

2

LLMs for AI-Powered Applications

In *Chapter 1, Introduction to Large Language Models*, we introduced **large language models (LLMs)** as powerful foundation models with generative capabilities as well as powerful common-sense reasoning. Now, the next question is: what should I do with those models?

In this chapter, we are going to see how LLMs are revolutionizing the world of software development, leading to a new era of AI-powered applications. By the end of this chapter, you will have a clearer picture of how LLMs can be embedded in different application scenarios, thanks to the new AI orchestrator frameworks that are populating the market of AI development.

In this chapter, we will cover the following topics:

- How LLMs are changing software development
- The copilot system
- Introducing AI orchestrators to embed LLMs into applications

How LLMs are changing software development

LLMs have proven to have extraordinary capabilities: from natural language understanding tasks (summarization, named entity recognition, and classification) to text generation, from common-sense reasoning to brainstorming skills. However, they are not just incredible by themselves. As discussed in *Chapter 1*, LLMs and, generally speaking, **large foundation models (LFMs)**, are revolutionizing software development by serving as platforms for building powerful applications.

In fact, instead of starting from scratch, today developers can make API calls to a hosted version of an LLM, with the option of customizing it for



applications, similar to the transition from single-purpose computing to time-sharing in the past.

But what does it mean, concretely, to incorporate LLMs within applications? There are two main aspects to consider when incorporating LLMs within applications:

- **The technical aspect**, which covers the *how*. Integrating LLMs into applications involves embedding them through REST API calls and managing them with AI orchestrators. This means setting up architectural components that allow seamless communication with the LLMs via API calls. Additionally, using AI orchestrators helps to efficiently manage and coordinate the LLMs' functionality within the application, as we will discuss later in this chapter.
- **The conceptual aspect**, which covers the *what*. LLMs bring a plethora of new capabilities that can be harnessed within applications. These capabilities will be explored in detail later in this book. One way to view LLMs' impact is by considering them as a new category of software, often referred to as *copilot*. This categorization highlights the significant assistance and collaboration provided by LLMs in enhancing application functionalities.

We will delve into the technical aspect later on in this chapter, while the next section will cover a brand-new category of software – the copilot system.

The copilot system

The copilot system is a new category of software that serves as an expert helper to users trying to accomplish complex tasks. This concept was coined by Microsoft and has already been introduced into its applications, such as M365 Copilot and the new Bing, now powered by GPT-4. With the same framework that is used by these products, developers can now build their own copilots to embed within their applications.

But what exactly is a copilot?

As the name suggests, copilots are meant to be AI assistants that work side by side with users and support them in various activities, from information retrieval to blog writing and posting, from brainstorming ideas to code review and generation.

The following are some unique features of copilots:

- **A copilot is powered by LLMs**, or, more generally, LFMs, meaning that these are the reasoning engines that make the copilot “intelligent.” This reasoning engine is one of its components, but not the only one. A copilot also relies on other technologies, such as apps, data sources, and user interfaces, to provide a useful and engaging experience for users. The following illustration shows how this works:

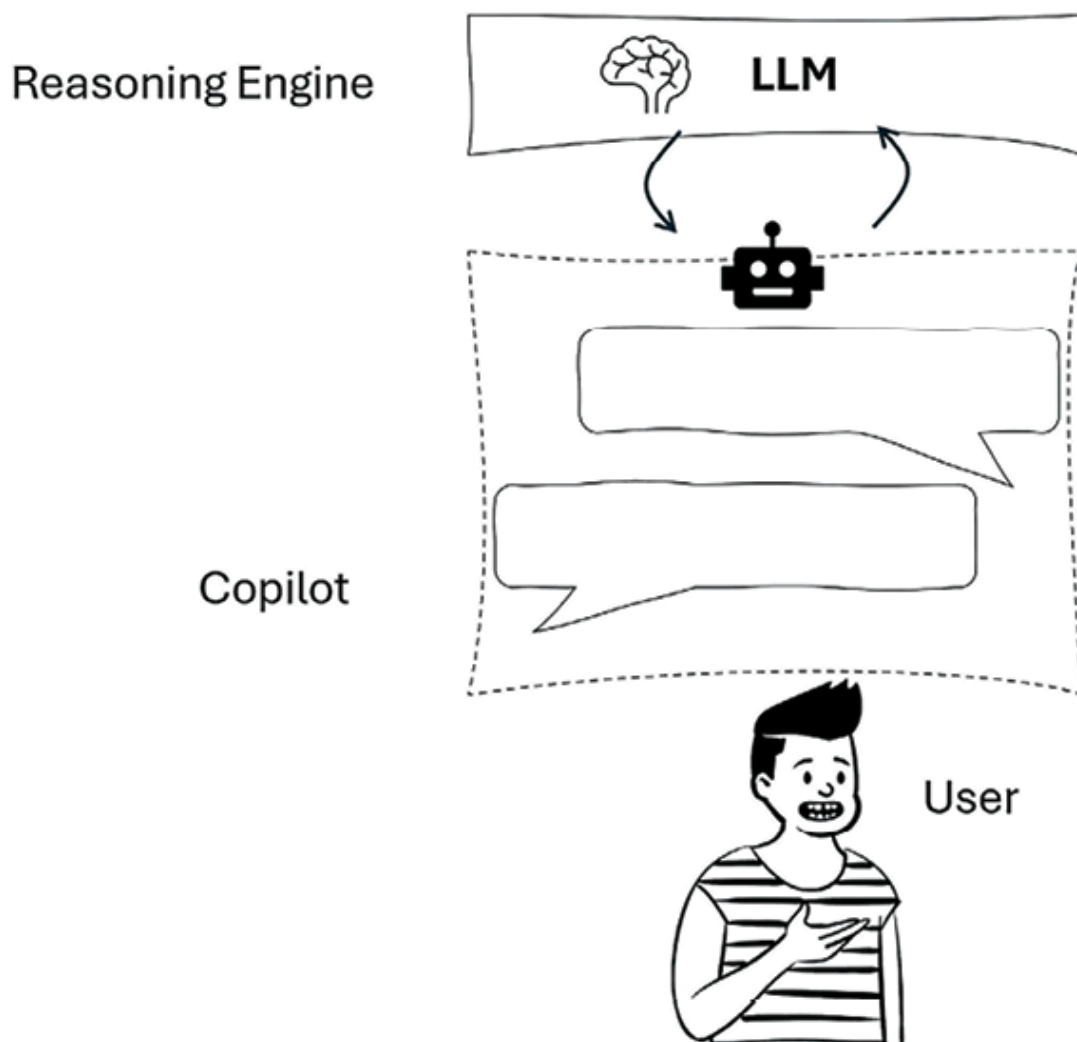




Figure 2.1: A copilot is powered by an LLM

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader and a free PDF/ePub copy** of this book are included with your purchase. Unlock them by scanning the QR code below, or visiting <https://www.packtpub.com/unlock/9781835462317>.



- A copilot is designed to have a **conversational user interface**, allowing users to interact with it using natural language. This reduces or even eliminates the knowledge gap between complex systems that need domain-specific taxonomy (for example, querying tabular data needs the knowledge of programming languages such as T-SQL) and users. Let's look at an example of such a conversation:

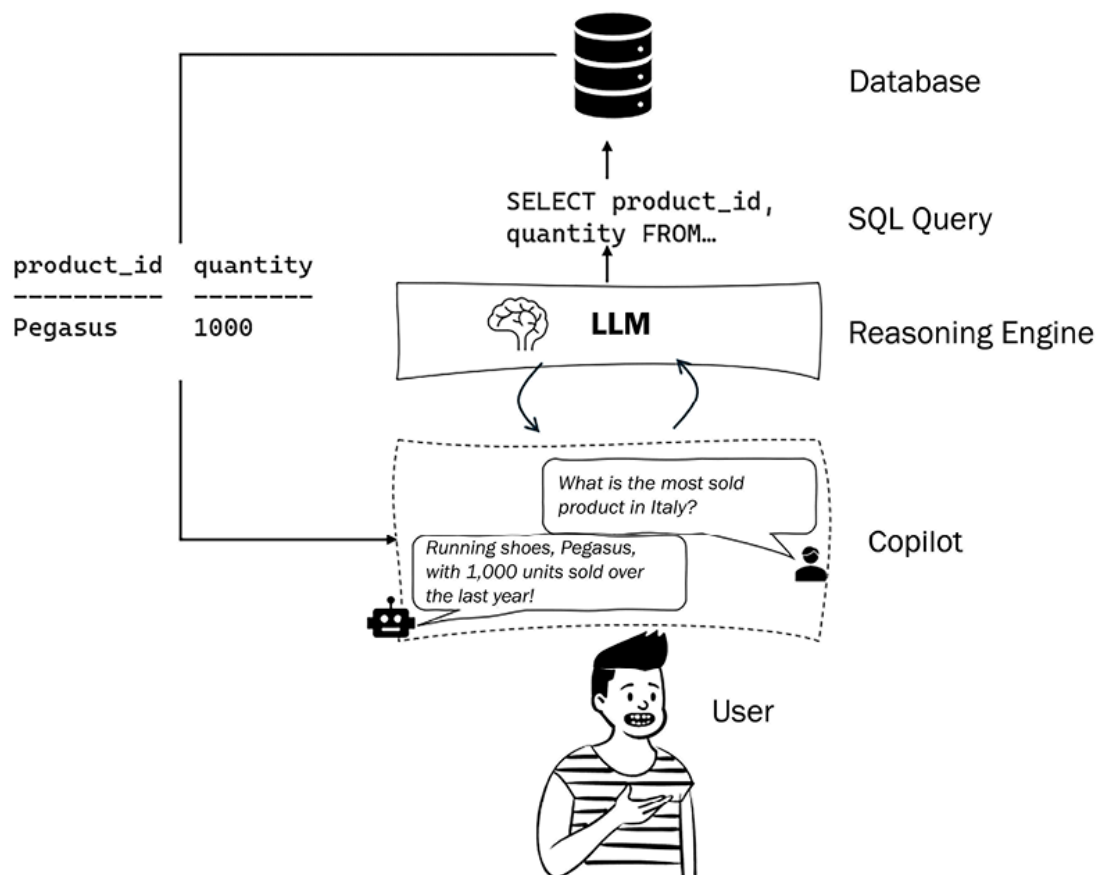


Figure 2.2: An example of a conversational UI to reduce the gap between the user and the database

- **A copilot has a scope.** This means that it is **grounded** to domain-specific data so that it is entitled to answer only within the perimeter of the application or domain.

Definition



Grounding is the process of using LLMs with information that is use case specific, relevant, and not available as part of the LLM's trained knowledge. It is crucial for ensuring the quality, accuracy, and relevance of the output. For example, let's say you want an LLM-powered application that assists you during your research on up-to-date papers (not included in the training dataset of your LLM). You also want your app to only respond if the answer is included in those papers. To do so, you will need to ground your LLM to the set of papers, so that your application will only respond within this perimeter.

