AI Agents:

From Language Generation to Task Automation

Louis Jourdain

"The future (of work) is agentic,"[1] or so the refrain goes. Terms like *AI agent*, *Agentic AI*, and their many derivatives have quickly entered both the vocabulary of AI evangelists and the discourse of non-technical audiences. Compared to acronyms such as *RAG* (Retrieval-Augmented Generation) or *MCP* (Model Context Protocol, see Chapter 3), the word *agent* is undeniably more accessible and evocative.

The great fantasy of computing has always been to automate the tasks that humans have no desire to perform. The democratization of large language models (LLMs), and their ability to follow instructions in few-shot settings, has made it possible to automate an ever-growing range of tasks—summarization, targeted information extraction, the drafting of administrative documents—without the need to design a dedicated software system.

Nevertheless, as we saw in the previous chapter, LLMs are nothing more than AI models that predict the next word. They come with significant limitations, most notably their lack of access to specific data sources, which drastically restricts the kinds of real-world tasks they can automate.

In response, much research has gone into extending the reasoning capabilities of these models, or equipping them with the ability to use external tools (*tool use*). Starting as early as 2023, researchers began to explore ways to integrate LLMs into systems capable of handling more complex tasks—such as resolving GitHub issues or tackling challenging QA problems—autonomously. Achieving this required less work on the models themselves and more on the surrounding software engineering: building frameworks to orchestrate calls to LLMs, though fine-tuning was often applied to improve performance.

These systems were given the name *agents*. In 2023, this led to a heterogeneous wave of research, before a clearer definition and emerging best practices began to take shape. The year 2024 marked a phase of stabilization in the definition and architecture of agents, and all indications suggest that 2025 will be the year when they begin moving into production—when the industrialization of AI agents truly begins.

Behind this latest wave of hype, however, lies more than a marketing fad. What began as a research direction exploring the emergent capabilities of large language models has rapidly scaled into a

1 https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/the-future-of-work-is-agentic

broader movement, one that promises consequences far beyond the lab. Like the advent of the Internet, and more recently the rise of generative AI in 2023, the “agentic turn” exemplifies those rare moments when a narrowly defined scientific advance expands into a global economic and cultural phenomenon—sometimes opportunity, sometimes bubble, often both.[2].

However, this new “revolution” carries distinctive features that set it apart from earlier ones. Notably, the surge of interest in *agentic AI* began while the previous wave—generative AI—was still in the process of mainstream adoption. In practice, building and successfully deploying agents presupposes the effective integration of simpler GenAI tools: without the hammer, the carpenter cannot yet build the house.

Today, nearly every tech company—or any business eager to brand itself as “AI-first”—announces ambitions to implement AI agents or even entire “agentic platforms.” Yet in reality, only a minority possess the technical and infrastructural capacity to do so. At the entry level, any competent Python developer can stitch together a functional agent using one of the many libraries now available. But scaling these systems into production requires much more: robust infrastructure, dedicated data teams capable of adapting agents to specific use cases, and storage systems that allow the observation and governance of agents across their lifecycle.

This explains why most of the major players—Microsoft, Google, IBM, Amazon, Salesforce, Dataiku, and others—are promoting their own *agentic ecosystems*. For now, these remain largely out of reach for smaller startups, who lack the necessary scale of resources. That does not mean, however, that they cannot build businesses *on top of* agentic AI; it simply means they must rely on the infrastructures of others.

The crucial shift, then, is no longer about the models themselves. New generations of models will continue to appear, cheaper and more powerful, driven by healthy competition among leading actors. The real inflection point lies in how these models are leveraged: in moving from generating human-like text at marginal cost, to orchestrating entire sequences of tasks—turning outputs into actions, and interactions into value.

