed well on the Needle In A Haystack test, all showed significant performance drops as the context length increased when applied to the RULER benchmark. Many others (e.g., [14] and [15]) found similar results and concluded that arbitrarily filling up the context window of LLMs would hurt performance. Some even called it “context

rot”[16], showcasing the importance of adding quality information. As such, there is a need to carefully construct and manage the model’s context window, thereby pointing to the importance of context engineering.

Cost and latency are other reasons for managing the context window. You could theoretically stuff everything in a 2-million token context length, like your entire external database, the full conversation history, some additional examples, etc. However, the Agent’s LLM has to process all these tokens, which significantly reduces latency. Costs also increase, considering more VRAM is needed when you increase the context length. Just dumping everything in the context is**,** therefore, a recipe for failure.

With context engineering, we aim to optimize the model’s context window with the right information, at the right place, and in the right format. It is the act of giving the LLM the appropriate context without overwhelming it. As such, and as shown in Figure 4-31, it is not about filling up the context window, but strategically choosing and placing information. Going back to our “LLM is a function” analogy, it is about optimizing the list of input tokens so that it produces the best possible output tokens.



You do not want too much or too irrelevant information as the costs of compute go up and the performance goes down. Likewise, if you give the LLM too little context and in an inefficient form, it will operate without having a good understanding of the context. It is a careful balance of providing just enough relevant information to the LLM to have it perform optimally. In other words, context engineering is much of an architectural problem that needs to be solved with lots of moving parts, like efficiently tracking, storing, and retrieving all existing and created information.

To help with context engineering, existing techniques for handling memory that focus on efficiency can be used. MemoryBank, for instance, dynamically adjusts the importance of memories and only keeps those that are truly important for the conversation history. Likewise, Search-o1 compresses the retrieved context and only gives back the relevant components without any noise. These dynamic memory techniques are exceptionally useful for context engineering as the information flows grow more complex.

What is inside the context?

We touched on it briefly before, but what exactly is inside the context? Let's illustrate a full context with an example and make it as concrete as possible. Imagine you ask an agent to perform deep research on reasoning LLMs and provide you with an overview of common techniques that are used in the field. We are going to be using the `messages` format to illustrate how the input to the LLM is continuously updated with the appropriate context.

Before starting with the example, let's explore the template for the system prompt first:

```
# Creating the system prompt
SYSTEM_PROMPT = """
You are a helpful research assistant. Your goal is to provide in-
depth analyses on complex AI-related topics.

<INSTRUCTIONS>
* Create a plan for the user's query
* Provide the necessary context
```

```
* If you lack information, you can ask clarifying questions to
the user
* Prioritize clarity of content.
* Make sure to use the current date to decide what is currently
SOTA.
</INSTRUCTIONS>

<TOOLS>
{information_about_tools}
</TOOLS>

<DATE>
{current_date}
</DATE>

<USER_QUERY>
{user_query}
</USER_QUERY>

Based on the provided context, start making a plan to research
this and check with the user for revisions.
"""
```

Note how the system prompt contains various XML tags to indicate what kind of information they contain. However, information seems to be missing, like the current data, information about tools, and the user's query. Other than the user's query, this information needs to be first retrieved from an external source. After doing so, the system prompt is generated with all kinds of context and passed to the Agent's LLM, which can be formatted as the following message.

```
# 1. Create a plan
messages = [{"role": "system", "content": SYSTEM_PROMPT}]
```

The LLM will generate a response with a short plan to research reasoning LLMs for the user to validate. Note that we will discuss planning and reflection in more detail in Chapter X. Let's assume the interaction goes as follows:

```
# 2. Reflect on the plan
messages = [
```

```
  {
    "role": "system", "content": SYSTEM_PROMPT},
  {
    "role": "assistant",
    "content": """
Here is my plan to research reasoning LLMs:
* Query the ArXiv database for the 10 most recent papers on
reasoning LLMs
* Summarize the abstracts
* Combine the summarizations for an overview
"""
  },
  {
    "role": "user", "content": "Search for surveys instead."
  }
]
```

The LLM suggested extracting the 10 most recent papers, which the user found to be inefficient and asked to search for surveys instead. Now that the Agent has executed its first task, namely creating a plan using its LLM, this information can be stored externally so it can be retrieved later on if necessary.

The plan is updated accordingly and kept in a special `PLAN.md` file that the Agent can update if necessary. Before it goes on and searches the ArXiv database, the current conversation history is cropped to only contain the `SYSTEM_PROMPT` and `PLAN.md`. This cropping can be a step executed by the Agent, but for the sake of simplicity, let's assume that after the user provided feedback, a simple script is always run that crops out this interaction. Thus, we are left with:

```
# 3. Crop context
messages = [
  {"role": "system", "content": SYSTEM_PROMPT},
  {"role": "assistant", "content": PLAN.md},
]
```

The context is kept to a minimum because there is no need for the LLM to see both the old and new plans.

Next, the Agent uses a tool (`search_arxiv`) to search for recent surveys on reasoning LLMs and extract the abstracts (ABSTRACTS).

```
# 4. Search on ArXiv
messages = [
  {"role": "system", "content": SYSTEM_PROMPT},
  {"role": "assistant", "content": PLAN.md},
  {
    "role": "assistant", "content": "<think>Let's search on
ArXiv"
    "tool_call": {
      "name": "search_arxiv",
      "arguments": { "search_term": "reasoning LLMs survey" }
   },
  {
    "role": "tool",
    "tool_call_id": "search_arxiv",
    "content": ABSTRACTS
  },
]
```

After the Agent has retrieved the abstracts, it decides to use another Agent to summarize the results.

```
# 5. Call the summarization agent with the abstracts
messages = [
  {"role": "system", "content": SYSTEM_PROMPT},
  {"role": "assistant", "content": PLAN.md},
  {
    "role": "assistant", "content": "<think>Let's search on
ArXiv"
    "tool_call": {
      "name": "search_arxiv",
      "arguments": { "search_term": "reasoning LLMs survey" }
   },
  {
    "role": "tool",
    "tool_call_id": "search_arxiv",
    "content": ABSTRACTS
  },
```

```
  {
    "role": "tool",
    "tool_call_id": "run_summarization_agent",
    "content": ABSTRACTS
  },
]
```

The summarization agent only needs to summarize the papers, so no other context is needed other than those papers to generate the summaries (`SUMMARIES`). As such, the messages of the summary agent that is created by `run_summarization_agent` could look like this:

```
# 5a. Run the summarization agent
messages_summary_agent = [
  {"role": "system", "content": "You are a summarization
expert."},
  {"role": "user", "content": "Summarize these papers: PAPERS"},
]
```

The resulting summaries (`SUMMARIES`) are passed back to the original Agent to parse and create a nice overview for the user.

```
# 6. Process summaries with the orchestrator (the original agent)
messages = [
  {"role": "system", "content": SYSTEM_PROMPT},
  {"role": "assistant", "content": UPDATED_PLAN},
  {"role": "assistant", "content": "I retrieved these summaries:
SUMMARIES."},
  {"role": "assistant", "content": "<think>Next, parsing
summaries.</think>"},
]
```

Note how this final messages variable contains a small part of the conversation history. Only the most relevant pieces of information are kept, and any irrelevant traces (like the full traces of the summarization agent) are not added to the context. During the entire process, the context is carefully managed by either the Agent itself or through external software.

Even in this simplified example, there is a lot of dynamic behavior between and within agents that update, store, and retrieve all kinds of context.

Moreover, all contexts in this example are texts and can be processed using a standard LLM. For certain use cases, however, you might want it to be able to process more modalities like images and sound. We will cover multimodal LLMs in more detail in Chapters X and X.

Context Engineering for Multi-Agent Systems

As we will cover more in-depth in Chapter X, multi-agent systems are where multiple agents work together to solve a given problem. Our example of a deep research agent used a summarization agent in its system, thereby working together. What makes context engineering especially difficult in multi-agent systems is that not only the context of the main agents needs to be carefully managed, so do the contexts of all other agents in the system. Moreover, the interaction between agents is also part of the shared context between those respective agents.

To manage this complex network of contexts, smaller agents can be used to handle some of the context "burden", as we illustrated in the deep research agent. By using a smaller agent with a smaller LLM for specific tasks, part of the context can be handled separately, leaving significant compute for the main agent (also called the orchestrator agent). What makes these small and/or specialized agents great for context engineering is that they can work on smaller and more manageable contexts, have clear responsibilities, and are easier to test and debug. These systems can be more reliable by separating tasks instead of having one agent juggle all kinds of different tasks and contexts.

Optimizing the Context

Throughout this chapter, we covered many different methods and techniques for handling the memory and context of LLMs and Agents. Optimizing what and how you put into the context is a multi-faceted problem that requires an understanding of various parts of your agent's architecture. Most strategies to optimize the context are built upon the main source of context, the memory modules of the Agent.

Although there are many strategies, let’s explore the most common ones.

Context Tracking and Storage

Before you give the Agent a possible relevant context, it first needs to be tracked and stored somewhere. We covered most of it already, as this relates to various forms of memory, and in particular episodic memory, which contains the actions the agent has taken thus far. Although episodic memory is seen as long-term memory, it is highly related to the conversation history of the Agent, which tends to capture the actions of the agent.

However, tracking the context goes beyond just the event traces of the agent. It also involves managing external knowledge sources, like a database of your proprietary data, which can serve as additional context. To set up these sources of knowledge, you will have to decide beforehand what kinds of information you want to track. Although it may seem obvious at first, there are many types of information you can track:

- Agent behavior
    - – Tool usage by the agent (and any sub-agents)
    - – Tool outputs and intermediate results
    - – Interactions between (sub-)agents
    - – Internal reasoning steps
    - – Conversation history
    - – Failures/successes
- User behavior
    - – User intent (explicit requests and goals)
    - – User feedback (edits, approvals, rejections)
- Knowledge sources

    - Snapshots of your proprietary database(s) for reproducibility and auditing
    - External documents (RAG, APIs, etc.)
    - Structured artifacts like `PLAN.md`, `REQUIREMENTS.md`, etc.
- System-level
    - Configuration (LLM hyperparameters, available tools, etc.)
    - Policies (guardrails, constraints, etc.)

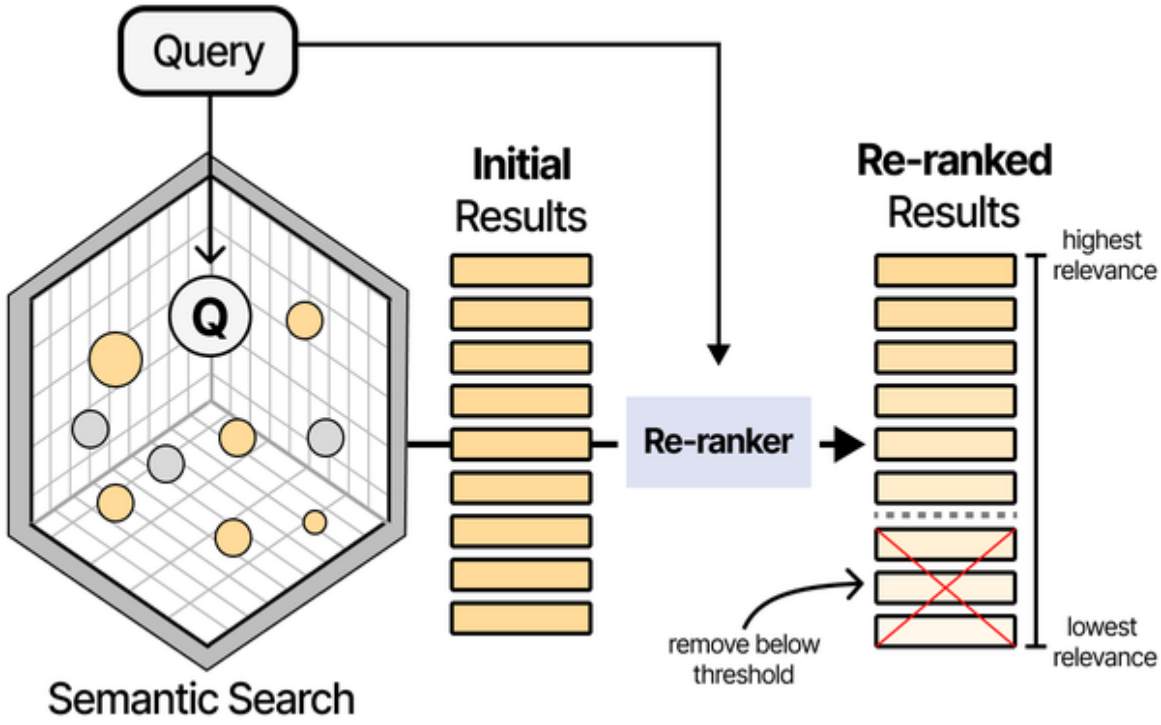
All the above are merely examples of what you could possibly track. In practice, not everything is going to be useful, and at the same time, many other things should be included. As we will explore later on, tracking these kinds of contexts and information also helps communicate the user's intent and debug these complex systems.

Context Selection

Assuming you have set up all your databases, the very first thing you can do to optimize the context is to have a system in place that selects the right context. As we discussed before, RAG is an amazing technique and example for selecting what is relevant. Although we have seen variants of RAG that improve on this, we haven't yet explored how we can further improve this selection process.

In RAG, the input documents are typically split into smaller parts, such as sentences or paragraphs, to isolate the information they contain and keep it to a single subject. However, when you then run a RAG pipeline, you typically get a collection of documents in return. For example, if we search a vector database for the "causes of climate change," the system might return documents about greenhouse gases, industrial activity, and deforestation. Although those documents might not directly answer our question, they are related.

To improve this process, we can use a re-ranker to refine the set of documents that were retrieved. This technique, often a language model, takes in both the query and retrieved documents to re-rank the retrieved documents based on their relevance to the query and to each other (see Figure 4-32). By providing the re-ranker with additional context (all retrieved documents), it can operate on far fewer documents than if we were to give it the entire database. Moreover, after the results have been re-ranked according to their relevance, we can choose to only keep the most relevant documents.



Re-ranking is only one of the many ways to select and create the right context. We can also structure the output to ensure the responses of the Agent are broken up into logical parts and only contain the necessary components. Likewise, we can employ business rules that give additional weights to certain pieces of information that always provide important context (much like a system prompt).

Note that selecting the right context can also mean selecting the right context for the right agent. By isolating the context across multiple

specialized agents, each agent is able to focus on a smaller part of the problem without being overwhelmed by the full context.

Context Compression

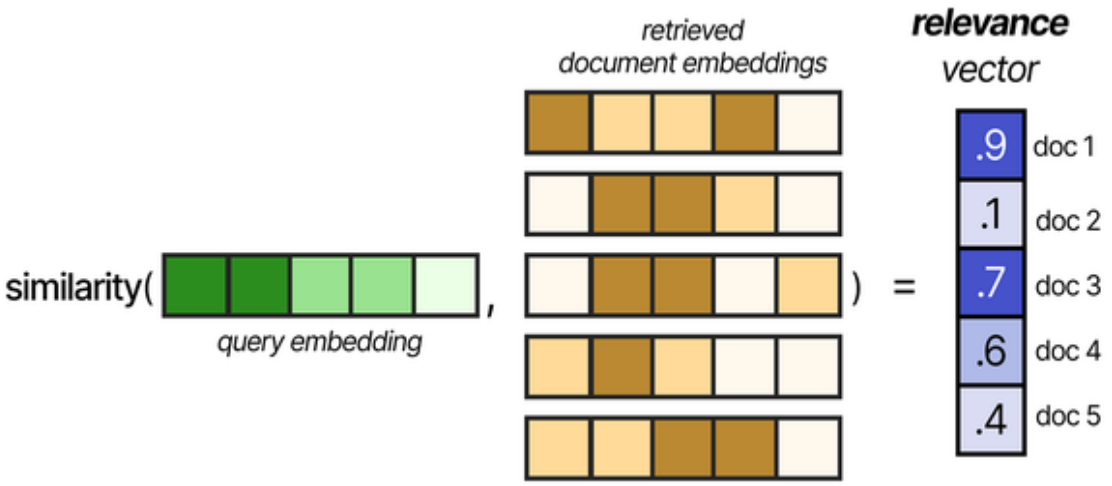
The goal of context engineering is to find a balance between what you put in the context and how much. As such, compressing the context as much as possible is an important strategy for optimizing the context.

A common way to handle compression is what we discussed at the beginning of this chapter, using an LLM to create summaries of your conversation history. As we explored in Search-o1, we can even compress the output of the RAG pipeline using an LLM to summarize the retrieved documents.

