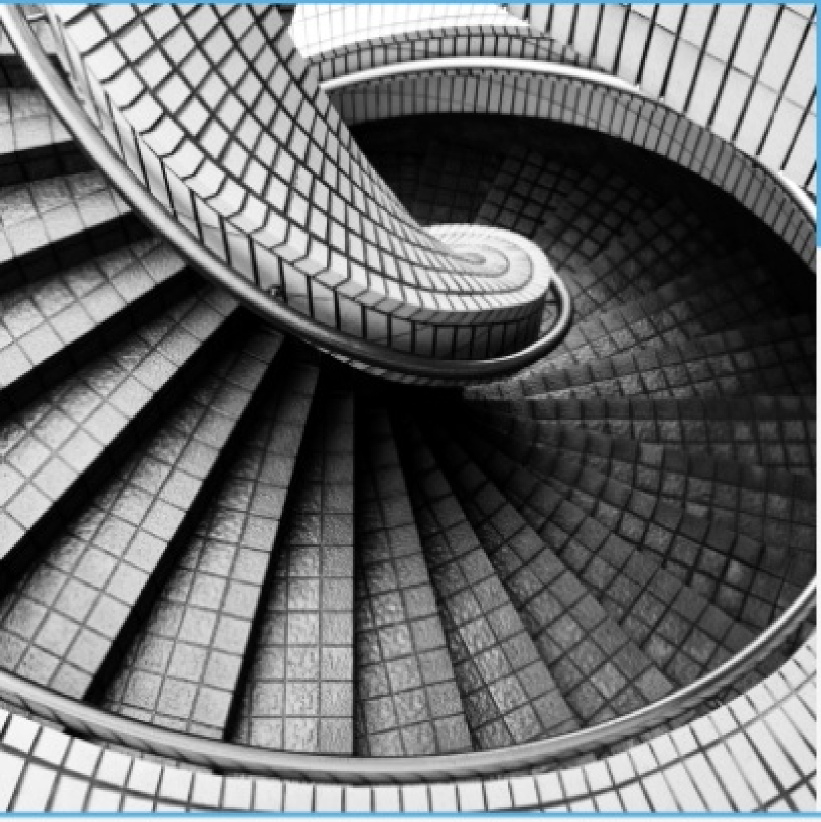
EXPERT INSIGHT

Early Access

AI Agents in Practice

Design, Implement, and Scale Autonomous AI Systems for Production



Valentina Alto

‹packt›

AI Agents in Practice

Early Access Publication: AI Agents in Practice

Early Access Production Reference: B32584

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80580-135-1

Table of Contents

AI Agents in Practice, First Edition: Design, Implement, and Scale Autonomous AI Systems for Production

1 Evolution of GenAI Workflows

Over the past two years, **large language models** (**LLMs**) have reshaped the landscape of AI. From simple prompt-based interactions to complex applications across industries, LLMs have evolved rapidly, fueled by breakthroughs in architecture, training techniques, and fine-tuning strategies. As their capabilities evolved, the shift from ChatGPT to today's agentic systems as of April 2025 marks a natural progression, where the addition of reasoning, planning, and action-taking capabilities represented a major technological leap.This chapter explores the foundations of LLMs, how they're built and consumed, and the differences between pre-trained and fine-tuned models. Most importantly, it sets the stage for the next leap forward: the emergence of AI agents.In this chapter, we will cover the following topics:

- Understanding foundation models and the rise of LLMs
- Latest significant breakthroughs
- Road to AI agents
- The need for an additional layer of intelligence: introducing AI agents

By the end of this chapter, you'll have a clear understanding of how LLMs evolved, how they're trained and deployed, and why the road to truly intelligent systems inevitably leads to the emergence of AI agents.

Technical requirements

You can access the complete code for this chapter in the book’s accompanying GitHub repository at https://github.com/PacktPublishing/AI-Agents-in-Practice.

Understanding foundation models and the rise of LLMs

AI has undergone a fundamental transformation thanks to the emergence of foundation models—versatile, general-purpose models that can be adapted across a wide range of tasks. Among them, LLMs have taken center stage, redefining how we interact with machines through natural language.

From narrow AI to foundation models

Before the rise of foundation models, the field of AI was dominated by *narrow AI*—systems built to perform one specific task and nothing else. Each use case required a custom pipeline: a unique dataset, a dedicated model architecture, and a specialized training routine. If you wanted to classify emails as spam or not spam, you'd build a spam filter. If you needed to extract names and places from documents, you'd create a named entity recognizer. Want to summarize a news article? That would mean yet another bespoke model.This fragmented approach had several drawbacks. Models were brittle—performing well only within the narrow domain they were trained for—and expensive to maintain. Any shift in the task or data distribution often meant retraining from scratch.The introduction of *foundation models* marked a fundamental shift in how we build and think about AI systems. These models are trained on vast and diverse datasets that span multiple domains and tasks. The idea is to teach a single model a general understanding of the world—its language, structure, and patterns—during a large-scale *pretraining* phase. Once this general knowledge is embedded, the model can be *adapted* to specific tasks with minimal additional data and compute.For example, instead of building a separate model for translating French to English, we can now take a pre-trained foundation model and fine-tune it on a smaller translation dataset. The pre-trained model already understands language syntax, grammar, and meaning. Fine-tuning simply aligns this understanding to a specific objective.The key innovation behind foundation models is *transfer learning*. Rather than learning from scratch, these models transfer knowledge gained from general training to specific problems. This dramatically improves efficiency, reduces the amount of labeled data required, and leads to more robust and flexible AI systems.Moreover, foundation models aren’t just about language. They apply across modalities: some models can process and generate not only text but also images, audio, or code.In essence, foundation models act as a "base brain" for AI—trained once, then repurposed many times over. This scalability and adaptability have unlocked entirely new possibilities in how we build intelligent systems, setting the stage for more autonomous and interactive applications such as AI agents.Now, we mentioned that foundation models are capable of managing a variety of data formats. Within the cluster of foundation models, we can find data-specific models that focus on only one data type, which is the case of LLMs.

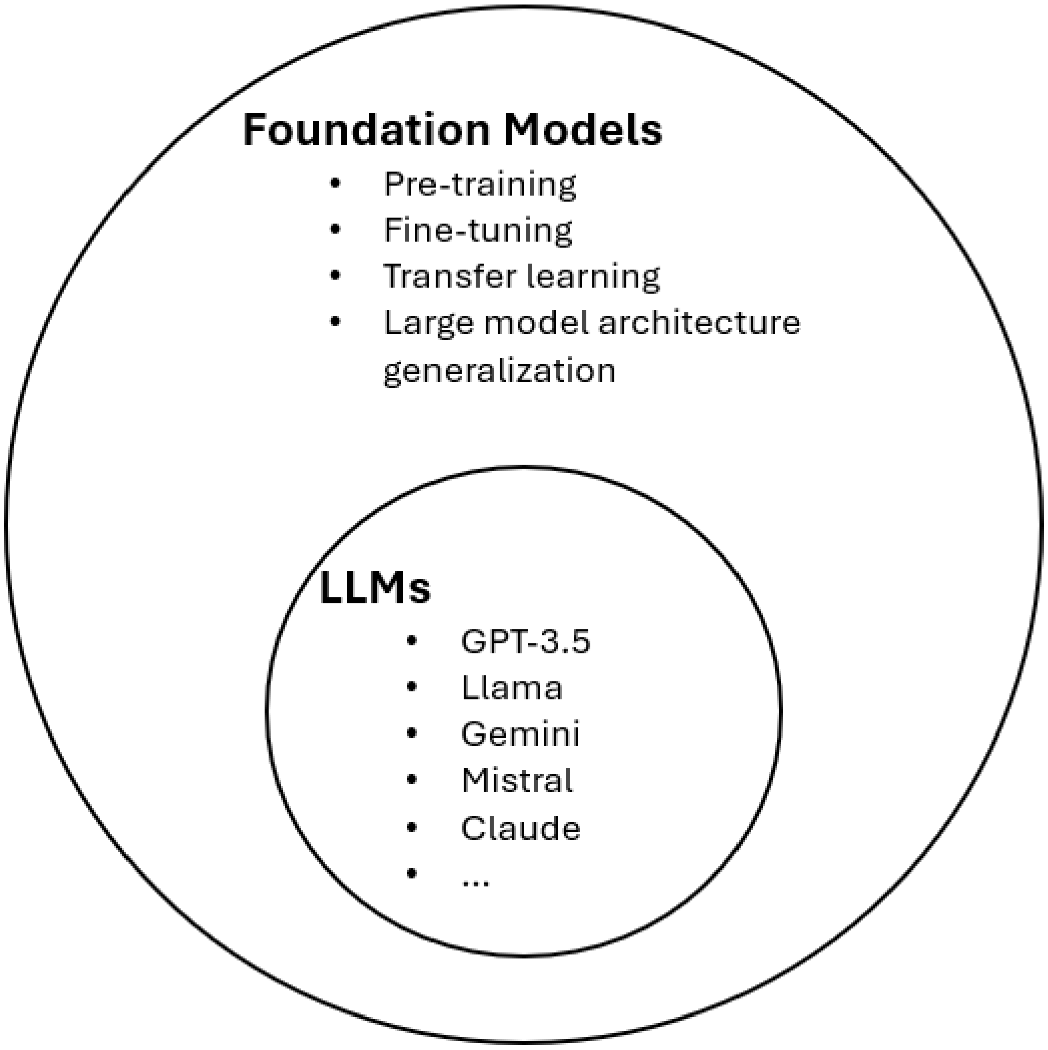


Figure 1.1: Features of LLMs

LLMs are, in essence, the language-specialized version of foundation models. They’re built on deep neural network architectures—particularly transformers—and are trained to predict the next word in a sequence. But this seemingly simple goal unlocks surprisingly emergent behaviors. LLMs can carry on conversations, answer intricate questions, write code, and even simulate reasoning.

Definition

> Emergent behaviors are complex capabilities that arise unexpectedly when a system reaches a certain scale, even though those capabilities weren’t explicitly programmed or anticipated. In the context of LLMs, these behaviors surface when models are scaled up in terms of data, parameters, and training time—unlocking new abilities that were absent in smaller versions.

As models scale, they begin to exhibit emergent properties, including the following:

- **In-context learning**: LLMs can learn to perform a task simply by being shown a few examples in the prompt—without any fine-tuning. This was not seen in smaller models.
- **Chain-of-thought reasoning**: By generating intermediate reasoning steps, LLMs can solve multi-step problems such as math word problems or logical puzzles—something they previously struggled with.
- **Analogical reasoning**: They can solve analogy problems (e.g., “cat is to kitten as dog is to...”) in a way that resembles human cognitive processing.
- **Arithmetic and logic**: At scale, LLMs develop the ability to handle tasks such as multi-digit arithmetic or logic puzzles, even if these tasks weren’t part of their training objective.
- **Understanding metaphors and humor**: Advanced LLMs can interpret new metaphors or even attempt jokes—demonstrating an abstract grasp of language and nuance.
- **Multi-task generalization**: Rather than being trained for one specific task, they can simultaneously handle translation, summarization, question answering, and more—without task-specific training.

These capabilities represent more than just better performance—they are qualitatively new behaviors that "emerge" only at scale, giving LLMs a surprisingly broad range of skills with real-world implications across domains.

Under the hood of an LLM

At the core of every LLM is a powerful neural network architecture—most commonly, a **transformer**. These networks are built to process and understand patterns in data, particularly human language, by learning statistical relationships across billions of text examples. Though loosely inspired by the structure of the human brain, LLMs function purely through mathematics, passing information through interconnected layers that adapt as the model trains.To make language computable, the first step is to convert it into numbers because neural networks can't process raw text. This happens through two key steps—**tokenization** and **embedding**:

- **Tokenization** breaks down sentences into smaller chunks called **tokens**. These could be full words or parts of words, depending on the model. For example, "The cat sat on the mat" might be split into individual words or smaller subword units, depending on the tokenizer used.
- **Embedding** takes these tokens and maps each one to a high-dimensional vector—a string of numbers that encodes its meaning and relationship to other words. These embeddings are learned during training so that similar words end up in similar regions of the model's "semantic space." This helps the model understand context and word usage, such as how "Paris" and "London" relate as cities.

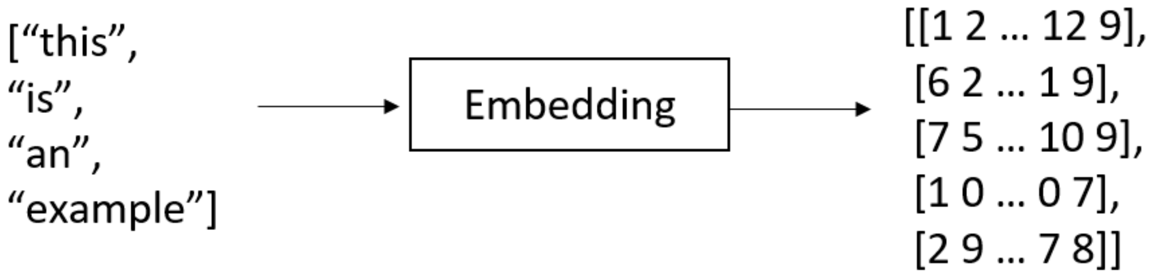


Figure 1.2: An example of embedding

Once the input is tokenized and embedded, it moves through the **transformer network** itself. Unlike traditional neural networks with just a few hidden layers, LLMs use dozens—or even hundreds—of stacked layers, each containing mechanisms called **attention heads**. These attention layers help the model decide which parts of the input are most relevant to a given prediction.

For instance, when completing a sentence, the model learns to focus more on specific previous words that influence what should come next.Training an LLM means teaching it to make better predictions over time. This is done through a method called **backpropagation**, where the model compares its predicted word to the correct one, calculates how far off it was, and then updates its internal parameters to reduce future errors.

Definition

Backpropagation is the core learning algorithm used to train neural networks. It works by comparing the model's prediction to the correct answer, calculating the error (called the loss), and then adjusting the network's internal parameters (weights) to reduce that error. This adjustment happens by "propagating" the error backward through the layers of the network—hence the name. Over time, this process helps the model make increasingly accurate predictions.

Let's say you type `The cat is on the...`. The model predicts the next word by assigning probabilities to possible continuations such as `mat`, `roof`, or `sofa`. It doesn't guess randomly—it relies on patterns it has seen during training.This process is repeated across vast amounts of data—millions or billions of sentences—enabling the model to gradually capture the structure and rhythm of language. The result is a system capable of not just completing sentences but also holding conversations, solving problems, and responding with context-aware, often remarkably fluent language.

How do we consume LLMs?

Once the training stage of an LLM is concluded, we need to understand how to predict the next token with this model, and that process is called inference.

Definition

In the context of machine learning and AI, **inference** refers to the process of running a trained model on new input data to generate predictions or responses. In LLMs, inference involves processing a prompt and producing text-based output, typically requiring significant computational resources, especially for large models.

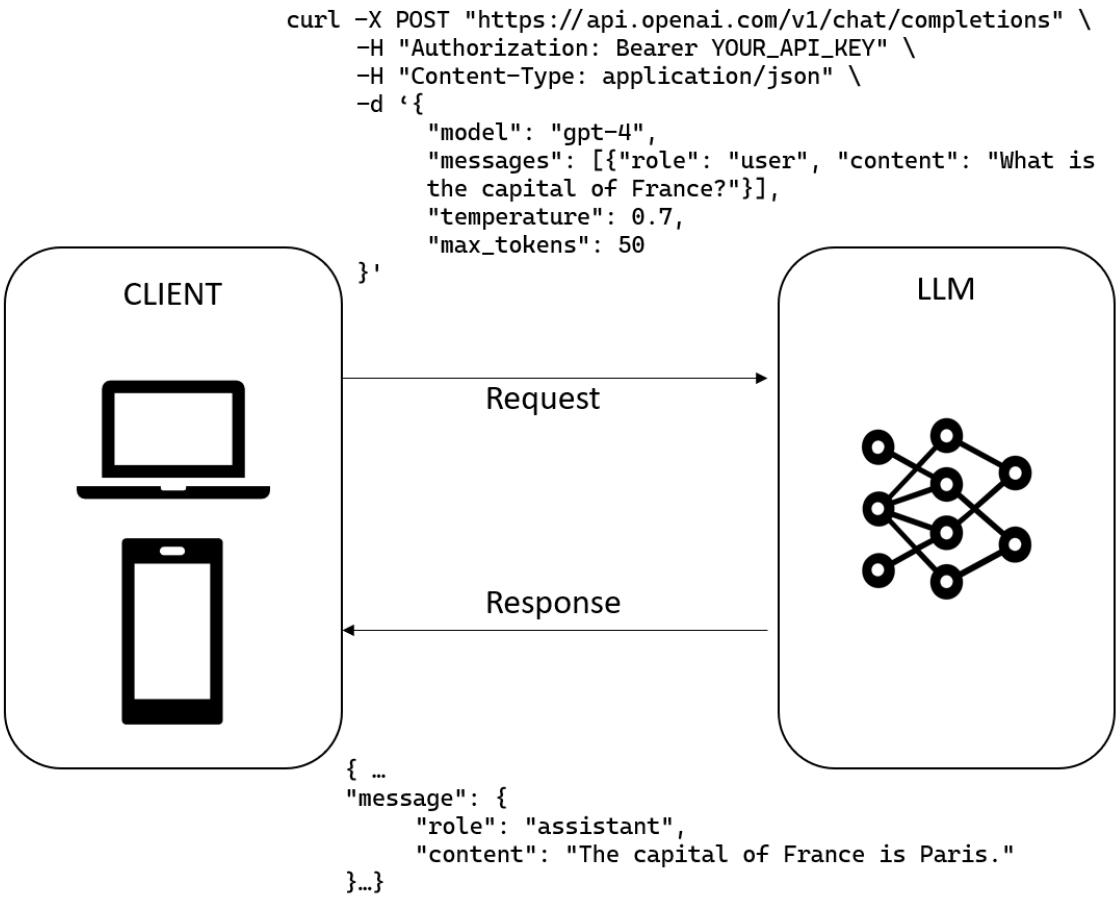
LLMs can be typically accessed through APIs, allowing developers to use them without managing complex infrastructure. This approach simplifies integration, making AI-powered applications more scalable and cost-effective.LLM providers such as **OpenAI**, **Azure AI**, and **Hugging Face** offer APIs that handle requests and return responses in real time. The process usually involves the following:

1. **Authentication**: Developers use API keys or OAuth tokens for secure access.