The promises of AI agents span a wide range of industries and functions. Companies dream of systems that can generate content—whether text, code, or multimedia—drawing on codified knowledge to produce material on demand. Others imagine agents handling customer interactions in call centers or support desks, automating responses while adapting them to the client’s history and needs. In sectors with large supply bases, procurement managers anticipate agents capable of conducting frequent price negotiations where small efficiencies compound into significant savings. And in contexts where large groups such as sales teams or maintenance crews must coordinate and track tasks, the notion of agentic AI as a silent orchestrator seems particularly attractive. These scenarios illustrate why businesses see agents as the “next frontier”: they touch domains where incremental productivity gains scale massively across entire organizations.

2 We can also interpret the Agentic Boom as the chronological and scientifical expansion of the Generative AI market.

From the academic side, we are witnessing what could be described as a “diarrhea of papers.” Arxiv is overflowing with explorations of what agentic AI might achieve. Some studies investigate how agents could build crypto portfolios or transform advertising markets. Others are more imaginative. If we list some intriguing papers I came accross :

- *Building crypto portfolios with agentic AI*
- *AI Agents and the Attention Lemons Problem in Two-Sided Ad Markets*
- *AI-Driven Generation of Old English: A Framework for Low-Resource Languages*
- *HAMLET: Hyperadaptive Agent-based Modeling for Live Embodied Theatrics*
- *TacticCraft: Natural Language-Driven Tactical Adaptation for StarCraft II*
- *Synthetic Socratic Debates: Examining Persona Effects on Moral Decision and Persuasion Dynamics*
- *AutoGen Driven Multi-Agent Framework for Iterative Crime Data Analysis and Prediction*

The breadth of imagination is undeniable. Yet most of these papers remain proof-of-concept rather than working technology. They are generally weaker than what industrial labs quietly develop, and we are still not seeing convincing examples of complex AI agents that consistently succeed with error rates better than human benchmarks. The research ecosystem is vibrant but fragile, long on vision and short on robustness.

There are at least two reasons accounting for this fragility. First, the field is split between engineers and researchers. Academia often overproduces theoretical prototypes with little follow-through, while industrial teams build systems but rarely share their findings. The tension is visible even in the public sphere, crystallized in clashes such as Yann LeCun versus Elon Musk on the future of AI. Second, the most advanced agentic systems are deliberately kept out of sight. Companies view agentic AI as a strategic advantage, and their researchers are often forbidden to publish. What we see in public, then, is either exploratory research or corporate marketing.

We are at a moment where the dream of agentic AI is racing ahead of reality. Everyone talks about agents, but the meanings diverge, the use cases are scattered, and the research is messy. The hype is enormous, but robust demonstrations are rare. It is precisely in this ambiguity that a structured perspective becomes necessary. The aim of this twelve-hour course is to cut through the noise: to clarify what AI agents are, what they are not (yet), and how to situate them on a continuum—from simple language assistants to systems approaching real autonomy.

In this first session, our aim will be to explore both the economic and societal potential of these emerging agentic technologies, while working toward a clearer understanding of what AI agents actually are in comparison to other forms of generative AI. We will then outline the essential components that a system must include in order to legitimately merit the label *agentic*. In the later parts of the course, we will examine these components in detail—considering their role, their implementation, and the engineering principles behind them—so that by the end you will have a solid grasp of what AI agents are and what they can truly achieve.

I) Agents in society : multiple approaches af this turning point

a) Economic opportunities

The rise of AI agents is not just a technical challenge—it is also shaping up to be one of the major economic opportunities of the decade. Market analysts estimate that by 2028, agentic AI could generate up to $450 billion in value, through revenue growth and cost savings across industries. Yet the path toward this opportunity is neither straightforward nor risk-free.