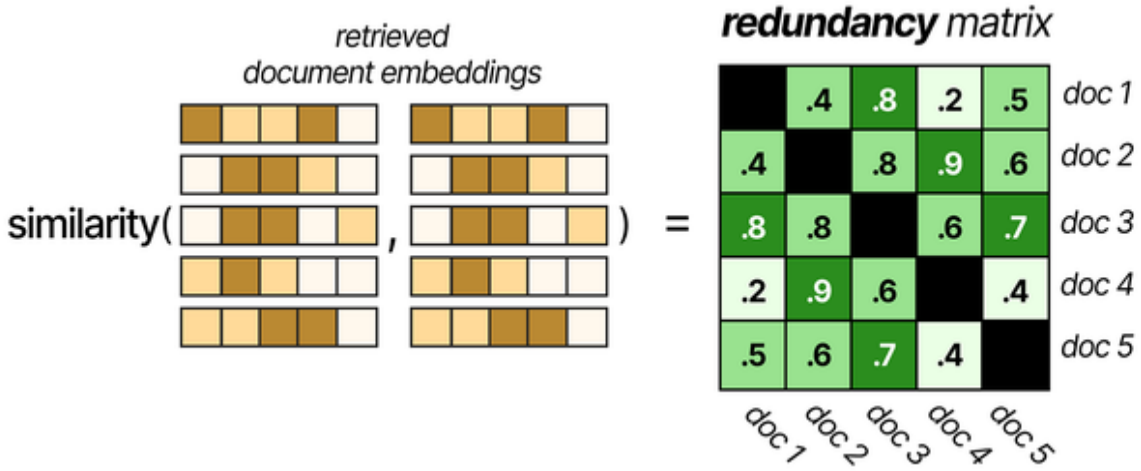
Another method of compressing the context is by reducing redundancy. Even with a re-ranker, the top 5 most relevant results might all contain very similar documents. Together, they are not bigger than the sum of their parts but smaller instead because they contain similar information. As such, we do not only want the most relevant documents, but in a way that they are still documents that each contain a new piece of information.

A common technique to use, whether they are the output documents of your RAG pipeline or other pieces of retrieved information, is called Maximal Marginal Relevance (MMR).[17] This technique uses a precalculated relevance vector and redundancy matrix to balance the diversity of documents.

First, the similarity between the retrieved document and query embeddings is calculated. This results in a *relevance vector* which has a value per document to indicate how similar/relevant the given document is to the query (see Figure 4-33).



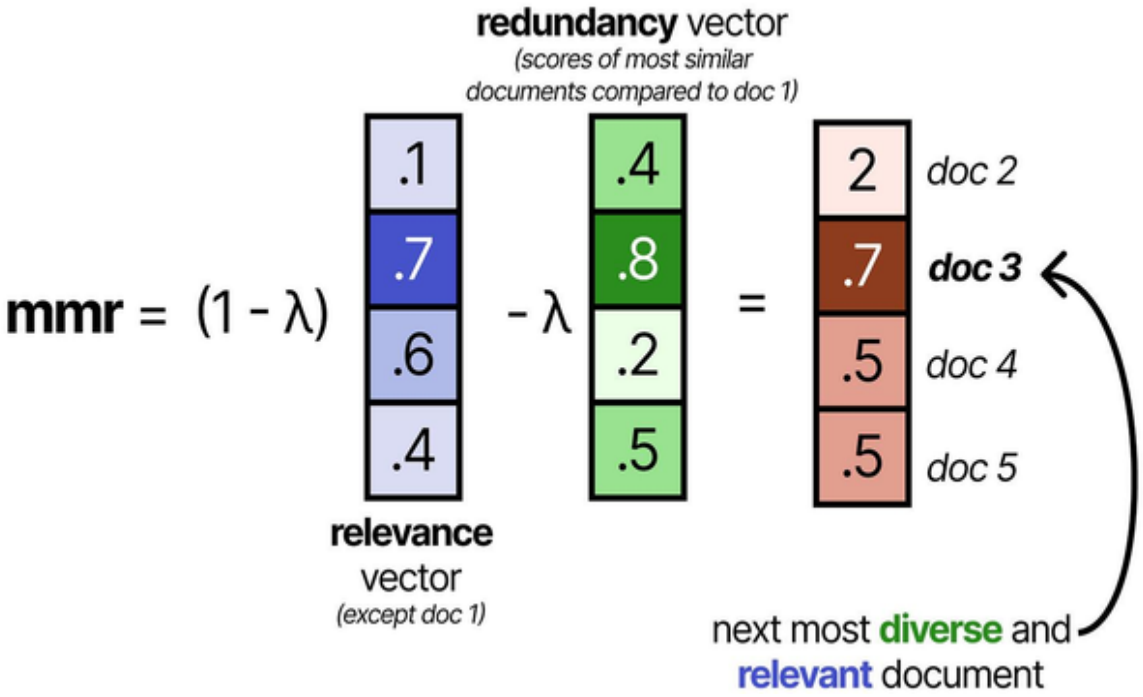
Second, the similarity between the retrieved documents is likewise calculated to construct a similarity matrix called the *redundancy matrix*. This matrix is used to potentially discard documents that are too similar to the ones we had already chosen (see Figure 4-34).



Then, the relevance vector and redundancy matrix are used iteratively to decide which retrieved documents are similar enough to a given query but dissimilar to all other retrieved documents. An important component is the λ parameter, which we can tweak to decide how diverse the output should be (a higher score indicates higher diversity).

Since we haven't chosen any documents, we start by selecting the one with the highest score in the relevance vector, namely document 1. Then, we take the relevance vector and multiply it by (1- λ) to decide the importance of

similarity over diversity. From the resulting scores (one for each document other than document 1), we subtract λ times the redundancy vector. This vector contains the highest scores in the redundancy matrix that relate to the documents we had already chosen (document 1). As demonstrated in Figure 4-35, document 3 is added as the next most diverse and relevant document.

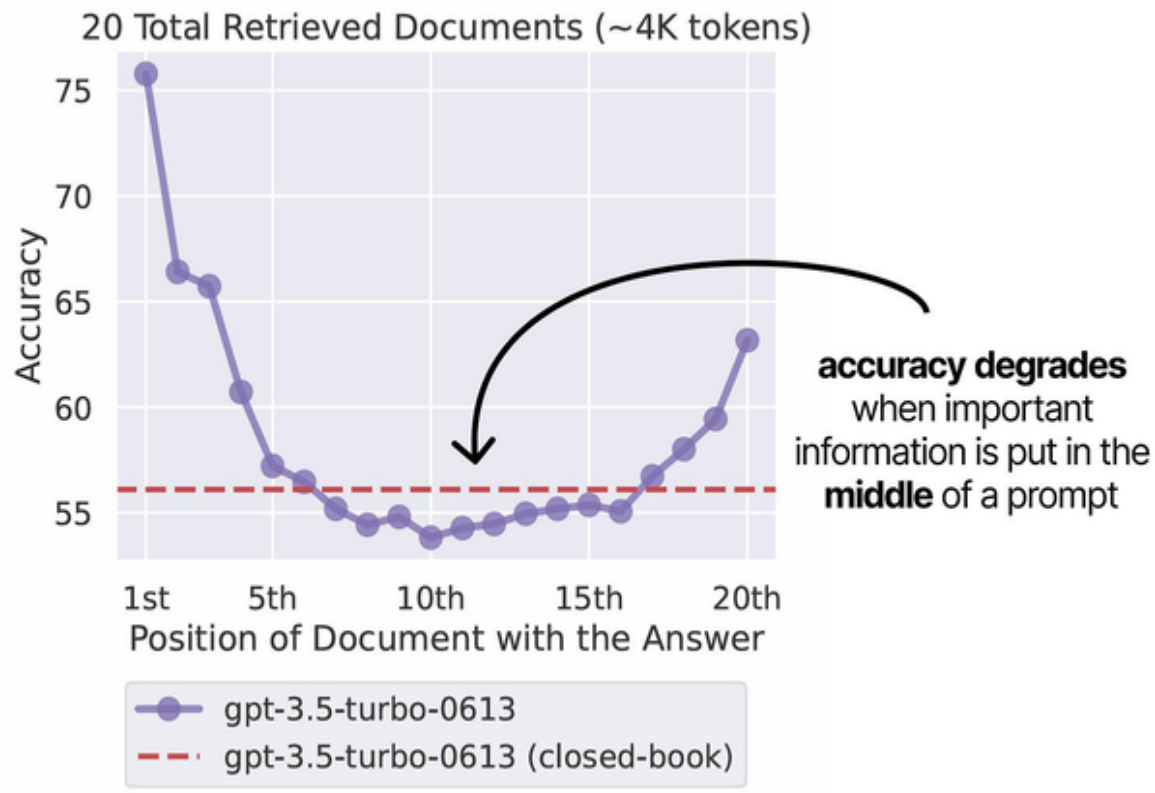


We continue this process, but instead of comparing to document 1 only, the redundancy vector contains the highest similarity scores that relate to either document 1 or 3, whichever is highest. This process is only repeated for the next document, since we wanted to bring down our original set of 5 documents to 3 in total.

Like re-ranking, MMR is merely an example of a technique that can be used to compress the output. We are not limited to using LLMs to directly compress the retrieved documents; we can instead use techniques like MMR to simply ignore certain documents because they are too similar to each other. Likewise, deduplication techniques can be used to remove contexts that are essentially duplicates of one another.

Context Ordering

The order of the context is vital to the performance of your agent. Early research into the position of important information in prompts found that LLMs have a tendency to pay more attention to the beginning and end of a prompt.[18] As a result, they often end up losing information in the middle, which they call the "lost-in-the-middle" phenomenon. Figure 4-36 is an annotated figure from the "Lost in the Middle: How Language Models Use Long Contexts" paper that nicely illustrates this concept.



This tendency to focus on the beginning and end of prompts is similar to human behavior. The serial-position effect states that people generally recall the first (primacy effect) and last (recency effect) items in a series best, whereas the middle items are recalled worst.[19] It is interesting to see how much of LLMs' behavior follows similar human tendencies.

Context as the Specification

Arguably, one of the most important things about context engineering is that it requires a shift in mindset. The context that is given to the agent can be seen as a tool for communication. Not just for the agent, but to the people that you work with. More specifically, the context that you give to the agent, including the query, `PLAN.md`, `REQUIREMENTS.md`, codebase, etc., all serve as a tool to communicate what your intention is. For example, when you allow a coding agent to fully create a PR on its own, it does not suffice to go through the code itself to check whether everything is there. The initial query, `PLAN.md`, etc., all serve as the initial specification of the PR and should be tracked as well.

As such, we can view the context of the agent as the specification of your feature. As agents are becoming more autonomous, it is important to track the intention of their behavior through the context that is given. Where we would view prompt engineering as user-facing, context engineering is a much more developer-oriented tool and requires careful communication of why the agent is executing certain tasks. The answer to the “why” starts with the user’s intention.

Think about it like this: how strange it is that we tend to throw away the input to our function (the LLM) and only keep track of the output! Not only for reproducibility, but also for communication, it allows you to understand why the agent has chosen certain tools, executions, and outputs. Moreover, this transparency of intention also serves as a great tool for debugging your agent.

Context, as the specification, brings about significant potential for domain-specific industries. The context that you give an agent changes drastically between use cases and applications. Health care requires a completely different context than law, for instance. As such, there is not a single framework for context engineering and instead requires developers to consider their domain-specific knowledge sources, like patient data in health care and research papers in academia.

Summary

In this chapter, we explored various methods for giving LLMs and Agents memory. We first covered various techniques for short-term memory, including the conversation history and methods for compressing it and keeping it manageable. Short-term memory is an important, but often underestimated component of enabling memory in intelligent systems.

Then, we explored long-term memory, including methods for RAG (MemoryBank) and agentic RAG (A-MEM and Search-o1). These methods are often inspired by human memory systems and may include techniques for degrading memory or deciding what's meant to be important.

Finally, we explored context engineering as the next frontier in memory. Where we used to engineer our prompts ourselves, the entire context window now requires careful consideration. We covered why this context is important for agents and various methods for optimizing it.

The context, being much more than just the user's prompt, contains all previously discussed forms of memory and potentially even more, like tools. In the next chapter, we will explore how tools can further enhance the capabilities of LLMs as an important component of agents. Moreover, we will cover how these tools can be called and the best practices for doing so. In Chapter 6, we cover planning and reflection capabilities of agents, where memory plays an important role.

---

1 Sumers, Theodore, et al. "Cognitive architectures for language agents." *Transactions on Machine Learning Research* (2023).

2 Zhang, Zeyu, et al. "A survey on the memory mechanism of large language model based agents." *ACM Transactions on Information Systems* (2024).

3 Hsieh, Cheng-Ping, et al. "RULER: What's the Real Context Size of Your Long-Context Language Models?." *arXiv preprint arXiv:2404.06654* (2024).

4 Lewis, Patrick, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." *Advances in neural information processing systems* 33 (2020): 9459-9474.

[5] Zhong, Wanjun, et al. “Memorybank: Enhancing large language models with long-term memory.” *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. No. 17. 2024.

[6] Xu, Wujiang, et al. “A-mem: Agentic memory for llm agents.” *arXiv preprint arXiv:2502.12110* (2025).

[7] Li, Xiaoxi, et al. “Search-o1: Agentic search-enhanced large reasoning models.” *arXiv preprint arXiv:2501.05366* (2025).

[8] Mei, Lingrui, et al. “A Survey of Context Engineering for Large Language Models.” *arXiv preprint arXiv:2507.13334* (2025).

[9] https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024

[10] Gregory Kamradt. Needle In A Haystack- pressure testing LLMs. Github, 2023. URL https://github.com/gkamradt/LLMTest NeedleInAHaystack/tree/main

[11] Anthropic. Long context prompting for Claude2.1. Blog, 2023. URLhttps://www.anthropic.com/index/claude-2-1-prompting.

[12] Team, Gemini, et al. “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.” *arXiv preprint arXiv:2403.05530* (2024).

[13] Hsieh, Cheng-Ping, et al. “RULER: What’s the Real Context Size of Your Long-Context Language Models?.” *arXiv preprint arXiv:2404.06654* (2024).

[14] Li, Tianle, et al. “Long-context llms struggle with long in-context learning.” *arXiv preprint arXiv:2404.02060* (2024).

[15] Levy, Mosh, Alon Jacoby, and Yoav Goldberg. “Same task, more tokens: the impact of input length on the reasoning performance of large language models.” *arXiv preprint arXiv:2402.14848* (2024).

[16] Hong, Kelly, Anton Troynikov, and Jeff Huber. *Context Rot: How Increasing Input Tokens Impacts LLM Performance.* Chroma, July 2025. https://research.trychroma.com/context-rot.

[17] Carbonell, Jaime, and Jade Goldstein. “The use of MMR, diversity-based reranking for reordering documents and producing summaries.” *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 1998.

[18] Liu, Nelson F., et al. “Lost in the middle: How language models use long contexts.” *arXiv preprint arXiv:2307.03172* (2023).

[19] Bennet B. Murdock Jr. 1962. The serial position effect of free recall. Journal of experimental psychology, 64(5):482.

Chapter 2. Tool Usage, Learning, and Protocols

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *mcronin@oreilly.com*.

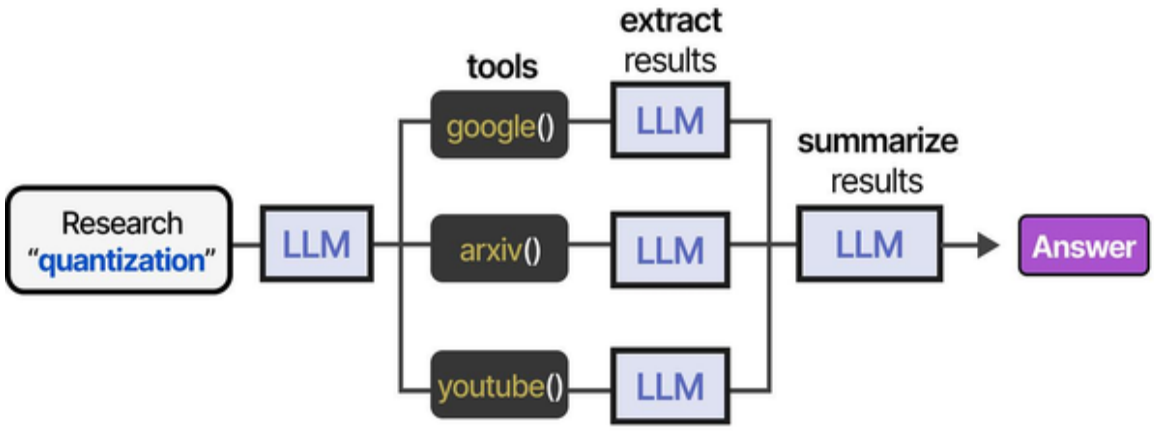
The tool module is such an interesting module for augmenting your LLM and generating autonomous behavior in agents. By themselves, LLMs are nothing more than functions that take in some string and output some string. Although that creates interesting chatbots, they cannot yet interact with their environment.

