BUILD AN AI Agent

Agents that reason, plan, and act autonomously

Jungjun Hur
Younghee Song

MANNING

MEAP Edition
Manning Early Access Program

Build an AI Agent (From Scratch)

Agents that reason, plan, and act autonomously

Version 3

welcome

Thank you for purchasing the MEAP edition of *Build an AI Agent (From Scratch)*.

To get the most out of this book, you don’t need to be an expert in AI or machine learning. If you are comfortable with the basics of Python—things like writing simple functions and classes—know how to call and use APIs, and have some familiarity with Git and GitHub, you’ll be more than ready to follow along. That’s all the background you need.

When I first started exploring the world of AI agents, I found myself overwhelmed. Every week, it seemed like a new agent framework, research paper, or tool was being released. Even as an engineer, I struggled to make sense of the rapid changes. I realized that if I only stayed a user of these frameworks, I would never truly understand how they worked or how to evaluate which ones to trust. That’s when I decided to dig deeper—and ultimately, that path led me to writing this book.

At the heart of this book is a hands-on project: together, we’ll build a simple but educational agent framework called **scratch_agents** and use it to create agents that solve problems step by step. As we build this framework, you’ll also discover that the essence of an agent lies in delivering the right context to an LLM so that its intelligence can be fully utilized. We call this process *Context Engineering*. Agent frameworks are essentially designed to manage this challenge effectively, and through this book you’ll learn how to design, implement, and experiment with this idea in practice.

This book is not about giving you yet another tool to memorize. It’s about empowering you to understand the inner workings of agent frameworks so you can go beyond surface-level usage. By the end of this journey, you won’t just be using existing frameworks—you’ll know how to open them up, understand their trade-offs, and even build your own when needed. That’s the kind of confidence and clarity this book aims to give you.

I hope this book helps you cut through the noise in this fast-moving field and gives you both confidence and clarity in working with AI agents. Please share your thoughts and questions in the liveBook discussion forum—your feedback will be invaluable in making this book the best it can be.

—Younghee Song & Jungjun Hur

Brief contents

1 What is an AI agent?

This chapter covers

- The landscape of AI agents today
- LLMs as the decision-making core of agents
- Workflows vs agents and when to use each
- GAIA benchmark for measuring agent performance
- Context engineering for building effective agents

You may have heard of agent-building frameworks like LangGraph, CrewAI, AutoGen, or OpenAI Agents. These frameworks make it easy to build agents quickly, but they also hide what's actually happening inside. This book takes a different approach: we'll build agents from scratch, understanding every component before relying on any framework.

Why build from scratch? Because agent development is fundamentally about debugging failures. When your agent gives a wrong answer or gets stuck in a loop, you need to understand exactly what went wrong. Did the LLM (Large Language Model) misinterpret the context? Did a tool return unexpected results? Was crucial information missing? Without understanding how agents work internally, diagnosing these problems is difficult, regardless of which tools you use. By building each component yourself, you'll develop the mental model needed to troubleshoot any agent system, whether you built it or inherited it.

This chapter establishes the foundation for everything that follows. We'll start by surveying the landscape of AI agents, from personal assistants to specialized coding tools, to understand what we're building toward. Then we'll examine how LLMs serve as the "brain" of an agent and what distinguishes a true agent from a simple workflow. We'll introduce GAIA, the benchmark we'll use throughout this book to measure our progress, and explore context engineering, the discipline that determines whether an agent succeeds or fails. By the end, you'll have a clear mental model of what agents are, when to use them, and the principles that will guide our implementation journey.

1.1 The age of AI agents

AI agents are rapidly transforming how we interact with technology. From personal assistants that help with everyday tasks to sophisticated systems that handle complex professional work, agents are becoming an integral part of both consumer products and enterprise solutions. Before diving into how to build them, let's survey the landscape of AI agents to understand what we're working toward.

Personal AI Agents are the most familiar form of AI agents today. Services like ChatGPT, Claude, and Gemini started as conversational chatbots where you asked a question, and they provided an answer. But these systems have evolved dramatically. Modern personal agents can analyze uploaded documents, search the web for current information, generate images, write and execute code, and even help with shopping decisions. They serve as general-purpose assistants that adapt to whatever task you bring to them, learning your preferences and communication style over time.

Customer-Facing Agents operate on behalf of businesses, interacting directly with customers in real time. These agents go beyond simple FAQ chatbots. They can access company policies, retrieve customer data, process transactions, and make decisions based on business rules. When you interact with a support agent that checks your order status, processes a refund, or helps troubleshoot a product issue, you're increasingly likely to be communicating with an AI agent. These systems must balance helpfulness with strict adherence to company guidelines and regulatory requirements.

Specialized Agents tackle domain-specific tasks that require deep expertise or extended processing time. Coding agents like Claude Code, Cursor, and Codex CLI can navigate codebases, implement features, fix bugs, and refactor code. These are tasks that previously required hours of developer time. Deep Research agents can conduct comprehensive investigations across hundreds of sources, synthesizing findings into detailed reports. Because these tasks often take significant time to complete, specialized agents frequently operate asynchronously, notifying users when results are ready. When connected to proprietary enterprise data, these agents become vertical solutions tailored to specific industries or use cases.

Despite their different applications and interfaces, all these agents share a common foundation: they are powered by Large Language Models. Understanding how LLMs work and how they enable agent capabilities is essential for anyone who wants to build effective agents. In this book, we use "AI agent" and "LLM agent" interchangeably, since virtually all modern AI agents are powered by LLMs.

1.2 Understanding LLM agents

All the agents we explored share something remarkable: they rely on LLMs as their decision-making core. But how does a system designed to predict the next word become capable of completing multi-step tasks autonomously? Let’s first unpack what makes LLMs uniquely suited to power agents, then examine the three essential components that transform an LLM into an agent. This will help you understand the agent loop that forms the foundation for everything we'll build in this book.

1.2.1 What is an LLM?

An LLM (Large Language Model) is a language model trained on nearly all publicly available text from the internet. While modern LLMs have evolved into multimodal models that process images, audio, and video, this book uses the term "LLM" to refer to all these variants, since they share the same foundational architecture.

The core principle of LLMs is simple: learn to predict the next word by processing massive amounts of text data. Given the sentence "Because it's summer vacation, I didn't go to _______," most would expect the blank to be filled with "school." Through large-scale training on this simple principle, LLMs develop an understanding of language structure, context, and meaning. LLMs process text by breaking it into tokens, which are typically words or word fragments. The model predicts the next token based on all preceding tokens.

Earlier approaches, like rule-based systems and reinforcement learning, also attempted to create intelligent agents. What sets LLMs apart is their generalization capability—the ability to handle diverse tasks without task-specific retraining. Figure 1.1 demonstrates this. First, LLM translates "banana" using an instruction and two examples (apple, orange)—this is a few-shot learning. Meanwhile, with zero-shot learning, the LLM succeeds with just the instruction, without any examples.

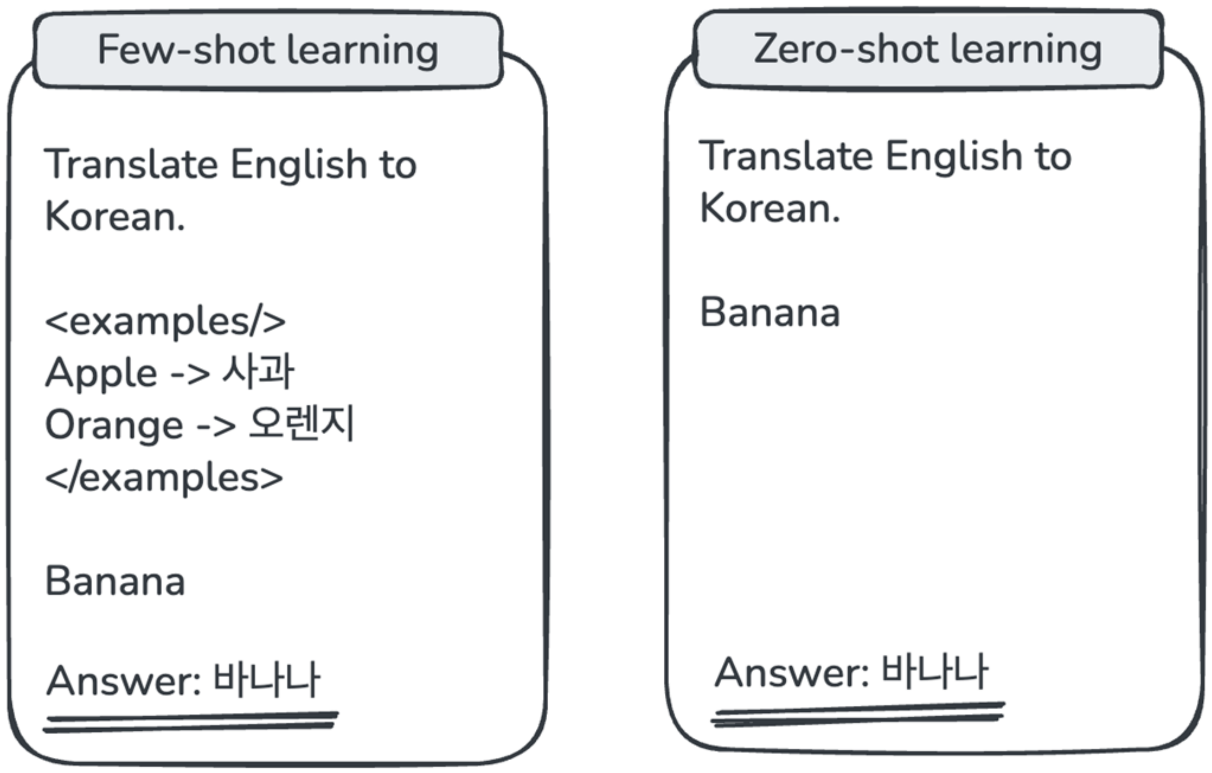


Figure 1.1 Example of a language model's generalization capability.

Recently, "reasoning models" have emerged that analyze tasks or formulate plans before producing results. The reasoning model doesn't answer immediately but first generates thinking tokens starting with <think>. These advances enable LLM agents to tackle complex and unfamiliar problems effectively.

1.2.2 What is an LLM Agent?

A representative example of an LLM agent is the research agent. When a user asks, "Summarize the 2024 Nobel Physics winners' research," the agent gathers information from multiple sources, analyzes findings, and produces a comprehensive report.

As shown in figure 1.2, the research agent doesn't simply generate an answer from its training data. It searches for

2024 Nobel Prize information, identifies the winners (Geoffrey Hinton and John Hopfield), explores academic papers, and consults Wikipedia. It then synthesizes all this information into a coherent report detailing the winners' achievements and their impact on AI.

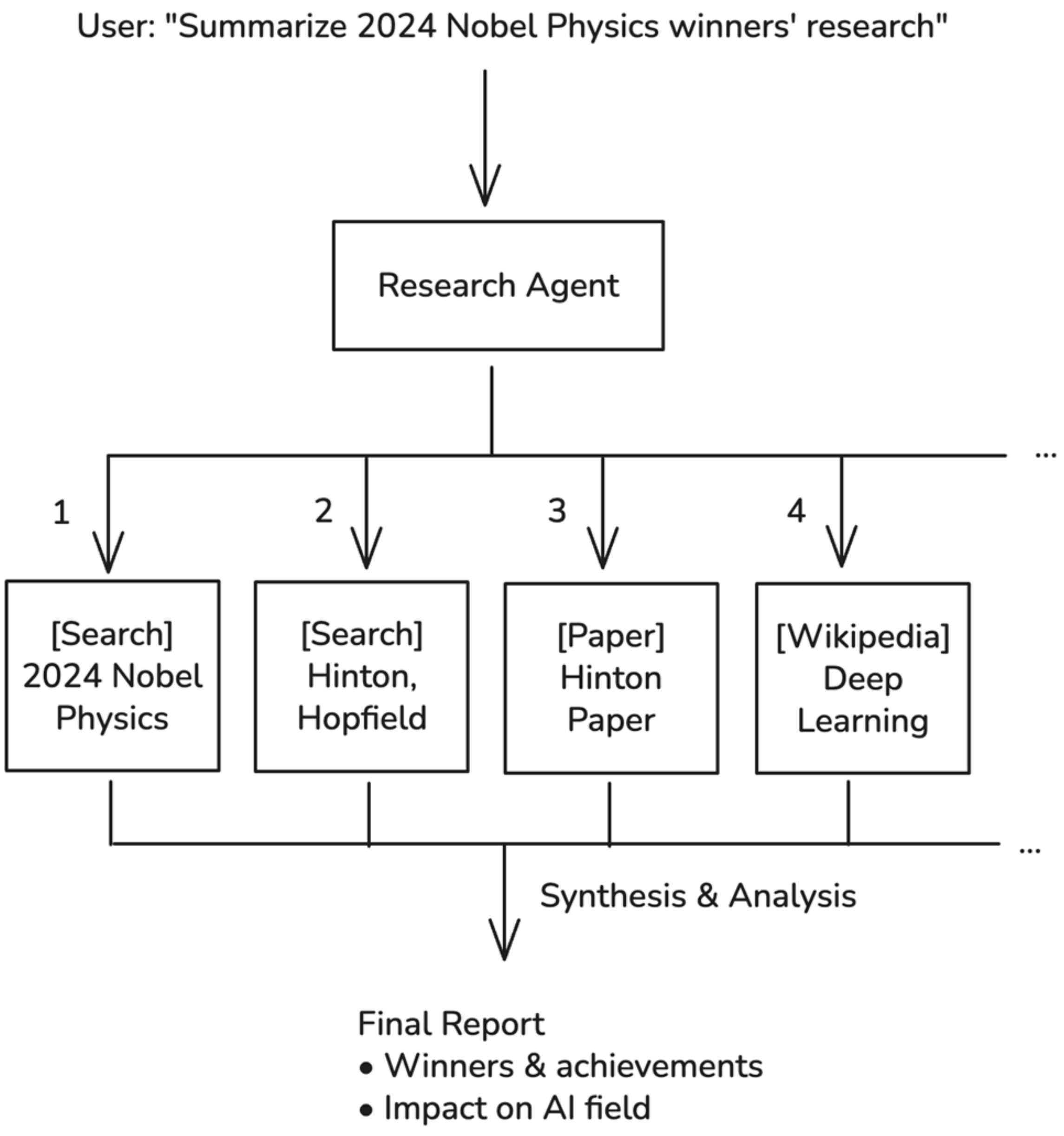

User: "Summarize 2024 Nobel Physics winners' research"
Research Agent
1
2
3
4
...
[Search] 2024 Nobel Physics
[Search] Hinton, Hopfield
[Paper] Hinton Paper
[Wikipedia] Deep Learning
...
Synthesis & Analysis
Final Report
• Winners & achievements
• Impact on AI field

Figure 1.2 User requests flow through the research agent, which branches into multiple searches and synthesis.

CORE DEFINITION: LLM + TOOL + LOOP

This example reveals the essence of an agent. An LLM agent is a program that autonomously decides what actions to take and when to stop based on its current context and goals. At its core, it consists of three elements.

The LLM serves as the agent's brain. It understands the current situation and decides what to do next. Tools are the means of interacting with the external world—web search, code execution, and database access. The Loop is the structure that repeats this process until the goal is achieved.

The LLM being the "brain" means it doesn't just generate text—it decides which tool to use and when to stop. This is what distinguishes an LLM agent from a plain LLM or traditional software.

HOW THE AGENT LOOP WORKS

The Loop is necessary because it's difficult to know in advance which tools will be needed or how many steps will be required to complete a task. Figure 1.3 shows the research task about the 2024 Nobel Prize winners from figure 1.2, viewed through the lens of the Agent Loop.

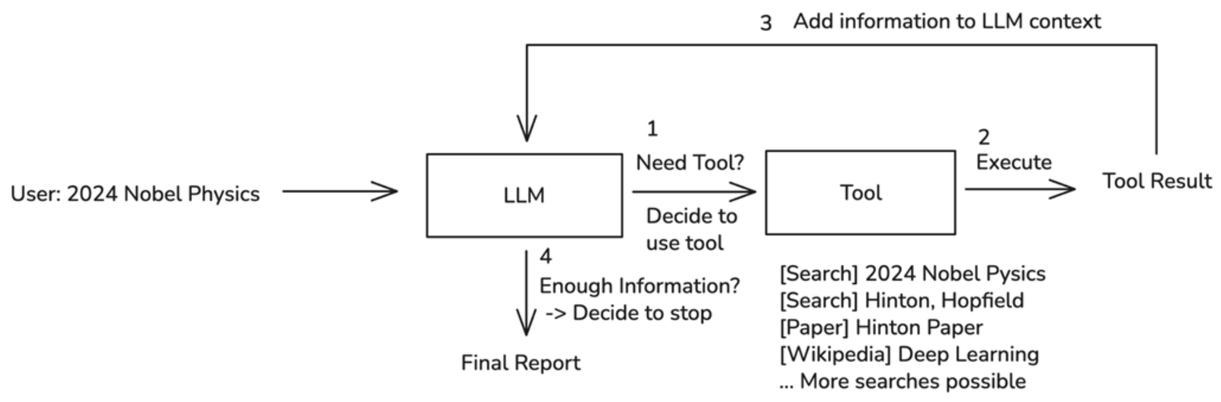


Figure 1.3 The LLM Agent's decision loop is an iterative process of LLM decision-making and tool use.

The process follows a continuous loop:

1. **The LLM evaluates the current context** and determines whether a tool is needed. If more information is required, it decides which tool to use based on what's missing.
2. **The selected tool is executed**—whether it's searching for "2024 Nobel Physics," looking up "Hinton, Hopfield," finding academic papers, or accessing Wikipedia for background knowledge.
3. **The tool's results are added back to the LLM's context**, enriching its understanding of the topic. This accumulated context allows the LLM to make more informed decisions in subsequent iterations.
4. **The LLM decides whether to continue or stop**. If it determines that sufficient information has been gathered to answer the user's question comprehensively, it generates the final report. Otherwise, it returns to step 1 for another iteration.

This iterative process of reasoning, acting, and observing enables the agent to handle tasks of unpredictable complexity from simple queries requiring a single search to complex research requiring dozens of information sources.

HOW LLM AGENTS DIFFER FROM LLMS AND TRADITIONAL SOFTWARE

Traditional software requires developers to define all actions in advance. Execution paths are fixed, and external interactions are implemented directly in code. An LLM can generate text, but on its own, it cannot interact with the external world or perform multi-step tasks.

An LLM agent combines and transcends both. The LLM dynamically decides actions based on context, interacts with the external world through tools, and repeats this process until the goal is achieved. Whether a research task ends with one web search or requires multiple academic papers and Wikipedia articles is unknown beforehand. The agent determines its own path based on the query's complexity and the information discovered along the way.

1.3 Workflow vs agent

When integrating LLMs into applications, developers face a fundamental architectural choice: how much control should the LLM have over the execution flow? This decision shapes everything from system reliability to operational costs. Figure 1.4 illustrates seven distinct levels of agency, progressing from left to right with increasing autonomy.

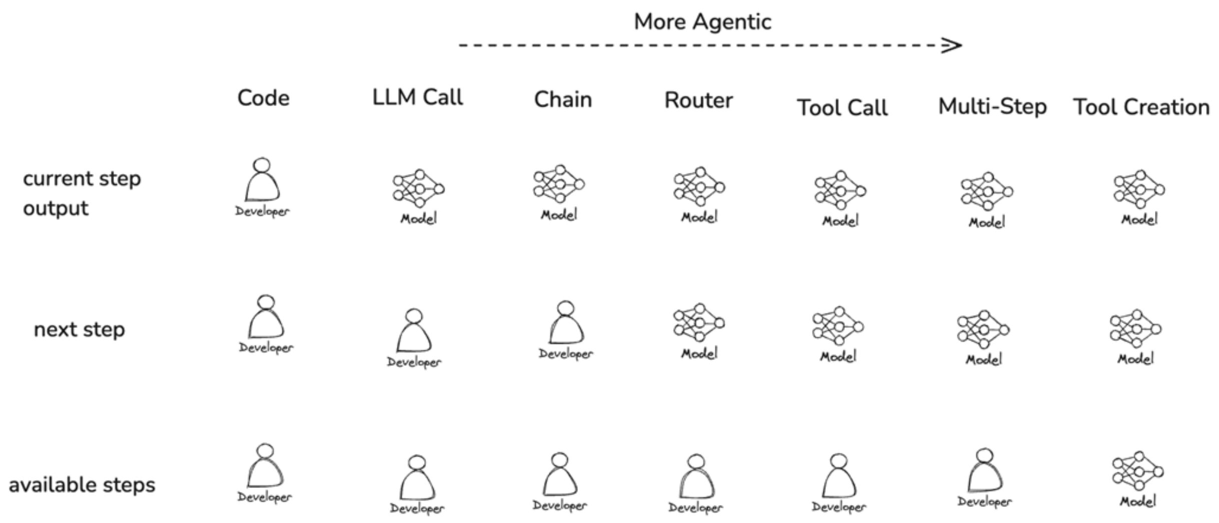


Figure 1.4 Progression of agency levels in LLM applications.

The three rows in the figure represent different aspects of control: who produces the current output (current step), who decides what happens next (next step), and who defines the available options (available steps). As we move rightward, the LLM gains more control over each of these dimensions.

These seven levels can be broadly categorized into two architectural approaches: **workflows** and **agents**. In workflows, developers predefine the execution flow and use LLMs to perform specific steps within that structure. In agents, LLMs dynamically determine their own processes, deciding which actions to take and when to stop. Understanding this distinction is essential for choosing the right approach for your use case.

1.3.1 Workflow: Developer-defined flow

A workflow is a system where developers explicitly design the sequence of operations, with LLMs executing specific steps within that predefined structure. The key characteristic is predictability: given the same input, the system follows the same path through the workflow.

LLM CALL

The most basic pattern is a single LLM call, where one prompt goes in, and one response comes out. This delegates a specific task to the model while keeping everything else under developer control. Examples include text classification, summarization, or answering a straightforward question. Despite its simplicity, a well-crafted single call can handle surprisingly complex tasks.

CHAIN

A chain connects multiple LLM calls in a predefined sequence, where the output of one step becomes the input for the next. The developer designs this flow in advance, and the LLM executes each step sequentially.

This pattern leverages a fundamental principle: LLMs perform best with focused, well-defined tasks. Rather than asking a model to "analyze this document and create a presentation," a chain might break this into discrete steps: extract key points, organize by theme, generate slide content, and refine for clarity. Each step is simpler and more likely to succeed than attempting everything at once.

ROUTER

A router introduces conditional logic where the LLM decides which predefined path to take next. Given a user query, the model might classify it and route it to the appropriate handler. Billing questions go to one workflow, technical support to another.

While the LLM makes a decision here, it's choosing from a fixed set of options that the developer has explicitly defined. The available paths, and what happens along each path, remain under developer control. This makes routers a workflow pattern rather than an agent pattern. The LLM influences the route but doesn't control the journey.

1.3.2 Agent: LLM-directed flow

An agent is a system where the LLM dynamically directs its own processes and tool usage, maintaining control over how it accomplishes tasks. Rather than following a predetermined path, the agent decides what to do next based on the current context and its progress toward the goal.

TOOL USE WITH A MULTI-STEP LOOP

The defining pattern of an agent combines two capabilities: the ability to use external tools and the autonomy to continue operating until the task is complete.

Since LLMs can only generate text, they cannot directly interact with external systems, access real-time information, or perform precise calculations. Tools bridge this gap by exposing external functionalities (web search, code execution, database queries, API calls) as callable functions. The LLM examines the available tools, decides whether one would help, and if so, specifies which tool to call with what parameters.

What transforms tool use into an agent is the loop. Rather than making a single tool call and stopping, the agent operates in a cycle: assess the current state, decide on an action, observe the result, and repeat until the task is complete. The LLM itself determines when to continue and when to stop.

This multi-step loop is the core mechanism that enables agents to handle tasks of unpredictable complexity. A simple query might resolve in one step; a complex research task might require dozens of iterations across multiple tools.

TOOL CREATION

At the highest level of autonomy, the agent doesn't just select from available tools. It creates new ones. This typically involves generating code to implement functionality that doesn't exist in the predefined toolkit.

For example, if an agent needs to process data in a specific format that no existing tool handles, it might write a custom parsing function, execute it, and use the results to continue its task. This capability allows agents to extend their own abilities dynamically, adapting to requirements that weren't anticipated when the system was designed.

1.3.3 Combining workflows and agents in practice

The distinction between workflows and agents isn't always binary in production systems. In practice, the most effective architectures often combine both approaches, using workflows to provide structure and predictability while embedding agents at specific points where flexibility is needed.

AGENTS AS NODES IN A WORKFLOW

A common pattern is to design an overall workflow with well-defined stages, but implement one or more of those stages as an agent. Consider a document processing pipeline:

1. **Document intake** (workflow): Validate format, extract metadata
2. **Content analysis** (agent): Research context, gather related information, synthesize findings
3. **Quality review** (workflow): Check against compliance rules, format output
4. **Delivery** (workflow): Route to the appropriate destination

The agent operates freely within its designated stage, making multiple tool calls, following chains of inquiry, and adapting to what it discovers. But the overall process maintains the predictability of a workflow. If the agent fails or produces unexpected results, the workflow can catch this at the quality review stage.

WHY COMBINE?

This hybrid approach offers several advantages. First, it contains complexity. Agent behavior is inherently less predictable, so limiting where agents operate makes the overall system easier to reason about and debug. Second, it optimizes costs. Agent loops can be expensive due to multiple LLM calls, so using them only where their flexibility is genuinely needed keeps operational costs manageable. Third, it enables graceful degradation. If an agent component fails, the workflow structure allows for fallback behaviors or human escalation at defined points.

The architectural choice isn't "workflow or agent" but rather "where in this system does agent behavior provide enough value to justify its costs?" This pragmatic framing will serve you well as you design your own LLM-powered applications.

1.4 Tasks that require agents

Before jumping into agent development, we need to ask the right questions in the right order. Two key decisions shape whether an agent is the appropriate solution:

1. Does this task require an LLM at all?
2. If an LLM is needed, does it require an agent, or will a workflow suffice?

Answering "no" at either stage means choosing a simpler, more cost-effective approach. Let's examine the criteria for each decision, then look at a benchmark that captures tasks where agents truly excel.

1.4.1 Tasks that require an LLM

Before considering the use of an LLM agent, you first need to determine whether the task itself requires an LLM. This foundational question matters because LLMs introduce significant overhead in terms of computational costs and potential for errors. This overhead may be unnecessary if the task can be solved with simpler, more deterministic approaches. For instance, if your task involves basic data processing with predictable inputs and outputs, traditional programming logic will be faster, cheaper, and more reliable than any LLM-based solution.

There are two main criteria for making this decision:

Tasks involving unstructured data. If a task requires analyzing unstructured data like text, images, or audio, it's a strong candidate for LLM usage. Traditional programming excels at structured data with clear schemas, but struggles with the ambiguity inherent in natural language or visual content. While models that handle diverse data formats are generally called Multimodal LLMs (or LMMs), they are fundamentally based on LLMs, so this book collectively refers to them as LLMs.

Input diversity. If user input is limited or the requested tasks are narrowly scoped, using a specialized ML model or rule-based system might be more cost-effective. Similarly, if the number of potential tasks is small and predefined, it's more efficient in terms of both cost and latency for a developer to hard-code the logic than to use an LLM. LLMs shine when inputs are unpredictable and varied, requiring flexible interpretation that can't be anticipated in advance.

1.4.2 Conditions for using agents

Once you've determined that an LLM is necessary, the next question is how to use it. Will a single LLM call suffice, or does the task require the iterative reasoning and tool use that define an agent?

Consider the following question:

"If Eliud Kipchoge could maintain his marathon world record pace indefinitely, how long would it take him to run from Earth to the Moon?"

Answering this question requires multiple steps. First, you need to search for who Kipchoge is and find his marathon world record. Then, you check Wikipedia for the distance between Earth and the Moon. Finally, you perform calculations to divide the distance by the speed and derive the final answer.

Anyone can do this task manually. However, the process of navigating between multiple websites to gather information, converting units, and performing calculations is time-consuming and error-prone. This kind of multi-step research task is exactly where agents shine.

As excitement around agents grows, there's a common trap people fall into: applying agents to every task simply because they're new and powerful. But just as you don't need a complex web framework to build a simple static website, choosing the right tool is key to building efficient systems.

Agents come with inherent trade-offs. First, costs are higher. Multiple LLM calls multiply API expenses compared to single requests. Second, latency increases. Response time accumulates with each reasoning step. Third, errors propagate. Mistakes in early steps cascade through the entire process.

Consider a customer service query that could be handled by a single LLM call costing $0.01. Processing the same query with an agent requiring 10 calls would cost $0.10. That's a 10x difference. When you're handling thousands of requests daily, this isn't just a technical decision; it's a business decision.

So when should you use an agent? You can decide based on these three criteria:

Task complexity. If it's difficult to predict how many steps a task will require, agents have the advantage. "Find the population of region A" is straightforward. In contrast, "Analyze how LLM agents will shape the future" requires gathering diverse information and synthesizing complex relationships. It's a task that's hard to predefine with static logic.

Task value. Since agents require multiple calls and dynamic decision-making, they cost more and respond more slowly. Therefore, the value of completing the task should outweigh the additional cost and latency.

Error cost and detectability. LLMs are prone to making incorrect decisions, and the risk increases with more calls. If errors lead to critical consequences, an agent might not be the best choice. Detectability also matters. In domains requiring specialized knowledge, users or developers may not even realize when an agent has produced faulty results.

1.4.3 GAIA: An agent gym

The Kipchoge question we examined earlier is actually from a benchmark called GAIA. In 2023, Meta and HuggingFace released GAIA (General AI Assistants), a dataset that systematically collects tasks requiring agents.

GAIA consists of question-answer pairs. Each question requires multi-step reasoning, web searches, calculations, and more. They're difficult to solve with a single LLM call, and it's not easy to predefine a clear workflow either. These are problems that naturally require an agentic approach.

GAIA was released in 2023, and since then, model capabilities have improved significantly, with more challenging benchmarks emerging. However, GAIA problems range from straightforward to genuinely difficult even for current models, making it well-suited for learning agent development. This book uses GAIA for three specific reasons.

Clear answers enable fast feedback. The core of agent development is "identifying when it fails and reducing those failures." When answers are clear, you can immediately verify whether you're heading in the right direction. You can rapidly iterate through cycles of experimentation and improvement without ambiguous evaluation criteria.

It's optimal for practicing the agent development cycle. The process of building an agent goes like this: First, determine whether an agent would help solve a specific problem. If there's potential, develop a prototype. Then observe when the agent fails, analyze the cause, and improve. And you repeat this observe-analyze-improve cycle. GAIA provides an appropriate difficulty level to experience this entire cycle from start to finish.

No domain knowledge required. Most GAIA problems center on web search and information synthesis. We all perform tasks in daily life that involve "searching, gathering information, and organizing it." You can intuitively understand the problems and solution processes without specialized knowledge in medicine or law.

Throughout this book, we'll use GAIA to track changes in agent performance as we add new techniques in each chapter. You'll see firsthand how each component, such as tool use, memory, and planning, actually improves the agent's problem-solving capabilities.

GAIA gives us a way to measure agent performance. But measurement alone doesn't improve an agent. What actually makes an agent better at solving these problems? The answer lies in how we engineer the context the LLM receives.

1.5 Context engineering

Before diving into agent development, let us clarify two terms that are often used interchangeably. A prompt is the input text that a user sends to an LLM. It can be further divided into system prompts, which specify how the model should respond and behave (such as "You are a friendly travel guide"), and user prompts (or messages), which contain the user's actual requests.

Context is a broader concept that encompasses all the information an LLM references when generating a response. This includes the system prompt, user messages, previous conversation history, tool execution results, retrieved documents, and more. In this sense, prompts are just one component of context.

Think of context as the LLM's working memory. Just as you can't solve a math problem if someone only tells you half the equation, an LLM can't complete a task if crucial information is missing from its context.

An LLM predicts the next token based solely on the information within its context window. The model generates text by leveraging patterns learned during training to make use of the information in the context. This simple fact is why context engineering is so critical in agent development.

1.5.1 Why agents fail

Agent failures can be attributed to two main causes. The first is insufficient model intelligence. This includes cases where the model fails to solve complex mathematical problems or makes errors in logical reasoning. Addressing this issue requires using a more powerful model or waiting for advances in model capabilities.

The second is a lack of necessary information. Even if a model is intelligent enough, it cannot produce correct answers if the information needed to complete the task is not in the context. For example, if someone asks, "What is our company's vacation policy?" but the company policy document is not in the context, even the most capable model cannot provide an accurate answer.

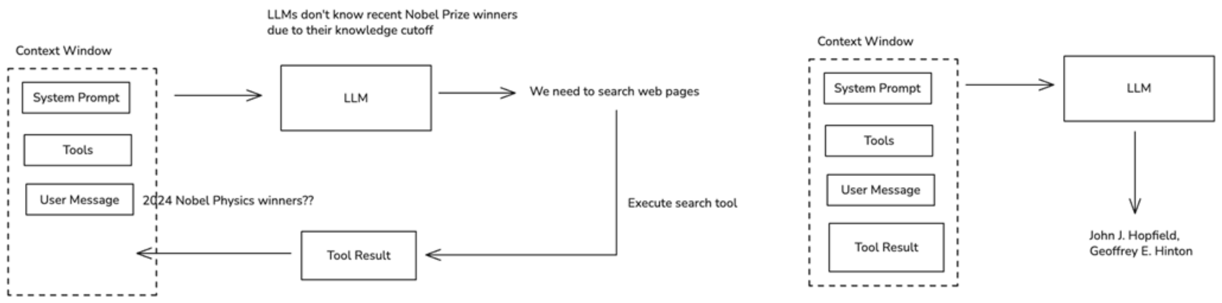
Interestingly, a significant portion of real-world agent failures stems from the second cause: information deficiency. An LLM can only be as smart as the context it is given. This is precisely why "what information to include in the context" is a key factor that determines agent performance.

1.5.2 From prompt engineering to context engineering

In the early days of LLM adoption, prompt engineering was the primary focus. Practitioners concentrated on how to write system prompts that would make models respond more accurately and which instructions would yield outputs in the desired format. Techniques like "Let's think step by step" and "You are an expert" emerged during this period.

However, the landscape changed as agents began using tools. When an agent performs a web search, the results are added to the context. When it executes code, the execution results are added. When it queries a database, the query results are added. Context began to change dynamically, evolving beyond static system prompts.

This shift raised a new question: "How do we maintain only the essential information needed for task completion in the context?" The approach to answering this question is context engineering.



Context Window
System Prompt
Tools
User Message
2024 Nobel Physics winners??
LLMs don't know recent Nobel Prize winners due to their knowledge cutoff
LLM
We need to search web pages
Execute search tool
Tool Result
Context Window
System Prompt
Tools
User Message
Tool Result
LLM
John J. Hopfield, Geoffrey E. Hinton

Figure 1.5 LLMs can only produce accurate, high-quality responses when sufficient information is provided in the context.

Context engineering is the discipline of providing the information an LLM needs to perform its tasks at the right time and in the right form. It goes beyond simply writing good prompts to comprehensively designing how tool execution results should be processed, how much conversation history to retain, and when to retrieve external knowledge.

1.5.3 Bigger context is not always better

The context windows of recent LLMs have grown dramatically. Most models now support context windows ranging from hundreds of thousands to a million tokens. So would filling the entire context window with all available data yield the best responses?

Unfortunately, no. Recent research has reported that model performance degrades as context length increases. This phenomenon is sometimes called Context Rot. The "Lost in the Middle" effect is particularly well documented, where models tend to miss important information located in the middle of long contexts.

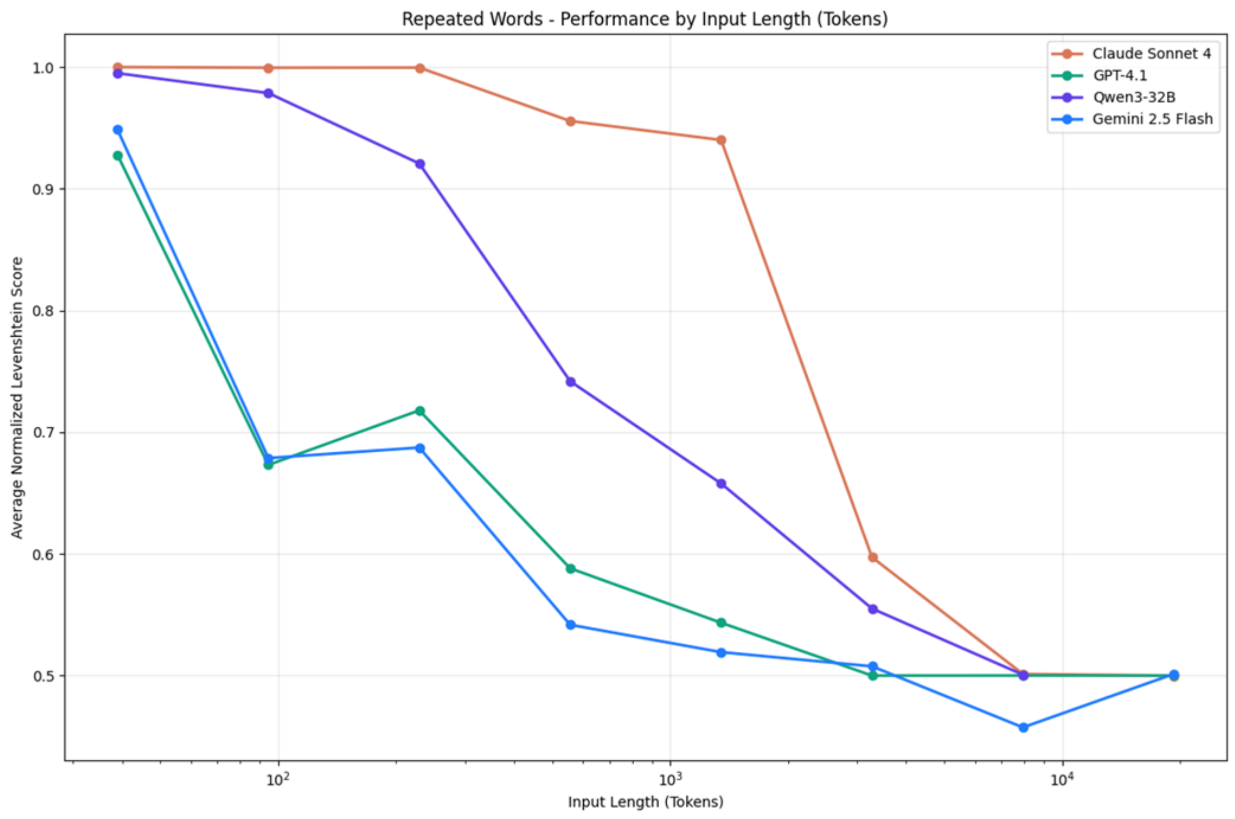


Figure 1.6 Even with large context windows, longer inputs can degrade model performance(Source: https://research.trychroma.com/context-rot).

This provides an important insight. To maximize model performance, we should not simply provide more information but rather selectively provide only highly relevant information. When the context is filled with unnecessary information, crucial details can get buried, or the model's attention can become dispersed.

1.5.4 Five context engineering strategies

Context engineering can be broadly categorized into five strategies. Each strategy is covered in detail across various chapters of this book.

Generation: Utilizing LLM-generated outputs within the context. This includes generating plans for complex tasks or reflecting on completed work to revise strategies. Beyond simply retrieving external information, this enables the LLM to structure and evolve the context on its own.

Retrieval: Fetching necessary information from external sources and adding it to the context. This includes web searches, database queries, file reading, and similar document retrieval from vector databases. This is a core strategy that enables agents to access up-to-date information or domain-specific knowledge not present in their training data.

Write: Recording information from the context to external storage. This includes saving important conversation content to long-term memory, recording intermediate calculation results to a scratchpad, or saving generated code to files. This overcomes the limitations of the context window and enables persistent storage of information.

Reduce: Shrinking the size of the context. This includes summarizing old conversations, deleting unnecessary information, or filtering information based on importance. This is essential for preventing Context Rot and focusing the model's attention on key information.

Isolate: Separating tasks or tools into distinct environments. This includes executing complex code in sandbox environments or assigning different specialized domains to separate agents. This prevents any single context from becoming overly complex and enables the construction of optimized contexts for each task.

Table 1.1 The five major categories of context engineering

Strategy	Description	Related Chapters
Generation	Constructing context with LLM-generated outputs	Chapter 7 (Planning & Reflection)
Retrieval	Bringing information from external sources into context	Chapter 3 (Tools), Chapter 5 (Knowledge Base), Chapter 6 (Memory)
Write	Saving from context to external storage	Chapter 6 (Memory), Chapter 8 (Workspace)
Reduce	Compressing and deleting context	Chapter 6 (Memory)
Isolate	Separating tasks/tools into distinct environments	Chapter 8 (Workspace), Chapter 9 (Multi-Agent)

Chapter 6 (Memory) is a hub where three strategies converge: Retrieval, Write, and Reduce. This is because memory systems involve retrieving past experiences (Retrieval), storing new experiences (Write), and summarizing or deleting old memories (Reduce).

1.5.5 The journey of this book

Throughout this book, we will apply context engineering strategies one by one to our agent as we progress through each chapter. Using the GAIA benchmark, we will quantitatively verify the impact of each strategy on agent performance.

Chapters 2 through 4 cover the journey of implementing a basic agent loop. In chapter 2, we learn how to use LLMs and test whether an LLM alone can solve GAIA benchmark problems. Some problems can be solved with just an LLM, but many require tools like web search or file reading. Chapter 3 addresses this by introducing how to build tools, how to make LLMs invoke them, and exploring MCP (Model

Context Protocol) as a way to integrate tools without building them from scratch. Chapter 4 combines these elements to implement an agent loop that iteratively uses tools to solve problems—not as a simple loop, but by building an abstracted Agent class.

Chapters 5 through 9 enhance context engineering by adding new elements to the basic agent loop. Chapter 5 adds retrieval capabilities by providing data sources that the agent can access. Chapter 6 implements conversation history management during task execution and long-term memory that allows insights gained from current tasks to be leveraged in future ones.

Chapter 7 explores planning the direction of task execution and adjusting course when lengthy data inputs occur or errors arise mid-process. Chapter 8 introduces context isolation by providing a workspace where the agent can execute code and work. Chapter 9 examines how to reduce and isolate context by creating multiple specialized agents.

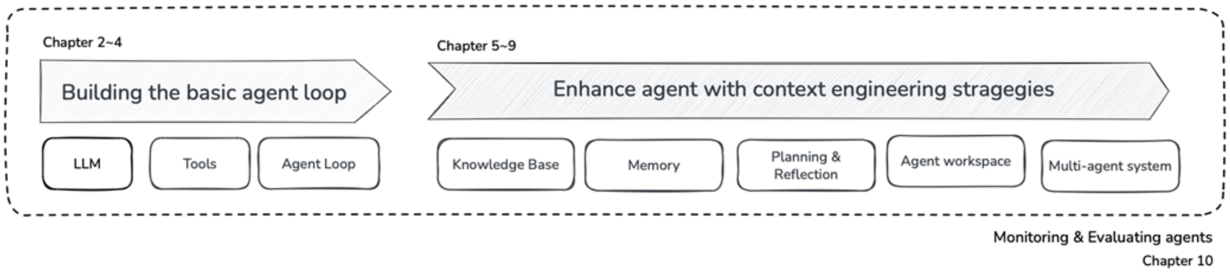


Figure 1.7 An overview of the journey through the book

Chapter 10 introduces methods for monitoring and evaluating agents. Throughout our journey, we used GAIA, a dataset with provided answers. However, when developing real-world agents, such helpful datasets are often unavailable. We examine commonly used datasets and explore evaluation methods for cases where no suitable dataset exists.

1.6 Prerequisites for reading this book

This book covers the full process of implementing LLM agents from scratch using plain Python. To fully understand and follow the content and hands-on exercises in this book, readers should meet the following basic prerequisites:

Required Preparation

1. Basic Knowledge of Python

All examples and exercises in this book are written in Python. Therefore, readers should be familiar with the following Python fundamentals:

- Basic data structures such as variables, data types, lists, and dictionaries
- Control flow structures like if statements and for/while loops
- Defining and calling functions
- Basic object-oriented programming (OOP) concepts, such as classes

If you're unfamiliar with these topics, it's recommended to study a beginner-level Python guide beforehand.

2. Development Environment Setup

To complete the exercises effectively, a local Python development environment is necessary.

- Install the latest version of Python in your local environment (recommended: Python 3.12 or later)
- Prepare a code editor or IDE (recommended: VS Code)
- It's helpful to learn how to install Python packages using pip or uv

3. API Keys and External Services

- OpenAI API key for accessing GPT models (primary LLM used in examples)
- Tavily API key for web search functionality in research agents
- Basic understanding of REST API concepts and HTTP requests

4. Cost Considerations

- OpenAI: The total cost for completing all exercises should be under $10
- Tavily Search API: Free tier includes up to 1,000 searches per month, which is sufficient for all book exercises
- We'll provide tips for cost-effective development and testing throughout the book

Once you've confirmed the above preparations, you're ready to begin your journey into LLM agent development with this book. Chapter 2 will establish the foundational understanding of LLMs that powers everything we'll build throughout this book.

1.7 Summary

- AI agents span a wide spectrum, from personal assistants like ChatGPT and Claude to customer-facing agents and specialized tools like Claude Code and Cursor. All share a common foundation: LLMs as their decision-making core.
- An LLM agent consists of three elements: the LLM (brain), tools (means of interacting with the external world), and a loop (iterative process until goal

completion). The LLM decides which tool to use and when to stop.

- Workflows are developer-defined execution flows where LLMs perform specific steps. Agents are LLM-directed flows where the model dynamically determines its own process. Production systems often combine both approaches.
- Use agents when tasks require multiple unpredictable steps, provide sufficient value to justify costs, and allow for error detection. The GAIA benchmark provides ideal practice problems for agent development.
- Context engineering is the discipline of providing the right information at the right time. Five strategies (Generation, Retrieval, Write, Reduce, Isolate) form the framework for building effective agents throughout this book.

2 The brain of AI agents: LLMs

This chapter covers

- Selecting the right LLM for agent development
- Using LLM APIs with LiteLLM
- Prompt engineering for agents
- Experiencing LLM limitations through the GAIA benchmark

Throughout this book, we'll build a Research Agent, a system that can interpret requests like "survey recent work on X" or "extract key findings from these PDFs," gather information from multiple sources, and synthesize findings into comprehensive answers. At the heart of this agent lies the LLM, serving as its decision-making brain. But before we can build an effective agent, we need to answer some fundamental questions: Which LLM should we use? How do we communicate with it programmatically? And crucially, what can an LLM do on its own, and where does it fall short?

We’ll address these questions head-on, starting with exploring how to choose an LLM for agent development, then get hands-on with APIs using LiteLLM to work seamlessly across providers. We'll examine prompt engineering principles that transform a general-purpose LLM into a reliable agent. Finally, we'll put LLMs to the test using the GAIA benchmark, directly experiencing their limitations when faced with real-world problems. This experiment will reveal exactly why agents need tools, setting the stage for everything we build in chapter 3 and beyond.

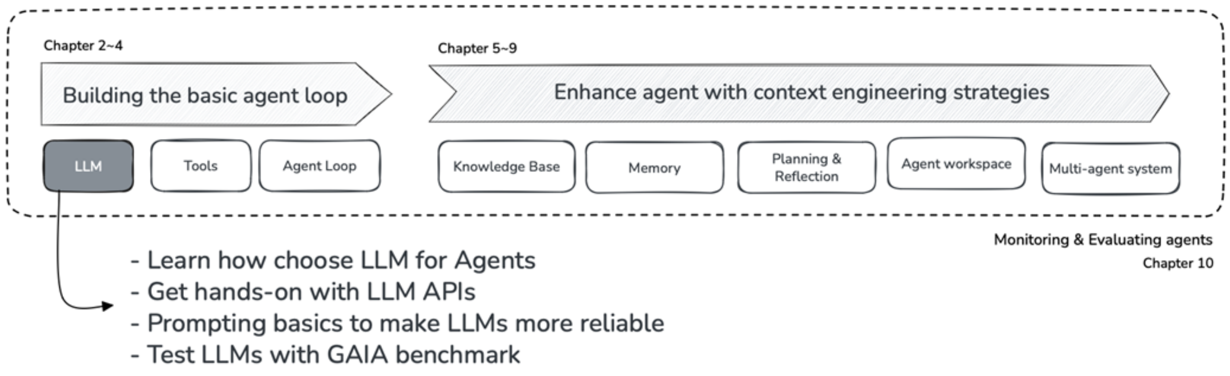

Chapter 2~4
Building the basic agent loop
Chapter 5~9
Enhance agent with context engineering strategies
LLM
Tools
Agent Loop
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Learn how choose LLM for Agents
- Get hands-on with LLM APIs
- Prompting basics to make LLMs more reliable
- Test LLMs with GAIA benchmark

Figure 2.1 Journey through the book – Chapter 2 in focus

2.1 Choosing LLMs for agents

When starting agent development, the first question you face is "Which LLM should I use?" This choice directly impacts your agent's performance, cost, and development speed. Let’s explore how to choose an LLM from an agent developer's perspective.

2.1.1 Starting with closed LLMs

LLMs fall into two broad categories. **Open LLMs** are models whose weights can be downloaded, run locally, and fine-tuned. Meta's Llama, Mistral AI's Mistral, Alibaba's Qwen, and DeepSeek fall into this category. **Closed LLMs** are accessible only through APIs; OpenAI's GPT, Anthropic's Claude, and Google's Gemini are the leading examples. This book focuses on building agents with Closed LLMs. Here's why Closed LLMs are the best choice when starting agent development.

RAPID EXPERIMENTATION AND VALIDATION

The core of agent development is quickly validating ideas and iterating. With Closed LLMs, you can start immediately with just an API key. Without preparing GPU servers or deploying model infrastructure, you can focus directly on implementing agent logic. Since you use the same environment from prototype to production, behaviors validated during development work identically in actual service.

WHEN YOU NEED TOP PERFORMANCE

An agent's performance ultimately depends on the LLM's intelligence. For an agent to perform complex tasks, it must accurately understand user intent, select appropriate tools, and derive results through multi-step reasoning. The models with the best performance today are still Closed LLMs. While Open LLMs are advancing rapidly, Closed LLMs maintain an edge in complex reasoning and tool-use accuracy. When the agent's "brain" is smart, the entire system works properly.

PRODUCTION-READY FEATURES

Modern agent development requires more than simple text generation. Closed LLM providers offer essential features for agents through stable APIs:

- **Tool Calling**: Naturally invoke external tools and APIs
- **Structured Output**: Generate responses in structured formats like JSON
- **Cost Optimization**: Efficient token usage through prompt caching and batch processing

These capabilities can be added to Open LLMs, but require custom implementation or third-party libraries. With Closed LLMs, you can immediately leverage battle-tested features.

2.1.2 Expanding to open LLMs

Once you've validated your agent's design with Closed LLMs, Open LLMs become worth considering in the next stage.

COST OPTIMIZATION

When handling high-volume traffic, self-hosting can be cheaper than API costs. Simple tasks like classification or extraction can be handled sufficiently by smaller Open models. This enables a strategy where core reasoning stays with high-performance Closed LLMs while auxiliary tasks are distributed to Open LLMs.

STABILITY AND CONTROL

Depending on external APIs means their outages become your outages. Self-hosted Open LLMs eliminate this external dependency. Additionally, when regulations require that sensitive data not leave your infrastructure, Open LLMs may be the only option.

A REALISTIC APPROACH

Transitioning to Open LLMs is best done gradually. First, validate your agent's design and behavior with Closed LLMs. Then port the same design to Open LLMs and test whether performance degradation falls within acceptable bounds. A hybrid configuration that keeps Closed LLMs for core reasoning while replacing only simple tasks with Open LLMs is also a good strategy.

2.1.3 Essential LLM capabilities for agents

Regardless of which LLM you choose, certain capabilities are essential for agent development.

TOOL CALLING (FUNCTION CALLING)

For an agent to interact with the external world, it must be able to invoke tools. The ability for the LLM to appropriately select and call tools like web search, code execution, and database queries is central to what makes an agent. We'll cover this capability in detail in chapter 3.

STRUCTURED OUTPUT

LLM outputs often need to integrate with other systems, not just be delivered to humans. The ability to generate structured output matching JSON or specific schemas is essential for integrating agents with real software systems.

LONG CONTEXT SUPPORT

Agents accumulate various information in their context: conversation history, tool execution results, retrieved documents, and more. Processing this information effectively requires a sufficiently long context window. Most recent LLMs support hundreds of thousands to a million tokens, but a longer context isn't always better. The key to context engineering is providing the right information at the right time in the right form.

DESIGNING FOR MODEL AGILITY IN A RAPIDLY CHANGING LANDSCAPE

The LLM market evolves at an astonishing pace. New models are released monthly, performance rankings shift regularly, and pricing structures are frequently updated. What's ideal today may be outdated tomorrow. Therefore, it's important to design with model flexibility from the start.

To stay informed, we can rely on trusted sources:

- **Artificial Analysis** (https://artificialanalysis.ai): Comprehensive comparison of quality, speed, and cost
- **LMArena Leaderboard** (https://lmarena.ai/leaderboard): User preference rankings based on real interactions
- **Official announcements from providers**: New model releases and pricing updates

This is also why we use LiteLLM throughout this book. With the same code, you can easily swap between different providers' models, allowing you to quickly test and transition when new models are released.