- Adoption is accelerating, but maturity remains low. As of today, only 2% of organizations report having deployed AI agents at scale. Another 12% have reached partial deployment, while 23% are still at the pilot stage. The vast majority (61%) remain in the exploration phase. Despite the enthusiasm, fewer than one in five organizations have the data and infrastructure maturity required for robust agentic deployment.
- Promises and pitfalls. Gartner predicts that by 2027, AI agents will augment or automate up to 50% of business decisions. At the same time, it warns that 40% of AI agent projects will be canceled—derailed by high costs, unclear business value, or the inability to control agents reliably. Trust in fully autonomous systems is actually declining, with confidence dropping from 43% to 27% in just one year.
- A shift toward human–AI collaboration. Rather than replacing humans altogether, the trajectory points toward blended teams. By 2028, nearly 40% of organizations are expected to integrate AI agents as “team members” within human workflows. This hybrid model—where humans handle judgment and accountability, while agents take care of routine or complex automation—may become the dominant driver of productivity and innovation.

In short, the “agentic revolution” is a double-edged sword: a multi-hundred-billion-dollar opportunity for those able to deploy agents effectively, but also a landscape of high failure rates and structural challenges for those who lack the necessary foundations.[3]

b) On a scheduled social impact

The “third industrial revolution” was often described as the fantasy of automating *white-collar jobs*. With generative AI, that fantasy has already begun to materialize. From Duolingo’s experiments with AI tutors to widespread layoffs at Microsoft and early shifts in the U.S. employment market, signs are emerging that entry-level knowledge work is especially vulnerable.

Dario Amodei, CEO of Anthropic, has even predicted that AI could wipe out half of entry-level white-collar jobs and drive 10–20% unemployment within 1–5 years. While such forecasts may be controversial, they underscore a growing concern: the impact of AI is not limited to repetitive manual labor, but extends deeply into cognitive work.

3 https://www.capgemini.com/insights/research-library/ai-agents/

Microsoft recently published a list of the 40 professions most affected by generative AI, based on a study of 200,000 real-world interactions with Copilot[4]. These jobs cluster around tasks such as:

- Writing and documentation
- Information retrieval and knowledge management
- Routine decision-making

The study measured both the *frequency* of such tasks in a profession and the *degree of success* with which AI could perform them, leading to a "coverage" score of how much of a given job could realistically be automated.

The results were striking: interpreters, translators, historians, passenger attendants, and sales representatives emerged among the most at-risk occupations. In the technology sector, data scientists and mathematicians also ranked high—jobs once thought to be shielded by technical expertise.

Yet there is an important limitation. These findings are based on *human-in-the-loop interactions*, where workers delegated portions of their routine to AI tools. Generative AI here acts as an assistant, not as an autonomous replacement. There is no orchestration of tasks, no ability to manage workflows or handle dependencies.

In other words: GenAI alone is not the direct threat. The real disruption may come with the arrival of agentic AI, where orchestration and autonomy enable systems not only to *assist* but to *act*—blurring the line between augmentation and replacement.

c) Philosophical implications

The emergence of AI Agents is currently challenging moral and philosophical view

The very term *agent* carries philosophical weight. As we will clarify later, AI systems only appear to act according to their own goals or reasoning—the nuance is crucial. Yet this appearance is enough to reignite long-standing debates on *autonomy, responsibility, and moral agency*.

Consider two scenarios:

- A forensic AI agent draws a flawed conclusion, and an innocent person is charged. Who bears responsibility?
- An AI stock analyst agent makes a poor recommendation, resulting in massive financial loss. Is the company to blame? The system's creators? Or—troublingly—*no one at all*?

These reflections echo earlier controversies surrounding prescriptive algorithms and classical machine learning, but the notion of *agents* heightens the stakes.

4 https://arxiv.org/pdf/2507.07935

Autonomy: Engineering vs. Moral

An *agent* implies a system that can act independently. In engineering terms, autonomy refers to the degree to which a system can perform start-to-finish tasks without direct human control. The more it can do on its own, the more autonomous it is.

But this must not be confused with moral autonomy—the capacity to set one's own goals and govern one's actions according to ethical principles. AI agents do not define their own purposes; their autonomy is limited to executing *assigned* objectives within constraints. The distinction between *autonomy-as-independence* and *autonomy-as-moral self-governance* is essential to avoid anthropomorphizing these systems.