A key trait that defines an agent is its ability to autonomously search for, select, and utilize tools, allowing it to interact with and influence its environment. With enough capabilities, agents can even create their own set of tools to use. The benefit of tools is not contained to interaction with the environment. Tools are typically used to access external knowledge or memories, as we discussed in the previous chapter. We can even use tools to access specialized (L)LMs that have capabilities that extend beyond what the agent is capable of, like multimodal LLMs or LLMs specialized for certain tasks (like coding). Generally, tools allow an LLM to take action

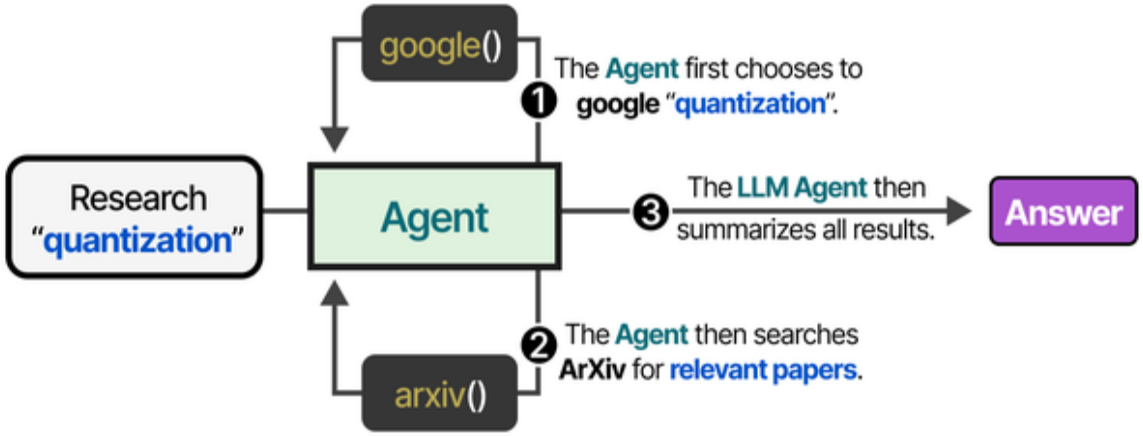
and interact with the external environment or extract data and use external applications (see Figure 5-1).

However, regular LLMs cannot search the web, use a calculator, or schedule appointments. They can only communicate the intention of doing so, without being able to act upon it. Without tools, agents can only *think* autonomously and not *act* autonomously.

The degree of autonomy also decides how tools are being used. As shown in Figure 5-2, if there is no autonomy but a fixed flow, then tools can be used in a predefined order. For instance, a research agent might always call tools like arXiv and Google to extract results, which are subsequently summarized.

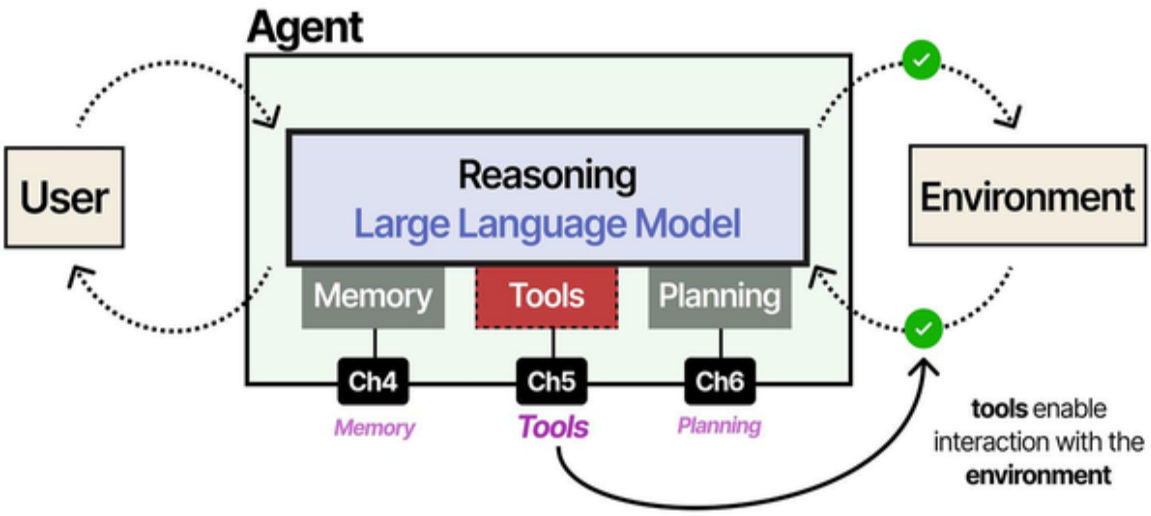


In contrast, systems with larger degrees of autonomy allow Agents to choose which tool to use and when. Illustrated in Figure 5-3, they are still sequences of LLM calls but with autonomous selection of tools decided by the Agent.



Tools are about much more than merely using them. How are they created? How is the output of a tool processed by the LLM or Agent? How are tools selected? How many tools can an LLM effectively handle?

Throughout this chapter, we will not only explore several types of tools but also how LLMs and agents learn how to use them and even how these tools can be standardized across different agentic systems. As shown in Figure 5-4, this is the module that allows interaction with the environment. It makes the next chapter, planning out actions to take, more than a theoretical exercise. After all, how could an LLM act autonomously without tools?

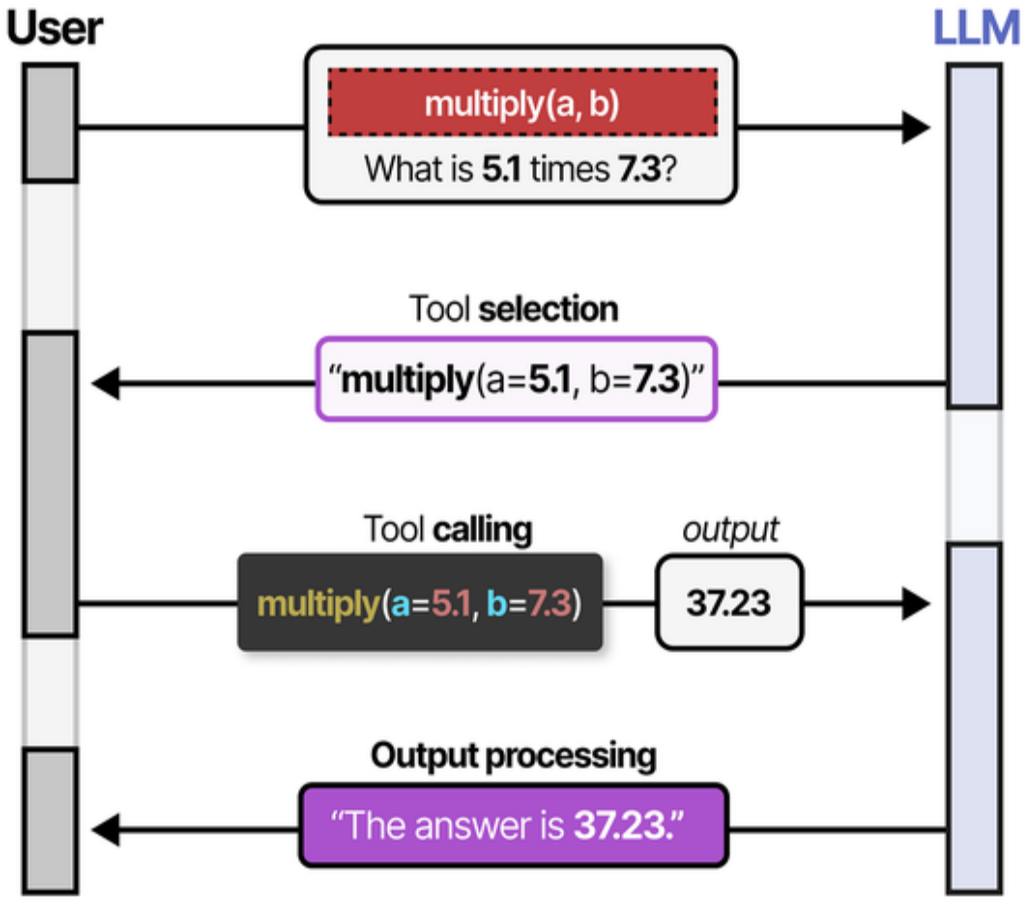


Tool Usage

Tool usage consists of more steps than you might expect. It does not start with an LLM using a tool, but actually creating and defining the tools, planning which tool to use instead, and eventually calling them. Although the steps in tool usage can be seen from many different perspectives (e.g.,[1],[2], and [3]), we explore the following:

- Tool Creation
- Tool Definition
- Tool Selection
- Tool Calling
- Output Processing

Although Agents might seem like they can do everything themselves (which they can to a certain degree), they require help from the user or developer to actually use the tools. The high-level process is shown in Figure 5-5, which is an adaptation of OpenAI's tool calling flow[4], where we additionally focus on creating the tool and highlight the steps a single tool call takes.



Note that each step may have various forms and implementations, many of which we will cover later on. For now, let's explore the most common way tool usage happens in these steps and explore them in more detail.

Tool Creation

Although tools can come in various forms, they are typically functions that can be accessed through either an API or some internal code. This can be as complex or minimal as is required for your application. LLMs, for instance, are not known for their mathematical capabilities. Let's create a simple function for multiplying two values that the LLM can use. Note that all coding examples are meant as pseudo-code, and the full examples can be found in the book's GitHub repository.

```
def multiply(a: float, b: float):
    """
    Multiply two numbers

    Args:
        a: First number
        b: Second number
    """
    return a*b
```

These tools should be heavily documented, as shown in the example, because the docstrings will also be communicated to the LLM. A clear benefit is that it motivates users to focus more on writing great documentation, which is often not a priority when developing code.

This tool can be accessed by directly calling `multiply(a, b)`, which is something the LLM cannot do, but can be done by an automated system. Note that external API calls may also call tools. Either way, a tool is generally considered to be some form of function that can be called in various ways. In our example, the most straightforward method would be to save it in a dictionary so that we can access the function by a string. In this case, our *database* is the `tools` variable:

```
tools = {
  "multiply": multiply
}
```

Although creating tools can be a straightforward process, the LLM should be taken into account when designing them. It is a similar process to designing code for people to use, where you may ask yourself: Does the user understand what the tool does? Is it clear what is being returned? Are the variable names descriptive? Is there documentation?

This is especially important for LLMs as they often explicitly need to be told what exists, what doesn’t, and what the tools are capable of.

Tool Definition

After creating the tool, we need to inform the LLM what the tool does; this is called tool definition. Without communicating to the LLM which tools exist, what they do, and how they are used, the LLM isn't able to properly use them.

There are several methods by which we can communicate this to the LLM:

Learning

(specific) tool definition and usage are learned during training

Prompting

The tool definition is shared with the LLM through structured prompts.

How to use (specific) tools and when they are needed is often learned during the fine-tuning state of an LLM, where the model learns how to follow instructions and use tools. This learning stage can be split up into two aspects: we either tune the model to *learn* about *specific tools* or learn how to use *tools in general*. Learning about specific tools can be costly and limit how many tools you can add to the model. Instead, the focus of recent models is typically their improved instruction-following capabilities and general tool use. Instead of learning about specific tools, they learn to recognize definitions of tools in their prompts and dynamically use them.

As (reasoning) LLMs are becoming easier to steer and dynamically learn to use tools, we can share the tool definitions through prompts. When you have a small set of tools the LLM should use, you can share that through the system prompt:

```
system_prompt = """
...

# Tool Definition
You can use the following tools:

- multiply(a, b): multiplies two numbers
- divide(a, b): divides a by b
```

```
- add(a, b): adds two numbers
- subtract(a, b): subtracts b from a

# Tool Usage
When you need to use them, write: [function_name(value1, value2)]
...
"""
```

Note that the definition here isn't following a standardized procedure. Frankly, this is something we made up on the spot to show you that an LLM can call a tool anyway you decide. If it uses "`TOOL(value1, value2)`" or perhaps structure it like "`TOOL, value1, value2`" does not matter. The only thing you need is a small piece of code that recognizes when a tool is being called. We will cover how the LLM actually calls a tool a bit later in this chapter.

Another method of sharing the definition of the tools with the LLM is through structured *function calling*. Instead of messing around with the prompts ourselves and optimizing how we describe and format each tool, we can define each tool in structured schemas instead. These are typically JSON Schema objects that describe the function's name, its description, and its parameters (types, required fields, ranges, etc.). An example of such a schema for our multiply tool is the following:

```
{
  "type": "function",
  "function": {
    "name": "multiply",
    "description": "Multiply two numbers",
    "parameters": {
      "type": "object",
      "properties": {
        "a": {
          "type": "number",
          "description": "First number"
        },
        "b": {
          "type": "number",
          "description": "Second number"
        }
      },
      "required": [
```

```
        "a",
        "b"
      ]
    }
  }
}
```

These schemas are often passed as a separate parameter when using external APIs. OpenAI, for instance, uses the `tools` parameter where you can send over the JSON schemas of your tools.[5] That allows the LLM or even the Agent to treat it as special metadata and process it beforehand if necessary.

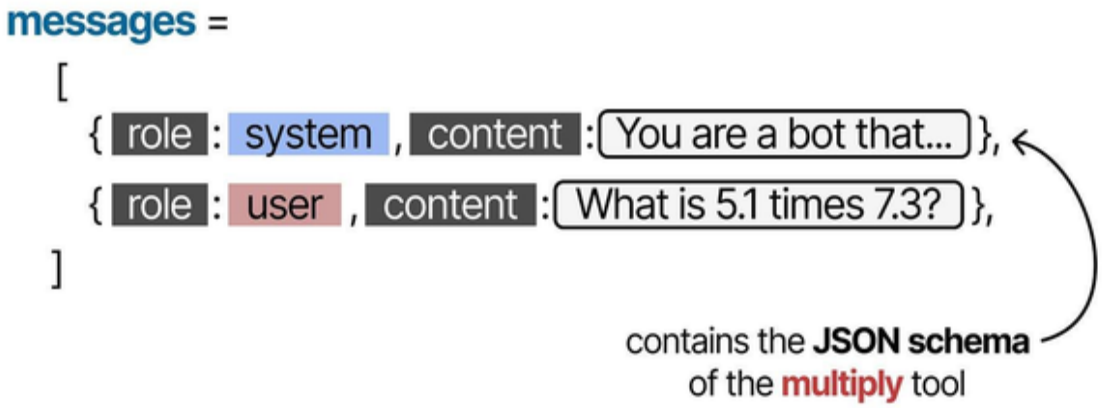
In practice, however, these schemas are typically processed as if they were "regular" prompts and put into, for example, the system prompt:

```
# Tool JSON Schema put into the system prompt
system_prompt = """
You are a bot that responds to mathematical queries.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools>
XML tags:

<tools>
{"type": "function", "function": {"name": "multiply",
"description": "Multiply two numbers", "parameters": {"type":
"object", "properties": {"a": {"type": "number", "description":
"First number"}, "b": {"type": "number", "description": "Second
number"}}, "required": ["a", "b"]}}}
</tools>

For each function call, return a json object with function name
and arguments within <tool_call></tool_call> XML tags:

<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>

"""
```

This example is using the transformers library to format the system prompt. Note that they have provided additional instructions on how to identify when a tool is being called by putting the call between XML tags (`<tool_call>` and `</tool_call>` respectively).

Although we are using structured JSON schemas for communication, the LLM still has to interpret how to use the tool and when, which makes tool usage a difficult task. Therefore, the more descriptive and clear the schema, the more likely the LLM will make the right tool call decisions.

At this point, the user can finally ask their question. Since they have access to the multiply function, let's keep the query simple: "What is 5.1 times 7.3?". To illustrate how this is going to be processed, we make use of the `messages` structure that we explored in the previous chapter. This is visualized in Figure 5-6 where the system prompt contains the definition of our tool.



Regardless of whether you use a JSON schema or describe the tool, there are several best practices for defining functions to take into account:

Document your tool with extensive descriptions of parameters, function names, and even examples

> This allows your agent to have a better understanding of what the tool is capable of.

Minimize the number of tools

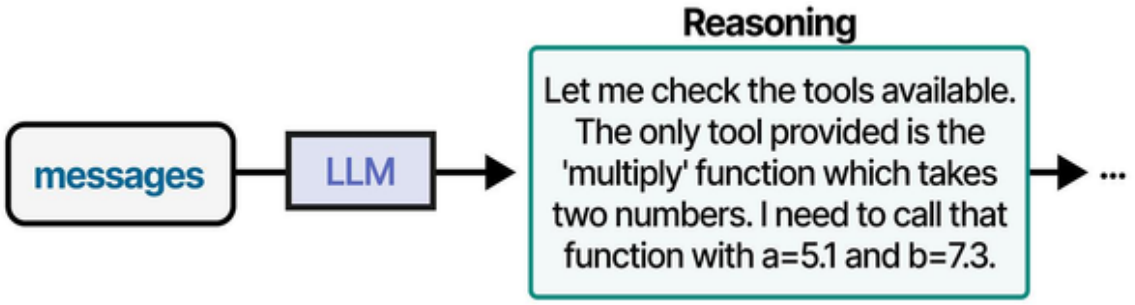
Although having many tools expands the capabilities of your agent, it will become much more difficult to select and use the appropriate tool.

Minimize the scope of a tool

Complex tools with many parameters are difficult to use, even for individuals, let alone an agent.