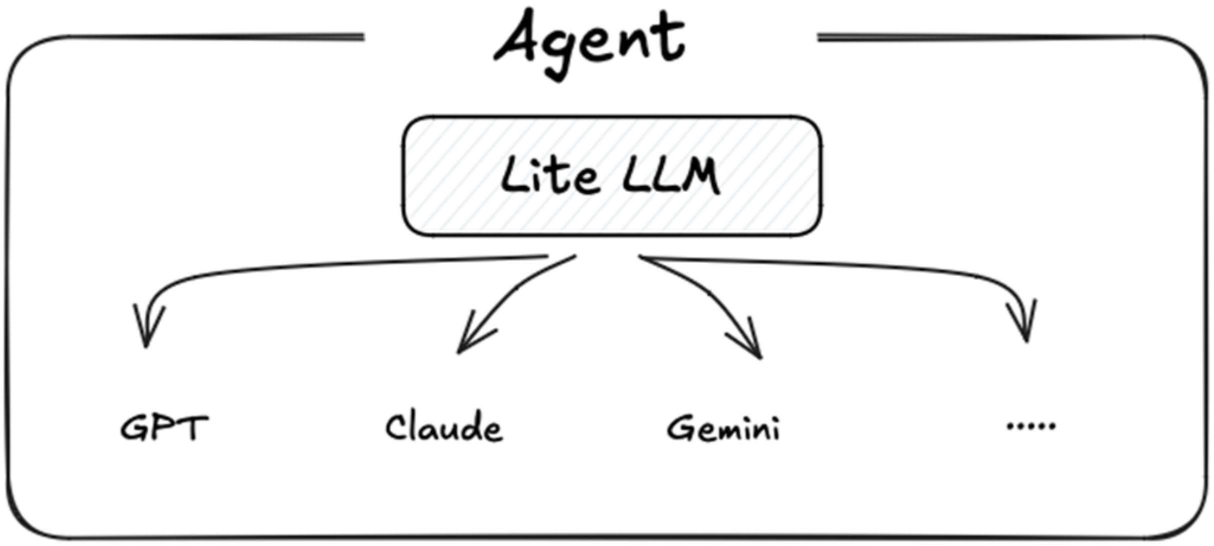


Figure 2.2 LiteLLM, a unified wrapper that lets you call multiple LLM providers through a single style

OUR CHOICE FOR THIS BOOK: OPENAI API

Throughout this book, we'll primarily use OpenAI's GPT models. For simple examples, we'll use gpt-5-mini to reduce costs, while agent implementations will center on gpt-5. The

OpenAI API is widely used with abundant learning resources, provides stable support for Tool Calling and Structured Output, and offers an easy-to-use Python library. This lets us focus on agent logic itself.

However, we designed our code to avoid vendor lock-in. Using LiteLLM, which we'll introduce in section 2.2, the same code can easily switch between OpenAI, Anthropic, Google, or even self-hosted Open LLMs.

A NOTE ON REASONING MODELS

You may have heard of "reasoning models" like OpenAI's o series, which spend additional thinking tokens to work through complex problems step by step. These models excel at tasks requiring deep analysis but come with higher latency and cost.

However, this distinction is becoming increasingly blurred. Anthropic's Claude now offers hybrid models that can switch between standard and reasoning modes. OpenAI's gpt-5 takes this further by automatically determining whether to engage in reasoning based on the query, so users no longer need to choose between model types. This trend toward unified models that adapt their reasoning depth to the task at hand simplifies the choice for agent developers.

2.2 LLM API basics for building agents

Now let's get hands-on with LLM APIs. In this section, we'll set up the development environment, learn the basics of OpenAI's API, and then use LiteLLM to unify multiple providers under a single interface. We'll also cover conversation management, structured output, and asynchronous calls—all essential capabilities for agent development.

2.2.1 Setting up the development environment

All code examples are available on GitHub at https://github.com/shangrilar/ai-agent-from-scratch

Start by cloning the repository.

```
git clone https://github.com/shangrilar/ai-agent-from-scratch
cd ai-agent-from-scratch
```

This book uses uv as the Python package manager. uv is faster than pip and handles virtual environment management and dependency resolution in one step. If you don't have uv installed, you can install it with the following command.

```
# macOS/Linux
curl -LsSf https://astral.sh/uv/install.sh | sh
```

The repository includes a `pyproject.toml` file, so a single command creates the virtual environment and installs all dependencies. Once installation is complete, launch Jupyter Lab to follow along with the examples.

```
uv sync
uv run jupyter lab
```

Next, you need to configure your API keys. Create a `.env` file in the project root directory and add your API keys. See Appendix ("API Key Setup") for instructions on obtaining API keys from each provider. At a minimum, you'll need an OpenAI API key to follow the examples in this book. You'll also need a HuggingFace token (`HF_TOKEN`) to access the GAIA benchmark dataset in section 2.4.

To use these environment variables in Python, load the `.env` file using the `python-dotenv` library, which is already included in the project dependencies. Since our notebook files are located in subdirectories like `chapter_02/`, we use `find_dotenv()` to locate the `.env` file in the project root.

Listing 2.1 Loading environment variables

```
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv())
```

The `find_dotenv()` function searches parent directories to find the .env file, so it works regardless of which subdirectory you're running your code from. Once loaded, LLM clients will automatically read their API keys from these environment variables.

2.2.2 Getting started with the OpenAI API

OpenAI's Chat Completions API has become the industry standard. Most LLM providers offer similar interfaces, so understanding the OpenAI API makes it easy to work with other providers.

Let's initialize the OpenAI client and send a simple request.

Listing 2.2 Basic OpenAI API call

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-5-mini",
    messages=[
        {"role": "system", "content": "You are a helpful assistan
t."},
        {"role": "user", "content": "What is the capital of Franc
e?"}
    ]
)

print(response.choices[0].message.content)
```

The two most important parameters in `chat.completions.create` are `model` and `messages`. The `model` parameter specifies which model to use, and `messages` contains the conversation.

The `messages` parameter is a list of message objects, each with a `role` and `content`. There are three roles:

- `developer(previous system)`: The system prompt that defines the model's overall behavior
- `user`: Messages from the user
- `assistant`: Responses generated by the model

The actual response text is found in `response.choices[0].message.content`. The reason `choices` is a list is that you can request multiple responses using the `n` parameter, though typically we only use the first one.

A commonly used parameter is `temperature`, which ranges from 0 to 2. Lower values produce more deterministic and consistent responses, while higher values generate more creative and varied outputs. Use 0–0.3 for tasks requiring accuracy, like data extraction or classification, and 0.7–1.0 for creative writing or brainstorming. The `max_tokens` parameter limits the maximum number of tokens to generate.

Other providers work similarly. For example, to use Anthropic's Claude.

Listing 2.3 Anthropic API call

```
from anthropic import Anthropic

client = Anthropic()

response = client.messages.create(
    model="claude-sonnet-4-5",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "What is the capital of Franc
e?"}
    ]
)

print(response.content[0].text)
```

While OpenAI and Anthropic APIs are similar, there are subtle differences: method names (`chat.completions.create` vs `messages.create`), response structure (`message.content` vs `content[0].text`), and required parameters (whether `max_tokens` is mandatory). Managing these differences across providers makes development cumbersome. In the next section, we'll see how to solve this problem.

2.2.3 Unifying providers with LiteLLM

When developing agents, you often need to compare different models or switch between them based on cost and performance requirements. If each provider has a different interface, such tasks become tedious. LiteLLM solves this problem; it's a library that lets you call over 100 LLMs through a unified interface.

Listing 2.4 Unified API calls with LiteLLM

```
from litellm import completion

response = completion(
    model="gpt-5-mini",
    messages=[{"role": "user", "content": "Hello!"}]
)

response = completion(
    model="claude-sonnet-4-5",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

A single `completion` function handles all providers. Just change the model name, and the response format is unified to the OpenAI format. API keys for each provider are automatically read from environment variables.

2.2.4 Conversation management: Handling stateless APIs

When using web services like ChatGPT or Claude, they appear to remember previous conversations. However, LLM APIs are stateless, so each API call is independent and has no memory of previous calls.

Let's verify this with code.

Listing 2.5 Demonstrating API statelessness

```
from litellm import completion

# First call
response1 = completion(
    model="gpt-5-mini",
    messages=[{"role": "user", "content": "My name is Jungjun."}]
)
print(response1.choices[0].message.content)

# Second call
response2 = completion(
    model="gpt-5-mini",
    messages=[{"role": "user", "content": "What is my name?"}]
)
print(response2.choices[0].message.content)
```

Even though we introduced ourselves in the first call, the second call has no memory of it. To maintain conversation continuity, the developer must manage the conversation history manually.

Listing 2.6 Managing conversation history

```
from litellm import completion

messages = []

# First exchange
messages.append({"role": "user", "content": "My name is Jungjun."})
response1 = completion(model="gpt-5-mini", messages=messages)
assistant_message1 = response1.choices[0].message.content
messages.append({"role": "assistant", "content": assistant_message
1})
print(assistant_message1)

# Second exchange - includes previous conversation history
messages.append({"role": "user", "content": "What is my name?"})
response2 = completion(model="gpt-5-mini", messages=messages)
assistant_message2 = response2.choices[0].message.content
print(assistant_message2)
```

We accumulate all conversation content in the `messages` list and pass the entire history with each call. User messages are added with the `user` role, and model responses with the `assistant` role. This allows the model to understand previous context and maintain a coherent conversation.

This approach becomes even more important in agent development. Agents need to add not just conversation history but also tool call results, search results, and intermediate reasoning steps to the context. As conversations grow longer, token usage increases, raising costs and slowing response times. Efficient context length management techniques are covered in detail in chapter 6.

2.2.5 Structured output

The natural language text generated by LLMs is great for humans to read, but inconvenient for programs to process. For an agent to call tools, it must output which tool to call with which arguments in a structured format. Structured Output is a feature that makes LLMs generate responses in a defined format, like JSON.

You can define the desired output format using Python's Pydantic library. Pydantic is a Python library for data validation that lets you define expected data structures as classes. By inheriting from `BaseModel`, you create a schema that specifies field names and their types. Pydantic then automatically validates any data against this schema, raising clear errors if the data doesn't match. This makes it easy to enforce data contracts and catch malformed inputs early. In this example, `ExtractedInfo` defines three fields: `name` and `email` are required strings, while `phone` is an optional string that defaults to `None` if not provided.

Listing 2.7 Structured output with Pydantic

```
from pydantic import BaseModel
from litellm import completion

class ExtractedInfo(BaseModel):
    name: str
    email: str
    phone: str | None = None

response = completion(
    model="gpt-5-mini",
    messages=[{
        "role": "user",
        "content": "My name is John Smith, my email is john@example.
com, and my phone is 555-1234."
    }],
    response_format=ExtractedInfo
)

result = response.choices[0].message.content
```

When you pass a Pydantic model to `response_format`, the LLM generates JSON matching that schema. You can use the parsed object directly from the response's `content` attribute.

Structured output plays a crucial role in agent development. In tool calling, which we'll cover in chapter 3, the LLM must output which tool to call with which arguments in a structured format. Converting user intent into appropriate actions is also an application of structured output.

2.2.6 Asynchronous calls

When developing agents, you'll often need to process multiple LLM requests simultaneously. This includes evaluating dozens of problems in benchmark tests, comparing responses from multiple models, or running multiple agents concurrently in a multi-agent system.

If you send requests sequentially, the total execution time equals the sum of each request's time. With asynchronous execution, you can start multiple requests simultaneously and process each response as it arrives.

Python handles asynchronous programming with `async/await` syntax and the `asyncio` library. LiteLLM supports asynchronous calls through the `acompletion` function.

Listing 2.8 Asynchronous LLM calls

```
import asyncio
from litellm import acompletion

async def get_response(prompt: str) -> str:
    response = await acompletion(
        model="gpt-5-mini",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content

prompts = [
    "What is 2 + 2?",
    "What is the capital of Japan?",
    "Who wrote Romeo and Juliet?"
]

# Execute all requests concurrently
tasks = [get_response(p) for p in prompts]
results = await asyncio.gather(*tasks)

for prompt, result in zip(prompts, results):
    print(f"Q: {prompt}")
    print(f"A: {result}\n")
```

Functions defined with `async def` are called coroutines. The `await` keyword waits for an asynchronous operation to complete while allowing other tasks to run in the meantime. `asyncio.gather` executes multiple coroutines concurrently and returns all results together.

If sending three requests sequentially takes 1 second each, the total time would be 3 seconds. By sending them asynchronously, you can receive all responses in about 1 second, which is the time of the longest request.

MANAGING CONCURRENT REQUESTS

When sending many requests simultaneously, two issues arise: rate limits from the API provider and transient failures from network issues or server overload. LiteLLM's `num_retries` parameter handles transient failures with automatic exponential backoff. For rate limiting, we can use Python's `asyncio.Semaphore` to limit how many requests run concurrently.

The key insight is that Semaphore controls access at the `async with` block level. By wrapping the `acompletion` call itself, we ensure that no matter how many tasks are running, only a limited number of actual API calls happen at once.

Listing 2.9 Rate-limited concurrent requests

```
import asyncio
from litellm import acompletion

# Limit to 10 concurrent requests
semaphore = asyncio.Semaphore(10)

async def call_llm(prompt: str) -> str:
    """LLM call with rate limiting and automatic retry."""
    async with semaphore:
        response = await acompletion(
            model="gpt-5-mini",
            messages=[{"role": "user", "content": prompt}],
            num_retries=3  # Automatic retry with exponential backof
f
        )
        return response.choices[0].message.content

# Even with 100 concurrent tasks, only 10 API calls run at a time
prompts = [f"What is {i} + {i}?" for i in range(100)]
tasks = [call_llm(p) for p in prompts]
results = await asyncio.gather(*tasks, return_exceptions=True)
```

The `call_llm` function wraps `acompletion` with both protections. The semaphore ensures that even if we create 100 tasks at once, only 10 API calls execute simultaneously. When one completes, the next waiting task proceeds. The `num_retries=3` parameter tells LiteLLM to automatically retry failed requests up to 3 times with exponential backoff, which is particularly useful for handling temporary rate limit errors.

The `return_exceptions=True` argument in `asyncio.gather` prevents a single failure from canceling all other tasks. Instead, exceptions are returned as values in the results list, allowing us to handle failures gracefully while still getting results from successful calls.

This pattern is essential for benchmark evaluations like GAIA, where we need to process hundreds of problems efficiently without overwhelming the API.

2.3 Enhancing agent intelligence: Prompt engineering

Once you've selected an LLM and learned how to use its API, you need to instruct that LLM to function properly as an agent. This is the domain of prompt design. Prompts fall into two main categories. User prompts are messages that users type into the chat interface, changing every turn and beyond the developer's control. System prompts, on the other hand, are developer-defined instructions that persist throughout the conversation, setting the agent's personality, permissions, constraints, and tool usage policies. This book focuses on system prompt design. A well-crafted system prompt transforms a general-purpose LLM into a reliable agent for specific tasks. While we cannot control user prompts, system prompts enable us to build agents that respond consistently to a wide range of user inputs.

2.3.1 The role of system prompts

System prompts are particularly important for agents. Agents act autonomously across multiple steps. System prompts define the behavioral rules that guide all of these decisions.

From a context engineering perspective, the system prompt is information that is always included in the context. Every time the agent calls a tool, analyzes results, or decides on its next action, the system prompt sits at the front of the context, guiding the agent's judgment. This is why the quality of the system prompt determines the quality of the agent's overall behavior.

What does a production agent's system prompt actually look like? Anthropic publishes Claude's system prompts (https://docs.anthropic.com/en/release-notes/system-prompts). Analyzing these prompts reveals the core principles of agent system prompt design. Note that the system prompt we're discussing here is the one automatically applied when using Claude through the claude.ai website or the mobile app's chat interface. When calling Claude through the API, developers write their own system prompts, so this particular prompt does not apply.

Claude's system prompt serves four main roles: defining the product's identity, specifying output format and style, setting boundaries on prohibited behaviors, and clarifying the limits of its knowledge.

DEFINING PRODUCT IDENTITY

The most fundamental role of a system prompt is defining "who" the agent is. Claude's system prompt includes a product information section that enables Claude to answer questions about itself accurately. For example, it specifies that the current model belongs to the Claude 4.5 model family, that there are three versions (Opus, Sonnet, and Haiku), and that various access methods exist, including the API and Claude Code.

Why is this information necessary? The LLM's training data may contain information about previous versions. By clearly providing current product information in the system prompt, you can prevent the model from stating outdated or inaccurate information. Additionally, by instructing the model to direct users to official documentation for features it doesn't know about, you prevent the provision of incorrect information.

The same applies when developing agents. An agent must accurately understand its own role and capabilities to clearly communicate to users what it can and cannot do. For example, if you're building a customer service agent, you should define in the system prompt what types of requests it can handle and when it should escalate to a human.

SPECIFYING OUTPUT FORMAT AND STYLE

A significant portion of Claude's system prompt is devoted to instructions about tone and formatting. Particularly notable are the detailed guidelines on bullet points and list usage. The prompt instructs Claude to avoid excessive formatting and to respond naturally in sentences and paragraphs during typical conversations. Lists are restricted to cases where the user explicitly requests them or where information is multifaceted enough that bullet points are essential.

Why are such detailed instructions necessary? LLMs tend to follow patterns from their training data, and since bullet points are heavily used in technical documentation and blog posts, models naturally prefer list formats. However, excessive formatting feels unnatural in everyday conversation. By explicitly adjusting this in the system prompt, you encourage responses appropriate to the situation.

SETTING BEHAVIORAL BOUNDARIES

The most important yet easily overlooked part of a system prompt is defining "what not to do." Claude's system prompt includes a refusal_handling section that clearly specifies the types of requests the model should refuse. These include content that threatens child safety, information for making chemical, biological, or nuclear weapons, and writing malicious code. It also specifies restrictions on creative content featuring real public figures and persuasive content.

This boundary-setting is particularly important in agent development for a specific reason. Because agents act autonomously, they can behave in unintended ways without clear boundaries. For example, if a customer service agent has the authority to process refunds but not to delete accounts, this must be made clear in the system prompt. If a financial agent can provide information but should not give investment advice, that boundary must also be clearly defined.

Claude's prompt also has a separate section for legal and financial advice. Instead of making confident recommendations, it instructs the model to provide factual information so users can make their own decisions, and to remind users that it is not a lawyer or financial advisor. Defining "what not to do" is just as important as defining "what to do."

CLARIFYING KNOWLEDGE LIMITS

Another important area is how to handle the model's knowledge limitations. Claude's system prompt specifies the knowledge cutoff date and instructs the model to honestly say it cannot know about events after that date. It also instructs the model to suggest turning on the web search tool for questions requiring current information.

This provides a crucial lesson for agent development. Agents must know their own limitations, and for requests that exceed those limits, they should either use appropriate tools or honestly inform the user. An agent that admits what it doesn't know and suggests alternatives is more trustworthy than one that pretends to know everything.

2.3.2 Guidelines for agent prompts

Now, let’s explore prompt design principles specific to agents. What is the fundamental difference between a chatbot and an agent? A chatbot converses with users, while an agent works on their behalf. This difference demands a fundamentally different approach to prompt design.

ACTING AUTONOMOUSLY

What happens when you ask a typical chatbot, "What's the weather like in Seoul today?" If the chatbot has access to a web search tool, it might respond, "Would you like me to search for that?" This seems polite, but from an agent's perspective, it's inefficient. When users ask about the weather, they want an answer, not a decision about whether to search.

Agents should act before asking questions. When the user's intent is clear, execute immediately without asking for permission. If they ask about the weather, search. If calculation is needed, calculate. If they ask to analyze a file, open the file. This is the first principle that distinguishes agents from chatbots.

This doesn't mean agents should act blindly in every situation. When the user's intent is too ambiguous to determine, clarification is necessary. The key principle is "if it's not ambiguous, don't ask, just execute." When reflecting this principle in a system prompt, you might write something like this.

```
When the user's intent is clear, execute immediately
without confirmation.

Only when intent is unclear, ask minimal questions to
clarify.
```

The last line is important. It allows the agent to act autonomously while providing a safety net to prevent it from going down the wrong path when the situation is genuinely ambiguous.

USING TOOLS STRATEGICALLY

Acting autonomously essentially means using tools appropriately. An agent's capabilities depend on what tools it can access and how well it uses them. How you design tool usage guidelines in the system prompt determines your agent's performance.

Setting the default attitude

First, establish the default attitude toward tool use. "Use when necessary" is vague; "use proactively, without asking permission" is clearer. For an agent with a search tool, you might specify: "For rapidly changing information (stock prices, breaking news), search immediately. For slower-changing topics (government positions, company CEOs), always search if the question asks about the current status." Clearly defining when to use tools is the key.

Defining trigger patterns

Providing guidance that helps the LLM identify when to invoke tools based on linguistic signals is effective. For example, if your agent has a tool that searches past conversations, you could define trigger patterns like this:

> Use the conversation search tool when you see expressions like:
>
> - Explicit references: "what we discussed...", "as I mentioned before..."
> - Time references: "what did we talk about yesterday", "show me last week's chat"
> - Implicit signals: "you suggested", "we decided", "my project", "that bug"

Providing specific linguistic patterns like these helps the LLM judge tool invocation timing more accurately. When developing agents, listing out how users typically express situations where a particular tool is needed proves helpful.

Specifying negative guides

Knowing when not to use tools is equally important. For a search tool, you might specify: "Do not search for timeless information, fundamental concepts, definitions, or well-established technical facts." Questions like "how do I write a for loop in Python" or "what is the Pythagorean theorem" don't require searching.

Unnecessary tool calls increase costs and latency. The bigger problem is that if an agent searches for everything, user experience suffers. Nobody wants to use an agent that makes them wait several seconds for simple questions. Specifying when not to use tools is an important part of agent prompt design.

Anchoring with examples

Finally, providing concrete input-output pairs as examples helps the LLM understand expected behavior much better.

```
Example 1:

User: "What's Samsung Electronics' current stock price?"

Action: [web_search: Samsung Electronics stock price]

Response: Samsung Electronics' current stock price is XX
won.

Example 2:

User: "What is the Pythagorean theorem?"

Action: Respond directly without searching

Response: The Pythagorean theorem states that in a right
triangle...
```

With examples like these, the LLM clearly understands how to behave in similar situations. Concrete examples serve as more powerful guides than abstract rules.

TOOL INFORMATION IS ALSO PART OF THE PROMPT

So far, we've explored how to instruct agents on "how to use" tools. But for an agent to use tools effectively, another element is necessary: descriptions of the tools themselves.

When providing tools to an LLM, you need to explain what each tool does, what parameters it requires, and what results it returns. The quality of these descriptions directly affects the LLM's tool selection accuracy. Even tools with identical functionality can be called at the wrong time or not called at all if their descriptions are vague.

How to write tool descriptions and how the mechanism by which LLMs invoke tools (Tool Calling) works will be covered in depth in chapter 3. For now, just remember that designing tool usage guidelines in the system prompt and defining the tools themselves are separate tasks, and both are critical to agent performance.

2.4 Experiencing LLM limitations with GAIA

So far, we've explored how to use LLM APIs and the principles of prompt design. Now let's apply LLMs to real problems and directly observe their capabilities and limitations. This experiment will make clear why agents need tools.

2.4.1 Why GAIA? Setting goals for agent development

Before diving into the experiment, let's discuss a crucial principle of agent development, which is that you need a clear target to aim for. When building any software system, having measurable goals is important. For agents, this becomes even more critical because their behavior is inherently unpredictable. Unlike traditional software, where you can trace through deterministic logic, agents make decisions dynamically based on context. Without a concrete benchmark, you might spend weeks tweaking prompts and adding features without knowing if you're actually making progress.

Throughout this book, we're building a Research Agent, a system that can gather information from multiple sources, analyze findings, and produce comprehensive answers. But how do we know if our agent is actually getting better? We need test cases that meet certain criteria.

- Have clear, verifiable answers (so we can measure accuracy objectively)
- Represent realistic user requests (so improvements translate to real-world value)
- Cover a range of difficulty levels (so we can track progress as we add capabilities)

This is where GAIA comes in. We introduced GAIA (General AI Assistants) in chapter 1 as an ideal benchmark for practicing agent development. Now we'll use it as our development compass.

THINKING OF GAIA AS SIMULATED USER INPUTS

Here's a useful mental model. Imagine each GAIA problem as a request from a real user of our Research Agent. When someone asks, "If Eliud Kipchoge could maintain his marathon pace indefinitely, how long would it take to run to the Moon?" they expect the agent to figure out how to answer, not to ask them where to find Kipchoge's pace or the Earth-Moon distance.

This framing helps us think about agent development the right way. We're not just trying to pass a benchmark; we're building a system that can handle the kinds of questions real users would ask. GAIA problems happen to have verified answers, which let us measure our progress objectively. But the goal is always to build an agent that genuinely helps users.

LOADING THE GAIA DATASET

Let's load the GAIA dataset from HuggingFace. GAIA is a gated dataset, which means you need a HuggingFace account and must accept the dataset's terms of use before accessing it. Visit https://huggingface.co/datasets/gaia-benchmark/GAIA and click the "Agree and access repository" button to accept the dataset's terms of use. Then ensure your `HF_TOKEN` environment variable is set as described in section 2.2.1. We'll focus on Level 1 problems in this chapter to establish a baseline of what LLMs can do without tools.

Listing 2.10 Loading GAIA Level 1 problems

```
from datasets import load_dataset

level1_problems = load_dataset("gaia-benchmark/GAIA", "2023_level1",
split="validation")
print(f"Number of Level 1 problems: {len(level1_problems)}")
```

GAIA problems are categorized into three difficulty levels. Level 1 questions generally require no tools, or at most one tool, with no more than 5 steps. Level 2 questions involve more steps, roughly between 5 and 10, and combining different tools is needed. Level 3 questions are designed for a near-perfect general assistant, requiring arbitrarily long sequences of actions, any number of tools, and access to the world in general. We start with Level 1 because our goal here is to establish what LLMs can do on their own, before we give them any tools.

UNDERSTANDING THE PROBLEM STRUCTURE

Each GAIA problem has a consistent structure. Let's examine the Kipchoge problem from chapter 1.

```
{
    'task_id': 'e1fc63a2-da7a-432f-be78-7c4a95598703',
    'Question': 'If Eliud Kipchoge could maintain his record-making
marathon pace '
                'indefinitely, how many thousand hours would it take
him to run '
                'the distance between the Earth and the Moon at its
closest approach? '
                'Please use the minimum perigee value on the Wikiped
ia page for the Moon '
                'when carrying out your calculation. Round your resu
lt to the nearest '
                '1000 hours and do not use any comma separators if n
ecessary.',
    'Level': 1,
    'Final answer': '17',
    'file_name': '',
    'Annotator Metadata': {
        'Steps': '1. Googled Eliud Kipchoge marathon pace...',
        'Tools': '1. A web browser.\n2. A search engine.\n3. A calcu
lator.',
        'Number of tools': '3'
    }
}
```

The `Question` field contains what a user would ask our agent. `Final answer` is the ground truth we'll compare against, in this case simply "17." The `file_name` field indicates whether the problem includes an attached file (empty here means no attachment). The `Annotator Metadata` reveals what the problem creators needed to solve it, which includes a web browser, a search engine, and a calculator.

This metadata is particularly revealing. Even though this is a Level 1 problem, the "easiest" category, the annotators needed three different tools to solve it. They had to search for Kipchoge's marathon pace, look up the Earth-Moon perigee distance on Wikipedia, and perform calculations.

2.4.2 Experiment setup

To systematically evaluate LLMs on GAIA problems, we need to set up several components: define the response format, create an API calling function with proper error handling, select models to test, and establish how we'll judge correctness.

DEFINING THE RESPONSE FORMAT

We'll use Structured Output as covered in section 2.2.5. This ensures we can reliably extract the model's answer for comparison with the ground truth.

Listing 2.11 Pydantic model for GAIA responses

```
from pydantic import BaseModel
class GaiaOutput(BaseModel):
    is_solvable: bool
    unsolvable_reason: str = ""
    final_answer: str = ""
```

The `is_solvable` field indicates whether the model believes it can solve the problem with its current capabilities. When `is_solvable` is False, the `unsolvable_reason` field captures why the model cannot provide an answer (e.g., "I need current information but cannot search the web"). The `final_answer` field contains the response we'll compare against the ground truth. Making both `unsolvable_reason` and `final_answer` optional with empty string defaults allows clean handling of both solvable and unsolvable cases.

BUILDING THE EVALUATION PIPELINE

We use GAIA's standard evaluation prompt, which instructs models to provide answers in a consistent format.

> You are a general AI assistant. I will ask you a question.
>
> First, determine if you can solve this problem with your current capabilities and set "is_solvable" accordingly.
>
> If you can solve it, set "is_solvable" to true and provide your answer in "final_answer".
>
> If you cannot solve it, set "is_solvable" to false and explain why in "unsolvable_reason".
>
> Your final answer should be a number OR as few words as possible OR a comma separated list of numbers and/or strings.
>
> If you are asked for a number, don't use comma to write your number neither use units such as $ or percent sign unless specified otherwise.
>
> If you are asked for a string, don't use articles, neither abbreviations (e.g. for cities), and write the digits in plain text unless specified otherwise.
>
> If you are asked for a comma separated list, apply the above rules depending on whether the element is a number or a string.

This prompt is designed to elicit answers that can be compared directly against GAIA's ground truth. The formatting rules (no commas in numbers, no units unless specified) ensure consistent comparison.

With our response format and system prompt defined, we need to build the infrastructure to evaluate multiple models across many problems efficiently. This requires handling several practical concerns: rate limits from different API providers, error handling for failed requests, and parallel execution to keep evaluation time reasonable.

Managing API rate limits

Different providers impose different rate limits on API requests, and your specific limits depend on your API tier and usage history. Rather than using a single semaphore for all requests, we create separate semaphores for each provider. This prevents a burst of requests to one provider from blocking requests to another.

Listing 2.12 Provider-specific rate limiting

```
PROVIDER_SEMAPHORES = {
    "openai": asyncio.Semaphore(30),
    "anthropic": asyncio.Semaphore(10),
}

def get_provider(model: str) -> str:
    """Extract provider name from model string."""
    return "anthropic" if model.startswith("anthropic/") else "opena
i"
```

The semaphore values here (30 for OpenAI, 10 for Anthropic) are conservative starting points that work for most API tiers. You should adjust these based on your specific rate limits, which you can find in each provider's documentation or dashboard. If you encounter rate limit errors during execution, reduce the values; if your tier allows higher concurrency, you can increase them for faster throughput.

The core solving function

The `solve_problem` function handles a single API call with proper rate limiting and error handling. It acquires the appropriate semaphore before making the request, ensuring we never exceed our concurrent request limit for any provider.

Listing 2.13 Core problem-solving function

```
async def solve_problem(model: str, question: str) -> GaiaOutput:
    """Solve a single problem and return structured output."""
    provider = get_provider(model)

    async with PROVIDER_SEMAPHORES[provider]:
        response = await acompletion(
            model=model,
            messages=[
                {"role": "system", "content": gaia_prompt},
                {"role": "user", "content": question},
            ],
            response_format=GaiaOutput,
            num_retries=2,
        )
        finish_reason = response.choices[0].finish_reason
        content = response.choices[0].message.content

        if finish_reason == "refusal" or content is None:
            return GaiaOutput(
                is_solvable=False,
                unsolvable_reason=f"Model refused to answer (finish_
reason: {finish_reason})",
                final_answer=""
            )
        return GaiaOutput.model_validate_json(content)
```

The function uses `async with` to acquire and release the semaphore automatically. Inside this block, we call LiteLLM's `acompletion` with our system prompt and the problem question. The `response_format=GaiaOutput` parameter enables structured output, and `num_retries=2` handles transient failures automatically.

We also handle the case where a model refuses to answer. Some models may decline certain questions for safety reasons, returning a "refusal" finish reason. Rather than raising an exception, we capture this as an unsolvable problem with an appropriate reason.

Validating answers

GAIA uses exact match evaluation, meaning the model's answer must match the ground truth exactly (ignoring case and whitespace). This simple validation function handles the comparison.

Listing 2.14 Answer validation

```
def is_correct(prediction: str | None, answer: str) -> bool:
    """Check exact match between prediction and answer (case-insensi
tive)."""
    if prediction is None:
        return False
    return prediction.strip().lower() == answer.strip().lower()
```

The case-insensitive comparison and whitespace stripping account for minor formatting differences that shouldn't count as errors. If the prediction is None (due to an error or refusal), we simply return False.

Evaluating individual problems

The `evaluate_gaia_single` function wraps `solve_problem` to capture all relevant metadata about each evaluation attempt. This includes not just whether the answer was correct, but also what the model predicted, whether it thought the problem was solvable, and any errors that occurred.

Listing 2.15 Single problem evaluation

```
async def evaluate_gaia_single(problem: dict, model: str) -> dict:
    """Evaluate a single problem-model pair and return result."""
    try:
        output = await solve_problem(model, problem["Question"])
        return {
            "task_id": problem["task_id"],
            "model": model,
            "correct": is_correct(output.final_answer, problem["Fina
l answer"]),
            "is_solvable": output.is_solvable,
            "prediction": output.final_answer,
            "answer": problem["Final answer"],
            "unsolvable_reason": output.unsolvable_reason,
        }
    except Exception as e:
        return {
            "task_id": problem["task_id"],
            "model": model,
            "correct": False,
            "is_solvable": None,
            "prediction": None,
            "answer": problem["Final answer"],
            "error": str(e),
        }
```

Capturing this metadata proves valuable for later analysis. When we examine failures, we can distinguish between problems the model attempted but got wrong versus problems it recognized as unsolvable. The `unsolvable_reason` field will help us categorize what capabilities the model is missing.

Running the full experiment

Finally, we need to orchestrate evaluation across all problems and models. The `run_experiment` function creates tasks for every problem-model combination and executes them concurrently using `asyncio.gather`. We use tqdm_asyncio.gather to display a progress bar during execution.

Listing 2.16 Running the full experiment

```
async def run_experiment(
    problems: list[dict],
    models: list[str],
) -> dict[str, list]:
    """Evaluate all models on all problems."""
    tasks = [
        evaluate_gaia_single(problem, model)
        for problem in problems
        for model in models
    ]

    all_results = await tqdm_asyncio.gather(*tasks)

    # Group results by model
    results = {model: [] for model in models}
    for result in all_results:
        results[result["model"]].append(result)

    return results
```

The function returns results grouped by model, making it easy to calculate per-model accuracy and analyze failure patterns.

Selecting models and running the experiment

For this experiment, we test four models representing different capability tiers from two providers. From OpenAI, we use gpt-5 (flagship model) and gpt-5-mini (cost-optimized). From Anthropic, we test claude-sonnet-4-5 (balanced performance) and claude-haiku-4-5 (fast and economical).

Listing 2.17 Model selection and execution

```python
MODELS = [
    "gpt-5",
    "gpt-5-mini",
    "anthropic/claude-sonnet-4-5",
    "anthropic/claude-haiku-4-5"
]

subset = level1_problems.select(range(20))
results = await run_experiment(subset, MODELS)
```

We start with a subset of 20 problems to keep initial experimentation fast and cost-effective. Once we've verified the pipeline works correctly, we can expand to the full dataset. The `dataset.select(range(20))` call creates a subset containing the first 20 problems from our Level 1 dataset.

2.4.3 Results and analysis: Why LLMs need tools

Let's examine what happened when we ran four models against 20 Level 1 GAIA problems.

Table 2.1 Experiment results across models

Model	Accuracy	Judged Solvable
gpt-5	10/20 (50%)	11/20 (55%)
gpt-5-mini	4/20 (20%)	8/20 (40%)
claude-sonnet-4-5	3/20 (15%)	4/20 (20%)
claude-haiku-4-5	3/20 (15%)	4/20 (20%)

ONLY FIVE PROBLEMS ARE TRULY SOLVABLE WITHOUT TOOLS

The most important finding is that only 5 of these 20 problems can actually be solved without external tools. These are problems where the answer can be derived purely from reasoning or from knowledge reliably present in training data. This means the theoretical maximum accuracy for any LLM without tools is 25%. Yet we see models attempting far more problems and sometimes succeeding. How?

MODELS VARY IN SELF-ASSESSMENT ACCURACY

One interesting pattern is how models differ in judging whether they can solve a problem. Some models were conservative, judging only four problems as solvable, which is close to the true count of five. These models showed good calibration, meaning when they said "I cannot solve this," they were usually correct about needing external information.

Other models were more ambitious, attempting 8 to 11 problems. Surprisingly, they often succeeded on problems that seemingly require external information. The likely explanation is training data. Modern LLMs are trained on vast amounts of web content, including Wikipedia articles and reference materials. When a problem asks about a well-documented fact, the model may have encountered this information during training.

However, this "knowledge" is unreliable for agent development. The model cannot distinguish between information it knows accurately versus information it is confabulating. We cannot depend on training data to provide current or precise information.

MODEL REFUSALS

Another observation is that some models refused to answer certain problems entirely, returning a refusal rather than attempting an answer. For example, one GAIA problem contains text written backwards:

```
.rewsna eht sa "tfel" drow eht fo etisoppo eht etirw ,ecnetnes siht
dnatsrednu uoy fI
```

When reversed, this simply asks to write the opposite of "left" as the answer. However, some models flagged this as potentially attempting to bypass safety filters through obfuscation and refused to engage with it.

Different providers and models have varying safety policies that determine what triggers a refusal. A question that one model answers without hesitation might cause another to refuse entirely. These policies evolve as providers update their safety guidelines, and the specific triggers are often not publicly documented in detail.

For production agents, this means you should test your specific use case across your target models and have fallback strategies when refusals occur. Check each provider's usage policies and safety documentation to understand their general approach, but expect that edge cases will require empirical testing rather than relying solely on documentation.

WHY 15 PROBLEMS REMAIN UNSOLVABLE

Analyzing the problems that models correctly identified as unsolvable reveals clear patterns in what capabilities are missing. Some problems require both web search and file reading, which is why the counts sum to more than 15.

Table 2.2 Required capabilities for unsolvable problems

Missing Capability	Count
Web search and webpage access	11
File reading	6

The Kipchoge problem from chapter 1 exemplifies why web access matters. Even if the model happens to know Kipchoge's marathon pace from training, the problem explicitly requires checking "the minimum perigee value on the Wikipedia page for the Moon." The user expects the agent to look this up, not to guess from training data.

File-dependent problems present an even clearer limitation. When a problem includes an attached spreadsheet or document, no amount of training data can help. The model simply cannot access information that it was never given.

KEY TAKEAWAYS

This experiment reveals three critical insights for agent development:

First, LLMs have reasonable self-awareness of their limitations. When properly prompted, models can often identify when they cannot solve a problem. This is valuable because an agent that knows what it cannot do can use appropriate tools instead.

Second, training data is not a substitute for tools. Even when models "know" information from training, this knowledge is unreliable for precise, current, or verifiable answers. Agents need tools to access authoritative sources.

Third, most real-world questions require external capabilities. In our sample, 75% of even the "easiest" GAIA problems required tools. For a Research Agent to be genuinely useful, it must be able to search the web, read files, and perform calculations.

We now have a baseline: without tools, even capable LLMs solve only a fraction of Level 1 problems. In chapter 3, we will give our agent the ability to use tools, starting with web search. We will then return to these same GAIA problems and measure how much our agent improves.

2.5 Summary

- When starting agent development, Closed LLMs offer practical advantages: out-of-the-box features like tool calling and structured output, stable APIs that let developers focus on agent logic, and top-tier performance for complex reasoning tasks. Open LLMs become relevant later for cost optimization.
- LiteLLM provides a unified interface across providers, enabling model flexibility in a rapidly evolving landscape. Key API patterns include conversation management, structured output with Pydantic, and asynchronous calls with rate limiting.
- System prompts define agent behavior: product identity, output format, behavioral boundaries, and knowledge limits. For agents, prompts should emphasize autonomous action and strategic tool use rather than asking for permission.
- GAIA benchmark reveals that even capable LLMs solve only 25% of "easy" problems without tools. 75% of problems require external capabilities like web search and file reading—motivating Chapter 3's focus on tools.
- Models show reasonable self-awareness of limitations when properly prompted, but training data knowledge is unreliable for precise, current, or verifiable answers. Agents need tools to access authoritative sources.

3 Enabling actions: Tool use

This chapter covers

- LLM limitations and why they need tools
- Tool calling and execution
- Building and integrating custom tools
- MCP for tool standardization

On their own, LLMs can't access external data or interact with external systems. They need tools and the ability to select and utilize those tools, which is central to implementing agents. A basic agent performs tasks step by step by repeatedly selecting the next action to take, and tools make this possible.

Now, let's explore how to build tools and inform the LLM how to use them. We'll also learn how the LLM chooses the appropriate tool for a given task and relays tool executions back to the LLM. Then we'll examine the Model Context Protocol (MCP) by Anthropic, an initiative that standardizes the development and use of tools, as shown in figure 3.1.

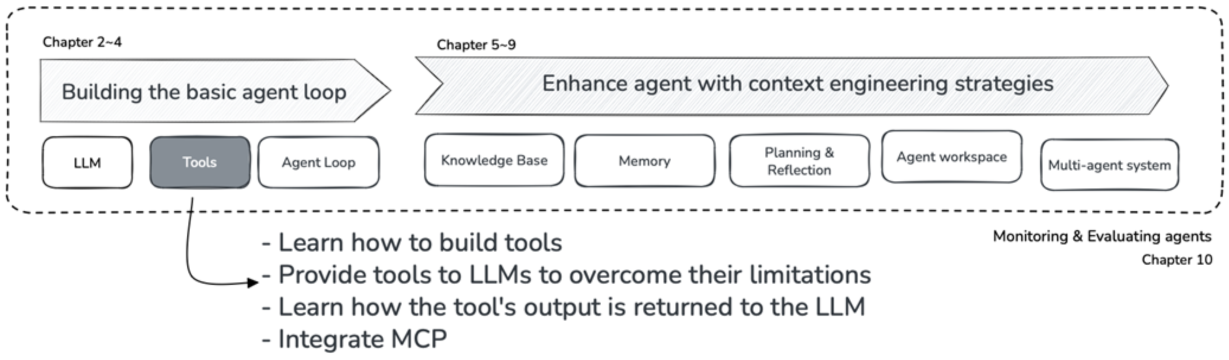

Chapter 2~4
Building the basic agent loop
Chapter 5~9
Enhance agent with context engineering strategies
LLM
Tools
Agent Loop
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Learn how to build tools
- Provide tools to LLMs to overcome their limitations
- Learn how the tool's output is returned to the LLM
- Integrate MCP

Figure 3.1 Book structure overview - Chapter 3 in focus.

As shown in figure 3.2, we'll construct an LLM system that takes real-world actions using tools, focusing on tool calling and tool execution components. We'll provide the LLM with context that includes information about the available tools and their usage (labeled as "tool_info" in the figure). If the LLM determines that a tool is needed (1), the tool is executed (2), and the result is added to the LLM's context (3). If the LLM then concludes that no further tool use is necessary and it can generate a final answer directly, it outputs that response (4), which is both added to the context and returned to the user (5).

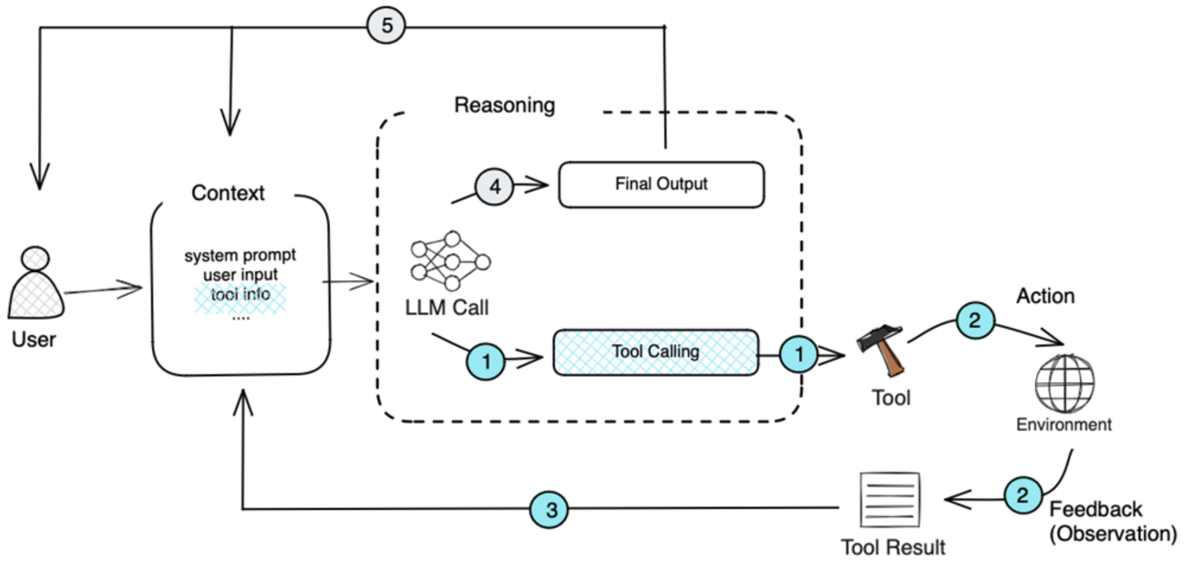


Figure 3.2 The LLM receives user input, reasons about it, selects the necessary tool, receives the execution result, and produces a final response.

So, what does it take for an LLM to become an agent that interacts with the external world and achieves goals? Let's take a closer look at the types of tools that help LLMs overcome their limitations.

3.1 LLM tools

Despite their impressive capabilities, LLMs have inherent limitations that prevent them from functioning as fully autonomous agents. These limitations fall into three categories: temporal, interaction, and functional.

First, LLMs face a **temporal limitation**—they cannot access information beyond their training data or retrieve data that changes in real time. This makes them unreliable for tasks requiring current knowledge, such as answering questions about recent events or live market conditions.

Second, LLMs have an **interaction limitation**—they cannot directly influence the external world. While they can generate text describing an action, they cannot actually book a flight, send an email, or control a device.

Third, LLMs suffer from **functional limitations** in specialized domains. Tasks requiring precise computation, code execution, or media generation often exceed what language models can reliably accomplish through text prediction alone. Tools address each of these limitations.

3.1.1 Why do we need LLM tools?

Information-augmenting tools bridge the temporal gap by injecting up-to-date knowledge into the agent's context. These include web search engines, domain-specific retrieval systems like arXiv or custom databases, real-time APIs for weather or stock prices, and internal knowledge systems that query company repositories. As illustrated in figure 3.2, these tools deliver Tool Results that extend the LLM's awareness beyond its training cutoff.

Action-executing tools overcome the interaction limitation by enabling agents to affect the real world. Examples include booking systems for hotels and flights, communication tools for emails and calendar management, automation tools for document generation, and IoT controllers for smart devices. These tools trigger Actions and receive Observations in return, forming a feedback loop with the environment.

Domain-specialized tools address functional limitations by handling tasks where LLMs fall short. Calculators, code interpreters, format converters, image generators, and voice synthesis engines allow hybrid problem-solving—combining the LLM's reasoning with reliable execution in specialized domains.

3.1.2 Types of LLM tools

Tools are categorized as custom-built by the developer or predefined by the LLM provider. Custom tools are developed or adapted by the developer to fit a specific environment. Examples include APIs that integrate with internal company systems or scripts for specialized data analysis. Custom tools can be fully tailored to project requirements. However, this also means the developer is responsible for building, maintaining, and securing them—bugs and security issues are the developer’s responsibility.

Non-custom Tools are predefined and offered directly by the LLM provider. Examples include OpenAI’s web search feature and Anthropic’s web search tool. These tools can be used immediately without any development effort, and the provider manages stability and performance. The trade-off is that they cannot be easily modified or extended, and they can be changed or removed at the provider’s discretion. The list of available non-custom tools can be found in the official documentation of the LLM provider.

LLM providers are continuously expanding their built-in tool offerings. OpenAI provides code interpreter, file search, and web browsing capabilities through their Assistants API.

Anthropic offers computer use and web search tools. Google's Gemini includes code execution and Google Search integration. These tools are production-ready, well-maintained, and require no implementation effort. Given this trend, you might wonder: why build custom tools at all?

This book focuses on custom tools for two reasons. First, understanding how tools work internally is essential for debugging agent failures. When an agent produces unexpected results, you need to know whether the issue lies in the tool's implementation, the LLM's tool selection, or the way results are processed. Built-in tools are black boxes that obscure these details. Second, real-world agents often require specialized capabilities that providers don't offer. Querying your company's internal database, integrating with proprietary APIs, or implementing domain-specific logic all require custom tools.

That said, provider tools have their place. For rapid prototyping or when a built-in tool matches your needs, using provider tools makes sense. The skills you develop building custom tools will help you evaluate when to build versus when to use what's already available.

3.2 How LLMs use tools

How does a model that can only generate text actually select and execute a tool? The answer lies in a mechanism called Tool Calling. To better understand Tool Calling, consider this metaphor: imagine the LLM as a wise sage locked in a room with no tools. Though it has learned a vast amount from countless texts, it can only communicate by passing notes under the door. It doesn’t know what tools exist outside, how they work, or how to use them directly. That’s where we come in—we send a note explaining what tools are available, how each one functions, and what task needs to be completed.

We can think of this note as a tool definition, a structured description that tells the LLM what tools (functions, APIs, or actions) are accessible, what parameters they expect, and what they do. Once the LLM receives this information, it can begin reasoning about which tool is appropriate for a given task and how to use it, purely through text generation. After the LLM decides which tool is needed, it generates a tool call—a structured output that includes the name of the tool and the arguments required to invoke it. Then, this tool call is passed to an external executor, which actually runs the tool or API in the real world.

Now, let’s take a closer look at how a tool definition is structured, how it is provided to the model, and how the tool call generation and execution process works. You can find all the code examples for this section in our GitHub repository at https://github.com/shangrilar/ai-agent-from-scratch.

3.2.1 How tool calling works

Tool calling involves a multi-step process, as illustrated in figure 3.3. To initiate tool usage, the developer sends a message (representing user input) to the LLM, along with tool definitions, which is step 1.

One important thing to note is that the LLM does not execute the tools itself. Rather, it generates a textual specification indicating which tool should be called and with what parameters. In this sense, the LLM acts as a mediator between the user's request and the available tools. The LLM-based application or agent handles the actual execution of the tool—this corresponds to step 3 in the diagram. Once the tool is executed, the result is returned to the LLM, which interprets the output and continues the conversation. The final response to the user incorporates this result.

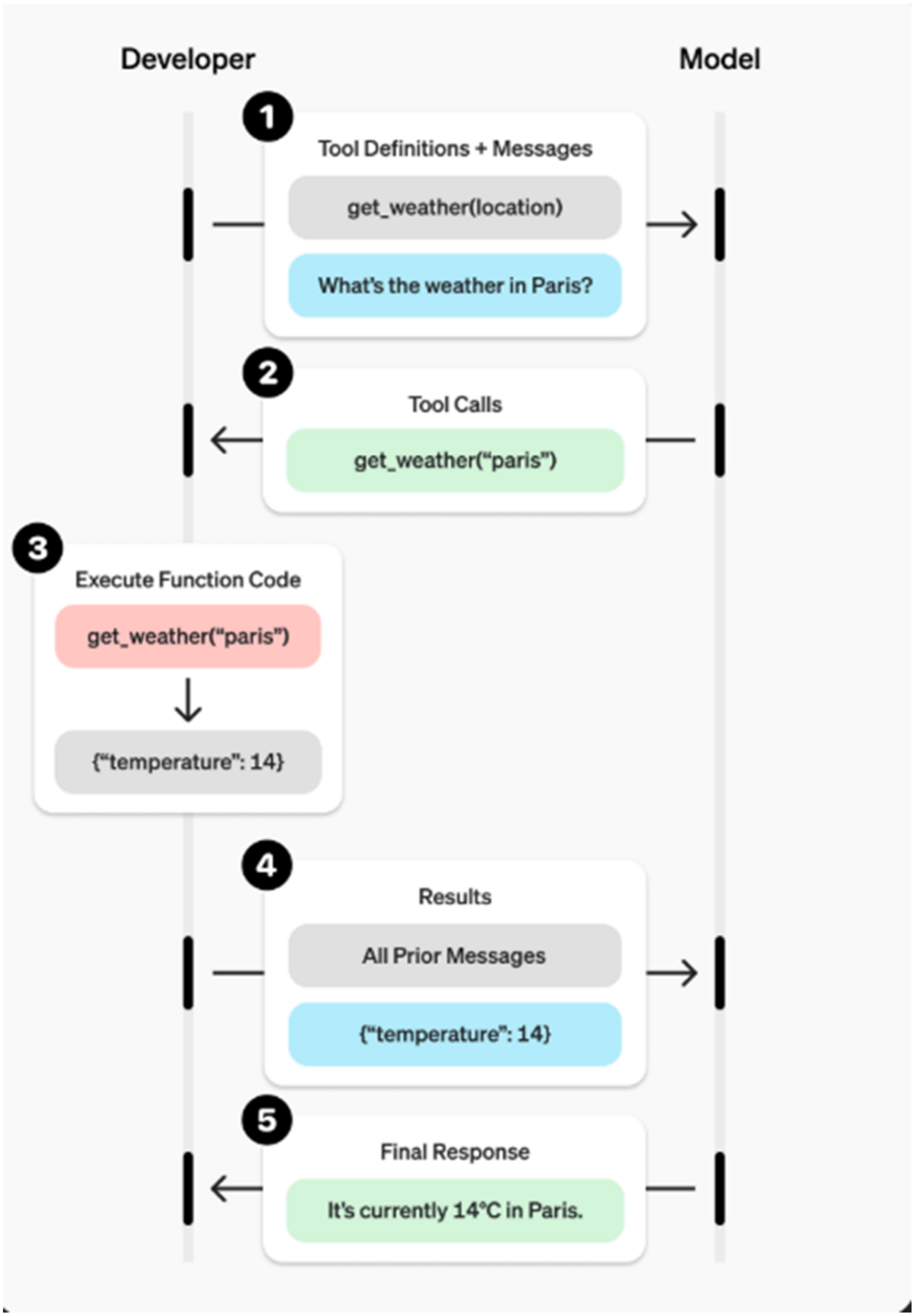

Developer
Model
1
Tool Definitions + Messages
get_weather(location)
What's the weather in Paris?
2
Tool Calls
get_weather("paris")
3
Execute Function Code
get_weather("paris")
{"temperature": 14}
4
Results
All Prior Messages
{"temperature": 14}
5
Final Response
It's currently 14°C in Paris.