The Responsibility Gap[5]

The rise of agentic AI reopens the troubling "responsibility gap." When an AI-driven action leads to harm, traditional accountability mechanisms falter. Humans involved—designers, deployers, or users—may all plausibly deny direct responsibility: *"It was the AI's fault, not mine."*

If left unresolved, this gap risks leaving victims without justice or recourse. Philosophers and ethicists have begun to propose responses:

- Preventative restrictions, limiting AI autonomy in sensitive areas so that humans remain directly responsible for final decisions.
- Shared accountability frameworks, where responsibility is distributed across designers, operators, and organizations.
- Legal adaptations, introducing new liability categories for AI-driven actions.

At its core, the debate is not only technical but moral: how much independence can we grant to artificial systems without undermining the foundations of responsibility, justice, and trust in society?

The Messianic Stance: AI Agents as a Technological Promise

Beyond their technical architecture, AI agents are increasingly framed in philosophical or even evangelical terms. In parts of the tech industry, agents are not simply tools, but a paradigm shift—promised as systems that will redefine the way humans live, work, and organize society.

Just as the Internet once promised to connect humanity, and smartphones to augment our daily lives, AI agents are portrayed as the next universal layer of infrastructure. In this vision, agents will become ubiquitous companions, embedded in education (personalized tutors), health (continuous medical advisors), finance (automated wealth managers), and governance (policy simulators or civic assistants). This framing often borders on messianic: agents are not just improving workflows, they are redefining human existence through constant delegation and augmentation.

Proponents often present autonomy not as a technical feature, but as a liberation narrative: agents will "free" humans from drudgery by handling complex, multi-step tasks on their own. The

5 https://medium.com/@luan.home/the-philosophy-of-agentic-ai-agency-autonomy-and-moral-responsibility-in-artificial-intelligence-a26a8f622a60

marketing around “copilots,” “companions,” and “digital colleagues” resonates with a quasi-spiritual idea of transcendence through delegation—humans unburdened, able to focus on creativity, relationships, or “higher” goals.

However, framing AI agents as humanity’s next redeemers can obscure practical realities. Unlike generative AI systems that mainly assist in bounded tasks, agents introduce orchestration, autonomy, and decision-making—raising control, accountability, and safety issues. The responsibility gap looms larger when the technology is marketed as inevitable salvation. By presenting agents as destiny rather than choice, evangelists risk discouraging sober debate about governance, limits, and failure modes.

=> The narratives surrounding AI agents often race ahead of the technology itself. This creates what some call *agentwashing*: companies rebranding existing chatbots or automation tools as “agents” to ride the hype cycle. Just as *greenwashing* inflates sustainability claims, *agentwashing* risks diluting the meaning of the term and raising unrealistic expectations.

At the same time, the reliability gap remains stark. Many agent prototypes can chain tasks impressively in demos, but they struggle in production—failing silently, making brittle decisions, or requiring constant human oversight. In short: the dream is moving faster than deployment realities.

This is the right moment to step back and define what we mean by *AI agent*. Beyond the hype and the promises, we need to situate agents on a continuum—from simple LLM-driven assistants to systems capable of sustained autonomy and orchestration.

II) AI Agents: a contemporary definition

With a touch of humor, one can see AI agents as an intermediate step between civil servants (the HR sense of the word “agent”) and AGI (Artificial General Intelligence, that is to say an AI system endowed with cognitive capacities equivalent or superior to those of a human being across the board). The purpose of a computational agent is to carry out a specific task with the same dexterity as a human agent would, in order to relieve them of this mission. If the mission itself is relatively clear, the technical definition is not at all straightforward and took more than a year to emerge.

a) Software agents: historical background

The scientific community eventually settled on the term “AI agent” to designate systems that rely on LLMs to automate complex tasks. Such automation goes beyond the simple creation of a dedicated prompt and generally requires additional processing both upstream and downstream of the call or calls to the LLM. In the end this is a reactivation of a term already introduced in the 1990s, without any single author to whom it can be attributed.