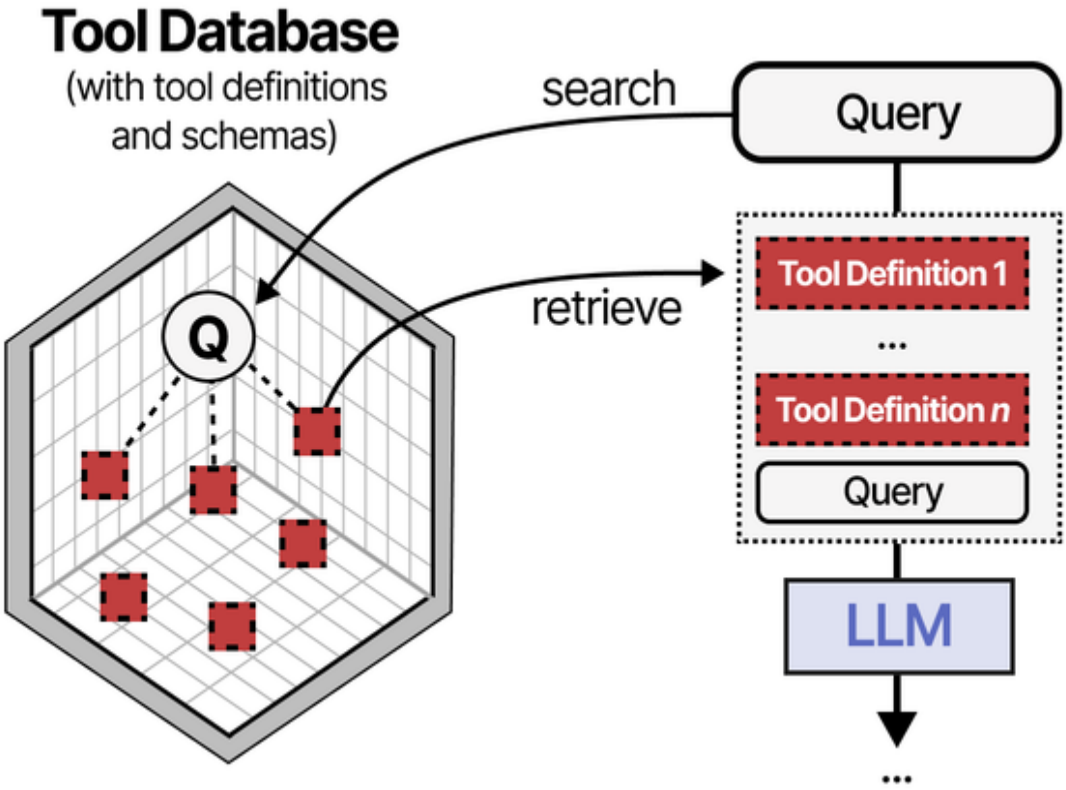
Tool Selection

Now that the tools are created and defined, we can start the process of calling the tool. The LLM first needs to select the right tool for a given query, which can be a difficult task. Especially with potentially dozens of complex tools, the LLM doesn't only need to select the most appropriate one (if one at all) but also use it correctly. Although we can share an extensive JSON schema for each tool, the LLM needs to be capable enough to actually follow it through. This is where reasoning LLMs shine. They can spend any number of tokens "thinking" about which tools to use and how to properly use them. Illustrated in Figure 5-7 is the reasoning process of LLMs to decide which tool to use and how.



Note that discovering the tools that exist or might be relevant becomes more important when the number of tools increases. As we discussed in the previous chapter, even when you have a large context window, filling it up to the brim with tool JSON schemas is bound to decrease the LLM's performance. As with context engineering, the process of discovering tools might be helped with methodologies like RAG, where you store all tool schemas in a separate database for the LLM to discover. Illustrated in

Figure 5-8 is this idea of using a vector database to discover which tools are most relevant to a user's query.

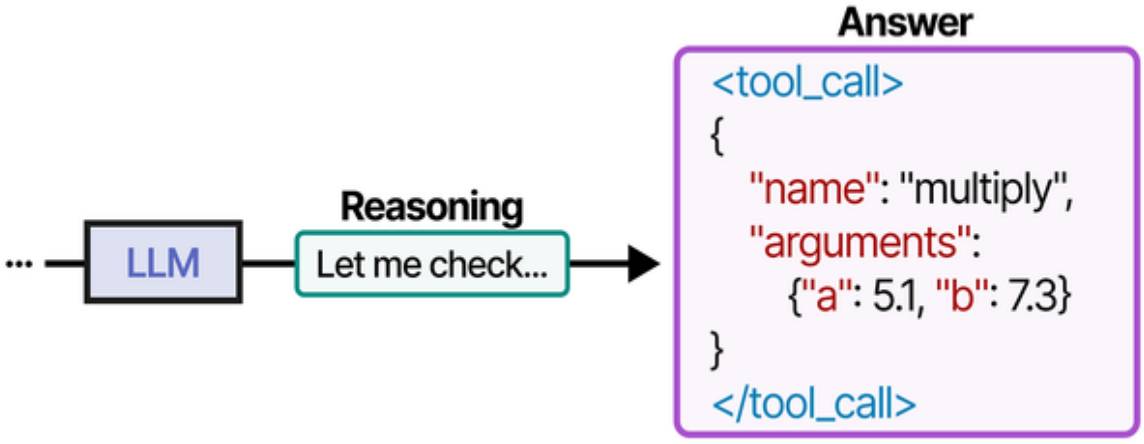


Note that the planning capabilities of LLMs are vital to having a good selection. For simple queries, like "What is 1+1?", the selection of tools is not going to be a challenge. However, when planning a vacation, a multi-step process is going to be needed where the agent first needs to plan out its behavior. This planning (and reflection) behavior is going to be discussed in the next chapter in more detail.

Tool Calling

Next, when the LLM ends its thinking process, it can start creating the tool call. It will output a string, and if the LLM is capable enough, correctly format the tool call with the given arguments. This is illustrated in Figure 5-9 with the system prompt that we defined in the "Tool Definition" section. Note that this is a correctly formatted answer with the `<tool_call>` and

`</tool_call>` tags together with the arguments and function call that can be parsed as JSON.



However, this answer is merely a string and will not execute the tool. In fact, you can view this output as merely the intention of the LLM to call a tool. Without any help from the user or additional software, nothing will happen. We will have to write a small piece of code that will extract the JSON call and execute it.

For this, we use regex (`re`) to extract the tool call from within its `<tool_call>` tags, convert it to JSON, and finally call it using our `tools` dictionary:

```
import re

# LLM's output
output = """
<tool_call>
{"name": "multiply", "arguments": {"a": 5.1, "b": 7.3}}
</tool_call>
"""

# Get tool call
match = re.search(r"<tool_call>\s*(\{.*?\})\s*</tool_call>",
output, re.DOTALL)
tool_call = json.loads(match.group(1))

# Calling the tool
name = tool_call["name"]
```

```
arguments = tool_call["arguments"]
tool_output = tools[name](**arguments)
```

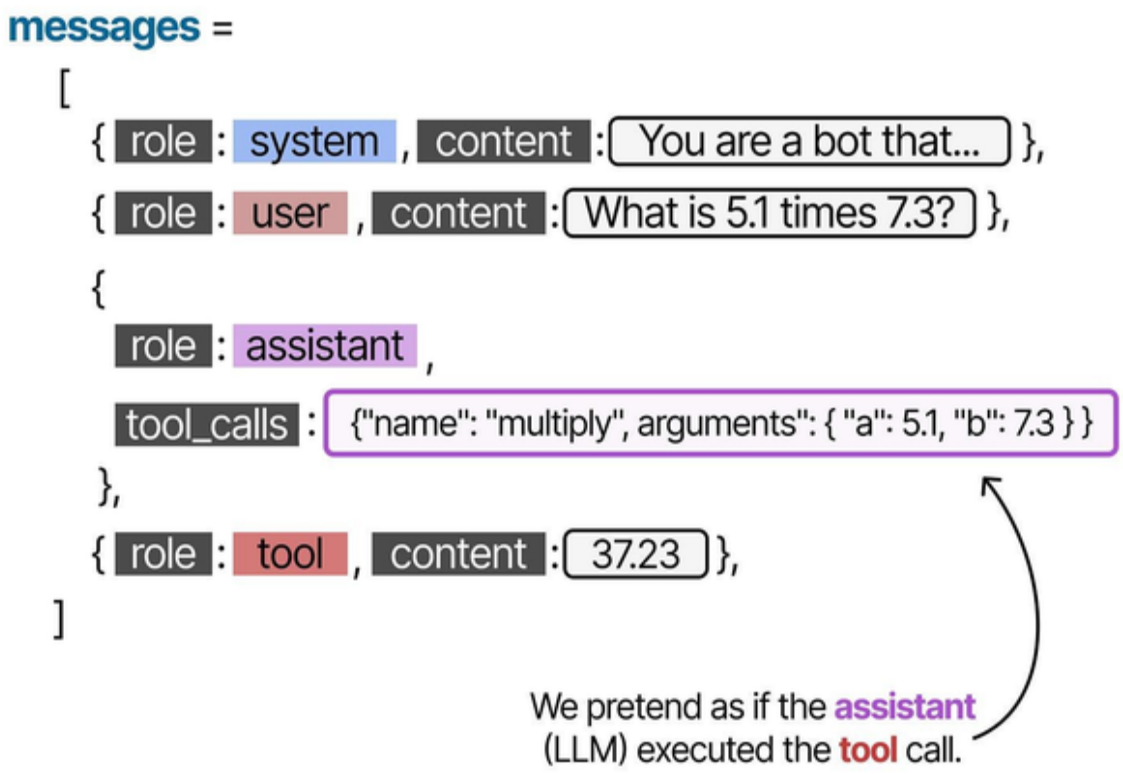
Note that all we have to do is extract the name and arguments from the JSON generated by the LLM. Then, we can call the tool with `tools[name](**arguments)` and generate our output (`tool_output`). This means that actually calling the tool is not executed by the LLM but by a system that we created. You can fully automate this process by wrapping it in a function that automatically extracts the JSON and then subsequently calls the tool.

Tool Output Processing

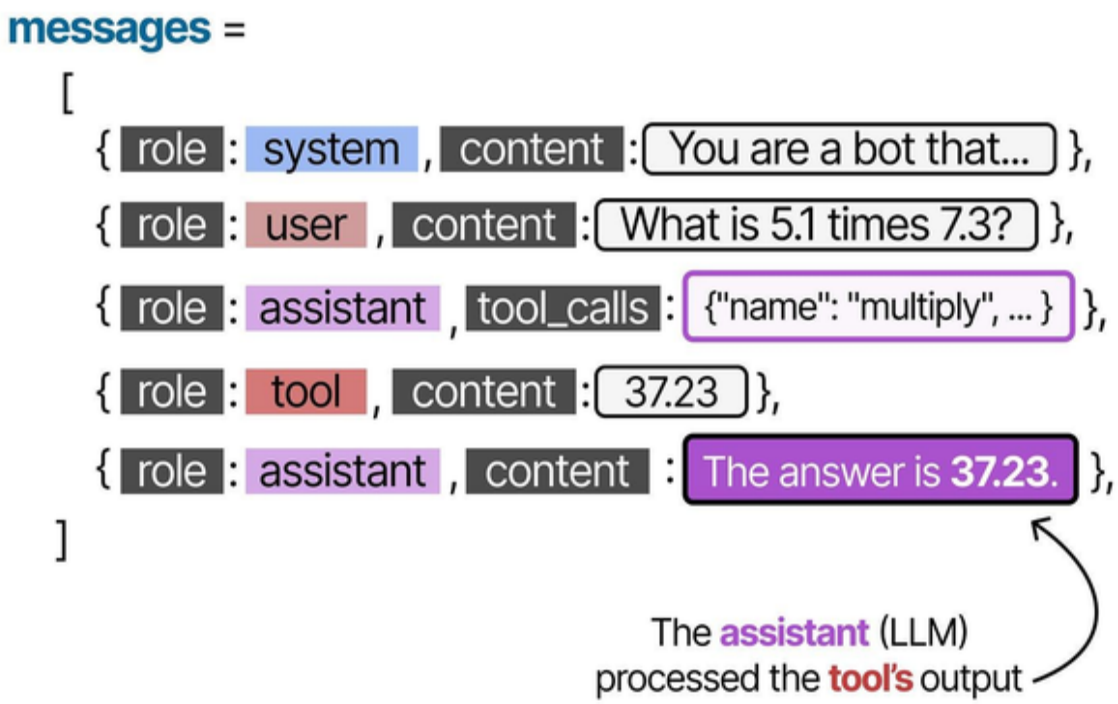
We still need to do something with the output of the tool call. In our example, the LLM ends with a tool call, which we process and call ourselves. To feed the output back into the LLM, we can use the `messages` structure that we explored in the previous chapter. Specifically, we can add messages with two roles:

- `assistant` — The assistant calls the multiply tool.
- `tool` - The output of the tool is returned.

By updating the messages to include this additional information, we are essentially informing the LLM that these steps were taken. Calling the tool was done outside of the LLM's view, and it therefore has no knowledge of what actually happened. As such, we pretend as if the LLM executed the tool whilst that was actually done by the user or an automated system. These updated messages are illustrated in Figure 5-10.



Finally, we can feed these messages back into the LLM to create our final answer. Although the tool has given us the appropriate output, the LLM might want to combine answers or present them in a nicer way. This process, along with the updated output, is presented in Figure 5-11.



With the many steps an LLM has to correctly follow to call the appropriate tool, it requires the LLM to be an effective orchestrator, which is very dependent upon the model's reasoning capabilities and overall reliability.

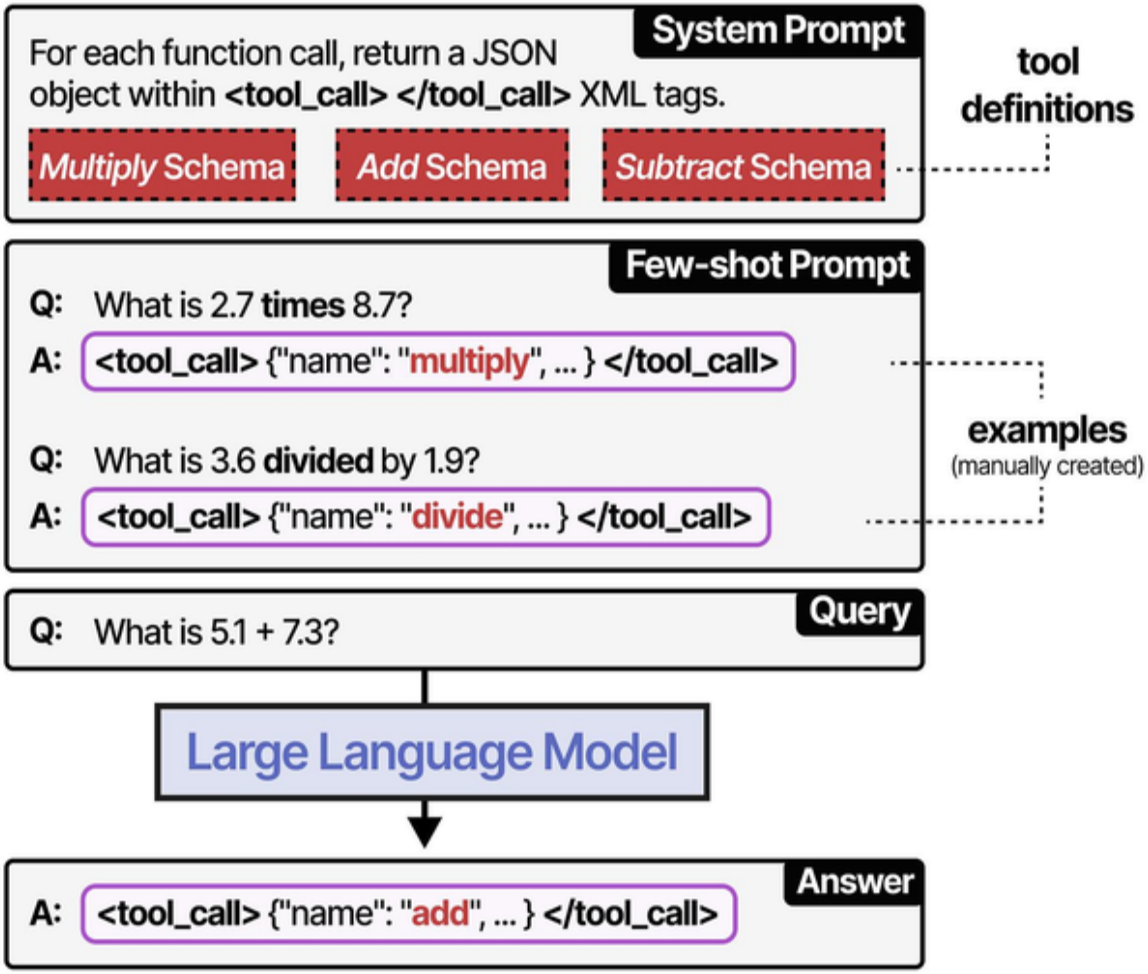
Tool Learning

Instilling tool-calling capabilities into LLMs can be a difficult task, especially when it has not been trained to do so. Learning tools can be achieved through various methodologies. In this section, we will explore the three most common categories of tool learning, namely in-context learning, supervised fine-tuning, and reinforcement learning.