Grounding is achieved through an architectural framework called retrieval-augmented generation (RAG), a technique that enhances the output of LLMs by incorporating information from an external, authoritative knowledge base before generating a response. This process helps to ensure that the generated content is relevant, accurate, and up to date.



What is the difference between a copilot and a RAG? RAG can be seen as one of the architectural patterns that feature a copilot. Whenever we want our copilot to be grounded to domain-specific data, we use a RAG framework. Note that RAG is not the only architectural pattern that can feature a copilot: there are further frameworks such as function calling or multi-agents that we will explore throughout the book.

For example, let's say we developed a copilot within our company that allows employees to chat with their enterprise knowledge base. As fun as it

can be, we cannot provide users with a copilot they can use to plan their summer trip (it would be like providing users with a ChatGPT-like tool at our own hosting cost!); on the contrary, we want the copilot to be grounded only to our enterprise knowledge base so that it can respond only if the answer is pertinent to the domain-specific context.

The following figure shows an example of grounding a copilot system:

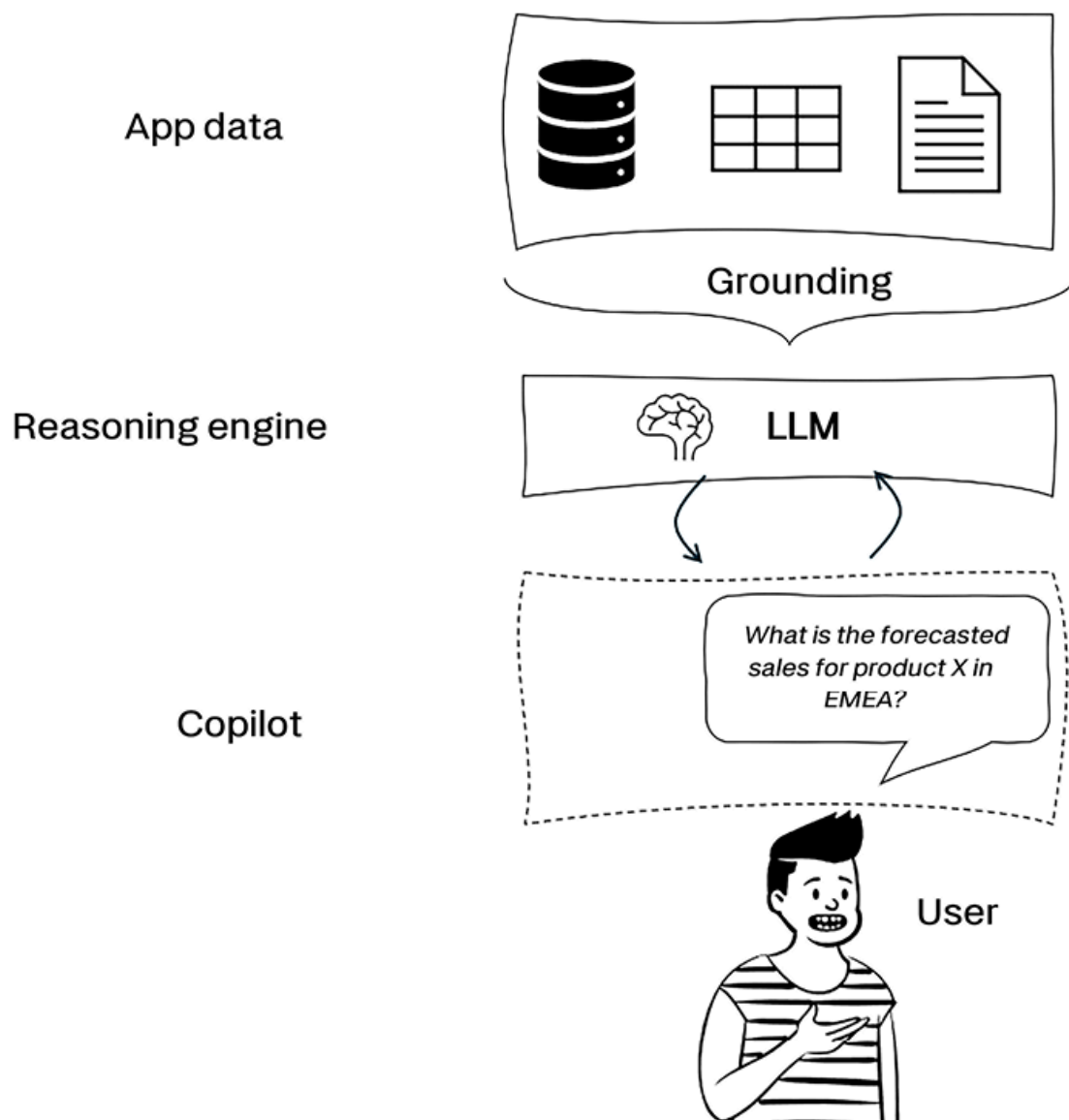


Figure 2.3: Example of grounding a copilot

- **The copilot's capabilities can be extended by skills**, which can be code or calls to other models. In fact, the LLM (our reasoning engine) might have two kinds of limitations:
 - **Limited parametric knowledge.** This is due to the knowledge base cutoff date, which is a physiological feature of LLMs. In fact, their

training dataset will always be “outdated,” not in line with the current trends. This can be overcome by adding non-parametric knowledge with grounding, as previously seen.

- **Lack of executive power.** This means that LLMs by themselves are not empowered to carry out actions. Let’s consider, for example, the well-known ChatGPT: if we ask it to generate a LinkedIn post about productivity tips, we will then need to copy and paste it onto our LinkedIn profile as ChatGPT is not able to do so by itself. That is the reason why we need plug-ins. Plug-ins are LLMs’ connectors toward the external world that serve not only as input sources to extend LLMs’ non-parametric knowledge (for example, to allow a web search) but also as output sources so that the copilot can actually execute actions. For example, with a LinkedIn plug-in, our copilot powered by an LLM will be able not only to generate the post but also to post it online.