Semantic history of the word “agent”:
The word “agent” simply designates a system that does something. It derives from the present participle of the Latin verb *agere* (agens, agentis) and means “one who acts, produces, puts into effect”. More precisely, the term comes from medieval scholastic philosophy (in physics and metaphysics) where it referred to a force or element capable of modifying another element, that is, one that exerts an action as opposed to what undergoes it (hence also the grammatical distinction between agent and patient). Later the word took on its administrative and political sense, referring to a person entrusted with a mission, and eventually became synonymous with “official” or “civil servant”.

In computer science, the word acquired a precise meaning: software dispatched across a network to perform a task on behalf of the user and without their intervention. This notion gained traction with the semantic web, where programs were set up to carry out searches on behalf of humans (crawlers, bots). A so-called “mobile” agent could move from one site to another, access data or resources, and do so while retaining not only its code but also its execution state, deciding autonomously on its next action. Examples include the system agents of operating systems that run in the background, indexing robots, or the user agent that manages the interface between the browser and the web. In this sense an agent in computer science is a program whose architecture allows it to manage its own execution without supervision. However, the term is closely related to several other important notions in computing.

Automata

Our computer programs are essentially scripts designed to execute a task, and they are not very adaptable. In most cases, several scenarios are anticipated in advance—depending on the program’s

interaction with the user or the environment—through routing systems or enumerations of possible actions. In fact, a complex program can often be represented as a graph where each node corresponds to an execution state of the program and each edge corresponds to the execution of certain actions that modify that state. Some of the libraries implementing AI agents, as we shall see later, are based on similar intuitions. Yet this way of conceiving and representing a computational system designed to automate a task is itself inherited from one of the foundational concepts of computer science: automata.

An *automaton* is a device that autonomously reproduces a sequence of predetermined actions without human intervention.[6]

Finite automata, or more generally state machines, are mathematical models used to describe the behavior of discrete systems. In such models, a system can exist in a finite number of states and transition from one to another in response to events or inputs. A program can therefore be seen as a deterministic automaton, where each state represents a specific configuration of the program (for example, a step in a form, a screen in an application, or a particular logical condition), and each transition corresponds to an action, a user input, or a satisfied logical condition. This paradigm underlies the development of state machines in embedded systems, video games, and certain software architectures.

However, this conception, powerful as it may be, quickly shows its limits as soon as the system becomes complex or must operate in dynamic and partially unknown environments. In such cases it becomes difficult, if not impossible, to predetermine all possible transitions or anticipate every scenario. This is precisely where intelligent agents come into play. An AI agent does not simply execute a fixed script: it observes, evaluates, decides, and acts in an environment that may be uncertain. In other words, if we were to compare a classical program to an automaton, an AI agent would be an entity capable of dynamically modifying its own automaton structure, of adding or removing states, or even of changing its transition strategy during execution. This paradigm shift—from a fixed system to an adaptive system—is what fundamentally distinguishes intelligent agents from traditional scripts, even if it remains their natural evolution. What is added is the capacity for reasoning and self-planning.

Programs and Multi-Agent Systems

Another notion from computer science to which AI agents are closely related is that of multi-agent systems. A multi-agent system is a system composed of a set of agents (in the broad sense, this may designate either a program or a user) acting within a given environment with which they interact according to defined rules. Each agent is at least partially autonomous, and there is no centralized control of the global system (in contrast to “hive mind” systems in which agents are tightly coordinated). Each agent interacts with the environment[7] and decides on its own action. For example, in a multi-agent system designed to manage the functioning of a greenhouse or a vivarium, the thermostat is an agent which can measure the temperature of the greenhouse through a thermometer sensor (observation of the environment) and, according to rules or instructions, decide to activate the temperature regulation mechanisms (action on the environment).