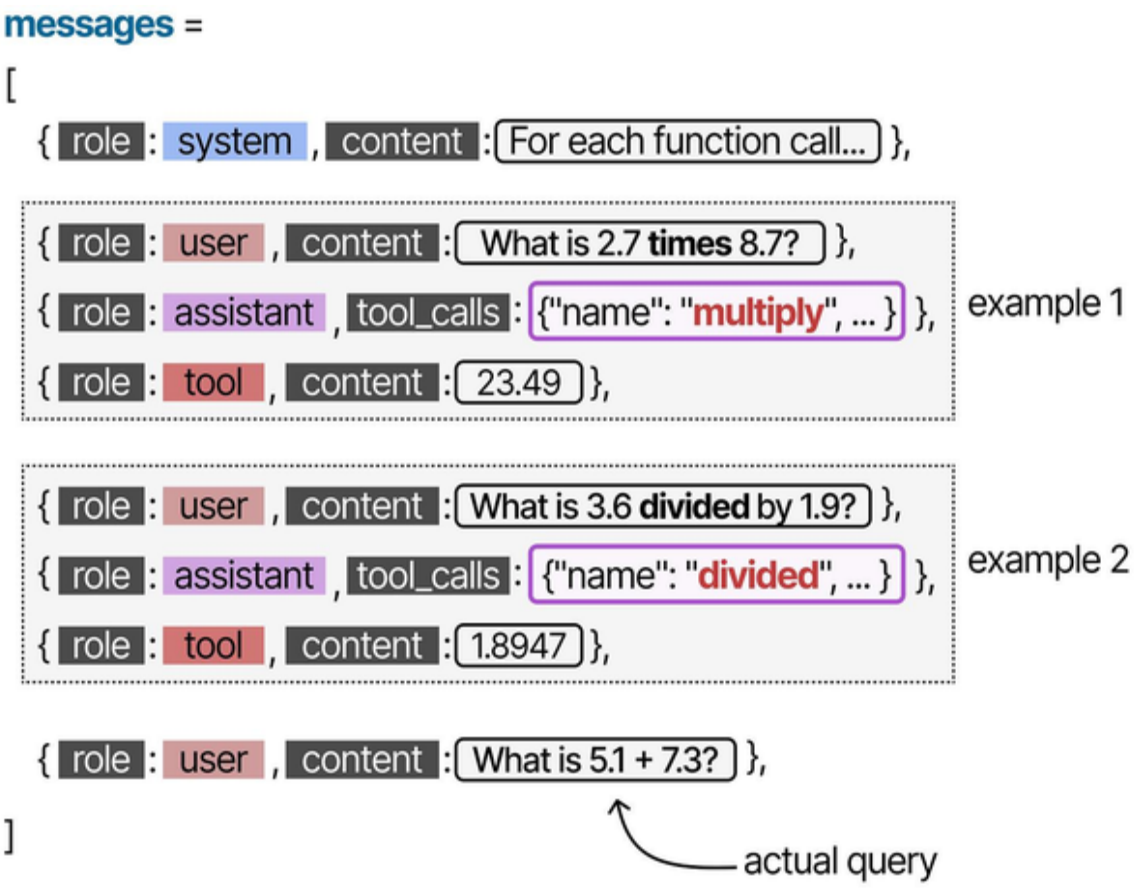
In-context Learning

In-context learning, the ability of LLMs to learn from a few examples in their context[6], is an exceptionally useful technique to enable new behavior in LLMs without the need to fine-tune them. As we covered in Chapter 3, it is also known as *few-shot prompting* or *few-shot learning*, where you

provide several examples to the LLM to learn from. Specifically, it is a prompt engineering technique where you give some examples of the behavior that you want the LLM to repeat. In our case, and as illustrated in Figure 5-12, we want the LLM to output the tool call in a specific format by following the examples we created ourselves.



In-context learning is especially helpful when leveraging the `messages` structure that we explored before. Remember when we pretended the assistant called a tool by adding it to the messages? We can apply the same concept and act as if the model were called tools before. As shown in Figure 5-13, we can create our messages in such a way that the model thinks it has already used some of the tools before. So when it subsequently gets a query, it has examples of how the tools should be called and what kinds of output can be expected.



Since LLMs are great at pattern recognition, providing these example tool calls helps them follow the structure of the tool call we had in mind. Note that in-context learning improves as the LLM becomes more capable. More performant models tend to better follow instructions and examples.

HuggingGPT

A great example of how far you can get by using in-context learning is HuggingGPT, an agentic framework that uses an LLM orchestrator to decide which tools are used to solve a given problem.[7] Interestingly, the tools are actually models themselves on the HuggingFace platform. The underlying idea is that the LLM could act as an orchestrator to manage existing AI models on the HuggingFace Hub to solve whatever problem it encounters.

Their pipeline revolves around proper prompt engineering and in-context learning to enable this orchestrator framework. It consists of four steps:

Task Planning

Extract the intention of the user's query and decompose it into solvable tasks

Model Selection

From the extracted tasks, select the appropriate model from HuggingFace's model hub

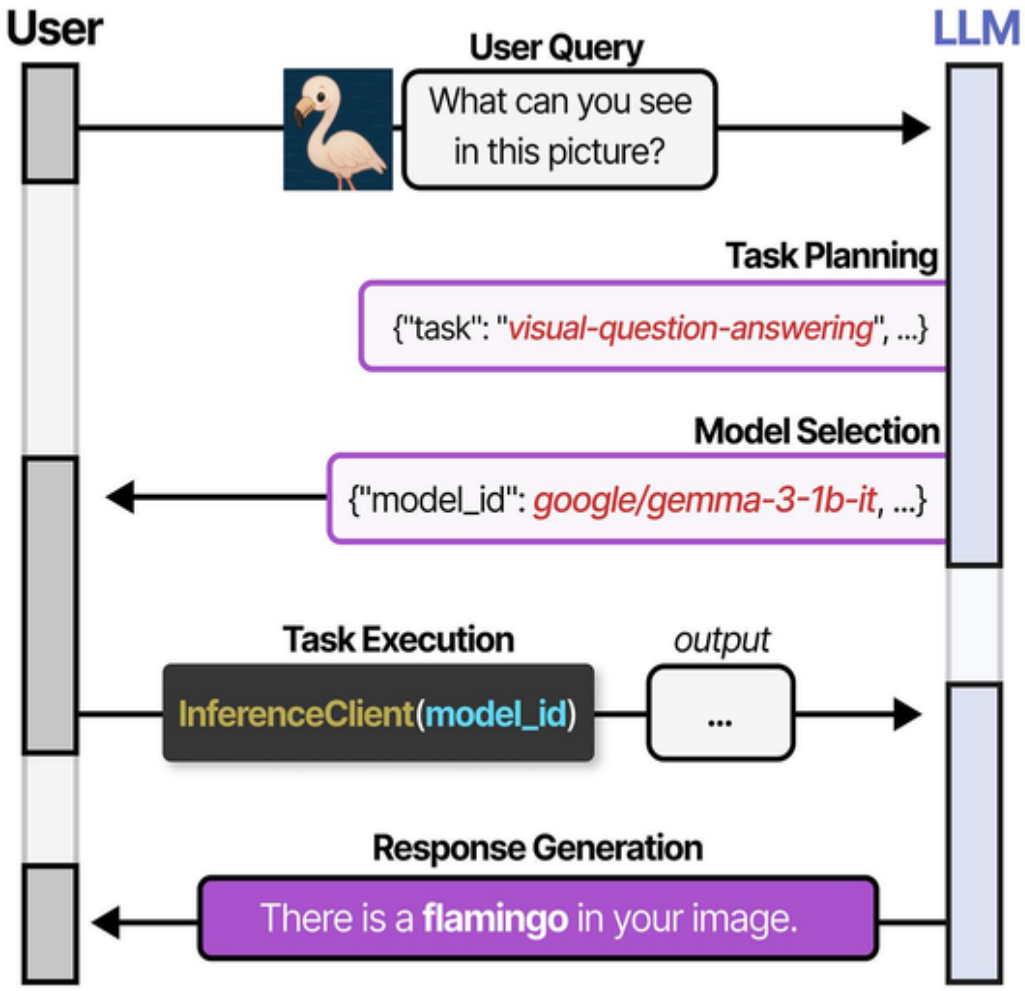
Task Execution

Call the selected models as tools.

Response Generation

Integrate the output of tools and generate a response

These steps are visualized in Figure 5-14.



Task Planning

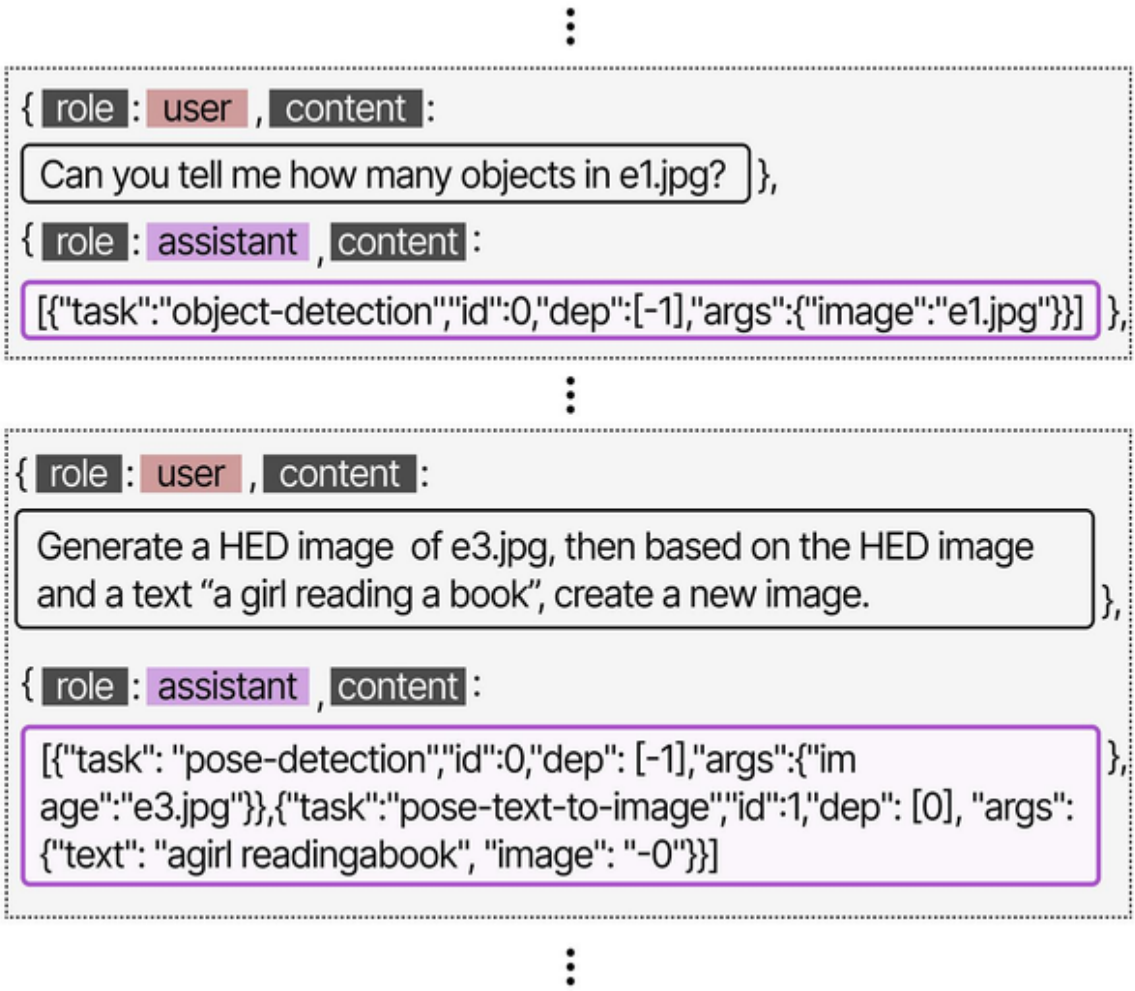
In *task planning*, the LLM decides which types of tasks are best suited for the given query. Tasks are essentially the categories of models that you can find on the HuggingFace Hub. A wide variety of tasks are available, ranging from text classification and summarization to image-to-text and object detection. In total, there are 24 tasks with potentially millions of models hosted on the HuggingFace Hub. The LLM chooses the task(s) and their order through a prompt design that consists of specification-based instruction demonstration parsing.

With specification-based instruction, instructions are provided on the formatting of tasks in JSON. It is a description of what we expect the output

should be rather than an example. A part of the full prompt is provided below that demonstrates this specification-based instruction:

```
"""
#1 Task Planning Stage
The AI assistant performs task parsing on user input, generating
a list of tasks with the following format: [{"task": task,"id",
task_id, "dep":dependency_task_ids, "args":{"text": text,
"image": URL, "audio": URL, "video": URL}}].

The "dep" field denotes the id of  the previous task which
generates a new resource upon which the current task relies.
The tag "<resource>-task_id" represents the generated text,
image, audio, or video from the dependency task with the
corresponding task_id.
The task must be selected from the following options:
{{AvailableTaskList}}.

Please note that there exists a logical connection and order
between the tasks.

In case the user input cannot be parsed, an empty JSON response
should be provided. Here are several cases for your reference:
{{Demonstrations}}.

To assist with task planning, the chat history is available as
{{ChatLogs}}, where you can trace the user-mentioned resources
and incorporate them into the task planning stage.
"""
```

To better understand this JSON parsing, examples are given of queries and outputs through in-context learning, which they named demonstration-based parsing. Illustrated in Figure 5-15 is an example of how these examples were structured.



Model Selection

After selecting the appropriate tasks to execute, the models within those specific tasks still need to be chosen. Below is the prompt template used for asking the agent to select the best model for the given task.

```
"""
#2 Model Selection Stage
Given the user request and the call command, the AI assistant
helps the user to select a suitable model from a list of models
to process the user request.

The AI assistant merely outputs the model id of the most
appropriate model.

The output must be in a strict JSON format:
{"id":"id", "reason":"your detail reason for the choice"}.
```

```
We have a list of models for you to choose from
{{CandidateModels}}. Please select one model from the list.
"""
```

Since each task on the Hugging Face Hub can potentially have hundreds of thousands of models, only a subset is extracted from the tasks that were selected in the previous step. Specifically, the authors used the download count on the Hub as a proxy for the model quality and extracted the Top-K models based on that ranking. This selection is contained in the `CandidateModels` variable and formatted like so:

```
{"model_id": id#1, "metadata": info#1, "description": model
description #1}
{"model_id": id#2, "metadata": info#2, "description": model
description #2}
...
{"model_id": id#K, "metadata": info#K, "description": model
description #K}
```

Task Execution

Next, each selected model is executed with their relevant arguments. Models are run in parallel if possible. For example, if prompted to generate summaries of different PDFs, separate models can run in parallel to execute this task. This resource dependency is carefully tracked to decide which models can be run in parallel and which models require other models to have a completed run.

Response Generation

Finally, the inputs and outputs of all previous steps are given to the model to generate the final response. This includes the user’s input, task planning, model selection, and task execution. The prompt details how the agent should parse the results and effectively create a summary.

```
"""
# 4 Response Generation Stage
With the input and the inference results, the AI assistant needs
to describe the process and results.
```

```
The previous stages can be formed as:

- User Input: {{ User Input }}
- Task Planning: {{ Tasks }}
- Model Selection: {{ Model Assignment }}
- Task Execution: {{ Predictions }}

You must first answer the user’s request in a straightforward
manner. Then describe the task process and show your analysis and
model inference results to the user in the first person. If
inference results contain a file path, must tell the user the
complete file path. If there is nothing in the results, please
tell me you can’t make it.
"""
```

You must first answer the user’s request in a straightforward manner. Then describe the task process and show your analysis and model inference results to the user in the first person. If inference results contain a file path, must tell the user the complete file path. If there is nothing in the results, please tell me you can’t make it.

Note that this framework used older LLMs like Alpaca-7b, Vicuna-7b, and GPT-3.5 to evaluate its performance. These models were not specifically trained on tool-calling tasks, but demonstrate tool-calling behavior from in-context learning. As such, HuggingGPT is a great example of how far you can go with structured prompting and in-context learning.

Supervised Fine-tuning

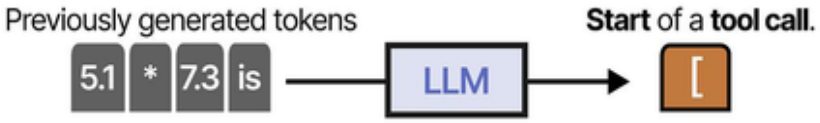
Although prompting is a straightforward method for enabling tool usage, it does require filling up the context window with additional instructions. As we explored in the previous chapter, we want to prevent filling up the context window, as it might degrade the performance of your model. There is also a risk of the model not following instructions through prompting if you have too many instructions.