   Definition

   > **Authentication** is how developers prove that their application has permission to access an external service. This is usually done using either API keys or OAuth tokens. An **API key** is a unique string provided by the service—such as a password—that identifies the app. **OAuth**, on the other hand, is a more flexible system that allows users to grant specific permissions to apps, issuing temporary access tokens in return. Both methods ensure that only authorized users or systems can make requests, helping protect sensitive data and resources.

2. **Sending a request**: A structured JSON request includes the model name, prompt, and parameters such as temperature (for randomness).
3. **Receiving a response**: The API returns a generated text output along with metadata such as token usage.



```
curl -X POST "https://api.openai.com/v1/chat/completions" \
     -H "Authorization: Bearer YOUR_API_KEY" \
     -H "Content-Type: application/json" \
     -d '{
          "model": "gpt-4",
          "messages": [{"role": "user", "content": "What is
          the capital of France?"}],
          "temperature": 0.7,
          "max_tokens": 50
     }'
```

```
{ …
"message": {
     "role": "assistant",
     "content": "The capital of France is Paris."
}…}
```

Figure 1.3: Example of an HTTP request to the LLM API

Note

Some LLM APIs support streaming responses, where the model outputs tokens incrementally rather than waiting to generate the full response before sending it. This approach helps mitigate the high latency often associated with large models. By delivering the first chunk of text quickly, streaming reduces perceived latency—the time the user waits before seeing any output—leading to a smoother, more responsive experience.

Now, a legitimate question might be: what if I want to run my model on my local computer? To answer this question, we first need to distinguish between the following:

- **Private LLMs**: These are proprietary models developed by companies such as OpenAI, Anthropic, or Google. They're closed source, meaning you can't see or modify their underlying code. Those models are typically only accessible via APIs, and they come with a pay-per-use, token-based cost.
- **Open source LLMs:** Open source models, such as Meta's LlaMA, Mistral, and Falcon, are freely available for anyone to download, modify, and deploy. This means developers can access the underlying trained parameters, run them on private infrastructure, and even use the underlying architecture to retrain the model from scratch.

Even for open source LLMs, however, many developers opt to access these models via APIs provided by platforms such as Azure AI Foundry and Hugging Face Hub.This approach offers several advantages:

- **Reduced infrastructure costs**: Running LLMs independently demands significant computational resources, which can be cost-prohibitive. Utilizing APIs shifts this burden to the service provider, allowing developers to leverage powerful models without investing in expensive hardware.
- **Scalability**: API services can dynamically scale to handle varying workloads, ensuring consistent performance without manual intervention.
- **Security and compliance**: Platforms such as Azure AI Foundry offer enterprise-grade security features, helping organizations meet compliance requirements and protect sensitive data.

In the context of AI agents and, more broadly, AI-powered apps, the most adopted path is that of consuming LLMs via APIs. Exceptions might be related to disconnected scenarios (e.g., running LLMs at offshore sites or remote locations) or regulatory constraints in terms of data residency (the LLM must reside in a specific country where there is no public cloud available).

Latest significant breakthroughs

The field of GenAI has experienced rapid advancements over the past few years, with breakthroughs that pushed the boundaries of efficiency, adaptability, and reasoning capabilities. In the following sections, we are going to explore some of the latest techniques that significantly enhance GenAI models' performance while reducing computational demands.

Small language models and fine-tuning

Small language models (**SLMs**) are becoming increasingly relevant as organizations seek efficient, cost-effective alternatives to large-scale AI systems.

Definition

> **SLMs** are a streamlined category of GenAI models designed to efficiently process and generate natural language while using fewer computational resources than their larger counterparts. Unlike LLMs, which can have hundreds of billions of parameters, SLMs typically contain only a few million to a few billion parameters.

The reduced size of SLMs allows them to be deployed in environments with limited hardware capabilities, such as mobile devices, edge computing systems, and offline applications. By focusing on domain-specific tasks, SLMs can deliver performance comparable to LLMs within their specialized areas while being more cost-effective and energy-efficient.SLMs can be designed as domain-specific models since their pre-training stage, or they can be adjusted and tailored after their first training (which will still be general-purpose, as LLMs). The process of further specializing a model on a specific domain is called **fine-tuning**.The fine-tuning process involves using smaller, task-specific datasets to customize the foundation models for particular applications.This approach differs from the first one because, with fine-tuning, the parameters of the pre-trained model are altered and optimized toward the specific task. This is done by training the model on a smaller labeled dataset that is specific to the new task. The key idea behind fine-tuning is to leverage the knowledge learned from the pre-trained model and fine-tune it to the new task, rather than training a model from scratch.

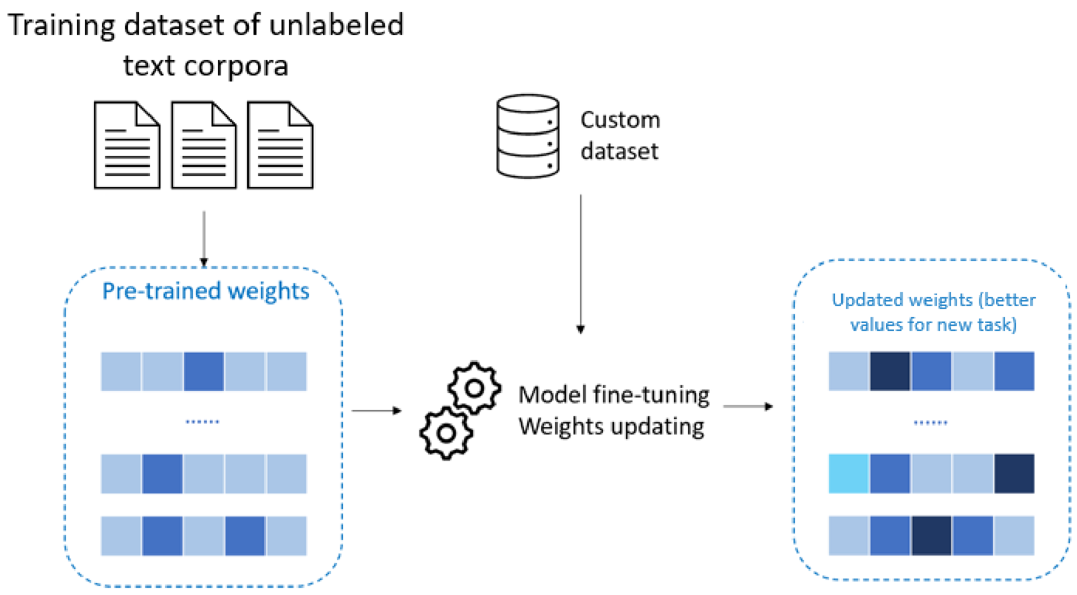

Training dataset of unlabeled text corpora
Custom dataset
Pre-trained weights
Model fine-tuning Weights updating
Updated weights (better values for new task)

Figure 1.4: Illustration of the process of fine-tuning

In the preceding figure, you can see a schema on how fine-tuning works on OpenAI pre-built models. The idea is that you have a pre-trained model available with general-purpose weights or parameters. Then, you feed your model with custom data, typically in the form of “key-value” prompts and completions. In practice, you are providing your model with a set of examples of how it should answer (completions) to specific questions (prompts).Here, you can see an example of how these key-value pairs might look:

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
...
```

Once the training is done, you will have a customized model that is particularly suited for a given task, for example, the classification of your company’s documentation.The major benefit of fine-tuning is that you can make pre-built models tailored to your use cases, without the need to retrain them from scratch, yet leveraging smaller training datasets and hence less training time and compute. At the same time, the model keeps its generative power and accuracy learned via the original training, the one that occurred on the massive dataset.Fine-tuning is particularly valuable for SLMs, as it enables them to achieve high performance while maintaining their efficiency.Several advanced fine-tuning techniques have been developed to optimize the process, especially for SLMs:

- **Low-rank adaptation** (**LoRA**): This method inserts trainable low-rank matrices into the model's layers, allowing adaptation to new tasks with minimal computational overhead. LoRA is highly efficient in terms of memory usage and is widely used to fine-tune large models on limited hardware.
- **Adapter tuning**: Instead of modifying the entire model, small neural network modules called **adapters** are added to each layer. During fine-tuning, only these adapters are updated, significantly reducing the number of trainable parameters while preserving the model's pre-trained knowledge.
- **Prefix tuning and prompt tuning**: These techniques guide the model's output by attaching learnable task-specific vectors or tokens to the input. Prefix tuning introduces trainable vectors at the beginning of the input sequence, whereas prompt tuning optimizes a set of prompt tokens to steer the model's behavior. Both approaches allow for efficient adaptation without altering the model's internal parameters.

By leveraging SLMs in combination with efficient fine-tuning methods, AI applications can achieve high levels of performance without the computational and financial burdens of massive models. This makes AI more accessible, sustainable, and scalable for a wide range of industries and use cases.

Model distillation

Model distillation, also known as **knowledge distillation** (**KD**), is a process where *heavy* LLMs (by *heavy*, we refer to their high number of parameters) transfer their knowledge to lighter LLMs or SLMs without a significant loss in performance.This is a crucial technique if we consider that the most powerful LLMs are typically made of billions—if not trillions—of parameters, making them computationally expensive for both training and inference. In fact, among the main benefits of distillation, we can mention the following:

- Reduced model size while preserving accuracy
- Improved inference speed and reduced latency
- Lower computational and energy costs

- Enables deployment on edge devices and mobile platforms

Distillation typically follows a structured training pipeline:

1. **Teacher model training**: A large, powerful LLM is pre-trained on vast datasets and fine-tuned for specific tasks.
2. **Soft label extraction**: When the LLM returns its output (known as hard labels), we know that each token is the result of a probabilistic computation—the one with the highest probability associated. However, for each prediction, we have an associated vector of probability which, in fact, provides a nuanced view of the teacher's predictions and thought process. These probabilities, referred to as soft labels, are extracted as they will be extremely useful to train the student model.
3. **Student model training**: A smaller model is trained using soft labels along with the ground truth, which serve as a rich source of information and nuanced predictions.
4. **Optimization and fine-tuning**: The student model undergoes additional refinements to further enhance its accuracy and efficiency.

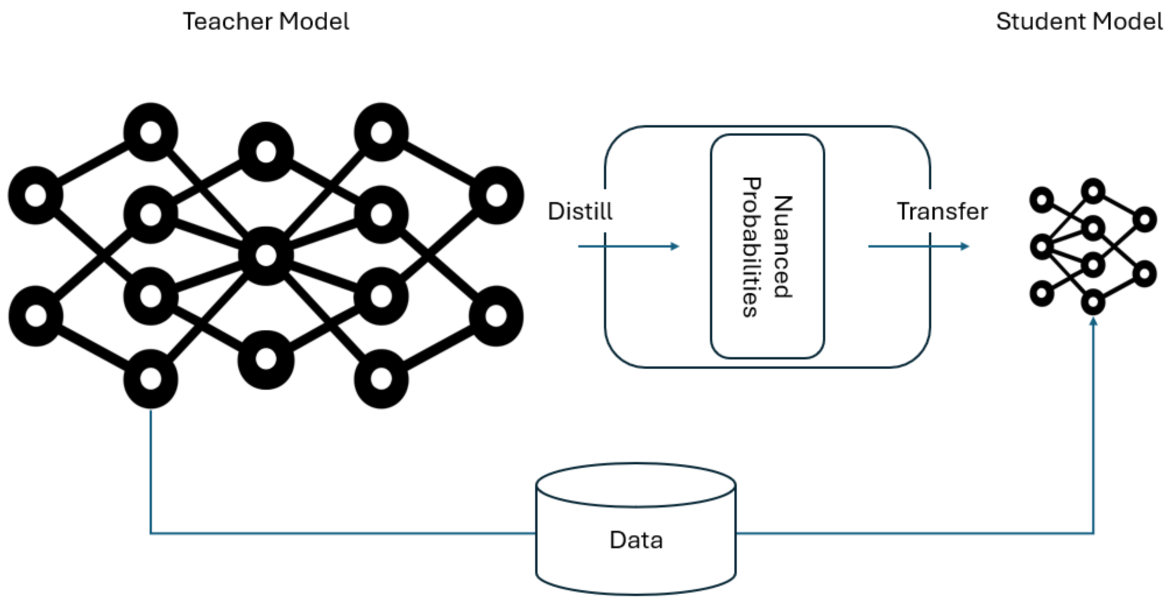


Figure 1.5: Generic framework for model distillation (source: https://arxiv.org/pdf/2006.05525)

As LLMs continue to grow in size and computational demands, distillation techniques enable more practical deployment while retaining high-quality outputs.

Reasoning models

In late 2024, a new class of AI models known as **reasoning language models** (**RLMs**) emerged, designed to enhance complex problem-solving capabilities beyond traditional LLMs. These models represent a significant shift in GenAI development, focusing on internal deliberation and step-by-step reasoning to tackle intricate tasks.Examples of RLMs are the following:

- **OpenAI's o1 model**: Released in September 2024, the o1 model introduced a "private chain-of-thought" mechanism, allowing the model to internally process and reason through problems before responding. This approach led to substantial improvements in fields such as mathematics and science, with o1 solving 83% of problems on the American Invitational Mathematics Examination (https://en.wikipedia.org/wiki/OpenAI_o1?utm_source=chatgpt.com), marking a notable increase in performance compared to previous models.

- **OpenAI's o3 model**: Building upon the advancements of o1, the o3 model, unveiled in December 2024, further enhanced reasoning capabilities by allocating more time for internal deliberation. This resulted in higher accuracy on complex tasks, including coding and advanced scientific queries. Notably, o3 achieved a 75.7% score on the ARC-AGI benchmark (https://arcprize.org/blog/oai-o3-pub-breakthrough), reflecting its superior problem-solving skills.

Definition

Abstraction and Reasoning Corpus for Artificial General Intelligence (**ARC-AGI**) is a benchmark designed to evaluate an AI system's ability to generalize and adapt to novel tasks, closely mirroring human-like intelligence. Introduced by François Chollet in 2019, ARC-AGI emphasizes problem-solving skills that require abstract reasoning without relying on extensive prior data or domain-specific training.

- **DeepSeek's R1 model**: In January 2025, Chinese start-up DeepSeek introduced R1, an open source reasoning model that matched the performance of leading models such as o1 while being developed at a fraction of the cost. The open source nature of R1 has facilitated widespread research and adaptation, contributing to its rapid adoption and impact.

The great differentiator of reasoning models is that they “take their time” before answering; unlike traditional LLMs, which generate responses in a single pass, RLMs engage in internal deliberation, processing multiple reasoning steps before arriving at a conclusion. This method enhances their ability to handle complex, multi-step problems, which is key the moment we start talking about AI agents, as we will see throughout the book.Also, RLMs are specifically trained to excel in tasks requiring advanced reasoning, such as complex mathematics, scientific research, and intricate coding challenges. This specialization enables them to outperform traditional LLMs in these domains.The natural consequence of the preceding feature is that the internal processing and extended reasoning pathways of RLMs necessitate more computational power and time per query compared to traditional LLMs. This trade-off results in superior performance on tasks demanding deep reasoning at the cost of increased resource consumption.

DeepSeek

In January 2025, everyone’s attention turned toward a new, breakthrough model called DeepSeek R1.DeepSeek, a Chinese AI company founded in 2023, has developed a series of advanced LLMs—culminating in its R1 series—that have disrupted the AI industry by demonstrating that high-performance models can be developed efficiently and cost-effectively. As the icing on the cake, DeepSeek open-sourced this training approach as well as the models themselves, so that everyone can download them and use them locally.These are the main features that make DeepSeek LLMs a leap forward in the field of GenAI:

- **Training approach**: What sets DeepSeek apart is its unique approach to training. Instead of relying heavily on manually labeled datasets, DeepSeek's R1-Zero model was trained using reinforcement learning (RL) alone. In RL, models learn by trial and error—receiving rewards for producing desirable outputs. In this case, the LLM was rewarded for generating coherent and accurate responses, encouraging it to develop reasoning capabilities independently.

- This pure-RL approach allowed DeepSeek to push boundaries, but it initially came with trade-offs: the model sometimes produced less readable or inconsistent language. To address this, the team adopted a multi-stage training process for the follow-up R1 model, combining different techniques to gradually improve the model's performance.
- The process began with supervised fine-tuning on a small, high-quality dataset—often referred to as cold start data—to establish a strong linguistic foundation. Then, RL was reintroduced to sharpen reasoning and decision-making abilities. The model also generated synthetic data, which was filtered through rejection sampling to remove poor outputs. This filtered data was used for further supervised training. A final RL phase helped improve the model's consistency and adaptability across diverse tasks.
- The result? A model that rivals top-tier alternatives such as OpenAI's o1 in quality—despite being trained with fewer resources and no large-scale human-annotated datasets.
- **Hardware utilization**: In an industry where access to cutting-edge hardware is often a limiting factor, DeepSeek has demonstrated that innovation can offset hardware constraints. The company successfully trained its flagship model, DeepSeek-R1, using approximately 2,000 NVIDIA H800 GPUs over a span of 55 days, incurring a cost of around $5.6 million, thanks to the training strategy explained previously. This achievement is particularly noteworthy given the U.S. export restrictions on advanced AI chips to China, underscoring DeepSeek's ability to optimize available resources effectively.
- **Open source**: DeepSeek's commitment to open source principles has fostered a collaborative environment that accelerates innovation. By making its models and training methodologies publicly available, DeepSeek invites researchers and developers worldwide to contribute to and build upon its work.

DeepSeek's advancements have sent ripples across the global AI community, challenging established players and prompting a reevaluation of existing practices.

Road to AI agents

The rapid evolution of GenAI has taken us from simple automation to increasingly intelligent systems capable of reasoning, learning, and decision-making. In recent years, LLMs have revolutionized how we interact with the surrounding ecosystem, enabling more natural conversations and sophisticated problem-solving.Let's explore the greatest milestones that have eventually fueled the rise of AI agents.

Text generation

Since the launch of ChatGPT in November 2022, the very first use case that users embraced was that of conversational text generation, such as the following:

- *"Generate a beginner-level description of the nuclear fission"*
- "Draft an email to send to the C-level board of my customer to invite them to our event"
- "List 10 ideas for an article about AI"
- "Generate an essay about the French Revolution I have to hand over by tomorrow"

Do any of the preceding scenarios look familiar to you?The text generation capability of LLMs was disruptive because it fundamentally changed how humans interact with technology, enabling AI to produce human-like text with unprecedented fluency and contextual understanding.

Note

> In this context, *text* also includes code, as LLMs have demonstrated significant capabilities in generating and assisting with programming tasks since the very beginning. Back then, common code-related tasks included code generation, debugging, optimization, translation, and explanation.

LLMs marked an unprecedented shift in the field of traditional AI and natural language processing, as they could generate coherent, creative, and contextually relevant text on demand. All of a sudden, such an incredible technology was available to every user with an internet connection, democratizing access to high-quality writing, accelerating automation in industries such as marketing and customer service, and even reshaping creative fields by assisting in storytelling, poetry, and scriptwriting.However, after the initial hype of having this free, conversational assistant that seemed to know anything about the world we live in, we soon started realizing that there was a *huge* limitation. ChatGPT and, more generally, LLMs carry knowledge that is limited to the data they have been trained on (this is *parametric knowledge*). Now, even if this data represents the whole web, users still need to deal with **dynamic**, **proprietary**, or **niche datasets** that were not part of the model's training data.Chatting with your data is, in fact, the next milestone in our GenAI roadmap.

Chat with your data

"I want to chat with my data." This statement leads to a specific technique called **retrieval-augmented generation** (**RAG**). This method allows the LLM to not only generate text but also retrieve relevant information from external sources before generating a response, ensuring accuracy, context relevance, and a lower risk of hallucination.

Definition

> In the context of language models, **hallucination** refers to the generation of information that is plausible-sounding but false or unsupported by factual data. This can undermine trust, especially in scenarios that demand accuracy.

The process of "scoping" the LLM to a predefined knowledge base is called **grounding**. A critical part of RAG is the **vector database** (**vector DB**), which efficiently stores and retrieves information using a vector representation called **embeddings**. This allows the LLM to search for semantically relevant information rather than just exact keyword matches.Let's break down each step of this technique:

1. **Retrieval or finding relevant information**: Instead of relying only on pre-trained knowledge, RAG first retrieves relevant data from an external knowledge base that has been properly vectorized. This could be PDFs, Word files, reports, research papers, structured records, tables, internal archives, and so on.