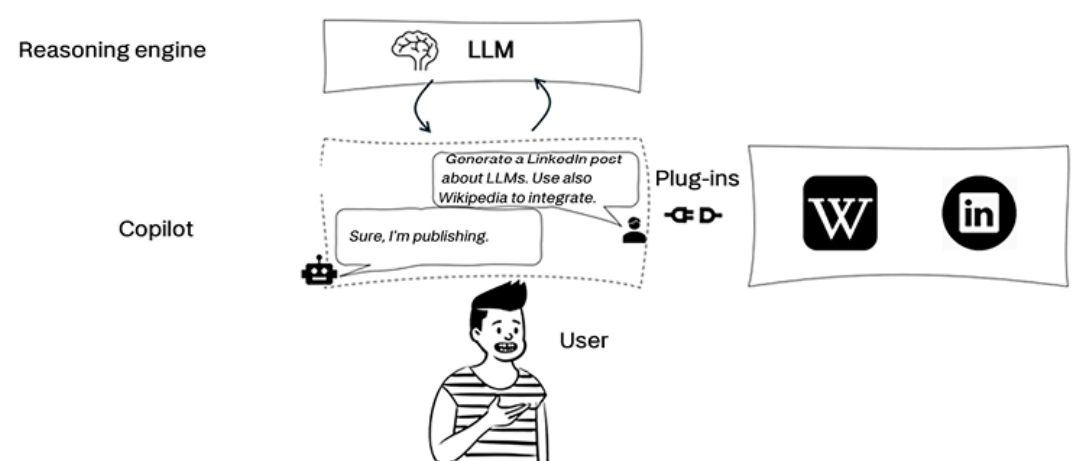


Figure 2.4: Example of Wikipedia and LinkedIn plug-ins

Note that the user’s prompt in natural language is not the only input the model processes. In fact, it is a crucial component of the backend logic of our LLM-powered applications and the set of instructions we provide to the model. This *metaprompt* or system message is the object of a new discipline called **prompt engineering**.

Definition



Prompt engineering is the process of designing and optimizing prompts to LLMs for a wide variety of applications and research topics. Prompts are short pieces of text that are used to guide the LLM's output. Prompt engineering skills help to better understand the capabilities and limitations of LLMs.

Prompt engineering involves selecting the right words, phrases, symbols, and formats that elicit the desired response from the LLM. Prompt engineering also involves using other controls, such as parameters, examples, or data sources, to influence the LLM's behavior. For example, if we want our LLM-powered application to generate responses for a 5-year-old child, we can specify this in a system message similar to “Act as a teacher who explains complex concepts to 5-year-old children.”

In fact, Andrej Karpathy, the previous Director of AI at Tesla, who returned to OpenAI in February 2023, tweeted that “English is the hottest new programming language.”

We will dive deeper into the concept of prompt engineering in *Chapter 4, Prompt Engineering*. In the next section, we are going to focus on the emerging AI orchestrators.

Introducing AI orchestrators to embed LLMs into applications

Earlier in this chapter, we saw that there are two main aspects to consider when incorporating LLMs within applications: a technical aspect and a conceptual aspect. While we can explain the conceptual aspect with the brand-new category of software called Copilot, in this section, we are going to further explore how to technically embed and orchestrate LLMs within our applications.

The main components of AI orchestrators

From one side, the paradigm shift of foundation models implies a great simplification in the domain of AI-powered applications: after producing models, now the trend is consuming models. On the other side, many roadblocks might arise in developing this new kind of AI, since there are LLM-related components that are brand new and have never been managed before within an application life cycle. For example, there might be malicious actors that could try to change the LLM instructions (the system message mentioned earlier) so that the application does not follow the correct instructions. This is an example of a new set of security threats that are typical to LLM-powered applications and need to be addressed with powerful counterattacks or preventive techniques.

The following is an illustration of the main components of such applications:

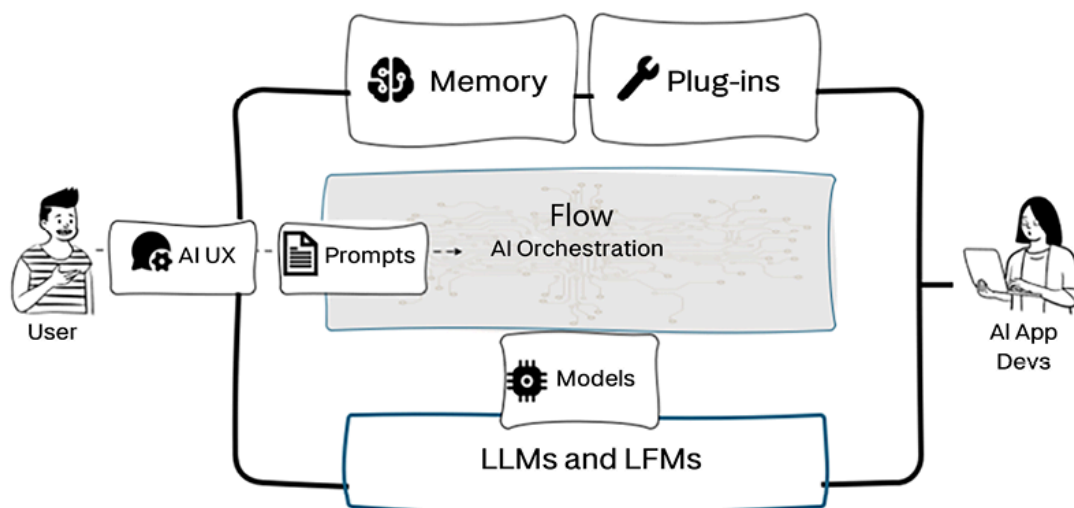


Figure 2.5: High-level architecture of LLM-powered applications

Let's inspect each of these components in detail:

- **Models:** The model is simply the type of LLM we decide to embed in our application. There are two main categories of models:
 - **Proprietary LLMs:** Models that are owned by specific companies or organizations. Examples include GPT-3 and GPT-4, developed by OpenAI, or Bard, developed by Google. As their source code and ar-

chitecture are not available, those models cannot be re-trained from scratch on custom data, yet they can be fine-tuned if needed.