Figure 3.3 Key steps in tool calling – tool definition, delivery, and LLM tool selection, source: https://platform.openai.com/docs/guides/function-calling?api-mode=responses.

Therefore, to enable tool calling, we must:

1. Define the tool definitions provided to the LLM.
2. Define the actual tools.
3. Provide the LLM with the task to perform and tool definitions from step 1.
4. Build a system that can execute the tools based on the tool calls generated by the LLM.
5. Reflect the execution results back into the LLM's context.

Now, let's take a look at how each of these steps can be implemented.

STEP 1: SPECIFY TOOL DEFINITIONS

The first step is to create a structured definition that tells the LLM what tools are available and how to use them. Think of this as an instruction manual for the LLM. Let's examine how this works by looking at a calculator tool definition from the outside in:

Starting at the outermost level, we specify `"type": "function"` to indicate this is a callable tool. Inside, we define the `function` object with three essential components:

- **name**: The tool's identifier that the LLM will use to call it ("calculator")
- **description**: When and why to use this tool ("Perform basic arithmetic operations")
- **parameters**: The specification of inputs needed to use the tool

The parameters section follows a JSON Schema format, defining each input the tool expects. For our calculator, we need three inputs: an operator (limited to add, subtract, multiply, or divide through the `enum` constraint), and two numbers. The `required` field ensures the LLM always provides all necessary inputs.

Listing 3.1 Calculator tool definition schema

```
calculator_tool_definition = {
    "type": "function",
    "function": {
        "name": "calculator",
        "description": "Perform basic arithmetic operations.",
        "parameters": {
            "type": "object",
            "properties": {
                "operator": {
                    "type": "string",
                    "description": "Arithmetic operation to perfor
m",
                    "enum": ["add", "subtract", "multiply", "divid
e"]
                },
                "first_number": {
                    "type": "number",
                    "description": "First number for the calculatio
n"
                },
                "second_number": {
                    "type": "number",
                    "description": "Second number for the calculatio
n"
                }
            },
            "required": ["operator", "first_number", "second_numbe
r"],
        }
    }
}
```

STEP 2: SET UP THE TOOL

A Python function, such as `calculator()`, can be defined to match the input schema provided earlier. Based on the function name and arguments generated by the LLM, we invoke the correct function with the appropriate parameters.

Listing 3.2 Calculator function implementation

```
def calculator(operator: str, first_number: float, second_number: fl
oat):
   if operator == 'add':
       return first_number + second_number
   elif operator == 'subtract':
       return first_number - second_number
   elif operator == 'multiply':
       return first_number * second_number
   elif operator == 'divide':
       if second_number == 0:
           raise ValueError("Cannot divide by zero")
       return first_number / second_number
   else:
       raise ValueError(f"Unsupported operator: {operator}")
```

STEP 3: EXECUTING THE TOOL CALLING

Now, the LLM receives the tool definitions and the user's question, and decides whether a tool is needed and, if so, which one. For example, consider two queries: "What's the capital of South Korea?" and "What is 1234 x 5678?" Using a tool definition that defines a calculator, the LLM responds to the first with a plain text answer, but responds to the second by generating a Tool Call, including the correct operator (`multiply`) and operands (`1234, 5678`).

Listing 3.3 Tool calling execution example

```
tools = [calculator_tool_definition]

response_without_tool = completion(
        model='gpt-5-mini',
        messages=[{"role": "user",
➥"content": "What is the capital of South Korea?"}],
        tools=tools
)
print(response_without_tool.choices[0].message.content)
print(response_without_tool.choices[0].message.tool_calls)

response_with_tool = completion(
        model='gpt-5-mini',
        messages=[{"role": "user", "content": "What is 1234 x 567
8?"}],
        tools=tools
)
print(response_with_tool.choices[0].message.content)
print(response_with_tool.choices[0].message.tool_calls)
The output is as follows:
# Capital question doesn’t need tool call
The capital of South Korea is Seoul.
None
# Multiplication question needs tool call
None
[ChatCompletionMessageFunctionToolCall(id='call_viaOEiQJ5VEB9YvKl95q
lDjM', function=Function(arguments='{"operator":"multiply","first_nu
mber":1234,"second_number":5678}', name='calculator'), type='functio
n')]
```

As you can see from the output, the model demonstrates intelligent tool selection. For the capital question, the model provides a direct answer ("The capital of South Korea is Seoul") without invoking any tools—indicated by `tool_calls` being `None`—since this information is already present in its training data. In contrast, for the multiplication question, the model recognizes it needs computational assistance and generates a tool call with the appropriate parameters, while the `content` field becomes `None`. This selective behavior shows that LLMs can discern when tools are necessary versus when they can answer directly from their knowledge base.

STEP 4: RUNNING THE TOOL

Now, the tool that the LLM has requested needs to be executed. Since the LLM itself cannot run any code, we—as the host system—must perform the execution.

The function name and arguments returned by the LLM through Tool Calling are extracted—`function_name` and `function_args`. If the selected function is the calculator, the calculator function is `executed` by passing in the arguments generated by the LLM.

Listing 3.4 Extract and execute tool from response

```
ai_message = response_with_tool.choices[0].message

if ai_message.tool_calls:
   for tool_call in ai_message.tool_calls:
       function_name = tool_call.function.name
       function_args = json.loads(tool_call.function.arguments)

       if function_name == "calculator":
           result = calculator(**function_args)
```

STEP 5: FEEDING RESULTS BACK TO THE LLM

Finally, we feed the result of the tool back into the LLM. This is done by appending a new message to the conversation with a role of `"tool"` and including the output in the content. The updated context is then passed back to the LLM, which generates the final response.

Listing 3.5 Feed tool results back to LLM

```
ai_message = response_with_tool.choices[0].message
messages.append({  #A
   "role": "assistant",  #A
   "content": ai_message.content,  #A
   "tool_calls": ai_message.tool_calls  #A
})

if ai_message.tool_calls:
   for tool_call in ai_message.tool_calls:
       function_name = tool_call.function.name  #B
       function_args = json.loads(tool_call.function.arguments)  #B

       if function_name == "calculator":
           result = calculator(**function_args)  #B

           messages.append({  #C
               "role": "tool",  #C
               "tool_call_id": tool_call.id,  #C
               "content": str(result)  #C
           })

final_response = completion(
   model='gpt-5-mini',
   messages=messages
)
print(“Messages: “, messages)
print("Final Answer:", final_response.choices[0].message.content)
```

#A Append AI's message with tool calls to conversation history
#B Extract tool call details and execute the calculator function
#C Add tool execution result to messages with tool role

The output is as follows:

```
Messages: [{'role': 'user', 'content': 'What is 1234 x 5678?'},
➥{'role': 'assistant', 'content': None, 'tool_calls':
➥[ChatCompletionMessageFunctionToolCall(id='call_TN8z8oUZ4g8hLwRfYT
YFhDGD',
➥function=Function(arguments='{"operator":"multiply",
➥"first_number":1234,"second_number":5678}', name='calculator'),
➥type='function')]},
➥{'role': 'tool', 'tool_call_id': 'call_TN8z8oUZ4g8hLwRfYTYFhDGD',
➥'content': '7006652'}]
Final Answer: 1234 × 5678 = 7,006,652
```

The execution follows a three-step feedback loop. First, the assistant's tool call request is added to the message history. Then, the calculator function is executed with the parsed arguments (multiply, 1234, 5678), returning 7006652. This result is appended as a "tool" role message with the matching `tool_call_id`. When this complete conversation history is sent back to the LLM, it interprets the raw result and formats it into a clear, human-readable answer with proper number formatting.

In our calculator example, the result from the calculator is the final answer, so feeding the result back to the LLM may seem redundant. However, for more complex tools—such as web search—it is crucial. For example, search results must be returned to the LLM so it can synthesize the final response from relevant sources. This loop of *call* → *execute* → *return* → *reason* is essential to consistent tool use.

3.2.2 How can an LLM choose tools

So, how does the LLM learn to select tools and generate the required inputs? The answer lies in how the LLM is trained on tool usage examples.

During training, LLMs are exposed to datasets containing tool definitions paired with appropriate tool calls. For example, given a request like "Please set the living room blinds to 75% openness," the training data shows the model that it should call the `set_blind_openness` function with arguments `room='living room'` and `openness_percentage=75`.

By learning from such examples, the model develops the ability to:

1. Recognize when a tool is needed based on the user's request
2. Select the appropriate tool from the available options
3. Extract the correct parameter values from the user's input
4. Format the tool call in the expected structure

This training process is similar to how LLMs learn other capabilities through pattern recognition across large amounts of data. The key difference is that tool calling requires the model to output structured data (function names and arguments) rather than free-form text.

If you're curious about what this training data looks like, you can explore datasets like NousResearch's hermes-function-calling-v1 on HuggingFace (https://huggingface.co/datasets/NousResearch/hermes-function-calling-v1), which contains examples of tool definitions and corresponding tool calls used to train models for function calling.

3.2.3 Guidelines for effective tool calling

The following best practices are adapted from OpenAI's official documentation on function calling. For LLMs to call tools reliably, a tool's schema must make when, what, and how crystal clear. Each pair of examples below illustrates a problematic schema, its corrected version, and a brief explanation of why the fix is effective.

CLEAR AND EXPLICIT FUNCTION DEFINITIONS

LLMs learn when and how to use tools based on function names and parameter descriptions. Define the tool’s purpose, parameter types, and expected outputs explicitly. A good rule of thumb: if an intern can understand and use the tool from the schema alone, so can an LLM.

Listing 3.6 Anti-pattern: vague tool definition

```
{
  "name": "book",
  "description": "Book something",
  "parameters": {
    "type": "object",
    "properties": { "when": { "type": "string" } }
  }
}
```

This schema hides the object of the action (e.g., 'book what?'), omits required fields, and leaves the time format unspecified. The model can’t infer which inputs are mandatory or how to format them.

Listing 3.7 Improved: explicit tool definition

```
{
  "name": "reserve_table",
  "description": "Create a restaurant table reservation.",
  "parameters": {
    "type": "object",
    "properties": {
      "restaurant_id": {
        "type": "string",
        "description": "Internal ID, e.g., rst_123"
      },
      "datetime": {
        "type": "string",
        "format": "date-time",
        "description": "ISO-8601 in the user's local time"
      },
      "party_size": {
        "type": "integer",
        "minimum": 1,
        "maximum": 20
      },
      "notes": {
        "type": "string",
        "description": "Allergies, occasion, etc. Optional."
      }
    }
    "required": ["restaurant_id", "datetime", "party_size"],
  }
}
```

The improved version states its intent plainly through the name and description, distinguishes between required and optional parameters, constrains formats and ranges to make inputs predictable, and defines explicit error codes that the model can surface or handle consistently.

APPLY SOFTWARE ENGINEERING BEST PRACTICES

Avoid ambiguous or logically conflicting parameters (e.g., `toggle_light(on: bool, off: bool)` allows contradictory inputs).

Replace mutually exclusive booleans with a single enum. Also, validate ranges and formats in the schema so that you can reject early with clear error codes.

Listing 3.8 Anti-pattern: conflicting parameters

```
{
  "name": "toggle_light",
  "parameters": {
    "type": "object",
    "properties": {
      "on": { "type": "boolean" },
      "off": { "type": "boolean" }
    }
  }
}
```

Here, the model can produce impossible combinations, such as `on=true` and `off=true`. That ambiguity forces extra prompt logic and increases the likelihood of call failures.

Listing 3.9 Improved: single enum parameter

```
{
  "name": "toggle_light",
  "description": "Control the light state",
  "parameters": {
    "type": "object",
    "properties": {
      "state": {
        "type": "string",
        "enum": ["on", "off"],
        "description": "The desired state of the light"
      }
    }
    "required": ["state"],
  }
}
```

This design removes contradiction by modeling state as a single enum, enables deterministic early rejection through brightness range checks, and provides concrete error codes that map cleanly to recovery strategies, such as retries, user guidance, or fallbacks.

REDUCE MODEL LOAD AND USE CODE EFFECTIVELY

Do not make the model resupply identifiers you already have. Let code remember context, and let the model focus on the decision itself.

It is inefficient to have the model regenerate information that has already been obtained. For example, if an `order_id` was retrieved in a previous step, it's more efficient to provide a parameterless function, such as `submit_refund()`, and pass the `order_id` automatically through code, rather than asking the model to call a function that requires `order_id` as an input.

Listing 3.10 Anti-pattern: model-supplied identifiers

```
{
  "name": "submit_refund",
  "parameters": {
    "type": "object",
    "required": ["order_id", "reason"],
    "properties": {
      "order_id": {
        "type": "string",
        "description": "The order ID to refund"
      },
      "reason": {
        "type": "string",
        "description": "Reason for the refund"
      }
    }
  }
}
```

This approach forces the model to recall and pass `order_id` correctly on every call, which increases error risk and token usage, especially in multi-step flows where identifiers can drift or be confused.

Listing 3.11 Improved: decision-only parameters

```
{
  "name": "submit_refund",
  "parameters": {
    "type": "object",
    "required": ["reason"],
    "properties": {
      "reason": {
        "type": "string",
      }
    }
  }
}
```

Now the model supplies only the decision input—the reason—while the calling code injects the correct `order_id` from context. This reduces the chance of refunding the wrong order, lowers prompt cost, and improves success rates across multi-turn workflows.

LIMIT THE NUMBER OF FUNCTIONS

As a general guideline, keep the number of tools under 20. Using too many tools can lead to selection errors. If your task requires many tools or complex logic, consider fine-tuning the model to improve Tool Calling precision.

3.3 Building tools and tool definitions for LLMs

Now that we understand how tool calling works, let's build a real tool from scratch. We'll implement a web search tool, one of the most essential capabilities for any research agent. Through this process, you'll experience firsthand what it takes to create a production-ready tool.

3.3.1 Implementing a web search tool

For web search, we’ll use Tavily over alternatives like Google Search API or Bing API, because it’s optimized for LLMs, delivering results that are easier to process. To use Tavily, first you need to sign up at https://www.tavily.com/. Then, retrieve your API key from https://app.tavily.com/home. Tavily allows up to 1,000 free API calls per month.

Before continuing, install the Tavily Python client:

```
uv add tavily-python
```

Then, store your API key in a `.env` file using the `TAVILY_API_KEY` variable.

```
TAVILY_API_KEY=<Your Tavily API KEY>
```

STARTING SIMPLE

Let's start with the simplest possible implementation to verify everything works. At this stage, our goal is just to confirm that we can connect to the API and retrieve results. We'll keep the configuration minimal and add complexity only when needed.

The code follows a straightforward pattern. First, we load environment variables using `load_dotenv()`, which reads our API key from the `.env` file. Then we create a `TavilyClient` instance with that key. The `search_web` function itself is minimal: it takes a query string, calls the Tavily API's `search()` method, and returns the results. We default `max_results` to 2 to keep responses concise during development and reduce API costs.

Listing 3.12 Basic web search function using Tavily

```
import os
from tavily import TavilyClient
from dotenv import load_dotenv

load_dotenv()

tavily_client = TavilyClient(os.getenv("TAVILY_API_KEY"))

def search_web(query: str, max_results: int = 2) -> list:

    response = tavily_client.search(query, max_results=max_results)
    return response.get("results")
```

Let’s try searching for Kipchoge’s marathon record to check if search_web works. Each result includes a title, URL, content snippet, and relevance score. The first result is his Wikipedia page, and the second is a BBC article. From these snippets, we can see his official marathon best time is 2:01:09 (Berlin 2022). Our basic tool works.

Listing 3.13 Testing the web search function

```
search_web("Kipchoge's marathon world record")
```

The output is as follows:

```
[
    {
        'title': 'Eliud Kipchoge - Wikipedia',
        'url': 'https://en.wikipedia.org/wiki/Eliud_Kipchoge',
        'content': 'Eliud Kipchoge is a Kenyan long-distance runner 
who
        ➥competes in the marathon and formerly specialized in the 5
000 metres.
        ➥Kipchoge is the 2016 and 2020 Olympic marathon champion, a
nd was the
        ➥world record holder in the marathon from 2018 to 2023, unt
il that
        ➥record was broken by Kelvin Kiptum at the 2023 Chicago Mar
athon.
        ➥Kipchoge has run 4 of the 10 fastest marathons in histor
y...
        ➥Personal bests Marathon: 2:01:09 (Berlin 2022)...',
        'score': 0.92125154,
        'raw_content': None
    },
    {
        'title': 'Eliud Kipchoge breaks two-hour marathon mark by 20 
seconds',
        'url': 'https://www.bbc.co.uk/sport/athletics/50025543',
        'content': 'Eliud Kipchoge has become the first athlete to r
un a
        ➥marathon in under two hours, beating the mark by 20 second
s. The
        ➥Olympic champion - who holds the official marathon world r
ecord of
        ➥2:01:39, set in Berlin, Germany in 2018 - missed out by 25 
seconds
        ➥in a previous attempt at the Italian Grand Prix circuit at 
Monza in
        ➥2017...',
        'score': 0.9014448,
        'raw_content': None
    }
]
```

ADDING SEARCH OPTIONS

Our basic function retrieves results, but real-world usage often demands more control. Consider these scenarios: a user wants only recent news articles about a topic, or they need results specifically from their country, or they want deeper research with more results. Each requirement needs additional parameters.

Tavily's API supports these through optional arguments. Let's expand our function to expose the most commonly needed options. We've added four new parameters. The `topic` parameter lets users focus on specific content types: "general" for broad web search, "news" for recent articles, or "finance" for financial information. The `time_range` parameter filters results by recency, useful when users need current information rather than historical content.

Listing 3.14 Web search function with additional options

```
def search_web(
    query: str,
    max_results: int = 2,
    topic: str = "general",
    time_range: str | None = None,
    country: str | None = None,
) -> list:

    response = tavily_client.search(
        query,
        max_results=max_results,
        topic=topic,
        time_range=time_range,
        country=country,
    )
    return response.get("results")
```

But this is just scratching the surface. Tavily's API supports nearly 20 parameters, including filtering by specific domains (including or excluding certain websites), controlling the depth of search (basic vs. advanced), including images in results, retrieving raw HTML content, and more. Each parameter requires understanding its purpose, valid values, and how it interacts with other options. The more parameters you expose, the more complex your tool definition becomes, and the harder it is for the LLM to use correctly.

HANDLING ERRORS GRACEFULLY

So far, we've assumed everything works perfectly. But what happens when things go wrong? An invalid API key returns a 401 authentication error. Exceeding your monthly usage limit triggers a 429 rate limit error. Network issues cause connection timeouts. The API might return malformed data, or the service could be temporarily unavailable.

A minimal approach catches all exceptions and returns an error message. This implementation wraps the API call in a try-except block and returns an error string if anything goes wrong. The return type is now `list | str`, indicating the function returns either a list of results or an error message string. This approach is simple and ensures the agent can continue operating even when the search fails.

Listing 3.15 Web search function with error handling

```
def search_web(
    query: str,
    max_results: int = 5,
    topic: str = "general",
    time_range: str | None = None,
) -> list | str:
    try:
        response = tavily_client.search(
            query,
            max_results=max_results,
            topic=topic,
            time_range=time_range,
        )
        return response.get("results")
    except Exception as e:
        return f"Error: Search failed - {e}"
```

However, this catch-all approach treats every error the same way. In production systems, different errors often require different responses. Authentication errors (401) might indicate a configuration problem that needs human attention. Rate limit errors (429) might benefit from automatic retry with backoff. Timeout errors might warrant a retry with a simpler query. Network errors might need different handling than API errors.

For our learning purposes, this simple error handling suffices. But as you build tools for production agents, you'll likely need to distinguish between error types and handle each appropriately. The complexity of robust error handling is yet another dimension of tool development that goes beyond simply calling an API.

3.3.2 Converting to tool definitions

So far, we’ve created a web search tool. Earlier in this chapter, we also built a `calculator` function for performing computations. To make these tools accessible to an LLM, we need to define them using a standardized tool definition format.

Since the functions we've written are simple, converting them into tool definitions manually won’t take much time. However, during development, functions may change frequently or grow more complex, making manual updates error-prone and inefficient. To address this, we can create and use a utility function that automatically converts a Python function into a tool definition.

To generate a tool definition from a function, we need to extract the function's name, docstring, and parameter details. In Python, this can be done using the `inspect` module.

To illustrate the process, let’s consider a sample function called `example_tool`. This function takes two inputs: `input_1`, which is a string, and `input_2`, which is an integer with a default value of 1.

Listing 3.16 Extract tool metadata using inspect

```
import inspect

def example_tool(input_1:str, input_2:int=1):
    """docstring for example_tool"""
    return

print(f"function name: {example_tool.__name__}")
print(f"function docstring: {example_tool.__doc__}")
print(f"function signature: {inspect.signature(example_tool)}")
```

The output is as follows:

```
function name: example_tool
function docstring: docstring for example_tool
function signature: (input_1: str, input_2: int = 1)
```

We can extract the function's name using the `__name__` attribute and its description using the `__doc__` attribute. The parameter types and whether they're required can be determined using `inspect.signature`. From the output, we can see that `input_1` is a required string, and `input_2` is an optional integer with a default value of 1.

Using this information, we can implement a utility function as follows.

Listing 3.17 Function to tool schema converter

```
def function_to_input_schema(func) -> dict:
    type_map = {
        str: "string",
        int: "integer",
        float: "number",
        bool: "boolean",
        list: "array",
        dict: "object",
        type(None): "null",
    }

    try:
        signature = inspect.signature(func)  #A
    except ValueError as e:
        raise ValueError(
            f"Failed to get signature for function {func.__name__}:
            ➥{str(e)}"
        )

    parameters = {}
    for param in signature.parameters.values():
        try:
            param_type = type_map.get(param.annotation, "string")  #
B
        except KeyError as e:
            raise KeyError(
                f"Unknown type annotation {param.annotation} for par
ameter
                ➥{param.name}: {str(e)}"
            )
        parameters[param.name] = {"type": param_type} #B

    required = [
        param.name  #C
        for param in signature.parameters.values()  #C
        if param.default == inspect._empty  #C
    ]

    return {
            "type": "object",
            "properties": parameters,
            "required": required,
        }
```

#A Extract function signature using inspect module
#B Map Python types to JSON Schema types with string as default
#C Parameters without default values are marked as required

Looking at the example code, we can see how `inspect.signature` is used to extract function parameters and convert their types into the appropriate format for a tool definition. It checks whether each parameter has a default value to determine if it's required and stores the required arguments in a list. Finally, it organizes all the information according to the standard tool definition format.

To verify that the `function_to_tool_definition` utility works correctly, we can try converting our web search tool.

Listing 3.18 Function to tool schema converter

```
def format_tool_definition(name: str, description: str, parameters:
dict) -> dict:
    return {
        "type": "function",
        "function": {
            "name": name,
            "description": description,
            "parameters": parameters,
        },
    }

def function_to_tool_definition(func) -> dict:
    return format_tool_definition(
        func.__name__,
        func.__doc__ or "",
        function_to_input_schema(func)
    )

function_to_tool_definition(search_web)
# {'type': 'function', 'function': {'name': 'search_web',
# ➥'description': 'Search the web for the given query.',
# ➥'parameters': {'type': 'object', 'properties':
# ➥{'query': {'type': 'string'}}, 'required': ['query']}}}
```

The resulting schema should include the function name and description, along with parameter types and whether each is required—just as expected.

```
search_tool_definition = function_to_tool_definition(search_web) pri
nt(search_tool_definition)
```

The output is as follows.

```
{'type': 'function', 'function': {'name': 'search_web', 'descriptio
n': 'Search the web for the given query.', 'parameters': {'type': 'o
bject', 'properties': {'query': {'type': 'string'}, 'max_results':
{'type': 'integer'}, 'topic': {'type': 'string'}, 'time_range': {'ty
pe': 'string'}}, 'required': ['query']}}}
```

Now we can convert all the tools we've created so far into schemas using this utility and prepare to solve our target problem.

3.3.3 End-to-end tool execution

Now let's put everything together. We have a working web search function and a utility to convert it into a tool definition. The final piece is building the execution infrastructure that connects the LLM's tool calls to actual function execution.

We need two components: a function to execute tools based on LLM output, and a control loop that manages the conversation between the LLM and tools.

BUILDING THE TOOL EXECUTION SYSTEM

First, we define a utility function that executes a tool and returns its result. This function takes a toolbox (a dictionary mapping tool names to functions) and a tool call from the LLM, then executes the appropriate function with the provided arguments. The function extracts the tool name and arguments from the LLM's response, looks up the corresponding function in the toolbox, and calls it with the parsed arguments. This simple pattern works for any tool as long as it's registered in the toolbox.

Listing 3.19 Tool execution utility function

```
def tool_execution(tool_box, tool_call):
    function_name = tool_call.function.name
    function_args = json.loads(tool_call.function.arguments)

    tool_result = tool_box[function_name](**function_args)
    return tool_result
```

Next, we build a control loop to manage the interaction between the LLM and tools. This function sends the user's question to the LLM along with available tool definitions. If the LLM requests a tool call, we execute it and feed the result back into the context. This cycle continues until the LLM returns a final response without requesting any tools.

Let's trace through what happens in this loop:

1. The LLM receives the system prompt, user question, and available tool definitions
2. If the LLM determines it needs external information, it generates a tool call
3. We append the assistant's message (containing the tool call) to the conversation history
4. We execute the requested tool and append the result as a "tool" message
5. The loop continues, sending the updated conversation back to the LLM
6. When the LLM has enough information to answer, it returns a response without tool calls, and we exit the loop

Listing 3.20 Simple agent loop for tool execution

```
from litellm import completion

def simple_agent_loop(system_prompt, question):
    tools = [search_web]
    tool_box = {tool.__name__: tool for tool in tools}
    tool_definitions = [function_to_tool_definition(tool) for tool i
n tools]

    messages = [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": question}
    ]

    while True:
        response = completion(
            model="gpt-5-mini",
            messages=messages,
            tools=tool_definitions
        )

        assistant_message = response.choices[0].message

        if assistant_message.tool_calls:
            messages.append(assistant_message)
            for tool_call in assistant_message.tool_calls:
                tool_result = tool_execution(tool_box, tool_call)
                messages.append({
                    "role": "tool",
                    "content": str(tool_result),
                    "tool_call_id": tool_call.id
                })
        else:
            return assistant_message.content
```

TESTING THE AGENT LOOP

Let's test our system with a question that requires current information.

Listing 3.21 Testing the agent loop

```
system_prompt = """You are a helpful assistant.
Use the search tool when you need current information."""

result = simple_agent_loop(
    system_prompt,
    "Who won the 2025 Nobel Prize in Physics?"
)
print(result)
```

The output is as follows.

```
The 2025 Nobel Prize in Physics was awarded jointly to John Clarke,
Michel H. Devoret, and John M. Martinis. They were recognized for ex
periments demonstrating macroscopic quantum tunnelling and energy qu
antisation in electric circuits — work underpinning superconducting
quantum technologies (announced Oct. 7, 2025)
```

The LLM recognizes it needs current information that isn't in its training data, calls the `search_web` tool with an appropriate query, receives the search results, and synthesizes a final answer. Without the tool, the LLM would have to admit it doesn't know or risk providing outdated information.

3.3.4 The challenges of custom tools

Building the web search tool in section 3.3.1 was straightforward. We installed the Tavily client, wrote a simple function, and converted it to a tool definition. The entire process took just a few minutes. However, this simplicity is deceptive. As you develop more tools and scale to real-world projects, several challenges emerge that make custom tool development surprisingly difficult.

UNDERSTANDING UNFAMILIAR APIS

Tavily was designed specifically for LLM applications, which made integration easy. The API is simple, the response format is already optimized for RAG, and the documentation is concise. But most services you want to integrate are not built with LLMs in mind.

Consider building a tool that searches your company's Slack workspace, retrieves files from Google Drive, or queries a Salesforce database. Each service has its own API conventions, rate limiting policies, pagination patterns, and response formats. Before writing any code, you need to spend hours reading documentation, understanding the data models, and figuring out how to transform responses into something useful for an LLM. This upfront investment multiplies with every new tool you need to build.

HANDLING ERRORS ROBUSTLY

Look back at our `search_web` function. It has minimal error handling. In a production environment, this would be inadequate. Real-world tools must handle network timeouts gracefully, manage API rate limits with appropriate backoff strategies, deal with service outages, validate inputs before making requests, and provide meaningful error messages that help the LLM (and ultimately the user) understand what went wrong.

Implementing robust error handling often requires more code than the core functionality itself. A simple web search function might be 10 lines, but a production-ready version with proper error handling, retries, and logging could easily be 50 lines or more.

DUPLICATED EFFORTS ACROSS TEAMS

In any organization building AI agents, multiple teams often need similar capabilities. The marketing team wants a web search tool for their content generation agent. The customer support team needs a web search for their FAQ bot. The research team requires it for their analysis pipeline.

Without a standard way to share tools, each team builds its own implementation. They solve the same problems independently, make similar mistakes, and maintain separate codebases. From an organizational perspective, this duplication wastes significant engineering resources. The same tool might exist in three or four slightly different versions across the company.

CODE REUSE IS HARDER THAN IT LOOKS

You might think the solution is simple: one team builds the tool, others reuse it. In practice, this rarely works smoothly. Different teams structure their tool definitions differently. Some use dictionaries, others use Pydantic models. Execution patterns vary between synchronous and asynchronous implementations. Return formats are inconsistent. Documentation quality differs dramatically.

When you try to adopt a tool from another team, you often spend more time adapting it to your codebase than you would have spent building it from scratch. Dependencies conflict, interfaces do not match, and assumptions embedded in the code do not apply to your use case. The friction of integration often outweighs the benefit of reuse.

ONGOING MAINTENANCE BURDEN

Building a tool is not a one-time effort. APIs evolve, and your tools must evolve with them. Services deprecate endpoints, change response formats, and introduce new required parameters. Security vulnerabilities in dependencies require updates. Users request new features and report edge cases you did not anticipate.

When you maintain one or two tools, this is manageable. But agents typically need many tools: web search, file operations, database queries, email, calendar, and domain-specific integrations. Each tool becomes a maintenance responsibility. As your tool collection grows, the cumulative maintenance burden becomes substantial.

THE NEED FOR STANDARDIZATION

These challenges are not unique to LLM tools. The software industry has solved similar problems before through standardization. REST APIs standardized how web services communicate. Package managers standardized how code is shared and versioned. Containerization standardized how applications are deployed.

The LLM tool ecosystem needs similar standardization: a common way to define tools, share them across projects, and integrate them regardless of which LLM provider you use. This is exactly what MCP, the Model Context Protocol, provides.

3.4 MCP: Standardizing tools

As we experienced firsthand in section 3.3, developing tools that LLM agents can use is time-consuming, and inconsistencies in implementation make them difficult to reuse.

In November 2024, Anthropic introduced MCP (Model Context Protocol) to bring a similar transformation to the world of LLM tool integration. If APIs established a common language for developers to communicate with services, MCP establishes a common language for LLM agents to communicate with tools. Just as you can call any REST API using the same HTTP patterns, an MCP-compatible agent can use any MCP-compatible tool without custom integration code.

Since its release, MCP has gained growing attention throughout early 2025. It opened the door for developers to use tools built by companies or communities without having to build them all from scratch. As shown in figure 3.4, with MCP, agents can easily integrate tools from services like Google Drive, Slack, Tavily, Brave Search, GitHub, and Figma without needing to write custom tool wrappers.

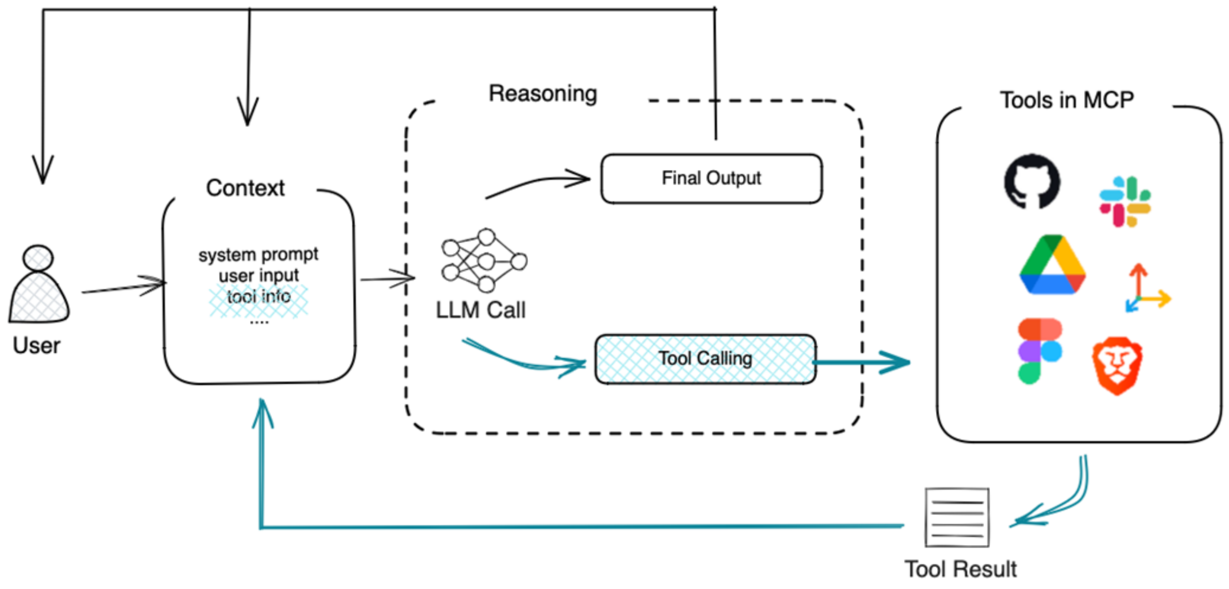


Figure 3.4 MCP makes it easy to extend an agent’s toolset by integrating third-party tools.

3.4.1 The core of MCP: server–client architecture

At its core, MCP separates the "brain" of the agent from its "arms and legs" (tools) through a three-component architecture: Host, Client, and Server.

UNDERSTANDING THE THREE COMPONENTS

Figure 3.5 illustrates how these components work together. Let's clarify each role:

The **Host** is the application that interacts with users and makes decisions using an LLM. It receives user input, performs reasoning, and decides which tools to use and when. Think of it as the "brain" of the system. Examples include Claude Desktop, an IDE with AI features, or the agent application you're building.

The **MCP Client** acts as a bridge between the Host and MCP Servers. When the Host decides it needs a tool, the Client handles the communication: requesting available tool definitions from servers and forwarding tool execution requests. Each Client maintains a connection to one MCP Server.

The **MCP Server** hosts the actual tools and executes them on demand. It responds to two types of requests: "what tools do you have?" (returning tool definitions) and "execute this tool with these arguments" (running the tool and returning results). Servers can connect to external services like databases, APIs.

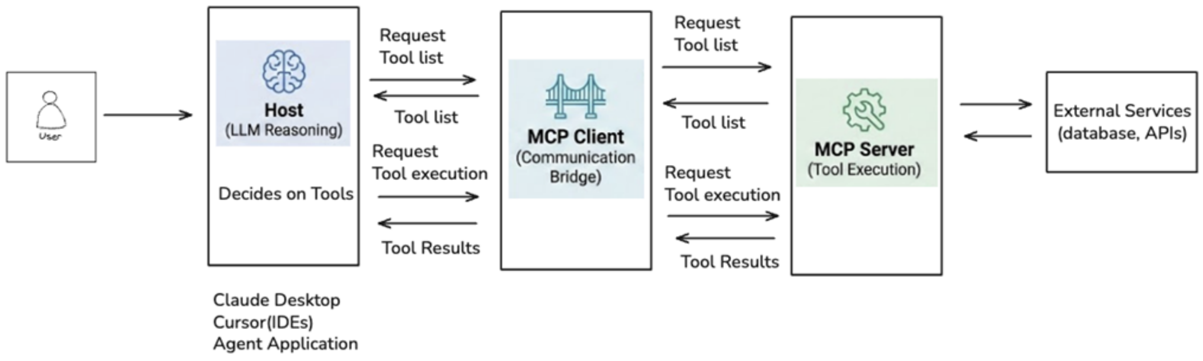


Figure 3.5 MCP server-client architecture. The Host contains the LLM for reasoning and user interaction, connecting through MCP Clients that manage server communication. MCP Servers expose tool definitions and handle execution, optionally integrating with external services like databases and APIs.

MCP'S SCOPE BEYOND TOOLS

MCP actually defines three types of capabilities that servers can provide: tools (functions the LLM can call), prompts (reusable prompt templates), and resources (data the LLM can access). In this book, we focus exclusively on tools since they're the most essential capability for building agents.

TWO STANDARDIZED INTERFACES

Regardless of which transport you use, MCP standardizes two key interfaces:

Tool Discovery: In section 3.3, we manually defined schemas for each tool. With MCP, the client can request available tool definitions, and the server returns them in a standardized format. Tool developers implement the schema once; any MCP-compatible client can automatically retrieve and use it.

Tool Execution: Previously, we built our own `tool_execution` function to bridge LLM output and actual function calls. MCP standardizes this too. The client sends a tool execution request to the server, which handles execution and returns the result in a consistent format.

TRANSPORT MECHANISMS

MCP supports three transport mechanisms for communication between clients and servers:

stdio (Standard I/O) is the simplest approach, where the client launches the server as a subprocess and communicates through standard input/output streams. Since everything runs locally on the same machine, there's no network overhead. This is ideal for local development, and when tools don't need to be shared across machines.

HTTP enables remote communication over the network. The client sends requests via HTTP, and the server streams responses back. This works well when servers need to run on separate machines or be shared across multiple clients.

WebSocket provides full bidirectional communication, allowing both client and server to initiate messages. This is useful when servers need to push updates to clients proactively.

Throughout this chapter, we'll use the stdio transport since it's the simplest to set up and sufficient for learning MCP concepts. When you need remote or shared tool servers in production, you can switch to HTTP or WebSocket without changing your tool implementations.

3.4.2 Hands-on: Running an MCP server

Before diving into implementation details, let's experience MCP firsthand by running an existing server. We'll use the official Tavily MCP server, which provides the same web search capability we built manually in section 3.3, but packaged as a ready-to-use MCP server.

MCP servers in the ecosystem are typically distributed as npm packages and run using `npx`. If you don't have Node.js installed, you'll need to install it first. For both macOS and Windows, simply download the installer for your operating system from https://nodejs.org/ and follow the installation steps.

Verify your installation by checking the version.

```
node --version
```

If the Node version is displayed, the installation was successful.

RUNNING THE TAVILY MCP SERVER

With Node.js installed, we can launch the Tavily MCP server using a single command. First, set your Tavily API key as an environment variable.

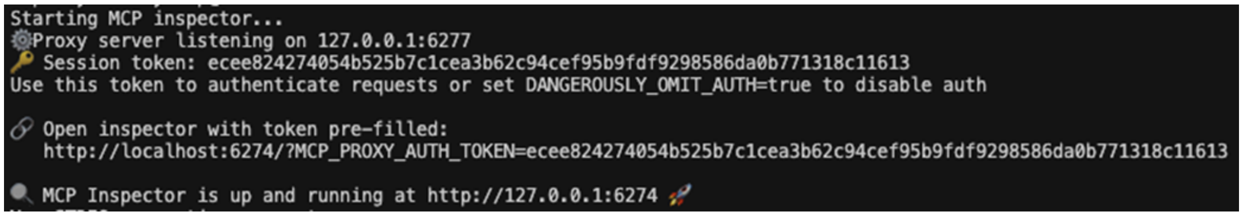
```
export TAVILY_API_KEY=<your tavily api key>
```

Now launch the server with the MCP Inspector, a browser-based tool for testing MCP servers.

```
npx @modelcontextprotocol/inspector npx -y tavily-mcp@latest
```

This command does several things at once. The `npx` command runs npm packages without installing them globally. The `@modelcontextprotocol/inspector` package provides a web interface for interacting with MCP servers.

The `-y` flag automatically confirms any prompts. Finally, `tavily-mcp@latest` is the official Tavily MCP server package.



```
Starting MCP inspector...
⚙️Proxy server listening on 127.0.0.1:6277
🔑 Session token: ecee824274054b525b7c1cea3b62c94cef95b9fdf9298586da0b771318c11613
Use this token to authenticate requests or set DANGEROUSLY_OMIT_AUTH=true to disable auth

🔗 Open inspector with token pre-filled:
   http://localhost:6274/?MCP_PROXY_AUTH_TOKEN=ecee824274054b525b7c1cea3b62c94cef95b9fdf9298586da0b771318c11613

🔍 MCP Inspector is up and running at http://127.0.0.1:6274 🚀
```

Figure 3.6 MCP Inspector startup log showing the server initialization and connection URL.

Open the URL shown in your terminal. If the URL includes a session token, you can connect without additional configuration.

EXPLORING THE SERVER WITH MCP INSPECTOR

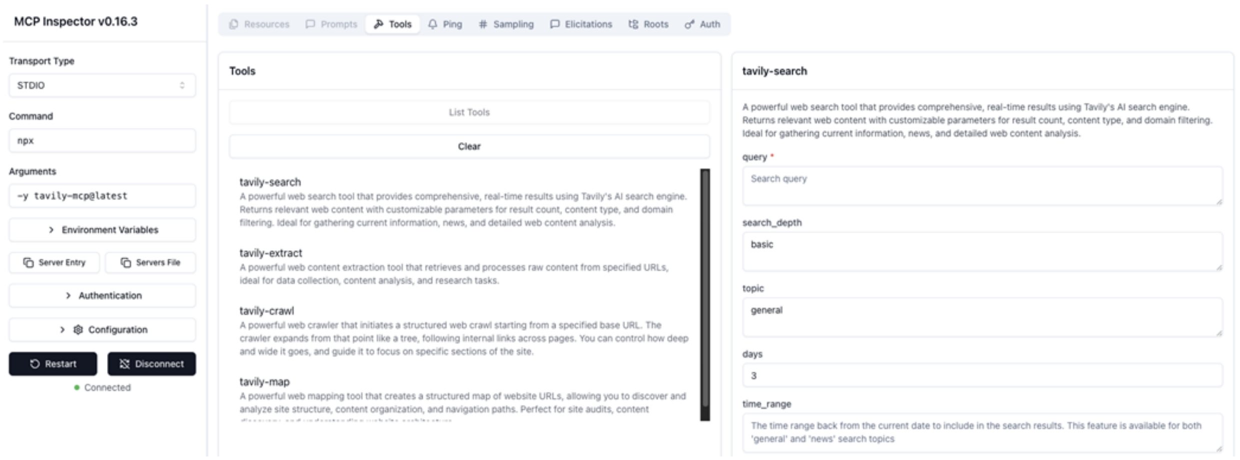
Once the Inspector UI loads in your browser, click the **Connect** button. After a successful connection, you'll see several tabs appear: Resources, Prompts, and Tools. These reflect the capabilities exposed by the MCP server.

Select the **Tools** tab and click **List Tools**. You'll see the tools that the Tavily MCP server provides.

- **tavily-search**: Web search functionality (similar to what we built in Section 3.3)
- **tavily-extract**: Extract content from specific URLs
- **tavily-crawl**: Crawl websites and extract structured data
- **tavily-map**: Generate site maps

Click on **tavily-search** to expand its details. You'll see the tool's description and parameter schema, formatted in the standard MCP format. Try entering a query like "2025 Nobel

Physics" and click **Run Tool**. The server executes the search and returns results directly in your browser.



Transport Type
STDIO

Command
npx

Arguments
-y tavily-mcp@latest

Tools

tavily-search
A powerful web search tool that provides comprehensive, real-time results using Tavily's AI search engine. Returns relevant web content with customizable parameters for result count, content type, and domain filtering. Ideal for gathering current information, news, and detailed web content analysis.

tavily-extract
A powerful web content extraction tool that retrieves and processes raw content from specified URLs, ideal for data collection, content analysis, and research tasks.

tavily-crawl
A powerful web crawler that initiates a structured web crawl starting from a specified base URL. The crawler expands from that point like a tree, following internal links across pages. You can control how deep and wide it goes, and guide it to focus on specific sections of the site.

tavily-map
A powerful web mapping tool that creates a structured map of website URLs, allowing you to discover and analyze site structure, content organization, and navigation paths. Perfect for site audits, content

tavily-search

A powerful web search tool that provides comprehensive, real-time results using Tavily's AI search engine. Returns relevant web content with customizable parameters for result count, content type, and domain filtering. Ideal for gathering current information, news, and detailed web content analysis.

query *
Search query

search_depth
basic

topic
general

days
3

time_range
The time range back from the current date to include in the search results. This feature is available for both 'general' and 'news' search topics

Figure 3.7 MCP Inspector interface showing the official Tavily MCP server, which provides a richer set of tools, including `tavily-search`, `tavily-extract`, `tavily-crawl`, and `tavily-map`.

WHAT WE JUST ACCOMPLISHED

Take a moment to appreciate what happened. With a single command, we:

1. Downloaded and ran a production-ready MCP server
2. Connected to it using a standard client interface
3. Discovered available tools automatically
4. Executed a tool and received results

Compare this to Section 3.3, where we spent considerable effort implementing just the basic search functionality. The Tavily MCP server includes multiple tools, robust error handling, and optimized response formatting, all ready to use without writing any code.

This is the power of standardization. Tool developers can focus on building reliable, feature-rich servers. Agent developers can focus on reasoning and orchestration, knowing that any MCP-compatible tool will work with their system.

In the next section, we'll look at the other side of this architecture: how to build an MCP client that can discover and use these tools programmatically within our agent.

3.4.3 Understanding the MCP client

In the previous section, we interacted with the Tavily MCP server through the Inspector's web interface. While helpful for testing, our agent needs to interact with MCP servers programmatically. This is where the MCP client comes in.

The MCP client is what your agent uses to discover and invoke tools from MCP servers. It handles the protocol details: establishing connections, requesting tool definitions, executing tools, and parsing results. Understanding how the client works is essential for integrating MCP tools into your own agents.

Install the MCP Python SDK for hands-on practice.

```
uv add mcp
```

CONNECTING TO AN MCP SERVER

Let's write a simple client that connects to the Tavily MCP server, lists available tools, and performs a search.

Let's trace through what happens in this code:

1. **StdioServerParameters** specifies how to launch the server. The command is `npx`, which runs npm packages without installing them globally. The args include `-y` to automatically confirm prompts and `tavily-mcp@latest` to

use the latest version. Environment variables pass the API key and system PATH.

2. **stdio_client** launches the server as a subprocess. The `async with` ensures proper cleanup when we're done. It returns read and write streams that the session uses for communication.
3. **ClientSession** provides the high-level API. After calling `initialize()`, we can interact with the server using standard methods like `list_tools()` and `call_tool()`.
4. **session.list_tools()** requests all available tools from the server. The server returns tool definitions including names, descriptions, and parameter schemas.
5. **session.call_tool()** executes a specific tool. We pass the tool name and a dictionary of arguments. The server runs the tool and returns the result.

Listing 3.22 Connecting to an MCP server

```
import asyncio
import os
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client
from dotenv import load_dotenv

load_dotenv()

server_params = StdioServerParameters(
    command="npx",
    args=["-y", "tavily-mcp@latest"],
    env={
        "TAVILY_API_KEY": os.getenv("TAVILY_API_KEY"),
    }
)

async with stdio_client(server_params) as (read_stream, write_strea
m):
    async with ClientSession(read_stream, write_stream) as session:
        await session.initialize()

        # List available tools
        tools_result = await session.list_tools()
        print("Available tools:")
        for tool in tools_result.tools:
            print(f"  - {tool.name}: {tool.description[:60]}...")

        result = await session.call_tool(
                "tavily-search",
                arguments={"query": "2025 Nobel Physics"}
            )
        print("Search Result:")
        print(result.content)
```

Running this code produces output similar to

```
Available tools:
  - tavily-search: A powerful web search tool that provides comprehe
nsive, real...
  - tavily-extract: A powerful web content extraction tool that retr
ieves and pr...
  - tavily-crawl: A powerful web crawler that initiates a structured
web crawl...
  - tavily-map: A powerful web mapping tool that creates a structure
d map of...
Search Result:
[TextContent(type='text', text='Detailed Results:\n\nTitle: Press re
lease: Nobel Prize in Physics 2025...
```

CONVERTING MCP TOOLS FOR LLM USE

The tools returned by `list_tools()` use MCP's schema format. To use these tools with LLMs, we need to convert them to whatever format the LLM provider expects.

Fortunately, we already built the foundation for this in Section 3.3.2. The `format_tool_definition` function we created takes a name, description, and parameters dictionary, and returns a properly structured OpenAI tool definition. MCP tool objects provide exactly these three pieces of information through their `name`, `description`, and `inputSchema` attributes. This makes the conversion straightforward.

Listing 3.23 Converting MCP tools to OpenAI format

```python
def mcp_tools_to_openai_format(mcp_tools) -> list[dict]:
    """Convert MCP tool definitions to OpenAI tool format."""
    return [
        format_tool_definition(
            name=tool.name,
            description=tool.description,
            parameters=tool.inputSchema,
        )
        for tool in mcp_tools.tools
    ]
```

With this conversion function, we can retrieve tools from any MCP server and immediately use them with OpenAI's API. Here's how it works in practice.

Listing 3.24 Retrieving tools in OpenAI format

```
async with stdio_client(server_params) as (read_stream, write_strea
m):
    async with ClientSession(read_stream, write_stream) as session:
        await session.initialize()

        # List available tools
        tools_result = await session.list_tools()
        openai_format_tools = mcp_tools_to_openai_format(tools_resul
t)
        for tool in openai_format_tools:
            print(tool)
```

3.4.4 Hands-on: Implementing an MCP server

So far, we've used external MCP servers built by others. While you can build effective agents using only community-provided servers, understanding how to create your own server gives you the confidence to navigate the MCP ecosystem. You'll know exactly what's happening under the hood, making it easier to debug issues, extend existing servers, or build custom tools for your specific needs.

The good news is that building an MCP server is surprisingly simple. Using the FastMCP library, you can convert an existing Python function into an MCP-compatible tool by adding just a decorator. Let's transform the `search_web` function we built in section 3.3 into a fully functional MCP server.

BUILDING THE SERVER

Let’s start by installing the required libraries.

```
uv add fastmcp
```

Now, create a new file called `tavily_mcp_server.py`. We'll take our existing `search_web` function and wrap it with the MCP infrastructure. The `FastMCP` instance creates our server with a name that identifies it to clients. The `@mcp.tool()` decorator registers our function as an MCP tool. FastMCP automatically extracts the function name, parameters, and type hints to generate the tool schema. The docstring becomes the tool's description, which helps LLMs understand when and how to use the tool. Finally, `mcp.run(transport='stdio')` starts the server using standard input/output for communication.

Listing 3.25 Custom Tavily MCP server implementation

```
import os
from tavily import TavilyClient
from dotenv import load_dotenv
from mcp.server.fastmcp import FastMCP

load_dotenv()

tavily_client = TavilyClient(os.getenv("TAVILY_API_KEY"))

mcp = FastMCP("custom-tavily-search")

@mcp.tool()
def search_web(query: str, max_results: int = 5) -> str:
    """
    Search the web using Tavily API.

    Args:
        query: Search query string
        max_results: Maximum number of results to return (default:
5)

    Returns:
        Search results as formatted string
    """
    try:
        response = tavily_client.search(
            query,
            max_results=max_results,
        )
        results = response.get("results", [])
        return "\n\n".join(
            f"Title: {r['title']}\nURL: {r['url']}\nContent: {r['con
tent']}"
            for r in results
        )
    except Exception as e:
        return f"Error searching web: {str(e)}"

if __name__ == "__main__":
    mcp.run(transport='stdio')
```

Notice how little code we added to our original function. The core logic remains unchanged; we simply wrapped it with the MCP infrastructure.

CONNECTING WITH THE CLIENT

Now let's verify our server works by connecting to it with the client code we learned in section 3.4.3. The only change we need is updating `StdioServerParameters` to point to our custom server instead of the official Tavily package.

Listing 3.26 Connecting to the custom MCP server

```
server_params = StdioServerParameters(
    command="uv",
    args=["run", "tavily_mcp_server.py"],
    env={
        "TAVILY_API_KEY": os.getenv("TAVILY_API_KEY"),
    }
)

async with stdio_client(server_params) as (read_stream, write_strea
m):
    async with ClientSession(read_stream, write_stream) as session:
        await session.initialize()

        # List available tools
        tools_result = await session.list_tools()
        print("Available tools:")
        for tool in tools_result.tools:
            print(f"  - {tool.name}: {tool.description}")
```

Running this code produces output like

```
Available tools:
  - search_web:
Search the web using Tavily API.

Args:
    query: Search query string
    max_results: Maximum number of results to return (default: 5)

Returns:
    Search results as formatted string
```

Compare this to Section 3.4.2, where the official Tavily server exposed four tools (tavily-search, tavily-extract, tavily-crawl, tavily-map). Our custom server exposes just one tool: `search_web`. Yet the client code is nearly identical. This is the power of standardization: whether you're connecting to a sophisticated official server or a simple custom one, the client interface remains the same.

WHAT WE ACCOMPLISHED

In this section, we completed the full MCP cycle:

1. We converted our custom `search_web` function into an MCP server with minimal changes
2. We connected to our server using the same client patterns we used for the official Tavily server
3. We verified that tool discovery and execution work correctly

This demonstrates the core value proposition of MCP: tool implementation and tool usage are completely decoupled. Server developers focus on building reliable tools. Agent developers focus on reasoning and orchestration. The protocol handles everything in between.