When you ask a question, the RAG pipeline converts it into a vector itself and searches for the most relevant pieces of information in the dataset by computing the distance function between the user’s vectorized query and the knowledge base’s vectorized chunks. Because of the way embeddings are computed, the lower the mathematical distance, the higher the semantic similarity. This step ensures that AI is not just generating responses from memory but actively retrieving the most relevant and up-to-date information from your data.

1. **Augmentation—enhancing AI understanding**: Once the system retrieves relevant documents, they are fed into the model along with the original query. This step augments the AI’s understanding by providing contextually rich input.

Instead of guessing or relying on general knowledge, the AI now has grounded, context-specific information to base its response on. This augmentation process ensures that the model’s output is the following:

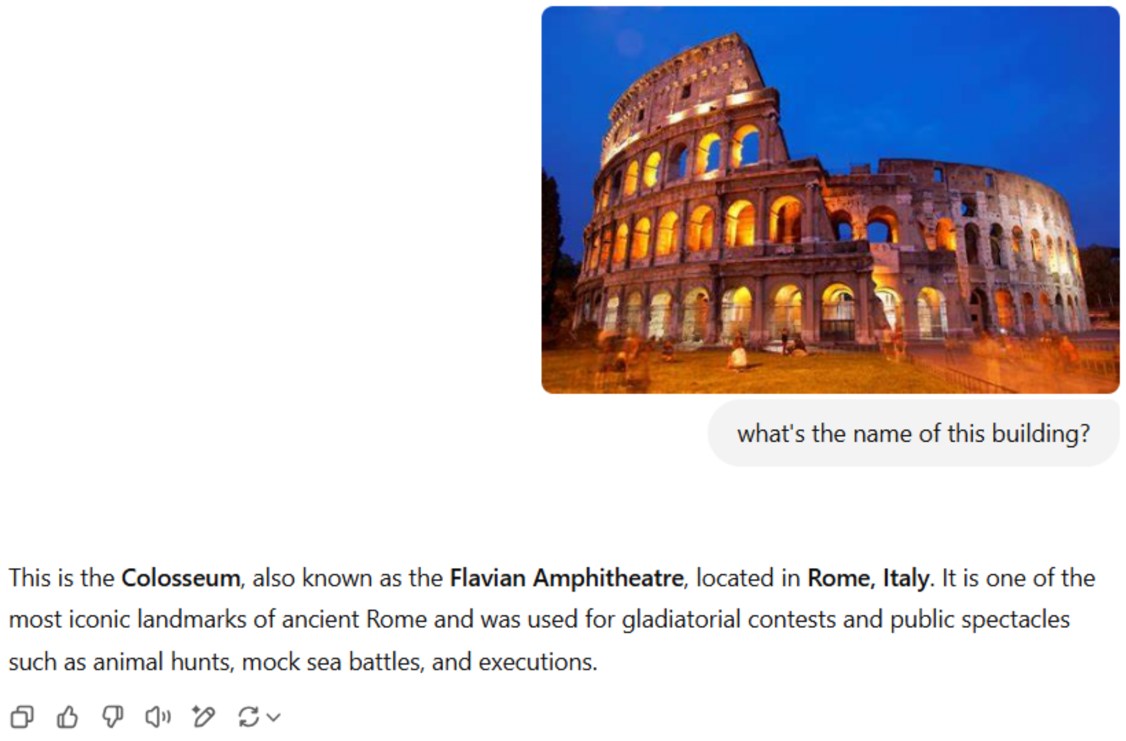
- More precise because it directly references relevant data
- More explainable since responses are backed by retrievable sources
- Less prone to hallucinations as AI generates text in a curated and grounded context

1. **Generation—creating a context-aware response**: With the augmented context, the AI then generates a response that is more informative, accurate, and aligned with the retrieved data. The final output is the following:
    - Factually grounded, as it incorporates retrieved knowledge
    - Context-aware, tailored to your specific dataset
    - Referenceable, meaning it can provide citations or source links when needed

This step ensures that the response is not just an AI-generated answer but one that is specifically crafted based on the **retrieved** and **augmented** knowledge from your data.In the next chapters, we will cover more about RAG and its role in agentic systems.Until now, we’ve only been talking about text data, but what if we want to interact with our models with images, video, or audio?

Multimodality

Multimodality in GenAI refers to the ability of models to process and generate content across multiple types of data, such as text, images, audio, and video. **Multimodal LLMs** (**MLLMs**) extend the capabilities of traditional LLMs by incorporating multiple modalities, allowing for more comprehensive understanding and richer interactions.Recent advances, such as OpenAI’s GPT-4V and Google’s Gemini, demonstrate how MLLMs can analyze images, generate captions, process voice inputs, and even reason across different formats in a single workflow.The key feature of LMMs is that they share the capability of generalization and adaptation with text-only LLMs. However, LMMs are capable of processing heterogeneous data with the idea of mirroring the way humans interact with the surrounding ecosystem—that is, with all our senses.A great example of a multimodal model is OpenAI’s GPT-4o, which is able to interact with users via text, images, and audio. Let’s see a couple of examples with images:

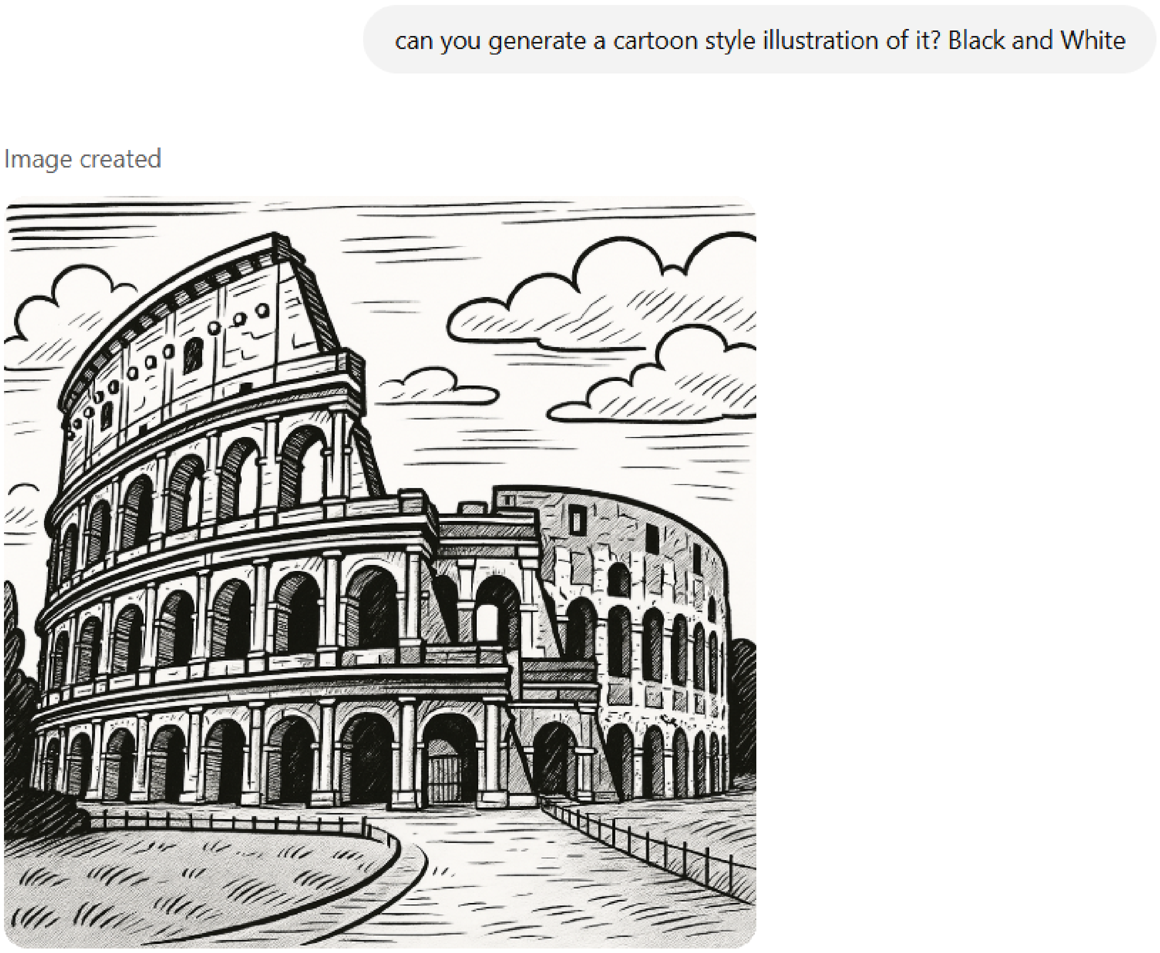


what's the name of this building?

This is the **Colosseum**, also known as the **Flavian Amphitheatre**, located in **Rome, Italy**. It is one of the most iconic landmarks of ancient Rome and was used for gladiatorial contests and public spectacles such as animal hunts, mock sea battles, and executions.

Figure 1.6: Example of ChatGPT reasoning over an image

As you can see, the model was able to analyze the image and reason over it. Let’s now go ahead and ask the model to generate an illustration:

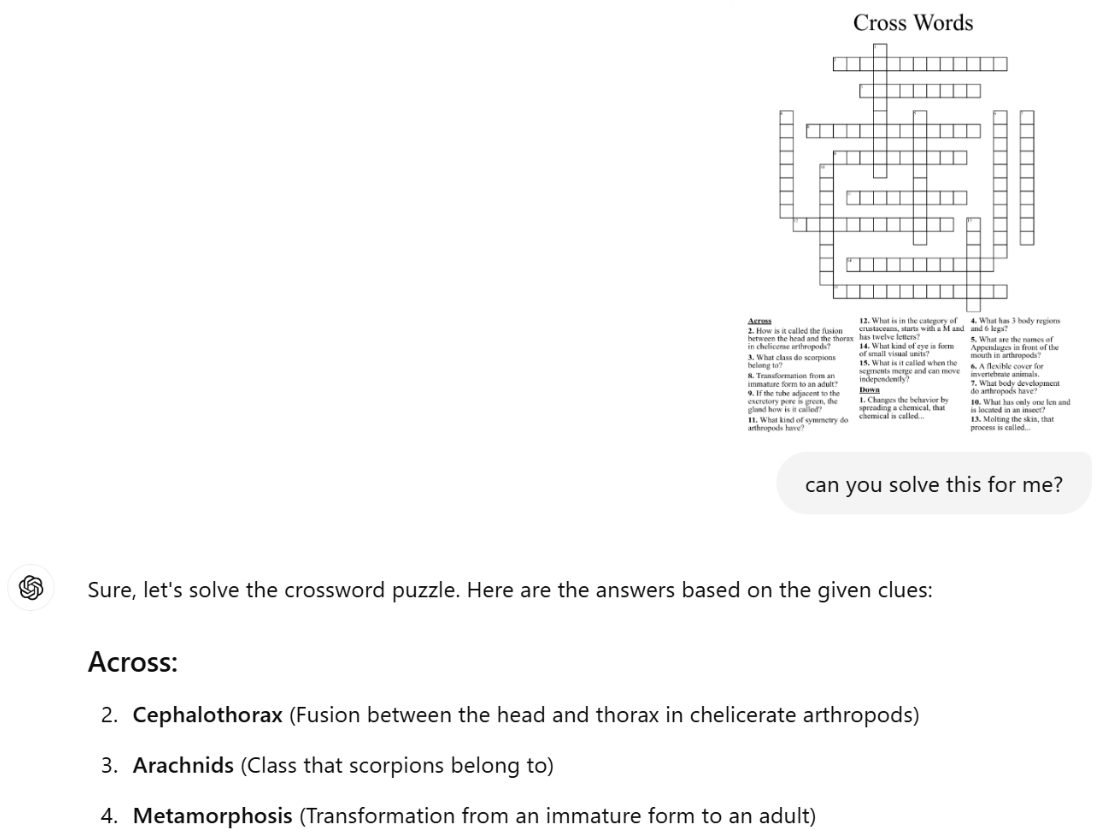


can you generate a cartoon style illustration of it? Black and White

Image created

Figure 1.7: Example of ChatGPT changing the style of an image

The most interesting fact about LMMs is that they preserve their reasoning capabilities, making them suitable for complex reasoning in heterogeneous data contexts. Let’s consider this last example (showing only the first lines of the response):



Cross Words

Across

2. How is it called the fusion between the head and the thorax in chelicerae arthropods?

3. What class do scorpions belong to?

8. Transformation from an immature form to an adult?

9. If the tube adjacent to the excretory pore is green, the gland how is it called?

11. What kind of symmetry do arthropods have?

12. What is in the category of crustaceans, starts with a M and has twelve letters?

14. What kind of eye is form of small visual units?

15. What is it called when the segments merge and can move independently?

Down

1. Changes the behavior by spreading a chemical, that chemical is called...

4. What has 3 body regions and 6 legs?

5. What are the names of Appendages in front of the mouth in arthropods?

6. A flexible cover for invertebrate animals.

7. What body development do arthropods have?

10. What has only one len and is located in an insect?

13. Molting the skin, that process is called...

can you solve this for me?

Sure, let's solve the crossword puzzle. Here are the answers based on the given clues:

Across:

2. **Cephalothorax** (Fusion between the head and thorax in chelicerate arthropods)
3. **Arachnids** (Class that scorpions belong to)
4. **Metamorphosis** (Transformation from an immature form to an adult)

Figure 1.8: Example of ChatGPT reasoning over a puzzle and solving it

As you may imagine, this opens a landscape of applications in various industries, and we are going to see some concrete examples in the upcoming chapters.

The need for an additional layer of intelligence: introducing AI agents

LLMs have demonstrated impressive capabilities in generating coherent text, answering questions, and even performing limited problem-solving. However, their fundamental design presents several limitations when it comes to real-world applications:

- **Lack of long-term memory**: Most LLMs operate within a fixed context window, meaning they forget previous interactions once the context limit is exceeded. This makes them incapable of learning from past experiences or maintaining continuity over time.

- **No persistent goals or autonomy**: LLMs respond to individual prompts but do not operate with a persistent goal-oriented mindset. They cannot proactively make decisions, self-correct, or refine their approach over time.
- **Limited reasoning and multi-step execution**: While LLMs can follow instructions within a single prompt, they struggle with executing multi-step workflows, handling complex decision-making, and maintaining logical coherence across long interactions.
- **Inability to interact with external systems**: Without additional integrations, LLMs cannot retrieve real-time information, call APIs, manipulate databases, or execute actions beyond text-based output.

To address these challenges, AI agents introduce an additional layer of intelligence that enables models to act autonomously, reason through tasks, interact with external environments, and learn from past interactions.We will define the anatomy of an AI agent in the next chapter. However, you can start thinking about it as a system that combines an LLM with additional capabilities such as memory, planning, and multi-step reasoning to perform tasks with minimal human intervention, leading to a great degree of autonomy. Unlike standard LLMs that generate static responses, AI agents can do the following:

- **Persist across interactions** by maintaining memory and adapting their behavior over time
- **Break down complex tasks** into smaller steps and execute them sequentially
- **Interact with tools and APIs** to retrieve real-time data, automate workflows, and take meaningful actions
- **Make decisions autonomously** based on learned knowledge, goals, and constraints

At their core, AI agents act as intelligent assistants capable of handling sophisticated tasks beyond simple question-answering. We will explore them in detail in the upcoming chapters.

Summary

Over the past two years, AI has undergone a profound transformation, shifting from simple LLM API calls to more sophisticated, interactive, and autonomous systems. The rapid evolution of LLMs has been marked by innovations such as

RAG, fine-tuning advancements, and reasoning-focused architectures, all aimed at improving efficiency, adaptability, and cost-effectiveness.Despite these advances, LLMs alone are not enough to meet the growing demand for AI systems that can operate autonomously, make decisions, and interact meaningfully with their environments.This shift marks a pivotal moment in AI development, where the focus is no longer just on making models **bigger** but on making them **smarter**. Instead of treating AI as a passive tool that responds to isolated prompts, we are now designing **agentic systems** that can act, learn, and adapt to complex real-world tasks.In the next chapter, we will examine the emergence of AI agents, their primary components, and the various forms they may take.

References

- *Knowledge Distillation: A Survey*: https://arxiv.org/pdf/2006.05525
- *OpenAI o1*: https://en.wikipedia.org/wiki/OpenAI_o1?utm_source=chatgpt.com
- *Reasoning Language Models: A Blueprint*: https://arxiv.org/abs/2501.11223
- *LoRA*: https://arxiv.org/abs/2106.09685
- *Adapter Tuning*: https://arxiv.org/abs/2304.01933
- *Prefix Tuning and Prompt Tuning*: https://ericwiener.github.io/ai-notes/AI-Notes/Large-Language-Models/Prompt-Tuning-and-Prefix-Tuning

2 The Rise of AI Agents

Join our book community on Discord

https://packt.link/4Bbd9

This chapter will explore the evolution of AI agents, tracing their roots from early **robotic process automation** (**RPA**) systems to the sophisticated multi-agent architectures of today. We will define what an AI agent truly is, break down its essential components, and examine the different types of AI agents that are shaping industries across the globe.In this chapter, we are going to cover the following topics:

- Evolution of agents from RPA to AI agents
- Definition of an AI agent
- Different types of AI agents
- Components of an AI agent

By the end of this chapter, you will have a clear understanding of the evolution of AI agents, their key components, and how they are transforming industries.

Technical requirements

You can access the complete code for this chapter in the book’s accompanying GitHub repository at https://github.com/PacktPublishing/AI-Agents-in-Practice.

Evolution of agents from RPA to AI agents

The journey from traditional rule-based automation to sophisticated AI-driven agents has been marked by significant technological advancements. Initially, automation was limited to rigid, predefined workflows, but with the rise of machine learning, reinforcement learning, and **large language models** (**LLMs**), AI agents have evolved to become more autonomous, intelligent, and capable of complex decision-making. Let's have a look at the different flavors agents have had in recent decades.

- **Robotic process automation (RPA**): This represents the earliest phase of automation, focused on rule-based systems designed to execute predefined tasks. These systems followed strict logical flows, executing actions based on explicit conditions and structured inputs. While effective for repetitive processes, RPA lacked flexibility, adaptation, and the ability to process unstructured data. Take, for example, a rule-based chatbot that follows a strict decision tree without learning from interactions; however, it cannot handle dynamic environments or unexpected inputs.
- **Traditional ML/RL-based agents:** As artificial intelligence progressed, traditional **machine learning** (**ML**) and **reinforcement learning** (**RL**) agents began to emerge. These agents could learn from data, make decisions based on probabilistic models, and optimize their actions through trial and error. Let's double-click on both of them:
  - **Rule-based agents**: These transitioned from static rule sets to machine-learning-based models that could classify and predict outcomes based on training data.

Take, for example, an early customer support chatbot that followed a strict decision tree to respond to user queries, routing them with a mechanism of named entity recognition.

Definition

Named entity recognition (NER) is a **natural language processing** (**NLP**) task that identifies and classifies key information, such as names of people, organizations, locations, dates, and other predefined entities, in a given text.

- **Reinforcement learning (RL) agents**: These learned by interacting with environments, optimizing actions for long-term rewards. RL agents were widely applied in gaming, robotics, and complex problem-solving domains.

For example, DeepMind's AlphaGo learned to play the board game Go by simulating millions of games and optimizing strategies through trial and error.However, these early agents had a major shortcoming: limited generalization. Take AlphaGo, for instance—while it mastered the game of Go through millions of simulated matches, its intelligence was narrow and domain-specific. AlphaGo couldn't apply its knowledge to other board games like chess or navigate unrelated tasks such as customer service or scheduling. This kind of AI could excel in a tightly scoped environment but failed to adapt when the rules, context, or input patterns shifted.This lack of flexibility revealed a broader challenge in AI: the need for agents that can reason across domains, understand ambiguous instructions, and adapt in real time to dynamic environments.That's where LLM-based agents come into play.

- **LLM-based agents:** With the advent of LLMs, AI agents became more capable of reasoning, planning, and interacting dynamically. Generative AI enabled these agents to not only respond to queries but also synthesize information, automate workflows, and integrate with various external systems – as we will see in detail throughout this chapter.