- **Open-source:** Models with code and architecture freely available and distributed, hence they can also be trained from scratch on custom data. Examples include Falcon LLM, developed by Abu Dhabi's **Technology Innovation Institute (TII)**, or LLaMA, developed by Meta.

We will dive deeper into the main set of LLMs available today in *Chapter 3, Choosing an LLM for Your Application*.

- **Memory:** LLM applications commonly use a conversational interface, which requires the ability to refer back to earlier information within the conversation. This is achieved through a “memory” system that allows the application to store and retrieve past interactions. Note that past interactions could also constitute additional non-parametric knowledge to be added to the model. To achieve that, it is important to store all the past conversations – properly embedded – into VectorDB, which is at the core of the application's data.

Definition

VectorDB is a type of database that stores and retrieves information based on vectorized embeddings, the numerical representations that capture the meaning and context of text. By using VectorDB, you can perform semantic search and retrieval based on the similarity of meanings rather than keywords. VectorDB can also help LLMs generate more relevant and coherent text by providing contextual understanding and enriching generation results. Some examples of VectorDBs are Chroma, Elasticsearch, Milvus, Pinecone, Qdrant, Weaviate, and **Facebook AI Similarity Search (FAISS)**.

FAISS, developed by Facebook (now Meta) in 2017, was one of the pioneering vector databases. It was designed for efficient similarity search and clustering of dense vectors and is particularly useful for multimedia documents and dense embeddings. It was initially an internal research project at Facebook. Its primary goal was to better utilize GPUs for identifying similarities related to user preferences. Over time, it evolved into

the fastest available library for similarity search and can handle billion-scale datasets. FAISS has opened up possibilities for recommendation engines and AI-based assistant systems.

- **Plug-ins:** They can be seen as additional modules or components that can be integrated into the LLM to extend its functionality or adapt it to specific tasks and applications. These plug-ins act as add-ons, enhancing the capabilities of the LLM beyond its core language generation or comprehension abilities.

The idea behind plug-ins is to make LLMs more versatile and adaptable, allowing developers and users to customize the behavior of the language model for their specific needs. Plug-ins can be created to perform various tasks, and they can be seamlessly incorporated into the LLM's architecture.

- **Prompts:** This is probably the most interesting and pivotal component of an LLM-powered application. We've already quoted, in the previous section, Andrej Karpathy's affirmation that "English is the hottest new programming language," and you will understand why in the upcoming chapters. Prompts can be defined at two different levels:
 - **"Frontend," or what the user sees:** A "prompt" refers to the input to the model. It is the way the user interacts with the application, asking things in natural language.
 - **"Backend," or what the user does not see:** Natural language is not only the way to interact, as a user, with the frontend; it is also the way we "program" the backend. In fact, on top of the user's prompt, there are many natural language instructions, or meta-prompts, that we give to the model so that it can properly address the user's query. Meta-prompts are meant to instruct the model to act as it is meant to. For example, if we want to limit our application to answer only questions related to the documentation we provided in VectorDB, we will specify the following in our meta-prompts to the model: *"Answer only if the question is related to the provided documentation."*

Finally, we get to the core of the high-level architecture shown in *Figure 2.5*, that is, the **AI orchestrator**. With the AI orchestrator, we refer to lightweight libraries that make it easier to embed and orchestrate LLMs within applications.

As LLMs went viral by the end of 2022, many libraries started arising in the market. In the next sections, we are going to focus on three of them: LangChain, Semantic Kernel, and Haystack.

LangChain

LangChain was launched as an open-source project by Harrison Chase in October 2022. It can be used both in Python and JS/TS. It is a framework for developing applications powered by language models, making them data-aware (with grounding) and agentic – which means they are able to interact with external environments.

Let's take a look at the key components of LangChain:

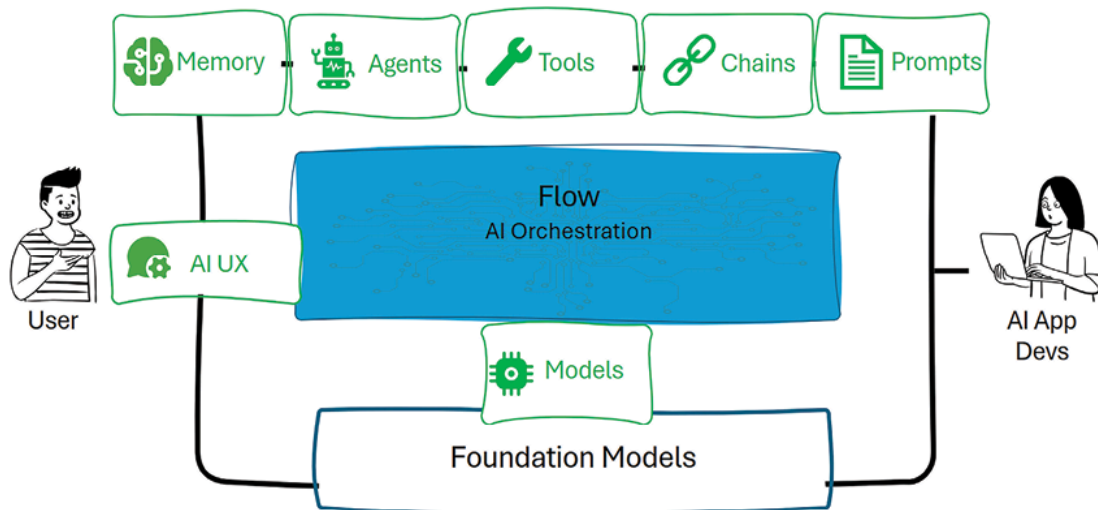



Figure 2.6: LangChain's components

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.



The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Unlock them by scanning the QR code below or visiting

<https://www.packtpub.com/unlock/9781835462317>.



Overall, LangChain has the following core modules:

- **Models:** These are the LLMs or LfMs that will be the engine of the application. LangChain supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the **Hugging Face Hub**.

Definition

Hugging Face is a company and a community that builds and shares state-of-the-art models and tools for natural language processing and other machine learning domains. It developed the Hugging Face Hub, a platform where people can create, discover, and collaborate on machine learning models and LLMs, datasets, and demos. The Hugging Face Hub hosts over 120k models, 20k datasets, and 50k demos in various domains and tasks, such as audio, vision, and language.

Alongside models, LangChain also offers many prompt-related components that make it easier to manage the prompt flow.

- **Data connectors:** These refer to the building blocks needed to retrieve the additional external knowledge (for example, in RAG-based scenarios) we want to provide the model with. Examples of data connectors are document loaders or text embedding models.
- **Memory:** This allows the application to keep references to the user's interactions, in both the short and long term. It is typically based on

vectorized embeddings stored in VectorDB.

- **Chains:** These are predetermined sequences of actions and calls to LLMs that make it easier to build complex applications that require chaining LLMs with each other or with other components. An example of a chain might be: take the user query, chunk it into smaller pieces, embed those chunks, search for similar embeddings in VectorDB, use the top three most similar chunks in VectorDB as context to provide the answer, and generate the answer.
- **Agents:** Agents are entities that drive decision-making within LLM-powered applications. They have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal.

LangChain offers the following benefits:

- LangChain provides modular abstractions for the components we previously mentioned that are necessary to work with language models, such as prompts, memory, and plug-ins.
- Alongside those components, LangChain also offers pre-built **chains**, which are structured concatenations of components. Those chains can be pre-built for specific use cases or be customized.

In *Part 2* of this book, we will go through a series of hands-on applications, all LangChain based. So, starting from *Chapter 5, Embedding LLMs within Your Applications*, we will focus much deeper on LangChain components and overall frameworks.

Haystack

Haystack is a Python-based framework developed by Deepset, a startup founded in 2018 in Berlin by Milos Rusic, Malte Pietsch, and Timo Möller. Deepset provides developers with the tools to build **natural language processing (NLP)**-based applications, and with the introduction of Haystack, they are taking them to the next level.

The following illustration shows the core components of Haystack:

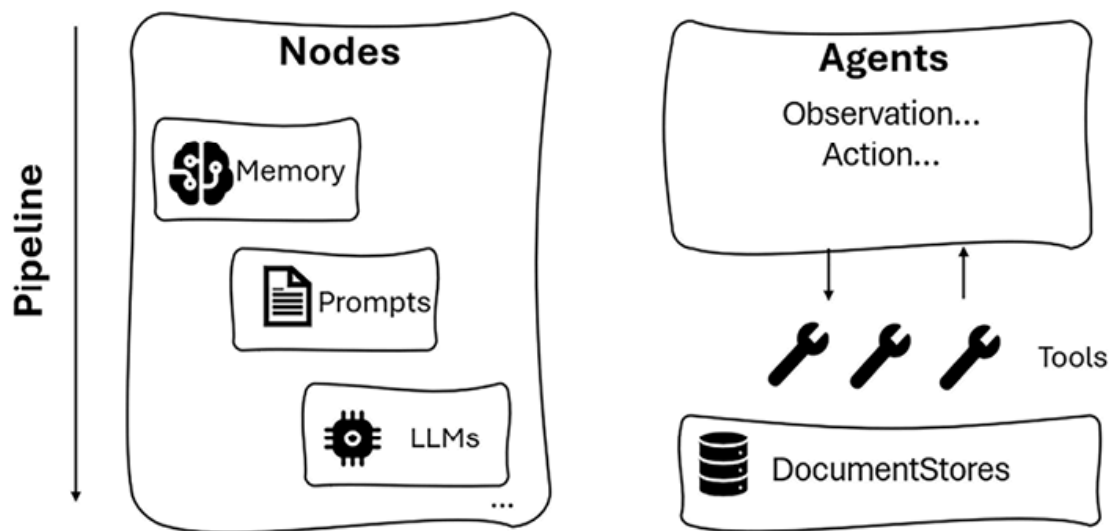


Figure 2.7: Haystack's components

Let's look at these components in detail:

- **Nodes:** These are components that perform a specific task or function, such as a retriever, a reader, a generator, a summarizer, etc. Nodes can be LLMs or other utilities that interact with LLMs or other resources. Among LLMs, Haystack supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the Hugging Face Hub.
- **Pipelines:** These are sequences of calls to nodes that perform natural language tasks or interact with other resources. Pipelines can be querying pipelines or indexing pipelines, depending on whether they perform searches on a set of documents or prepare documents for search. Pipelines are predetermined and hardcoded, meaning that they do not change or adapt based on the user input or the context.
- **Agent:** This is an entity that uses LLMs to generate accurate responses to complex queries. An agent has access to a set of tools, which can be pipelines or nodes, and it can decide which tool to call based on the user input and the context. An agent is dynamic and adaptive, meaning that it can change or adjust its actions based on the situation or the goal.
- **Tools:** There are functions that an agent can call to perform natural language tasks or interact with other resources. Tools can be pipelines or nodes that are available to the agent and they can be grouped into toolkits, which are sets of tools that can accomplish specific objectives.