With tools and MCP understood, we have all the building blocks needed for a complete agent. In chapter 4, we'll combine these capabilities into a robust agent framework, implementing proper tool abstraction, error handling, and the iterative reasoning loop that transforms an LLM with tools into a true agent.

3.5 Summary

- Tools extend LLM capabilities, enabling them to access APIs, perform calculations, and retrieve external information. Tools fall into three categories: information-augmenting, action-executing, and domain-specialized.
- Tool calling is the mechanism which LLMs generate structured outputs specifying which tool to use and with what parameters. The LLM acts as a mediator between user requests and available tools, but does not execute tools itself.
- Tool execution is handled externally. The LLM generates specifications that a separate system executes, with results fed back into the LLM's context for further reasoning.
- Tool definitions are structured schemas that describe available tools, their parameters, and expected outputs. Clear and explicit definitions are essential for reliable tool selection by the LLM.
- Building custom tools involves implementing the tool function, creating tool definitions, and building execution infrastructure. While straightforward for simple cases, custom tools create a maintenance burden and inconsistency as projects scale.
- MCP (Model Context Protocol) standardizes tool development through a server-client architecture. Servers host and execute tools, while clients discover

and invoke them. This separation enables tool reuse across projects and providers.

4 Implementing a basic ReAct agent

This chapter covers

- ReAct, the core pattern behind modern AI agents
- Managing agent state with ExecutionContext
- Unifying local functions and MCP tools
- Building the LLM communication layer
- Implementing a ToolCallingAgent with the think-act loop
- Adding structured output with Pydantic models
- Testing agent performance on the GAIA benchmark

We’ve explored how LLMs work and given them the ability to use tools. Now it's time to combine these pieces into something more powerful: an agent that can reason about what it needs, take actions to get it, and continue until it solves the problem.

As shown in figure 4.1, we’ll now complete the foundation of the basic agent loop by building an Agent using the ReAct pattern, chaining multi-step reasoning and actions to solve problems end-to-end, and adding robust error handling. ReAct (Reasoning + Acting) is not a framework but a way of designing agents that mirrors how humans solve problems: assess the situation, decide what information or action is needed, execute that action, observe the results, and repeat

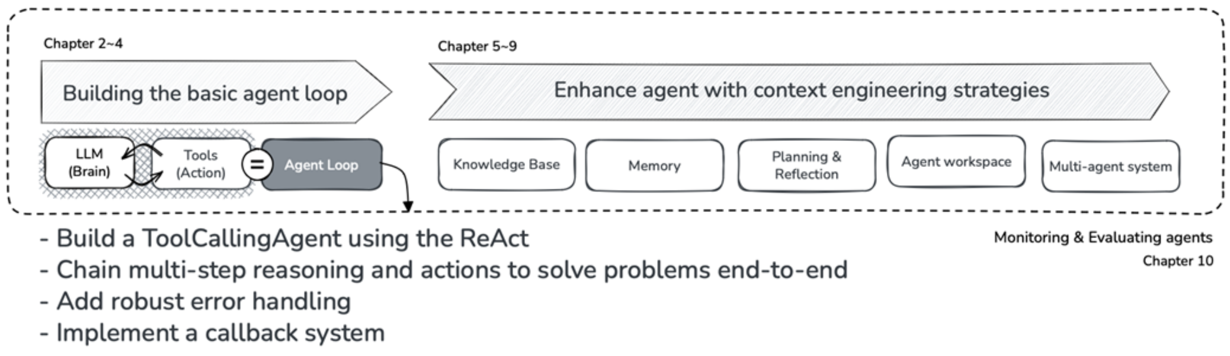

Chapter 2~4
Building the basic agent loop
LLM (Brain)
Tools (Action)
=
Agent Loop
Chapter 5~9
Enhance agent with context engineering strategies
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Build a ToolCallingAgent using the ReAct
- Chain multi-step reasoning and actions to solve problems end-to-end
- Add robust error handling
- Implement a callback system

Figure 4.1 Book structure overview - Chapter 4 in focus.

By the end of this chapter, you'll have a working Agent that can search the web for real-time information, chain multiple steps of reasoning to solve problems, handle errors gracefully, and return structured output in any format you specify. We'll test it on the GAIA benchmark. This agent will serve as the foundation for everything that follows: memory systems, planning strategies, knowledge retrieval, and multi-agent collaboration.

4.1 How ReAct agents work

Think about how you would answer this question: "If marathon runner Eliud Kipchoge could maintain his world record pace indefinitely, how long would it take him to reach the Moon?"

You wouldn't try to answer immediately. Instead, you'd think: "I need Kipchoge's marathon pace and the Earth-Moon distance." Then you'd search for that information, calculate the result, and formulate your answer. This natural cycle of thinking and acting is exactly how ReAct agents operate.

ReAct, which stands for Reasoning plus Acting, is the foundational pattern behind most modern AI agents. The core idea is simple: instead of trying to answer everything at once, an agent alternates between reasoning about what it needs and taking actions to get it.

4.1.1 The think-act cycle

Let's see how a ReAct agent would tackle the Kipchoge problem:

```
Thought: I need to find Kipchoge's marathon world record pace.
Action: search_web("Kipchoge marathon world record")
Observation: Kipchoge's record is 2:01:09 for 42.195 km (Berlin 202
2)

Thought: Now I need the Earth-Moon distance at closest approach.
Action: search_wikipedia("Moon perigee distance")
Observation: The Moon's perigee (closest approach) is 356,500 km

Thought: I can now calculate the time needed.
Action: calculator("356500 / (42.195 / 2.0186) * 1")
Observation: 17,034 hours

Thought: I have enough information to answer.
Final Answer: Approximately 17,000 hours
```

Notice the pattern. At each step, the agent explicitly states what it's thinking, takes a concrete action, and observes the result. This cycle repeats until the agent has gathered enough information to provide a final answer.

What makes this approach powerful is its adaptability. If the first search returned unhelpful results, the agent could try a different query. If the calculation seemed wrong, it could verify by searching for additional information. Unlike a fixed script that breaks when something unexpected happens, a ReAct agent adjusts its approach based on what it discovers.

This mirrors how humans naturally solve problems. We don't execute a predetermined sequence of steps. We think, act, observe the results, and decide what to do next. The LLM handles the reasoning, while tools handle the actions. Together, they form a system that can tackle problems neither could solve alone.

4.1.2 From text parsing to tool calling

Early ReAct implementations faced a frustrating problem: they relied entirely on text parsing. The agent had to output responses in an exact format, like `Action: search["query"]`, and any deviation would break the system. A missing bracket, an extra space, or a slightly different keyword would cause parsing failures. Developers spent enormous effort crafting prompts that begged the LLM to follow precise formatting rules, yet errors remained common.

Tool Calling eliminates this fragility. Instead of parsing free-form text, the LLM generates structured outputs that specify exactly which tool to call and with what arguments. The format is guaranteed by the API, not by hoping the LLM follows instructions perfectly.

Figure 4.2 illustrates this evolution. On the left, the traditional approach requires explicit "Thought" and "Action" text that must be parsed. On the right, modern Tool Calling handles this natively.

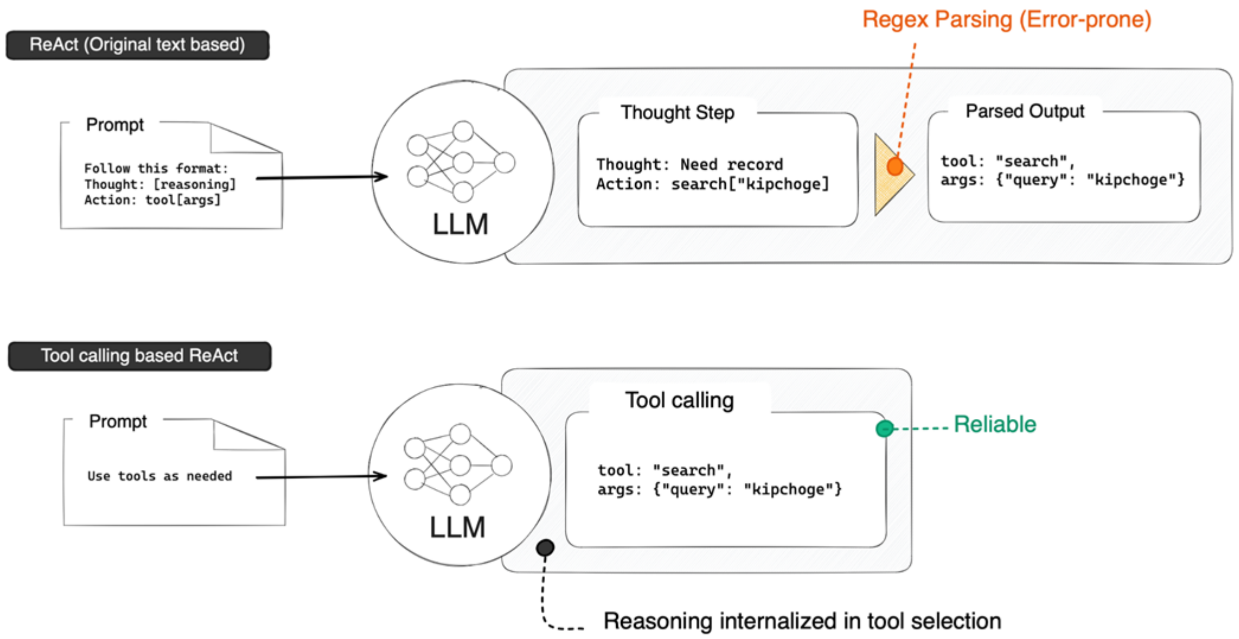


Figure 4.2 Evolution from text-based ReAct to tool-calling ReAct. The explicit "Thought" step has been internalized into the LLM's tool selection process, eliminating the need for fragile text parsing.

You might wonder: if there's no explicit "Thought" step anymore, is it still ReAct? Yes. The reasoning hasn't disappeared. It has moved inside the LLM. As we saw in chapter 3, modern LLMs are trained on extensive tool-calling examples, learning to recognize which tool is appropriate for which situation. When the LLM selects a tool and specifies its arguments, it has already performed the reasoning internally. The pattern remains the same: reason about what's needed, act to get it, observe the result, repeat.

This is why Tool Calling-based agents are still called ReAct agents across major frameworks like LangGraph, Google ADK, and HuggingFace SmolAgents. The core principles remain intact: adaptive problem-solving through iterative reasoning and action, with the LLM dynamically selecting tools based on context rather than following a predetermined script.

Now we're ready to build our own agent. Let’s look at the complete architecture of what we're going to implement, starting with the end result and working backward to understand each component we need to build.

4.2 Agent architecture overview

Before diving into implementation details, let's see what we're building toward. Understanding the complete picture first will help you appreciate why each component exists and how they work together. The code for this chapter can be found here: https://github.com/shangrilar/ai-agent-from-scratch/tree/main/chapter_04_basic_agent.

4.2.1 The completed agent

Here's how our finished agent will look in action.

Listing 4.1 Agent usage example

```
from scratch_agents import Agent, LlmClient
from scratch_agents.tools import calculator, search_web

agent = Agent(
    model=LlmClient(model="gpt-5-mini"),
    tools=[calculator, search_web],
    instructions="You are a helpful assistant"
)

result = await agent.run("What is 1234 * 5678?")
```

This simple interface hides significant complexity. The agent receives a user question, decides whether to use tools, executes them if needed, and returns a final answer. For the multiplication question above, it would call the calculator tool and return "7,006,652."

But what happens behind this clean API? To build an agent that actually works, we need to understand the information flow and the components that make it possible.

4.2.2 Information flow: The core design

An agent's job is straightforward in concept: receive user input, ask the LLM what to do next, execute tools if requested, feed results back to the LLM, and repeat until done. This cycle continues until the LLM decides it has enough information to provide a final answer.

WHY MESSAGES ALONE AREN'T ENOUGH

You might think that maintaining a simple list of messages (the conversation history with the LLM) would suffice. After all, that's what we did in chapter 2 when managing conversations. However, running an agent requires tracking much more information:

- **events**: The history of all interactions, including user messages, LLM responses, and tool execution results
- **current_step**: A counter to prevent infinite loops (what if the agent keeps calling tools forever?)
- **execution_id**: A unique identifier for debugging and logging (which execution produced this error?)
- **state**: A scratchpad for dynamic data during execution, such as task progress, intermediate results, or user preferences that tools might need
- **final_result**: A flag to know when execution is complete

If we scatter this information across different variables and pass them individually to each method, the code becomes unwieldy. Every method would need a long parameter list, and adding new information (like memory in chapter 6) would require modifying every method signature.

The solution is to consolidate everything into a single container. All methods receive this one container, and when new information is needed, we only modify the container definition. This container is **ExecutionContext**, the central storage for all execution states.

THE EXECUTIONCONTEXT-CENTERED DESIGN

Figure 4.3 illustrates how information flows through our agent, with ExecutionContext at the center.

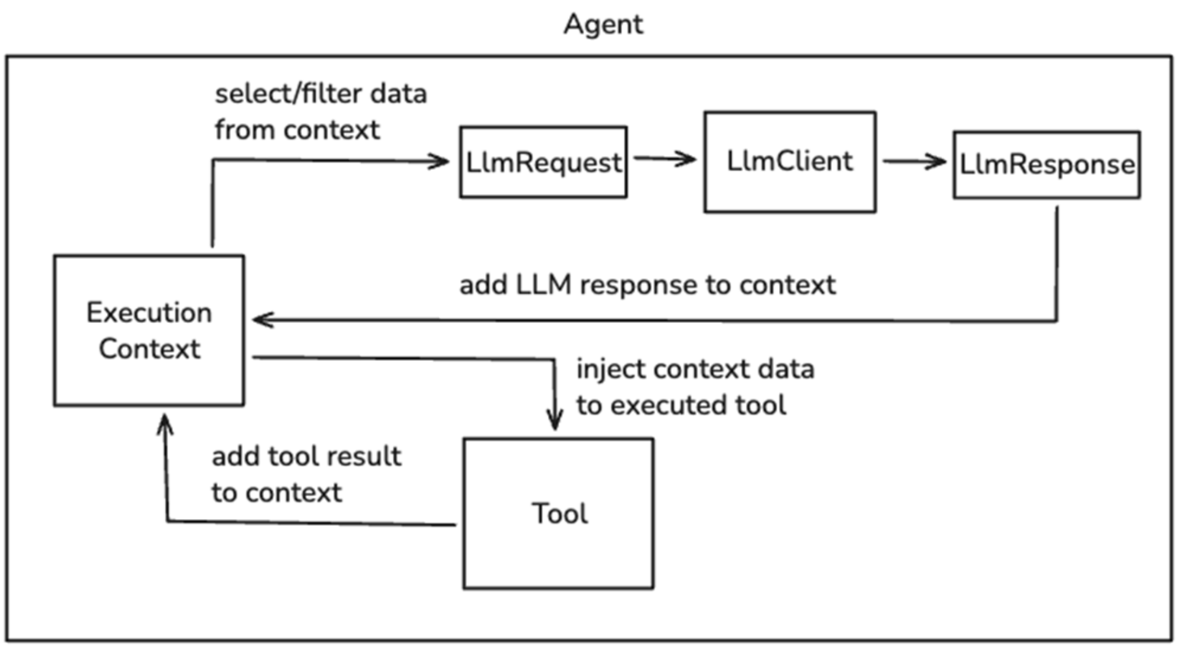

Agent
select/filter data from context
LlmRequest
LlmClient
LlmResponse
add LLM response to context
Execution Context
inject context data to executed tool
add tool result to context
Tool

Figure 4.3 Information flow in the agent architecture. ExecutionContext sits at the center, managing all execution state. Arrows show: (1) Context to LlmRequest to LlmClient for outgoing LLM calls, (2) LlmResponse back to Context for incoming responses, and (3) bidirectional flow between Context and Tools for tool execution.

ExecutionContext serves as the central storage where all information converges. When the agent needs to call the

LLM, it extracts relevant information from ExecutionContext into an LlmRequest. When the LLM responds, the response flows back into ExecutionContext. When tools execute, the agent passes ExecutionContext to them so they can access any execution state they need, and their results are recorded back into ExecutionContext.

This architecture provides two key benefits. First, it simplifies method signatures since every method just receives the context. Second, it enables **context propagation**, where the agent can pass ExecutionContext to tools that need access to execution state (for example, a tool that checks user permissions before acting).

Notice the separation between LlmRequest and ExecutionContext. While ExecutionContext stores everything, LlmRequest is a curated subset containing only what the LLM needs for a specific call. This separation is where context engineering happens: deciding what information to include, what to omit, and how to format it for optimal LLM performance.

4.2.3 Components we need to build

With the architecture clear, here's our implementation roadmap. Each component has a specific role in managing the information flow we just discussed

Table 4.1 Agent components and implementation roadmap

Order	Component	Section	Role
1	ExecutionContext	4.3	Central storage manages all information during execution
2	Tool Abstraction	4.4	Unifies tools from chapter 3 under a consistent interface that can receive Context
3	LLM Communication Layer	4.5	LlmRequest selects information for the LLM; LlmClient handles API calls; LlmResponse standardizes responses
4	Agent	4.6	Orchestrator that creates Context, coordinates information flow, and implements the think-act loop

We'll build from the foundation up. ExecutionContext comes first because everything else depends on it. Tool abstraction follows because we need a consistent way to define and execute tools. The LLM communication layer standardizes how we talk to language models. Finally, the Agent class brings everything together into the think-act loop that solves problems.

4.3 ExecutionContext: The agent's central storage

We need ExecutionContext to avoid scattering state across multiple variables and to simplify how components share information. Now, let's determine *what* we need to store by examining what actually happens when an agent runs.

4.3.1 What happens during agent execution?

Recall the Kipchoge problem from section 4.1. When an agent solves this problem, a series of events unfolds:

1. User asks: "How long would it take Kipchoge to run to the Moon?"
2. LLM decides it needs information → requests `search_web` tool.
3. Tool executes and returns: "Kipchoge's record is 2:01:09 for 42.195km."
4. LLM needs more information → requests `search_wikipedia` tool.
5. Tool executes and returns: "Moon's perigee is 356,500 k.m."
6. LLM can now calculate → requests `calculator` tool.
7. Tool executes and returns: "17034"
8. LLM provides the final answer: "Approximately 17,000 hours."

Looking at this sequence, we can identify three distinct types of occurrences:

Messages are text exchanges in the conversation. The user's initial question and the LLM's final answer are both messages. Each message has a role (user, assistant, or system) and text content.

Tool Calls occur when the LLM decides to use a tool. The LLM specifies which tool to call and with what arguments. In step 2 above, the LLM requests `search_web` with the query "Kipchoge marathon world record."

Tool Results capture what happens when tools execute. They include the tool's output and whether execution succeeded or failed. Step 3 above would be a successful result containing Kipchoge's marathon time.

Let's define these as data types.

Listing 4.2 Content types for agent communication

```
class Message(BaseModel):
    """A text message in the conversation."""
    type: Literal["message"] = "message"
    role: Literal["system", "user", "assistant"]
    content: str

class ToolCall(BaseModel):
    """LLM's request to execute a tool."""
    type: Literal["tool_call"] = "tool_call"
    tool_call_id: str
    name: str
    arguments: dict

class ToolResult(BaseModel):
    """Result from tool execution."""
    type: Literal["tool_result"] = "tool_result"
    tool_call_id: str
    name: str
    status: Literal["success", "error"]
    content: list

ContentItem = Union[Message, ToolCall, ToolResult]
```

The `type` field in each class serves as a discriminator, making it easy to identify what kind of content we're dealing with. The `tool_call_id` links a ToolResult back to its originating ToolCall, which is essential when multiple tools execute in parallel.

These content types capture *what* happened, but for debugging and analysis, we also need to know *who* produced each piece of content and *when*. We wrap content items with metadata using an Event class.

Listing 4.3 Event class for execution tracking

```
class Event(BaseModel):
    """A recorded occurrence during agent execution."""
    id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    execution_id: str
    timestamp: float = Field(default_factory=lambda: datetime.now().
timestamp())
    author: str  # "user" or agent name
    content: List[ContentItem] = Field(default_factory=list)
```

The `execution_id` groups all events from a single agent run, enabling us to trace an entire problem-solving session. The `author` field distinguishes between user input and agent actions. Here's how one step from the Kipchoge problem would look as an Event.

```
Event(
    execution_id="abc-123",
    author="research_agent",
    content=[ToolCall(
        tool_call_id="call-1",
        name="search_web",
        arguments={"query": "Kipchoge marathon world record"}
    )]
)
```

Each step in the agent's execution becomes an Event, creating a complete audit trail that proves invaluable when debugging.

4.3.2 Implementing ExecutionContext

Now that we know what to store, implementing ExecutionContext is straightforward.

Listing 4.4 ExecutionContext implementation

```
@dataclass
class ExecutionContext:
    """Central storage for all execution state."""

    execution_id: str = field(default_factory=lambda: str(uuid.uuid4
()))
    events: List[Event] = field(default_factory=list)
    current_step: int = 0
    state: Dict[str, Any] = field(default_factory=dict)
    final_result: Optional[str | BaseModel] = None

    def add_event(self, event: Event):
        """Append an event to the execution history."""
        self.events.append(event)

    def increment_step(self):
        """Move to the next execution step."""
        self.current_step += 1
```

Each field serves a specific purpose:

- `execution_id`: Unique identifier for this execution session, automatically generated
- `events`: Chronological list of all Events that occur during execution
- `current_step`: Counter to prevent infinite loops (agent stops after max_steps)
- `state`: Flexible key-value store for custom data that tools might need, such as API configurations or intermediate results
- `final_result`: Holds the agent's answer when execution completes

ExecutionContext can also be passed to tools that need access to the execution state. A tool might read from `state` or examine previous events in the execution history. For this to work, we need a consistent way to define tools that can optionally receive context. Now, let’s implement Tool Abstraction to unify the function-based tools under a common interface that supports this capability.

4.4 Tool abstraction

In chapter 3, we built tools as simple Python functions and created a utility to convert them into tool definitions for the LLM. This approach worked well for learning the basics of tool calling. But as we prepare to build a complete agent framework, several practical challenges emerge that our function-based approach doesn't handle well.

4.4.1 Why we need a unified tool interface

Let's revisit what we built in chapter 3. We created functions like `calculator()` and `search_web()`, then used `function_to_tool_definition()` to generate the schemas that LLMs need.

```
def search_web(query: str, max_results: int = 5) -> str:
    """Search the web for information."""
    return tavily_client.search(query, max_results=max_results)
search_tool_definition = function_to_tool_definition(search_web)
search_web(query="Python tutorials")
```

This works, but the function and its metadata exist as separate entities. When building an agent, we need to pass definitions to the LLM and map names back to functions for execution, keeping everything in sync manually.

With tool abstraction, everything stays bundled together.

```
@tool
def search_web(query: str, max_results: int = 5) -> str:
    return tavily_client.search(query, max_results=max_results)

search_web.tool_definition
search_web.execute(query="Python tutorials")
```

Let's implement this with two classes: `BaseTool`, defining the interface, and `FunctionTool` wrapping existing functions.

4.4.2 BaseTool: The foundation

Every tool in our framework will inherit from BaseTool, an abstract base class that defines what it means to be a tool. Let's examine what this interface needs to provide.

A tool must have three essential properties: a name for identification, a description explaining its purpose, and a tool_definition schema that tells the LLM how to use it. The tool must also be executable, which we'll represent with an async execute method that receives ExecutionContext along with any arguments.

The initialization provides sensible defaults: if no name is given, it uses the class name; if no description is given, it uses the docstring. This reduces boilerplate when creating simple tools. The `tool_definition` property returns the schema that tells the LLM what parameters the tool accepts. We store it as `_tool_definition` to allow subclasses to generate it lazily if needed.

Listing 4.5 BaseTool abstract class

```
class BaseTool(ABC):
    """Abstract base class for all tools."""

    def __init__(
        self,
        name: str = None,
        description: str = None,
        tool_definition: Dict[str, Any] = None,
    ):
        self.name = name or self.__class__.__name__
        self.description = description or self.__doc__ or ""
        self._tool_definition = tool_definition

    @property
    def tool_definition(self) -> Dict[str, Any] | None:
        return self._tool_definition

    @abstractmethod
    async def execute(self, context: ExecutionContext, **kwargs) ->
Any:
        pass

    async def __call__(self, context: ExecutionContext, **kwargs) ->
Any:
        return await self.execute(context, **kwargs)
```

The `execute` method is abstract because every tool must implement its own logic. Notice that it always receives `context` as its first parameter. This is the key design decision that enables context propagation: the Agent passes ExecutionContext to every tool, and tools can use it if they need access to execution state. Tools that don't need context simply ignore it.

Finally, `__call__` provides syntactic convenience. Instead of writing `tool.execute(context, query="test")`, we can write `tool(context, query="test")`.

With BaseTool defined, we have a contract that all tools must follow. In the next section, we'll implement FunctionTool to wrap the simple functions we created in Chapter 3.

4.4.3 FunctionTool: Wrapping functions

FunctionTool acts as an adapter that wraps existing functions with the BaseTool interface. The key challenge is context propagation: the Agent passes ExecutionContext to every tool, but most functions don't expect a context parameter. They have signatures like `search_web(query: str)`, not `search_web(context: ExecutionContext, query: str)`.

The solution is to inspect each function's signature at initialization. If the function has a `context` parameter, we pass it. If not, we omit it. Simple functions work unchanged, while context-aware functions can access execution state when needed.

During initialization, we check whether the function accepts a `context` parameter. The `execute` method uses this information to decide whether to forward the context. The Agent treats all tools identically, passing context to every one, and each tool decides whether to use it. The `_generate_definition` method reuses `function_to_input_schema` from Chapter 3 to automatically create the tool schema from the function's type hints.

Listing 4.6 FunctionTool implementation

```
class FunctionTool(BaseTool):
    """Wraps a Python function as a BaseTool."""

    def __init__(
        self,
        func: Callable,
        name: str = None,
        description: str = None,
        tool_definition: Dict[str, Any] = None
    ):
        self.func = func
        self.needs_context = 'context' in inspect.signature(func).pa
rameters

        name = name or func.__name__
        description = description or (func.__doc__ or "").strip()
        tool_definition = tool_definition or self._generate_definiti
on()

        super().__init__(
            name=name,
            description=description,
            tool_definition=tool_definition
        )

    async def execute(self, context: ExecutionContext, **kwargs) ->
Any:
        """Execute the wrapped function."""
        if self.needs_context:
            result = self.func(context=context, **kwargs)
        else:
            result = self.func(**kwargs)

        # Handle both sync and async functions
        if inspect.iscoroutine(result):
            return await result
        return result

    def _generate_definition(self) -> Dict[str, Any]:
        """Generate tool definition from function signature."""
        parameters = function_to_input_schema(self.func)
        return format_tool_definition(self.name, self.description, p
arameters)
```

Here's FunctionTool in action. The function's name, docstring, and parameters are automatically extracted to create a fully functional tool.

```
def calculator(expression: str) -> float:
    """Calculate mathematical expressions."""
    return eval(expression)

calc_tool = FunctionTool(calculator)

print(calc_tool.name)           # "calculator"
print(calc_tool.description)    # "Calculate mathematical expression
s."
await calc_tool(context, expression="1234 * 5678")  # 7006652
```

THE @TOOL DECORATOR

For convenience, we provide a decorator that creates FunctionTool instances. The following code is equivalent to writing `calculator = FunctionTool(calculator)`, but with cleaner syntax

```
@tool
def calculator(expression: str) -> float:
    """Calculate mathematical expressions."""
    return eval(expression)
```

The decorator also accepts optional parameters to override the tool's metadata. Instead of using the function's name and docstring, you can specify custom values that will appear in the tool definition sent to the LLM.

```
@tool(name="web_search", description="Search the internet for inform
ation")
def search_web(query: str) -> str:
    """Search the web."""
    return tavily_client.search(query)
```

Throughout this book, we'll use `@tool` for its cleaner syntax. The implementation is a thin wrapper around FunctionTool. See the GitHub repository for details.

4.4.4 Integrating MCP Tools

We explored MCP and connected to the MCP servers to discover and execute tools. Now we need to use those tools in our agent framework. The challenge is that MCP tools live on external servers and are executed via `session.call_tool()`, while our Agent expects BaseTool instances with an `execute()` method.

The solution is to create a wrapper function for each MCP tool that calls the server, then wrap that function with FunctionTool. This approach reuses everything we've already built.

The `load_mcp_tools` function connects to an MCP server, retrieves all available tools, and converts each one into a FunctionTool. It takes connection parameters (like the command to start the server) and returns a list of tools ready for use with our agent.

Listing 4.7 Loading tools from MCP servers

```
async def load_mcp_tools(connection: dict) -> list[BaseTool]:
    """Load tools from an MCP server and convert to FunctionTool
s."""
    tools = []

    async with stdio_client(StdioServerParameters(**connection)) as
(read, write):
        async with ClientSession(read, write) as session:
            await session.initialize()
            mcp_tools = await session.list_tools()

            for mcp_tool in mcp_tools.tools:
                func_tool = _create_mcp_tool(mcp_tool, connection)
                tools.append(func_tool)

    return tools
```

The core logic lives in `_create_mcp_tool`. For each MCP tool, it creates a wrapper function called `call_mcp` that establishes a connection to the server and executes the tool with whatever arguments are passed. This wrapper function is then wrapped with FunctionTool. Notice that we pass `tool_definition` explicitly rather than letting FunctionTool generate it, because the MCP server already provides a complete schema via `mcp_tool.inputSchema`.

Listing 4.8 Creating FunctionTool from the MCP tool

```
def _create_mcp_tool(mcp_tool, connection: dict) -> FunctionTool:
    """Create a FunctionTool that wraps an MCP tool."""

    async def call_mcp(**kwargs):
        async with stdio_client(StdioServerParameters(**connection))
as (read, write):
            async with ClientSession(read, write) as session:
                await session.initialize()
                result = await session.call_tool(mcp_tool.name, kwar
gs)
                return _extract_text_content(result)

    tool_definition = {
        "type": "function",
        "function": {
            "name": mcp_tool.name,
            "description": mcp_tool.description,
            "parameters": mcp_tool.inputSchema,
        }
    }

    return FunctionTool(
        func=call_mcp,
        name=mcp_tool.name,
        description=mcp_tool.description,
        tool_definition=tool_definition
    )
```

With this adapter, MCP tools integrate seamlessly with our agent. The following example loads tools from the Tavily MCP server and combines them with a local calculator tool. The agent doesn't know or care that some tools are local functions and others call remote MCP servers. They all implement the same BaseTool interface.

```
connection = {
    "command": "npx",
    "args": ["-y", "tavily-mcp@latest"],
    "env": {"TAVILY_API_KEY": os.getenv("TAVILY_API_KEY")}
}
mcp_tools = await load_mcp_tools(connection)

agent = Agent(
    model=LlmClient(model="gpt-5"),
    tools=[calculator, *mcp_tools],
    instructions="You are a helpful assistant."
)
```

A NOTE ON CONNECTION MANAGEMENT

This implementation creates a new connection for each tool call. While simple to understand, this approach adds overhead. In production systems with frequent MCP tool usage, you might maintain a persistent session to avoid repeated connection setup.

4.5 LLM Communication layer

With ExecutionContext storing our execution state and tools unified under BaseTool, we need one more piece: a way to communicate with the LLM. This communication layer bridges the gap between our internal data structures and the LLM's API requirements.

4.5.1 Why a communication layer?

Consider what happens when our agent needs to call the LLM. ExecutionContext contains a list of Events, each holding ContentItems like Messages, ToolCalls, and ToolResults. But LLM APIs expect a specific message format: a list of dictionaries with roles and content. Someone needs to translate between these representations.

We also face the challenge of provider diversity. While we use LiteLLM to abstract away most provider differences (as we learned in chapter 2), we still need to structure our requests consistently and parse responses into a standard format. Without this layer, translation logic would scatter throughout the Agent class, making it harder to maintain and extend.

The solution is three components working together:

LlmRequest packages what we want to send: instructions, conversation contents, and available tools. It serves as a staging area where we select and organize information before the API call.

LlmClient handles the actual API communication. It transforms LlmRequest into the format LLM APIs expect, makes the call, and converts the response back.

LlmResponse standardizes what we receive. Regardless of which provider we use, responses come back in the same format that our Agent can process.

This separation keeps each component focused. LlmRequest knows nothing about API formats. LlmClient knows nothing about ExecutionContext. Each piece does one job well.

4.5.2 LlmRequest: Selecting what to send

LlmRequest is the outbound gate from our agent to the LLM. It holds everything needed for a single LLM call, organized into four components.

Listing 4.9 LlmRequest class

```
class LlmRequest(BaseModel):
    """Request object for LLM calls."""
    instructions: List[str] = Field(default_factory=list)
    contents: List[ContentItem] = Field(default_factory=list)
    tools: List[BaseTool] = Field(default_factory=list)
    tool_choice: Optional[str] = None
```

The `instructions` field holds system prompt fragments. Rather than a single monolithic prompt, we allow multiple instruction strings that get combined. This flexibility proves useful when instructions come from different sources: base agent instructions, task-specific guidance, or dynamically generated context.

The `contents` field contains the conversation history as ContentItem objects: Messages from users and assistants, ToolCalls the LLM requested, and ToolResults from executions. This is the core context the LLM uses to understand the current situation.

The `tools` field lists available tools as BaseTool instances. LlmClient will extract its definitions when building the API request.

The `tool_choice` field controls how the LLM selects tools. Setting it to `"auto"` lets the LLM decide freely. Setting it to `"required"` forces tool usage, which becomes important for structured output in section 4.6.

Notice what LlmRequest does not contain: it has no reference to ExecutionContext or Events. The Agent is responsible for extracting relevant information from ExecutionContext and packaging it into LlmRequest. This separation is intentional. It makes LlmRequest a simple data container while keeping context selection logic in the Agent where it belongs.

This is where context engineering happens. In section 4.6, we will implement `_prepare_llm_request()` in the Agent class, which decides what information from ExecutionContext goes into each LlmRequest. For now, we will flatten all events into contents. In Chapter 6, this becomes the extension point for memory and context management strategies.

4.5.3 LlmResponse: Standardizing what we receive

LlmResponse is the inbound gate, standardizing LLM responses regardless of which provider generated them.

Listing 4.10 LlmResponse class

```
class LlmResponse(BaseModel):
    """Response object from LLM calls."""
    content: List[ContentItem] = Field(default_factory=list)
    error_message: Optional[str] = None
    usage_metadata: Dict[str, Any] = Field(default_factory=dict)
```

The `content` field contains what the LLM produced, represented as ContentItem objects. A response might include a Message with text, one or more ToolCalls requesting tool execution, or both. Using the same ContentItem types we defined in section 4.3 keeps our data model consistent throughout the system.

The `error_message` field captures failures. When an API call fails due to network issues, rate limits, or invalid requests, we store the error here rather than raising an exception. This allows the Agent to handle failures gracefully, perhaps by retrying or informing the user.

The `usage_metadata` field tracks token consumption. Knowing how many input and output tokens each call uses helps with cost management and debugging. For now, we simply store this information. Chapter 10 covers monitoring and cost tracking in detail.

4.5.4 LlmClient: The provider adapter

LlmClient bridges LlmRequest and LlmResponse with actual LLM APIs. Thanks to LiteLLM, which we introduced in chapter 2, we can support multiple providers through a single implementation.

Listing 4.11 LlmClient class

```
class LlmClient:
    """Client for LLM API calls using LiteLLM."""

    def __init__(self, model: str, **config):
        self.model = model
        self.config = config

    async def generate(self, request: LlmRequest) -> LlmResponse:
        """Generate a response from the LLM."""
        try:
            messages = self._build_messages(request)
            tools = [t.tool_definition for t in request.tools] if re
quest.tools else None

            response = await acompletion(
                model=self.model,
                messages=messages,
                tools=tools,
                **({"tool_choice": request.tool_choice}
                   if request.tool_choice else {}),
                **self.config
            )

            return self._parse_response(response)
        except Exception as e:
            return LlmResponse(error_message=str(e))
```

The constructor takes a model identifier (like `"gpt-5-mini"` or `"anthropic/claude-sonnet-4-5"`) and optional configuration parameters such as temperature or max_tokens.

The `generate` method orchestrates the API call in three steps. First, it builds the messages list from the request. Second, it extracts tool definitions with a simple list comprehension. Third, it calls LiteLLM's `acompletion` and parses the response. If anything fails, it returns an LlmResponse with the error captured rather than crashing.

BUILDING MESSAGES

The `_build_messages` method transforms LlmRequest contents into the message format that LLM APIs expect. The main complexity here is that OpenAI's API requires assistant messages and their tool calls to appear together in a single message object, while we store them as separate ContentItems.

Listing 4.12 Building messages for LLM API

```
def _build_messages(self, request: LlmRequest) -> List[dict]:
    """Convert LlmRequest to API message format."""
    messages = []

    for instruction in request.instructions:
        messages.append({"role": "system", "content": instruction})

    for item in request.contents:
        if isinstance(item, Message):
            messages.append({"role": item.role, "content": item.cont
ent})

        elif isinstance(item, ToolCall):
            tool_call_dict = {
                "id": item.tool_call_id,
                "type": "function",
                "function": {
                    "name": item.name,
                    "arguments": json.dumps(item.arguments)
                }
            }
            # Append to previous assistant message if exists
            if messages and messages[-1]["role"] == "assistant":
                messages[-1].setdefault("tool_calls", []).append(too
l_call_dict)
            else:
                messages.append({
                    "role": "assistant",
                    "content": None,
                    "tool_calls": [tool_call_dict]
                })

        elif isinstance(item, ToolResult):
            messages.append({
                "role": "tool",
                "tool_call_id": item.tool_call_id,
                "content": str(item.content[0]) if item.content else
""
            })

    return messages
```

The logic handles each ContentItem type differently. Messages become standard message objects. ToolCalls get appended to the preceding assistant message's `tool_calls` array, since our agent stores them consecutively from the same LLM response. If a ToolCall appears without a preceding assistant message, we create one with null content. ToolResults become tool-role messages linked back to their originating call via `tool_call_id`.

PARSING RESPONSES

The `_parse_response` method converts API responses back into our standard format. The method handles both text responses and tool calls. When the LLM returns text, we wrap it in a Message. When it requests tools, we create ToolCall objects. Both can appear in the same response, and our ContentItem list accommodates this naturally.

Listing 4.13 Parsing LLM API response

```
def _parse_response(self, response) -> LlmResponse:
    """Convert API response to LlmResponse."""
    choice = response.choices[0]
    content_items = []

    if choice.message.content:
        content_items.append(Message(
            role="assistant",
            content=choice.message.content
        ))

    if choice.message.tool_calls:
        for tc in choice.message.tool_calls:
            content_items.append(ToolCall(
                tool_call_id=tc.id,
                name=tc.function.name,
                arguments=json.loads(tc.function.arguments)
            ))

    return LlmResponse(
        content=content_items,
        usage_metadata={
            "input_tokens": response.usage.prompt_tokens,
            "output_tokens": response.usage.completion_tokens,
        }
    )
```

4.5.5 Putting it together

With all three components defined, here is how they work together. We create an LlmClient with a model identifier, build an LlmRequest containing our instructions and conversation contents, and call `generate()` to get an LlmResponse. The response's content list contains the LLM's output as ContentItem objects, which we can iterate through to extract the answer.

Listing 4.14 Using the LLM communication layer

```
from scratch_agents import LlmClient, LlmRequest, Message

# Create client
client = LlmClient(model="gpt-5-mini")

# Build request
request = LlmRequest(
    instructions=["You are a helpful assistant."],
    contents=[Message(role="user", content="What is 2 + 2?")],
)

# Generate response
response = await client.generate(request)

# Response contains the answer
for item in response.content:
    if isinstance(item, Message):
        print(item.content)  # "4"
```

The communication layer provides clean separation of concerns. LlmRequest packages what to send. LlmClient handles how to send it. LlmResponse standardizes what comes back.

4.6 Implementing the agent

With ExecutionContext storing our state, tools unified under BaseTool, and LlmClient handling LLM communication, we're ready to build the Agent class that orchestrates everything. The Agent brings together all the components we've built, coordinating information flow between the LLM and tools to solve problems step by step.

Figure 4.4 shows the complete picture of what we're implementing. The Agent initializes with `_setup_tools()`, then `run()` creates an ExecutionContext and repeatedly calls `step()` until completion. Each `step()` performs one think-act cycle:

`think()` calls the LLM to decide what to do next, and `act()` executes any requested tools.

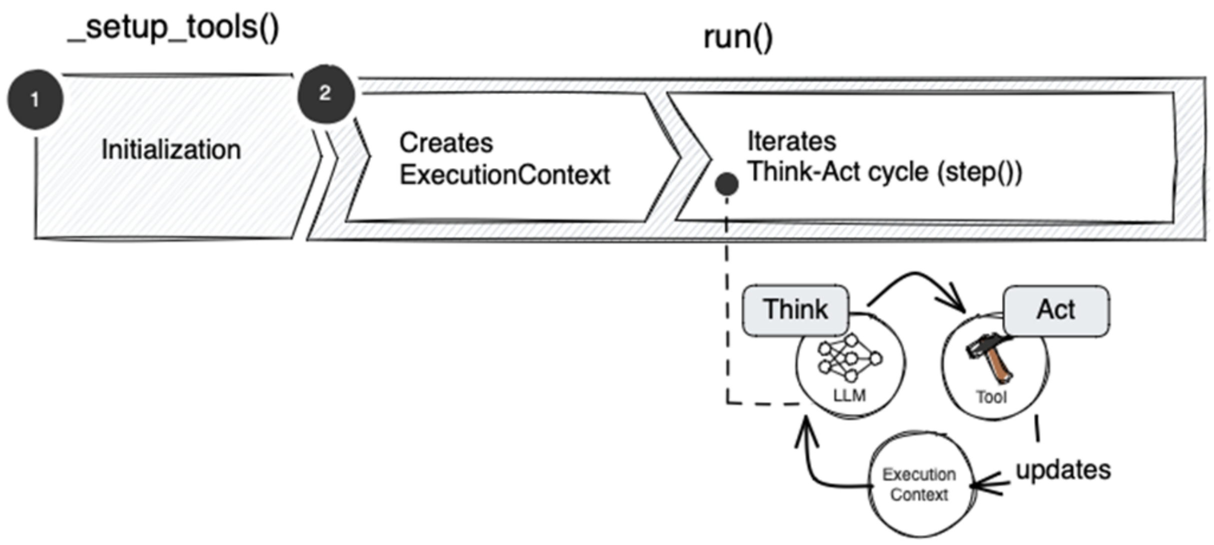


Figure 4.4 Agent implementation roadmap showing the five core methods and their execution flow. The numbered sequence illustrates how each component works together to solve problems.

We'll implement these five methods in a logical order:

1. `_setup_tools()`: Registers the tools the agent will use, converting the list into a dictionary for efficient lookup
2. `run()`: The entry point that receives user requests, manages the execution loop, and returns the final result
3. `step()`: Performs one complete think-act cycle, updating ExecutionContext with each interaction
4. `think()`: Calls the LLM to analyze the situation and decide the next action
5. `act()`: Executes the tools selected by the LLM and captures their results

By the end of this section, you'll have a working agent that can solve multi-step problems. We'll also add structured output capabilities, allowing the agent to return typed

Pydantic models instead of free-form text when your application requires predictable data formats.

4.6.1 Agent class structure

Let's start with the Agent class constructor and the `_setup_tools()` method. The constructor receives all the components we've built in previous sections and stores them for use during execution.

Listing 4.15 Agent class constructor

```
class Agent:
    def __init__(
        self,
        model: LlmClient,
        tools: List[BaseTool] = None,
        instructions: str = "",
        max_steps: int = 10,
    ):
        self.model = model
        self.instructions = instructions
        self.max_steps = max_steps
        self.tools = self._setup_tools(tools or [])
```

The parameters serve distinct purposes:

- `model`: The LlmClient instance that handles LLM communication
- `tools`: List of BaseTool instances the agent can use
- `instructions`: System prompt that defines the agent's behavior
- `max_steps`: Safety limit to prevent infinite loops when the agent keeps calling tools without reaching a conclusion

The `_setup_tools()` method prepares the tools list for use. For now, it simply returns the list unchanged, but this method serves as an extension point for future enhancements like adding default tools.

Listing 4.16 Setting up agent tools

```
def _setup_tools(self, tools: List[BaseTool]) -> List[BaseTool]:
    return tools
```

With the basic structure in place, let's implement the `run()` method that orchestrates the entire execution.

4.6.2 The run() Method

The `run()` method is the entry point for agent execution. It creates the execution environment, manages the think-act loop, and returns the result.

To give callers access to the full execution trace, we define `AgentResult` that bundles the final output with its `ExecutionContext`. This allows inspection of every step the agent took, which proves invaluable for debugging and analysis.

Listing 4.17 AgentResult dataclass

```
@dataclass
class AgentResult:
    """Result of an agent execution."""
    output: str | BaseModel
    context: ExecutionContext
```

The method first checks if an ExecutionContext was provided, creating one if not. It then wraps the user's input in an Event and adds it to the context. The main loop repeatedly calls `step()` until either a final result is obtained or the maximum step limit is reached. After each step, we check if the last event represents a final response and extract the result if so.

Listing 4.18 Agent run method

```
async def run(
    self,
    user_input: str,
    context: ExecutionContext = None
) -> str:
    # Create or reuse context
    if context is None:
        context = ExecutionContext()

    # Add user input as the first event
    user_event = Event(
        execution_id=context.execution_id,
        author="user",
        content=[Message(role="user", content=user_input)]
    )
    context.add_event(user_event)

    # Execute steps until completion or max steps reached
    while not context.final_result and context.current_step < self.m
ax_steps:
        await self.step(context)

        # Check if the last event is a final response
        last_event = context.events[-1]
        if self._is_final_response(last_event):
            context.final_result = self._extract_final_result(last_e
vent)

    return AgentResult(output=context.final_result, context=context)
```

The helper methods handle completion detection and result extraction. The `_is_final_response()` method checks whether an event represents a final answer by examining its contents. An event is final when it contains neither tool calls nor tool results, meaning the LLM provided a direct answer. The `_extract_final_result()` method iterates through the event's content to find the assistant's message.

Listing 4.19 Completion detection helpers

```
def _is_final_response(self, event: Event) -> bool:
    """Check if this event contains a final response."""
    has_tool_calls = any(isinstance(c, ToolCall) for c in event.cont
ent)
    has_tool_results = any(isinstance(c, ToolResult) for c in event.
content)
    return not has_tool_calls and not has_tool_results

def _extract_final_result(self, event: Event) -> str:
    for item in event.content:
        if isinstance(item, Message) and item.role == "assistant":
            return item.content
    return None
```

Usage is straightforward. Access the answer via `result.output`, and examine execution details through `result.context` when needed.

```
result = await agent.run("What is 1234 * 5678?")
print(result.output)                      # "7006652"
print(result.context.current_step)
```

4.6.3 The step() method

The `step()` method performs one complete think-act cycle. It prepares a request for the LLM, gets the LLM's decision, and executes any tools the LLM requests.

The method starts by calling `_prepare_llm_request()` to package the current context into a format suitable for the LLM. It then calls `think()` to get the LLM's response and wraps that response in an Event, recording it in the context. If the response contains tool calls, the method calls `act()` to execute them and record the results as another Event. Finally, it increments the step counter.

Listing 4.20 Agent step method

```
async def step(self, context: ExecutionContext):
    # Prepare what to send to the LLM
    llm_request = self._prepare_llm_request(context)

    # Get LLM's decision
    llm_response = await self.think(llm_request)

    # Record LLM response as an event
    response_event = Event(
        execution_id=context.execution_id,
        author=self.name,
        content=llm_response.content,
    )
    context.add_event(response_event)

    # Execute tools if the LLM requested any
    tool_calls = [c for c in llm_response.content if isinstance(c, T
oolCall)]
    if tool_calls:
        tool_results = await self.act(context, tool_calls)
        tool_event = Event(
            execution_id=context.execution_id,
            author=self.name,
            content=tool_results,
        )
        context.add_event(tool_event)

    context.increment_step()
```

The `_prepare_llm_request()` method extracts information from ExecutionContext and packages it into an LlmRequest. It flattens all events into a list of content items, combines them with the agent's instructions and available tools, and sets the tool choice to "auto" so the LLM can decide whether to use tools.

Listing 4.21 Preparing LLM request from context

```
def _prepare_llm_request(self, context: ExecutionContext) -> LlmRequ
est:
    # Flatten events into content items
    flat_contents = []
    for event in context.events:
        flat_contents.extend(event.content)

    return LlmRequest(
        instructions=[self.instructions] if self.instructions else
[],
        contents=flat_contents,
        tools=self.tools,
        tool_choice="auto" if self.tools else None,
    )
```

Notice the separation between ExecutionContext and LlmRequest. ExecutionContext holds the complete execution history, while LlmRequest contains only what we choose to send to the LLM. Currently, we send everything, but this separation is where context engineering happens. You could summarize old messages, omit irrelevant tool results, or inject additional information, all by modifying how you build the LlmRequest without touching the original context.

4.6.4 The think() and act() methods

The `think()` and `act()` methods handle the two core operations in each step: getting the LLM's decision and executing the requested tools.

The `think()` method is intentionally simple. It takes the prepared LlmRequest and passes it to the LlmClient, returning whatever response comes back. All the complexity of building the request happens in `_prepare_llm_request()`, and all the complexity of parsing the response happens in LlmClient. This keeps `think()` focused on a single responsibility.

Listing 4.22 Agent think method

```
async def think(self, llm_request: LlmRequest) -> LlmResponse:
    return await self.model.generate(llm_request)
```

The `act()` method executes the tools that the LLM requested. It first builds a dictionary mapping tool names to tool instances for efficient lookup. Then it iterates through each ToolCall, retrieves the corresponding tool, executes it with the provided arguments, and collects the results. The method passes ExecutionContext to each tool execution, allowing tools to access execution state if they need it. Each execution is wrapped in a try-except block, capturing failures as error results rather than crashing the entire agent.

Listing 4.23 Agent act method

```
async def act(
    self,
    context: ExecutionContext,
    tool_calls: List[ToolCall]
) -> List[ToolResult]:
    tools_dict = {tool.name: tool for tool in self.tools}
    results = []

    for tool_call in tool_calls:
        if tool_call.name not in tools_dict:
            raise ValueError(f"Tool '{tool_call.name}' not found")

        tool = tools_dict[tool_call.name]

        try:
            output = await tool(context, **tool_call.arguments)
            results.append(ToolResult(
                tool_call_id=tool_call.tool_call_id,
                name=tool_call.name,
                status="success",
                content=[output],
            ))
        except Exception as e:
            results.append(ToolResult(
                tool_call_id=tool_call.tool_call_id,
                name=tool_call.name,
                status="error",
                content=[str(e)],
            ))

    return results
```

When a tool execution fails, the error message becomes part of the conversation history. The LLM sees this failure in the next step and can adapt its approach, perhaps trying a different tool or rephrasing its query. This graceful error handling is one of the advantages of the ReAct pattern: the agent can recover from failures rather than stopping entirely.

4.7 Adding structured output

In chapter 2, we tested LLMs on GAIA benchmark problems using Structured Output. We defined a `GaiaOutput` Pydantic model with fields like `is_solvable`, `unsolvable_reason`, and `final_answer`, then passed it as the `response_format` parameter to ensure the LLM returned data in a predictable structure. This approach worked well for standalone LLM calls.

However, the Agent we built in section 4.6 returns only free-form text. When the LLM finishes reasoning and provides its final answer, `_extract_final_result()` simply extracts the text content from the assistant message. This works fine for conversational interactions, but many production scenarios require structured data that downstream systems can reliably parse.

Consider an agent that analyzes customer feedback. Instead of returning "The sentiment is positive with a confidence of 85%", you might need it to return a structured object with `sentiment: "positive"` and `confidence: 0.85`. Or imagine a research agent that must return findings in a specific format for a report generation pipeline. Free-form text would require fragile parsing logic that could break whenever the LLM phrases things differently.

To address this, we'll add structured output capability to our Agent. This exercise serves two purposes: you'll reinforce your understanding of the Agent architecture by modifying its core methods, and you'll learn a pattern for extending agents with new features.

4.7.1 The approach: Tools as output formatters

The key insight is that we can leverage our existing tool-calling infrastructure to enforce structured outputs. Instead of having the LLM generate free text and then trying to parse it into a structure, we create a special tool that accepts our desired output schema as its input. When the agent needs to provide its final answer, it calls this tool with properly structured data, guaranteeing type safety and format compliance.

This approach has several advantages. First, it reuses our existing tool-calling mechanism, keeping the implementation simple and consistent. Second, it leverages the LLM's extensive training on tool calling to ensure proper formatting. The LLM already knows how to generate valid JSON matching a schema; we simply give it a schema that matches our desired output structure.

The implementation requires modifications to five methods, each playing a specific role in the Agent's architecture.

Method	Role	Modification
`__init__`	Stores configuration	Add `output_type` parameter
`_setup_tools()`	Prepares available tools	Create `final_answer` tool dynamically
`_prepare_llm_request()`	Packages context for LLM	Force tool usage when structured output needed
`_is_final_response()`	Detects completion	Check for `final_answer` tool success
`_extract_final_result()`	Extracts the answer	Return Pydantic model from ToolResult

Let's implement each modification.

4.7.2 Modifying the agent

Constructor changes

First, we add an `output_type` parameter to the constructor. When provided, this should be a Pydantic model class that defines the expected output structure. We also add `output_tool_name` to track the name of our dynamically created tool.

Listing 4.24 Agent constructor with structured output

```
class Agent:
    def __init__(
        self,
        model: LlmClient,
        tools: List[BaseTool] = None,
        instructions: str = "",
        max_steps: int = 10,
        output_type: Optional[Type[BaseModel]] = None,  # New parame
ter
    ):
        self.model = model
        self.instructions = instructions
        self.max_steps = max_steps
        self.output_type = output_type
        self.output_tool_name = None  # Will be set if output_type p
rovided
        self.tools = self._setup_tools(tools or [])
```

Dynamic tool creation in _setup_tools()

The key logic happens in `_setup_tools()`. When an `output_type` is specified, we dynamically create a `final_answer` tool that accepts the specified Pydantic model as its input. This tool simply returns whatever structured data it receives, but by making it a tool, we ensure the LLM must format its response according to our schema.