At a high level, the power of LLM-based agents is that they can leverage models like GPT-4o not only to undberstand context and retrieve relevant information but also to orchestrate a set of components that enable the agent to interact with the surrounding environment. This is the "extra layer of intelligence" that differentiates modern AI agents from both previous RPA systems and LLMs themselves.In addition to that, the moment we leverage large multi-modal models, AI agents can incorporate different modalities, such as text, speech, vision, and structured data, to interact in more human-like ways.For example, you can think about an LLM-based agent as a retail assistant that can process spoken questions, analyze product images, and query inventory databases in real time.

- **Multi-agent systems and self-replicating agents:** A major breakthrough in AI agent evolution has been the introduction of multi-agent systems, where multiple AI agents collaborate to solve complex tasks. These systems allow for task delegation, specialization, and parallel execution, leading to greater efficiency and autonomy. Take, for example, a multi-agent research system where one agent retrieves papers, another summarizes them, and a third generates actionable insights for a team.

In addition to that, we could also provide those agents with “self-replication” capabilities, meaning that they can generate additional agents to handle subtasks, effectively scaling themselves to meet demand. Take, for example, an AI project manager that spawns specialized sub-agents to handle design, coding, and testing in a software development workflow.

- **AGI agents–the next frontier:** The ultimate goal of AI evolution is the development of **Artificial General Intelligence** (**AGI**) agents—systems that can perform any intellectual task a human can. AGI agents would integrate reasoning, planning, memory, and self-improvement to function autonomously across a wide range of applications.

At the time of writing this book, this is something which still has to be fully qualified as common understanding, yet it is an exciting time to witness the always evolving boundaries of AI agents.Throughout this book, we will mainly focus on single, LLM-based agents, touching on the multi-agentic framework in *Chapter 7*. Let’s start with a definition of what an AI agent is.

Components of an AI agent

An **AI agent** is a software-based entity capable of perceiving its environment, reasoning about its goals, making decisions, and executing actions—often autonomously—through interaction with external systems. Unlike traditional automation, which follows pre-programmed rules, AI agents can dynamically adapt based on context, leverage external tools, and incorporate memory to improve decision-making over time.At a technical level, an AI agent consists of several core components:

- **LLM**: The reasoning engine of the agent, providing natural language understanding, response generation, and task planning. LLMs like GPT-4, Claude, and Gemini enable agents to process user inputs, generate responses, and even engage in multi-step reasoning.
- **System message**: Think about the system message as the "mission" of the agent, as it provides underlying **instructions** that shape the agent's behavior. In addition to the main goal, system messages define tone, role, and constraints (e.g., "You are a customer support assistant; respond concisely and empathetically").
- **Memory**: Enables an agent to retain context over time, improving continuity and personalization. At a high level, memory can be differentiated into short-term (session-based) and long-term (databases storing past interactions). However, there are many nuances of agentic memories, including short-term, episodic, and procedural, that we are going to explore in *Chapter 4*.
- **Tools**: Extend an agent's capabilities beyond the LLM. Agents interact with external tools such as APIs, databases, search engines, and automation scripts to fetch real-time data, perform calculations, or trigger external processes.
- **Knowledge base**: Stores structured and unstructured domain knowledge that the agent can reference. This includes **retrieval-augmented generation** (**RAG**) systems, proprietary company data, or specialized knowledge repositories for enhanced decision-making.

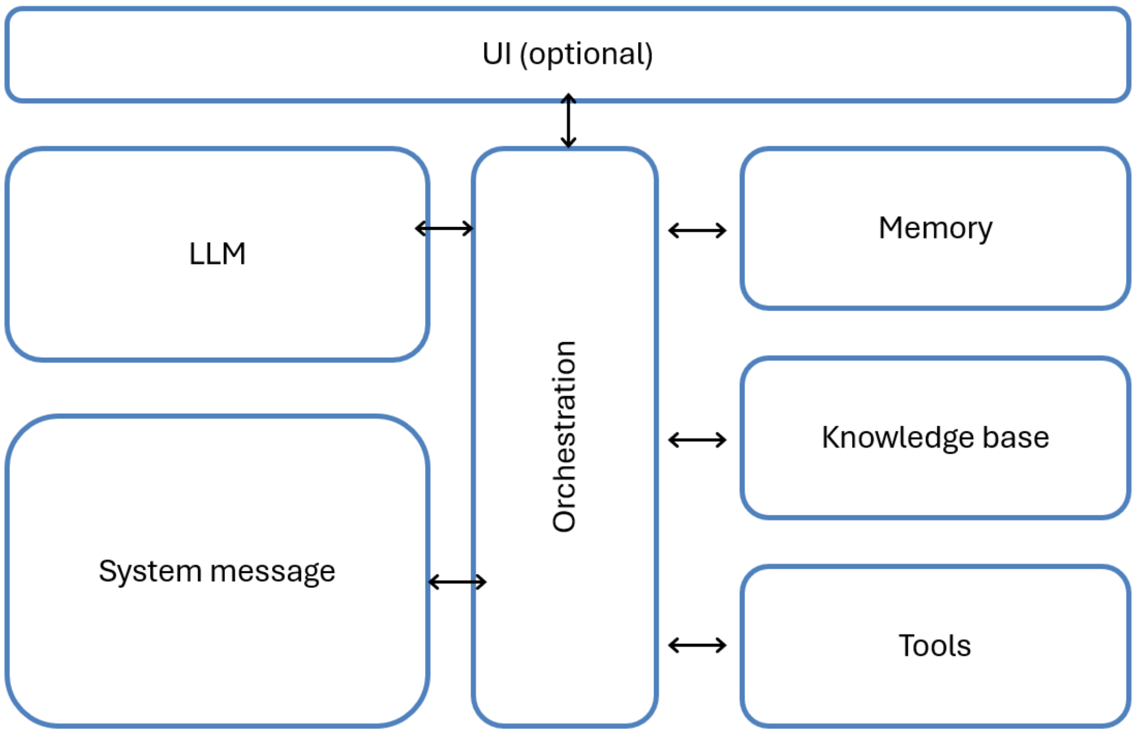


Figure 2.1: Main components of an AI agent

On top of that, we also have an orchestration layer, meant to govern the flow of tasks within the agent, ensuring coordination between components.

Note

AI agents might or might not have a user interface. On one side, agents can be user-facing conversational applications that react based on a user's input (for example, a customer service AI agent addressing a user's query about a specific product). On the other side, they could also act "behind the scenes" in automation workflows; if this is the case, they might not need a UI at all, as they are triggered by events (for example, an AI agent that provides resolution to issues everytime a ticket is raised in the system of record).

Let's consider the following example. Imagine an academic institution developing an AI agent designed to help high-school students grasp complex STEM topics. By leveraging large language models, memory, and orchestration, this agent provides personalized tutoring, references authoritative sources, and adapts to each student's learning needs.

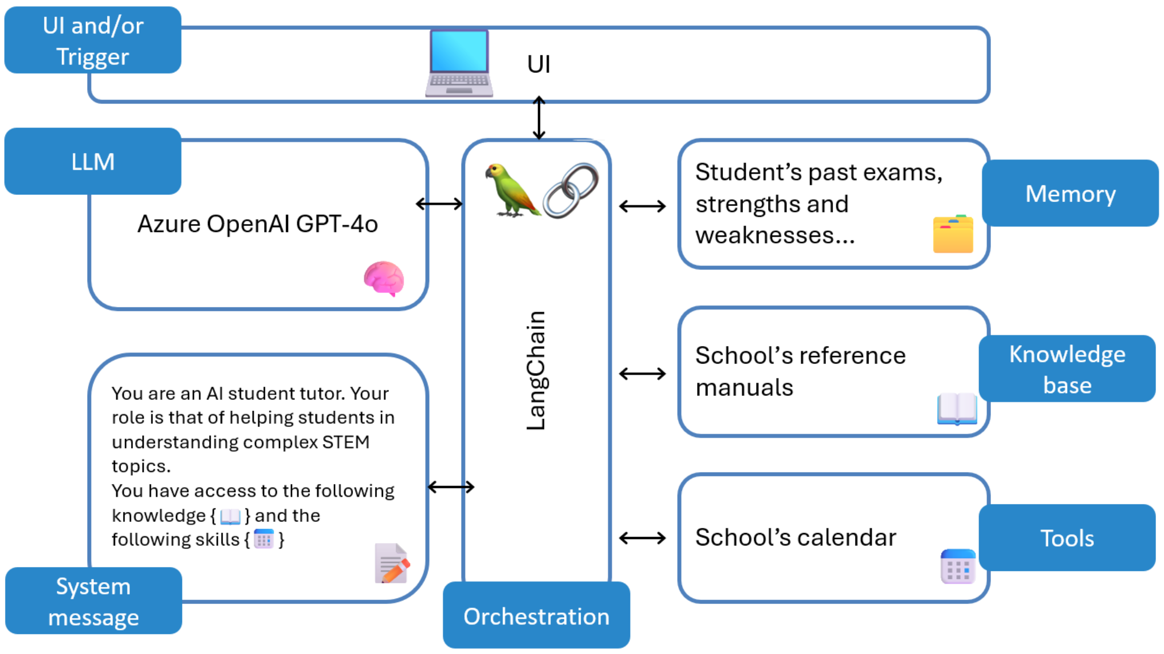

UI and/or Trigger
UI
LLM
Azure OpenAI GPT-4o
LangChain
Student's past exams, strengths and weaknesses...
Memory
School's reference manuals
Knowledge base
You are an AI student tutor. Your role is that of helping students in understanding complex STEM topics. You have access to the following knowledge { } and the following skills { }
School's calendar
Tools
System message
Orchestration

Figure 2.2: Example of an AI tutor assistant

Let's zoom in on each component:

- **LLM**: This acts as the core reasoning engine, the "brain" of the agent, providing explanations, solving problems, and answering student questions in a conversational manner – all thanks to the additional information provided by the agent's components.

Note

It's important to remember that LLMs are typically trained on public or generic datasets. This means they often lack a deep contextual understanding of specific industries, proprietary data, or organizational processes unless explicitly grounded in such information. That's why providing an external knowledge base relevant to the specific use case can equip agents with domain-specific knowledge, improving accuracy, trustworthiness, and usefulness in real-world scenarios.

- **System message**: This defines the agent's personality, ensuring it remains aligned with its educational purpose (after all, we don't want an AI tutor that will do students' homework on their behalf, but rather an assistant that can support them in the learning process, strenghening their weaknesses and focusing on specific learning areas).
- **Orchestration**: This ensures smooth interaction between the UI, LLM, and various components. It routes requests intelligently, deciding when to fetch external data, refer to stored student performance history, or generate content directly from the LLM.
- **Memory**: This tracks students' chat sessions to keep the conversation relevant (short-term). Plus, it stores previous students' interactions to have relevant knowledge about their academic profile. This knowledge allows the agent to use strengths and weaknesses data to reinforce challenging topics and optimize lesson plans.
- **Knowledge**: Relevant knowledge that the agent will need to retrieve to answer specific questions is stored. It is particularly useful if we need to ground the model on a specific set of documents (for example, a school's manuals).
- **Tools and API integrations**: This is the place where we provide our agent with tools to perform actions. An example might be students' and school calendars. This will allow the agent to book a lesson on a student's behalf, according to availability and compatibility with the academic curriculum.
- **UI (student interface)**: This provides an interactive chat-based learning experience, integrating text, diagrams, and step-by-step problem-solving.

Here's how it works in practice:

1. A student asks a complex physics question about Newtonian mechanics.
2. The **LLM** processes the query, using prior interactions and contextual memory.

3. The **orchestrator** determines whether the answer requires reference material, past student performance data, or external web searches.
4. If necessary, the agent retrieves relevant information from the **school's reference manuals**.
5. The LLM **tailors the explanation** to the student's level, reinforcing areas where they've struggled in past exams.
6. The student receives an interactive response, complete with step-by-step explanations, visual aids, and practice questions.
7. Eventually, the agent offers the student the option of booking some extra lessons on the topic in available slots, according to its calendar.
8. If the student accepts, the agent will book lessons on their behalf.

Now, the question is: how can the agent know when to invoke specific knowledge or a specific tool?What makes this mechanism so powerful is the language model's ability to understand natural language. Every time a tool or component—like a "book meeting" action—is initialized, it's not just defined by its underlying logic (such as the API call and POST request to add an event to a calendar). It's also accompanied by a natural language description. This description explains, in plain terms, what the tool does and what kind of output it returns. The LLM reads this description and uses it to decide *when* and *how* to invoke the tool during a task. In essence, the model isn't just executing code—it's reasoning through the available actions based on their human-readable descriptions.

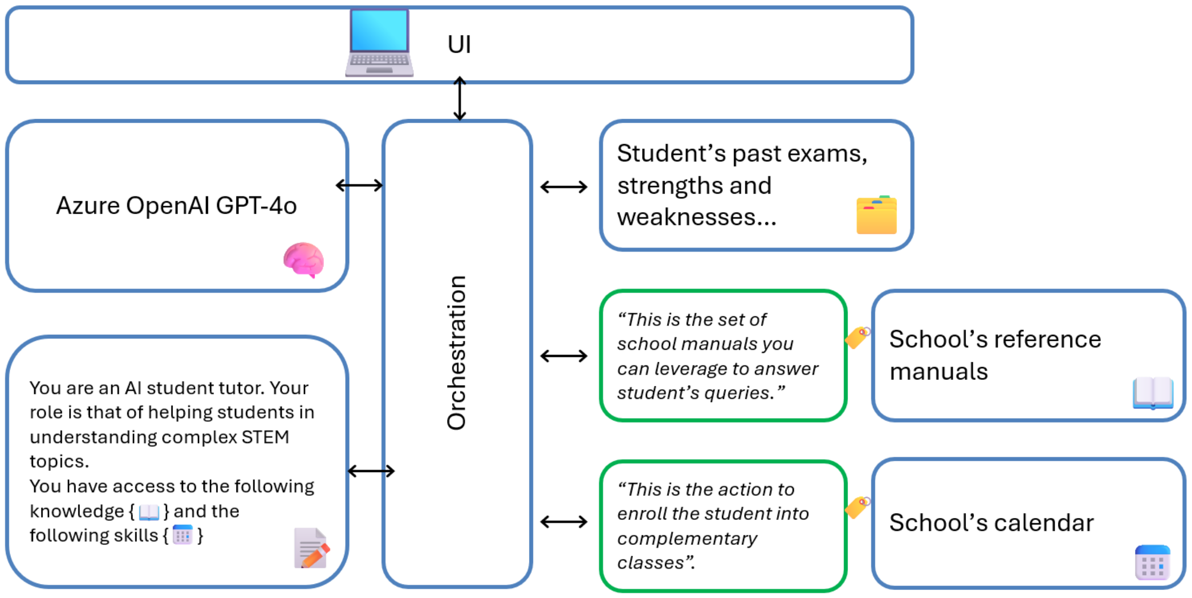

UI
Azure OpenAI GPT-4o
Orchestration
Student's past exams, strengths and weaknesses...
"This is the set of school manuals you can leverage to answer student's queries."
School's reference manuals
You are an AI student tutor. Your role is that of helping students in understanding complex STEM topics.
You have access to the following knowledge { } and the following skills { }
"This is the action to enroll the student into complementary classes".
School's calendar

Figure 2.3: Example of how to describe the agent's components in natural language

Now, the moment the user asks the agent something, the agent – powered by the brain of the LLM – will go and read through all its components' descriptions and understand which one to invoke to solve the user's query.As we will see, there are different strategies we can define for the agent to invoke the proper tool. For example, we might want to have a tool always invoked as the first one, and then let the agent decide whether it needs to invoke other tools or not. One strategy to address this prescriptive order is to simply write it in the system message. For example:You are a helpful AI assistant. You have access to the following tools:

- Tool A
- Tool B

When you receive the user's query, always invoke Tool A. If you cannot accomplish the task with Tool A, then invoke Tool B. Make sure NOT to invoke Tool B before trying Tool A.These strategies are defined at the orchestrator level, as we will see in *Chapter 3*.

Different types of AI agents

AI agents come in varying levels of complexity and capability, ranging from simple retrieval-based agents to fully autonomous systems. Understanding these different types helps organizations and developers select the right kind of AI agent for specific use cases. In this section, we are going to cluster AI agents into three primary types: retrieval agents, task agents, and autonomous agents.

Retrieval agents

In *Chapter 1*, we introduced the concept of RAG as a technique in GenAI applications where an LLM retrieves relevant documents or snippets from a knowledge base (properly embedded and stored in a VectorDB) before generating responses.**Retrieval AI agents** build upon the foundations of RAG but incorporate advanced agentic behaviors, making them more autonomous and adaptive. In fact, we are adding to a standard RAG pipeline an additional layer of intelligence and planning that allows the agent to “strategize” on how to retrieve the most relevant pieces of information.

Note

> Retrieval AI agents are often referred to as agentic RAG. With this approach, knowledge sources are treated as “tools,” meaning that they will come with a description in natural language so that the agent can decide which source to invoke depending on the user’s query. Once the source is invoked, the retrieval mechanism follows the same pattern as traditional RAG, yet we will have this additional layer of intelligence that can decide whether it is enough to answer or not, and if necessary, invoke further sources.

Let’s consider the following example. Let’s say we want to build an AI assistant for a doctor to quickly retrieve information about treatments. Given the doctor asks: *"What are the latest treatments for Type 2 diabetes?"*, let’s see how the two approaches compare:

- **Traditional RAG approach**:

    - The RAG system retrieves the top three relevant articles from the database.
    - The model extracts relevant text from those articles and generates a response summarizing key treatments.
    - If the retrieved documents do not fully answer the doctor's question; the model cannot refine the search unless the doctor manually submits a new query.

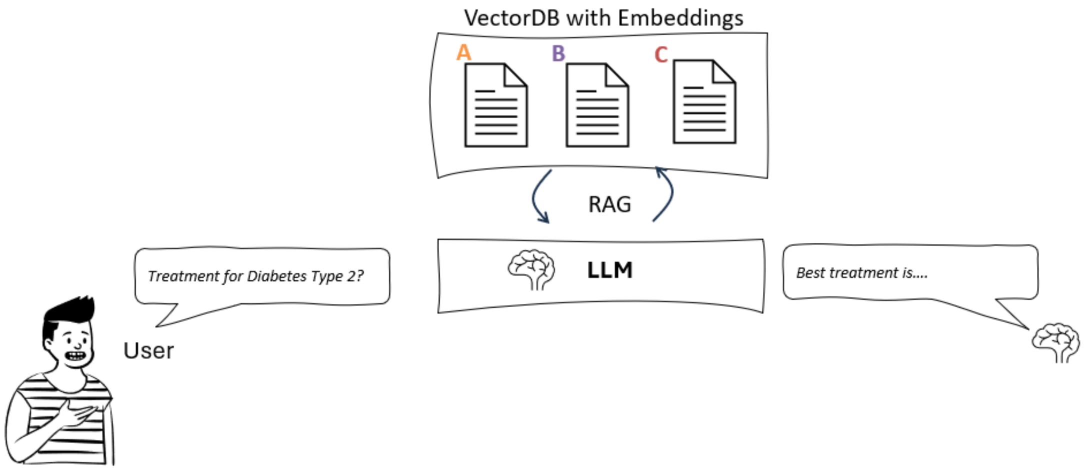


Figure 2.4: Traditional RAG pipeline

- **Retrieval AI agent approach**
    - The agent retrieves an initial set of documents and analyzes them.
    - It detects that some retrieved studies are outdated, so it refines its search criteria and retrieves more recent publications.
    - It recognizes a gap in information regarding a specific drug and fetches a dedicated study on that drug.
    - Finally, it synthesizes all retrieved sources into a comprehensive answer, ensuring relevance and completeness.

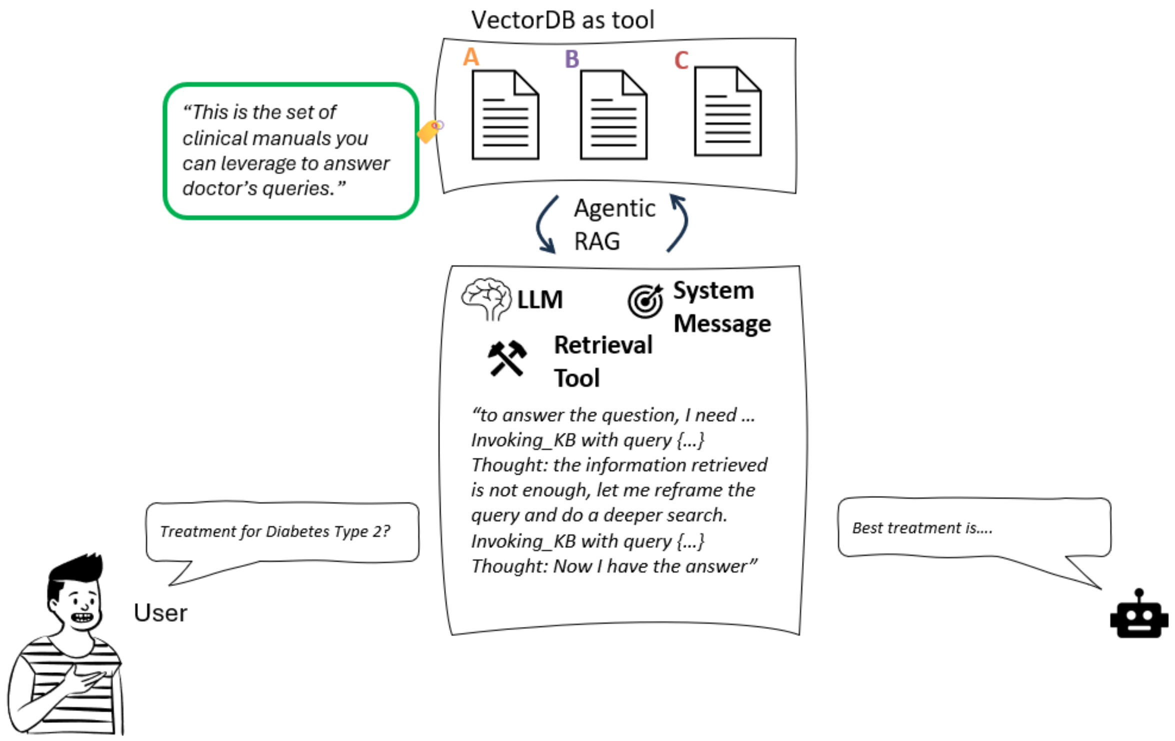

VectorDB as tool
A
B
C
"This is the set of clinical manuals you can leverage to answer doctor's queries."
Agentic RAG
LLM
System Message
Retrieval Tool
"to answer the question, I need ...
Invoking_KB with query {...}
Thought: the information retrieved is not enough, let me reframe the query and do a deeper search.
