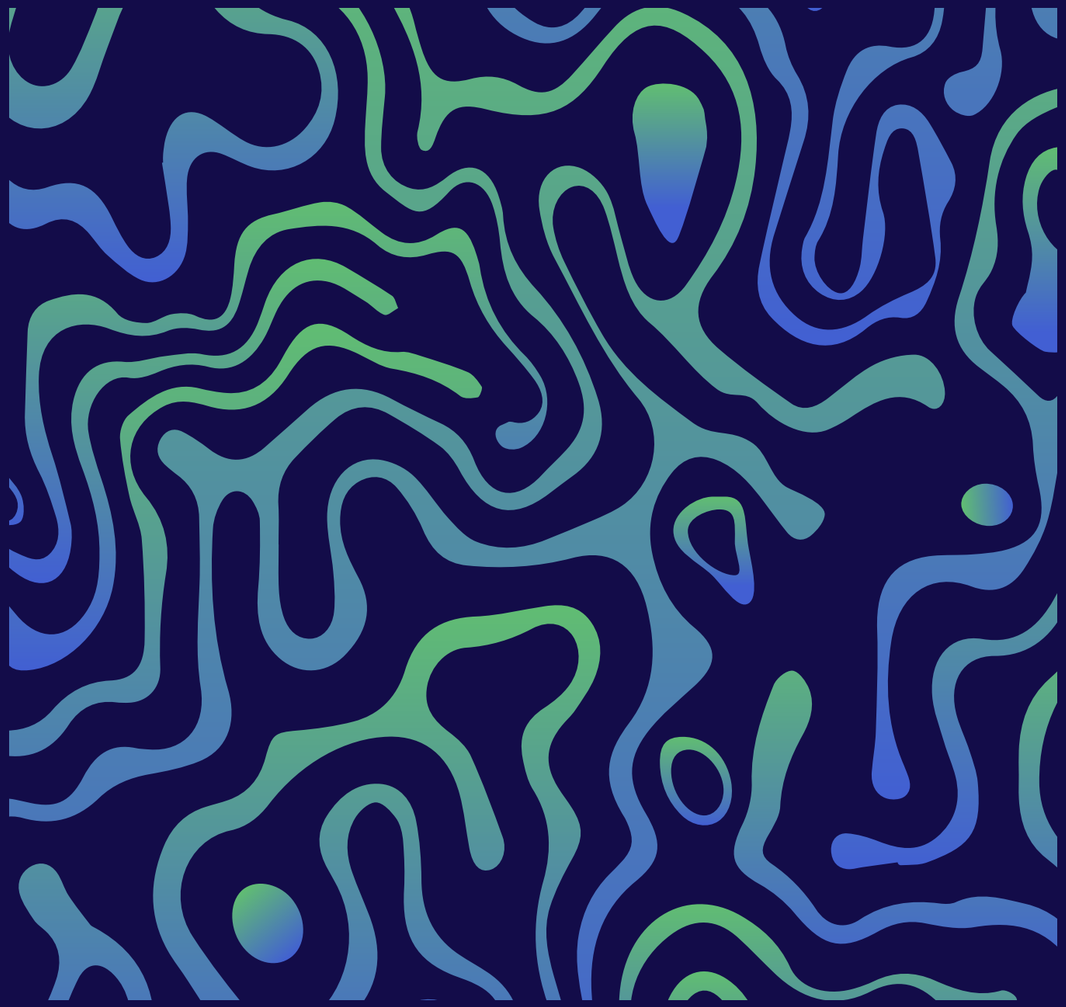
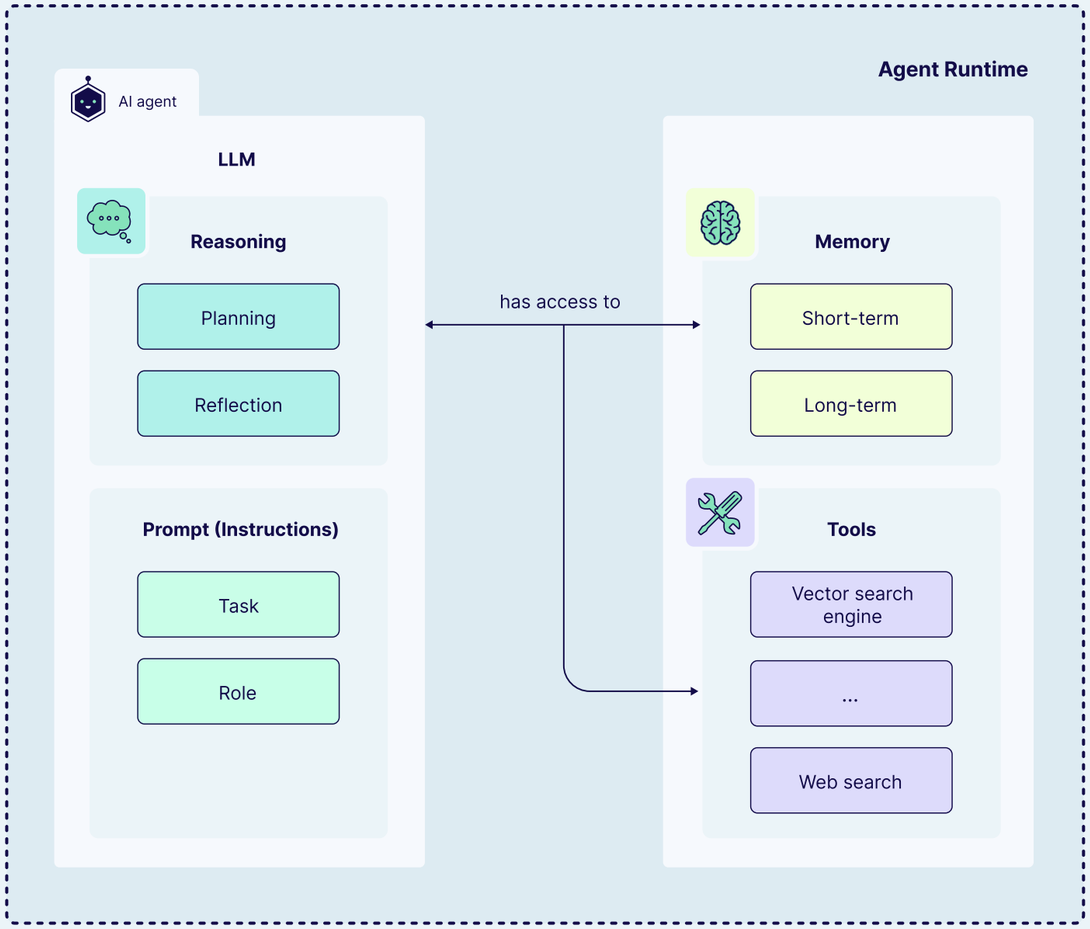