Notice how we use Python's closure to capture `self.output_type` in the tool definition. The `@tool` decorator we implemented earlier automatically handles the Pydantic model validation, ensuring that any call to `final_answer` must provide data matching our schema.

Listing 4.25 Dynamic output tool creation

```
def _setup_tools(self, tools: List[BaseTool]) -> List[BaseTool]:
    if self.output_type is not None:
        @tool(
            name="final_answer",
            description="Return the final structured answer matching
the required schema."
        )
        def final_answer(output: self.output_type) -> self.output_ty
pe:
            return output

        tools = list(tools)  # Create a copy to avoid modifying the
original
        tools.append(final_answer)
        self.output_tool_name = "final_answer"

    return tools
```

Forcing tool usage in _prepare_llm_request()

When structured output is required, we need to ensure the agent actually uses the `final_answer` tool instead of providing a free-text response. The OpenAI API and most other providers support a `tool_choice` parameter that can force the model to use tools. We modify `_prepare_llm_request()` to set this appropriately.

The logic is straightforward: if we have an output tool (meaning structured output is needed), we set `tool_choice="required"` to force the LLM to use a tool. Otherwise, we fall back to the original behavior.

Listing 4.26 Forcing tool usage for structured output

```
def _prepare_llm_request(self, context: ExecutionContext) -> LlmRequ
est:
    flat_contents = []
    for event in context.events:
        flat_contents.extend(event.content)

    # Determine tool choice strategy
    if self.output_tool_name:
        tool_choice = "required"  # Force tool usage for structured
output
    elif self.tools:
        tool_choice = "auto"
    else:
        tool_choice = None

    return LlmRequest(
        instructions=[self.instructions] if self.instructions else
[],
        contents=flat_contents,
        tools=self.tools,
        tool_choice=tool_choice,
    )
```

Detecting completion in _is_final_response()

To properly detect when the agent has produced its final structured output, we modify `_is_final_response()`. When structured output is required, we look for a successful result from our `final_answer` tool. Only when this specific tool returns successfully do we consider the agent's work complete. For free-text responses, we maintain the original behavior.

Listing 4.27 Detecting structured output completion

```
def _is_final_response(self, event: Event) -> bool:
    if self.output_tool_name:
        # For structured output: check if final_answer tool succeede
d
        for item in event.content:
            if (isinstance(item, ToolResult)
                and item.name == self.output_tool_name
                and item.status == "success"):
                return True
        return False

    # Original logic for free-text responses
    has_tool_calls = any(isinstance(c, ToolCall) for c in event.cont
ent)
    has_tool_results = any(isinstance(c, ToolResult) for c in event.
content)
    return not has_tool_calls and not has_tool_results
```

Extracting structured results

Finally, we update `_extract_final_result()` to extract structured data from tool results. When a required output tool was specified, we look for its successful execution and return the validated Pydantic model instance. This gives the calling code a strongly-typed object to work with rather than raw text.

Listing 4.28 Extracting structured results

```
def _extract_final_result(self, event: Event) -> str | BaseModel:
    if self.output_tool_name:
        # Extract structured output from final_answer tool result
        for item in event.content:
            if (isinstance(item, ToolResult)
                and item.name == self.output_tool_name
                and item.status == "success"
                and item.content):
                return item.content[0]

    # Original logic for free-text responses
    for item in event.content:
        if isinstance(item, Message) and item.role == "assistant":
            return item.content
    return None
```

4.7.3 Using structured output in practice

With these modifications in place, using structured output becomes straightforward. Here's how you might build an agent that analyzes customer feedback and returns structured sentiment data.

Listing 4.29 Using structured output in practice

```
from pydantic import BaseModel
from typing import Literal, List

class SentimentAnalysis(BaseModel):
    sentiment: Literal["positive", "negative", "neutral"]
    confidence: float
    key_phrases: List[str]

agent = Agent(
    model=LlmClient(model="gpt-5-mini"),
    tools=[],
    instructions="Analyze the sentiment of the provided text.",
    output_type=SentimentAnalysis
)

result = await agent.run("This product exceeded my expectations! Hig
hly recommend.")
print(f"Sentiment: {result.sentiment}")      # "positive"
print(f"Confidence: {result.confidence}")    # 0.92
print(f"Key phrases: {result.key_phrases}")  # ["exceeded expectatio
ns", "highly recommend"]
```

The agent processes the input, reasons about the sentiment, and when ready to respond, calls the `final_answer` tool with a properly structured `SentimentAnalysis` object. The return type is the Pydantic model itself, not a string, so you can access fields directly with full type safety.

This pattern extends naturally to more complex scenarios. You could define output schemas for research summaries, data extraction results, or any structured format your application requires. By integrating structured output through our existing tool-calling mechanism, we've added powerful type safety and format guarantees without complicating the agent's core logic.

4.8 Testing with the GAIA benchmark

In chapter 2, we tested LLMs directly on GAIA problems and observed their limitations. Most problems required information that the LLM didn't have, leading to low accuracy. Now let's see how our Agent performs when equipped with tools.

4.8.1 From LLM to agent

The evaluation setup remains largely the same as in chapter 2. We reuse `GaiaOutput`, `gaia_prompt`, `is_correct()`, and the experiment runner structure. The key difference is what solves each problem: instead of a direct LLM call, we now use an Agent with tools.

First, we load tools from the Tavily MCP server, which we explored in chapter 3.

Listing 4.30 Loading MCP tools for GAIA

```
from scratch_agents.tools import load_mcp_tools

tavily_connection = {
    "command": "npx",
    "args": ["-y", "tavily-mcp@latest"],
    "env": {"TAVILY_API_KEY": os.getenv("TAVILY_API_KEY")}
}

mcp_tools = await load_mcp_tools(tavily_connection)
```

Next, we create an Agent configured for GAIA evaluation. The Agent receives the MCP tools for web search and uses `GaiaOutput` as its structured output type.

Listing 4.31 Creating GAIA evaluation agent

```
def create_gaia_agent(model: str, tools: list) -> Agent:
    return Agent(
        model=LlmClient(model=model),
        tools=tools,
        instructions=gaia_prompt,
        output_type=GaiaOutput,
        max_steps=15,
    )
```

The core change is in `solve_problem`. Where chapter 2 called `acompletion()` directly, we now call `agent.run()`. The returned `AgentResult` contains both the output and the full execution context.

Listing 4.32 Solving problems with agent

```
SEMAPHORE = asyncio.Semaphore(3)

async def solve_problem(agent: Agent, question: str) -> AgentResult:
    async with SEMAPHORE:
        return await agent.run(question)
```

The semaphore now limits concurrent Agent executions rather than individual LLM calls. Each Agent execution may involve multiple internal LLM calls as it reasons and uses tools, but we limit parallel problem-solving to avoid overwhelming the APIs.

The rest of the evaluation pipeline, including `evaluate_single()` and `run_experiment()`, follows the same structure as chapter 2. The only adjustment is accessing the answer via `result.output` instead of parsing the LLM response directly. See the GitHub repository for the complete implementation.

4.8.2 Results

The experiment compares an agent with web search tools against an agent without web search tools on ten GAIA problems that require web access. Both agents use gpt-5 as the underlying model.

Listing 4.33 Running GAIA experiment

```
models = [
    "gpt-5",
]

results = await run_experiment(
    problems=list(problems),
    models=models,
    tools=mcp_tools,
)
```

DEBUGGING WITH EXECUTION TRACES

When problems fail, we need to understand why. Since `AgentResult` includes the full `ExecutionContext`, we can examine exactly what happened during execution. For convenient visualization in Jupyter notebooks, we provide a `display_trace()` utility.

Listing 4.34 Debugging with execution traces

```
from scratch_agents.utils import display_trace
result = await agent.run(
    "If Eliud Kipchoge could maintain his marathon pace, "
    "how many thousand hours to reach the Moon?"
)

print(f"Answer: {result.output.final_answer}")
print(f"Steps: {result.context.current_step}")

display_trace(result.context)
```

This trace reveals the agent's reasoning: it identified the need for two pieces of information, searched for each, and produced the final answer. When debugging failures, you can pinpoint exactly where things went wrong.

THE IMPACT OF WEB SEARCH TOOLS

To quantify the value of web search capabilities, we ran a controlled experiment comparing agents with and without search tools on GAIA Level 1 problems that require web access. We selected ten problems where the official GAIA annotations indicated web search as a required tool.

Condition	Correct	Total cost
With web search	7/10	$1.89
Without web search	2/10	$0.26

The results demonstrate a 50 percentage point improvement when web search tools are available. Without search capabilities, the agent could only answer questions where the information happened to exist in its training data. With search tools, the agent successfully retrieved current information from Wikipedia, academic papers, and other web sources.

Examining the failures reveals interesting patterns. The agent without web search either admitted it could not solve the problem (returning `is_solvable=False`) or hallucinated incorrect answers. For instance, when asked about Mercedes Sosa's studio albums between 2000-2009, the agent with search correctly found 3 albums from Wikipedia, while the agent without search guessed 4.

REMAINING CHALLENGES

While web search dramatically improves performance on information retrieval tasks, many GAIA problems require additional capabilities. A significant portion of Level 1 problems includes attached files such as PDFs, spreadsheets, images, and audio files that the agent must process to find the answer. Our current agent cannot handle these file-based problems.

In chapter 5, we'll extend our agent with file processing tools that can read PDFs, parse spreadsheets, analyze images, and transcribe audio. This will unlock the remaining GAIA problems and move us closer to a truly general-purpose AI assistant.

4.9 Summary

- ReAct (Reasoning + Acting) is the foundational pattern behind modern AI agents, where the LLM alternates between reasoning about what it needs and taking actions to get it. Modern implementations use tool calling instead of fragile text parsing.
- ExecutionContext centralizes all execution state, including events, step count, execution ID, and flexible state storage. This design simplifies method signatures and enables context propagation to tools that need access to execution state.
- Tool abstraction through BaseTool and FunctionTool provides a consistent interface for all tools. The @tool decorator wraps simple functions, while load_mcp_tools() integrates external MCP servers, allowing the agent to use both local and remote tools seamlessly.
- The LLM communication layer separates concerns cleanly: LlmRequest packages what to send, LlmClient handles API communication via LiteLLM, and LlmResponse standardizes what comes back. This separation is where context engineering happens.

- The Agent class orchestrates everything through five core methods: _setup_tools() prepares available tools, run() manages the execution loop, step() performs one think-act cycle, think() calls the LLM, and act() executes requested tools.
- Structured output enables agents to return type-safe Pydantic models instead of free-form text. By creating a dynamic final_answer tool from the output schema, we leverage existing tool-calling infrastructure to guarantee format compliance.

5 Building knowledge bases with RAG

This chapter covers

- Why agents need external knowledge bases
- Different search methods: keyword, vector, graph, and structure-based search
- Implementing vector search from scratch with embeddings, chunking, and similarity calculation
- Building structure-based search tools for file system exploration
- Extending agent capabilities through the Callback pattern

In Chapters 2 through 4, we built the foundation of an AI agent: connecting to LLMs, implementing tool use, and creating the agent loop. Now we enter a new phase. As figure 5.1 shows, we'll now enhance our basic agent with context engineering strategies. This chapter tackles the first of these enhancements: building knowledge bases with RAG (Retrieval-Augmented Generation).

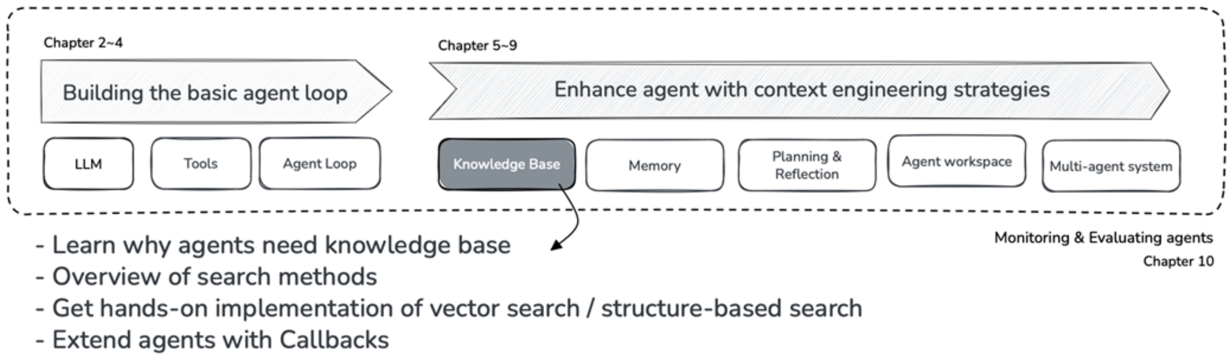

Chapter 2~4
Building the basic agent loop
LLM
Tools
Agent Loop
Chapter 5~9
Enhance agent with context engineering strategies
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Learn why agents need knowledge base
- Overview of search methods
- Get hands-on implementation of vector search / structure-based search
- Extend agents with Callbacks

Figure 5.1 Book structure overview - Chapter 5 in focus.

We’ll begin with the basic components of vector search: how embeddings capture text meaning, why chunking is necessary, and how vector databases operate. We’ll implement a mini vector search system to experience the full flow. Next, we extend to structure-based search using GAIA benchmark zip file problems, where the agent navigates folder structures and reads files like a human developer. Finally, we introduce the Callback pattern for extending agent behavior, implementing Human in the Loop approval and automatic search result compression.

Since this book focuses on LLM agents, it doesn't deeply cover all search methods in RAG. Detailed discussions of keyword search techniques (BM25, TF-IDF), graph search implementations (knowledge graph construction, query languages), or score combination strategies for hybrid search are beyond scope. Our emphasis is on how agents use knowledge base tools, focusing on vector search and structure-based search through practical exercises.

For readers who want to learn more, please refer to:

- *A Simple Guide to Retrieval Augmented Generation* – https://www.manning.com/books/a-simple-guide-to-retrieval-augmented-generation
- *Essential GraphRAG* – https://www.manning.com/books/essential-graphrag

5.1 The problem of using internal data

In chapter 3, we implemented web search through the Tavily API. When users ask questions, we find relevant information from the web, add it to the context, and have the LLM generate answers. What is the essence of web search? It's extracting only a few relevant results from billions of web pages based on a search query. Search engines like Google or Bing handle this complex task for us, so we can obtain the information we need with just a simple search query.

However, in real work environments, there are many problems that web search cannot solve. These are cases where we need to utilize data that isn't publicly available, such as internal company documents, project codebases, and personal files. These internal data sources require the same principle as web search: we need to find only the parts relevant to the current question from the vast amount of data and put them into the context. We'll implement search functionality so that agents can effectively utilize internal data.

5.1.1 The simple case: Single file

Let's start with the simplest situation. The GAIA benchmark includes problems with attached files. For example, suppose a CSV file is given with the question "What month had the highest sales in this data?"

The solution here is straightforward. Just read the file and put the entire thing into the context. If the file is a few hundred lines, it fits easily within modern LLMs' context windows. We can solve such problems by simply adding a file reading tool to the agent we implemented in chapter 4.

5.1.2 What if there are multiple files?

The GAIA benchmark also includes problems with attached zip files rather than single files. When you extract them, multiple files and folders emerge. The problem is that we don't know in advance which file contains the answer.

For example, suppose you're given a code archive along with the question "What type of database is used in this project?"

When you extract the provided zip file, you get this structure:

```
project/
├── README.md
├── requirements.txt
├── src/
│   ├── main.py
│   ├── database.py
│   └── utils.py
├── config/
│   └── settings.json
└── docs/
    └── architecture.md
```

The answer might be in requirements.txt, or in database.py, or in settings.json. Reading all files and putting them into the context is inefficient, and becomes impossible as the number of files grows.

What's needed in this situation is exploration. First, understand the overall structure, then use file names and folder names as clues to select seemingly relevant files, and examine their contents. If needed, explore additional files. This is identical to how developers explore new codebases. Code projects are particularly suited to this approach because they have systematic structures. Tools like Claude Code, Cursor, and Gemini CLI understand code in exactly this way. This is structure-based search.

5.1.3 What if the data is large or extensive?

There are situations that structure-based exploration cannot handle. First, when a single file is too large to fit in the context. Log files with tens of thousands of lines or large documents fall into this category. Second, when the sheer number of documents is too great. If a company wiki has thousands of documents, exploring them one by one isn't realistic. Also, information in company wikis is often not systematically organized.

Therefore, we need a mechanism to selectively retrieve only the necessary parts. For a question about "how to connect to the database," instead of putting tens of thousands of lines of code into the context, we should find only the database-related portions. This is why search technologies like vector search or keyword search are necessary.

5.2 Types of search methods

RAG (Retrieval-Augmented Generation) is exactly what its name suggests: a method that searches (Retrieval) for necessary information, adds (Augmented) it to the LLM input, and then generates (Generation) a response. Rather than the LLM answering solely from its own knowledge, it references information found from external data to formulate its response.

Figure 5.2 shows the overall flow of RAG. When a user question comes in, the system first searches for relevant information from the Knowledge Base. The search results are passed to the LLM along with the original question, and the LLM generates a response using this information as reference. Through this process, RAG can overcome the LLM's knowledge limitations and leverage up-to-date information or private data.

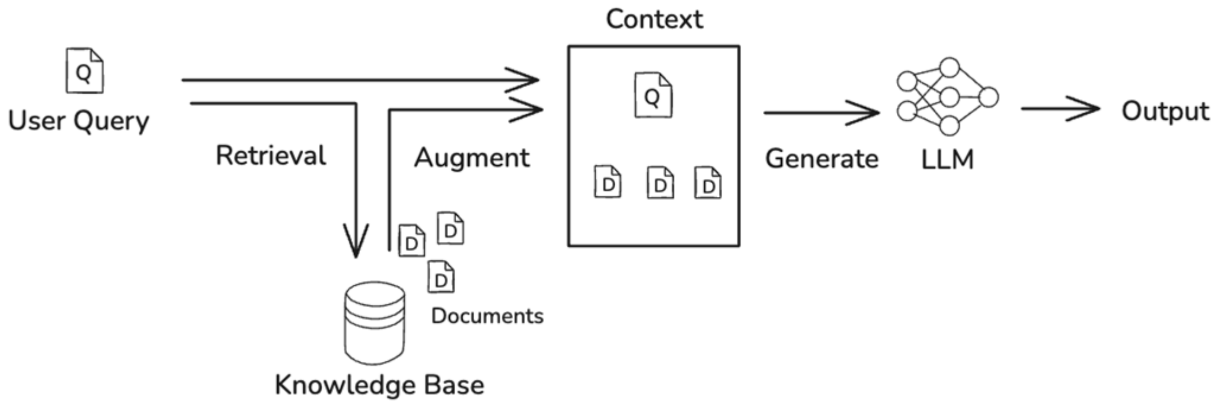


Figure 5.2 The overall flow of RAG (Retrieval-Augmented Generation). When a user question comes in, the system searches for relevant information from the Knowledge Base, then passes the search results along with the question to the LLM to generate a response.

We already experienced RAG through web search, where the LLM resolved tasks by finding relevant information with search queries and adding those results to its context to generate answers. In this broad sense, RAG encompasses all forms of information retrieval and utilization. The difference in this chapter is that we'll build search tools ourselves rather than using pre-built ones.

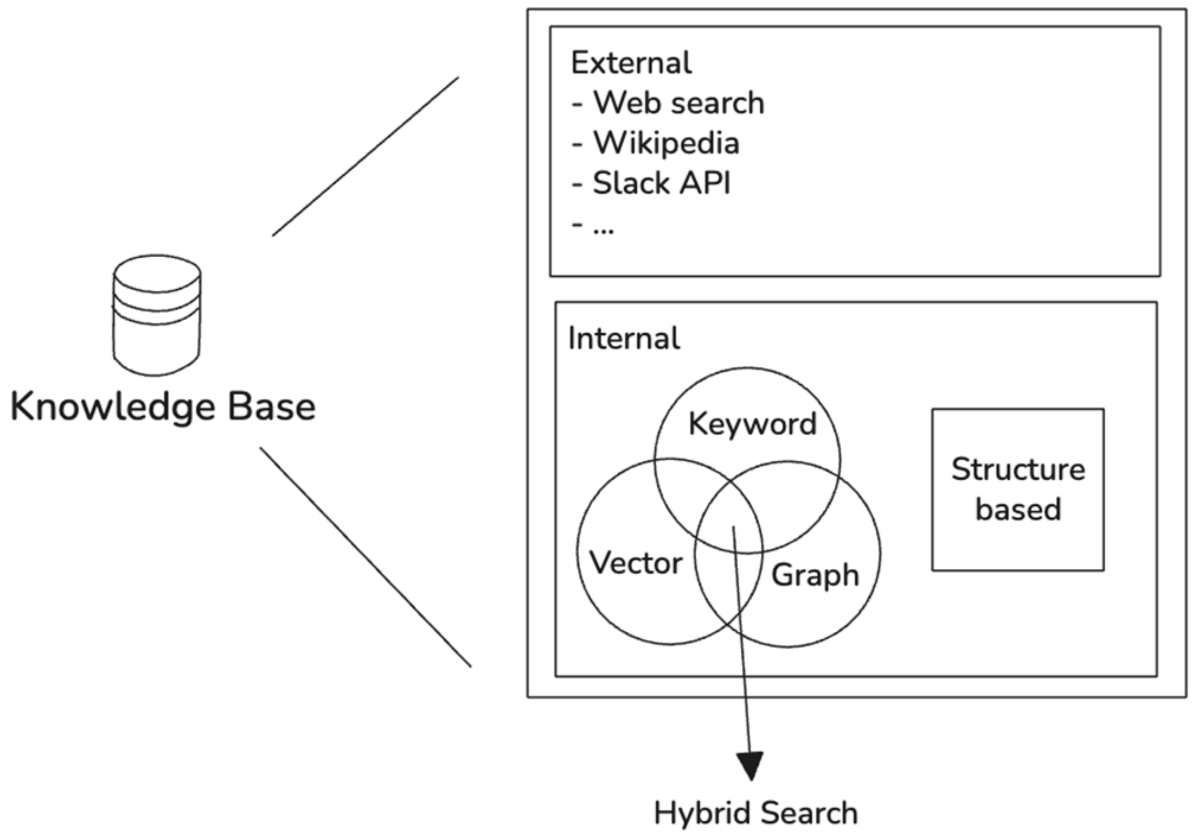


Figure 5.3 Types of Search - External/Internal repositories, Keyword/Vector/Graph search, and Structure-based search.

There are several types of search methods, as illustrated in figure 5.3. The appropriate method varies depending on data characteristics and question types. Different approaches are effective when exact word matching is needed, when searching for semantically similar content, when traversing relationships, or when directly examining file structures. In this section, we'll explore the principles of four major search methods and when each is most suitable.

5.2.1 Keyword search

Keyword search is the most traditional search method, evaluating relevance by calculating the frequency and importance of words contained in documents. Representative algorithms include BM25 and TF-IDF.

TF-IDF (Term Frequency-Inverse Document Frequency) combines two metrics. It multiplies how frequently a specific word appears within a document (TF) by how rare that word is across the entire document collection (IDF). Words like "the" that appear everywhere have low importance, while words that appear intensively only in specific documents have high importance. BM25 is an improved algorithm over TF-IDF that provides more accurate results by adjusting for document length and reflecting the saturation effect of word frequency.

Keyword search is most effective when finding exact words or phrases. There's no method faster or more accurate than keyword search when searching for a function name like `get_user_by_id`, finding error code `404`, or checking a specific configuration value like `MAX_CONNECTIONS`. As a technology proven over decades, it guarantees fast search speeds even on large-scale datasets.

However, keyword search has fundamental limitations. It cannot handle synonyms or similar expressions. Figure 5.4 illustrates this problem. "Sofa" and "couch" refer to the same furniture, but keyword search doesn't know these two are related. Searching for "sofa" won't find documents containing "couch." Without building a separate synonym dictionary, this semantic connection cannot be automatically recognized.

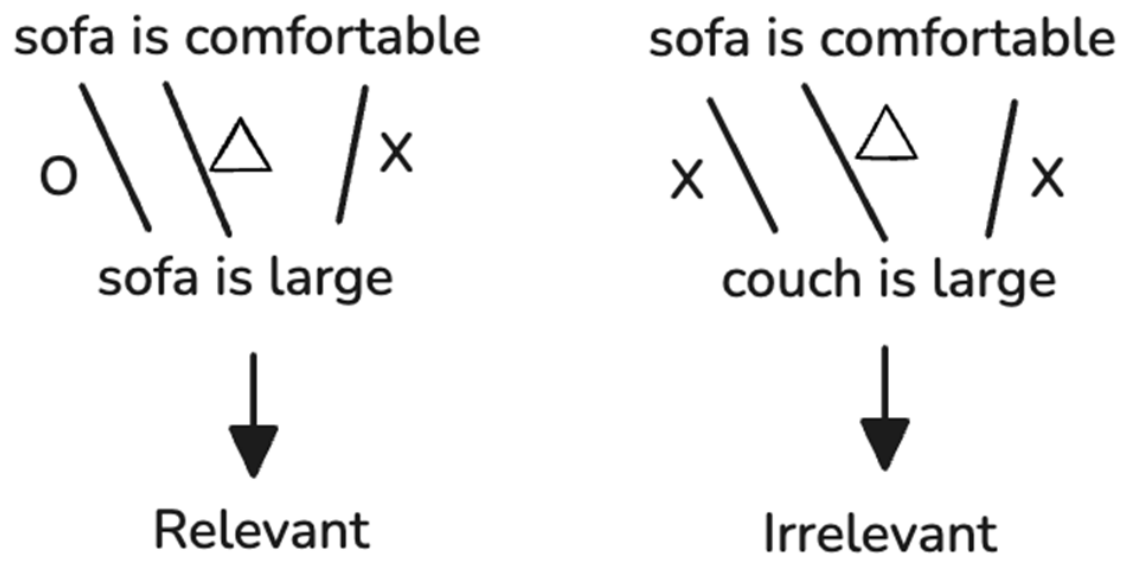


Figure 5.4 Keyword Search - While "sofa" and "couch" are similar words, a keyword search cannot determine their similarity without separate processing. Common expressions like "is" that appear frequently everywhere are not considered important (shown with triangles). The left example is judged relevant due to overlapping "sofa," but the right example is judged irrelevant despite semantic relevance.

5.2.2 Vector search

Vector search is a method that quantifies the meaning of text and calculates similarity. First, it uses an embedding model to transform text into a point in a high-dimensional vector space. In this process, texts with similar meanings are mapped to nearby positions, while texts with different meanings are mapped to distant positions.

Embedding means representing words or sentences as arrays of hundreds to thousands of numbers. For example, the word "cat" might become a vector like [0.2, -0.5, 0.8, ...]. The key point is that this transformation preserves meaning. "Cat" and "kitten" are different words but have similar meanings, so they're placed at nearby positions in vector space. In contrast, "cat" and "car" are similar in spelling but different in meaning, so they're placed far apart.

Figure 5.5 shows the principle of vector search. Documents are pre-converted to vectors and stored, and when searching, the query is also converted to a vector in the same way. Then the system calculates distances between the query vector and stored document vectors to find and return the closest ones. Distance calculation typically uses cosine similarity or Euclidean distance. Cosine similarity is the cosine value of the angle between two vectors, where values closer to 1 indicate more similarity and values closer to 0 indicate less relevance. Euclidean distance is the straight-line distance between two points, where smaller values indicate more similarity.

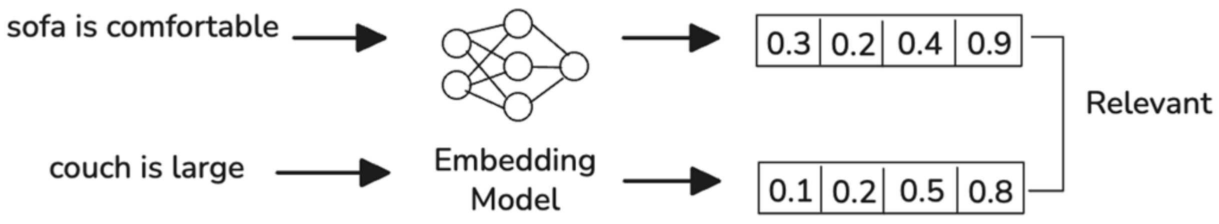


Figure 5.5 Vector Search Method - Determines relevance by converting data into vectors. Using a well-trained embedding model, it can determine that "sofa" and "couch" are similar without synonym processing.

The biggest advantage of vector search is that natural language questions can be used directly. If you ask "how to store information in a database," semantically related code like "persist data to storage" can be found even without the exact same phrase. The synonym handling that was problematic in keyword search is also naturally resolved. This is because the embedding model has already learned during training that "sofa" and "couch" are used in similar contexts.

Modern embedding models also consider context. The same word is converted to different vectors depending on the situation. "Bank" becomes different vectors when referring to a financial institution versus a riverbank. This context-aware characteristic enables more accurate semantic search.

However, vector search isn't always the best choice. When exact word matching is needed, for example when finding specific function names or variable names, keyword search is more effective. Keyword search is also advantageous when searching for special characters or code syntax. For these reasons, hybrid search combining keyword and vector search is widely used in practice.

In section 5.3, we'll examine how vector search works in detail by implementing embeddings, chunking, and vector databases ourselves.

5.2.3 Graph search

Graph search is a method that represents and traverses information as a graph structure consisting of nodes (entities) and edges (relationships). Figure 5.6 shows a simple example. From the sentence "Alice writes Python code," "Alice" and "Python code" become nodes, and "writes" becomes an edge connecting them. Once the graph is built, questions like "What does Alice write?" or "Who writes Python code?" can be answered by traversing the graph.

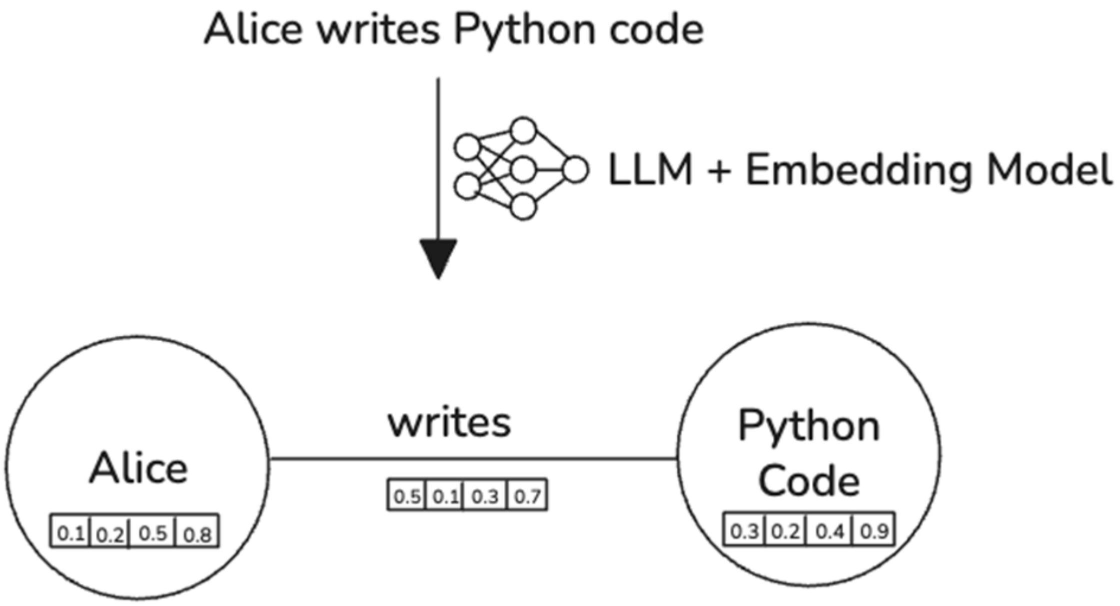


Figure 5.6 Graph Search Method - Extracts entities and relationships between entities from sentences and represents them as a graph. Each entity and relationship is sometimes converted to embedding vectors to utilize both vector search and graph search together.

Graph search excels at complex relationship-based questions. Multi-step relationship queries like "all code that calls this function," "other code written by developers who participated in this project," or "files recently modified by someone who has collaborated with A" can be effectively handled through graph traversal. It's useful for code dependency analysis, organizational structure understanding, and knowledge graph exploration.

The biggest challenge in graph search is extracting entities and relationships from unstructured text. In the past, complex rules or machine learning models were required, but recent advances in LLMs have greatly simplified this process. By instructing an LLM to "extract people, organizations, and places mentioned in this document and identify relationships between them," a knowledge graph can be automatically constructed. Thanks to this automation, graph search is receiving renewed attention.

However, graph search has the disadvantage of high construction costs and requiring specialized knowledge in graph modeling. Considerations such as which entities and relationships to extract and how to design the graph schema are necessary. This book does not cover graph search directly. Interested readers should refer to Manning's Essential GraphRAG (https://www.manning.com/books/essential-graphrag).

5.2.4 Structure-based search

Structure-based search is a method that leverages the structure itself for exploration when data is already systematically organized. A representative example is code repositories. Code is systematically organized into folders and files, and each file has clear structures like classes, functions, and variables.

Consider how a developer explores a new codebase. First, they use the `tree` command to grasp the overall folder structure. They check top-level directories like `src/`, `tests/`, and `docs/`, then examine the substructure of modules of interest. Next, they use `grep` or IDE search functionality to find specific functions or classes. They can quickly find where the `get_user_by_id` function is defined and where it's called. Finally, they open the found files and read the entire code to understand the detailed implementation.

This exploration method is effective because the code structure itself contains information. File names (`user_service.py`), folder structures (`/services/auth/`), and function names (`authenticate_user`) are all meaningful metadata. Without complex search algorithms, the desired code can be found using just this structural information.

Modern coding tools like Claude Code, Cursor, and Gemini CLI use exactly this approach. Because LLMs are familiar with bash commands like `tree`, `grep`, and `cat`, agents can explore codebases like human developers. Like a human developer, they first grasp the structure, search for related files, read necessary parts, and gradually expand their understanding.

Structure-based search is also suitable for GAIA's zip file problems discussed in section 5.1. When the archive is extracted, multiple files and folders emerge. The agent goes through a process of first grasping the structure, getting hints from file names, selecting related files, and confirming their contents. We'll implement this approach directly in section 5.4.

GUIDE FOR CHOOSING SEARCH METHODS

The four search methods we've examined each have different suitable situations. Table 5.1 summarizes which method to choose for each situation.

Table 5.1 Recommended search methods by situation

Situation	Recommended Method	Reason
Exact word/code search	Keyword Search	Exact matching needed for `get_user_by_id`, error code `404`, etc.
Natural language questions	Vector Search	Semantic-based search needed for queries like "how to store data"
Relationship-based questions	Graph Search	Relationship traversal needed for "code that calls this function"
Folder/file structure exploration	Structure-based Search	When structure itself is information, like zip files or codebases
Production environments	Hybrid Search	Combining advantages of multiple methods

In practice, using a single method alone is rare. Hybrid search is common. The most representative approach combines keyword search and vector search. Keyword search captures exact matches while vector search captures semantic similarity. The scores from both search results are normalized and a weighted average is calculated to determine the final ranking. This way, both exact names like the `get_user_by_id` function and semantic questions like "how to get user information" can be handled.

In this book, we'll implement vector search and structure-based search directly. Vector search has received the most attention as a search method since the emergence of LLMs, and structure-based search is garnering significant interest centered on coding agents as LLM performance and context windows have grown larger. We only introduced the concepts of keyword search and graph search, but they can be additionally integrated into the system we build in this book as needed.

5.3 Practicing vector search

The core of vector search is converting text to vectors and calculating distances. Let’s convert text to vectors and observe how semantic similarity is expressed as distance. We'll implement embedding, chunking, and vector search in sequence, then conclude with a hands-on exercise integrating these techniques to extract relevant information from web search results.

5.3.1 Embedding: Converting text to vectors

Embedding is the process of converting text into numerical arrays with hundreds to thousands of dimensions. The key insight is that semantically similar texts are placed close together in vector space. Embedding models learn to represent these semantic relationships as vector distances through training on massive amounts of text.

Let's implement a function that converts text to embedding vectors using OpenAI's text-embedding-3-small model. This model offers excellent price-to-performance ratio, making it suitable for learning. The function below wraps single strings in a list for processing and extracts each embedding vector from the API response, returning them as a numpy array.

Listing 5.1 Implementing the get_embeddings function

```
from openai import OpenAI
import numpy as np

client = OpenAI()

def get_embeddings(texts, model="text-embedding-3-small"):
    """Convert text to embedding vectors."""
    if isinstance(texts, str):
        texts = [texts]  #A

    response = client.embeddings.create(input=texts, model=model)  #
B
    return np.array([item.embedding for item in response.data])  #C
```

#A Wrap single string in list for uniform processing
#B Call OpenAI embedding API
#C Extract embedding vectors as numpy array

Now let's verify that embeddings properly capture semantic similarity. We prepare three sentences: the first about a cat sleeping on a sofa, the second about a kitten playing with a toy, and the third about a dog running in a park. The `cosine_similarity` function calculates the similarity between the first sentence and the other two. Cosine similarity is the cosine of the angle between two vectors: values closer to 1 indicate high similarity, while values closer to 0 indicate low similarity.

Listing 5.2 Comparing embedding similarity

```
from sklearn.metrics.pairwise import cosine_similarity

sentences = [
    "The cat is sleeping on the couch",
    "A kitten is playing with a toy",
    "The dog is running in the park"
]
embeddings = get_embeddings(sentences)

cat_kitten = cosine_similarity([embeddings[0]], [embeddings[1]])[0]
[0]  #A
cat_dog = cosine_similarity([embeddings[0]], [embeddings[2]])[0][0]
#A

print(f"Cat vs Kitten: {cat_kitten:.3f}")
print(f"Cat vs Dog: {cat_dog:.3f}")
```

#A Calculate cosine similarity between embedding vectors

The results show that similarity between "cat" and "kitten" sentences is higher than between "cat" and "dog" sentences. Although "cat" and "kitten" are different words, they're semantically related. Keyword search would have treated them as completely different documents. This is the solution to the "sofa vs couch" problem described earlier. Vector search can find semantically related content without exact word matching.

5.3.2 Chunking: Dividing long text into search units

Embedding an entire document as a single vector causes several problems. First, it may exceed the embedding model's token limit. Second, long documents contain multiple topics, and the vector becomes "averaged out," reducing search accuracy. For example, if a manual containing "installation instructions," "troubleshooting," and "product specifications" is converted to a single vector, searching for "installation instructions" might not surface that document in top results. Third, returning entire documents as search results wastes context. The solution is to split documents into small units (chunks) and embed each chunk separately.

Chunk size involves tradeoffs. Too small, and context information is lost: you can't know what "this" refers to. Too large, and multiple topics mix together, reducing search accuracy. Generally, 200 to 1,000 tokens is appropriate, roughly 150 to 750 words in English. The optimal chunk size depends on document characteristics. Information-dense text like technical documentation benefits from smaller chunks, while narrative writing benefits from larger chunks to maintain context.

An important concept in fixed-length chunking is overlap. Simply cutting text every 500 characters can break context at chunk boundaries. For example, if the sentence "This function processes data. The processed result is" gets cut at a chunk boundary, the first chunk doesn't know what "the processed result" is, and the second chunk doesn't know what "this function" refers to. Overlap allows adjacent chunks to share some text, mitigating this problem. Typically, overlap is set to 10-20% of the chunk size.

The function below implements fixed-length chunking. The while loop cuts text from the `start` position by `chunk_size` to create chunks. The key part is calculating the next start position: if we haven't reached the end of the document, we set it to `end - overlap` to go back by the overlap amount. At the document's end, we use `end` as-is to terminate the loop.

Listing 5.3 Fixed-length chunking function

```
def fixed_length_chunking(text, chunk_size=500, overlap=50):
    """Split text into fixed-length chunks."""
    chunks = []
    start = 0

    while start < len(text):
        end = start + chunk_size
        chunk = text[start:end].strip()
        if chunk:
            chunks.append(chunk)
        start = end - overlap if end < len(text) else end  #A

    return chunks
```

#A Move back by overlap amount to maintain context between chunks

A simple test demonstrates the effect of overlap. Splitting 283-character text with `chunk_size` 100 and `overlap` 20 produces more chunks (4) than without overlap (3). Thanks to overlap, each chunk shares 20 characters with adjacent chunks, maintaining context connections.

Listing 5.4 Implementing vector search

```
sample = "A" * 283
chunks = fixed_length_chunking(sample, chunk_size=100, overlap=20)
print(f"Original: {len(sample)} chars → {len(chunks)} chunks")
```

5.3.3 Implementing vector search

So far we've learned how to convert text to embeddings and split long text into chunks. Now let's implement actual search. The goal is to find the chunks most similar to a query. The core logic is surprisingly simple.

The function below operates in four steps. First, it converts the query to a vector. Then `cosine_similarity` calculates similarity between the query vector and all chunk vectors at once. `argsort()[::-1]` sorts indices by descending similarity, selecting the top `top_k`. Finally, it returns dictionaries pairing each chunk with its similarity score.

Listing 5.5 Implementing vector search

```
def vector_search(query, chunks, chunk_embeddings, top_k=3):
    """Find the most similar chunks to the query."""
    query_embedding = get_embeddings(query)
    similarities = cosine_similarity(query_embedding, chunk_embeddin
gs)[0]
    top_indices = similarities.argsort()[::-1][:top_k]

    results = []
    for idx in top_indices:
        results.append({
            'chunk': chunks[idx],
            'similarity': similarities[idx]
        })
    return results
```

Let's verify the operation with a simple test. We prepare four short documents, generate their embeddings, and search for "AI and coding." We call `get_embeddings` to pre-compute all document embeddings, then pass the query along with them to `vector_search`.

Listing 5.6 Testing vector search

```
documents = [
    "Python is a programming language",
    "Machine learning uses Python extensively",
    "Cats are popular pets",
    "Deep learning is a subset of machine learning"
]

doc_embeddings = get_embeddings(documents)

results = vector_search("Artificial Intelligence", documents, doc_em
beddings, top_k=4)
for r in results:
    print(f"{r['similarity']:.3f}: {r['chunk']}")
```

The output is as follows

```
0.353: Deep learning is a subset of machine learning
0.307: Machine learning uses Python extensively
0.162: Python is a programming language
0.048: Cats are popular pets
```

Searching for "Artificial Intelligence" surfaces the "deep learning" and "machine learning" documents at the top. Keyword search would have failed to find documents lacking the exact words "artificial" or "intelligence." This is the essence of vector search: semantic search is possible with just embeddings and similarity calculations, without complex algorithms.

WHY VECTOR DATABASES ARE NECESSARY

The approach we've implemented has limitations. Since it calculates similarity with all chunks, time complexity is *O(n).* With a million documents, every search requires a million calculations. Vector databases solve this problem. They support fast search through ANN (Approximate Nearest Neighbor) algorithms, store embeddings efficiently, and provide metadata-based filtering. Popular vector databases include ChromaDB, Pinecone, Weaviate, and Qdrant. For this exercise, our direct implementation suffices since we're handling small-scale data, and understanding the principles is our goal.

Major vector databases and their features:

- **Chroma**: A lightweight database that can run locally, suitable for prototyping and small-scale projects. It requires no separate server setup.

  https://www.trychroma.com/

- **Pinecone**: A fully managed cloud service with excellent scalability but requires payment. Choose this when you want to reduce infrastructure management burden in large-scale production environments.

  https://www.pinecone.io/
- **Weaviate**: Open-source while providing enterprise-grade features. Supports GraphQL API, hybrid search, and integration with various embedding models.

  https://weaviate.io/
- **Qdrant**: Written in Rust with excellent performance and powerful filtering capabilities. Supports both on-premises and cloud.

  https://qdrant.tech/
- **Milvus**: Optimized for large-scale vector processing and supports GPU acceleration. Suitable for large-scale systems handling billions of vectors.

  https://milvus.io/ko

5.3.4 Exercise: Finding relevant information from web search results

Let's integrate the embeddings, chunking, and vector search we've learned. The goal is to find only the parts related to a specific topic from actual web search results. This exercise lets you directly experience vector search's semantic matching capability.

EXERCISE SCENARIO

Searching the web for "2025 Nobel Prize" returns mixed results across physics, chemistry, medicine, and other fields. What if you want to find only content related to "quantum computing"? Let's see if we can find related content even when the exact phrase "quantum computing" doesn't appear in the search results.

STEP 1: PREPARE WEB SEARCH RESULTS

Search Tavily for "2025 Nobel Prize" and collect the results. The key parameter here is `include_raw_content=True`, which retrieves the full page content rather than just snippets. We iterate through results, keeping only those with `raw_content` available, and store the title, full content, and URL for each.

Listing 5.7 Fetching web search results

```
from tavily import TavilyClient

tavily = TavilyClient()
response = tavily.search(
    "2025 Nobel Prize winners",
    max_results=10,
    include_raw_content=True
)

search_results = []
for result in response['results']:
    if result.get('raw_content'):
        search_results.append({
            'title': result['title'],
            'content': result['raw_content'],
            'url': result['url']
        })
```

STEP 2: CHECK TOKEN COUNT

How many tokens would it consume to feed web search results directly to an LLM? We use tiktoken's `encoding_for_model` to get an encoder for gpt-5, then convert text to a token list with the `encode` method and measure its length.

Listing 5.8 Counting tokens in search results

```
import tiktoken

enc = tiktoken.encoding_for_model("gpt-5")
full_text = "\n\n".join([
    f"Title: {r['title']}\n{r['content']}"
    for r in search_results
])
total_tokens = len(enc.encode(full_text))
print(f"Total characters: {len(full_text)}")
print(f"Total tokens: {total_tokens}")
```

The output is as follows:

```
Total characters: 136448
Total tokens: 37312
```

If the token count reaches several thousand, putting all of it in context every time is inefficient. Costs increase, and unnecessary information becomes noise that can degrade performance. Extracting only relevant portions can reduce this to just a few hundred tokens.

STEP 3: CHUNKING AND EMBEDDING

Split the full text into chunks and generate embeddings for each. Use `fixed_length_chunking` to create chunks of 500 characters with 50-character overlap, then pass the entire chunk list to `get_embeddings` to obtain all embeddings in a single API call.

Listing 5.9 Chunking and embedding search results

```
all_chunks = []
for result in search_results:
    text = f"Title: {result['title']}\n{result['content']}"
    chunks = fixed_length_chunking(text, chunk_size=500, overlap=50)
    for chunk in chunks:
        all_chunks.append({
            'text': chunk,
            'title': result['title'],
            'url': result['url']
        })

print(f"Total chunks: {len(all_chunks)}")

chunk_texts = [c['text'] for c in all_chunks]
chunk_embeddings = get_embeddings(chunk_texts)
```

STEP 4: EXECUTE VECTOR SEARCH

Search for "quantum computing" to find relevant chunks. Pass the query, chunk list, and chunk embeddings to the `vector_search` function we implemented earlier, requesting the top 3 results.

Listing 5.10 Executing vector search on chunks

```
query = "quantum computing"
results = vector_search(query, chunks, chunk_embeddings, top_k=3)

print(f"Query: '{query}'\n")
print("=" * 60)
for i, r in enumerate(results, 1):
    print(f"\n[{i}] Similarity: {r['similarity']:.3f}")
    print(f"{r['chunk'][:300]}...")
```

The output shows the top 3 chunks most semantically similar to "quantum computing":

```
Query: 'quantum computing'

[1] Similarity: 0.596
Content about quantum mechanics as the foundation of digital technol
ogy...

[2] Similarity: 0.579
Content about quantum computing's economic potential reaching $2 tri
llion by 2035...

[3] Similarity: 0.566
Content about Physics laureates working on quantum experiments...
```

The results demonstrate that vector search successfully identified quantum-related content across different contexts: foundational physics, economic impact, and Nobel Prize winners, all with similarity scores above 0.5.

STEP 5: TOKEN SAVINGS EFFECT

Let's check how many tokens we save by using only the top 3 chunks. We concatenate the selected chunks, calculate the token count, and output the savings rate compared to total tokens.

Listing 5.11 Calculating token savings

```
top_chunks = [r['chunk'] for r in results]
selected_text = "\n\n".join(top_chunks)
selected_tokens = len(enc.encode(selected_text))

print(f"Total tokens: {total_tokens}")
print(f"Selected tokens: {selected_tokens}")
print(f"Savings rate: {(1 - selected_tokens/total_tokens)*100:.1
f}%")
The output shows a dramatic reduction in token usage:
Total tokens: 37312
Selected tokens: 304
Savings rate: 99.2%
```

By extracting only the three most relevant chunks, we reduced context from over 37,000 tokens to just 304, achieving a 99.2% savings rate while preserving the information most relevant to "quantum computing."

KEY INSIGHTS

This exercise confirmed two key points. First, vector search finds semantically related content without exact keyword matching. Second, extracting only relevant portions instead of feeding entire web search results into context significantly saves tokens.

In practice, LLM-optimized search services like Tavily already implement similar functionality internally. By using `search_depth="advanced"` along with the `chunks_per_source` parameter, you can achieve comparable results without building your own chunking and vector search pipeline:

```
response = tavily.search(
    "2025 Nobel Prize winners",
    search_depth="advanced",
    chunks_per_source=3
)
```

This returns pre-chunked, relevance-filtered content directly from the API. However, understanding the underlying mechanics, as we've done in this exercise, helps you customize the approach for specific use cases and troubleshoot when built-in solutions fall short. In section 5.5, we'll cover how to apply this technique to Callbacks for automatically compressing web search results.

5.3.5 Structure-based search

Exploration is necessary when there are multiple files. Extracting a compressed file reveals multiple folders and files, and we cannot know in advance which file contains the answer. In such situations, there is an effective approach that doesn't require complex search algorithms: leveraging the structure of the file system itself.

GAIA benchmark problems with zip files can be approached in the same way. Extracting the archive reveals multiple files and folders. The agent first examines the structure, picks relevant files based on hints from filenames, and then checks the contents. Let’s implement this process directly.

The tools we will implement in this section are as follows:

Tool	Purpose
unzip_file	Extract zip files
list_files	Examine folder structure
read_file	Read various file formats including text, CSV, Excel files
read_media_file	Analyze image, audio and PDF files using LLM

Let's start by learning how to prepare the GAIA dataset with attached files.

5.3.6 Preparing the GAIA dataset

Some GAIA benchmark problems come with files attached alongside the question text. These include various formats such as zip, pdf, xlsx, images, and audio. The agent must open and analyze these files to find the correct answer. In this section, we use problems with attachments to practice structure-based search.

In chapter 2, we loaded the GAIA dataset using `load_dataset()`. However, this approach only retrieves dataset and does not include attached files. To download the attached files, we need to use `snapshot_download()`.

The following code loads the GAIA dataset and downloads the attached files. `load_dataset()` provides metadata such as question text and answers as a Dataset object, while `snapshot_download()` saves the attached files locally. We set `local_dir` to the project root so the cache can be reused across different chapters.

Listing 5.12 Loading GAIA dataset with attachments

```
from datasets import load_dataset
from huggingface_hub import snapshot_download
from pathlib import Path

# Load dataset (same as Chapter 2)
dataset = load_dataset("gaia-benchmark/GAIA", "2023_all", split="val
idation")

# Download attached files
NOTEBOOK_DIR = Path.cwd()
PROJECT_ROOT = NOTEBOOK_DIR.parent
CACHE_DIR = PROJECT_ROOT / "gaia_cache"

snapshot_download(
    repo_id="gaia-benchmark/GAIA",
    repo_type="dataset",
    allow_patterns="2023/validation/*",
    local_dir=CACHE_DIR
)
```

When an agent extracts or modifies files, it becomes difficult to restore the original state. To solve this problem, we implement a function that copies from the cache to a working folder. Calling this function before each problem ensures we always start from the same initial state.

Listing 5.13 Resetting workspace function

```
import shutil

WORK_DIR = NOTEBOOK_DIR / "gaia_workspace"

def reset_workspace():
    """Restore the workspace to its initial state."""
    shutil.rmtree(WORK_DIR, ignore_errors=True)
    shutil.copytree(CACHE_DIR / "2023/validation", WORK_DIR)
    print(f"Workspace reset: {WORK_DIR}")

reset_workspace()
```

Let's check which problems have attached files.

Listing 5.14 Finding problems with file attachments

```
problems_with_files = [p for p in dataset if p.get('file_name')]
problem_with_zip = [p for p in problems_with_files if p['file_nam
e'].endswith('.zip')]

print(f"Total problems: {len(dataset)}")
print(f"Problems with attachments: {len(problems_with_files)}")
print(f"Total problems with zip files: {len(problem_with_zip)}")

problem = problem_with_zip[0]
print(f"Question: {problem['Question'][:100]}...")
print(f"File name: {problem['file_name']}")

file_path = WORK_DIR / problem['file_name']
print(f"File exists: {file_path.exists()}")
```

The `file_name` field stores the attachment filename. Combining it with the working directory path gives us the actual file path. Now let's implement the tools that allow our agent to explore and read these files.

5.3.7 Implementing file system tools

Think about how a developer explores a new codebase. They first use the `tree` command to understand the overall structure. They pick relevant-looking files based on hints from folder names and filenames. They open files to check the contents and explore additional files as needed. Coding agents like Claude Code, Cursor, and Gemini CLI operate in exactly the same way.

UNZIP_FILE: EXTRACT ARCHIVES

When a GAIA problem includes a zip file attachment, the first thing the agent must do is extract it. This can be implemented using the standard library `zipfile`. If no extraction path is specified, the function creates a folder with the zip filename and returns a list of extracted files so the agent can plan its next steps.