Invoking_KB with query {...}
Thought: Now I have the answer"
Treatment for Diabetes Type 2?
User
Best treatment is....

Figure 2.5: Agentic RAG pipeline

In conclusion, agentic RAG can lead to several improvements over traditional RAG:

- **Multi-step and recursive retrieval**: Instead of a single-pass retrieval, AI agents iteratively refine their search, breaking down complex queries into multiple steps.
- **Contextual awareness**: They maintain a memory of previous interactions, allowing them to ask clarifying questions or adjust retrieval strategies dynamically.
- **Tool-driven query execution**: Retrieval AI agents can interact with APIs, databases, or vector search engines, making them capable of fetching real-time and structured data.
- **Adaptive knowledge augmentation**: Unlike static retrieval in RAG, AI agents can enrich responses by fetching information from different sources and synthesizing them contextually.
- **Autonomous decision-making**: These agents can determine when to retrieve more information, which sources to query, and how to refine

their results for optimal relevance.

Retrieval agents are the simplest form of AI agents, yet the extra layer of intelligence is already demonstrating great improvements for the overall user experience. However, the real power of an AI agent comes to life when they can combine retrieval skills with actionable tasks.

Task agents

Task agents go beyond information retrieval by performing specific actions. These agents are designed to automate workflows and replace repetitive tasks for users. Unlike retrieval agents, they execute predefined actions in response to user commands or external triggers.

Note

When talking about AI agents, you will often hear the terms tasks, tools, skills, plugins, functions, and actions as interchangeable ways to refer to the agent's capabilities of "doing things." You will also see that different AI orchestrators come with different terminology. Let's try to get some clarity:

Tasks define what needs to be accomplished and can range from simple actions, like sending an email, to complex processes involving multiple actions.

Tools provide external means to perform tasks, like a data visualization tool to create charts or a language translation service to interpret text in different languages.

Plugins extend functionality through integration with other platforms, and they typically come with a set of operations or functions that can be executed against that platform (list rows, append new record…).

Functions outline internal methods of operation – for example, a get_weather function, properly defined, will be able to return the current weather in a given location.

Skills represent the agent's learned proficiencies and are typically defined in a declarative way (natural language). You can think about skills as “mini-prompts” that are invoked only in cases where that specific skill is needed.

Actions are the concrete steps or operations that an AI agent takes in response to a given situation or input. They are the real-time manifestations of an agent's functions and skills, leading to observable outcomes.

Let’s consider once again an example in the healthcare domain, this time from the perspective of the general practitioner’s office receptionist, John.John manages a high volume of appointment requests. Patients book visits through various channels: phone calls, emails, and an online booking system. Managing last-minute cancellations and rescheduling requests is time-consuming and often leads to gaps in the schedule.A typical process in John’s day might look like the following:

1. John receives an email from Patient X to book an appointment and shares some preferences in terms of date and time
2. John checks the availability of the specialist practitioner required and tries to match the earliest slot possible with Patient X’s preferences
3. John doesn’t find any match, hence goes back to Patient X to find alternatives
4. Finally, John and Patient X agree on a slot and the appointment is scheduled

If you think about the above steps, they are nothing but tasks that John is meant to perform to achieve the goal – scheduling the appointment in an optimized slot for both the practitioner and the patient.Whenever we want to map and enhance a business process with an AI agent – more specifically, a task agent – a good practice is that of transposing the human tasks into agentic tasks. Let’s see, for example, how a task agent can assist John:

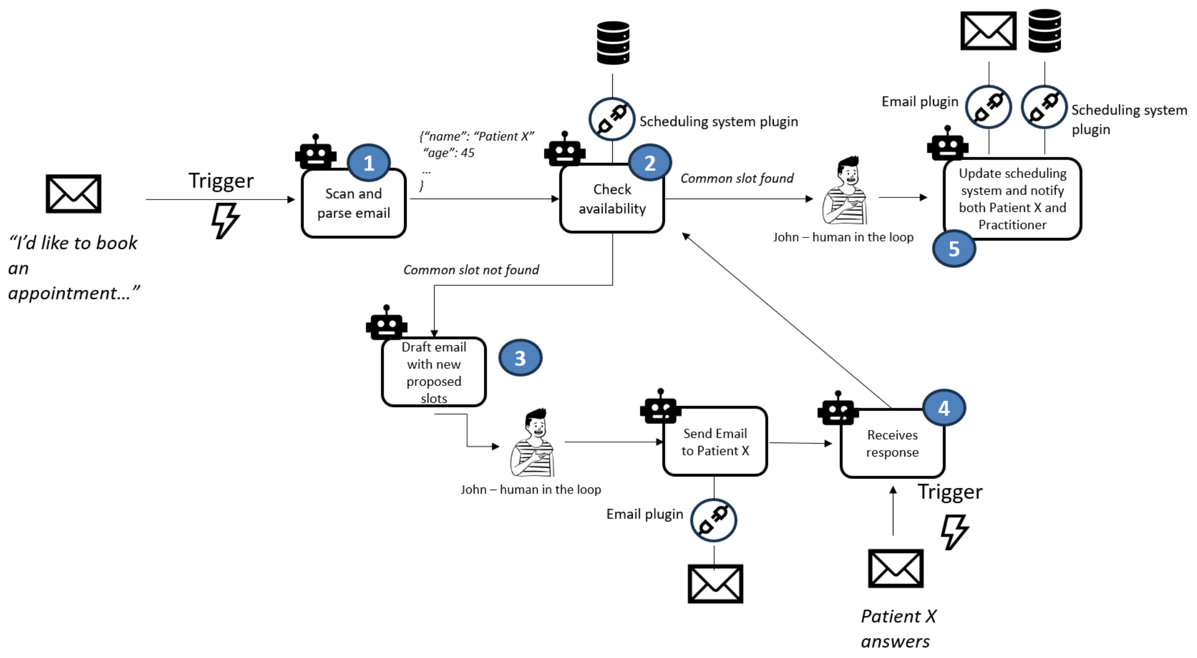


Figure 2.6: How a task is performed by a task agent

1. The AI agent automatically scans the email received from Patient X. It extracts key details like the Patient's name and contact details, preferred date and time, and the specialist practitioner required.
2. The AI agent checks the availability, invoking the plugin (the tool we equip our agent with) to the clinic's scheduling system. It matches Patient X's preferences with the earliest available slots for the specialist practitioner. If there's a match, it proceeds to step 5.
3. The AI agent finds no match. Since no match is found, the AI agent generates a list of the next best available slots based on the specialist's schedule. It also drafts a response email to Patient X, leveraging a writing skill, with suggested alternatives, but John reviews and approves it before sending.
4. Patient X responds with a new preference and either:
   A. Accepts one of them (proceed to step 5)
   B. Requests new options, then the AI agent repeats step 3.
5. Once John and Patient X agree on a slot, the AI agent automatically schedules the appointment in the system, leveraging the same plugin as above. Plus, it sends a confirmation email to Patient X with the details,

leveraging an email plugin. Finally, it updates the specialist's calendar and notifies them of the booking.

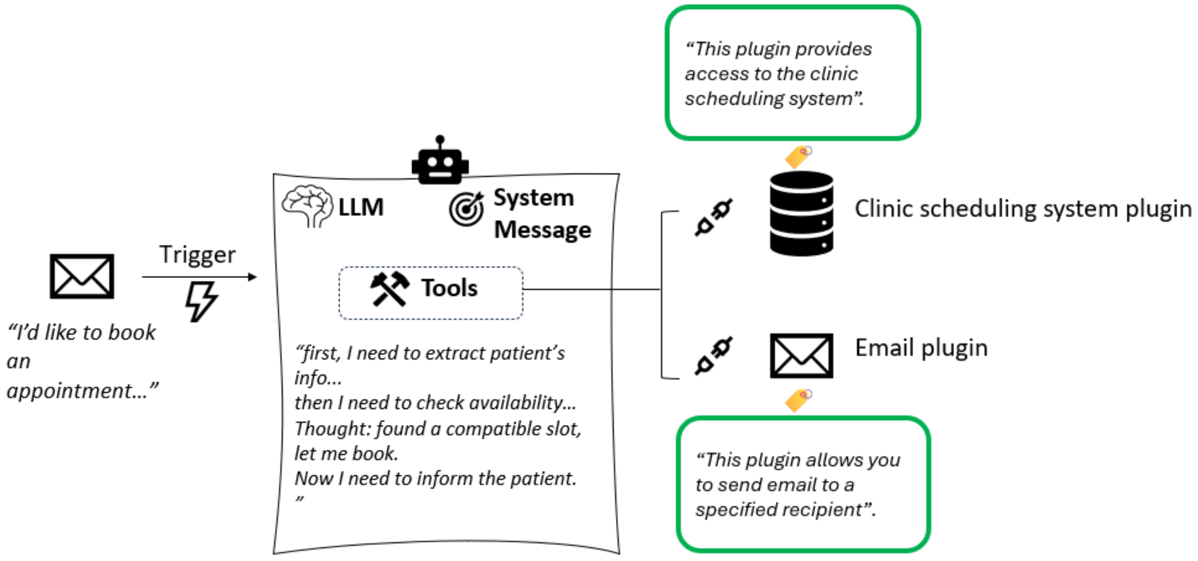

"This plugin provides access to the clinic scheduling system".
LLM
System Message
Clinic scheduling system plugin
Trigger
Tools
"I'd like to book an appointment..."
"first, I need to extract patient's info...
then I need to check availability...
Thought: found a compatible slot, let me book.
Now I need to inform the patient.
"
Email plugin
"This plugin allows you to send email to a specified recipient".

Figure 2.7: Example of the anatomy of the AI agent for the practitioner's office

As you can see, the AI agent acts as John's assistant, handling repetitive scheduling tasks while he focuses on in-person patient interactions.

Autonomous agents

Autonomous agents represent the **most advanced** category of AI agents. Unlike retrieval and task agents, which operate within predefined boundaries, autonomous agents **strategically orchestrate multiple tasks and retrieval processes**, making real-time decisions to optimize workflows. These agents exhibit a high degree of independence, adaptability, and contextual awareness, allowing them to perform complex operations with minimal human intervention.The key distinction of autonomous agents lies in their ability to:

1. Combine retrieval and action: They can both find information (like a retrieval agent) and act on it (like a task agent).

2. Plan and self-adjust: They dynamically adapt based on new information or changing constraints.
3. Perform multi-step workflows: They break down complex tasks into subtasks, execute them iteratively, and adjust based on results. Let's continue with John's clinic example. As the clinic gets busier, managing appointments, cancellations, and reschedules becomes overwhelming. A task agent helped streamline individual actions, but now an autonomous agent takes over the end-to-end scheduling process with minimal supervision. Here is how it works, step by step:
   A. Intake and prioritization: The agent monitors all channels (email, portal, phone transcripts), extracts patient preferences, urgency, and specialist needs, and ranks requests based on priority. For example, a canceled appointment opens a slot, and the agent immediately matches it to Patient X, who's been waiting for a similar time.
   B. Planning and optimization: It reviews the full daily schedule, identifies conflicts or idle gaps, and builds an optimized plan—shuffling low-priority visits to make room for urgent ones.
   C. Execution with feedback: The agent messages patients with options, updates calendars, books appointments, and sends confirmations—all automatically. If preferences change, it loops back, refining its actions.
   D. Real-time adaptation: A doctor calls in sick. The agent halts new bookings, reschedules affected patients, and notifies staff—handling all steps autonomously unless human input is needed.
   E. Continuous learning: At day's end, it analyzes outcomes, updates patient preferences, and adjusts future prioritization logic.

The autonomous agent can plan, retrieve, decide, act, adapt, and learn—all without relying on predefined workflows. John now focuses on edge cases, while the agent intelligently handles the rest.Autonomous agents represent the **next step in AI-driven process automation**. By merging **retrieval AI capabilities** (context awareness, real-time query refinement) with **task execution skills** (appointment scheduling, automated notifications), autonomous agents can **fundamentally reshape** business processes and daily operations.

Note

> Even if autonomous agents resonate very well with the concept of business process automation, keep in mind that they can also represent a new enhancement for customer experience. For example, in the above scenario, rather than calling or sending an email, Patient X could leverage a conversational UI that the AI agent provides (this could be via the clinic website or WhatsApp channel). By doing so, Patient X will experience a new and smoother way of interacting with the clinic, while the AI agent is capturing the intent, asking more questions if further information is needed, and orchestrating the backend to execute its tasks.

There are different degrees of autonomy we can provide our agents with, and the decision is based upon the business scenario as well as the level of confidence we have in the accuracy of the solution.

Summary

AI agents have progressed from basic automation tools to sophisticated autonomous systems, transforming business operations and professional workflows. This chapter explored the three primary types: retrieval agents, which enhance knowledge access through Agentic RAG; task agents, which automate specific actions like scheduling and email management; and autonomous agents, which combine retrieval and execution with strategic decision-making to optimize complex workflows. Deploying the right type of AI agent for each use case is key to achieving impactful automation and enhancing users’ experience.Starting from the next chapter, we are going to dive deeper into each component of an AI agent, starting with the AI orchestration.

References

- DeepMind’s AlphaGo: https://en.wikipedia.org/wiki/AlphaGo#:~:text=AlphaGo%20is%20a%

20computer%20program%20that%20plays%20the,version%20that%20competed%20under%20the%20name%20Master.%20%5B3%5D

- Autonomous agents: https://www.techtarget.com/searchenterpriseai/definition/autonomous-AI-agents
- Reinforcement learning: https://www.tensorflow.org/agents/tutorials/0_intro_rl
- AGI: https://www.ibm.com/think/topics/artificial-general-intelligence

3 The Need for an AI Orchestrator

Since the emergence of large language models and the explosion of AI applications, developers have faced a growing challenge: how to effectively manage and coordinate increasingly complex AI systems. As AI agents become more capable and autonomous, their behaviors must be structured, monitored, and optimized—often across multiple tools, services, and data sources. This growing complexity has created an urgent need for orchestration: a way to ensure these intelligent components work together seamlessly toward a common goal.AI orchestrators emerged in response to this need. Rather than simply offering pre-built components, they provide a framework for structuring interactions, managing dependencies, and maintaining control over multi-agent or modular workflows—all while accelerating development and reducing operational risk.In this chapter, we are going to cover the following topics:

- Introduction to AI orchestrators
- Core components of AI orchestrators
- Overview of the most popular AI orchestrator in the market
- How to choose the right orchestrator for your AI agent

By the end of this chapter, you will be familiar with the most popular AI orchestrators and how to leverage them for your unique agentic use case.

Introduction to AI orchestrators

It is now clear that leveraging large language models (LLMs) goes far beyond simple API calls—it involves orchestrating tools, managing memory, and coordinating complex interactions to build truly intelligent systems. While early AI integrations relied on direct interactions with models, modern AI agents demand a more structured approach to manage workflows, integrate external tools, and handle memory efficiently. This is where **AI orchestrators** come into play.An AI orchestrator serves as the central hub that coordinates interactions between the model, tools, memory stores, APIs, and other external systems. It ensures that AI agents operate effectively and in a controlled manner.

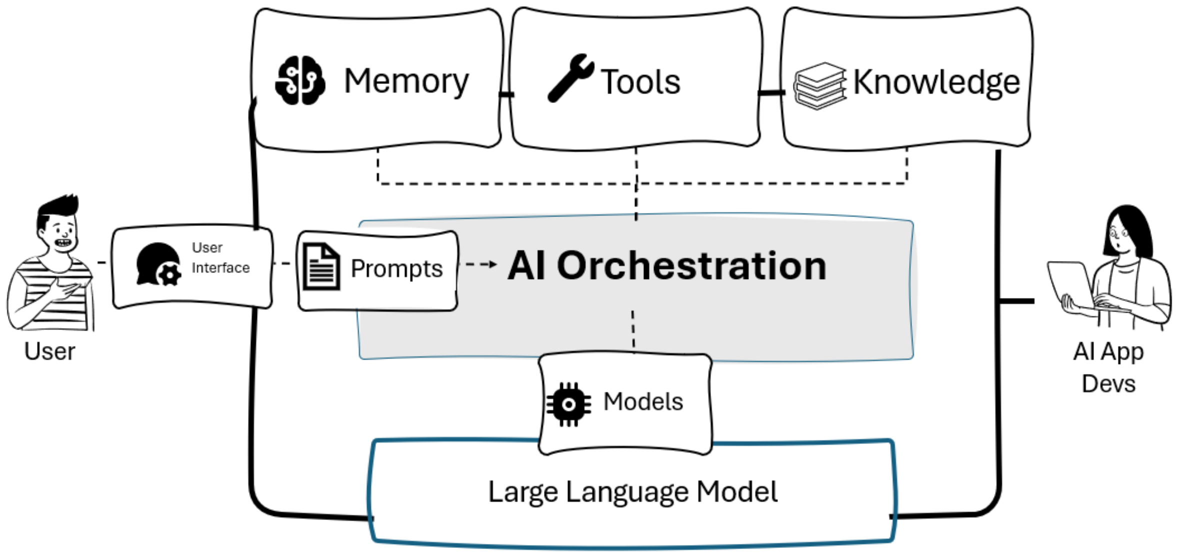


Figure 3.1: Example of an AI orchestrator layer in a typical development framework

AI orchestrators can help in the following:

- **Managing complexity**: AI workflows often involve multiple coordinated steps such as retrieval, reasoning, and action execution. Orchestrators automate and structure these processes, making systems easier to scale and maintain.

- **Enhancing scalability**: Orchestrators handle high loads by distributing tasks, caching responses, and parallelizing operations—critical for handling multiple users or token-intensive tasks.
- **Ensuring context awareness**: Since LLMs have limited memory, orchestrators integrate vector databases and memory systems to help agents retain information and deliver more coherent, personalized experiences.