Fundamentals of agentic architectures

Agentic architectures are composed of one or more agents with memory and access to tools. This section discusses the components of AI agents and the atomic patterns in agentic architectures. We will discuss important considerations, such as when to use more than a single agent or how to connect multiple agents in a multi-agent architecture.

Components of AI agents

Although AI agents are designed for autonomous decision-making, they rely on a larger framework of components to function properly. This framework is referred to as its architecture. It consists of a Large Language Model (LLM) with a task and a role, which enables the agent to reason effectively, tools that help the agent complete its tasks, and memory that allows the agent to learn from past experiences.



Reasoning enables AI agents to actively "think" throughout the problem-solving process. In agentic architectures, reasoning serves two key functions: planning, where the agent decomposes complex tasks into smaller steps and selects appropriate tools, and reflecting, where it evaluates outcomes and iteratively adjusts its approach based on results and external data.

Memory enables capturing and storing context and feedback across multiple interactions and sessions. Short-term memory stores more immediate information, like conversation history, which helps the agent determine which steps to take next to complete its overall goal. Long-term memory stores information and knowledge accumulated over time, allowing for personalization of the agent and improved performance over time.

Tools expand the capabilities of AI agents beyond the knowledge of their original dataset and allow them to dynamically interact with external resources and applications, real-time data, or other computational resources. These tools are used to perform specific tasks, like searching the web, retrieving data from an external database, or reading or sending emails that help the agent achieve their target.

Vector databases in agentic architectures

Vector databases can be used for different purposes in agentic architectures.

Tools

Vector databases are most commonly used as tools for agents as part of Retrieval-Augmented generation (RAG) pipelines.

In this case, the tool is a custom search tool that connects to a vector database. The vector database acts as an external knowledge source and stores your own proprietary unstructured data, such as text or images. The agent can call the search tool to conduct a vector, hybrid, or keyword search over the connected vector database.

In an agentic Rag pipeline, an AI agent retrieves information from external knowledge sources and uses it to answer user queries.

Memory

Vector databases can also be used for memory in agentic architectures. Storing information about past interactions in a vector database allows agents to retrieve information from memory semantically.

Single-agent vs. multi-agent architectures

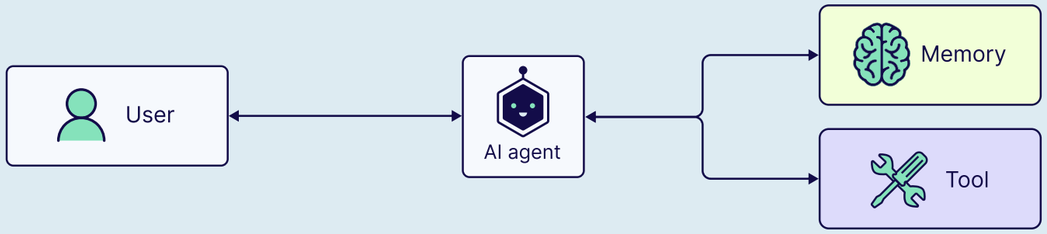
When building an agentic system you can build either a single-agent or a multi-agent architecture. Agentic AI systems use an LLM as the brain of the operation. This LLM has access to tools. At any given time, the LLM evaluates whether a tool is useful to solve (a part of) the query. This is referred to as a 'single-agent architecture'.