- **DocumentStores:** These are backends that store and retrieve documents for searches. DocumentStores can be based on different technologies, also including VectorDB (such as FAISS, Milvus, or Elasticsearch).

Some of the benefits offered by Haystack are:

- **Ease of use:** Haystack is user-friendly and straightforward. It's often chosen for lighter tasks and rapid prototypes.
- **Documentation quality:** Haystack's documentation is considered high-quality, aiding developers in building search systems, question-answering, summarization, and conversational AI.
- **End-to-end framework:** Haystack covers the entire LLM project life cycle, from data preprocessing to deployment. It's ideal for large-scale search systems and information retrieval.
- Another nice thing about Haystack is that you can deploy it as a REST API and it can be consumed directly.

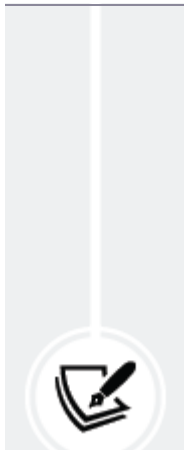
Semantic Kernel

Semantic Kernel is the third open-source SDK we are going to explore in this chapter. It was developed by Microsoft, originally in C# and now also available in Python.

This framework takes its name from the concept of a “kernel,” which, generally speaking, refers to the core or essence of a system. In the context of this framework, a kernel is meant to act as the engine that addresses a user's input by chaining and concatenating a series of components into pipelines, encouraging **function composition**.

Definition

In mathematics, function composition is a way to combine two functions to create a new function. The idea is to use the output of one function as the input to another function, forming a chain of functions. The composition of two functions f and g is denoted as $(f \circ g)$, where the function g is applied first, followed by the function $f \rightarrow (f \circ g)(x) = f(g(x))$.



Function composition in computer science is a powerful concept that allows for the creation of more sophisticated and reusable code by combining smaller functions into larger ones. It enhances modularity and code organization, making programs easier to read and maintain.

The following is an illustration of the anatomy of Semantic Kernel:

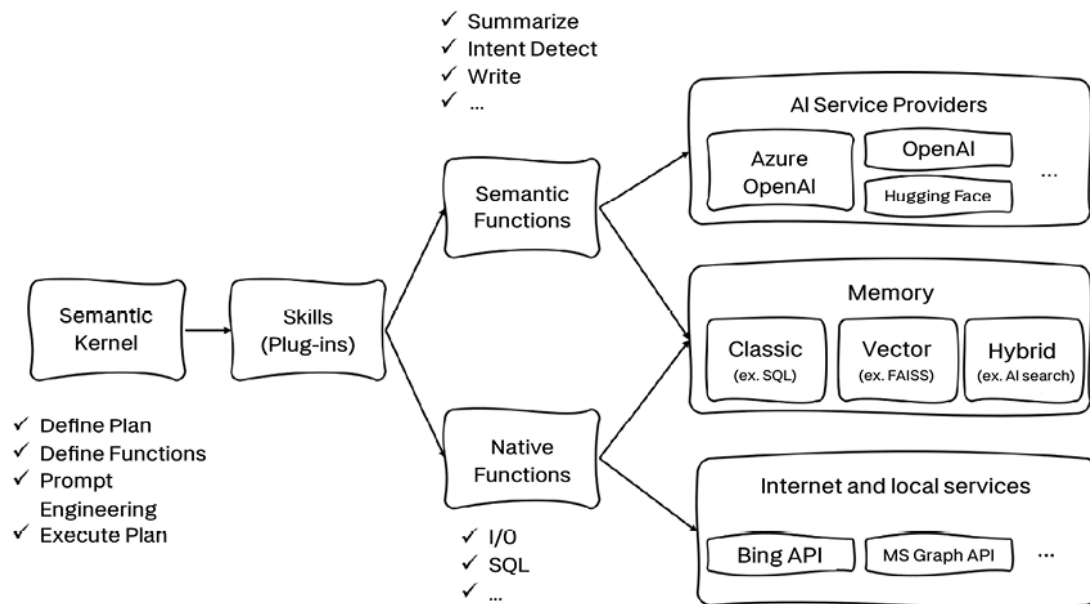


Figure 2.8: Anatomy of Semantic Kernel

Semantic Kernel has the following main components:

- **Models:** These are the LLMs or LFM s that will be the engine of the application. Semantic Kernel supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the Hugging Face Hub.
- **Memory:** It allows the application to keep references to the user's interactions, both in the short and long term. Within the framework of Semantic Kernel, memories can be accessed in three ways:
 - **Key-value pairs:** This consists of saving environment variables that store simple information, such as names or dates.
 - **Local storage:** This consists of saving information to a file that can be retrieved by its filename, such as a CSV or JSON file.