- **Facilitating tool integration**: Orchestrators streamline the use of APIs, search engines, and databases by managing task execution and ensuring smooth interaction between LLMs and external tools.
- **Improving reliability and monitoring**: From logging to human-in-the-loop feedback, orchestrators offer tools to catch errors, prevent hallucinations, and ensure systems run securely and reliably.

To better understand the need for an AI orchestrator in the specific context of AI agents, we need to introduce three important features of the latter: autonomy, abstraction, and modularity.

Autonomy

Autonomy refers to an AI agent's capacity to operate **independently**, making decisions and executing actions without human intervention. This self-directed behavior enables AI agents to perform tasks, adapt to new situations, and pursue goals based on their learned experiences.The autonomy of an AI agent implies that the steps that the agent is going to take are not necessarily known in advance.For example, let’s consider a non-agentic workflow as simple as the following:

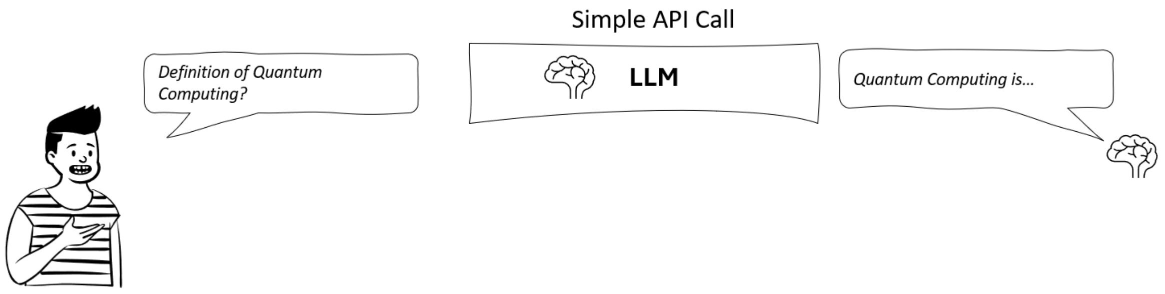


Figure 3.2: Example of a direct API call to the LLM

Whenever we prompt an LLM, we are doing an API call, which is the only step that comprises this workflow. Even in the scenario of a **retrieval augmented generation** (**RAG**) workflow, steps are known in advance:

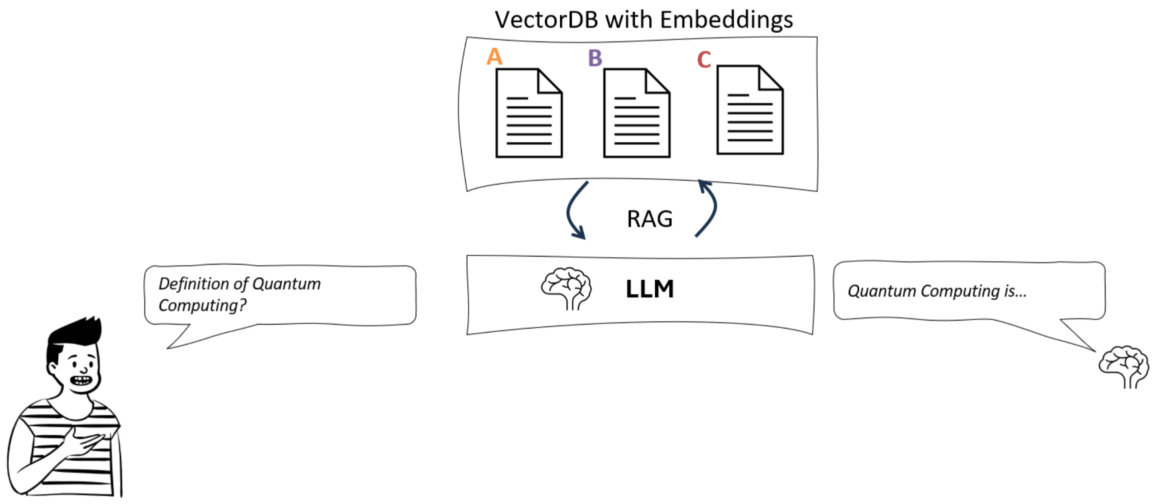


Figure 3.3: Example of retrieval-augmented generation pattern

Let’s now consider an autonomous agentic approach. Let’s say we have an agent with two tools:

- Weather tool, a function that takes two parameters: city and unit of measurement.
- Location tool, a function that takes the current position of the user, leveraging the GPS position. This function takes no parameters.

Both functions come with a natural language description, as per the AI agent’s anatomy. Our workflow design will let the agent decide which tool to invoke.

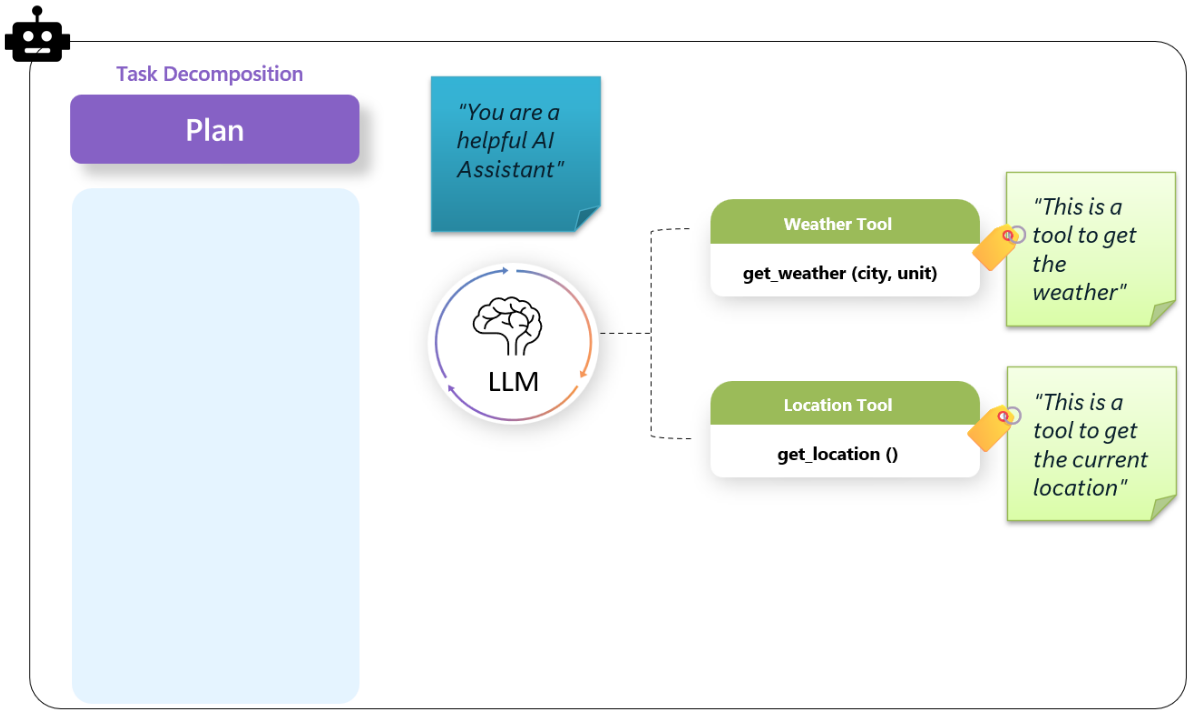

Task Decomposition
Plan
“You are a helpful AI Assistant”
LLM
Weather Tool
get_weather (city, unit)
“This is a tool to get the weather”
Location Tool
get_location ()
“This is a tool to get the current location”

Figure 3.4: Example of an agentic pattern

Let’s say that a new user asks, “What’s the weather for tomorrow?”. The following things will happen:

1. The agent will read through the descriptions of its tools and understand that it needs to invoke the weather tool. However, it’s missing the two parameters, but thanks to its autonomy, it can look around to retrieve them. It soon understands that it can leverage the location tool to get the first parameter: the output of that function will serve as the parameter of the weather tool.

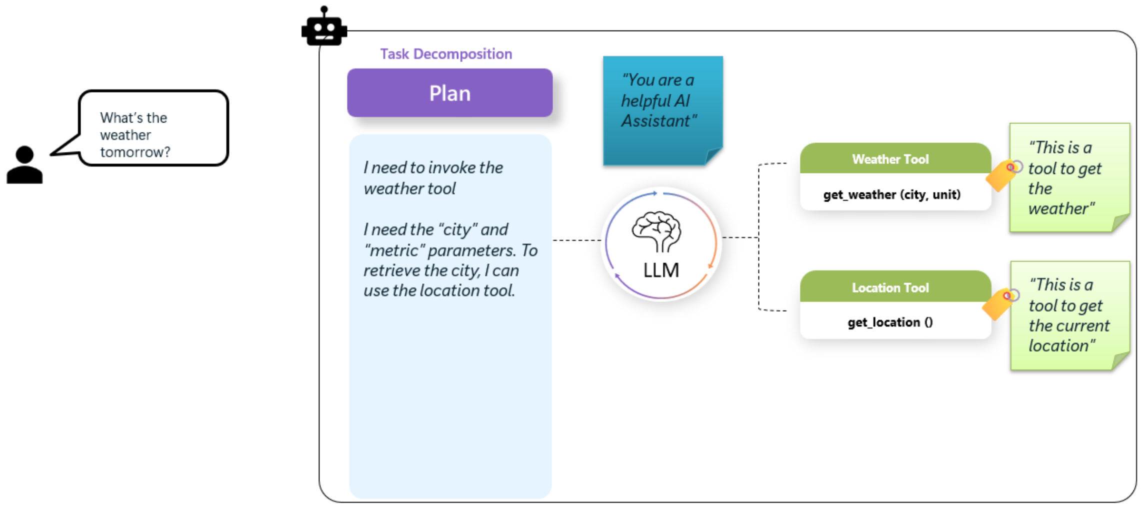

What's the weather tomorrow?
Task Decomposition
Plan
I need to invoke the weather tool
I need the "city" and "metric" parameters. To retrieve the city, I can use the location tool.
"You are a helpful AI Assistant"
LLM
Weather Tool
get_weather (city, unit)
"This is a tool to get the weather"
Location Tool
get_location ()
"This is a tool to get the current location"

Figure 3.5: Example of an AI agent invoking a tool to retfrieve a parameter

2. For the second parameter, the agent cannot make it alone: it needs to ask the user. So it does, asking the user which kind of unit of measurement is needed. Once the user responds, the agent is able to properly invoke the tool with both parameters.

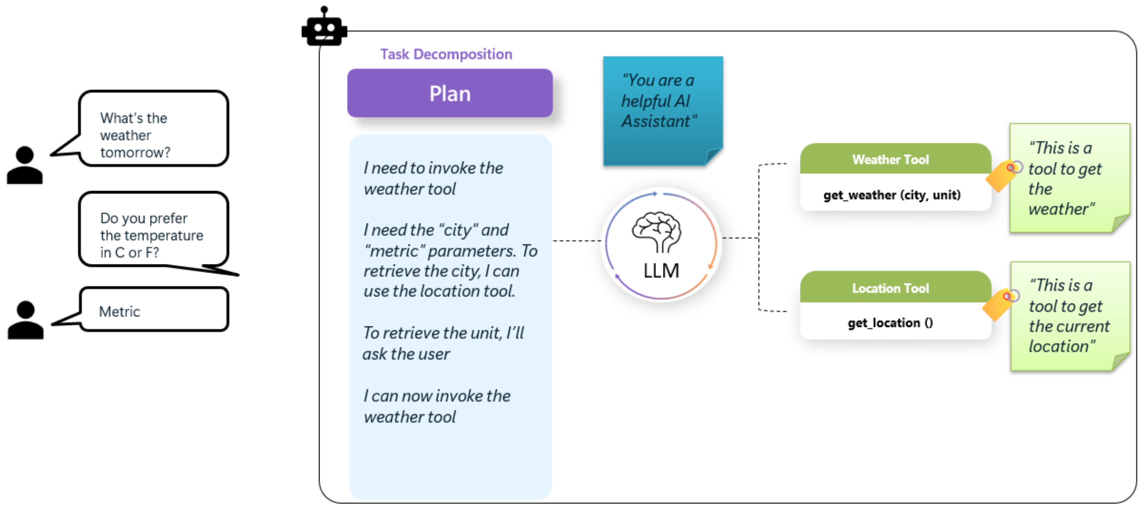

What's the weather tomorrow?
Do you prefer the temperature in C or F?
Metric
Task Decomposition
Plan
I need to invoke the weather tool
I need the "city" and "metric" parameters. To retrieve the city, I can use the location tool.
To retrieve the unit, I'll ask the user
I can now invoke the weather tool
"You are a helpful AI Assistant"
LLM
Weather Tool
get_weather (city, unit)
"This is a tool to get the weather"
Location Tool
get_location ()
"This is a tool to get the current location"

Figure 3.6: Example of an AI agent asking the user for a missing parameter

3. The agent observes the output of the weather tool and ascertains it now knows the final answer for the user.

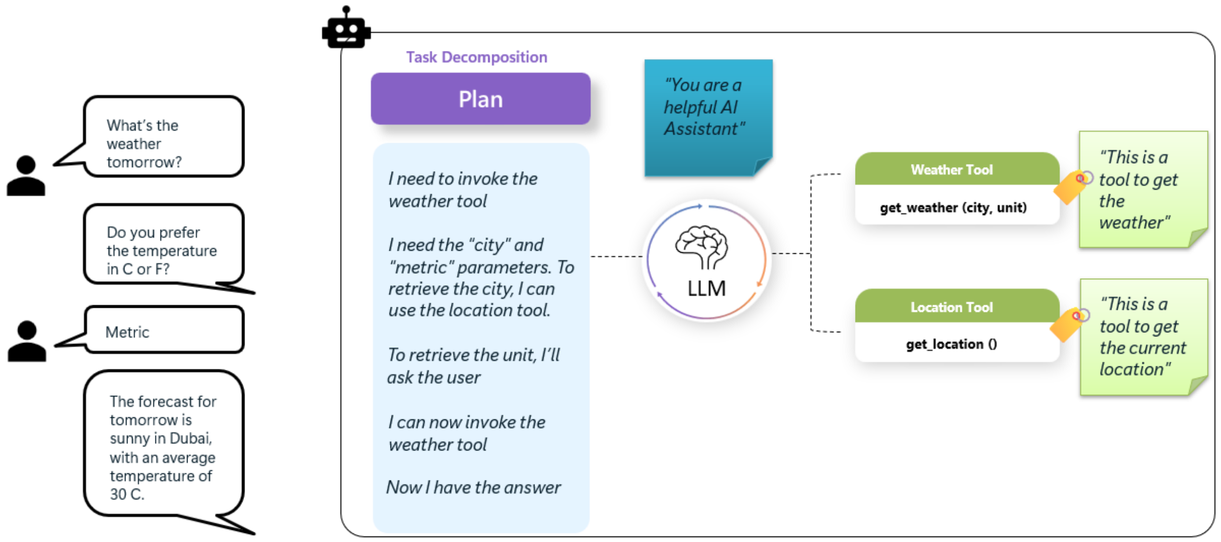

What's the weather tomorrow?
Do you prefer the temperature in C or F?
Metric
The forecast for tomorrow is sunny in Dubai, with an average temperature of 30 C.
Task Decomposition
Plan
I need to invoke the weather tool
I need the "city" and "metric" parameters. To retrieve the city, I can use the location tool.
To retrieve the unit, I'll ask the user
I can now invoke the weather tool
Now I have the answer
"You are a helpful AI Assistant"
LLM
Weather Tool
get_weather (city, unit)
"This is a tool to get the weather"
Location Tool
get_location ()
"This is a tool to get the current location"

Figure 3.7: Example of an AI agent invoking a tool with the retrieved parameters

Can you imagine how many *"if…else"* statements would have been needed to replicate such a degree of autonomy in a standard **Robotic Process Automation** (**RPA**) process? And even if we could manage a similar scenario, what if the user asks something that has not been hardcoded in the process? Adaptability, self-critique, are self-adjustment are key features of agentic autonomy.There are many degrees of autonomy we can provide our agents with: it all boils down to the workflow we set up and the planning strategy we instruct our agents to follow. As we will see throughout this chapter, we can define a plan where the agent will execute tools in a specific order, we can plan to let the agent loop over a tool until it reaches a specific output before going to the next one, or we can set the agent totally free to use all tools as many times as it wishes, when it needs them.Designing the proper agentic workflow is an architectural design conversation that is key when building your agentic state.

Abstraction and modularity

Abstraction refers to breaking down and simplifying complexity. It is what makes these systems comprehensible and scalable. But beyond simplification, it also enables modular design, a fundamental principle in building intelligent systems.Modularity breaks down complex problems into smaller, reusable components, each handling a specific part of the challenge. This approach offers several advantages:

- **Interchangeability**: Components can be swapped, upgraded, or replaced without affecting the entire system
- **Reusability**: Well-designed modules can be repurposed across different projects, improving efficiency
- **Scalability**: Independent yet seamlessly integrated components make it easier to expand solutions

In multi-agent systems, abstraction and modularity allow for the creation of cooperative agents, each specializing in specific tasks while interacting dynamically. This mirrors human problem-solving, where we divide, delegate, and collaborate to tackle complexity effectively.A great way to understand abstraction and modularity in agentic patterns is by looking at a multi-agent traffic management system in a busy metropolitan city, where different levels of agents handle different levels of abstraction, ensuring smooth operation without overwhelming any single entity.

Note

> We will cover multi-agent systems in more detail in *Chapter 7*. However, it is important to note that a standalone agent can always be consumed by another agent as a “tool” with the very same approach of coming with a natural language description of its capabilities. For example, an “SQL agent” can be a tool for a “project manager agent,” the moment the latter needs to query an SQL database.
>
> Henceforth, in multi-agent systems – and in the upcoming examples – think about agents as potential tools for other agents.

At the most granular level, we have intersection controllers, which operate at a single traffic light or intersection. These agents rely on real-time data from cameras and sensors to adjust traffic signals based on vehicle congestion, pedestrian movement, and emergency vehicle priority.They don't worry about what’s happening in the next city block or the broader urban landscape; their only job is to optimize traffic flow at their specific location. If a sudden influx of cars appears at a junction, they may extend green-light durations to ease congestion.Zooming out, we have district-level traffic coordinators. These agents don’t micromanage individual traffic lights but instead analyze traffic flow across multiple intersections within a neighborhood or district.They use data from intersection controllers, GPS tracking, and public transport systems to identify congestion patterns, reroute vehicles, and balance the flow of cars across the area. If they detect excessive delays in one zone, they adjust how much time a light is on for the traffic signal for multiple intersections rather than just one.More importantly, they direct intersection-level agents, ensuring their adjustments align with broader district-wide traffic goals.At the highest level, we have the citywide traffic management system, responsible for optimizing the flow of millions of vehicles across the entire metropolitan area. This agent doesn’t focus on specific traffic lights or individual congestion points; instead, it allocates resources, predicts long-term patterns, and makes strategic adjustments.Using data from weather reports, major event schedules, accidents, and public transportation networks, this agent might reroute entire roads, coordinate construction schedules to minimize disruption, or implement city-wide emergency response plans in case of major incidents.If an accident occurs on a major highway, the citywide system redirects district-level agents to adjust traffic patterns, which in turn instruct intersection controllers to reroute vehicles efficiently.This layered structure demonstrates the power of abstraction and modularity in multi-agent systems:

- Intersection agents handle local, real-time decisions, adjusting traffic lights and prioritizing immediate flow
- District-level agents analyze and coordinate groups of intersections, optimizing traffic across a wider region
- Citywide agents focus on the big picture, planning for long-term efficiency, emergency responses, and systemic optimizations

This mirrors how software architectures, AI systems, and even corporate structures function in the real world. Whether it’s frontline workers executing tasks, middle managers coordinating efforts, or executives setting the overall vision, abstraction enables complex systems to remain scalable, efficient, and resilient.By designing multi-agent AI architectures with this layered approach, we ensure each agent focuses only on what it needs to handle, preventing system overload and enabling adaptive, real-time decision-making at scale, just like a smart traffic system managing a bustling city.If this doesn’t sound like a real thing to you, let’s have a look at OpenAI’s tool called **Operator**, which acts as an autonomous agent, capable of performing tasks in a web browser, such as booking tickets or filling online orders.OpenAI’s Operator follows a hierarchical multi-agent approach similar to the traffic management system. Each agent operates at a different level of abstraction, ensuring efficiency and adaptability without overwhelming any single component.

- **Web controllers (low-level agents)**: These agents handle execution: moving the mouse, clicking buttons, and entering text. They don’t analyze or plan—they simply follow commands.
- **Vision and reasoning (mid-level agents)**: These agents interpret the web interface. The Vision Agent processes screenshots, detecting relevant elements, while the Reasoning Agent determines the next action (clicking, typing, or scrolling). This layer abstracts away the details of execution, focusing on understanding and decision-making.
- **The planner/orchestrator (high-level agent**): The top-level agent oversees the entire system, ensuring that web interactions align with broader goals—whether it’s searching for information or filling out a form. It delegates tasks to mid-level agents, ensuring smooth and strategic navigation.

This structured approach highlights why abstraction is critical in multi-agent design:

- Low-level agents execute without worrying about decisions
- Mid-level agents focus on interpreting and planning
- High-level agents handle overall strategy, without getting into technical details

By leveraging this modular design, OpenAI’s Operator adapts dynamically, handling different websites without requiring manual programming. This scalable and generalizable architecture is a prime example of how multi-agent systems drive real-world AI applications.From an architectural perspective, all these components – agents, skills, plugins – can be seen as repeatable assets in your organization. In this context, AI orchestrators ensure that these components work together without being tightly coupled, preventing complexity from overwhelming the system.Following the preceding hierarchical example, with an AI orchestrator, you can easily define the following:

- **Execution agents** (**low-level)**: These handle raw tasks such as API calls, database queries, or web scraping, executing commands without decision-making
- **Reasoning agents (mid-level)**: They analyze data, determine actions, and select the right tools, abstracting execution details
- **Orchestration and planning (high-level)**: The orchestrator oversees workflows, breaking down tasks, distributing them across agents, and adapting dynamically

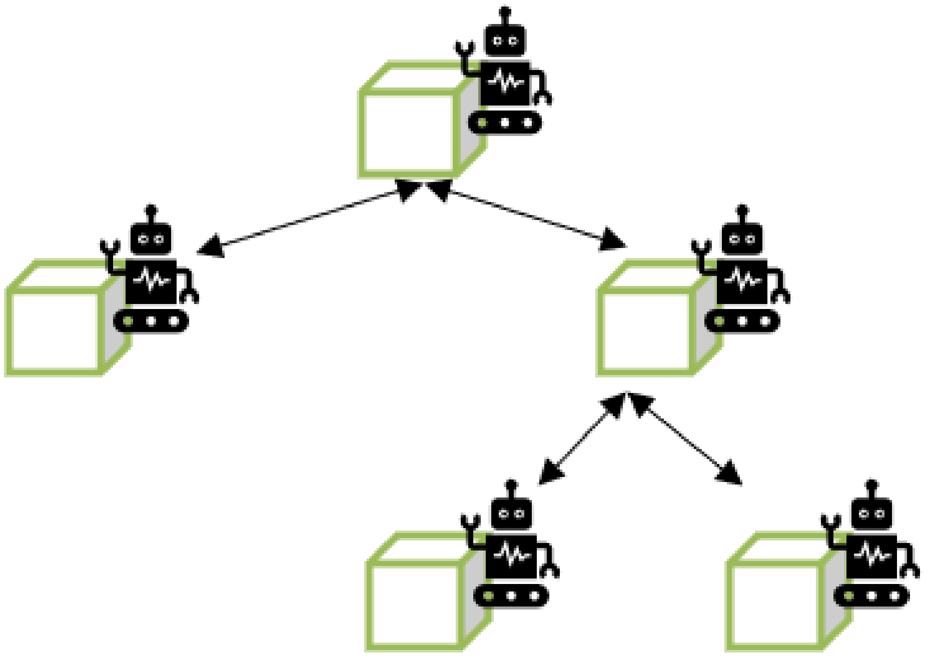


Figure 3.8: AI agents hierarchy

By structuring AI systems this way, orchestrators enable adaptive, generalizable intelligence, ensuring seamless interaction between components without manual intervention.

Core components of an AI orchestrator

Now that we've explored why AI orchestrators are critical in managing complexity, scalability, context, and reliability in agentic systems, it's time to examine how they work under the hood. At the heart of every orchestrator lies a set of foundational components, including workflow execution, memory handling, tool integration, error detection, and security enforcement. Each component plays a crucial role in ensuring that AI agents operate efficiently and reliably.

Workflow management

