LLMs enable the development of intelligent agents capable of tackling complex, non-repetitive tasks that defy description as deterministic workflows. By spli_ing reasoning into steps in different ways and orchestrating them in a relatively simple way, agents can demonstrate a significantly higher task completion rate on complex open tasks.

This agent-based approach can be applied across numerous domains, including RAG systems, which we discussed in *Chapter 4*. As a reminder, what exactly is *agentic RAG*? Remember, a classic pa_ern for a RAG system is to retrieve chunks given the query, combine them into the context, and ask an LLM to generate an answer given a system prompt, combined context, and the question.

We can improve each of these steps using the principles discussed above (decomposition, tool calling, and adaptation):

Dynamic retrieval hands over the retrieval query generation to the LLM. It can decide itself whether to use sparse embeddings, hybrid methods, keyword search, or web search. You can wrap retrievals as tools and orchestrate them as a LangGraph graph.

Query expansion tasks an LLM to generate multiple queries based on initial ones, and then you combine search outputs based on reciprocal fusion or another technique.

Decomposition of reasoning on retrieved chunks allows you to ask an LLM to evaluate each individual chunk given the question (and filter it out if it's irrelevant) to compensate for retrieval inaccuracies. Or you can ask an LLM to summarize each chunk by keeping only information given for the input question. Anyway, instead of throwing a huge piece of context in front of an LLM, you perform many smaller reasoning steps in parallel first.This can not only improve the RAG quality by itself but also increase the amount of initially retrieved chunks (by decreasing the relevance threshold) or expand each individual chunk with its neighbors.

In other words, you can overcome some retrieval challenges with LLM reasoning. It might increase the overall performance of your application,

but of course, it comes with latency and potential cost implications.
Reflection steps and iterations task LLMs to dynamically iterate on retrieval and query expansion by evaluating the outputs after each iteration. You can also use additional grounding and a_ribution tools as a separate step in your workflow and, based on that, reason whether you need to continue working on the answer or the answer can be returned to the user.

Based on our definition from the previous chapters, RAG becomes agentic RAG when you have shared partial control with the LLM over the execution flow. For example, if the LLM decides how to retrieve, reflects on retrieved chunks, and adapts based on the first version of the answer, it becomes agentic RAG. From our perspective, at this point, it starts making sense to migrate to LangGraph since it's designed specifically for building such applications, but of course, you can stay with LangChain or any other framework you prefer (compare how we implemented map-reduce video summarization with LangChain and LangGraph separately in *Chapter 3*).

Multi-agent architectures

In *Chapter 5*, we learned that decomposing a complex task into simpler subtasks typically increases LLM performance. We built a plan-and-solve agent that goes a step further than CoT and encourages the LLM to generate a plan and follow it. To a certain extent, this architecture was a multi-agent one since the research agent (which was responsible for generating and following the plan) invoked another agent that focused on a different type of task – solving very specific tasks with provided tools. Multi-agentic workflows orchestrate multiple agents, allowing them to enhance each other and at the same time keep agents modular (which makes it easier to test and reuse them).

We will look into a few core agentic architectures in the remainder of this chapter, and introduce some important LangGraph interfaces (such as streaming details and handoffs) that are useful to develop agents. If you're interested, you can find more examples and tutorials on the LangChain

documentation page at https://langchainai.
github.io/langgraph/tutorials/#agent-architectures. We'll begin
with discussing the importance of specialization in multi-agentic systems,
including what the consensus mechanism is and the different consensus
mechanisms.

Agent roles and specialization

When working on a complex task, we as humans know that usually, it's
beneficial to have a team with diverse skills and backgrounds. There is much
evidence from research and experiments that suggests this is also true for
generative AI agents. In fact, developing specialized agents offers several
advantages for complex AI systems.

First, specialization improves performance on specific tasks. This allows you
to:

Select the optimal set of tools for each task type.

Craft tailored prompts and workflows.

Fine-tune hyperparameters such as temperature for specific contexts.

Second, specialized agents help manage complexity. Current LLMs struggle
when handling too many tools at once. As a best practice, limit each agent to
5-15 different tools, rather than overloading a single agent with all available
tools. How to group tools is still an open question; typically, grouping them
into toolkits to create coherent specialized agents helps.