6 Definition from wikipédia. Cf etymologyhttps://bailly.app/automatos which is ironic, since automaton originally means « able think by itself » while automata in CS are predetermine, without thee notion of thought or reasoning

7 https://en.wikipedia.org/wiki/Intelligent_agent

Multi-agent systems have often been used to model the natural evolution of human societies or animal populations, with applications sometimes extending to the social sciences, such as sociology, and more prominently to economics. They are also used in certain industries, notably in video games, for the management of Non-Player Characters (NPCs).

Designing a multi-agent system today involves several essential roles for AI. There is the need for AI to manage the decision-making of individual agents. There is also the need for AI, sometimes in distributed form, to handle the distribution of tasks among agents. In other words, agents must be able to plan and orchestrate the execution of tasks collectively. This is as much a matter of software engineering as of artificial intelligence, since it requires conceiving an effective and well-distributed architecture capable of running all these agents simultaneously.

Managing agents indeed raises important technical challenges that must be addressed. How can multiple agents act at the same time on the same shared environment, without conflict over shared resources, dynamic choices, or scheduling? How should one handle the prioritization of agents' actions relative to one another? How can their programming be dynamically adapted to changes in the environment, so that not only do environmental changes alter the reactions of agents according to their protocol, but perhaps also modify the protocol of the agents themselves?

The implementation of AI agents therefore brings back into focus a set of software engineering problems belonging to a domain that had become somewhat niche and had lost visibility. The definition of AI agents as it has crystallized today is directly the heir of different branches of computer science—whether or not practitioners are aware of this—and this heritage strongly influences the way agents are implemented, especially when it comes to workflow management and multi-agent architectures.

b) The different degrees of agentivity: a gradual definition

Despite these necessary historical reminders, we have still not reached a fully satisfactory definition of what an AI agent actually is. This definition has been difficult to establish because researchers have explored directions that were sometimes very different from one another. Yet over time a preliminary consensus has begun to emerge. If we compare a few existing definitions:

- "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." (Stuart Russell and Peter Norvig, in their foundational textbook written before the era of generative AI)
- "An agent is an AI model capable of reasoning, planning, and interacting with its environment." (Hugging Face)
- "An artificial intelligence (AI) agent refers to a system or program that is capable of autonomously performing tasks on behalf of a user or another system by designing its workflow and utilizing available tools." (IBM)
- "A generative AI system is considered an agent when it goes beyond creating output based on a prompt and begins to act on information to achieve a desired outcome." (Babak Hodjat, CTO of AI at Cognizant)

All these definitions, although they correctly describe what an AI agent is, remain relatively vague. The agent is essentially defined by its capacities—a constructivist kind of definition. It can read its environment, reason about its observations, and modify that environment. Implementing agents therefore means implementing new capabilities[8]. At the same time, the agent is also defined in negative terms: it is what goes beyond the generative capabilities of an LLM.

Arriving at a single, simple definition does not seem possible. From what degree of autonomy do we begin to speak of an agent? Is providing contextual information in a prompt not already a way of making an LLM interact with its environment? Since the capabilities of LLMs continue to evolve, how does the boundary between a simple LLM and an agent itself evolve? The key point here is that nothing in this field is extremely stable or fully satisfactory at the moment.

Rather than seeking a strict frontier between the LLM and the agent, it is more fruitful to consider that computational systems integrating LLM calls can be situated along different degrees of agentivity.[9]

Starting from the simplest case, one can describe a gradual scale of agentivity, moving from systems that are little more than LLM wrappers to systems that exhibit more autonomy and orchestration.

Agency Level	Description	How that's called	Example Pattern
☆☆☆	LLM output has no impact on program flow	Simple Processor	`process_llm_output(llm_response)`
★☆☆	LLM output determines an if/else switch	Router	`if llm_decision(): path_a() else: path_b()`
★★☆	LLM output determines function execution	Tool Caller	`run_function(llm_chosen_tool, llm_chosen_args)`
★★★	LLM output controls iteration and program continuation	Multi-step Agent	`while llm_should_continue(): execute_next_step()`
★★★	One agentic workflow can start another agentic workflow	Multi-Agent	`if llm_trigger(): execute_agent()`

At the lowest level, the LLM is nothing more than a simple processor. It receives an input and produces a textual output, with no impact on program flow. The model is simply a transformation function.