However, it also sometimes makes sense to initialize multiple agents, each responsible of solving a certain group of tasks. These are called 'multi-agent architectures'. Often, we may still have one agent (LLM) acting as the lead of the whole operation: the main agent.

The choice depends on your use case and how complex the required agent actions are.

Single-agent architectures

have a single AI agent that independently resolves tasks.



Strengths

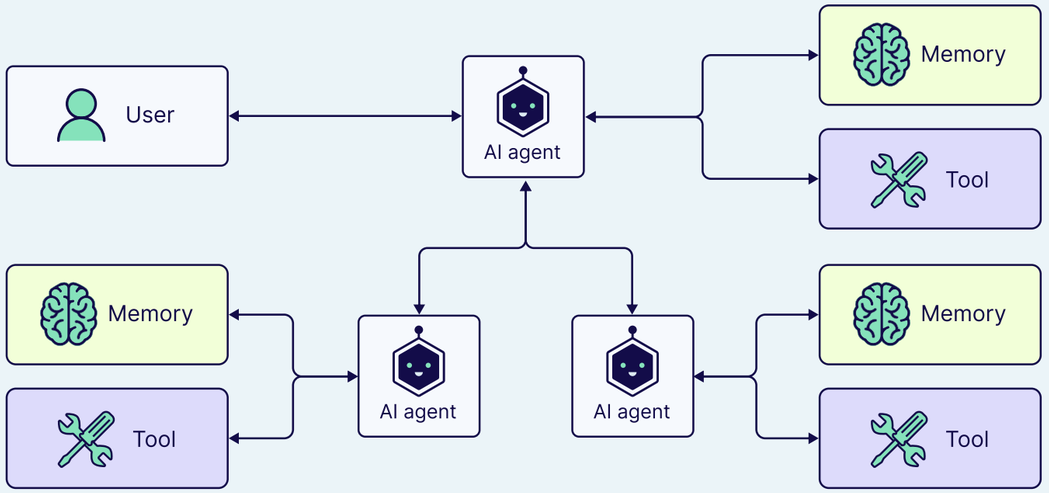
- Low complexity and thus easier to develop and manage.
- No coordination between multiple agents required.
- May require fewer computational resources for a single powerful agent than multiple less powerful agents.

Weaknesses

- May struggle with complex or dynamic tasks.
- Limited in handling tasks that require collaboration or diverse expertise.
- Agent can get confused and use incorrect tool call arguments if the agent has too many different tool options available.
- May require a larger, more expensive model to handle multiple reasoning steps.

Multi-agent architectures

have multiple AI agents collaborating to resolve tasks.



Note that each agent is equipped with its own memory here. However, you can also have memory for the compositional agentic architecture.

Strengths

- Capable of handling complex and dynamic tasks.
- Capable of parallel processing for efficiency.
- Possible to use smaller models specialized for distinct tasks.

Weaknesses

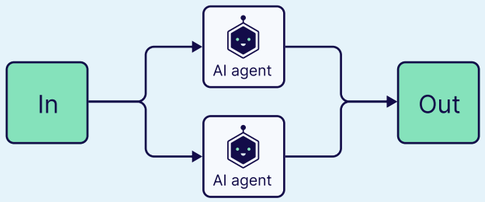
- Increased complexity due to multiple agents collaborating with each other.
- Requires robust mechanisms to manage interactions.
- Harder to debug and optimize due to added complexity.
- May require more resources as more agents are added to the system.

As you can see both, single-agent and multi-agent architectures have both strengths and weaknesses. Single-agent architectures are ideal when the task is straightforward and well-defined and you don’t have specific resource constraints. On the other hand, multi-agent architectures are helpful when the use case is complex and dynamic, requires more specialized knowledge and collaboration, or has scalability and adaptability requirements.

Patterns in multi-agent systems

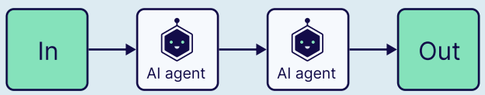
As the name suggests, multi-agent systems consist of multiple agents working together to solve complex tasks. These systems can be structured using various design patterns, each having its own strengths and weaknesses. These patterns are atomic and not mutually exclusive. That means you could design a multi-agent system that, e.g., has routers, loops, and parallel design patterns.

If these patterns sound familiar to you, it's because there's nothing new here. We're borrowing the design patterns of connecting components from other domains, such as software engineering. Thus, this section is only intended as a refresher of possible patterns.



Parallel

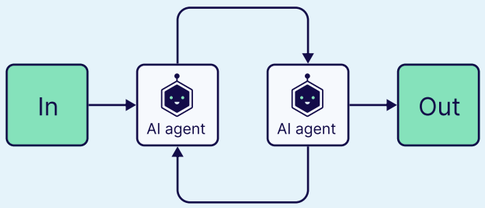
Multiple agents work simultaneously on different parts of a task.



Sequential

Tasks are processed sequentially, where one agent's output becomes the input for the next.