Listing 5.15 Implementing unzip_file tool

```
import zipfile
from pathlib import Path

def unzip_file(zip_path: str, extract_to: str = None) -> str:
    """Extract a zip file to the specified directory."""
    zip_path = Path(zip_path)

    if not zip_path.exists():
        return f"File not found: {zip_path}"

    # Default extraction path: create folder with zip filename
    if extract_to is None:
        extract_to = zip_path.parent / zip_path.stem
    else:
        extract_to = Path(extract_to)

    extract_to.mkdir(parents=True, exist_ok=True)

    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        file_list = zip_ref.namelist()
        zip_ref.extractall(extract_to)

    # Format results
    result = f"Extracted {len(file_list)} files to {extract_to}/\n
\n"
    result += "Contents:\n"
    for f in file_list[:20]:
        result += f"  - {f}\n"
    if len(file_list) > 20:
        result += f"  ... and {len(file_list) - 20} more files\n"

    return result
```

LIST_FILES: EXAMINE FOLDER STRUCTURE

After extracting, we need to see what files exist. The `list_files` tool returns a list of files and folders at the specified path. Directories are distinguished with a `/` suffix, and files include size information. Hidden files are excluded, and directories are sorted first to make the structure easier to understand.

Listing 5.16 Implementing list_files tool

```python
from pathlib import Path

def list_files(path: str = ".") -> str:
    """List files and directories in the given path."""
    path = Path(path)

    if not path.exists():
        return f"Path not found: {path}"

    if not path.is_dir():
        return f"Not a directory: {path}"

    items = []
    for item in sorted(path.iterdir()):
        if item.name.startswith('.'):
            continue

        if item.is_dir():
            items.append(f"{item.name}/")
        else:
            items.append(f"{item.name}")

    # Sort directories first
    dirs = [i for i in items if i.endswith('/')]
    files = [i for i in items if not i.endswith('/')]

    result = f"Directory: {path}\n"
    for item in dirs + files:
        result += f"  {item}\n"

    return result
```

READ_FILE: READING VARIOUS FILE FORMATS

Once we understand the file structure, we need to read the contents. GAIA problems include attachments in various formats beyond text files, including CSV, Excel, and PDF. The `read_file` tool checks the extension and reads the file appropriately, returning the content as text.

The main function routes to the appropriate handler based on the file extension.

Listing 5.17 Implementing read_file tool

```
from pathlib import Path

TEXT_EXTENSIONS = ['.txt', '.py', '.js', '.json', '.md', '.html',
                   '.css', '.xml', '.yaml', '.yml', '.log', '.sh']
SPREADSHEET_EXTENSIONS = ['.xlsx', '.xls', '.csv']

def read_file(file_path: str, start_line: int = 1, end_line: int = -
1) -> str:
    """Read file content. Supports txt, py, json, md, csv, xlsx."""
    path = Path(file_path)

    if not path.exists():
        return f"File not found: {file_path}"

    ext = path.suffix.lower()

    if ext in TEXT_EXTENSIONS:
        return _read_text_file(file_path, start_line, end_line)
    elif ext == '.csv':
        return _read_csv(file_path)
    elif ext in SPREADSHEET_EXTENSIONS:
        return _read_excel(file_path)
    else:
        return _read_text_file(file_path, start_line, end_line)
```

Text files are returned with line numbers. The `start_line` and `end_line` parameters allow reading only specific ranges, which is useful for checking just the needed parts of long files.

Listing 5.18 Reading text files with line numbers

```
def _read_text_file(file_path: str, start_line: int, end_line: int)
-> str:
    with open(file_path, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    # Adjust line numbers (1-indexed to 0-indexed)
    start_idx = max(0, start_line - 1)
    end_idx = len(lines) if end_line == -1 else min(end_line, len(li
nes))

    selected_lines = lines[start_idx:end_idx]

    result = []
    for i, line in enumerate(selected_lines, start=start_line):
        result.append(f"{i:4d} | {line.rstrip()}")
    return '\n'.join(result)
```

CSV and Excel files are read with pandas and converted to markdown tables. The markdown format makes it easy for LLMs to understand tabular structure.

Listing 5.19 Reading CSV and Excel files

```
import pandas as pd

def _read_csv(file_path: str) -> str:
    df = pd.read_csv(file_path)
    return df.to_markdown(index=False)

def _read_excel(file_path: str) -> str:
    df = pd.read_excel(file_path)
    return df.to_markdown(index=False)
```

READ_MEDIA_FILE: ANALYZING IMAGES AND AUDIO

Some GAIA problems include image, audio, or PDF file attachments. These files require LLM analysis rather than simple text extraction. We separate this from read_file because media analysis always requires a specific question: "What should I look for?"

The main function routes to the appropriate handler based on file extension. We'll examine the PDF handler in detail since it combines text extraction with visual analysis.

Listing 5.20 Implementing read_media_file tool

```
import fitz  # pymupdf
import base64
from pathlib import Path
from openai import OpenAI

IMAGE_EXTENSIONS = ['.png', '.jpg', '.jpeg', '.gif', '.webp', '.bm
p']
AUDIO_EXTENSIONS = ['.mp3', '.wav', '.m4a', '.flac', '.ogg', '.web
m']
PDF_EXTENSIONS = ['.pdf']

def read_media_file(file_path: str, query: str) -> str:
    """Analyze an image, audio, or PDF file using LLM."""
    ext = Path(file_path).suffix.lower()

    if ext in IMAGE_EXTENSIONS:
        return _analyze_image(file_path, query)
    elif ext in AUDIO_EXTENSIONS:
        return _analyze_audio(file_path, query)
    elif ext in PDF_EXTENSIONS:
        return _analyze_pdf(file_path, query)
    else:
        return f"Unsupported media format: {ext}"

def _analyze_image(file_path: str, query: str) -> str:
    with open(file_path, "rb") as f:
        image_data = base64.b64encode(f.read()).decode("utf-8")

    ext = Path(file_path).suffix.lower().lstrip('.')
    media_type = "image/jpeg" if ext == "jpg" else f"image/{ext}"

    client = OpenAI()
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[{
            "role": "user",
            "content": [
                {"type": "text", "text": query},
                {"type": "image_url", "image_url": {
                    "url": f"data:{media_type};base64,{image_data}"
                }}
            ]
        }]
    )
    return response.choices[0].message.content

def _analyze_audio(file_path: str, query: str) -> str:
    with open(file_path, "rb") as f:
        audio_data = base64.b64encode(f.read()).decode("utf-8")

    ext = Path(file_path).suffix.lower().lstrip('.')

    client = OpenAI()
    response = client.chat.completions.create(
        model="gpt-4o-audio-preview",
        messages=[{
            "role": "user",
            "content": [
                {"type": "text", "text": query},
                {"type": "input_audio", "input_audio": {
                    "data": audio_data,
                    "format": ext
                }}
            ]
        }]
    )
    return response.choices[0].message.content

def _analyze_pdf(file_path: str, query: str) -> str:
    doc = fitz.open(file_path)

    # Extract text for context
    text_content = ""
    for page in doc:
        text_content += page.get_text()

    # Convert pages to images
    images = []
    for page in doc[:5]:  # First 5 pages
        pix = page.get_pixmap(matrix=fitz.Matrix(2, 2))
        img_bytes = pix.tobytes("png")
        images.append(base64.b64encode(img_bytes).decode('utf-8'))

    # Build content with text and images
    content = [{
        "type": "text",
        "text": f"{query}\n\nExtracted text:\n{text_content[:3000]}"
    }]

    for img_b64 in images:
        content.append({
            "type": "image_url",
            "image_url": {"url": f"data:image/png;base64,{img_b64}"}
        })

    client = OpenAI()
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": content}]
    )
    return response.choices[0].message.content
```

The PDF handler combines two approaches: text extraction using pymupdf and page-to-image conversion for visual analysis. Text extraction alone cannot handle tables, diagrams, or scanned documents. By sending both the extracted text and page images to the LLM, the agent can answer questions about any PDF content.

Image analysis uses gpt-4o with base64-encoded image data, while audio analysis uses gpt-4o-audio-preview which supports native audio input. The implementations follow the same pattern as PDF analysis. See the complete code in the repository.

5.3.8 Connecting tools to the agent

Now let's connect the file system tools we implemented to the Agent from chapter 4. We wrap each function with the `@tool` decorator and register them along with the web search tools from chapter 3.

The following code shows how to register the file system tools with the agent. Since file exploration requires multiple sequential steps, we set `max_steps` to 20.

Listing 5.21 Registering file system tools with agent

```
from scratch_agents.tools import tool
from scratch_agents.agent import Agent

# File system tools
@tool
def unzip_file(zip_path: str, extract_to: str = None) -> str:
    """Extract a zip file to the specified directory."""
    # Implementation from Listing 5.15
(...)

# Configure with existing tools
tools = [
    search_web,
    unzip_file,
    list_files,
    read_file,
    read_media_file,
]

agent = Agent(
    model=LlmClient(model="gpt-5"),
    tools=tools,
    instruction="You are a helpful assistant that can search the web 
and "
                "explore files to answer questions.",
    max_steps=20
)
```

Now the agent can explore the file system like a human developer. When given a zip file, it extracts the archive, examines the folder structure, reads relevant files, and finds the answer.

5.3.9 Solving GAIA zip file problems

Let's solve a problem from the actual GAIA benchmark that includes a zip file attachment. We’ll demonstrate the complete exploration process from extraction to finding the answer. First, we reset the workspace and select a problem with a zip file attachment. We include the file path in the prompt so the agent knows where the attachment is located.

Listing 5.22 Preparing GAIA zip file problem

```
# Reset workspace to clean state
reset_workspace()

# Select a problem with zip file attachment
zip_problems = [p for p in problems_with_files if p['file_name'].end
swith('.zip')]
problem = zip_problems[1]

file_path = WORK_DIR / problem['file_name']

# Construct prompt including file location
prompt = f"""{problem['Question']}

The attached file is located at: {file_path}
"""
print(prompt)
```

Now let's run the agent and observe the file structure exploration process.

Listing 5.23 Running agent on zip file problem

```
response = await agent.run(prompt)
print(response)
```

The agent's exploration process follows a systematic pattern. First, it extracts the zip file using unzip_file. Then it calls list_files to examine the folder structure. Two files emerge: Job Listing.pdf and Applicants.xlsx. The filenames alone reveal their relationship clearly —one describes the job requirements, the other lists candidates.

The agent first analyzes the PDF using read_media_file to extract the qualification requirements. Then it reads the Excel file with read_file to review all applicants' data. Finally, it cross-references the requirements against each candidate to find the answer.

The key observation is that the agent navigates file structures without complex search algorithms. Filenames and extensions serve as natural hints. "Job Listing" suggests requirements to match against, while "Applicants" indicates candidate data to filter. The agent uses these naming conventions and format cues to efficiently locate the information needed to solve the problem.

5.4 Extending agents with callbacks

The Agent we implemented in chapter 4 provides basic functionality for executing tools and receiving results. However, in real-world deployments, situations arise where you need to obtain user approval before executing a tool, or compress search results to manage context efficiently. Do we need to modify the Agent's core logic to address these requirements?

Fortunately, we don't. Using the Callback pattern, we can extend the Agent's behavior without touching its core code. In this section, we'll add a Callback mechanism to the Agent and learn how to use it through two practical examples.

5.4.1 The need for agent extension

Let's consider extension points for our agent from a Context Engineering perspective. During the process of the agent calling the LLM and executing tools, there are several points where we can intervene.

Intervening before an LLM call allows us to inject additional information into the context, compress old messages, or check for cached responses. After the LLM responds, we can validate or post-process the results. Before tool execution, we can request user approval for dangerous operations or return cached results. After tool execution, we can compress results or mask sensitive information. Finally, after the entire execution completes, we can save the conversation or generate summaries.

All of this is possible through Callbacks. Callbacks are user-defined functions that are invoked at specific execution points. They enable us to add various capabilities without modifying the agent's core logic.

The types of Callbacks we'll use are as follows, as shown in figure 5.7.

`before_llm_callback` and `after_llm_callback` execute before and after LLM calls. They're used for manipulating context or intercepting responses.

`before_tool_callback` and `after_tool_callback` execute before and after tool execution. These are the callbacks we'll focus on in this section. They're used for approving tool execution or compressing results.

`after_run_callback` executes after the entire agent run completes. It can be used to save conversation content to memory, and we'll use it when we cover Memory in chapter 6.

In this section, we'll implement and utilize tool callbacks that are directly related to this chapter’s topic. The remaining callbacks follow the same pattern, and the complete code is available in the GitHub repository.

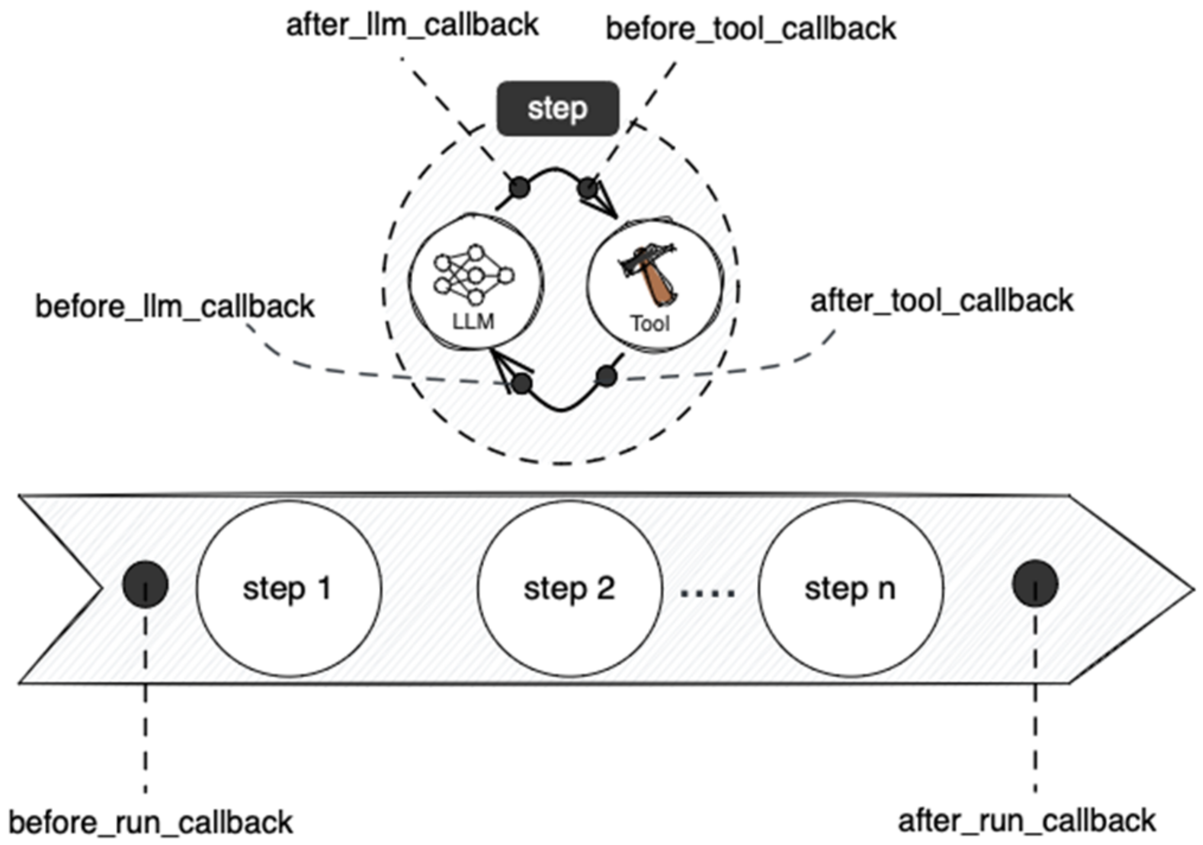


Figure 5.7 Points where Callbacks are inserted in the agent execution flow. LLM callbacks are called before and after think(), and tool callbacks are called before and after each tool execution within act().

5.4.2 Implementing tool callbacks

Now let's add tool callback support to the Agent class. First, we define how callbacks behave.

`before_tool_callback(context, tool_call)` is called before a tool executes. If this callback returns a value, the actual tool execution is skipped and that value is used as the result. If it returns None, the actual tool executes.

`after_tool_callback(context, tool_result)` is called after a tool executes. If this callback returns a value, it replaces the original result. If it returns None, the original result is preserved.

When multiple callbacks are registered, execution stops at the first callback that returns a non-None value. This is called the Early Exit pattern.

We modify the Agent constructor to add callback parameters. It accepts two lists: `before_tool_callbacks` and `after_tool_callbacks`. If callbacks aren't provided, they're initialized as empty lists so that subsequent code can iterate over them without None checks.

Listing 5.24 Adding callback parameters to Agent

```
class Agent:
    def __init__(
        self,
        model: LlmClient,
        tools: List[BaseTool] = None,
        instructions: str = "",
        max_steps: int = 10,
        output_type: Optional[Type[BaseModel]] = None,
        before_tool_callbacks: List[Callable] = None,
        after_tool_callbacks: List[Callable] = None,
    ):
        self.model = model
        self.instructions = instructions
        self.max_steps = max_steps
        self.tools = self._setup_tools(tools or [])
        self.output_type = output_type

        # Initialize callback lists
        self.before_tool_callbacks = before_tool_callbacks or []
        self.after_tool_callbacks = after_tool_callbacks or []
```

Next, we refactor the `act()` method. The original code executed tools directly for each tool_call, but now it's divided into three stages.

In the first stage, we iterate through `before_tool_callbacks` and execute each callback. If a callback returns a non-None value, we store that value in `tool_response` and break out of the loop. We use `inspect.isawaitable()` to support asynchronous callbacks as well.

In the second stage, we execute the actual tool only if `tool_response` is still None. If a callback has already provided a result, this stage is skipped.

In the third stage, we iterate through `after_tool_callbacks`. If a callback returns a new ToolResult, it replaces the original result.

Listing 5.25 Refactoring act() method with callbacks

```
import inspect

async def act(
    self,
    context: ExecutionContext,
    tool_calls: List[ToolCall]
) -> List[ToolResult]:
    tools_dict = {tool.name: tool for tool in self.tools}
    results = []

    for tool_call in tool_calls:
        if tool_call.name not in tools_dict:
            raise ValueError(f"Tool '{tool_call.name}' not found")

        tool = tools_dict[tool_call.name]
        tool_response = None
        status = "success"

        # Stage 1: Execute before_tool_callbacks
        for callback in self.before_tool_callbacks:
            result = callback(context, tool_call)
            if inspect.isawaitable(result):
                result = await result
            if result is not None:
                tool_response = result
                break

        # Stage 2: Execute actual tool only if callback didn't provi
de a result
        if tool_response is None:
            try:
                tool_response = await tool(context, **tool_call.argu
ments)
            except Exception as e:
                tool_response = str(e)
                status = "error"

        tool_result = ToolResult(
            tool_call_id=tool_call.tool_call_id,
            name=tool_call.name,
            status=status,
            content=[tool_response],
        )

        # Stage 3: Execute after_tool_callbacks
        for callback in self.after_tool_callbacks:
            result = callback(context, tool_result)
            if inspect.isawaitable(result):
                result = await result
            if result is not None:
                tool_result = result
                break

        results.append(tool_result)

    return results
```

5.4.3 Human in the loop: Tool execution approval

While it's efficient for agents to perform tasks automatically, this can sometimes be risky. Sensitive operations like file deletion, email sending, or database modifications should require user confirmation before execution. The Human in the Loop pattern is designed for exactly these situations, as shown in figure 5.8.

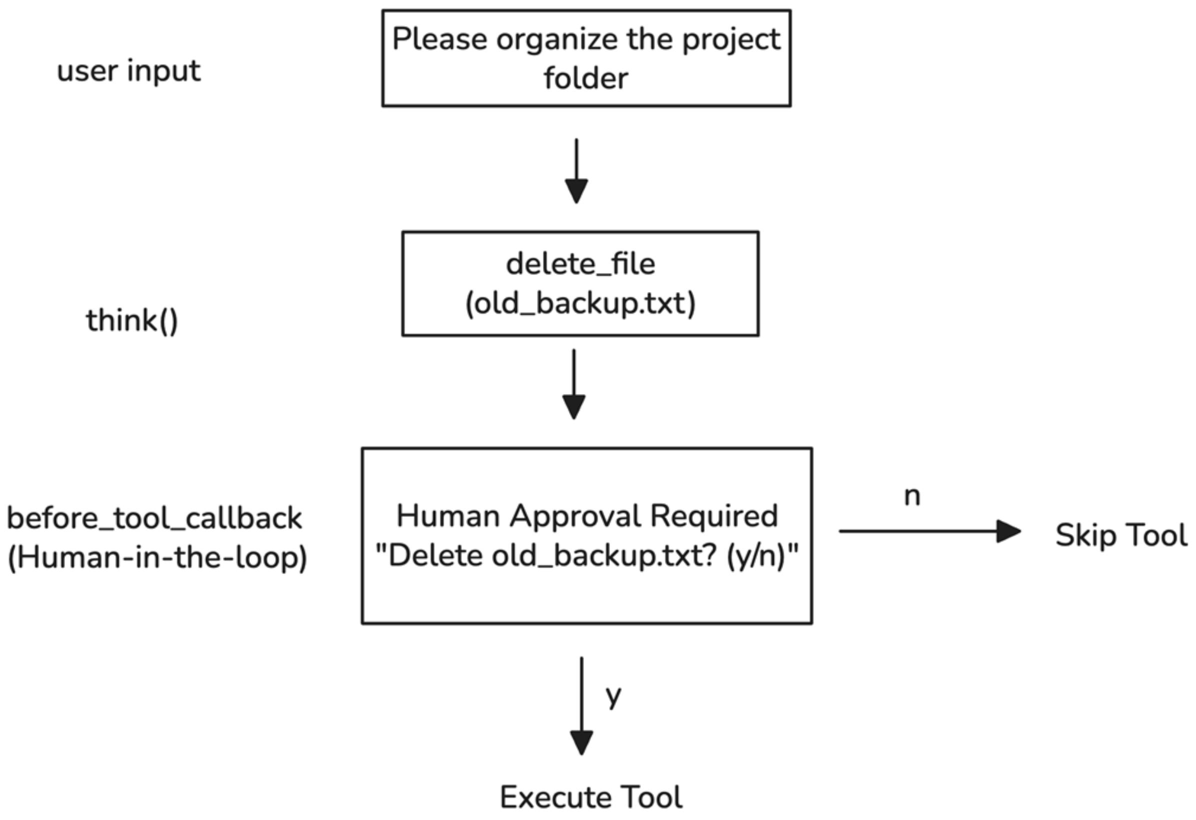

user input
Please organize the project folder
think()
delete_file (old_backup.txt)
before_tool_callback (Human-in-the-loop)
Human Approval Required "Delete old_backup.txt? (y/n)"
n
Skip Tool
y
Execute Tool

Figure 5.8 Tool execution approval flow in the Human in the Loop pattern. When a user requests "Clean up the project folder," the agent pauses when it tries to call the delete_file tool and requests approval from the user.

Let's create a file deletion tool as an example of a dangerous tool that requires approval. This is a dummy implementation that only returns a message without actually deleting files.

Listing 5.26 Implementing delete_file tool

```
@tool
def delete_file(file_path: str) -> str:
    """Deletes a file. This action cannot be undone."""
    # Only returns message instead of actual deletion (for demo)
    return f"File {file_path} has been deleted."
```

Now let's implement a callback that requests approval only when dangerous tools like delete_file are called. The callback first checks if the tool name is in the list of tools requiring approval. If not, it returns None to allow immediate execution. If it is, the callback displays the tool information and waits for user input. Entering 'y' returns None to allow actual tool execution, while any other input returns a rejection message to skip the tool execution.

Listing 5.27 Implementing approval callback

```
# List of dangerous tools requiring approval
DANGEROUS_TOOLS = ["delete_file", "send_email", "execute_sql"]

def approval_callback(context: ExecutionContext, tool_call: ToolCal
l):
    """Requests user approval before executing dangerous tools."""
    # Execute immediately if not a dangerous tool
    if tool_call.name not in DANGEROUS_TOOLS:
        return None

    print(f"\n⚠ Dangerous tool execution requested")
    print(f"Tool: {tool_call.name}")
    print(f"Arguments: {tool_call.arguments}")

    response = input("Do you want to execute? (y/n): ").lower().stri
p()

    if response == 'y':
        print("✅ Approved. Executing...\n")
        return None  # Proceed with actual tool execution
    else:
        print("❌ Denied. Skipping execution.\n")
        return f"User denied execution of {tool_call.name}"
```

The callback's return value is key. Returning None causes the `act()` method to execute the actual tool. Returning a string uses that as the tool result, and the actual tool doesn't execute. The LLM can see this rejection message in the next step and try a different approach.

Let's apply this callback to an agent. Safe tools like list_files or read_file execute without approval, while approval is requested only when delete_file is called.

Listing 5.28 Creating agent with approval callback

```
agent = Agent(
    model=LlmClient(model="gpt-5"),
    tools=[list_files, read_file, delete_file],
    instructions="You are a helpful assistant that can explore and m
anage files.",
    max_steps=20,
    before_tool_callbacks=[approval_callback],
)
```

When executed, you'll see the following flow. `list_files` executes without approval since it's not in the dangerous tools list. When `delete_file` is called, the approval prompt appears.

5.4.4 Compressing search results

In the exercise in section 5.3.4, we confirmed that web search results can consume over 37,000 tokens, and extracting only relevant portions through vector search can save over 99%, reducing it to 304 tokens. However, performing this process manually every time is cumbersome. Let's automate this at the agent level using `after_tool_callback`.

The idea is simple. After the web search tool executes, if the result is too long, we use vector search to extract only the portions relevant to the query. This callback applies only to the search tool and passes through results from other tools unchanged.

The callback flow is as follows. First, we check the tool name and return None if it's not `search_web`. If it is the search tool, we check the result length and return it unchanged if it's under 2,000 characters. For long results, we find the query for that search and perform the chunking and vector search we implemented in section 5.3. Finally, we return a new ToolResult containing only the top 3 chunks.

Listing 5.29 Implementing search compressor callback

```
def search_compressor(context: ExecutionContext, tool_result: ToolRe
sult):
    """Callback that compresses web search results."""
    # Pass through unchanged if not a search tool
    if tool_result.name != "search_web":
        return None

    original_content = tool_result.content[0]

    # No compression needed if result is short enough
    if len(original_content) < 2000:
        return None

    # Extract search query matching the tool_call_id
    query = _extract_search_query(context, tool_result.tool_call_id)
    if not query:
        return None

    # Use functions implemented in section 5.3
    chunks = fixed_length_chunking(original_content, chunk_size=500,
overlap=50)
    embeddings = get_embeddings(chunks)
    results = vector_search(query, chunks, embeddings, top_k=3)

    # Create compressed result
    compressed = "\n\n".join([r['chunk'] for r in results])

    return ToolResult(
        tool_call_id=tool_result.tool_call_id,
        name=tool_result.name,
        status="success",
        content=[compressed]
    )
```

The `_extract_search_query` helper function finds the query for a specific search call from the ExecutionContext. Since the LLM can request multiple searches in parallel, it's safer to match exactly by tool_call_id rather than simply finding the most recent search. We iterate through events looking for items that are ToolCall type, have the name search_web, and match the ID, then return the query argument.

Listing 5.30 Extracting search query from context

```
def _extract_search_query(context: ExecutionContext, tool_call_id: s
tr) -> str:
    """Extracts the search query for a specific tool_call_id from co
ntext."""
    for event in context.events:
        for item in event.content:
            if (isinstance(item, ToolCall)
                and item.name == "search_web"
                and item.tool_call_id == tool_call_id):
                return item.arguments.get("query", "")
    return ""
```

Now let's create an agent that applies both callbacks together. We register the approval callback in `before_tool_callbacks` and the compression callback in `after_tool_callbacks`.

Listing 5.31 Creating agent with multiple callbacks

```
agent = Agent(
    model=LlmClient(model="gpt-5"),
    tools=[search_web, list_files, read_file, delete_file],
    instructions="You are a helpful assistant that can search the we
b and "
                 "explore files to answer questions.",
    max_steps=20,
    before_tool_callbacks=[approval_callback],
    after_tool_callbacks=[search_compressor],
)
```

When web search executes, results are automatically compressed. The work we performed manually in section 5.3.4 is now applied automatically during agent execution.

We achieve the same token savings and noise reduction benefits without modifying the agent's core logic at all.

KEY INSIGHTS

In this section, we learned that we can extend agent behavior without modifying its core logic through the Callback pattern.

`before_tool_callback` intervenes before tool execution. If it returns a value, actual execution is skipped and that value is used as the result. It can be used for Human in the Loop approval, cache checking, input validation, and more.

`after_tool_callback` intervenes after tool execution. If it returns a value, it replaces the original result. It can be used for search result compression, sensitive information masking, result format conversion, and more.

By combining multiple callbacks, we can handle complex requirements cleanly. Each callback has a single responsibility and can be added or removed as needed.

The Human in the Loop implementation in this section operates synchronously. The entire program blocks while the `input()` function waits for user input. This is fine for testing in a local environment, but you can't use this approach in web services or API servers. In chapter 6, we'll implement asynchronous Human in the Loop where the agent doesn't pause in the middle but instead requests approval through a response to the user and resumes execution when the user's response arrives.

We'll also use `after_run_callback` in chapter 6. We'll use this callback to save conversation content to Memory after a conversation ends. You'll continue to see how the callback pattern is utilized as a core mechanism for extending agent functionality.

5.5 Summary

- Internal data requires different approaches than web search. Single files can be read directly into context, but multiple files need exploration, and large-scale data requires search mechanisms to extract only relevant portions.
- Four search methods serve different needs: keyword search for exact matching, vector search for semantic similarity, graph search for relationship traversal, and structure-based search for navigating organized file systems. Production systems often combine multiple methods.
- Vector search converts text to numerical vectors where semantic similarity becomes geometric distance. The pipeline involves chunking long documents, generating embeddings, and calculating cosine similarity to find the most relevant content.
- Structure-based search leverages file system organization as information itself. Folder names, file names, and directory hierarchies provide meaningful metadata that agents can navigate using simple tools like list_files and read_file.
- The Callback pattern extends agent behavior without modifying core logic. before_tool_callback enables Human in the Loop approval for dangerous operations, while after_tool_callback enables automatic compression of search results.
- Context engineering through search dramatically reduces token usage. In our exercise, extracting only relevant chunks reduced over 37,000 tokens to just 304 tokens,

achieving a 99% savings rate while preserving the information needed to answer the query.

6 Adding memory to your agent

This chapter covers

- The role of memory in LLM agents
- Managing context growth with sliding window, compaction, and summarization
- Implementing sessions for multi-turn conversations
- Building asynchronous human-in-the-loop workflows
- Creating long-term memory for cross-session knowledge retention

Memory is what separates a stateless tool from an intelligent assistant. Without memory, an agent cannot recall previous events within the same task, continue conversations from earlier sessions, or learn from past experiences. Each interaction starts from scratch, forcing users to repeat context and preventing the agent from improving over time.

This chapter addresses memory in three usage patterns. First, we implement context optimization strategies to prevent context explosion during complex problem-solving. Second, we build Session and SessionManager to maintain conversation continuity across multiple interactions, extending this architecture to support asynchronous human-in-the-loop workflows. Finally, we create a long-term memory system that extracts, stores, and retrieves knowledge across session boundaries using vector search.

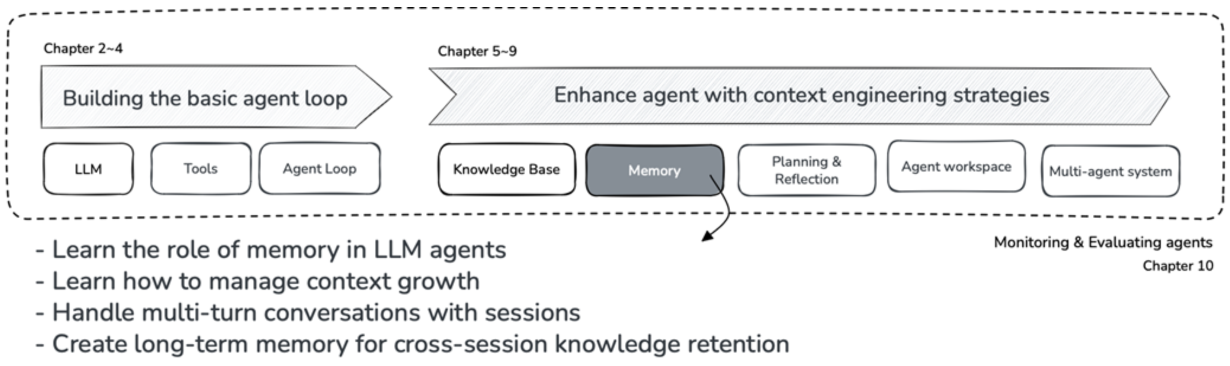

Chapter 2~4
Building the basic agent loop
LLM
Tools
Agent Loop
Chapter 5~9
Enhance agent with context engineering strategies
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Learn the role of memory in LLM agents
- Learn how to manage context growth
- Handle multi-turn conversations with sessions
- Create long-term memory for cross-session knowledge retention

Figure 6.1 Book structure overview: Chapter 6 in focus.

6.1 The anatomy of agent memory

Memory transforms an agent from a simple tool into an intelligent assistant that learns and adapts. Without memory, even the most sophisticated agent starts from scratch every time, unable to reference previous conversations or learn from past experiences.

In this section, we examine the limitations of our current agent, distinguish between short-term and long-term memory, and further divide short-term memory into two patterns based on agent usage. We then clarify how the Context Engineering framework from chapter 1 connects to memory.

6.1.1 Limitations of the current memory architecture

The agent developed so far relies on a raw `ExecutionContext`. It operates simply: append everything during execution, then discard everything after. While this works for simple tasks, it lacks a mechanism to *manage* information effectively.

Consider two scenarios. First, a problem of **space**: when solving a complex GAIA problem, tool call results accumulate rapidly. The context can exceed the LLM's window limit, crashing the agent before it finishes. Second, a problem of **time**: as a personal assistant across multiple conversations, the agent forgets everything between sessions. A user who said "I'm James, I work as a marketer" yesterday must repeat this information today.

These limitations stem from the absence of memory management. Let’s categorize these problems into three distinct challenges and outline our solutions.

THREE CHALLENGES OF THE CURRENT MEMORY ARCHITECTURE

Before diving into the challenges, let's establish three concepts that form the foundation for everything we'll build in this chapter.

A **session** is a single, continuous conversation between a user and an agent. Think of it as opening a new chat window—everything discussed within that window belongs to the same session. Closing the window and starting a new conversation later creates a new session.

Short-term memory stores the work done so far: conversation history, tool call results, and intermediate reasoning—everything stored in `ExecutionContext.events`.

Long-term memory is information that persists after sessions end. When a user mentions "I'm a marketer" today, long-term memory ensures the agent remembers this next week, even in a completely new session.

CHALLENGE 1: CONTEXT EXPLOSION

Many agents are used for one-off execution. When we ran the GAIA benchmark, we executed `run()` once and terminated. For such cases, our current implementation is sufficient. However, in complex problems, multiple tool calls can cause context to grow rapidly. The core challenge: "How do we compress context during the current execution?"

CHALLENGE 2: LACK OF CONVERSATION CONTINUITY

What happens when an agent receives multiple requests from users, like a conversational chatbot, or needs to pause because certain tool executions require user approval? How do we maintain state across multiple `run()` calls?

Without proper session management, each `run()` call operates independently. The conversation breaks. Figure 6.2 illustrates this: on the left, without a session, a follow-up request after asking about Project Alpha, leaves the agent unaware of which project is meant. On the right, with a session, context is preserved and the agent correctly understands the reference.

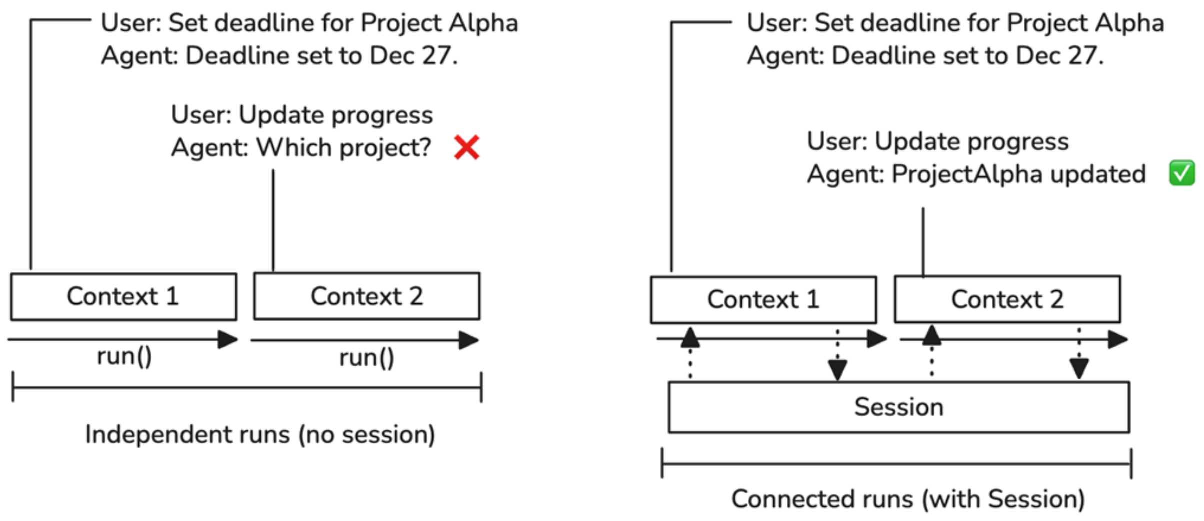

User: Set deadline for Project Alpha
Agent: Deadline set to Dec 27.
User: Update progress
Agent: Which project?
Context 1
Context 2
run()
run()
Independent runs (no session)
User: Set deadline for Project Alpha
Agent: Deadline set to Dec 27.
User: Update progress
Agent: ProjectAlpha updated
Context 1
Context 2
Session
Connected runs (with Session)

Figure 6.2 Without Session (left), each run() creates a new ExecutionContext that's discarded after completion. With Session (right), multiple run() calls share the same Session, maintaining conversation continuity.

In web services, servers operate statelessly. Each HTTP request is processed independently, and server memory is released when the request ends. When a user sends another message 10 minutes later, it's a new request.

Maintaining conversation context requires explicit work: assign a unique `session_id` to each conversation, save session state to external storage when `run()` ends, and load the previous state when the next `run()` starts.

CHALLENGE 3: LACK OF PERSISTENT KNOWLEDGE

Consider a personalized AI assistant. Information like "I'm a marketer" shared today should be remembered next week—even in a new conversation. What should survive after a session ends, and how?

This differs fundamentally from challenge 2. Conversation continuity maintains context within the same `session_id`; you can close and reopen the chat window, and context persists as long as it's the same session. Persistent knowledge shares information across different `session_ids`—the agent remembers the user when starting a brand new conversation.

Figure 6.3 shows how this works. When session 1 ends, the system examines whether any information is worth preserving. If so, that knowledge is stored in long-term memory. When a new session begins, the agent searches and retrieves relevant information as needed.

Although the figure simplifies the process by showing storage at the session level, in practice the system operates at each `run()` execution—storing and retrieving incrementally for finer control.

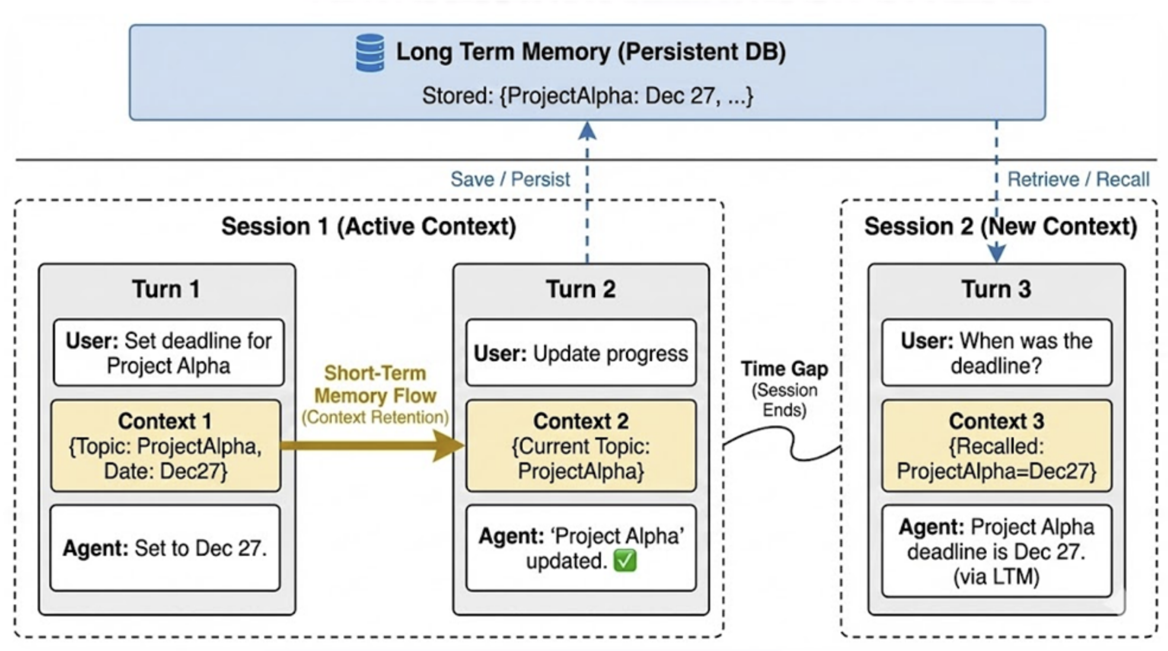

Long Term Memory (Persistent DB)
Stored: {ProjectAlpha: Dec 27, ...}
Save / Persist
Retrieve / Recall
Session 1 (Active Context)
Session 2 (New Context)
Turn 1
User: Set deadline for Project Alpha
Context 1 {Topic: ProjectAlpha, Date: Dec27}
Agent: Set to Dec 27.
Short-Term Memory Flow (Context Retention)
Turn 2
User: Update progress
Context 2 {Current Topic: ProjectAlpha}
Agent: 'Project Alpha' updated. ✅
Time Gap (Session Ends)
Turn 3
User: When was the deadline?
Context 3 {Recalled: ProjectAlpha=Dec27}
Agent: Project Alpha deadline is Dec 27. (via LTM)

Figure 6.3 Long-term memory enables information sharing across sessions. Each run() creates an ExecutionContext, multiple runs form a Session, and important information from sessions is extracted and stored for future use.

6.1.2 Context engineering and memory

In chapter 1, we defined Context Engineering as "the art of providing the right information at the right time" and introduced five strategies. Chapter 6 (Memory) is a hub where three of these strategies converge.

Table 6.1 Context engineering strategies and related chapters

Strategy	Description	Related Chapters
Generation	Constructing context with LLM-generated outputs	Chapter 7
Retrieval	Bringing information from external sources into context	Chapter 3, Chapter 5, Chapter 6
Write	Saving from context to external storage	Chapter 6, Chapter 8
Reduce	Compressing and deleting context	Chapter 6
Isolate	Separating tasks/tools into distinct environments	Chapter 8, Chapter 9

The patterns above map to context engineering strategies as follows. Patterns 1 and 2 relate to Reduce and Write. To manage expanding context, we shrink it using methods like sliding window, compaction, and summarization, or separate and store it externally. Pattern 3 saves important information to external storage (Write) and retrieves it for use (Retrieval).

Table 6.2 Mapping each pattern to strategies

Pattern	Primary Strategy	Description
Patterns 1, 2	Reduce + Write	Managing extensive context via sliding window, compaction, summarization
Pattern 3	Write + Retrieval	Extract and store memories (Write) → Search via tool in next session (Retrieval)

We group Patterns 1 and 2 together because both address the same problem: "How do we manage the context being sent to the LLM right now?"

- Pattern 1: When context grows within a single `run()`
- Pattern 2: When conversation history accumulates across multiple `run()` calls

The solutions also use the same Reduce strategy. Sliding window keeps only the most recent N messages. Compaction deletes tool calls and their results, or saves them to a file. Summarization condenses old conversations.

The difference is where the data resides. In Pattern 1, we work with `events` within `ExecutionContext`. In Pattern 2, `SessionManager` loads `events` stored in a `Session`, then processes them the same way.

6.2 Managing context during execution

The core challenge of Pattern 1 is that context grows rapidly as tool calls and results accumulate while solving complex problems. Let's implement concrete strategies to address this problem.

6.2.1 Separating storage from presentation

The first principle of context management is separating storage from presentation. All events recorded in `ExecutionContext.events` remain immutable; only the `LlmRequest.contents` sent to the LLM are processed.

WHY THIS SEPARATION MATTERS

Think of `ExecutionContext` as your ground truth: a complete, structured record of everything that happened. `LlmRequest` is simply a computed projection that you can refine and optimize for each specific model call.

This separation provides several benefits. Without the original history, tracing the cause of agent failures becomes difficul. To answer, "Why did it draw the wrong conclusion from these search results?" you need the original search results from that moment. Additionally, some tools can reference results from previously performed operations. If you delete originals, these tools may not function properly.

Consider posting photos to social media: you keep the original photo and only apply filters when posting. With the original preserved, you can always process it differently later. Our context management works the same way. `ExecutionContext` is the repository that maintains the complete execution history, while `LlmRequest` processes that history for delivery to the LLM. You can change how the LLM sees the context without migrating or modifying your stored data.

IMPLEMENTATION VIA CALLBACKS

In chapter 4, we introduced `before_llm_callback`. This callback executes just before the LLM call, providing an opportunity to process the `LlmRequest`.

The callback receives both `ExecutionContext` and `LlmRequest`. It reads the current state from `ExecutionContext` and modifies the contents of `LlmRequest`. If the callback returns `None`, the LLM call proceeds with the modified request; if it returns an `LlmResponse`, the LLM call is skipped and that response is used instead. The key point is that `ExecutionContext.events` is read-only; modifications occur only in `LlmRequest.contents`.

```
before_llm_callback(context: ExecutionContext, request: LlmRequest)
-> Optional[LlmResponse]
```

To flexibly apply context optimization strategies, you need to adjust configuration values like token thresholds. The following function addresses this using the closure pattern. `create_optimizer_callback` takes an optimization function and a threshold as arguments to generate a callback function. The generated callback only invokes the optimization function when the token count exceeds the threshold.

Listing 6.1 Creating an optimizer callback with threshold

```
def create_optimizer_callback(apply_optimization, threshold: int = 5
0000):
    """Factory function that creates a callback applying optimizatio
n strategy"""

    async def callback(
        context: ExecutionContext,
        request: LlmRequest
    ) -> Optional[LlmResponse]:
        token_count = count_tokens(request)

        if token_count < threshold:
            return None

        # Support both sync and async functions
        result = apply_optimization(context, request)
        if inspect.isawaitable(result):
            await result
        return None

    return callback
```

Optimization functions follow the `(context, request) -> None` signature. The reason for receiving context is that execution state (e.g., how far compression has progressed) needs to be recorded in `context.state`. Now, let's look at specific optimization strategies.

6.2.2 Sliding window strategy

The simplest approach is to keep only the most recent N messages. By discarding older messages and keeping only the newest, you limit the context size.

BASIC IMPLEMENTATION

The following function implements a sliding window strategy with three steps: (1) iterate through contents to find the user's original question; (2) unconditionally preserve everything up to the user message, ensuring the agent retains the problem context; (3) keep only the most recent `window_size` items from subsequent items (assistant messages, tool calls, tool results).

Listing 6.2 Sliding window strategy implementation

```
def apply_sliding_window(
    context: ExecutionContext,
    request: LlmRequest,
    window_size: int = 20
) -> None:
    """Sliding window that keeps only the most recent N messages"""

    contents = request.contents

    # Find user message position
    user_message_idx = None
    for i, item in enumerate(contents):
        if isinstance(item, Message) and item.role == "user":
            user_message_idx = i
            break

    if user_message_idx is None:
        return

    # Preserve up to user message
    preserved = contents[:user_message_idx + 1]

    # Keep only the most recent N from remaining items
    remaining = contents[user_message_idx + 1:]
    if len(remaining) > window_size:
        remaining = remaining[-window_size:]

    request.contents = preserved + remaining
```

To use the sliding window, pass it to `create_optimizer_callback`.

Listing 6.3 Using sliding window with optimizer callback

```
optimizer = create_optimizer_callback(
    apply_optimization=apply_sliding_window,
    threshold=30000
)

agent = Agent(
    model=LlmClient(model="gpt-5"),
    before_llm_callback=optimizer
)
```

LIMITATIONS

The sliding window is simple to implement and predictable, but has several limitations. If N is small, you may lose important context; if N is large, it doesn't solve the context explosion problem. More fundamentally, message count isn't a good indicator of context size. A file read result is one message but can occupy tens of thousands of tokens. Therefore, more precise management requires directly measuring token counts.

6.2.3 Token counting

The LLM's actual limit is determined by token count, not message count. A single file read result might take up 10,000 tokens, while 20 short conversations combined might only be 500 tokens. Accurate context management requires directly measuring token counts.

INTRODUCING TIKTOKEN

tiktoken is a tokenizer library provided by OpenAI. Since it's identical to the tokenizer actually used by GPT models, you can calculate exact token counts. Even when using other models (Claude, Gemini, etc.), it can be used as a rough estimate.

The following function calculates the total token count of an LlmRequest. The key is to use `build_messages` to convert to the actual format sent to the API, then calculate tokens. This gives you an accurate token count that includes instructions, contents, and tool definitions.

Listing 6.4 Token counting with tiktoken

```
def count_tokens(request: LlmRequest) -> int:
    """Calculate total token count of LlmRequest"""
    import tiktoken
    from .llm import build_messages

    # Select encoding for model, use default on failure
    try:
        encoding = tiktoken.encoding_for_model(request.model_id or
"gpt-5")
    except KeyError:
        encoding = tiktoken.get_encoding("o200k_base")

    # Convert to API message format then count tokens
    messages = build_messages(request)
    total_tokens = 0

    for message in messages:
        # Per-message overhead (role, separators, etc.)
        total_tokens += 4

        # Content tokens
        if message.get("content"):
            total_tokens += len(encoding.encode(message["content"]))

        # tool_calls tokens
        if message.get("tool_calls"):
            for tool_call in message["tool_calls"]:
                func = tool_call.get("function", {})
                if func.get("name"):
                    total_tokens += len(encoding.encode(func["nam
e"]))
                if func.get("arguments"):
                    total_tokens += len(encoding.encode(func["argume
nts"]))

    # Tool definition tokens
    if request.tools:
        for tool in request.tools:
            tool_def = tool.tool_definition
            total_tokens += len(encoding.encode(json.dumps(tool_de
f)))
```

```
    return total_tokens
```

WHY OPTIMIZATION MATTERS EVEN BELOW THE LIMIT

Most modern models support context windows of 128K to over 1M tokens. You might think optimization is only necessary when approaching these limits. However, context bloat creates problems well before reaching the maximum.

Context optimization isn't just about avoiding errors. Even within the limit, bloated context creates three problems. First, cost and latency grow with token count. Model pricing is per-token, and time-to-first-token increases with context size. Second, irrelevant information distracts the model from the immediate task. Research shows that models struggle to find relevant information buried in long contexts, a phenomenon known as "lost in the middle." Third, you need to reserve space for the model's response.

Recent research and production experience show that performance degradation begins well below the advertised context limit. Chroma's "Context Rot" study (2025) found that model performance becomes increasingly unreliable as input length grows, even on simple tasks. Manus's engineering team notes that "model performance tends to degrade beyond a certain context length, even if the window technically supports it."

The exact threshold varies by model, task complexity, and use case, but the consensus is clear: the optimal working context is much smaller than the maximum window size. As a general rule, minimize context to what's actually needed for the task.

In this book's examples, we use a small threshold value (around 20K tokens) to clearly observe compaction and context management behaviors.

6.2.4 Compaction strategy

The core idea of compaction is to replace already-processed data with lightweight references. If you've read a file and completed analysis, there's no need to keep the entire file contents in context. Instead, you leave a hint saying "this file was read and can be re-read if needed."

This approach follows a "scope by default" principle: give the agent the minimum context needed, and let it explicitly request more when necessary. The agent sees a reference like "Search results processed. Re-search if needed." If it actually needs that data again, it calls the tool.

Compaction divides into two types based on the compression target:

TOOLCALL COMPRESSION: WRITE OPERATIONS

Tools like `create_file`, which store data externally, include large data in the ToolCall's arguments. For example, saving a 10,000-character report to a file results in a ToolCall like this:

```
ToolCall(
    name="create_file",
    arguments={"path": "report.md", "content": "... 10000 characters
of report content ..."}
)
```

There's no need to keep maintaining content already saved to a file in the context. Remove the content from the

ToolCall's arguments and replace it with a reference message.

TOOLRESULT COMPRESSION: READ OPERATIONS

Tools like `read_file` or `search_web`, which fetch data from external sources, include large data in the ToolResult's content. File contents or search results can occupy tens of thousands of tokens.

If the agent has already processed the information, leave only "what information was retrieved" instead of the original data. If the agent needs to reference that content again, it can call the tool again.

IMPLEMENTATION

The following code implements compaction. The key is processing both ToolCalls and ToolResults while iterating through contents with a single for loop. Since ToolCalls always appear before their corresponding ToolResults, when you encounter a ToolCall, you can save its arguments for use when compressing the ToolResult later.

`TOOLCALL_COMPACTION_RULES` defines tools whose `ToolCall` arguments should be compressed, and `TOOLRESULT_COMPACTION_RULES` defines tools whose `ToolResult` content should be compressed.