A great alternative is to train your model using supervised fine-tuning to distill knowledge and capabilities on tool calling into the model itself. This was especially popular in 2023 and the beginning of 2024 as it proved to be an effective and relatively cheap method for adding tool calling capabilities.

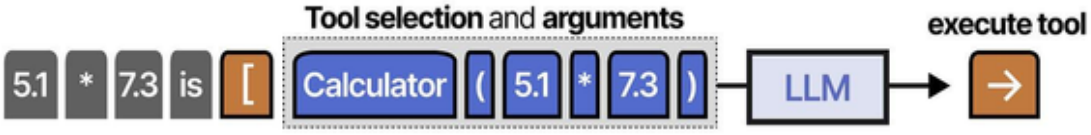
ToolFormer

An interesting technique to explore supervised fine-tuning for tool calling (including some other tricks) is Toolformer, a model trained to decide which APIs to call and how.[8]

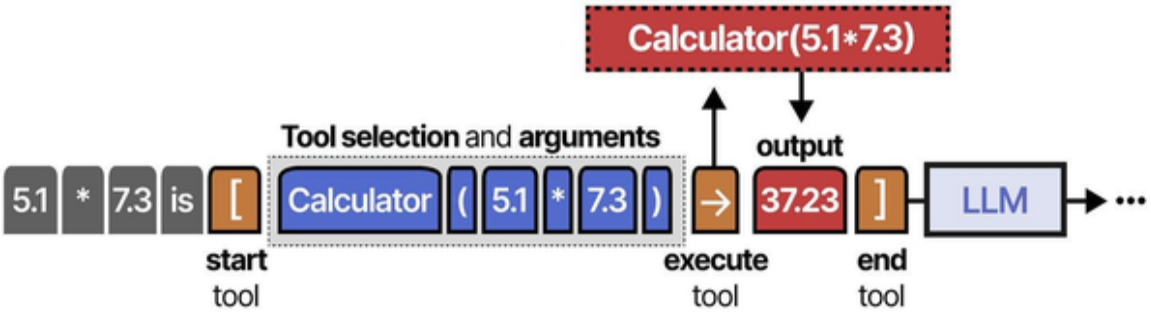
Before exploring the training process, let's first dive into their approach to calling a tool. In previous examples, the tool that should be called was structured using JSON schemas, and the output of the call was sent back to the LLM to process. Toolformer takes a different approach by embedding the call and its output in the text it's generating. It does so by using the [and] tokens to indicate the start and end of calling a tool. When given a prompt, "*What is 5.1 times 7.3?*", it starts generating tokens until it reaches the [token (shown in Figure 5-16).



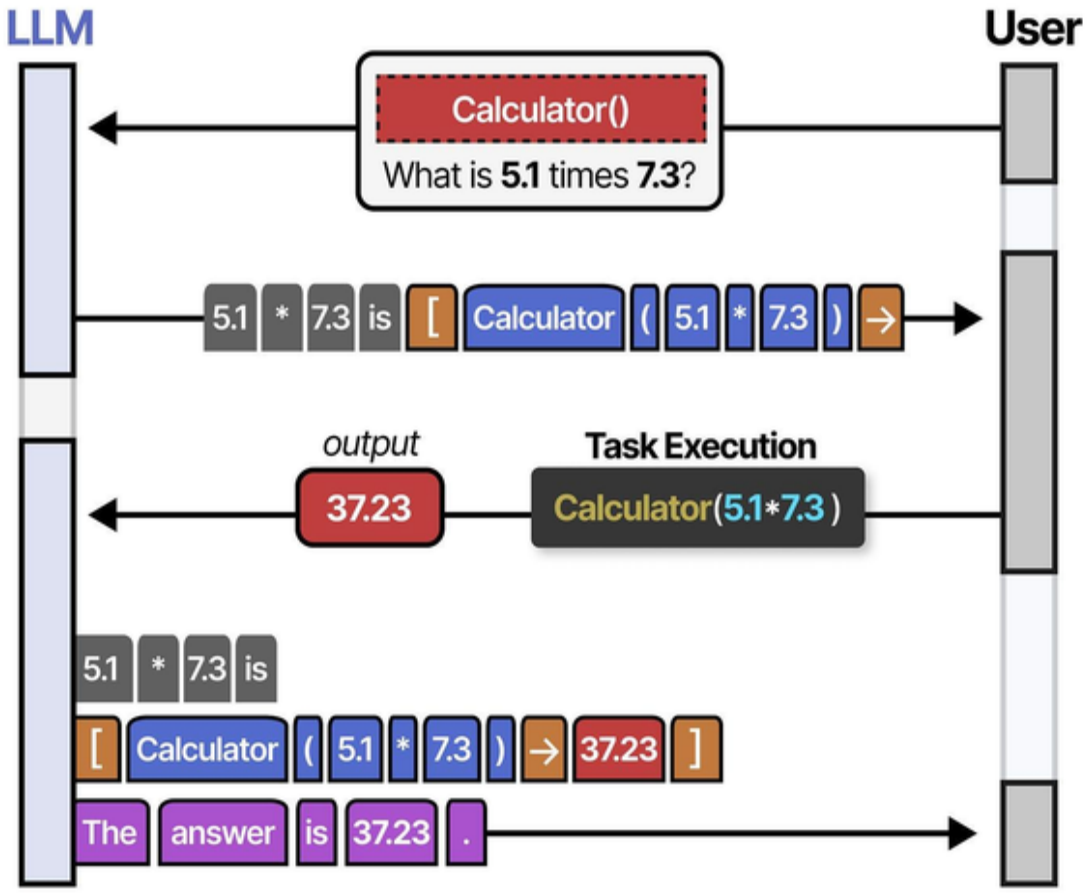
The [token tells the LLM that it should start selecting the tool and generate the appropriate parameters following a fixed format (more on this later!). It does this until it reaches the → token (Figure 5-17).



The → token tells the LLM that it can now completely stop generating tokens will the user or an automated system executes the selected tool. Instead of appending the output through the messages structure as we explored before, it is added to the previously generated tokens along with the] token (shown in Figure 5-18).



The] token tells the LLM that the tool has been executed and that the tool call, along with the output, is now part of the previously generated tokens. Then, the LLM can decide to continue generating tokens if necessary. Figure 5-19 demonstrates the end-to-end process of calling the tool, adding the output during generation, and giving back an answer.



The input/output of this process is therefore as follows:

Input

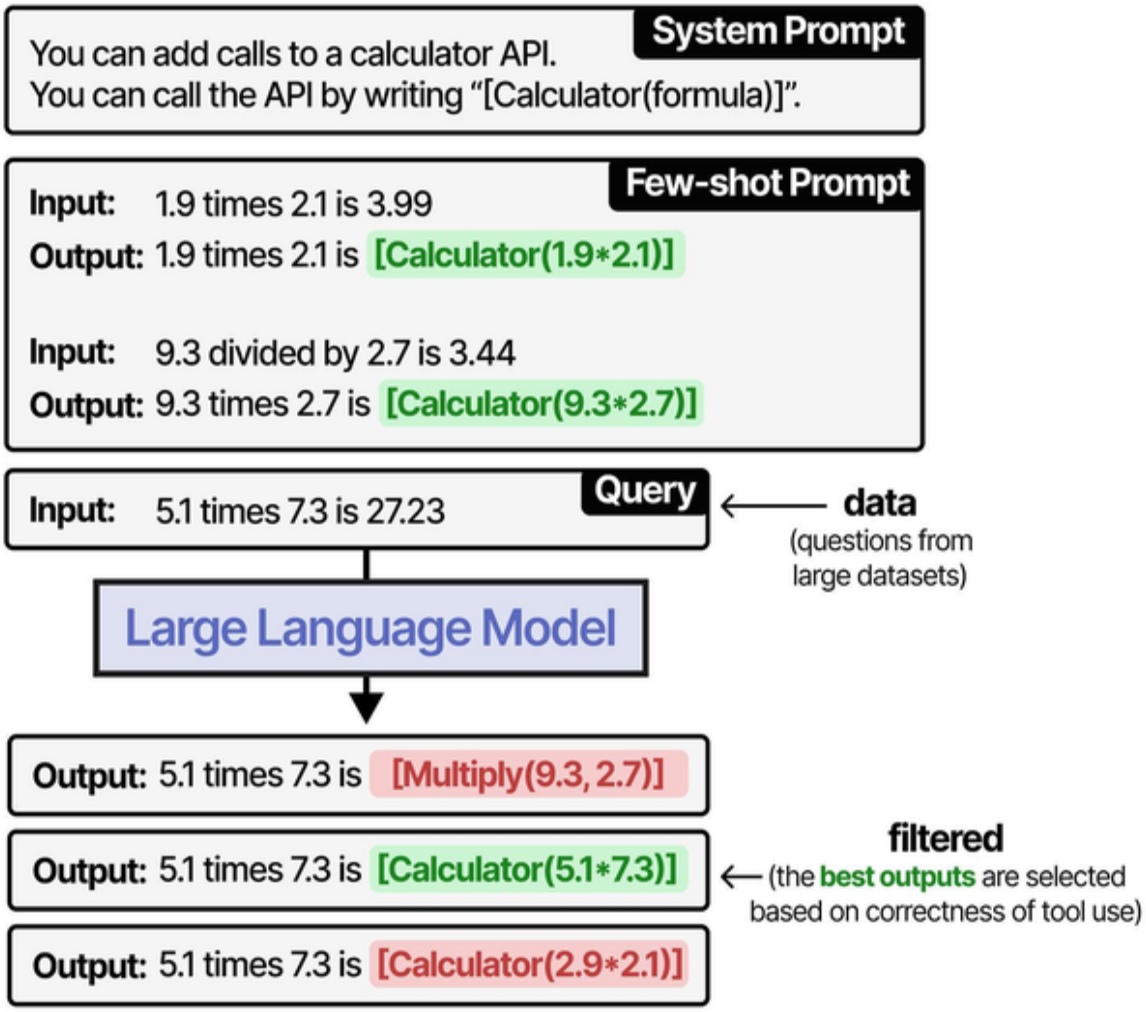
“What is 5.1 times 7.3?”

Output

“5.1 * 7.3 is [Calculator(5.1*7.3)→37.23] The answer is 37.23.”

This process of adding the output whilst generating tokens is an important construct and is often used in various models. By embedding the output during generation, the model uses tools more naturally and generates fluent language. It also lends itself quite well for reasoning LLMs, where tools are called during the reasoning process instead of having to go back and forth between messages (remember Search-o1 from the previous chapter?).

To enable this in-line tool calling behavior in Toolformer, the model needs to be fine-tuned first. To do so, the authors carefully generated a dataset with many different examples of tool use. For each tool, a few-shot prompt was manually created and used to sample outputs. By sampling different outputs (much like the sampling with Search against Verifiers we explored in the reasoning chapter), the best output could be extracted by filtering on the correctness of the tool use, output, and loss decrease. The full process of creating this dataset is shown in Figure 5-20.



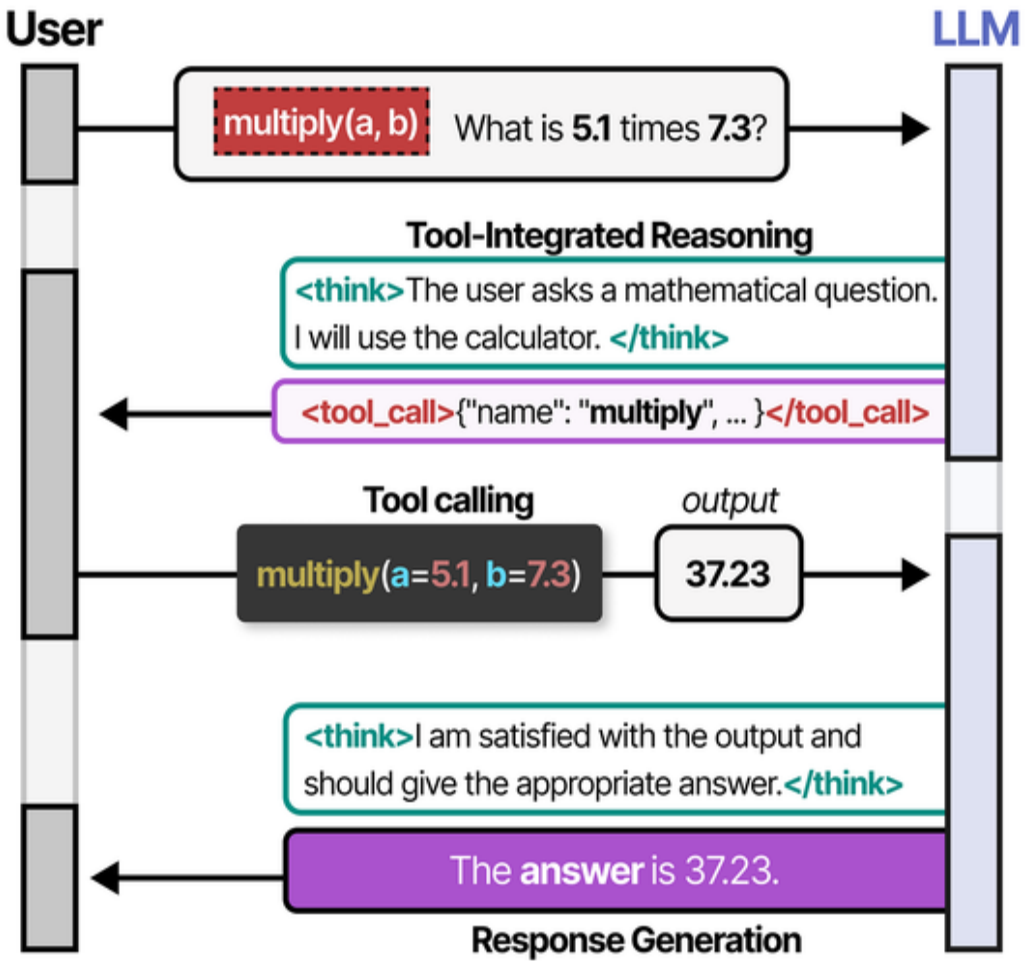
In other words, the original data was adjusted to include tool calls instead of generating the answer by the LLM itself. All inputs were essentially updated to contain the appropriate tool.

Finally, the LLM (GPT-J, a variant of GPT-3) was fine-tuned using this updated data using supervised fine-tuning. At the time, this technique showed significant improvements over zero-shot performance and competed with larger models. However, supervised fine-tuning on this data made generalization difficult. Supervised fine-tuning tends to be sensitive to the exact wording and prompt that is being used because it attempts to recreate what it is being shown. As we will explore next, Reinforcement Learning tends to be a much more stable technique for generalization in tool use.

Reinforcement Learning

As we explored in Chapter 3, reinforcement learning is an excellent method of training or fine-tuning to align your model to certain rewardable tasks. Compared to supervised fine-tuning, where an LLM trains on fixed examples with the correct answer, reinforcement learning relies on trial and error. In reinforcement learning, LLMs develop improved reasoning strategies not from being told exactly what to do (the mimicking behavior of supervised fine-tuning) but from repeatedly exploring feedback signals.

In the context of reinforcement learning, tool-learning is often integrated into the thinking process of models that support advanced reasoning. This is called Tool-Integrated Reasoning (TIR), which involves incorporating tools into the reasoning traces of an LLM. Figure 5-21 illustrates this process of calling a tool *during* reasoning and continuing the reasoning process after receiving the output.



Note that such a TIR trajectory might involve multiple tool invocations, where the final answer is determined by all these intermediate tool calls and outputs.

Although we will delve more deeply into reinforcement learning in Chapter X, let's explore how it can be used to enable tool learning and Tool-Integrated Reasoning.

ToolRL

A recent example showcasing how reinforcement learning can be used to enable TIR is ToolRL.[9] This framework uses GRPO, which we briefly covered in DeepSeek-R1's training procedure in Chapter 3. To differentiate between stages of thinking, tool calling, and answering the query, they used

the `<thinking></thinking>`, `<tool_call></tool_call>`, and `<answer></answer>` tokens, respectively.

Compared to DeepSeek-R1, their usage of rewards in GRPO is quite straightforward. Two rewards are defined to enable tool usage:

Correctness

Is a tool called correctly?