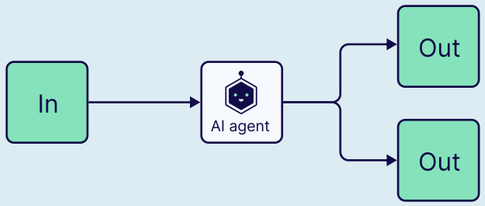
Example: Multi-step approvals.



Loop

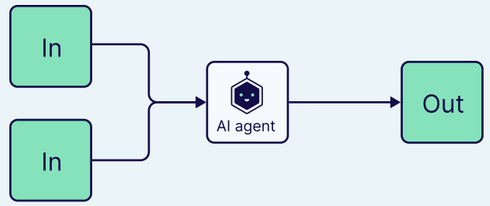
Agents operate in iterative cycles, continuously improving their outputs based on feedback from other agent(s).

Example: Evaluation use cases, such as code writing and code testing.



Router

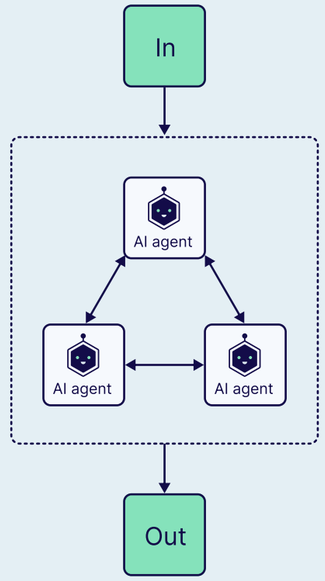
A central router determines which agent(s) to invoke based on the task or input.



Aggregator

or synthesizer

Agents contribute outputs that are collected and synthesized by an aggregator agent into a final result.



Network

or horizontal

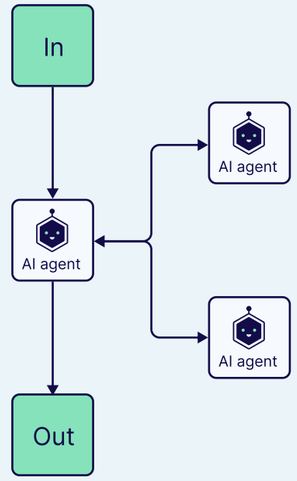
Agents communicate directly with one another in a many-to-many fashion, forming a decentralized network.

Pros:

- Distributed collaboration and group-driven decision-making.
- The system remains functional even if some agents fail.

Cons:

- Managing communication among agents can become challenging.
- More communication may cause inefficiencies and the possibility of agents duplicating efforts.



Hierarchical

or vertical

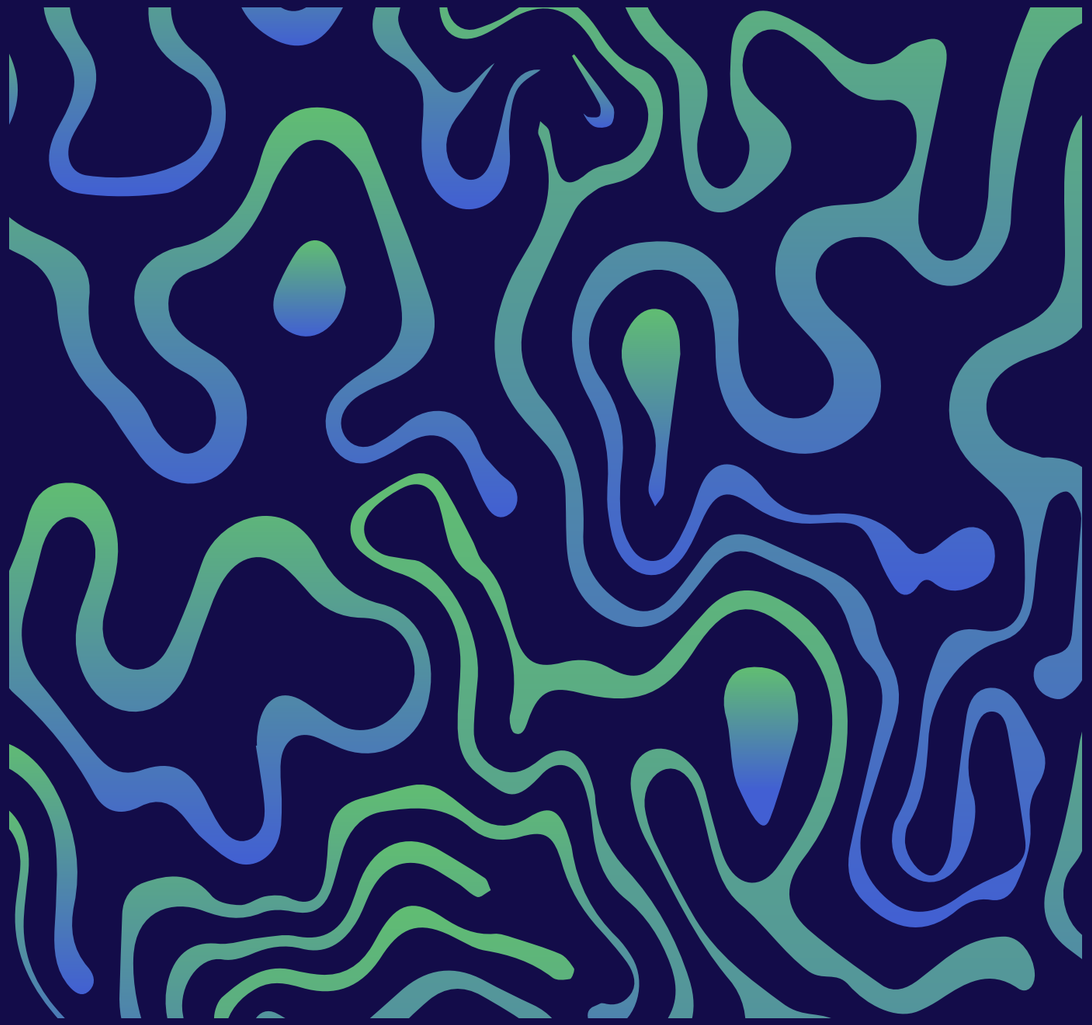
Agents are organized in a tree-like structure, with higher-level agents (supervisor agents) managing lower-level ones.