- **Semantic memory search:** This is similar to LangChain's and Haystack's memory, as it uses embeddings to represent and search for text information based on its meaning.
- **Functions:** Functions can be seen as skills that mix LLM prompts and code, with the goal of making users' asks interpretable and actionable. There are two types of functions:
 - **Semantic functions:** These are a type of templated prompt, which is a natural language query that specifies the input and output format for the LLM, also incorporating prompt configuration, which sets the parameters for the LLM.
 - **Native functions:** These refer to the native computer code that can route the intent captured by the semantic function and perform the related task.

To make an example, a semantic function could ask the LLM to write a short paragraph about AI, while a native function could actually post it on social media like LinkedIn.

- **Plug-ins:** These are connectors toward external sources or systems that are meant to provide additional information or the ability to perform autonomous actions. Semantic Kernel offers out-of-the-box plug-ins, such as the Microsoft Graph connector kit, but you can build a custom plug-in by leveraging functions (both native and semantic, or a mix of the two).
- **Planner:** As LLMs can be seen as reasoning engines, they can also be leveraged to auto-create chains or pipelines to address new users' needs. This goal is achieved with a planner, which is a function that takes as input a user's task and produces the set of actions, plug-ins, and functions needed to achieve the goal.

Some benefits of Semantic Kernel are:

- **Lightweight and C# support:** Semantic Kernel is more lightweight and includes C# support. It's a great choice for C# developers or those using the .NET framework.
- **Wide range of use cases:** Semantic Kernel is versatile, supporting various LLM-related tasks.

- **Industry-led:** Semantic Kernel was developed by Microsoft, and it is the framework the company used to build its own copilots. Hence, it is mainly driven by industry needs and asks, making it a solid tool for enterprise-scale applications.

How to choose a framework

Overall, the three frameworks offer, more or less, similar core components, sometimes called by a different taxonomy, yet covering all the blocks illustrated within the concept of the copilot system. So, a natural question might be: “Which one should I use to build my LLM-powered application?” Well, there is no right or wrong answer! All three are extremely valid. However, there are some features that might be more relevant for specific use cases or developers’ preferences. The following are some criteria you might want to consider:

- **The programming language you are comfortable with or prefer to use:** Different frameworks may support different programming languages or have different levels of compatibility or integration with them. For example, Semantic Kernel supports C#, Python, and Java, while LangChain and Haystack are mainly based on Python (even though LangChain also introduced JS/TS support). You may want to choose a framework that matches your existing skills or preferences, or that allows you to use the language that is most suitable for your application domain or environment.
- **The type and complexity of the natural language tasks you want to perform or support:** Different frameworks may have different capabilities or features for handling various natural language tasks, such as summarization, generation, translation, reasoning, etc. For example, LangChain and Haystack provide utilities and components for orchestrating and executing natural language tasks, while Semantic Kernel allows you to use natural language semantic functions to invoke LLMs and services. You may want to choose a framework that offers the functionality and flexibility you need or want for your application goals or scenarios.
- **The level of customization and control you need or want over the LLMs and their parameters or options:** Different frameworks may

have different ways of accessing, configuring, and fine-tuning the LLMs and their parameters or options, such as model selection, prompt design, inference speed, output format, etc. For example, Semantic Kernel provides connectors that make it easy to add memories and models to your AI app, while LangChain and Haystack allow you to plug in different components for the document store, retriever, reader, generator, summarizer, and evaluator. You may want to choose a framework that gives you the level of customization and control you need or want over the LLMs and their parameters or options.

- **The availability and quality of the documentation, tutorials, examples, and community support for the framework:** Different frameworks may have different levels of documentation, tutorials, examples, and community support that can help you learn, use, and troubleshoot the framework. For example, Semantic Kernel has a website with documentation, tutorials, examples, and a Discord community; LangChain has a GitHub repository with documentation, examples, and issues; Haystack has a website with documentation, tutorials, demos, blog posts, and a Slack community. You may want to choose a framework that has the availability and quality of documentation, tutorials, examples, and community support that can help you get started and solve problems with the framework.

Let's briefly summarize the differences between these orchestrators:

Feature	LangChain	Haystack	Semantic Kernel
LLM support	Proprietary and open-source	Proprietary and open source	Proprietary and open source
Supported languages	Python and JS/TS	Python	C#, Java, and Python
Process orchestration	Chains	Pipelines of nodes	Pipelines of functions

Deployment	No REST API	REST API	No REST API
Feature	LangChain	Haystack	Semantic Kernel

Table 2.1: Comparisons among the three AI orchestrators

Overall, all three frameworks offer a wide range of tools and integrations to build your LLM-powered applications, and a wise approach could be to use the one that is most in line with your current skills or the company's overall approach.

Summary

In this chapter, we delved into the new way of developing applications that LLMs have been paving, as we introduced the concept of the copilot and discussed the emergence of new AI orchestrators. Among those, we focused on three projects – LangChain, Haystack, and Semantic Kernel – and we examined their features, main components, and some criteria to decide which one to pick.

Once we have decided on the AI orchestrator, another pivotal step is to decide which LLM(s) we want to embed into our applications. In *Chapter 3, Choosing an LLM for Your Application*, we are going to see the most prominent LLMs on the market today – both proprietary and open-source – and understand some decision criteria to pick the proper models with respect to the application use cases.

References

- LangChain repository: <https://github.com/langchain-ai/langchain>
- Semantic Kernel documentation: <https://learn.microsoft.com/en-us/semantic-kernel/get-started/supported-languages>
- Copilot stack: <https://build.microsoft.com/en-US/sessions/bb8f9d99-0c47-404f-8212-a85fffd3a59d?source=/speakers/ef864919-5fd1-4215-b611-61035a19db6b>
- The Copilot system: <https://www.youtube.com/watch?v=E5g20qmeKpg>

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/11m>