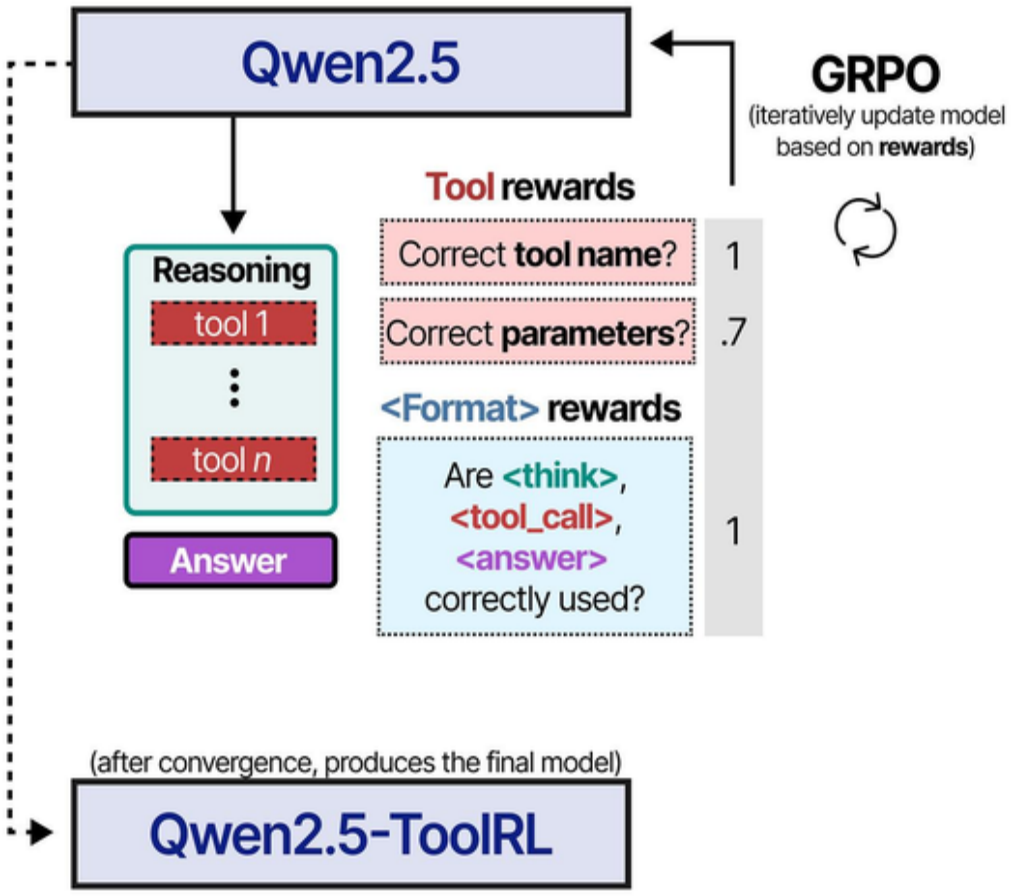
- Scores are given based on whether the correct tool names and parameters were used.

Format

Is the appropriate format used?

- A positive reward is given if all required fields appear in the correct order.

To train the model, 4,000 samples of tool-integrated reasoning traces were sampled from various datasets and used for fine-tuning various Qwen2.5 models[10]. Figure 5-22 illustrates how GRPO was used to fine-tune these models.



Interestingly, the authors also experimented with length rewards to encourage longer reasoning traces but found that longer traces do not consistently improve task performance and may even harm smaller models. Long reasoning traces might therefore not be ideal for tool use tasks.

Note that GRPO is a very flexible framework and allows you to develop the rewards that are best suited for a given use case. As such, this strategy of using tool-based rewards in GRPO can also be used for non-reasoning models by simply removing or updating the format reward.

Search-R1

To further explore what reinforcement learning is capable of, let’s take a closer look at Search-R1, an efficient reinforcement learning framework for integrating search as a tool into an LLM’s reasoning process.[11] In this

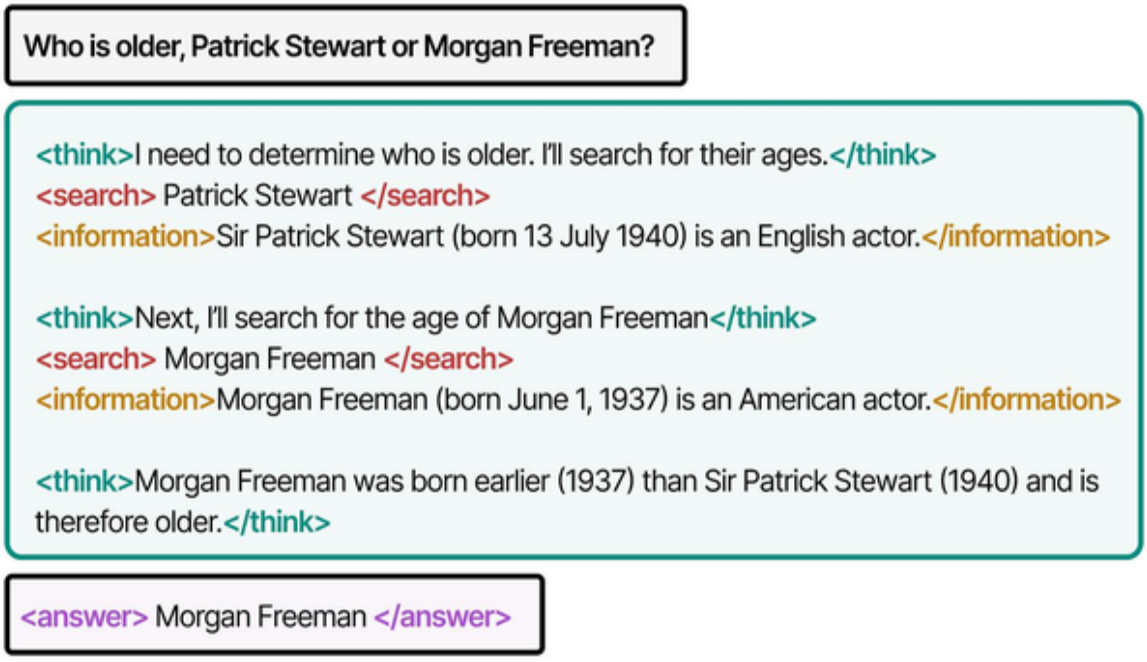
framework, the LLM learns to generate one or more search queries during step-by-step reasoning autonomously.

The framework starts with specifying how the model should interleave reasoning with the search engine call. As illustrated in Figure 5-23, the prompt template is structured into three parts. First, the reasoning traces are created with `<think></think>` tokens, then the search engine calling function with `<search></search>` where the output is reintegrated with `<information></information>`, and finally, the answer through `<answer></answer>` tokens. Note that the reasoning traces and the search engine can be interleaved several times.

> Answer the given question. You must conduct reasoning inside <think> and </think> first every time you get new information. After reasoning, if you find you lack some knowledge, you can call a search engine by <search> query </search> , and it will return the top searched results between <information> and </information>. You can search as many times as you want. If you find no further external knowledge needed, you can directly provide the answer inside <answer> and </answer> without detailed illustrations.

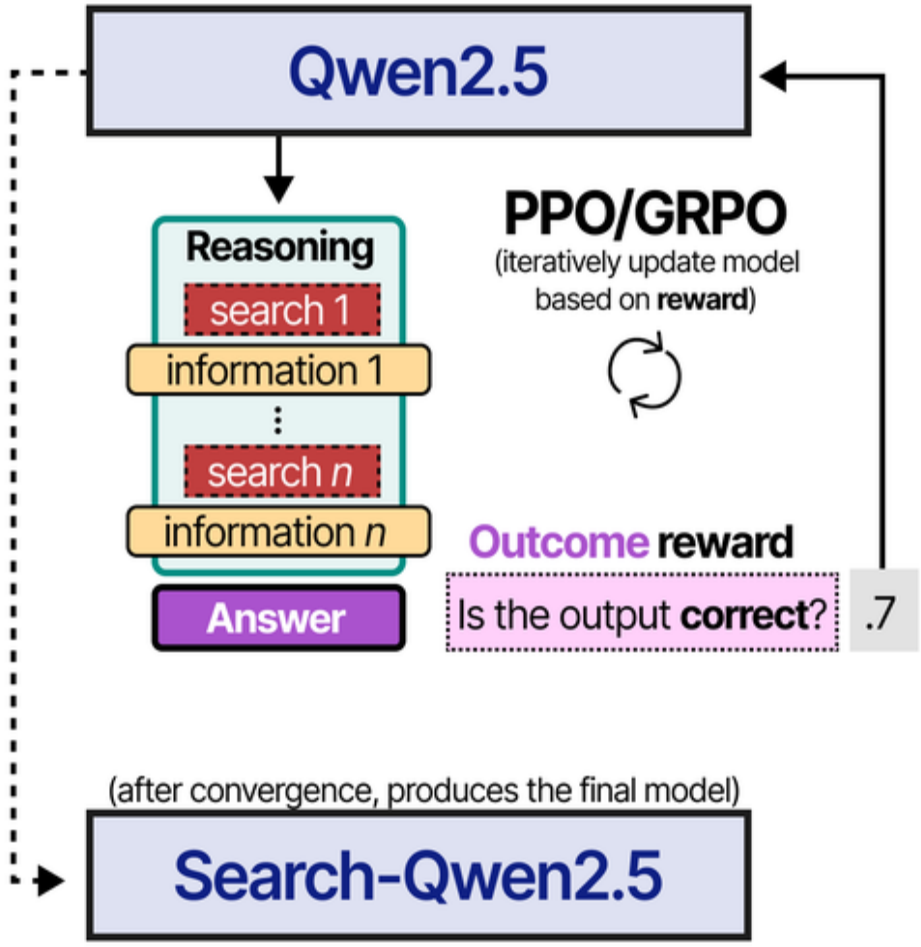
What makes this template particularly interesting is that the authors focus on a single tool, search. The reason for this was the upcoming popularity of DeepResearch, a framework where reasoning LLMs are coupled with search engines to create agentic systems that allow for in-depth research on various topics. The authors of Search-R1 created this framework as a strong open-source alternative to the proprietary systems out there.

The result of such a template is, like ToolRL, tool-interleaved reasoning with multi-turn search engine calls. We illustrated an example of such a process in Figure 5-24. Note that the `<search></search>` tool can be any application, like the open-access archive of academic papers arXiv, or a combination of various sources.



The approach of training the algorithm with reinforcement learning is rather straightforward. The authors adopted a simplified outcome-based reward function. Rather than creating all different kinds of formatting and accuracy rewards (like DeepSeek-R1), only accuracy rewards were used (based on the task). Since the underlying model (Qwen-2.5) already has strong structural adherence, there was no need for formatting rewards.

The authors explored both PPO and GRPO as reinforcement learning algorithms, as illustrated in Figure 5-25. Both are, at the time of writing (September 2025), two of the most popular methods of reinforcement learning for LLMs, as we will explore more in depth in Chapter X. Note that the losses in PPO and GRPO are typically calculated over the entire sequence of tokens, including the output of the search engine. In Search-R1, the tokens of the search engine's output were masked (ignored) to prevent the model from attempting to control the search engine's output, which were not directly LLM-generated (which can create unexpected dynamics). This is called *Loss Masking for Retrieved Tokens*.

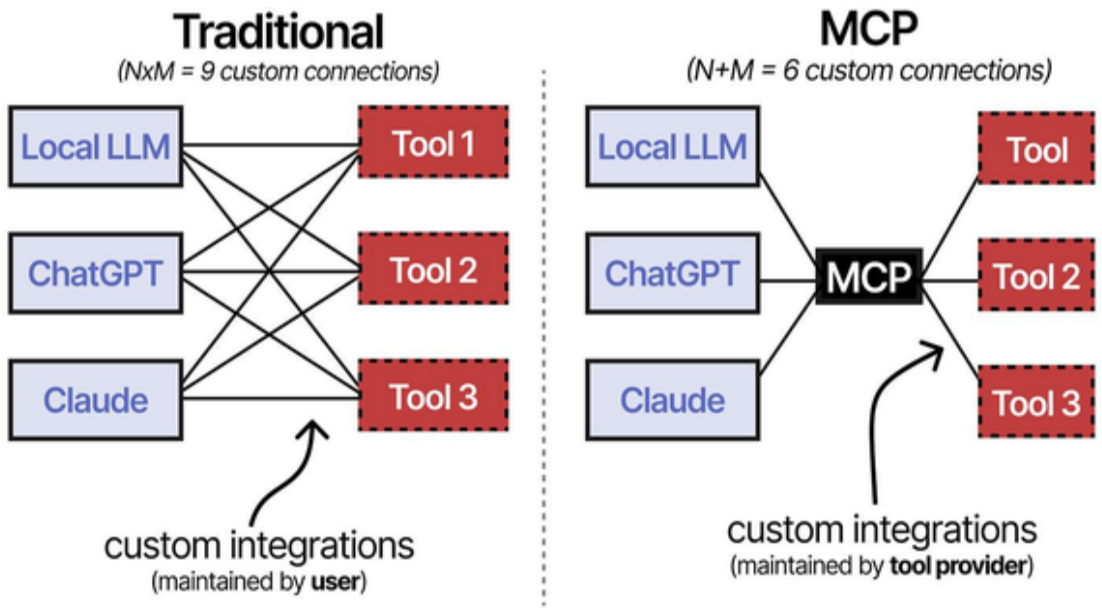


These kinds of frameworks and methodologies leveraging reinforcement learning have become increasingly popular as the reward structure for tool calling is generally quite straightforward. It is trivial to check if a tool has been called correctly and if the correct arguments have been used (like in ToolRL). Moreover, reinforcement learning works well for rewards that are verifiable, like coding and tool calling. As such, there has been an increase in models that were trained using reinforcement learning and additionally adopted tool-based rewards, like the strong open-source Qwen3[12] and GPT-OSS[13] models.

Model Context Protocol (MCP)

In the previous sections, we explored how to connect tools to LLMs, making them capable of much more than text generation. Although their ability to then use tools is incredible, it is not a free lunch. Imagine you have developed several prompts to instruct your LLM on how to use tools. As is typical in this field, a new LLM release that you would like to try out, together with all the tools. Now imagine it has a new way of calling tools, which means you will have to create new integrations for all your tools. This is the NxM problem, where N is the number of LLMs and M is the number of tools available. You would have to write custom integrations for every LLM/tool combination.

Model Context Protocol (MCP) solves this problem by standardizing how you would connect tools and APIs with different structures to your LLM. It is an open standard and framework developed by Anthropic.[14] As a protocol, it facilitates two-way communication between tools and LLMs. It is often referred to as the "USB-C port of AI" due to its universal nature, allowing for any LLM to implement any tool that follows this protocol. Seen in Figure 5-26, instead of manually creating connections between LLMs and tools, MCP creates only a single connection that can be maintained indefinitely by the tool provider.



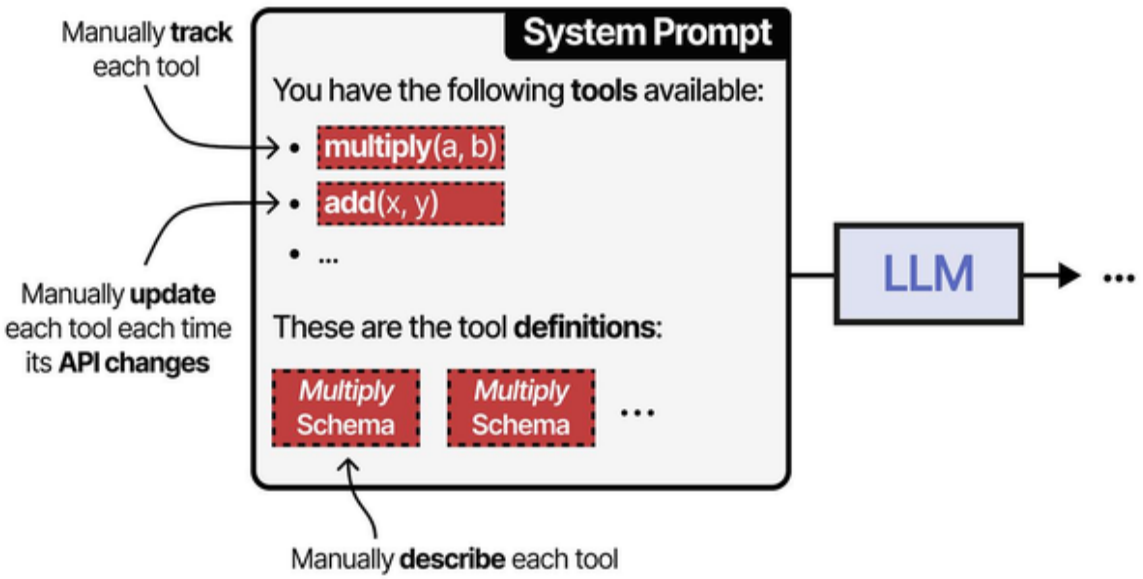
By having the MCP server handle the integrations and communicate the tools to the LLMs, it becomes N+M connections that need to be maintained instead of NxM connections. Moreover, as long as the tool provider has an MCP server, connecting the server to your LLM is relatively straightforward, but more on that later!

The maintenance of the integration, therefore, also moves from the user to the tool provider. Without MCP, if the arXiv's API were to suddenly change drastically, then each user would have to adjust their integrations. With MCP, any changes to the API would need to be resolved only once by the maintainers of that API. Then, the updates can be rolled out to all users without any intervention from their side.

To illustrate this point a bit further, if you were to add tools to your LLM manually, all tools would have to be:

- manually **tracked** and fed to the LLM,
- manually **described** (including its expected JSON schema)
- manually **updated** whenever its API changes

As shown in Figure 5-27, this can be quite the hassle for maintaining your tools.