Pros:

- Clear division of roles and responsibilities among agents at different levels.
- Streamlined communication
- Suitable for large systems with a structured decision flow.

Cons:

- Failure at upper levels can disrupt the entire system.
- Lower-level agents have limited independence.



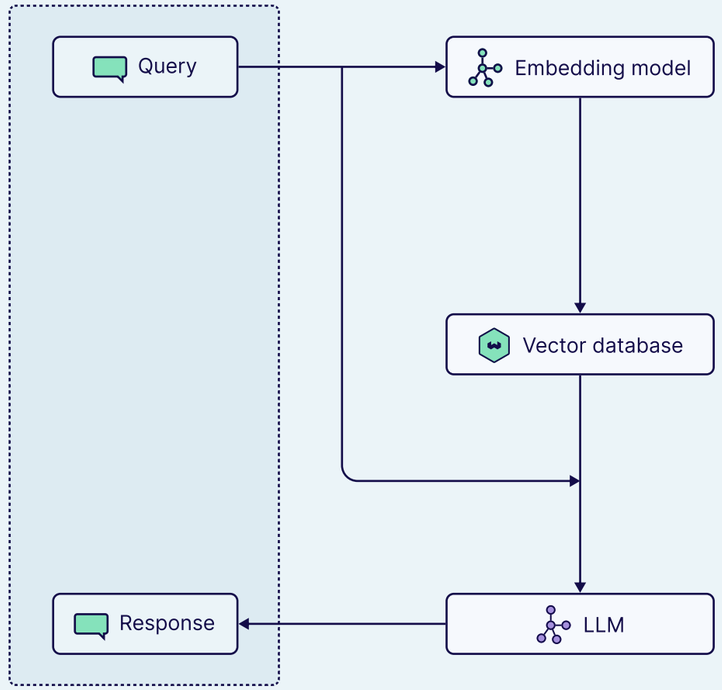
Examples of agentic architectures

This section discusses a few examples of architectures for agentic RAG pipelines. The overall architecture depends on your use cases' requirements. For less complex use cases, a simple single-agent router architecture may be sufficient, while for more complex use cases, a multi-agent architecture with specialized agents may be necessary.

Revisiting naive RAG architecture

Before diving into the variety of different agentic architectures for retrieval-intensive use cases, let's revisit the naive RAG architecture to remind us of its limitations.

The naive (non-agentic) RAG architecture usually consists of an embedding model, a vector database, and a generative LLM. This non-agentic naive approach is a one-shot solution that uses the user query directly to retrieve additional information and then uses the retrieved information directly in the prompt.

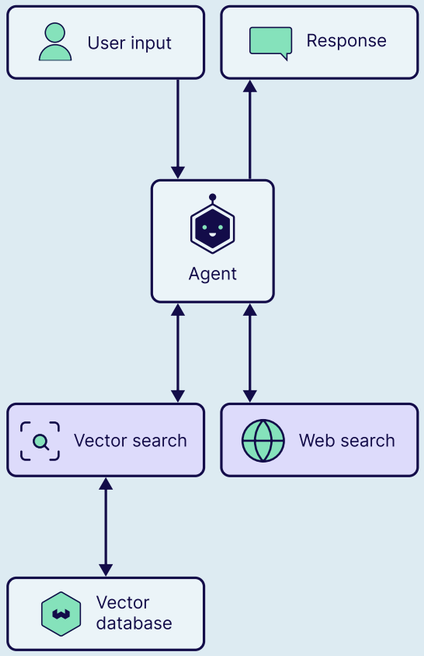


While the beauty of the naive approach lies in its simplicity, it leaves a lot of room for errors:

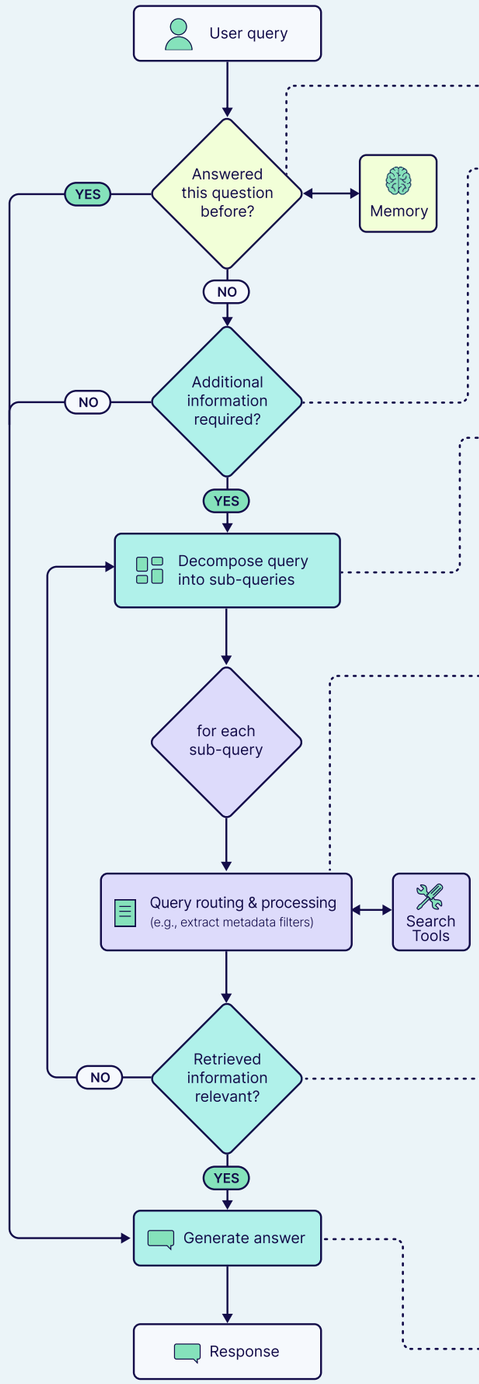
- The raw user query without any further processing might not be suitable for vector search as sometimes rewording or usage of metadata filters can be useful for better retrieval performance. Also, some user queries are complex and may require decomposition into smaller queries for improved processing.
- There is no validation step to determine whether the retrieved information is relevant to the user query.
- Information is only retrieved once.

Single-agent architecture

In a single-agent RAG architecture, a multipurpose agent is responsible for retrieving the required information and generating the response based on that information.



On the left, you can see an example agentic workflow of how this multipurpose agent can retrieve additional information to generate a more factual, accurate answer.



1. First, the agent accesses memory to check if the query has been answered before, and decides if the question can be answered directly from memory.

2. If the question hasn't been answered before, the agent can reason and evaluate if the question requires any additional information to answer the user query.

3. If the agent decides it needs additional information, it can analyze the user query. If the user query is complex, the agent can decompose the it into simpler sub-queries.

   Query decomposition
   Query decomposition is a technique that breaks down complex queries into simpler sub-queries. This is useful for answering multifaceted questions requiring information from different knowledge sources. Thus, decomposing complex user queries into sub-queries can lead to more precise and relevant search results.

4. The agent can then route the user query (or the sub-queries) to the most suitable knowledge source (query routing) and even determine additional function arguments (e.g., extract metadata filters) before calling the search function.

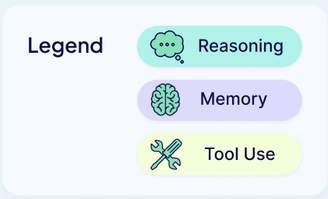
   Query routing
   If you have at least two external knowledge sources, the agent can decide which one to retrieve additional context from. Note, that the external knowledge sources don't have to be limited to (vector) databases. You can retrieve further information from other tools, such as web search tools, as well.

   Query Transformation
   Once a (sub)query is routed to a knowledge source, the agent can perform additional transformations to format the search query into the right shape for the target search tool to achieve optimal results. For example if you are retrieving information from a vector database, the agent can determine whether to conduct a vector, hybrid, or keyword search. Additionally, the agent can extract metadata filters from the user query or even decide if the retrieved results need to be grouped or otherwise aggregated.

5. After retrieval, the agent can evaluate the retrieved information.

   Evaluation
   The agent can evaluate the new information within the context of the original user query. For example, the agent cannot only evaluate if the retrieved information is relevant to answer the user query but it can also evaluate if it thinks there is missing information. If there is missing or irrelevant information, the agent can revisit the query processing strategy and decide if it should try a different knowledge source or if it needs to adjust the query transformations.

6. Finally, the agent can generate an answer and respond to the user query.



As you can see, the majority of the agentic workflow in a multipurpose RAG agent revolves around improving retrieval. Therefore, in some use cases, it might make sense to scope out specialized agents with dedicated tasks and roles at which they can accelerate.

For example, you could scope out an agent specialized for retrieving information from external knowledge sources (query agent) which is specialized for query decomposition, query routing, query transformation, and evaluation.

Multi-agent architectures

Although the single-agent architecture overcomes the static limitations of a naive RAG pipeline, it is limited to only one agent with reasoning, retrieval, and answer generation in one agent.