Listing 6.5 Compaction strategy implementation

```
# Tools to compress ToolCall arguments
TOOLCALL_COMPACTION_RULES = {
    "create_file": "[Content saved to file]",
}

# Tools to compress ToolResult content
TOOLRESULT_COMPACTION_RULES = {
    "read_file": "File content from {file_path}. Re-read if neede
d.",
    "search_web": "Search results processed. Query: {query}. Re-sear
ch if needed.",
    "tavily_search": "Search results processed. Query: {query}. Re-s
earch if needed.",
}

def apply_compaction(context: ExecutionContext, request: LlmRequest)
-> None:
    """Compress tool calls and results into reference messages"""

    tool_call_args: Dict[str, Dict] = {}
    compacted = []

    for item in request.contents:
        if isinstance(item, ToolCall):
            # Save arguments (for use when compressing ToolResult la
ter)
            tool_call_args[item.tool_call_id] = item.arguments

            # If the ToolCall itself is a compression target (create
_file, etc.)
            if item.name in TOOLCALL_COMPACTION_RULES:
                compressed_args = {
                    k: TOOLCALL_COMPACTION_RULES[item.name] if k ==
"content" else v
                    for k, v in item.arguments.items()
                }
                compacted.append(ToolCall(
                    tool_call_id=item.tool_call_id,
                    name=item.name,
                    arguments=compressed_args
                ))
            else:
                compacted.append(item)

        elif isinstance(item, ToolResult):
            # If ToolResult is a compression target (read_file, sear
ch_web, etc.)
            if item.name in TOOLRESULT_COMPACTION_RULES:
                args = tool_call_args.get(item.tool_call_id, {})
                template = TOOLRESULT_COMPACTION_RULES[item.name]
                compressed_content = template.format(
                    file_path=args.get("file_path", args.get("path",
"unknown")),
                    query=args.get("query", "unknown")
                )
                compacted.append(ToolResult(
                    tool_call_id=item.tool_call_id,
                    name=item.name,
                    status=item.status,
                    content=[compressed_content]
                ))
            else:
                compacted.append(item)
        else:
            compacted.append(item)

    request.contents = compacted
```

This turns "100K tokens of noise in every prompt" into a precise, on-demand resource. The data can be huge, but the context window remains lean.

TRADE-OFFS WITH PROMPT CACHING

There's an important factor to consider when applying compaction: prompt caching.

Prompt caching is a cost-optimization feature provided by LLM APIs. When the initial portion of prompts is identical across consecutive requests, the API reuses previously computed results. This can reduce costs by up to 90% and speed up response times. OpenAI, Anthropic, and Google all support this feature, and it's applied automatically without special configuration.

The problem is that compaction modifies past messages. This matters because rompt caching works by reusing computation for identical prefixes. Consider what happens across two consecutive steps:

```
Step 5: [system][user][msg1][msg2][msg3][msg4][msg5]
Step 6: [system][user][msg1][msg2][msg3][msg4][msg5][msg6]
                                                    ↑ only this ne
eds new computation
```

When you modify messages in the middle of context (like compressing old tool results), everything after that point becomes different.

```
Step 6 (Compacted): [system][user][msg1][compressed msg2][msg3]...[m
sg6]
                                              ↑ changed → everything
after is cache miss
```

The solution is threshold-based optimization. Below a certain threshold, keep the context stable to preserve caching benefits. Only modify when the cost of context overflow exceeds the cost of cache misses. The `create_optimizer_callback` we already implemented handles this role.

```
# Below 50000 tokens, don't apply Compaction (maintain caching benef
its)
# At 50000 tokens or above, apply Compaction (prevent degradation)
optimizer = create_optimizer_callback(
    apply_optimization=apply_compaction,
    threshold=50000
)
```

6.2.5 Summarization strategy

Compaction only targets specific tool calls and results. If the general conversation or the agent's reasoning becomes lengthy, compaction alone can't solve it. In these cases, summarization is used as a last resort.

SELECTING SUMMARIZATION TARGETS

Summarization selects the oldest messages first as summarization targets. However, the user's original question must be preserved. The agent must not forget what it's currently trying to solve.

Summarization targets only the intermediate process: which tools were called, what information was discovered, and the progress made. This information is summarized and added to the system prompt.

DESIGNING THE SUMMARY PROMPT

The following is a prompt for summarizing the agent's intermediate process. Requesting structured output ensures summaries are consistent.

```
SUMMARIZATION_PROMPT = """You are summarizing an AI agent's work pro
gress.

Given the following execution history, extract:
1. Key findings: Important information discovered
2. Tools used: List of tools that were called
3. Current status: What has been accomplished and what remains

Be concise. Focus on information that will help the agent continue i
ts work.

Execution History:
{history}

Provide a structured summary."""
```

IMPLEMENTATION

The following function implements summarization. The core logic consists of four steps:

1. Find the user message position to mark the starting point of the preservation range.
2. Keep the most recent `keep_recent` messages in their original form.
3. Summarize the intermediate process between them using the LLM.
4. Reconstruct the result as "user question + summary message + recent messages."

Recording the last summarization position in `context.state` prevents re-summarizing portions that have already been summarized in subsuquent summarization calls.

Listing 6.6 Summarization strategy implementation

```
async def apply_summarization(
    context: ExecutionContext,
    request: LlmRequest,
    llm_client: LlmClient,
    keep_recent: int = 5
) -> None:
    """Replace old messages with a summary"""

    contents = request.contents

    # Find user message position
    user_idx = None
    for i, item in enumerate(contents):
        if isinstance(item, Message) and item.role == "user":
            user_idx = i
            break

    if user_idx is None:
        return

    # Check previous summary position (skip already-summarized porti
ons)
    last_summary_idx = context.state.get("last_summary_idx", user_id
x)

    # Calculate summarization target range
    summary_start = last_summary_idx + 1
    summary_end = len(contents) - keep_recent

    # Overlap prevention: exit if nothing to summarize or range is i
nvalid
    if summary_end <= summary_start:
        return

    # Determine portions to preserve (no overlap)
    preserved_start = contents[:last_summary_idx + 1]
    preserved_end = contents[summary_end:]
    to_summarize = contents[summary_start:summary_end]

    # Generate summary
    history_text = format_history_for_summary(to_summarize)
    summary = await generate_summary(llm_client, history_text)
    # Add summary to instructions
    request.append_instructions(f"[Previous work summary]\n{summar
y}")

    # Keep only preserved portions in contents
    request.contents = preserved_start + preserved_end

    # Record summary position
    context.state["last_summary_idx"] = len(preserved_start) - 1


def format_history_for_summary(items: List[ContentItem]) -> str:
    """Convert ContentItem list to text for summarization"""
    lines = []
    for item in items:
        if isinstance(item, Message):
            lines.append(f"[{item.role}]: {item.content[:500]}...")
        elif isinstance(item, ToolCall):
            lines.append(f"[Tool Call]: {item.name}({item.argument
s})")
        elif isinstance(item, ToolResult):
            content_preview = str(item.content[0])[:200] if item.con
tent else ""
            lines.append(f"[Tool Result]: {item.name} -> {content_pr
eview}...")
    return "\n".join(lines)


async def generate_summary(llm_client: LlmClient, history: str) -> s
tr:
    """Generate history summary using LLM"""

    request = LlmRequest(
        instructions=[SUMMARIZATION_PROMPT.format(history=history)],
        contents=[Message(role="user", content="Please summarize.")]
    )

    response = await llm_client.generate(request)

    for item in response.content:
        if isinstance(item, Message):
            return item.content
    return ""
```

COMPACTION VS. SUMMARIZATION

These two strategies differ fundamentally. Compaction is deterministic: you can predict exactly what will be compressed based on tool names. Summarization is semantic: an LLM decides what's important. Deterministic operations are cheaper, faster, and more predictable, so we apply them first.

There's another important difference: reversibility. With compaction, the agent can restore original data by calling the tool again. With summarization, the original reasoning is permanently lost. Therefore, try compaction first and only apply summarization when that's insufficient.

6.2.6 Hierarchical context management

In what order should we apply these strategies? Considering cost and effectiveness, the hierarchical approach illustrated in figure 6.4 is reasonable.

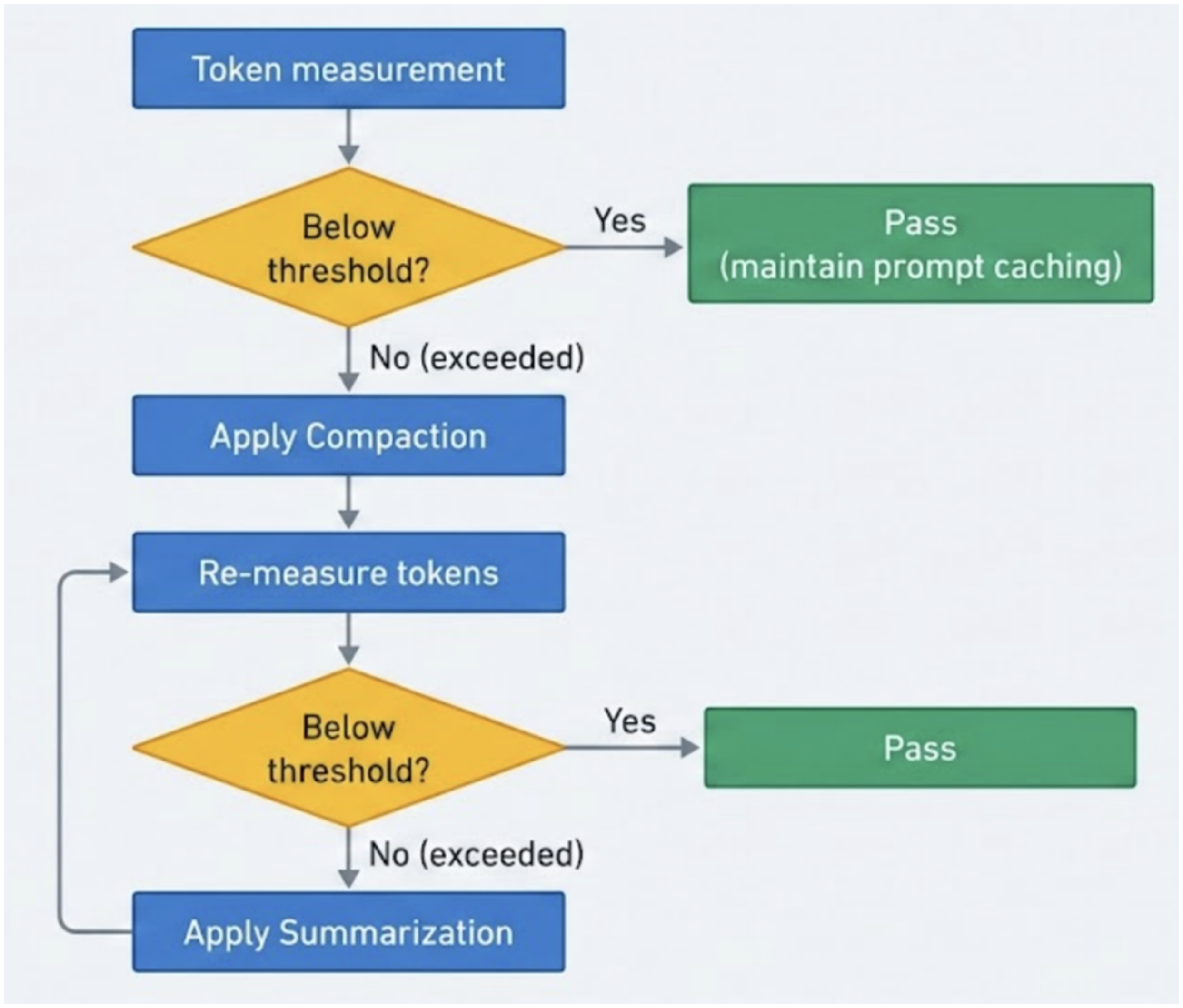


Figure 6.4 Context management strategy decision flow.

First, measure the token count. If it is below the threshold, do nothing. This maintains prompt caching benefits and avoids unnecessary processing.

If the threshold is exceeded, apply Compaction. Compaction is free and fast. In most cases, file reads or search results are the main culprits of context growth, so Compaction alone is often sufficient.

If the threshold is still exceeded after compaction, apply summarization. This is the last resort. It incurs additional

LLM call costs, but it's better than failure due to context errors.

INTEGRATED IMPLEMENTATION

To apply multiple strategies sequentially, implementing them as a class is cleaner. The following class integrates all strategies. The `__call__` method is registered as `before_llm_callback` and executes the three stages sequentially. After each stage, tokens are re-measured and if below the threshold, it returns immediately.

Listing 6.7 Hierarchical ContextOptimizer class

```
class ContextOptimizer:
    """Hierarchical context optimization strategy"""

    def __init__(
        self,
        llm_client: LlmClient,
        token_threshold: int = 50000,
        enable_compaction: bool = True,
        enable_summarization: bool = True,
        keep_recent: int = 5
    ):
        self.llm_client = llm_client
        self.token_threshold = token_threshold
        self.enable_compaction = enable_compaction
        self.enable_summarization = enable_summarization
        self.keep_recent = keep_recent

    async def __call__(
        self,
        context: ExecutionContext,
        request: LlmRequest
    ) -> Optional[LlmResponse]:
        """Register as before_llm_callback"""

        # Step 1: Measure tokens
        if count_tokens(request) < self.token_threshold:
            return None

        # Step 2: Apply Compaction
        if self.enable_compaction:
            apply_compaction(context, request)

            if count_tokens(request) < self.token_threshold:
                return None

        # Step 3: Apply Summarization
        if self.enable_summarization:
            await apply_summarization(
                context,
                request,
                self.llm_client,
                self.keep_recent
            )

        return None
```

REGISTERING WITH THE AGENT

The following code demonstrates how to register `ContextOptimizer` as the agent's `before_llm_callback`. The optimization LLM uses a less expensive model (gpt-5-mini) to reduce costs, while the agent's main model uses a more powerful model (gpt-5).

```
optimizer = ContextOptimizer(
    llm_client=LlmClient(model="gpt-5-mini"),
    token_threshold=50000,
    enable_compaction=True,
    enable_summarization=True
)

agent = Agent(
    model=LlmClient(model="gpt-5"),
    tools=[search_web, read_file, create_file],
    instructions="You are a helpful research assistant.",
    max_steps=30,
    before_llm_callback=optimizer
)
```

6.3 Continuous execution: Session and state management

We've examined the core challenge of Pattern 2: Continuous Execution: maintaining conversation context across multiple `run()` calls. Currently, our agent creates a new `ExecutionContext` with each `run()` call, causing all information to be lost when execution ends. It remembers neither yesterday's conversation nor the one from moments ago.

The Session and SessionManager we'll implement in this section address this problem. By storing conversation history in a Session and accessing it again with the same `session_id`, we can seamlessly continue previous conversations. The following example demonstrates this capability:

```
# Maintaining conversation continuity with the same session_id
result1 = await agent.run("My name is Alice", session_id="session_
1")
# → "Nice to meet you, Alice!"

result2 = await agent.run("What's my name?", session_id="session_1")
# → "Your name is Alice."
```

However, the Session's role extends beyond simple chat history storage. Sessions can store not only conversation history but also intermediate execution state. By leveraging this capability, we can overcome the fundamental limitations of the synchronous human-in-the-loop implementation from chapter 5.

In chapter 5, we implemented user approval using the `input()` function. Before executing a dangerous tool, the agent would prompt "Do you approve? (Y/N)" in the terminal and wait for the user's response. While this approach works fine during local development, it's unusable in production services.

The solution is to pause the agent's execution when a tool requiring approval is encountered, save the state, and resume later. When the agent encounters a tool call requiring approval, it saves the current execution state to the Session and returns immediately. The user can then click a button on a web page, respond to a notification on a mobile app, or even approve via email days later. Once approval arrives, the saved state is restored and execution continues.

In this section, we'll achieve two goals:

Part 1: Session Fundamentals. We'll explore the structure of the Session class and SessionManager, then integrate them with our agent to support multi-turn conversations.

Part 2: Human-in-the-Loop. We'll leverage Session's state persistence capabilities to implement an asynchronous tool approval workflow. Through this, we'll see firsthand that Session offers value far beyond simple chat history storage.

6.3.1 The session class

A Session is a container that holds all information that must persist across multiple `run()` calls. It serves two core purposes. First, it stores conversation history so that previous conversations can be continued. Second, it stores arbitrary execution state so that interrupted tasks can be resumed.

Before designing the Session's structure, we need to clarify its relationship with the `ExecutionContext` we implemented in chapter 4. An `ExecutionContext` holds all events and states that occur during a single `run()` execution. Once execution ends, that `ExecutionContext` is no longer used. A Session, by contrast, persists across multiple `run()` executions. Therefore, a Session has a longer lifecycle than an `ExecutionContext`. The pattern works as follows: when each `run()` execution completes, necessary information is saved to the Session; when the next `run()` execution begins, information is restored from the Session. This relationship is expressed in figure 6.5.

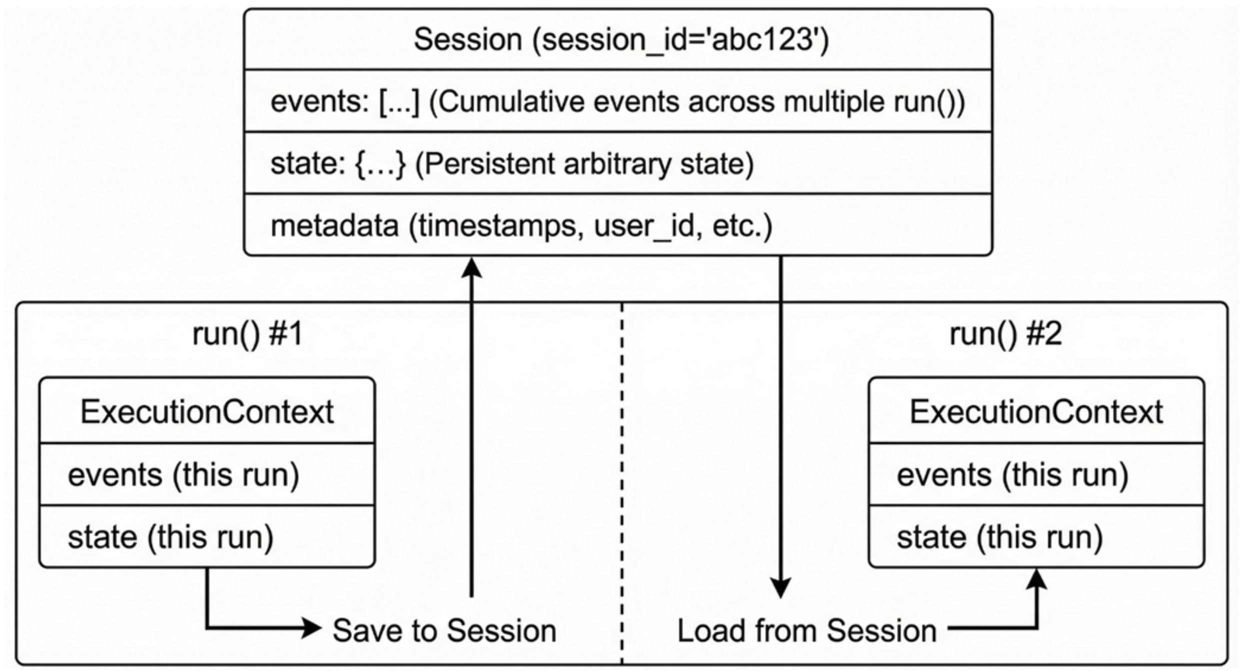

Session (session_id='abc123')
events: [...] (Cumulative events across multiple run())
state: {...} (Persistent arbitrary state)
metadata (timestamps, user_id, etc.)
run() #1
ExecutionContext
events (this run)
state (this run)
Save to Session
run() #2
ExecutionContext
events (this run)
state (this run)
Load from Session

Figure 6.5 Relationship between Session and ExecutionContext.

Now let's implement the Session class. The `session_id` uniquely identifies the session, while `user_id` optionally identifies the user. The `events` list stores all events accumulated across multiple `run()` calls, and the `state` dictionary stores arbitrary execution state. The timestamps track when the session was created and last updated.

Listing 6.8 Session class design

```
class Session(BaseModel):
    """Container for persistent conversation state across multiple r
un() calls."""

    session_id: str
    user_id: str | None = None
    events: list[Event] = Field(default_factory=list)
    state: dict[str, Any] = Field(default_factory=dict)
    created_at: datetime = Field(default_factory=datetime.now)
    updated_at: datetime = Field(default_factory=datetime.now)
```

A notable aspect of the Session class design is the versatility of the `state` field. With its `dict[str, Any]` type, it can store any kind of data. It might hold a pending tool call awaiting approval in a human-in-the-loop scenario, or it might store the progress of a long-running task. We adopted this generic design so that diverse execution patterns can be supported without modifying the Session class itself.

6.3.2 Managing sessions with SessionManager

The `Session` class is merely a data container; it doesn't know how to create or persist itself. We need a separate manager responsible for creating, retrieving, and saving sessions. This role belongs to `SessionManager`.

The core responsibilities of `SessionManager` are:

- **Create**: Make a new session and store it.
- **Get**: Find and return an existing session by `session_id`.
- **Get or Create**: Return an existing session if found, otherwise create a new one.
- **Save**: Persist changes to a session back to storage.

We first define these responsibilities as an abstract class, then provide concrete implementations depending on the storage type. For example, `InMemorySessionManager` stores sessions in a Python dictionary, while production environments might swap this for a database-backed implementation.

DEFINING THE ABSTRACT CLASS

First, we define the interface that all `SessionManager` implementations must follow. The `create` method generates a new session, `get` retrieves an existing one, and `save` persists changes to storage. Since `get_or_create` is simply a combination of `get` and `create`, we implement it as a regular method rather than an abstract one. Subclasses only need to implement `get` and `create`, and `get_or_create` works automatically.

Notice that all methods are declared as `async`. For simple implementations like `InMemorySessionManager`, asynchronous operations aren't necessary. However, implementations that use databases or external APIs require them. By defining an asynchronous interface in the abstract class, we can swap implementations without modifying calling code.

Listing 6.9 BaseSessionManager abstract class

```
from abc import ABC, abstractmethod

class BaseSessionManager(ABC):
    """Abstract base class for session management."""

    @abstractmethod
    async def create(
        self,
        session_id: str,
        user_id: str | None = None
    ) -> Session:
        """Create a new session."""
        pass

    @abstractmethod
    async def get(self, session_id: str) -> Session | None:
        """Retrieve a session by ID. Returns None if not found."""
        pass

    @abstractmethod
    async def save(self, session: Session) -> None:
        """Persist session changes to storage."""
        pass

    async def get_or_create(
        self,
        session_id: str,
        user_id: str | None = None
    ) -> Session:
        """Get existing session or create new one."""
        session = await self.get(session_id)
        if session is None:
            session = await self.create(session_id, user_id)
        return session
```

IN-MEMORY IMPLEMENTATION

Now we implement `InMemorySessionManager` for development and testing environments. It uses a Python dictionary as storage, meaning all sessions are lost when the program terminates. In production, you would replace this with a database-backed implementation.

The `create` method instantiates a new `Session` object and stores it in the dictionary. If a session with the same `session_id` already exists, it raises a `ValueError` to prevent accidental overwrites. The `get` method looks up a session in the dictionary and returns `None` if not found. The `save` method stores the session in the dictionary.

Listing 6.10 InMemorySessionManager implementation

```
class InMemorySessionManager(BaseSessionManager):
    """In-memory session storage for development and testing."""

    def __init__(self):
        self._sessions: dict[str, Session] = {}

    async def create(
        self,
        session_id: str,
        user_id: str | None = None
    ) -> Session:
        """Create a new session."""
        if session_id in self._sessions:
            raise ValueError(f"Session {session_id} already exists")

        session = Session(
            session_id=session_id,
            user_id=user_id
        )
        self._sessions[session_id] = session
        return session

    async def get(self, session_id: str) -> Session | None:
        """Retrieve a session by ID."""
        return self._sessions.get(session_id)

    async def save(self, session: Session) -> None:
        """Save session to storage."""
        self._sessions[session.session_id] = session
```

With `Session` and `SessionManager` in place, letsintegrates them into the agent to enable multi-turn conversations.

6.3.3 Integrating sessions into the agent

We now need to integrate these components into our existing agent architecture. The integration involves two parts: extending `ExecutionContext` to reference a session and modifying the agent's `run()` method to load and save sessions.

EXTENDING EXECUTIONCONTEXT

The `ExecutionContext` we implemented in chapter 4 served as a container for all state within a single execution. It stored execution history in the `events` list and arbitrary runtime state in the `state` dictionary. With sessions introduced, these responsibilities shift. Events and state now live inside the `Session`, while `ExecutionContext` holds a reference to the current session.

The key design principle here is backward compatibility. Existing code extensively uses `context.events` and `context.state`. We use properties to provide transparent access, allowing this code to work without modification.

The following code shows the extended `ExecutionContext`. The new fields are `session` and `session_manager`. The `session` field references the session that the current execution belongs to, while `session_manager` handles session persistence and retrieval. The `events` and `state` properties internally return `session.events` and `session.state`, so existing code continues to work unchanged.

Listing 6.11 ExecutionContext with session support

```
@dataclass
class ExecutionContext:
    # Existing attributes...

    # Session support
    session: Session | None = None
    session_manager: BaseSessionManager | None = None

    @property
    def events(self) -> list[Event]:
        if self.session:
            return self.session.events

    @property
    def state(self) -> dict[str, Any]:
        if self.session:
            return self.session.state

    def add_event(self, event: Event):
        """Append an event to the execution history."""
        self.events.append(event)
        # Update session timestamp if available
        if self.session:
            self.session.updated_at = datetime.now()
```

MODIFYING AGENT INITIALIZATION

Next, we modify the `Agent` class constructor to accept a `SessionManager`. When provided, the agent stores it; when not provided, it defaults to `InMemorySessionManager`.

Listing 6.12 Agent initialization with SessionManager

```
class ToolCallingAgent:
    """Tool-calling agent with session support."""

    def __init__(
        self,
        model: LlmClient,
        tools: list[BaseTool] | None = None,
        instructions: str = "",
        max_steps: int = 10,
        session_manager: BaseSessionManager | None = None,
    ):
        self.model = model
        self.instructions = instructions
        self.max_steps = max_steps
        self.tools = self._setup_tools(tools or [])

        # Use provided session manager or create default
        self.session_manager = session_manager or InMemorySessionMan
ager()
```

MODIFYING THE RUN() METHOD

The most important changes occur in the `run()` method. Session support requires three additions. First, we accept a `session_id` parameter to specify which session to execute within. Second, we load or create the session at the start of execution. Third, we save the session when execution completes.

In the following code, notice the key changes. When `session_id` is provided, `get_or_create()` retrieves the session. If the session already exists, previous conversation history is contained in `session.events`. When creating `ExecutionContext`, we inject both the `session` and `session_manager`. After execution completes, `save()` persists the session.

Listing 6.13 run() method with session support

```
async def run(
    self,
    user_input: str,
    session_id: str | None = None,
    context: ExecutionContext | None = None,
) -> AgentResult:
    """Execute the agent with optional session support."""

    # Load or create session if session_id is provided
    session = None
    if session_id:
        session = await self.session_manager.get_or_create(session_i
d)

    # Create context with session support
    if context is None:
        context = ExecutionContext(
            session=session,
            session_manager=self.session_manager,
        )

    # Existing run() logic...

    # Save session if available
    if session:
        await self.session_manager.save(session)

    return AgentResult(output=context.final_result, context=context)
```

6.3.4 Basic example: Multi-turn conversation

Now that we've integrated Session and SessionManager into the agent, let's verify it works in practice. We create an agent with `InMemorySessionManager`, which stores sessions in a dictionary in memory. This implementation is suitable for testing and single-process applications.

Listing 6.14 Creating an agent with session support

```
agent = Agent(
    model=model,
    instructions="You are a helpful assistant.",
    session_manager=InMemorySessionManager()
)
```

WITHOUT SESSION_ID

What happens when we don't specify a `session_id` or use different values for each call? Each `run()` invocation is treated as an independent conversation, with no memory of previous interactions.

```
result1 = await agent.run("Hi! My name is Alice and I'm a software e
ngineer.")
print(f"Response 1: {result1.output}")

result2 = await agent.run("What's my name and what do I do?")
print(f"Response 2: {result2.output}")
```

The output will be similar to the following:

```
Response 1: Hello Alice! Nice to meet you. It's great to hear that
you're a software engineer. How can I help you today?

Response 2: I don't have any previous context about you.
Could you please tell me your name and what you do?
```

The agent has no memory of the first conversation in the second call.

WITH THE SAME SESSION_ID

Now let's use the same `session_id` for both calls. The two `run()` invocations share the same Session, preserving conversational context.

```
result1 = await agent.run(
    "Hi! My name is Alice and I'm a software engineer.",
    session_id="alice_session"
)

result2 = await agent.run(
    "What's my name and what do I do?",
    session_id="alice_session"
)
```

The output will be similar to the following:

```
Response 1: Hello Alice! Nice to meet you. It's great to hear that
you're a software engineer. How can I help you today?

Response 2: Your name is Alice, and you're a software engineer.
```

This time, the agent correctly recalls information from the previous interaction.

UNDERSTANDING THE INTERNAL FLOW

Let's examine what happens internally to understand the difference between these two scenarios.

When using the same session_id:

1. First `run()`: `get_or_create("alice_session")` creates a new Session.
2. The conversation is stored in `session.events`, and `save()` persists it.
3. Second `run()`: `get_or_create("alice_session")` returns the existing Session.
4. ExecutionContext references this Session, so `context.events` includes the previous conversation.
5. The LLM generates its response with full access to the prior context.

When session_id is not specified or differs:

1. Each `run()` call creates a new Session.
2. The new Session's `events` list is empty, so the LLM responds without prior context.

In practice, you'll use sessions when building conversational applications where users interact across multiple messages. Without sessions, each `run()` call is completely independent, which is appropriate for one-off tasks like GAIA benchmark problems.

6.3.5 Data structures for tool confirmation

Now let’s implement an asynchronous human-in-the-loop workflow. This will demonstrate that Session provides value far beyond simple chat history storage: it enables pausing and resuming execution at arbitrary points, which is essential for workflows requiring human approval.

To implement the workflow, we first need to define data structures. We need types that represent which tool calls are awaiting approval and what decisions the user has made.

PENDING TOOL CALL

`PendingToolCall` represents a tool call waiting to be executed. It's similar to the `ToolCall` we defined in chapter 4, but includes additional information needed for approval requests. The `tool_call` field preserves the original ToolCall object as-is, and the `confirmation_message` field contains the message to display to the user. For example: "This action will delete database.db. Do you want to continue?"

Listing 6.15 PendingToolCall class

```
class PendingToolCall(BaseModel):
    """A tool call awaiting user confirmation."""

    tool_call: ToolCall
    confirmation_message: str
```

TOOL CONFIRMATION

`ToolConfirmation` represents the user's approval decision. The `tool_call_id` identifies which tool call this response corresponds to, and the `approved` field indicates whether it was approved. `modified_arguments` is an optional field containing modified values if the user changed the arguments. For example, a user might request: "I approve the file deletion, but change the target file to temp.txt."

Listing 6.16 ToolConfirmation class

```
class ToolConfirmation(BaseModel):
    """User's decision on a pending tool call."""

    tool_call_id: str
    approved: bool
    modified_arguments: dict | None = None
```

EXTENDING AGENTRESULT

The `AgentResult` we implemented in chapter 4 was a container for agent execution results. The `output` field stored the final result, and the `context` field stored the ExecutionContext. To support human-in-the-loop, we need to distinguish whether execution has completed or is paused waiting for approval.

Add a `status` field to express this. "complete" means normal completion, "pending" means waiting for approval, and "error" means an error occurred. When the status is "pending", the `pending_tool_calls` field contains the list of tool calls awaiting approval.

Listing 6.17 AgentResult with pending support

```
from typing import Literal

class AgentResult(BaseModel):
    """Result of agent execution."""

    output: str | BaseModel | None = None
    context: ExecutionContext
    status: Literal["complete", "pending", "error"] = "complete"
    pending_tool_calls: list[PendingToolCall] = Field(default_factor
y=list)
```

SERIALIZING STATE WITH PYDANTIC

The Session's `state` field is of type `dict[str, Any]`, which can store arbitrary data. However, Python dictionaries cannot directly store Pydantic model objects. Problems arise when saving the session to a database or serializing it to JSON.

Pydantic's `model_dump()` and `model_validate()` methods solve this problem. `model_dump()` converts a Pydantic object to a dictionary, and `model_validate()` restores a dictionary back to a Pydantic object. Using this pattern, you can safely store and restore complex objects in Session's state.

The following code shows the pattern for storing and restoring pending tool calls. When the `act()` method discovers a tool requiring approval, it serializes with `model_dump()` and saves to state. On the next `run()` call, it restores with `model_validate()` and resumes execution.

```
# Saving: when pending occurs in act()
pending_calls = [PendingToolCall(tool_call=tc, confirmation_message=
msg)]
context.state["pending_tool_calls"] = [p.model_dump() for p in pendi
ng_calls]

# Restoring: when resuming run()
raw_pending = context.state.get("pending_tool_calls", [])
pending_calls = [PendingToolCall.model_validate(d) for d in raw_pend
ing]
```

This serialization pattern allows you to store various execution states without modifying the Session class. Beyond human-in-the-loop scenarios, you can store progress of long-running tasks, checkpoint data, intermediate results collected by the agent, and more in the same way.

Now the data structures are ready. Let’s add the approval requirement feature to tools themselves and implement the agent's pause and resume logic.

6.3.6 Extending tools for confirmation

How can we determine whether a tool requires approval? The most intuitive approach is to include this information in the tool itself. By extending the `BaseTool` class we implemented in chapter 4 to add confirmation-related attributes, the agent can check these attributes before executing a tool and pause execution when necessary.

EXTENDING BASETOOL

Let’s add two attributes to the `BaseTool` class: `requires_confirmation`, a boolean indicating whether the tool requires user approval before execution, and `confirmation_message_template`, a template for the approval request message shown to the user. The template uses `{name}` and `{arguments}` placeholders, which are dynamically filled with actual tool call information.

Listing 6.18 BaseTool with confirmation support

```
class BaseTool(ABC):
    # Existing attributes: name, description, parameters, execute
()...

    # Confirmation support
    requires_confirmation: bool = False
    confirmation_message_template: str = (
        "The agent wants to execute '{name}' with arguments: {argume
nts}. "
        "Do you approve?"
    )

    def get_confirmation_message(self, arguments: dict[str, Any]) ->
str:
        """Generate a confirmation message for this tool call."""
        return self.confirmation_message_template.format(
            name=self.name,
            arguments=arguments
        )
```

The `get_confirmation_message()` method fills the template with actual arguments to generate the final message. When the agent pauses tool execution, it includes this message in the `PendingToolCall` to present to the user.

DEFINING TOOLS THAT TEQUIRE CONFIRMATION

We also modify the `@tool` decorator implemented in chapter 4 to accept `requires_confirmation` and `confirmation_message` parameters. The complete modified code is available in the GitHub repository. Here is an example of defining tools that require confirmation using the updated decorator.

`delete_file` is configured with `requires_confirmation=True`, requiring user approval before execution. In contrast, `get_weather` is a read-only operation and executes immediately without approval. Using the `{arguments[key]}` format in custom messages allows including specific argument values in the message, enabling users to understand exactly what action will be performed before deciding whether to approve.

Listing 6.19 Tools with confirmation requirements

```
@tool(
    name="delete_file",
    description="Delete a file from the filesystem",
    requires_confirmation=True,
    confirmation_message="⚠ The agent wants to delete '{arguments[f
ilename]}'. "
                        "This action cannot be undone. Do you approv
e?"
)
def delete_file(filename: str) -> str:
    """Delete the specified file."""
    import os
    os.remove(filename)
    return f"Successfully deleted {filename}"


@tool(
    name="get_weather",
    description="Get current weather for a location"
)
def get_weather(location: str) -> str:
    """Get weather information. No confirmation needed."""
    return f"Weather in {location}: Sunny, 22°C"
```

6.3.7 Implementing pause and resume in the agent

Now that we've added the confirmation requirement attribute to tools, it's time to implement the logic for the agent to check this attribute and pause or resume execution. We'll modify the `act()` and `run()` methods to complete the human-in-the-loop workflow.

INTERNAL FEATURES VS EXTERNAL EXTENSIONS

Before diving into implementation, let's address an important design decision. We’ve explored how to customize agent behavior using callbacks, modifying prompts with `before_llm_callback` and post-processed tool results with `after_tool_callback`. These callbacks are external extension points injected by users.

However, Tool Confirmation is different. This isn't a feature that users optionally add; it's a core feature that the framework itself provides. The logic of stopping execution when encountering a tool requiring approval and resuming upon receiving approval is part of the agent's internal behavior.

For this reason, Tool Confirmation handling is implemented inside the `_prepare_llm_request()` method rather than as a callback. This method is responsible for the internal logic of constructing requests to send to the LLM, and it converts approval results into ToolResults to naturally include them in the conversation flow, as shown in figure 6.6.

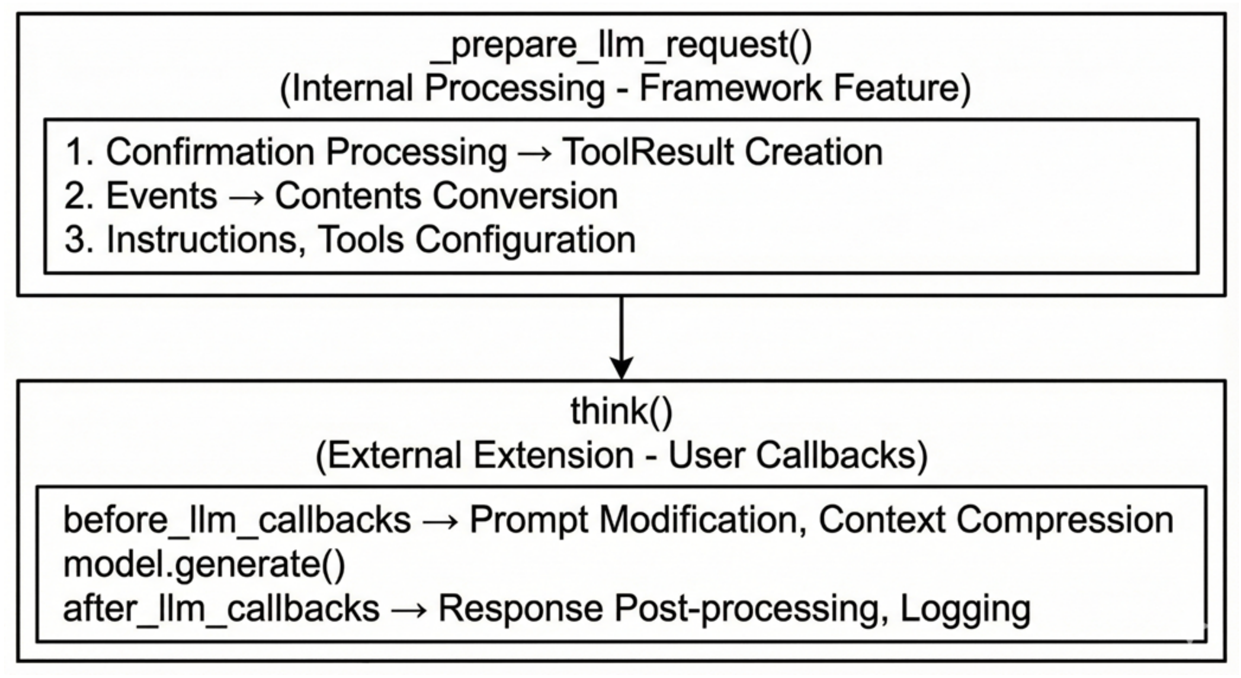

_prepare_llm_request()
(Internal Processing - Framework Feature)
1. Confirmation Processing → ToolResult Creation
2. Events → Contents Conversion
3. Instructions, Tools Configuration
think()
(External Extension - User Callbacks)
before_llm_callbacks → Prompt Modification, Context Compression
model.generate()
after_llm_callbacks → Response Post-processing, Logging

Figure 6.6 Internal features vs external extensions in agent architecture.

In this structure, `_prepare_llm_request()` handles the framework's essential logic, while callbacks handle user customization. With the two layers clearly separated, each can focus on its own responsibilities.

DESIGN PRINCIPLES

Before implementation, let's establish two core design principles.

Pre-execution Blocking: Tools requiring approval must receive user approval before execution. It's meaningless to execute a tool first, show the results, and then ask, "I performed this action. Is that okay?" Actions like file deletion or payment processing cannot be undone.

Whitelist Approval: Users can approve only some of the pending tool calls. When the agent requests three tool calls, the user should be able to approve two and reject one. Only tools included in the approval list are executed, and tools not in the list are automatically rejected.

MODIFYING THE ACT() METHOD

The `act()` method we implemented in chapter 4 immediately executed all tool calls returned by the LLM. Now we modify it to check the `requires_confirmation` attribute and store tools requiring approval in `context.state` instead of executing them.

Listing 6.20 Modified act() method with confirmation check

```
async def act(
    self,
    context: ExecutionContext,
    tool_calls: List[ToolCall]
) -> List[ToolResult]:
    tools_dict = {tool.name: tool for tool in self.tools}
    pending_calls = []

    for tool_call in tool_calls:
        tool = tools_dict.get(tool_call.name)

        # Check if confirmation is required
        if tool.requires_confirmation:
            pending = PendingToolCall(
                tool_call=tool_call,
                confirmation_message=tool.get_confirmation_message(
                    tool_call.arguments
                )
            )
            pending_calls.append(pending)
            continue  # Do not execute, do not create ToolResult

        # Execute tools that don't require confirmation
        # ... existing tool execution logic ...

    # Store pending calls in session state
    if pending_calls:
        context.state["pending_tool_calls"] = [
            p.model_dump() for p in pending_calls
        ]

    return results
```

Let's examine the key changes. As we iterate through tool calls, we check each tool's `requires_confirmation` attribute. For tools that require approval, we create a `PendingToolCall` object, add it to the `pending_calls` list, and move on to the next tool without creating a ToolResult. Only tools that don't require approval are executed immediately and return results.

The return type of `act()` remains the same `List[ToolResult]`. Since pending tool call information is stored in `context.state`, callers can check `context.state` to determine pending status. This approach adds new functionality while maintaining compatibility with the existing interface.

MODIFYING THE RUN() METHOD

The `run()` method must handle two scenarios. First, when a tool requiring approval is encountered during a new execution, it pauses execution and returns with a pending status. Second, when resuming a previously paused execution, it registers the user's approval decisions in `context.state` and continues execution.

Listing 6.21 Modified run() method with pause and resume

```
async def run(
    self,
    user_input: str | None = None,
    session_id: str | None = None,
    tool_confirmations: List[ToolConfirmation] | None = None,
    context: ExecutionContext | None = None,
) -> AgentResult:

    # Register confirmations for processing in _prepare_llm_request
    if tool_confirmations:
        context.state["tool_confirmations"] = [
            c.model_dump() for c in tool_confirmations
        ]

    # Main agent loop
    while not context.final_result and context.current_step < self.m
ax_steps:
        await self.step(context)

        # Check for pending confirmations after each step
        if context.state.get("pending_tool_calls"):
            pending_calls = [
                PendingToolCall.model_validate(p)
                for p in context.state["pending_tool_calls"]
            ]
            return AgentResult(
                status="pending_confirmation",
                context=context,
                pending_tool_calls=pending_calls,
            )
```

Let's review the key changes:

tool_confirmations parameter: Accepts a list of user approval decisions. When resuming from a previously pending state, this list is registered in `context.state`. The actual processing is performed in `_prepare_llm_request()` during the next `step()` call.

Pending check: After executing `step()` in the agent loop, check for pending tool calls in `context.state`. If present, immediately return with `pending_confirmation` status to pause execution.

Optional user_input parameter: When resuming a paused execution, only `tool_confirmations` is required; `user_input` is now optional.

MODIFYING _PREPARE_LLM_REQUEST()

The `_prepare_llm_request()` method processes approval decisions registered in `context.state`. It executes approved tools and records results as events, while logging rejection messages for rejected tools. This method is now async because `await` is required when executing approved tools.

Listing 6.22 Modified _prepare_llm_request() for confirmation processing

```
async def _prepare_llm_request(
    self,
    context: ExecutionContext
) -> LlmRequest:
    """Package context into an LlmRequest, processing confirmations
first."""

    # Process pending confirmations if both are present
    if ("pending_tool_calls" in context.state and
        "tool_confirmations" in context.state):

        confirmation_results = await self._process_confirmations(con
text)

        # Add results as an event so they appear in contents
        if confirmation_results:
            confirmation_event = Event(
                execution_id=context.execution_id,
                author=self.name,
                content=confirmation_results,
            )
            context.add_event(confirmation_event)

        # Clear processed state
        del context.state["pending_tool_calls"]
        del context.state["tool_confirmations"]

    # Flatten events into content items
    flat_contents = []
    for event in context.events:
        flat_contents.extend(event.content)

    # ... existing LlmRequest construction logic ...
```

The processing order is important. First, process approval decisions to generate `ToolResults` and record them as events; then construct `contents` including those events. This ensures the LLM can see the approval/rejection results in the next step.

This ordering is guaranteed because `_prepare_llm_request()` is an internal method. If we implemented this logic `before_llm_callback`, a timing issue would occur: newly added events wouldn't be included in `contents` because callbacks receive an already-constructed `LlmRequest`. This is another reason why framework internal features should be handled in internal methods rather than callbacks.

IMPLEMENTING CONFIRMATION PROCESSING LOGIC

The `_process_confirmations()` method performs the actual approval processing. For each pending tool call, it checks approval status, executes approved tools, and records rejection messages for unapproved cases (either explicit rejection or absence from the approval list).

Listing 6.23 _process_confirmations() implementation

```
async def _process_confirmations(
    self,
    context: ExecutionContext
) -> List[ToolResult]:
    tools_dict = {tool.name: tool for tool in self.tools}
    results = []

    # Restore pending tool calls from state
    pending_map = {
        p["tool_call"]["tool_call_id"]: PendingToolCall.model_valida
te(p)
        for p in context.state["pending_tool_calls"]
    }

    # Build confirmation lookup by tool_call_id
    confirmation_map = {
        c["tool_call_id"]: ToolConfirmation.model_validate(c)
        for c in context.state["tool_confirmations"]
    }

    # Process ALL pending tool calls
    for tool_call_id, pending in pending_map.items():
        tool = tools_dict.get(pending.tool_call.name)
        confirmation = confirmation_map.get(tool_call_id)

        if confirmation and confirmation.approved:
            # Merge original arguments with modifications
            arguments = {
                **pending.tool_call.arguments,
                **(confirmation.modified_arguments or {})
            }

            # Execute the approved tool
            try:
                output = await tool(context, **arguments)
                results.append(ToolResult(
                    tool_call_id=tool_call_id,
                    name=pending.tool_call.name,
                    status="success",
                    content=[output],
                ))
            except Exception as e:
                results.append(ToolResult(
                    tool_call_id=tool_call_id,
                    name=pending.tool_call.name,
                    status="error",
                    content=[str(e)],
                ))
        else:
            # Rejected: either explicitly or not in confirmation lis
t
            if confirmation:
                reason = confirmation.reason or "Tool execution was
rejected by user."
            else:
                reason = "Tool execution was not approved."

            results.append(ToolResult(
                tool_call_id=tool_call_id,
                name=pending.tool_call.name,
                status="error",
                content=[reason],
            ))

    return results
```

There are three key points in this implementation:

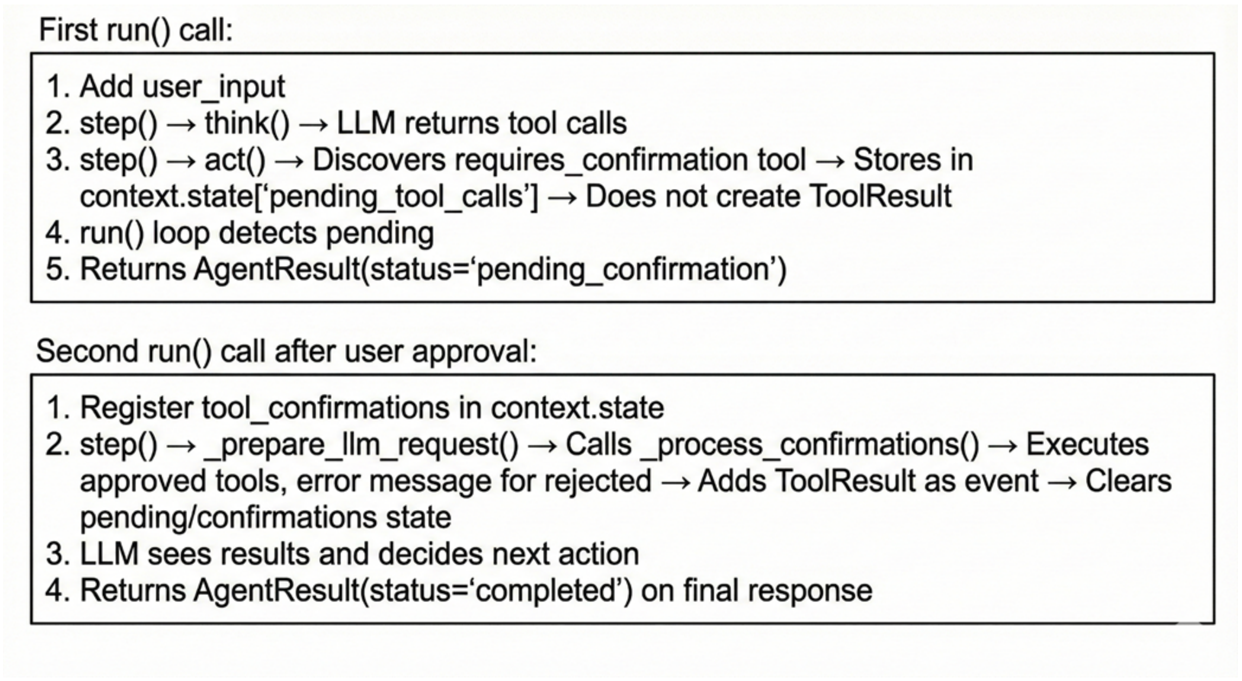
Whitelist approval: We iterate through all pending tool calls and execute only those that are in the approval list with `approved=True`. Any not in the approval list or with `approved=False` are treated as rejections.

Argument merging: When `modified_arguments` is provided, we merge it with the original arguments using `{**original, **modified}`. Modified arguments overwrite the originals, allowing users to selectively modify specific arguments, such as only updating the file path.

Clear feedback: Rejected tools are also recorded as `ToolResults`. Since the LLM expects results for tool calls, we must explicitly inform it when a tool is rejected so it can respond appropriately. For example, if a file deletion is rejected, the LLM can suggest alternative approaches or ask the user for clarification.

COMPLETE FLOW SUMMARY

Let's summarize the complete flow of the pause-and-resume mechanism we've implemented.



First run() call:

1. Add user_input
2. step() → think() → LLM returns tool calls
3. step() → act() → Discovers requires_confirmation tool → Stores in context.state['pending_tool_calls'] → Does not create ToolResult
4. run() loop detects pending
5. Returns AgentResult(status='pending_confirmation')

Second run() call after user approval:

1. Register tool_confirmations in context.state
2. step() → _prepare_llm_request() → Calls _process_confirmations() → Executes approved tools, error message for rejected → Adds ToolResult as event → Clears pending/confirmations state
3. LLM sees results and decides next action
4. Returns AgentResult(status='completed') on final response

Figure 6.7 Overall flow of the pause and resume mechanism.

6.3.8 Complete example: Human-in-the-loop workflow

Now let's combine all the components we've implemented to demonstrate a complete human-in-the-loop workflow. We'll create a simple scenario where the agent attempts to delete a file, pauses for user approval, and then resumes execution based on the user's decision.

SETTING UP THE EXAMPLE

First, we define a tool that requires confirmation and create an agent with session support. The `delete_file` tool is configured with `requires_confirmation=True`, so the agent will pause and wait for user approval before executing.

Listing 6.24 Setting up tools and agent for human-in-the-loop

```
from agent import Agent, InMemorySessionManager, ToolConfirmation
from tools import tool

@tool(
    name="delete_file",
    description="Delete a file from the filesystem",
    requires_confirmation=True,
    confirmation_message="⚠ Delete '{arguments[filename]}'? This ca
nnot be undone."
)
def delete_file(filename: str) -> str:
    import os
    os.remove(filename)
    return f"Deleted {filename}"

@tool(name="list_files", description="List files in a directory")
def list_files(directory: str = ".") -> str:
    import os
    return "\n".join(os.listdir(directory))

agent = Agent(
    model=model,
    tools=[delete_file, list_files],
    instructions="You are a helpful file management assistant.",
    session_manager=InMemorySessionManager()
)
```

STEP 1: INITIAL REQUEST

The user asks the agent to delete a temporary file. When the agent attempts to call the `delete_file` tool, execution pauses because confirmation is required.

Listing 6.25 Initial request triggering pending confirmation

```
result = await agent.run(
    "Please delete the file named temp.txt",
    session_id="file_session"
)

print(f"Status: {result.status}")
# Status: pending_confirmation

print(f"Pending tools: {len(result.pending_tool_calls)}")
# Pending tools: 1

for pending in result.pending_tool_calls:
    print(f"Tool: {pending.tool_call.name}")
    print(f"Arguments: {pending.tool_call.arguments}")
    print(f"Message: {pending.confirmation_message}")
# Tool: delete_file
# Arguments: {'filename': 'temp.txt'}
# Message: ⚠ Delete 'temp.txt'? This cannot be undone.
```

The agent returns immediately with `status="pending_confirmation"`. The `pending_tool_calls` list includes details about the tool awaiting approval, such as the confirmation message shown to the user.

STEP 2: USER APPROVAL

In a real application, the user might click a button on a web page or respond to a mobile notification. Here, we simulate approval by creating a `ToolConfirmation` object and resuming execution.