* A first step upward is to use the LLM as a dynamic router. Instead of relying on explicit if/else branching inside the program, one delegates the choice of branch to the LLM. This makes it possible to build more complex systems, adaptable to heterogeneous data. For example, the LLM may analyze the content of a document, check whether certain conditions are satisfied (the presence of a date of birth, the candidate being under 26 years old, and so on), and choose the next stage of processing accordingly. In this case, the LLM does not really interact with its environment, but it already demonstrates the ability to make dynamic decisions.

8 https://arxiv.org/pdf/2308.11432
9 https://huggingface.co/docs/smolagents/en/conceptual_guides/intro_agents

* A further step is reached when the LLM is allowed to determine the next function to call, given the current state of processing. Here, the system begins to resemble what is usually called a *tool-using agent*. The agent shows a certain degree of autonomy, but remains limited to the actions it is authorized to carry out.

* The next stage corresponds to multi-step agents. In this case, the LLM is capable of reasoning about a general task, deciding which actions to perform next, and iteratively continuing until a satisfactory termination condition is reached. The model therefore alternates between reasoning, planning, and delegating actions to external functions, and then reintegrating their results into its own reasoning process. At this point one is exploiting the reasoning and planning capacities of LLMs to their full extent in order to execute a task.

* Finally, the highest level is reached when several agents interact, exchanging information and coordinating to carry out a task. For example, in order to automate the screening of CVs, one agent could be tasked with reading the CVs and extracting skills, another agent with comparing these skills to a job description, and a “manager” agent with orchestrating the process as a whole. In such systems, orchestration functions are added to the use of LLMs to plan and coordinate a workflow.

This gradual view highlights the distinction between a monotonic workflow—in which an LLM, sometimes equipped with tools, is used in a fixed sequence of steps—and an actual agent. We speak of a conversational agent when the input received and the actions taken are organized dynamically rather than predetermined, opening the door to more flexible and adaptive forms of automation.

If we return to the initial definition proposed by Russell and Norvig, one can schematize an agent as a system whose core is an augmented LLM (see section II), endowed with capacities that allow it to perceive and act upon its environment.

In the absence of a strict definition, it is useful to think of agentivity as a spectrum, measured according to the autonomy the system has in predicting the next actions it will carry out. This is also the main distinction when user input is taken into account. A conversational agent is truly an agent only if the system dynamically chooses what sequence of actions to perform after each conversational turn.

The boundaries here are porous, since what we are really doing is constructing systems out of existing building blocks (functions, LLMs, and so on). Workflows and agents are composable structures: an agent can itself become a component of another agent. Reusability is therefore an important factor. Two aspects matter most in determining where a system falls on the spectrum of agentivity, and in how it may evolve: its **components** and its **architecture**.

c) As many definitions as implementations

If agents are all supposed to achieve the same thing—that is, managing the interface between LLMs and an environment in order to automate complex tasks—the strategies used to reach this goal can be quite different. How should intermediate states of processing be stored? How should communication between components be designed? How should the components of an agent be organized? Different ways of addressing these technical questions have given rise to different implementations of agentic systems, each with their own characteristics.