Thus, MCP not only solves the NxM problem but also the problem of standardization. Note that MCP is not the only protocol for standardizing communication, like Agent 2 Agent (A2A) for standardizing inter-agent communication.[15] Although there are others, in 2025, it is arguably one of the most popular protocols out there.[16]

Core Components

To achieve all these amazing capabilities, MCP consists of four components:

MCP Server

: Provides context, tools, and capabilities to the LLMs

MCP Host

: Llm application (such as Cursor) that manages connections

MCP Client

: Maintains one-to-one connections with MCP servers

Resources

: The tools, data, or services that are provided locally or remotely

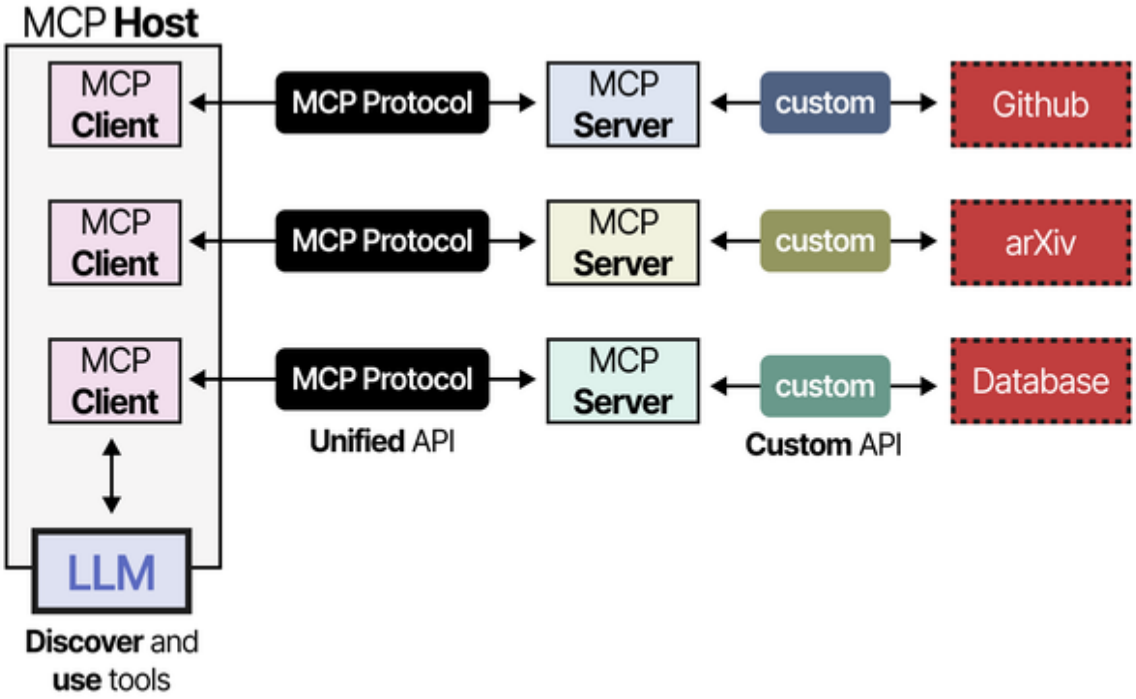
The *MCP Host* is any application that uses an LLM to use external tools. Typical examples are chat assistants like ChatGPT or Claude and IDE extensions like Cursor or GitHub Copilot. This is the “brain” of the MCP flow and makes calls to the MCP Servers via the MCP Clients.

The *MCP Client* maintains connections with the MCP servers. They exist within the host and handle the connection management, discovery of tool capabilities, request forwarding, etc. Compared to the host, it is a piece of code that handles the communication with the MCP Servers, whereas the MCP Host only initiates the communication.

The *MCP Server* is a lightweight program that exposes APIs and tools via the MCP standard. These servers often connect to a specific data source or service. For instance, an MCP server might connect to all API endpoints of arXiv to search, load, and view academic papers.

To sum up, the **MCP Host** (e.g., GitHub Copilot) contains an **MCP Client** that connects to multiple **MCP Servers** (which provide tools, resources, or data sources).

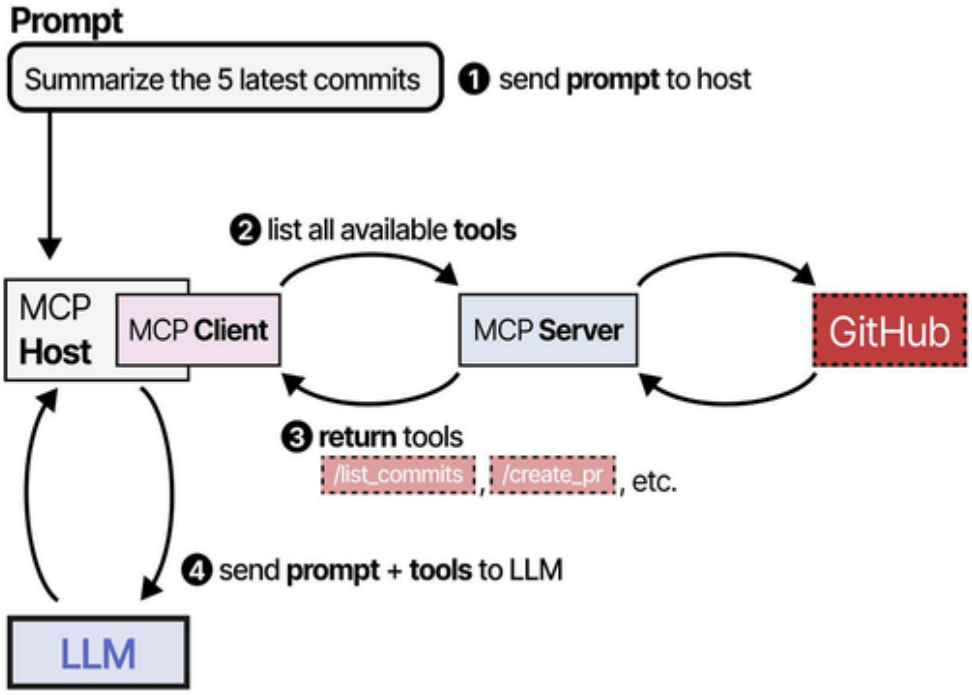
These three components are visualized in Figure 5-28.



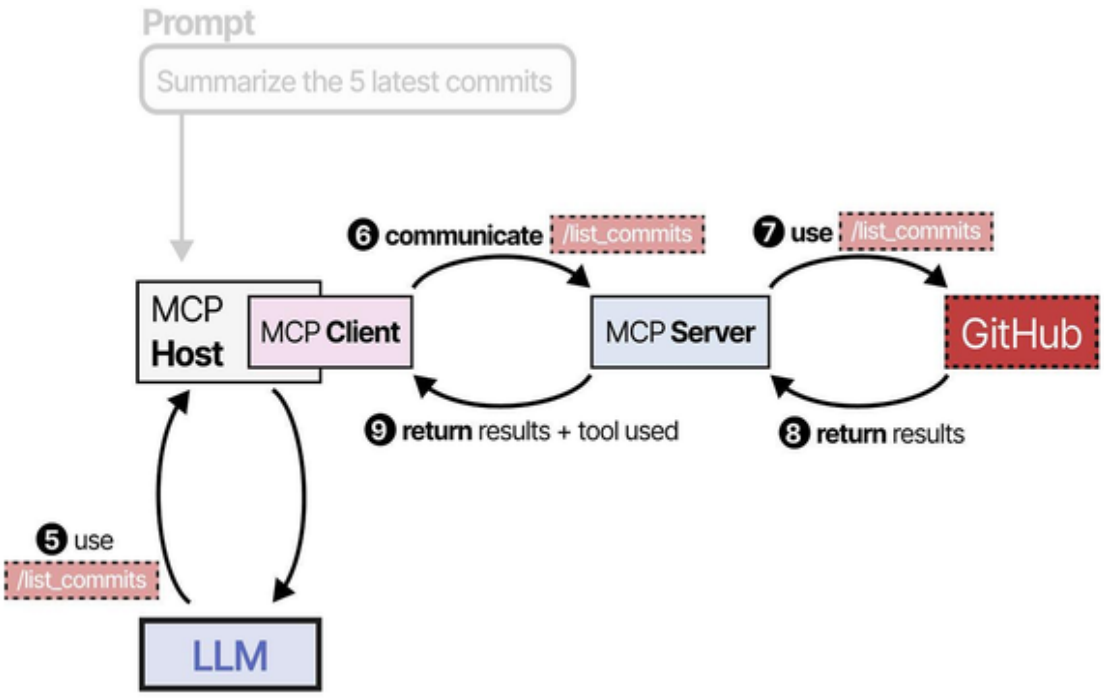
The MCP Flow

MCP can be a bit of a mystery, even when showing and describing the core components. Instead, let us go through an example of what it would be like to use the MCP to discover and call tools. Imagine you want your AI assistant (perhaps GitHub Copilot or Claude code) to summarize the 5 latest commits from your repository. This flow is numbered in the upcoming Figures so we can accurately track each and individual step.

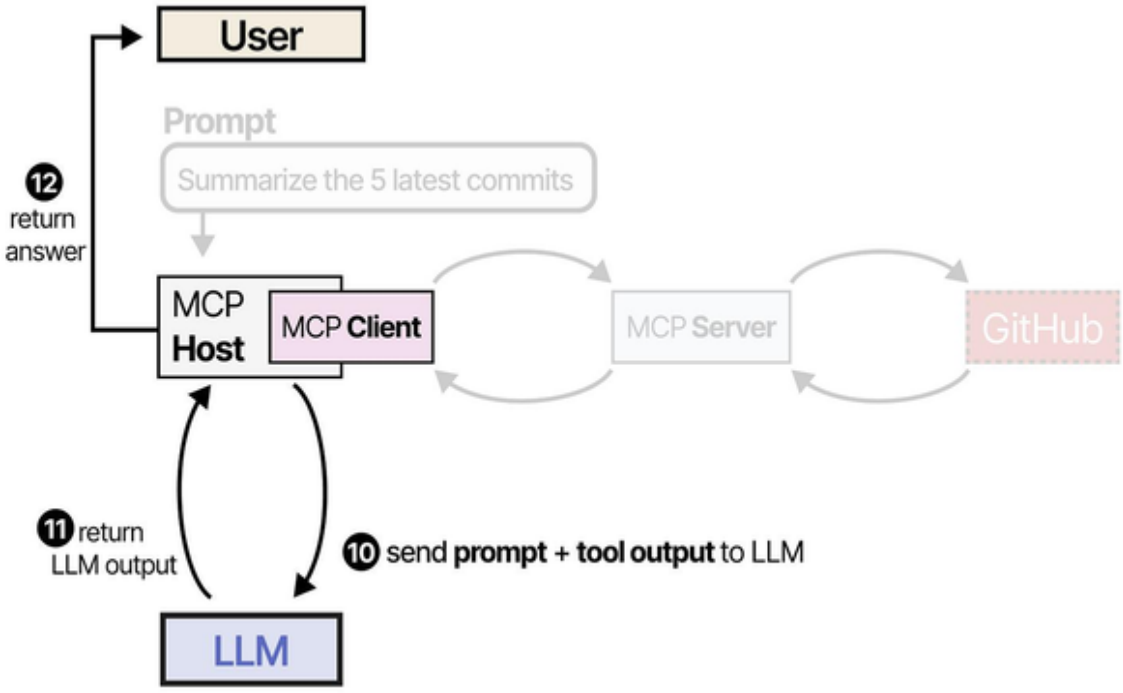
It would all start with the user's query: "Summarize the 5 latest commits". Seen in Figure 5-29, this prompt is sent to the MCP Host (1), which asks the MCP Server, through the MCP Client, which tools are available (2). The MCP Server is connected to the set of tools (GitHub) and returns the list of all available API calls back to the MCP Host (3). API calls might include common methodologies like listing all commits (`/list_commits`) or creating a pull request (`/create_pr`). Then, the initial prompt, together with available tools, is sent to the LLM (4).



Next, the LLM may choose to use any of the tools that were returned (Figure 5-30). Since the user's query is about commits, the LLM decides that it wants the MCP server to use the /list_commits tool (5). The MCP Client communicates this action to the MCP Server (6), which finally executes the command (7). The output of the tool usage is returned to the MCP server (8), which communicates it back to the MCP Client and Host through the MCP Protocol (9).



When the LLM receives the results (10), it can choose to run another tool or return the output to the MCP Host and then to the user. In our example, the LLM decides to summarize the 5 latest commits that it received (11) and return the summary back to the user (12). See Figure 5-31.



What makes these sets of steps so special is that LLM can discover tools that exist, choose which one to use, and does not have to think much about deprecated API functionalities.

Note that the LLM should still have tool-calling capabilities. Whenever it wants to execute a given tool, it should follow the MCP protocol, which follows a JSON-like structure. This structure, following the JSON-RPC 2.0 Specification, is also communicated by the MCP Client, which serves as the middleman between the LLM and the protocol.

Summary

In this chapter, we explored how tool calling makes LLMs have the capabilities to interact with the world. We first covered the fundamentals of tool calling and how LLMs call tools in practice. We saw that as text-to-text entities, LLMs merely communicate the intent to call a tool. The act of actually calling the tool falls either to the user themselves or to external software that automates this process. The flow of calling a tool involved the tool creation, definition, selection, calling, and output processing.

Then, we explored three methods of having LLMs learn those tool-calling capabilities. First was in-context learning, where you can define the tool's schemas and definition in the prompt for the LLM to follow. We covered HuggingGPT as a great example of in-context learning to enable tool calling. Second was supervised fine-tuning, where an LLM is fine-tuned on specific tool-calling capabilities. We covered ToolFormer as one of the first successful attempts to use supervised fine-tuning for tool calling. Lastly, we covered reinforcement learning as one of the most prominent techniques for instilling tool-calling capabilities. Of note were ToolRL and Search-R1, which both adopt GRPO, a popular reinforcement learning algorithm used in DeepSeek-R1.

We ended the chapter with one of the most exciting things in the realm of tool calling, the Model Context Protocol (MCP). We explored how MCP standardizes the usage of tools, which led to the widespread usage of tools without the need for custom solutions.

---

1 Wang, Zhiruo, et al. "What are tools anyway? a survey from the language model perspective." *arXiv preprint arXiv:2403.15452* (2024).

2 Liu, Bang, et al. "Advances and challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems." *arXiv preprint arXiv:2504.01990* (2025).

3 Wang, Lei, et al. "A survey on large language model based autonomous agents." *Frontiers of Computer Science* 18.6 (2024): 186345.

4 https://platform.openai.com/docs/guides/function-calling

5 https://platform.openai.com/docs/guides/function-calling

6 Dong, Qingxiu, et al. "A survey on in-context learning." *arXiv preprint arXiv:2301.00234* (2022).

7 Shen, Yongliang, et al. "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face." *Advances in Neural Information Processing Systems* 36 (2023): 38154-38180.

8 Schick, Timo, et al. "Toolformer: Language models can teach themselves to use tools." *Advances in Neural Information Processing Systems* 36 (2023): 68539-68551.

[9] Qian, Cheng, et al. “Toolrl: Reward is all tool learning needs.” *arXiv preprint arXiv:2504.13958* (2025).

[10] Qwen, “Qwen2.5 Technical Report.” arXiv:2412.15115, 2024.

[11] Jin, Bowen, et al. “Search-r1: Training llms to reason and leverage search engines with reinforcement learning.” *arXiv preprint arXiv:2503.09516* (2025).

[12] Yang, An, et al. “Qwen3 technical report.” *arXiv preprint arXiv:2505.09388* (2025).

[13] Agarwal, Sandhini, et al. “gpt-oss-120b & gpt-oss-20b model card.” *arXiv preprint arXiv:2508.10925* (2025).

[14] “Introducing the Model Context Protocol.” *Anthropic*, www.anthropic.com/news/model-context-protocol. Accessed 13 Sep. 2025.

[15] Google. A2a: Agent2agent protocol, 2025. URL https://github.com/google/A2A. Accessed: 2025-10-02.

[16] Yang, Yingxuan, et al. “A survey of ai agent protocols.” *arXiv preprint arXiv:2504.16736* (2025).

About the Authors

Maarten Grootendorst is a senior clinical data scientist at IKNL (Netherlands Comprehensive Cancer Organization). He holds master's degrees in organizational psychology, clinical psychology, and data science, which he leverages to communicate complex machine learning concepts to a wide audience. With his popular blogs, he has reached millions of readers by explaining the fundamentals of artificial intelligence—often from a psychological point of view. He is the author and maintainer of several open source packages that rely on the strength of large language models, such as BERTopic, PolyFuzz, and KeyBERT. His packages are downloaded millions of times and used by data professionals and organizations worldwide.

Jay Alammar is director and engineering fellow at Cohere (pioneering provider of large language models as an API). In this role, he advises and educates enterprises and the developer community on using language models for practical use cases. Through his popular AI/ML blog, Jay has helped millions of researchers and engineers visually understand machine learning tools and concepts from the basic (ending up in the documentation of packages like NumPy and pandas) to the cutting-edge (Transformers, BERT, GPT-3, Stable Diffusion). Jay is also a cocreator of popular machine learning and natural language processing courses on Deeplearning.ai and Udacity.