Listing 6.26 Resuming execution with user approval

```
# User approves the deletion
confirmation = ToolConfirmation(
    tool_call_id=result.pending_tool_calls[0].tool_call.tool_call_i
d,
    approved=True
)

result = await agent.run(
    session_id="file_session",
    tool_confirmations=[confirmation]
)

print(f"Status: {result.status}")
# Status: complete

print(f"Output: {result.output}")
# Output: I've deleted the file temp.txt as requested.
```

When we call `run()` again with the same `session_id` and the approval decision, the agent resumes execution. It executes the approved tool, receives the result, and generates a final response.

6.4 Long-term memory: Accumulating knowledge across sessions

The Session and SessionManager allowed us to continue conversations within the same session_id. However, once a session completely ends, its information is lost. In this section, we implement long-term memory to accumulate and utilize knowledge beyond session boundaries.

Long-term memory is a prime example of combining Context Engineering's Write (storage) and Retrieval (search) strategies. After a session ends, important information is saved to external storage (Write), and in new sessions, this information is retrieved and added to the context (Retrieval).

6.4.1 The structure of long-term memory

The SessionManager we implemented and the MemoryManager introduced here address different problems.

SessionManager maintains conversation continuity within a session. When you call run() multiple times with the same session_id, the previous conversation content is preserved. However, starting a new conversation with a different session_id prevents access to the previous session content.

MemoryManager stores important information even after a session ends, allowing retrieval and use in future sessions. Examples include "the user mentioned they're a marketer" or "what approach was used when solving this problem before."

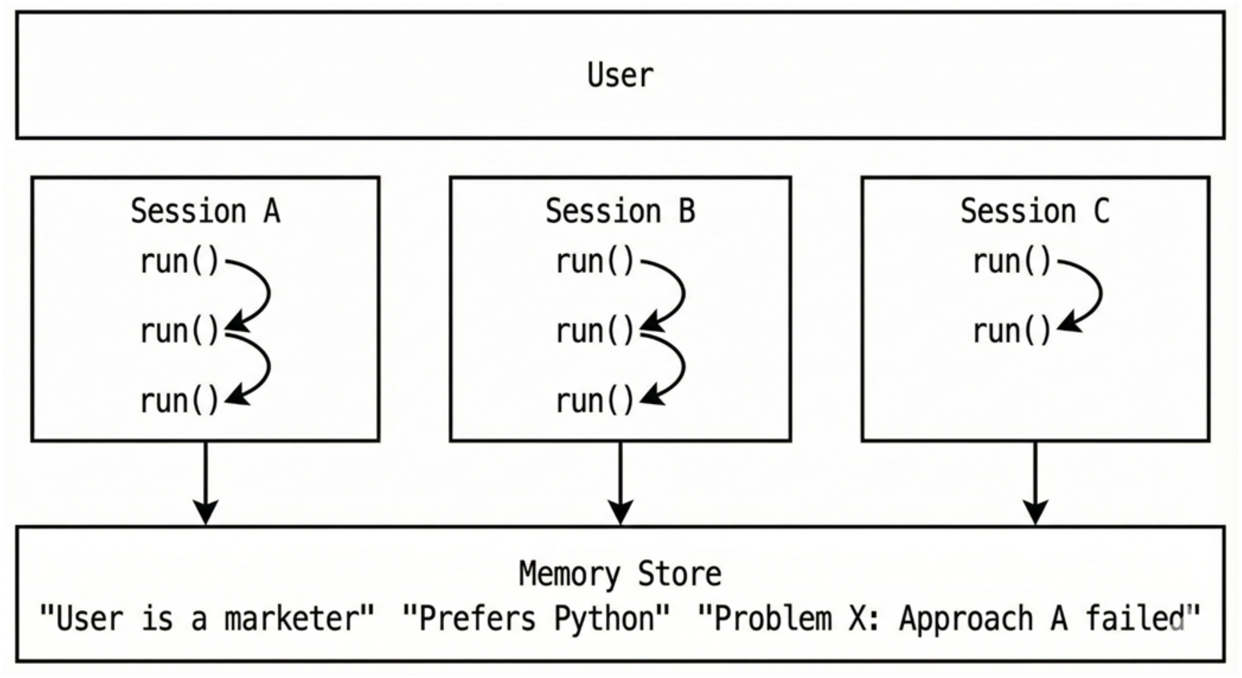


Figure 6.8 The role of MemoryManager.

SessionManager handles horizontal continuity within sessions (conversation flow), while MemoryManager handles vertical continuity across sessions (knowledge accumulation).

THE TWO PILLARS: STORAGE AND RETRIEVAL

Long-term memory systems have two core operations: storage (Write) and retrieval. The simplest approach is to convert text directly into vector embeddings for storage, then find and return vectors similar to the query text during retrieval. However, storing entire conversations introduces noise, and important information can get buried.

Recent memory libraries like mem0 and graphiti improve on this by using LLMs to extract only information worth preserving. When storing newinformation, they perform one of four operations to manage conflicts with existing memories.

- ADD stores new information that doesn't already exist.
- UPDATE modifies existing memories.
- DELETE removes contradictory memories.
- SKIP ignores information that's already stored.

In this chapter, we implement only ADD and SKIP. User profile information (name, occupation, preferences, etc.) can change over time and requires UPDATE and DELETE, but problem-solving records are different. Even forthe same problem, different approaches or results are worth storing as separate records (ADD), while completely identical attempts can be ignored (SKIP). There's no need to modify or delete past failure records; keeping them helps avoid repeating the same mistakes. This immutability of memories simplifies our implementation.

6.4.2 Information extraction: Structured output

If the information to extract from memory is simple facts, text format is sufficient. Information that fits in a single sentence, like "the user is a marketer" or "prefers Python" can be stored and searched as-is.

Sometimes the information to store is more complex. Consider GAIA problem-solving records: instead of just "solved this problem," we need to store what the problem was, what approach was used, and the result. This information must be structured so the agent can reference past successful approaches or avoid failed ones when encountering similar problems.

EXTENDING LLMCLIENT

We add two methods to LlmClient for information extraction and vector embeddings. The `generate()` method from chapter 4 takes LlmRequest as input and is specialized for agent execution. Simpler interfaces are needed for basic text generation.

The `ask()` method takes a prompt and returns text or JSON. When a `response_format` parameter is provided, it returns output in the requested format.

```
# Simple text generation
answer = await llm_client.ask("What is 2 + 2?")

# JSON structured output
data = await llm_client.ask(prompt, response_format=GaiaOutput)
```

DEFINING THE MEMORY SCHEMA

We use Pydantic to define a schema for GAIA problem-solving records. The schema captures the essential elements of a problem-solving attempt: `task_summary` describes what the problem asked, `approach` records the methods and tools used to solve it, `final_answer` stores the agent's submitted answer, `is_correct` indicates whether the answer was right, and `error_analysis` captures why the attempt failed when it did.

Listing 6.27 TaskMemory schema definition

```
class TaskMemory(BaseModel):
    """Structured memory for GAIA problem-solving records"""

    task_summary: str = Field(description="What the problem asked")
    approach: str = Field(description="Methods and tools used to sol
ve it")
    final_answer: str = Field(description="The agent's submitted ans
wer")
    is_correct: bool = Field(description="Whether the answer was cor
rect")
    error_analysis: str | None = Field(
        default=None,
        description="Why the attempt failed, if it did"
    )

    def to_embedding_text(self) -> str:
        """Generate text for vector search"""
        return f"Task: {self.task_summary}"
```

The `to_embedding_text()` method generates text for vector search. Here, it includes only the problem summary so that similar problems can be found during retrieval. The approach and answer are stored as metadata and used after retrieval.

EXTRACTION PROMPT DESIGN

We design a prompt that extracts structured memories from the agent's execution history. The prompt instructs the LLM to analyze the complete execution trace and identify the key elements: what problem was being solved, what approach was taken, and whether it succeeded.

```
TASK_MEMORY_EXTRACTION_PROMPT = """You are analyzing an agent's prob
lem-solving record to extract a learning record.

Analyze the following execution history:

<execution_history>
{execution_history}
</execution_history>
"""
```

The execution history contains the agent's reasoning process, tool calls, intermediate results, and feedback added by the user indicating whether the answer was correct. The LLM synthesizes all of this into a structured memory.

DUPLICATE CHECK PROMPT

Before storing a new memory, we check whether similar information already exists. This prevents the memory store from filling with redundant entries about the same problem.

Listing 6.28 DuplicateCheckResult schema and prompt

```
class DuplicateCheckResult(BaseModel):
    """Result of duplicate check"""
    decision: str = Field(description="ADD (new information) or SKIP
(duplicate)")
    reason: str = Field(description="Explanation for the decision")

DUPLICATE_CHECK_PROMPT = """Compare the new memory against existing
memories to determine if it's a duplicate.

Existing memories:
{existing_memories}

New memory:
{new_memory}

Respond with one of:
- ADD: This is new information that should be stored
- SKIP: Similar information already exists, no need to store

Judgment criteria:
- Same problem with different approach or different result counts as
new information
- Same problem with same approach and same result is a duplicate
"""
```

The prompt provides existing memories as context and asks the LLM to make a binary decision. The `DuplicateCheckResult` schema captures both the decision and the reasoning behind it.

6.4.3 Building a vector store with ChromaDB

In chapter 5, we learned the principles of vector search through hands-on implementation. We coveredthe process of calculating cosine similarity and returning top results. Now we’ll implement storage and retrieval using a vector database. For our practice, we use ChromaDB, which is lightweight and simple to configure.

INSTALLING CHROMADB AND CORE CONCEPTS

ChromaDB is a lightweight vector database that runs locally. You can use it immediately by installing the Python package, with no separate server setup required, making it ideal for prototyping and learning.

```
uv add chromadb
```

The core concept in ChromaDB is the Collection. A Collection groups related documents, similar to a table in relational databases. Each Collection stores document text, embedding vectors, and metadata together.

BASIC USAGE

The following code demonstrates ChromaDB's basic storage and retrieval flow. We create an in-memory client with `chromadb.Client()` and a collection with `get_or_create_collection()`. When you specify an embedding function in the `embedding_function` parameter, it automatically converts text to vectors during both storage and retrieval.

Listing 6.29 ChromaDB basic usage

```
import chromadb
from chromadb.utils.embedding_functions import OpenAIEmbeddingFuncti
on

# Create client (in-memory)
client = chromadb.Client()

# Set up embedding function
embedding_fn = OpenAIEmbeddingFunction(
    model_name="text-embedding-3-small"
)

# Create collection
collection = client.get_or_create_collection(
    name="my_memories",
    embedding_function=embedding_fn
)

# Store documents - just pass text, embeddings are generated automat
ically
collection.add(
    ids=["mem_001", "mem_002"],
    documents=["The cat is sleeping on the sofa", "The dog is playin
g in the park"],
    metadatas=[{"category": "cat"}, {"category": "dog"}]
)

# Search - just pass query text, embeddings are generated automatica
lly
results = collection.query(
    query_texts=["Where is the cat resting"],
    n_results=1
)

print(results["documents"])  # [['The cat is sleeping on the sofa']]
print(results["metadatas"])  # [[{'category': 'cat'}]]
```

When you specify an `embedding_function`, the document text is automatically embedded when calling `add()`, and `query_texts` are also automatically embedded when calling `query()` to perform similarity search. The embedding generation and cosine similarity calculation are all handled internally.

USING METADATA

The `metadatas` parameter stores additional information for each document. You can use this metadata in search results to check supplementary information such as document category, creation time, or source. In the memory store we implement in the next section, we store information like approach and correctness as metadata.

6.4.4 Implementing TaskMemoryManager

Now we implement `TaskMemoryManager`, which stores and retrieves GAIA problem-solving records using ChromaDB. This class handles the entire pipeline of extracting structured memories from execution history, checking for duplicates, and storing them in ChromaDB.

CLASS INITIALIZATION

The `TaskMemoryManager` constructor receives an `LlmClient` through injection, which is used for memory extraction and duplicate checking. The ChromaDB client and collection are also initialized here.

Listing 6.30 TaskMemory Manager initialization

```
import uuid
import chromadb
from chromadb.utils.embedding_functions import OpenAIEmbeddingFuncti
on


class TaskMemoryManager:
    """Memory manager for GAIA problem-solving learning"""

    def __init__(
        self,
        llm_client: LlmClient,
        collection_name: str = "task_memories",
    ):
        self.llm_client = llm_client

        # ChromaDB setup
        self.client = chromadb.Client()
        embedding_fn = OpenAIEmbeddingFunction(
            model_name="text-embedding-3-small"
        )
        self.collection = self.client.get_or_create_collection(
            name=collection_name,
            embedding_function=embedding_fn
        )
```

MEMORY EXTRACTION METHODS

We implement methods to extract structured memories from execution history. The `_format_execution_history()` converts the event list into a text format suitable for the LLM. It iterates through each event's content and formats messages, tool calls, and tool results separately.

In `_extract_memory()`, we pass `response_format=TaskMemory` so the LLM generates JSON matching the TaskMemory schema. Since `LlmClient` parses this and returns a `TaskMemory` instance, no separate JSON parsing logic is needed.

Listing 6.31 Memory extraction methods

```
class TaskMemoryManager:
    # ... previous code ...

    async def _extract_memory(self, execution_history: str) -> TaskM
emory | None:
        """Extract structured memory from execution history"""
        prompt = TASK_MEMORY_EXTRACTION_PROMPT.format(
            execution_history=execution_history
        )

        try:
            return await self.llm_client.ask(
                prompt=prompt,
                response_format=TaskMemory
            )
        except Exception as e:
            print(f"Memory extraction failed: {e}")
            return None

    def _format_execution_history(self, events: list[Event]) -> str:
        """Convert event list to text"""
        lines = []
        for event in events:
            for item in event.content:
                if isinstance(item, Message):
                    lines.append(f"[{item.role}]: {item.content}")
                elif isinstance(item, ToolCall):
                    lines.append(f"[Tool Call]: {item.name}({item.ar
guments})")
                elif isinstance(item, ToolResult):
                    content_preview = str(item.content[0])[:500] if
item.content else ""
                    lines.append(f"[Tool Result]: {item.name} -> {co
ntent_preview}")
        return "\n".join(lines)
```

DUPLICATE CHECK METHOD

This method checks whether a new memory duplicates an existing one. If there are no existing search results, it returns False. It converts existing memories to text and includes them, along with the new memory, in the prompt. When a response is received with the `DuplicateCheckResult` schema, the decision can be accessed directly through `result.decision`. If the decision is "SKIP", it means the memory is a duplicate and should not be stored, so the method returns True to prevent storage.

Listing 6.32 Duplicate check method

```
class TaskMemoryManager:
    # ... previous code ...

    async def _is_duplicate(
        self,
        new_memory: TaskMemory,
        existing_results: dict
    ) -> bool:
        """Determine if a new memory duplicates an existing one"""
        if not existing_results["metadatas"] or not existing_results
["metadatas"][0]:
            return False

        # Convert existing memories to text
        existing_texts = []
        for meta in existing_results["metadatas"][0]:
            existing_texts.append(
                f"task_summary: {meta.get('task_summary')}, "
                f"- approach: {meta.get('approach')}, "
                f"is_correct: {meta.get('is_correct')}"
            )

        prompt = DUPLICATE_CHECK_PROMPT.format(
            existing_memories="\n".join(existing_texts),
            new_memory=f"task_summary: {new_memory.task_summary}, "
                       f"approach: {new_memory.approach}, "
                       f"is_correct: {new_memory.is_correct}"
        )

        try:
            result = await self.llm_client.ask(
                prompt=prompt,
                response_format=DuplicateCheckResult
            )
            return result.decision == "SKIP"
        except Exception:
            return False  # Proceed with storage if parsing fails
```

SAVE METHOD

The `save()` method brings all components together. It converts the execution history to text, extracts structured memory using the LLM, searches for similar existing memories to check for duplicates, and stores the memory in ChromaDB if it is not a duplicate. For documents, it generates searchable text using `to_embedding_text()`. For metadata, it converts all `TaskMemory` fields to a dictionary using `model_dump()` for storage.

Listing 6.33 TaskMemoryManager save() method

```
class TaskMemoryManager:
    # ... previous code ...

    async def save(self, context: ExecutionContext) -> str | None:
        """
        Extract and save memory from execution context.

        Returns:
            memory_id if saved, None if ignored as duplicate
        """
        # 1. Convert execution history to text
        execution_history = self._format_execution_history(context.e
vents)

        # 2. Extract structured memory using LLM
        memory = await self._extract_memory(execution_history)
        if memory is None:
            return None

        # 3. Duplicate check - search with the same text used for st
orage
        existing = self.collection.query(
            query_texts=[memory.to_embedding_text()],
            n_results=3
        )
        if await self._is_duplicate(memory, existing):
            return None

        # 4. Store in ChromaDB
        memory_id = str(uuid.uuid4())
        self.collection.add(
            ids=[memory_id],
            documents=[memory.to_embedding_text()],
            metadatas=[memory.model_dump()]
        )
        return memory_id
```

SEARCH METHOD

The `search()` method retrieves memories related to a query and returns them as a list of TaskMemory objects. When we call `collection.query()`, ChromaDB handles embedding generation, similarity calculation, and sorting. By converting the result metadata to TaskMemory, we can access the `task_summary`, `approach`, `final_answer`, `is_correct`, and `error_analysis` fields in a type-safe way.

Listing 6.34 TaskMemoryManager search() method

```
class TaskMemoryManager:
    # ... previous code ...

    async def search(self, query: str, top_k: int = 5) -> list[TaskM
emory]:
        """Search for memories related to the query."""
        results = self.collection.query(
            query_texts=[query],
            n_results=top_k
        )

        if not results["metadatas"] or not results["metadatas"][0]:
            return []

        return [TaskMemory(**meta) for meta in results["metadatas"]
[0]]
```

INTEGRATING MEMORYMANAGER INTO THE AGENT

We add `memory_manager` to the Agent class. The `memory_manager` is optional. For simple tasks that do not require long-term memory, it can be left as None. In the `run()` method, we pass the `memory_manager` to ExecutionContext so that tools can access it via `context.memory_manager`.

Listing 6.35 Adding memory_manager to Agent

```
class Agent:
    def __init__(
        ...
        memory_manager: TaskMemoryManager | None = None,  # Added
    ):
        ...
        self.memory_manager = memory_manager  # Added
```

6.4.5 Retrieving memories

The most intuitive way to use stored memories is to implement retrieval as a tool that the agent can call when needed. But what if certain memories should always be retrieved before the LLM sees the request? For example, retrieving user profile information at the start of every conversation or checking whether a similar problem was solved before. You could implement a separate callback for this, but then memory-related logic would be split between a Tool and a callback. A cleaner approach is to let the Tool itself inject context into the LLM request.

TOOL-BASED RETRIEVAL

We start with a MemoryTool that retrieves past problem-solving records. The agent can call this tool to check for similar problems. The `execute()` method searches the memory store and formats results so the LLM can understand what approaches worked and which ones failed.

Listing 6.36 MemoryTool for explicit retrieval

```
class MemoryTool(BaseTool):
    """Tool for retrieving past problem-solving records"""

    def __init__(self):
        super().__init__(
            name="recall_memory",
            description=(
                "Search for past problem-solving records. "
                "Use this to check if similar problems were solved b
efore."
            ),
        )

    async def execute(self, context: ExecutionContext, query: str) -
> str:
        """Search memories and return formatted results"""
        memories = await context.memory_manager.search(query, top_k=
3)

        if not memories:
            return ""

        return self._format_memories(memories)

    def _format_memories(self, memories: list[TaskMemory]) -> str:
        """Format memories for display"""
        results = []
        for i, mem in enumerate(memories, 1):
            status = "Correct" if mem.is_correct else "Incorrect"
            text = f"""[Record {i}]
- Problem: {mem.task_summary}
- Approach: {mem.approach}
- Answer: {mem.final_answer}
- Result: {status}"""
            if not mem.is_correct and mem.error_analysis:
                text += f"\n- Error analysis: {mem.error_analysis}"
            results.append(text)
        return "\n\n".join(results)
```

With this tool registered, the agent can decide when to search for relevant experiences.

AUTOMATIC INJECTION WITH PROCESS_LLM_REQUEST

Tool-based retrieval works well when searching is optional. But for some use cases, you want memories retrieved automatically before every LLM call. Instead of creating a separate callback, override the `process_llm_request()` method in BaseTool. This method runs before each LLM request and can modify the request directly.

The key insight is that you can reuse the `execute()` logic inside `process_llm_request()`. Rather than duplicating the search and formatting code, `process_llm_request()` simply `calls execute()` and appends its result to the LLM request.

Whether the tool is callable by the LLM or only used for automatic injection depends on `tool_definition`. If `tool_definition` is a dict, the LLM can call it explicitly. If `tool_definition` is None, the tool won't appear in the LLM's tool list, but `process_llm_request()` still runs before every LLM call. For simplicity, we use automatic injection here by setting `tool_definition=None`.

Listing 6.37 MemoryTool with automatic injection

```
class MemoryTool(BaseTool):
    # ... execute, _format_memories same as above ...

    def __init__(self):
        super().__init__(
            name="recall_memory",
            description="...",
            tool_definition=None,  # Automatic injection only
        )

    async def process_llm_request(
        self,
        context: ExecutionContext,
        request: LlmRequest,
    ) -> None:
        """Inject relevant memories before LLM call"""
        user_msgs = [c for c in request.contents if isinstance(c, Me
ssage) and c.role == "user"]
        if not user_msgs:
            return

        result = await self.execute(context, user_msgs[-1].content)
        if not result:
            return

        request.append_instructions(
            f"""The following are records from similar problems solv
ed in the past:

<PAST_EXPERIENCES>
{result}
</PAST_EXPERIENCES>

Reference successful approaches and avoid approaches that led to fai
lures."""
        )
```

The only change from the previous version is adding `tool_definition=None` in **init**. This makes the tool invisible to the LLM, but its `process_llm_request()` method still runs before every LLM call, automatically injecting relevant memories. If you want the LLM to call this tool explicitly instead, simply remove the `tool_definition=None` line.

INTEGRATING INTO THE AGENT

To make `process_llm_request()` work, we modify the Agent's `_prepare_llm_request()` method. In chapter 4, we built the tools list with a simple comprehension:

```
tools = [t.tool_definition for t in request.tools]
```

Now we need two changes. First, filter out tools where `tool_definition` is None, since some tools may only provide automatic injection without being callable by the LLM. Second, call each tool's `process_llm_request()` method. Note that the method becomes async since `process_llm_request()` is asynchronous.

Listing 6.38 Modified _prepare_llm_request() for tool injection

```
class Agent:
    # ... existing code ...

    async def _prepare_llm_request(
        self, context: ExecutionContext
    ) -> LlmRequest:
        flat_contents = []
        for event in context.events:
            flat_contents.extend(event.content)

        # Filter tools that should be exposed to the LLM
        llm_tools = [t for t in self.tools if t.tool_definition is n
ot None]

        request = LlmRequest(
            instructions=[self.instructions] if self.instructions el
se [],
            contents=flat_contents,
            tools=llm_tools,
            tool_choice="auto" if llm_tools else None,
        )

        # Let tools modify the request
        for tool in self.tools:
            await tool.process_llm_request(context, request)

        return request
```

The filtering logic allows a tool to participate in `process_llm_request()` without appearing in the LLM's tool list. MemoryTool uses this pattern: it sets `tool_definition=None` so the LLM cannot call it, but its `process_llm_request()` still runs to inject memories automatically.

6.5 Summary

- Memory transforms an agent from a stateless tool into an intelligent assistant that learns and adapts across interactions.

- The three usage patterns address different challenges: Pattern 1 (One-off Execution) manages context within a single run, Pattern 2 (Continuous Execution) maintains conversation across multiple runs, and Pattern 3 (Long-term Memory) preserves knowledge across sessions.
- Context management strategies follow a hierarchical approach: measure tokens first, apply Compaction for tool results, and use Summarization as a last resort. The key principle is separating storage from presentation—ExecutionContext remains immutable while LlmRequest is optimized for each call.
- Session and SessionManager enable multi-turn conversations by persisting events and state across run() calls. The same architecture supports asynchronous Human-in-the-Loop workflows, where execution pauses for user approval and resumes when confirmation arrives.
- Long-term memory combines Write and Retrieval strategies: structured information is extracted from execution history, checked for duplicates, stored in a vector database, and retrieved when relevant to new tasks.
- Memory retrieval can be implemented as an explicit tool the agent calls on demand, or as automatic injection via process_llm_request() that adds relevant context before every LLM call.

7 Planning and reflection for complex tasks

This chapter covers

- The limitations of reactive agents on complex tasks
- Planning tools that decompose problems into task lists
- Reflection tools that enable progress checking and failure recovery
- The complementary cycle of planning and reflection

We’ve built an agent that observes the current situation, decides which tool to call, and repeats this cycle. This reactive approach works well for simple tasks but struggles with complex problems requiring multiple steps. The agent loses direction, forgets what it has done, or fails to recover from errors. Planning and reflection address these limitations by giving agents "time to think." Planning decomposes complex problems into manageable tasks before execution. Reflection pauses to check progress and adjust direction when things go wrong.

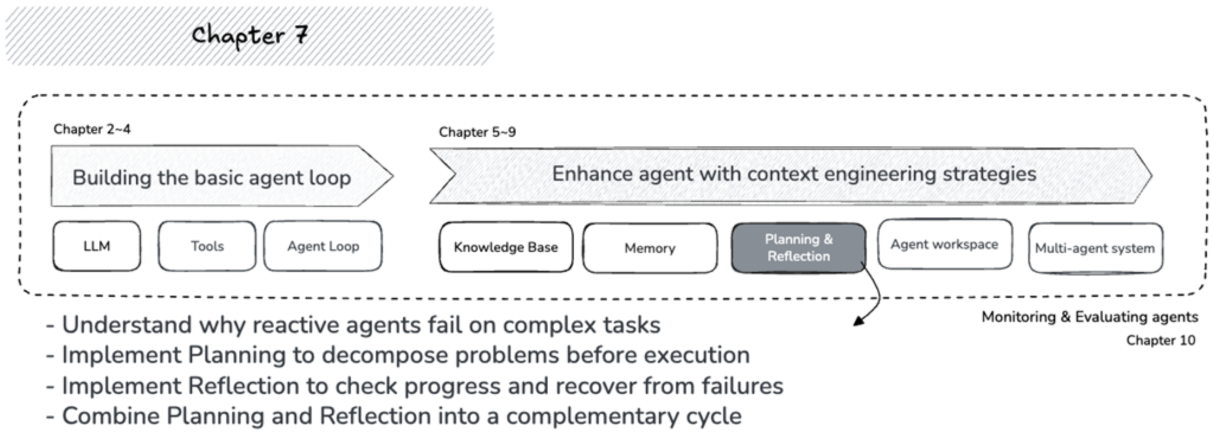

Chapter 7
Chapter 2~4
Building the basic agent loop
LLM
Tools
Agent Loop
Chapter 5~9
Enhance agent with context engineering strategies
Knowledge Base
Memory
Planning & Reflection
Agent workspace
Multi-agent system
Monitoring & Evaluating agents
Chapter 10
- Understand why reactive agents fail on complex tasks
- Implement Planning to decompose problems before execution
- Implement Reflection to check progress and recover from failures
- Combine Planning and Reflection into a complementary cycle

Figure 7.1 The roles of planning and reflection in AI agents.

We'll examine why reactive agents fail and how human experts approach complex tasks differently. We'll then implement a simple Planning tool that records task lists in the context, followed by a reflection tool that enables progress checking and failure recovery. Finally, we combine both strategies into a complementary cycle and see how they resolve the failure modes of reactive agents.

7.1 Giving agents time to think

Planning and reflection are two core strategies that give agents time to think. We'll first examine the limitations of the ReAct approach, then find solutions in how human experts work. We'll also explore how the "time to think" principle, validated in LLM research, applies to agents, and understand what roles planning and reflection play from a Context Engineering perspective.

7.1.1 The limitations of ReAct

The agent we've implemented operates using the ReAct approach. It observes the current situation, decides which tool to call, observes the result again, and decides on the next action. This approach excels at adaptability. If search results differ from expectations, it tries different search terms; if a tool fails, it looks for alternatives. This is because it selects whatever action seems most appropriate at each moment.

However, this approach is similar to driving without GPS. At every intersection, you go in "whichever direction looks best from what you can see right now." This works fine when the destination is close, but when you need to find a complex route, you easily get lost.

In complex problems, ReAct agents exhibit several typical failure modes. They lose direction by trying to do too much at once. They execute multiple web searches to quickly retrieve information needed for a task, then fail to properly utilize the results. ReAct agents forget what they did in the middle, failing to use information found previously, resulting in repeated searches. They hastily declare, "I found the answer," but attempt to answer with only partial information. They fail to recover from failures. When tool calls fail or go in the wrong direction, instead of analyzing the cause and finding alternatives, they repeat the same attempts until reaching max_steps.

What do these problems have in common? The agent only thinks about "what to do right now" and doesn't consider "how to approach this overall" or "whether things are progressing well so far."

7.1.2 How human experts work

Consider how a skilled researcher handles a complex question. When asked, "How many hours would it take to reach the Moon at Kipchoge's marathon pace?" they don't start searching immediately.

First, they decompose the question: "What's Kipchoge's marathon record? I can calculate the pace from that. What's the distance to the Moon? I should use the closest approach distance." Then they investigate one thing at a time, organizing along the way: "Found the marathon record, converting to kilometers per hour gives about 21km/h." When unexpected results appear, they adjust their approach: "The Moon distance varies across sources. Should I use the average distance or the perigee? The question asked about 'closest approach,' so I'll use the perigee." Before the final answer, they check for anything missing: "Marathon record, pace calculation, Moon distance, time calculation... all done."

The same pattern appears in software development. Experienced developers don't start writing code immediately. They first write a PRD (Product Requirements Document) or spec document, then proceed with implementation and testing while referencing that document. When problems arise during implementation, they revisit the spec and adjust their approach. Coding agents like Claude Code and Cursor follow this method too. When implementing complex features, they first create an implementation plan, then check progress against the plan at each stage. This is exactly planning (writing specs) and reflection (checking against specs).

Anthropic's agent development guide emphasizes this point:

> *"Inspiration for these practices came from knowing what effective software engineers do every day."*
>
> *— Anthropic, "Building effective agents."*

The message is to design agents inspired by how human experts work.

7.1.3 Why time to think matters

A pattern repeatedly confirmed in LLM research is that giving time to think improves performance compared to requiring immediate answers. Chain-of-Thought prompting significantly improves math problem accuracy just by adding the phrase "Let's think step by step." This works because it makes the model explicitly generate intermediate reasoning steps. Reasoning Models took this principle further, generating thinking tokens before answers, greatly enhancing complex problem-solving capabilities.

Planning and reflection apply this principle to agents. Planning is time to think before acting: "What should I do? In what order should I proceed?" Reflection is time to look back after acting: "What have I learned so far? Am I on the right track? Should I change direction?" This grants agents a form of metacognition: the ability to examine their own thinking process.

From a context engineering perspective, planning and reflection share an interesting commonality. Neither directly performs tasks. They don't search the web, perform calculations, or read files. Instead, they generate text signals that control the flow of work. Planning adds "what to do next" to the context. When text like "1. Research Kipchoge's record, 2. Research Moon distance, 3. Calculate time" enters the context, the LLM references this plan in subsequent steps to maintain direction. Reflection adds "evaluation and direction so far" to the context. Text like "Marathon record acquired. Moon distance research: Wikipedia failed, need to switch to web search" influences the LLM's next decision.

Among the five context engineering strategies introduced in chapter 1, Planning and Reflection belong to the generation strategy. Unlike retrieval, which brings information from external sources, or write, which stores information externally, this approach has the LLM generate text within the context itself to control the workflow. Now let's examine how to implement these two strategies.

7.2 Planning: Setting direction

The core of planning is decomposing complex problems into manageable units. Even the to-do lists we use in daily life are a small form of decomposition. Breaking down a large task like "write a report" into "gather materials → write draft → review and revise" makes it much easier to handle.

Let's use the Kipchoge problem as an example.

Original question: "How many thousand hours would it take to reach the Moon at Kipchoge's marathon pace?"

What happens if an agent receiving this question starts searching immediately? It searches "Kipchoge moon hours," and when the desired results don't appear, it tries different search terms and easily loses direction in the process. Instead, decomposing the question yields:

Decomposed tasks:

1. Research Kipchoge's marathon record → Calculate pace
2. Research the distance to the Moon (perigee)
3. Calculate time = distance ÷ speed
4. Round to thousands of hours

With this decomposition, each step becomes clear, and the completion criteria become obvious. You can easily judge the success of each step with questions like "Did I find the marathon record?" or "Did I confirm the Moon distance?"

7.2.1 When is planning necessary?

Not every problem requires planning. Since planning requires LLM calls, using it in unnecessary situations becomes overhead.

When planning is necessary:

- Problems requiring multiple steps of research (e.g., the Kipchoge problem).
- Problems requiring combining multiple pieces of information (e.g., comparing stock prices of two companies).

When planning is unnecessary:

- Simple questions solvable with a single search (e.g., "What's the weather in Seoul today?").
- Tasks with already clear procedures (e.g., "Translate this text for me").

Creating a plan for "What's the weather today?" is wasteful. For a problem that can be solved with one search, creating a plan like "1. Confirm location, 2. Search weather, 3. Organize results" wastes unnecessary tokens and time.

To have agents make this distinction themselves, the planning tool's description must clearly state "when to use and when not to use it," as we'll see in our implementation.

7.2.2 Implementing the planning tool

Let's create the simplest form of a planning tool. Before implementing, we need to make a few design decisions.

Should we manage only a single plan or multiple plans simultaneously?

In complex systems, you might need to separate main plans from sub-plans or track multiple goals simultaneously. But in most cases, one plan per question is sufficient. Managing multiple plans requires additional logic to decide "which plan to reference." Let's start simple.

When modifying a plan, should we update only the changed parts or regenerate the entire thing?

Supporting partial modifications like "change task 3's status to completed" can save tokens. But this approach requires logic to accurately track the current plan state, and bugs from referencing wrong indices or losing synchronization are common. Instead, regenerating the entire plan each time simplifies implementation. The LLM just rewrites the entire plan, saying "the current state is this." For plans with 5-10 tasks, the additional token cost is minimal.

How should we represent each task's progress status?

Distinguishing whether a task hasn't started yet (pending), is in progress (in_progress), or is completed (completed) makes it easy for the LLM to determine "what to do next."

TASK DATA STRUCTURE

The Task class is a simple data structure that stores task content and status. The `__str__` method applies different checkbox formats for each status, allowing the LLM to quickly grasp progress.

Listing 7.1 Task data structure definition

```
from typing import Literal, List
from pydantic import BaseModel

class Task(BaseModel):
    content: str
    status: Literal["pending", "in_progress", "completed"]

    def __str__(self):
        if self.status == "pending":
            return f"[ ] {self.content}"
        elif self.status == "in_progress":
            return f"[>] **{self.content}**"
        elif self.status == "completed":
            return f"[x] ~~{self.content}~~"
        return self.content
```

Pending tasks display as empty checkboxes `[ ]`, in_progress tasks as progress indicators `[>]` with bold text, and completed tasks as completion marks `[x]` with strikethrough. This format is a common pattern in to-do lists, so the LLM understands it easily.

THE CREATE_TASKS TOOL

Now, let's implement the planning tool. We can create it simply using the `@tool` decorator we learned in chapter 4. The tool receives a task list and returns it as a string, and this result is recorded in the context as a ToolResult.

The tool's description is the key information the LLM uses to judge "when to use this tool." The description clearly states:

- **When to use:** Complex questions requiring multiple steps of research or combining information from different sources.
- **When not to use:** Simple questions solvable with a single search, tasks with obvious procedures.

- **How to use:** Mark completed tasks as completed, set the next task to in_progress.

Writing the description in detail prevents the LLM from calling the planning tool in unnecessary situations.

Listing 7.2 Planning tool implementation

```
from typing import List
from scratch_agents.tools import tool

@tool
def create_tasks(tasks: List[Task]) -> str:
    """Create or update a task plan.

    WHEN TO USE:
    - Complex queries requiring multiple steps of research
    - Questions that need to combine information from different sour
ces

    WHEN NOT TO USE:
    - Simple questions answerable with a single search
    - Tasks with obvious, straightforward procedures

    HOW TO USE:
    - Regenerate the entire task list with updated statuses
    - Mark completed tasks as 'completed'
    - Mark the next task to work on as 'in_progress'
    - Keep future tasks as 'pending'
    """
    result = []
    for task in tasks:
        result.append(str(task))
    return "\n".join(result)
```

The tool implementation itself is very simple. It receives a task list and converts it to a string to return. The returned string is recorded in the context as a ToolResult, allowing the LLM to reference the plan in subsequent steps.

Why implement it so simply? LLMs already have powerful language understanding and generation capabilities. Instead of implementing complex logic in code, leveraging the LLM's capabilities is more effective. The tool simply records the plan in the context, while the actual planning and state management are handled by the LLM.

7.2.3 Planning tool usage example

Now, let's add the planning tool to an agent and see how it actually works.

Listing 7.3 Running an agent with the Planning tool

```
from scratch_agents import Agent, LlmClient
from scratch_agents.tools import search_web

agent = Agent(
    model=LlmClient(model="gpt-5-mini"),
    tools=[create_tasks, search_web],
    instructions="You are a helpful assistant that plans before acti
ng on complex queries.",
    max_steps=10,
)

result = await agent.run(
    "If Eliud Kipchoge could maintain his marathon pace indefinitel
y, "
    "How many thousand hours would it take him to reach the Moon?"
)
```

When the agent receives this question, it first calls create_tasks to make a plan. The tool's return value is recorded in the context as a ToolResult.

```
Step 1: create_tasks call
Result:
[ ] Find Kipchoge marathon world record time
[ ] Calculate pace in km/h
[ ] Find Earth-Moon distance at perigee
[ ] Calculate time and convert to thousands of hours
```

Then it updates the status as it progresses through each step. When the first task is completed, it marks it as completed and sets the next task to in_progress.

```
Step 2: search_web("Kipchoge marathon world record") call

Step 3: create_tasks call (state update)
Result:
[x] ~~Find Kipchoge marathon world record time~~
[>] **Calculate pace in km/h**
[ ] Find Earth-Moon distance at perigee
[ ] Calculate time and convert to thousands of hours
```

When the plan is recorded in the context, the LLM references this plan in subsequent steps. It's just like a person looking at a to-do list to decide what to do next. Thanks to the checkbox format, you can see at a glance which tasks are completed, which task is currently in progress, and which tasks remain.

PLANNING GRANULARITY TRADE-OFF

If tasks are too granular, create_tasks calls for status updates become excessive, increasing execution time. Rather than separating every simple action, it's more efficient to group tasks into meaningful units. There's always a tradeoff between the granularity of planning and system response latency.

7.2.4 Extension directions

The planning tool we've implemented so far is intentionally simple. This is sufficient for most cases, but if you need to handle more complex tasks, you can extend in the following directions.

Adding completion criteria: Explicitly define when each task is complete. Instead of "Research Kipchoge's record," including specific completion conditions like "Confirm Kipchoge's marathon record time and distance" allows the LLM to more accurately judge whether a task is complete.

Hierarchical structure (Task/Action): Decompose large tasks (Tasks) into smaller actions (Actions). For example, under a "Market research" Task, you might have "Search competitor A," "Search competitor B," and "Compare results" Actions.

Dependency management (deps): Express order relationships between tasks. "Research Kipchoge's record" and "Research Moon distance" are independent of each other and can proceed simultaneously, but "Calculate time" is only possible after both tasks are completed. Making such dependencies explicit enables parallel execution.

However, these extensions significantly increase complexity. Code is needed to manage dependency graphs, track hierarchical structures, and coordinate parallel execution. It's best to add these incrementally only when simple planning tools don't solve the problem.

The key is questions, not structure. More important than complex dependency graphs is guiding the LLM to ask itself the right questions. Including questions like the following in the tool's description is effective:

- What do I need to know to solve this problem?
- What order would be efficient to proceed in?

- How will I know when each step is complete?

7.3 Reflection: Checking and correcting

If planning is "making a plan before acting," reflection is "stopping in the middle to check." A skilled researcher naturally pauses when search results differ from expectations, when synthesizing multiple sources, or checking for anything missing before the final answer. They don't just search; they immediately search again and promptly answer. They ask themselves along the way, "What have I found so far?" and "Am I on the right track?"

ReAct agents lack this checking habit. At each moment, they only think about "what to do next" and don't ask "is this going well so far?" Even when search results are strange, they move on to the next step; even when information conflicts, they just proceed.

Reflection makes agents ask themselves these questions:

- What have I learned so far?
- How close am I to the original goal?
- Is the current approach effective?
- Is there anything that needs to change?

When the answers to these questions are recorded in the context, the LLM references this information in subsequent decisions. If planning adds "what to do next" to the context, reflection adds "evaluation and direction so far" to the context.

7.3.1 When is reflection necessary?

Think about the moments when people naturally pause during work. These moments are exactly when agents need reflection, too.

When a meaningful step has been completed

People naturally pause when a step ends. "First step done. Let me organize before moving on." Agents can also check progress and prepare for the next step when completing a stage of the plan. This is the most predictable checkpoint.

When an error occurs in a tool

When a search fails, or an API doesn't respond, people think, "Why isn't this working? I should find another way." Without reflection, agents easily repeat the same failures until reaching max_steps. Errors are clear signals, making this a particularly important moment for checking.

When synthesizing multiple pieces of information

When a lot of materials have been gathered, people pause, saying, "Wait, let me organize this first." Checking is especially important when there's contradictory information. When searching for the Moon distance in the Kipchoge problem, you might get different numbers like "average distance 384,400km" and "perigee 356,500km, apogee 406,700km." To judge which value to use, you need to stop and think. "The problem asked about 'closest approach,' so I should use the perigee."

When a review is needed before the final answer

Just before answering, people check "Let me make sure nothing is missing." To prevent agents from hastily declaring "I found the answer," a self-verification step is needed before completion. This is the most important but easily overlooked checkpoint.

When reflection is unnecessary

Not every moment requires checking. Doing reflection after every tool call only increases overhead. During simple, straightforward operations, it becomes an unnecessary delay. If everything is proceeding as expected, continuing is more efficient.

Difference from summarization

Summarization (covered in chapter 6) is different from reflection. Summarization is compression that triggers automatically when context length exceeds a threshold. Its purpose is to reduce the token count. In contrast, reflection is selectively executed by the LLM based on the judgment that "checking is needed." Its purpose is to check direction and make corrections if necessary.

7.3.2 Implementing the reflection tool

Like the planning tool, we implement the reflection tool simply. Instead of creating complex logic in code, we leverage the LLM's judgment capabilities.

Design decisions

The reflection parameter is free-form text. Since different content is needed depending on the situation (error analysis, information synthesis, progress checking, etc.), forcing structure would reduce flexibility.

The need_replan flag is a coordination signal with the planning tool. When it's judged that "the current plan is no longer valid," setting this flag to True encourages modifying the plan in the next step.

THE REFLECTION TOOL

The tool's description specifies four usage situations: PROGRESS REVIEW (when completing a step), ERROR ANALYSIS (when a tool fails), RESULT SYNTHESIS (when synthesizing information), and SELF CHECK (before the final answer). Including specific examples for each situation guides the LLM to call the tool at appropriate times.

Listing 7.4 Reflection tool implementation

```
from scratch_agents.tools import tool

@tool
def reflection(analysis: str, need_replan: bool = False) -> str:
    """Pause and analyze progress before continuing.

    WHEN TO USE:
    1. PROGRESS REVIEW - After completing a meaningful step
       "Kipchoge's record found: 2:01:09. Moving to moon distance re
search."

    2. ERROR ANALYSIS - When a tool fails or returns unexpected resu
lts
       "Wikipedia tool failed. Cause: service unavailable. Alternati
ve: use web search."

    3. RESULT SYNTHESIS - When combining information from multiple s
ources
       "Two different moon distances found. Problem asks for closest
approach, so using perigee: 356,500km."

    4. SELF CHECK - Before providing final answer
       "Have all required data: marathon pace 20.81km/h, moon distan
ce 356,500km. Ready to calculate."

    WHEN NOT TO USE:
    - After every single tool call (excessive overhead)
    - During simple, straightforward operations
    - When everything is proceeding as expected

    Args:
        analysis: Your assessment of current situation and next dire
ction
        need_replan: Set True if the current plan needs modification
    """
    if need_replan:
        return f"Reflection recorded (REPLAN NEEDED): {analysis}"
    return f"Reflection recorded: {analysis}"
```

The tool implementation itself is very simple. It receives the analysis content and returns it as a string. This result is recorded in the context as a ToolResult, which the LLM references in subsequent decisions.

If the planning tool recorded "what to do next" in the context, the reflection tool records "evaluation so far" in the context. Both are generation strategies that add LLM-generated text to the context to influence subsequent decisions.

7.3.3 The real value of reflection: Failure recovery

Reflection's value becomes apparent not when everything is going smoothly, but when problems arise. Let's intentionally create a failure situation to see the reflection's effect.

Creating a failing tool

Create a Wikipedia tool that always fails. In reality, external APIs can fail for various reasons: server overload, network issues, API changes, etc.

Listing 7.5 A Wikipedia tool that always fails

```
@tool
def get_wikipedia_page(title: str) -> str:
    """Get a Wikipedia page by title."""
    raise RuntimeError("Wikipedia service is temporarily unavailabl
e")
```

Running with reflection

Add the reflection and Wikipedia tools to an agent and run them.

Listing 7.6 Running an agent with reflection for error recovery

```
agent = Agent(
    model=LlmClient(model="gpt-5-mini"),
    tools=[get_wikipedia_page, search_web, reflection],
    instructions="Always try Wikipedia first when researching topic
s.",
    max_steps=5,
)

result = await agent.run(
    "What is the distance from Earth to Moon at perigee?"
)
```

With reflection, the agent analyzes the failure and finds alternatives.

7.3.4 Running an agent that uses reflection for research synthesis

Reflection is useful not only for failure recovery but also when synthesizing multiple pieces of information. Here's an example of organizing search results.

Listing 7.7 Task data structure definition

```
agent = Agent(
    model=LlmClient(model="gpt-5-mini"),
    tools=[search_web, reflection],
    instructions="You are a research assistant.",
)

result = await agent.run(
    "Research recent developments in quantum computing."
)
```

After the agent performs multiple searches, it synthesizes the results with reflection. Without reflection, the agent might have kept adding searches or answered without properly organizing the collected information. Through reflection, the judgment "I've gathered enough information" is explicitly recorded in the context, and this guides the next step toward generating an answer.

7.4 Integrating planning and reflection

Planning and reflection don't operate separately, but form a single cycle, as shown in figure 7.2. Let’s see how these two strategies connect and summarize how the limitations of ReAct are resolved.

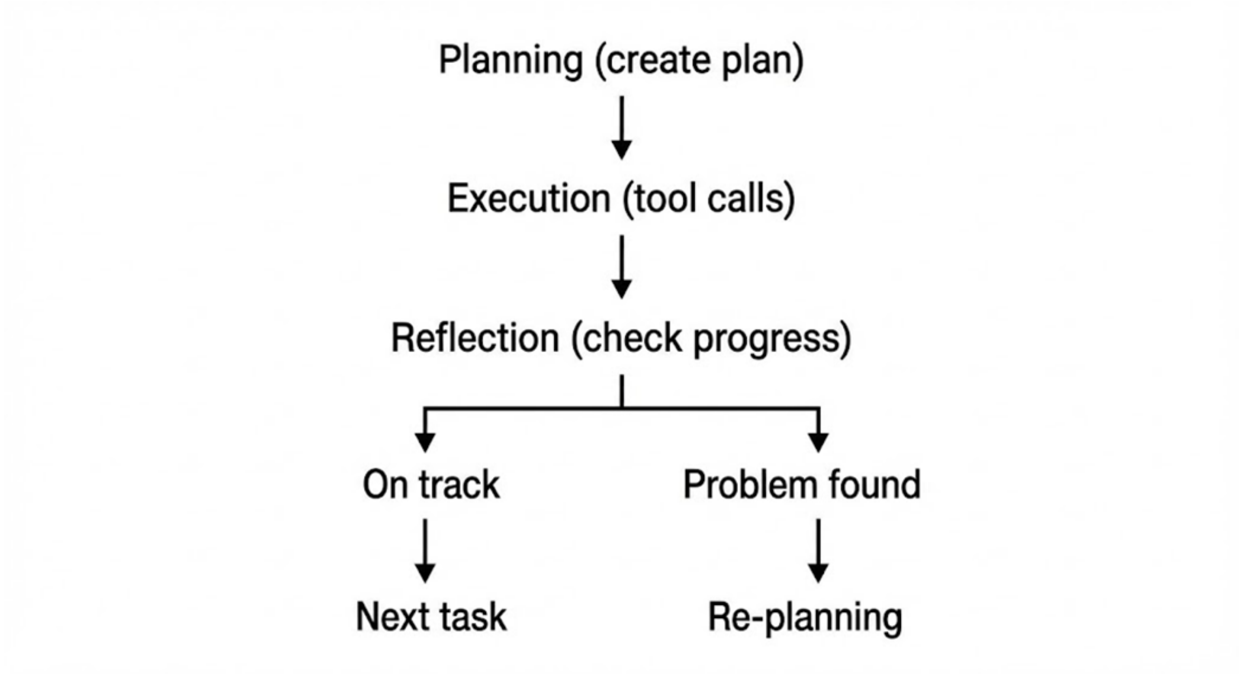


Figure 7.2 The Planning-Reflection cycle in agent execution.

The agent first decomposes tasks and establishes order through planning. Then it executes work by calling tools according to the plan. After execution, it checks progress through reflection. If the check shows things are going well, it moves to the next task; if a problem has occurred, it modifies the plan.

The `need_replan` parameter of the reflection tool implemented serves as this connecting link. When reflection determines that "the current plan is no longer valid," it sets `need_replan=True`. When this signal is recorded in the context, the LLM calls `create_tasks` again in the next step to modify the plan.

For example, consider when the Wikipedia tool fails in the Kipchoge problem. In reflection, you might analyze "Wikipedia service unavailable, need to switch to web search," but set `need_replan=False`. This is because the plan itself (research Moon distance) is still valid; only the method needs to change. On the other hand, if search results revealed "Kipchoge holds the half-marathon record, not the marathon record," you should set `need_replan=True`. Since the premise of the question itself has changed, the entire plan needs to be reconsidered.

In this way, planning and reflection are not independent features but form a complementary cycle. Planning provides direction, reflection checks the direction, and when necessary, it returns to planning.

7.4.1 Failure modes and solutions

The first failure mode was "losing direction by trying to do too much at once." This happens when agents execute multiple searches simultaneously and fail to properly utilize the results. Planning solves this problem. It decomposes tasks into clear units and provides structure to proceed one at a time. When decomposed like "1. Research Kipchoge's record, 2. Research Moon distance, 3. Calculate," the agent only moves to the second task after completing the first.

The second failure mode is "hastily declaring that an answer has been found." The agent attempts to answer with only partial information before gathering all the necessary details. Planning solves this by specifying completion criteria, and the SELF CHECK step of reflection verifies "Have I secured all the information?" The "Before providing final answer" usage timing, as included in the reflection tool's description, serves exactly this role.

The third failure mode is "forgetting what was done in the middle,” when the agent fails to leverage previously found information and repeats the same searches while going through multiple steps. Planning mitigates this by recording the status of each task in the plan to track "how far we've gotten." Reflection explicitly records "what we've learned so far" in the context through PROGRESS REVIEW. When these texts remain in the context, the LLM references this information in subsequent decisions.

The fourth failure mode is "failing to recover from failures." The agent repeats the same attempts until reaching max_steps instead of analyzing causes and finding alternatives when tool calls fail or go in the wrong direction. This is Reflection's core value. ERROR ANALYSIS analyzes failure causes and suggests alternatives. When an analysis like "Wikipedia service unavailable. Cause: server overload. Alternative: use web search" is recorded in the context, the LLM tries alternatives instead of repeating the same failure.

The following table summarizes this content:

Table 7.1 ReAct failure modes and solutions

Failure Mode	Planning Solution	Reflection Solution
Trying too much at once	Task decomposition, proceed one at a time	
Premature completion declaration	Specify completion criteria	Self-verification (SELF CHECK)
Losing direction/forgetting	Record status in plan	Progress check (PROGRESS REVIEW)
Unable to recover from failure		Cause analysis, suggest alternatives (ERROR ANALYSIS)

Planning mainly solves "looking ahead" problems, while Reflection solves "looking back" problems. Some problems are solved through their collaboration. A skilled researcher doesn't search immediately but first decomposes the question (planning), investigates one thing at a time while organizing along the way (reflection), and adjusts their approach when unexpected results appear (re-planning). The integration of planning and reflection implements this natural working style in agents.

In the next chapter, we will explore how to give agents code execution capabilities through a sandbox environment. Code is the expression of procedures for performing tasks in a programming language, which inherently shares similar characteristics with the planning we covered in this chapter. The sandbox becomes a new workspace for the agent, much like how humans work through computers.

7.5 Summary

- Planning and reflection give agents "time to think." Instead of reacting moment by moment, agents plan before acting and check after acting. This grants metacognition: the ability to examine their own process.

- Planning decomposes complex problems into clear, manageable units. When "1. Research Kipchoge's record, 2. Research Moon distance, 3. Calculate time" is recorded in the context, the LLM references this plan to maintain direction across multiple steps.
- Reflection is valuable when problems arise, not when everything goes smoothly. When tools fail or results are unexpected, Reflection enables cause analysis and alternative strategies instead of repeating the same failures.
- Planning and reflection form a complementary cycle. Planning provides direction, reflection checks the direction, and when necessary, it triggers re-planning. Neither works in isolation.
- From a context engineering perspective, both are generation strategies. Planning adds "what to do next" to the context, while reflection adds "evaluation and direction so far." These texts influence the LLM's subsequent decisions.