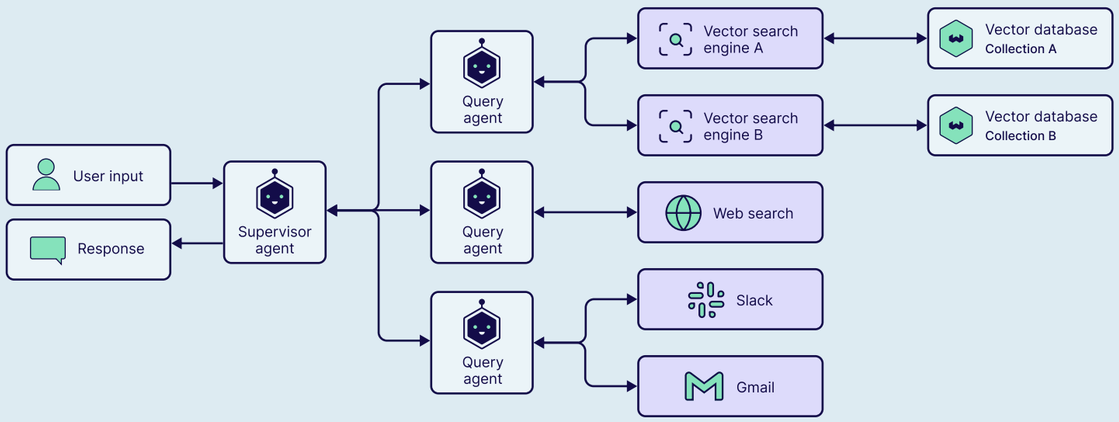
Having one multipurpose agent can lead to lower performance if the agent's task is too broad and not well-defined. Therefore, we discussed on the previous page that scoping out agents with smaller, more well-defined tasks can help them excel at their specific role, such as splitting the multipurpose agent into a query agent specialized for retrieval and an agent that generates the answer from the retrieved information.

If you have a more complex use case, it can be beneficial to chain multiple agents into a multi-agent architecture. For example, if you have a use case that requires multiple tools or specialized sub-tasks with specialized agents.

Now this is where things get interesting, as there are endless possibilities of chaining agents together in multi-agent architectures. This section by no means aims to be an exhaustive list. Instead, this section shows examples of architectures and their considerations so you can start building your own architectures suited for your specific use case.

Hierarchical example

Let’s begin with a simple hierarchical architecture, where you have one supervisor agent orchestrating multiple specialized agents.



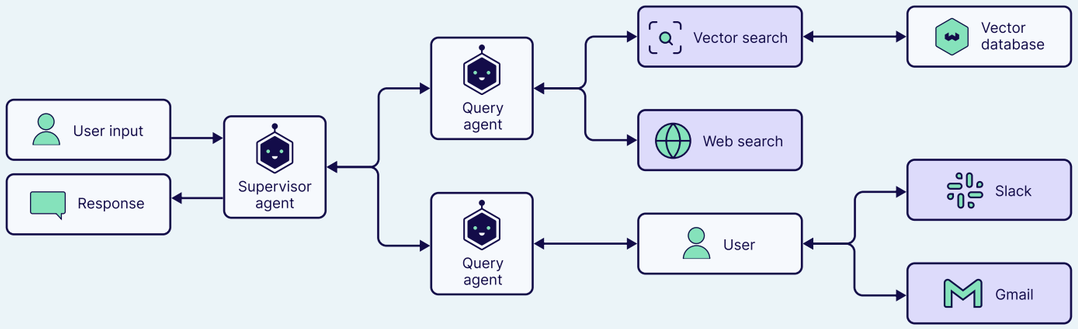
For example, you can have one supervisor agent who coordinates information retrieval among multiple specialized agents for querying information from external knowledge sources (query agents):

One agent could retrieve information from proprietary internal data sources, another agent could also specialize in retrieving public information from web searches, and a third agent could specialize in retrieving information from your personal accounts, such as email or chat.

By assigning each query agent by a specialization, you can increase the chances that each agent excells at its task and provide it only with the tools it requires. Additionally, you can limit the access to certain tools with sensitive data to ensure data security, for example when using API calls to personal accounts, such as email, calendars or chat messages.

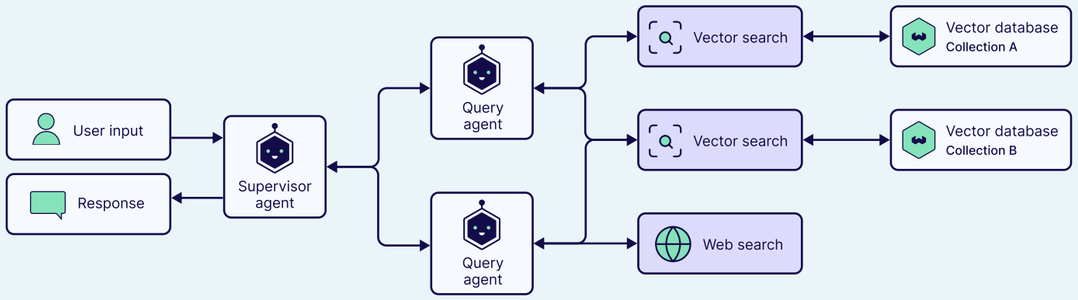
Human-in-the-loop example

If you don’t want to give an agent access to sensitive information, such as an employee’s personal emails, calendar, or chat messages, you can build a human-in-the-loop architecture. In this case, you can use a specialized agent that asks for human input before moving on to the next action.