One of the primary functions of an AI orchestrator is to define and manage structured workflows. Workflows dictate how tasks are executed, whether they occur sequentially, in parallel, or through conditional logic. Below we list some of the most common workflows you can encounter:

- **Sequential workflows**: Tasks are executed step by step in a predefined order. *Example*: A document-processing AI agent first extracts text from images, then summarizes the content, and finally translates it into another language.
- **Parallel workflows**: Multiple tasks are performed simultaneously to optimize efficiency. *Example*: A financial analysis AI agent can process multiple stock trends at the same time to provide a comprehensive market report.
- **Conditional workflows**: Execution paths change based on specific conditions. *Example*: A customer support AI agent might escalate complex queries to a human agent if sentiment analysis detects frustration.
- **Hierarchical workflows**: Tasks are organized in a structured, multi-layered manner, where high-level AI agents delegate subtasks to specialized agents. *Example*: A project management AI agent oversees an engineering workflow, delegating tasks to coding, testing, and deployment AI agents while ensuring overall progress tracking.
- **Group chat workflows**: AI agents collaborate in a conversational environment, exchanging insights and adjusting their actions based on real-time interactions. *Example*: A team of AI agents (e.g., a research assistant, a fact-checking bot, and a summarization model) discusses a topic dynamically, refining outputs before presenting the final response to a user.

Note

Workflow management goes hand in hand with the concept of autonomy we introduced in the preceding section. For example, in a group chat type of workflow, you are providing your multi-agent system with a high degree of autonomy; on the other hand, a sequential workflow is more predictable as you are clearly stating the sequence of agents to be invoked.

AI orchestrators provide developers with tools to design, modify, and optimize these workflows dynamically, making them essential for creating scalable and adaptable AI applications.

Memory and context handling

Effective AI agents need access to historical interactions and external knowledge bases to deliver relevant responses and maintain continuity. Orchestrators handle this through various memory management techniques:

- **Short-term memory**: Stores session-based context, allowing AI agents to recall details within an ongoing conversation. *Example*: A virtual assistant remembers a user’s previous question during a chat session.
- **Long-term memory**: Retains knowledge over extended periods, often stored in vector databases. *Example*: A medical AI system remembers a patient's medical history for personalized recommendations, such as transcripts of past visits, medical reports, allergies or medications taken, and so forth.
- **Semantic memory caching**: When AI orchestrators manage memory, they use caching strategies to optimize retrieval speed and efficiency. Semantic memory caching involves storing frequently asked information in a way that allows AI agents to recall facts, concepts, and relationships without relying on session-based history. *Example*: A customer service AI agent might recall a user's past complaints and retrieve the resolution faster.

Definition

In the context of computing, caching is a technique used to store data temporarily to expedite future access. Traditionally, applications with low latency and high throughput requirements leverage in-memory caching, which involves storing data directly within a system's RAM, enabling rapid data retrieval due to the high-speed nature of RAM. However, in-memory caches typically rely on exact key-value pairs for data retrieval, meaning that a request must match the stored key precisely to retrieve the corresponding data.

With the advent of LLM-powered apps, a new caching system has been introduced: Semantic caching. This focuses on the meaning and context of data (leveraging embeddings) rather than exact matches. This approach stores the results of queries along with their semantic context, enabling the system to recognize and retrieve relevant data even if the new query isn't an exact match to a previous one.

By efficiently managing memory, AI orchestrators ensure that agents provide coherent, informed, and context-aware responses.

Tool and API integration

AI agents often require access to external resources such as databases, APIs, and computational tools. Orchestrators facilitate seamless integration by enabling agents to do the following:

- Fetch real-time data from APIs (e.g., fetching weather updates for a travel assistant AI)
- Access and query databases (e.g., retrieving order details for an e-commerce AI assistant)
- Leverage external computation tools (e.g., using a machine learning API for fraud detection in banking applications)

Orchestrators allow these integrations to be managed efficiently, ensuring AI agents operate with up-to-date and accurate information.

Error handling and monitoring

To ensure AI applications remain reliable, orchestrators implement robust error-handling and monitoring mechanisms:

- **Logging and analytics**: Captures detailed logs of AI interactions for debugging and optimization.

- **Automated error detection**: Identifies failed processes and retries or escalates them automatically.
- **Performance tracking**: Monitors response times, accuracy, and overall system health.
- **Human-in-the-loop integration**: Allows human review for critical decisions. *Example*: A medical AI assistant requires human confirmation before making a diagnosis.

By proactively handling errors and providing comprehensive monitoring, AI orchestrators help maintain high system reliability and trustworthiness.

Security and compliance

Security is a top priority in AI systems, especially when dealing with sensitive data. AI orchestrators incorporate multiple security measures, including the following:

- **Authentication and access control**: Ensuring only authorized users and systems can interact with AI agents
- **Rate limiting**: Preventing abuse by controlling the number of requests an AI agent can process within a specific timeframe
- **Data privacy compliance**: Adhering to regulations such as GDPR or HIPAA by managing user data securely
- **Bias and safety filters**: Implementing safeguards to prevent biased or harmful AI outputs

Security and compliance mechanisms help ensure AI agents operate safely, ethically, and within legal frameworks.The core components of an AI orchestrator—workflow management, memory handling, tool integration, error detection, and security—form the foundation for building robust and efficient AI applications. By leveraging these capabilities, developers can create AI agents that are not only powerful but also reliable, scalable, and secure. Understanding these components allows for better decision-making when selecting or designing an AI orchestration framework.

Overview of the most popular AI orchestrators in the market

Several AI orchestrators have emerged as leaders in this space, each offering unique capabilities tailored to different use cases. Some focus on modularity and flexibility, allowing developers to customize workflows, while others prioritize user-friendly interfaces for rapid prototyping. Here, we’ll explore the most widely used AI orchestrators as of May 2025, highlighting their key strengths and ideal applications.

- **LangChain**: LangChain is a modular framework designed for building applications powered by LLMs. It provides essential components for integrating external tools, managing memory across interactions, and defining agent-based workflows. As an open source project, LangChain boasts extensive documentation and a strong community, making it a go-to choice for developers looking to build robust AI-driven applications.
- **LlamaIndex (formerly GPT Index)**: LlamaIndex specializes in optimizing data retrieval for LLMs, ensuring efficient access to both structured and unstructured data sources. It is particularly effective when combined with LangChain to build knowledge-driven AI agents that require sophisticated search and indexing capabilities. Its ability to bridge the gap between large-scale data and generative AI makes it an invaluable tool for organizations handling vast information repositories.
- **AutoGen**: AutoGen is tailored for developing multi-agent AI workflows, enabling LLM-powered agents to communicate and collaborate on complex tasks. By automating interactions between AI entities, AutoGen facilitates research, reasoning, and content generation, allowing AI systems to make more informed decisions through structured dialogue. It is well-suited for applications that require multiple specialized agents working together toward a common goal.
- **Langflow**: Langflow simplifies the process of designing AI agent workflows through an intuitive visual interface. By integrating seamlessly with LangChain and other orchestration tools, it enables

rapid prototyping and real-time visualization of agent interactions. This makes it particularly useful for developers and researchers who want to experiment with AI-driven automation without deep diving into code-heavy implementations.

- **Semantic Kernel (SK)**: Developed by Microsoft, Semantic Kernel bridges the gap between AI and enterprise applications by combining machine learning capabilities with traditional software development practices. It supports a plugin-based approach, allowing developers to integrate AI-powered workflows into existing business systems. Semantic Kernel is designed to enhance productivity by embedding AI-driven automation directly into corporate software environments.
- **LangGraph**: LangGraph introduces a structured approach to multi-agent collaboration by leveraging graph-based workflows. It provides a framework for designing sophisticated agent-to-agent interactions, ensuring that AI systems communicate in an organized and scalable manner. This makes it particularly valuable for orchestrating AI applications where different agents must collaborate dynamically to solve complex problems.

Now, the question is: how do I choose the right orchestrator for my AI agent?

How to choose the right orchestrator for your AI agent

Selecting an AI orchestrator depends on several factors, including the complexity of the application, the level of customization required, the available ecosystem, and ease of deployment. Here are some key criteria to consider when choosing an orchestrator:

- **Ease of use and modularity**: If you are looking for a quick and modular way to integrate LLMs into applications, **LangChain** is a great choice due to its well-documented, flexible architecture.
  Example: A start-up developing an AI chatbot for customer support might use LangChain to quickly prototype and integrate with its existing database and APIs.

- **Data-intensive applications**: If your AI agent relies heavily on structured or unstructured data retrieval, **LlamaIndex** is optimized for efficiently integrating external knowledge sources. *Example*: A legal AI assistant retrieving and analyzing case law across multiple document repositories would benefit from LlamaIndex’s retrieval capabilities.
- **Multi-agent workflows**: If your application requires multiple agents to interact with each other dynamically, **AutoGen** or **LangGraph** are ideal choices for orchestrating complex AI interactions. *Example*: Research assistant AI where multiple agents collaborate—one summarizing documents, another fact-checking, and a third generating reports—would benefit from these orchestrators.
- **Enterprise-grade AI applications**: If you need strong enterprise integration and security, **Semantic Kernel** is well-suited for Microsoft-based environments and structured AI workflows. *Example*: A corporate AI-powered analytics tool that integrates with Microsoft Teams and SharePoint would align well with Semantic Kernel.
- **Visual workflow design**: If you prefer a no-code or low-code interface for AI workflow design, **Langflow** provides an intuitive UI for rapid prototyping and debugging of AI agent interactions. *Example*: A marketing team creating an AI-driven content generator without deep coding expertise could leverage Langflow’s visual interface for rapid workflow design.

The choice of an AI orchestrator should align with the goals and technical requirements of your AI system. While some orchestrators specialize in modular development, others focus on scalability, multi-agent collaboration, or enterprise integrations. Understanding these distinctions will help you select the best tool for your specific use case.

Summary

AI orchestrators play a pivotal role in the development and deployment of intelligent systems, providing the necessary framework to manage workflows, integrate tools, and maintain efficiency. As AI applications continue to evolve, orchestrators ensure that AI agents operate autonomously, handle complex tasks, and adapt to dynamic requirements.Throughout this chapter, we have explored the fundamental components of AI orchestrators, including workflow management, memory handling, and security. We have also examined some of the most popular orchestrators available today, each offering unique strengths tailored to specific use cases.Selecting the right AI orchestrator depends on various factors, such as integration needs, scalability, and workflow complexity. By understanding their core functionalities, developers and businesses can make informed decisions when choosing an orchestration tool that aligns with their goals.Starting from the next chapter, we are going to deep dive into some of the most compelling components of AI agents, starting from memory and context management.

References

- OpenAI Operator: https://openai.com/index/introducing-operator/
- LangChain: https://www.langchain.com/
- LlamaIndex (formerly GPT Index): https://www.llamaindex.ai/
- AutoGen: https://www.microsoft.com/en-us/research/project/autogen/
- Langflow: https://www.langflow.org/
- Semantic Kernel (SK): https://github.com/microsoft/semantic-kernel
- LangGraph: https://www.langgraph.dev/