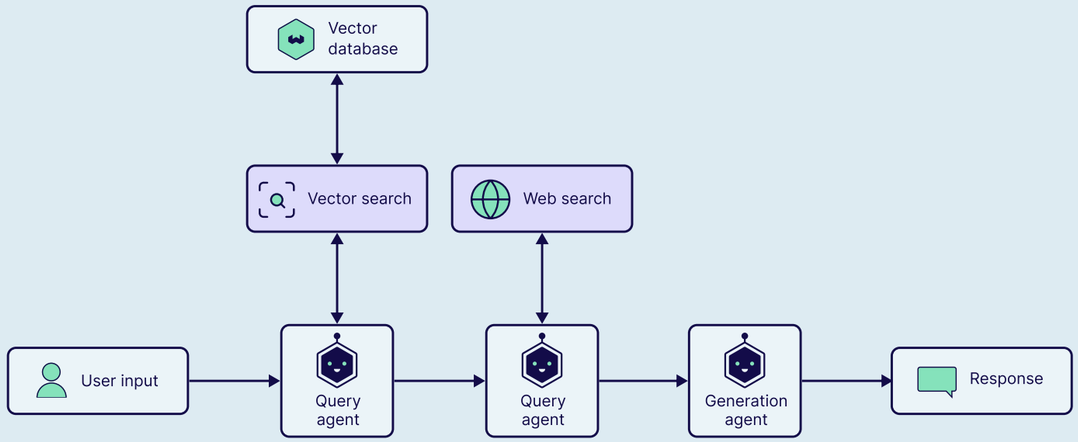
Shared tools example

On the other hand, depending on your use case, it can also be helpful to have different agents with access to the same tool. For example, if your agents need to have access to central user information, then it might be necessary to provide them with a search tool over a central database.



Sequential example

So far, we have looked at hierarchical multi-agent architectures only but if there is no need for a supervisor agent, you can also choose a network (or horizontal) architecture pattern.



Above you can see an example of a horizontal architecture pattern with three sequential agents:

1. The first query agent retrieves information based on the user input by calling a vector search tool.
2. Then, the second query agent retrieves additional information based on the user input and the information retrieved by the first query agent by calling a web search tool.
3. Finally, the last generation agent generates a response based on the user input, the information from the vector search, and the information from the web search.

Having two separate query agents connected sequentially can be useful when the two agents use different tools and the second agent acts based on the input of the first agent.

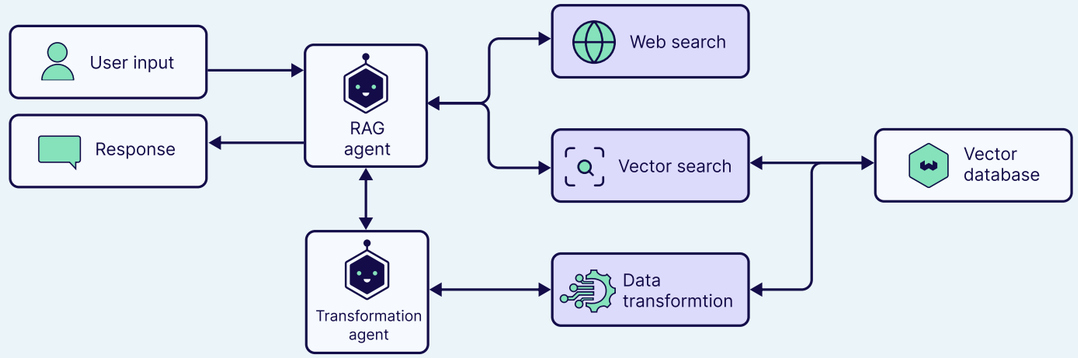
Shared database with different tools example

Data inside of databases is not always clean, organised or well separated. Historically these types of issue would be solved by having a database administrator or teams dedicated to quality and enrichment having to support massive and complex pipelines to alter or create new views on data.

But what if you could replace this work by an AI agent that transforms user data at insert-time and/or transforms existing collection data at scale? This agent's task is to transform existing data to enrich user data, provide complex data analysis, and transform unsearchable data into searchable formats.

For example, let's say you have customer reviews and want to add some attributes. You could go ahead and have the agent generate a new property of attributes so we can later better such across those. Or you could generate review summaries for a product from all the existing summaries.

To accommodate such a data transformation, you can incorporate a data transformation agent as shown below. This agent, instead of having a vector search tool, it would have a data transformation tool to access the same database as the agent querying the database.



Memory transformation through tool use

Since past interactions can be stored in a vector database acting as memory, a data transformation agent can also be used on memory. This can be useful if you want to e.g., summarize past interactions. You could summarize the last five interactions or summarize what has been discussed about a certain topic.