Comparative Table of LLM Agent Frameworks

Framework	Key Characteristics	Advantages	Drawbacks
LangChain / LangGraph	- Python, modular, chain-oriented - LangGraph: graph-based orchestrations - Tools, memory, planning - Multi-LLM compatible	+ Highly flexible + Huge ecosystem (LangSmith, many tools) + Multi-agent possible with LangGraph + Very popular	– Complex to learn – Verbose – Risk of overhead for small projects
Semantic Kernel	- C# and Python - Plugin architecture (skills) - Integrated planner - Strongly oriented toward productivity and enterprise apps	+ Ideal for .NET ecosystem + Microsoft support + Enterprise extensibility	– Less rich than LangChain for RAG or advanced agents – Steeper learning curve for planner
CrewAI	- Python + No-code studio - Role/task orchestration - Sequential orchestration - 700+ API/tool integrations	+ Very productive for MVPs or startups + Integrated UI/monitoring + Good code/no-code balance	– Less flexible for free orchestration – Heavy for simple projects – Telemetry enabled by default
smol-agents (HF)	- Python - Very lightweight (1 script = 1 agent) - Calls Python functions from generated code - HF Transformers integration	+ Extremely simple + Highly customizable + Few dependencies + Great for local or experimental agents	– No native memory – No multi-agent coordination – Less mature – Requires user-provided monitoring and security
OpenAI Swarm	- Python - Agents + handoffs - Stateless execution - Client-side / edge-first design	+ Minimalistic and educational + Highly customizable + Private and secure (no persistence)	– Not suited for production – No long-term memory – Requires supervision and strict error handling
Autogen (Microsoft)	- Python - Asynchronous inter-agent communication - Configurable roles - Native multi-agent support + human-in-the-loop	+ Very powerful for coordination + Open-source with Microsoft support + Scalable async workflows	– High complexity – Limited UI/monitoring unless integrated manually
Google ADK (Agent Development Kit)	- Python/Go - Strong focus on reproducibility and deployment - Artifact system for persistent outputs - Cloud-ready integrations (Vertex AI, GCP services) - Modular plugin system	+ Clean artifact system (manages files, graphs, logs) + Cloud-native orchestration + Good for structured workflows and observability + Designed for production-scale deployment	– Newer ecosystem, less community adoption – Tied to Google Cloud stack – Some features still experimental

This is still a fast-moving ecosystem, and new libraries are proposed on a regular basis. These competing implementations are not only in competition but also complement and respond to one another. Some favor simplicity (for instance smolagents), others emphasize the capacity to handle highly abstract tasks (crewAI). When working with agents, one should build on the foundations already established by the community rather than reinventing the wheel. A great deal of technical code has already been written—for example, for handling faulty LLM outputs or retry limits—and rewriting it would require significant effort. At the same time, it is not advisable to become completely dependent on a framework and its own evolution.

We will briefly present a few of the most widely used libraries and their specific characteristics. From this short comparative overview, which you should of course supplement with your own research, several important points can be drawn:

- After a period of experimentation and stand-alone implementations, a call was made to return to simplicity and to code that can be read in its entirety before being used as-is[10]. This observation from Anthropic was corroborated by Hugging Face.
- Not all frameworks share the same objectives, and care must be taken to compare what is actually comparable.
- Disruptions to this landscape remain possible, for example with the arrival of agents endowed with vision or the capacity for "computer use."
- There is a clear trade-off between the simplicity and interpretability of a framework, and the complexity of the tasks it is able to handle. Frameworks such as LangChain or CrewAI promise to automate very complex tasks, but are relatively opaque, costly in LLM calls, and leave the user with little visibility or control over what is happening during execution. In contrast, libraries such as smolagents offer fewer capabilities but greater explainability and transparency regarding the steps of the process.
- There is also a trade-off between the simplicity of the code one has to write and the degree of control the user retains.
- None of these solutions is perfect or complete. For instance, one can observe a general lack of reflection on memory management in most frameworks, even though this is a central issue for building chatbots.
- These solutions should be seen as complementary. LangGraph provides good management of frameworks by representing them as graphs, CrewAI has pushed further the reflection on multi-agent systems, and smolagents is more interpretable and adaptable.

It follows that the choice of library should always be adapted to the specific project for which one wants to implement agents.

10 https://www.anthropic.com/engineering/building-effective-agents

III) AI Agents from an engineering perspective

a) Activity : spot the differences between Open AI API and Chat GPT

We will begin with a practical exercise. You will compare the **OpenAI API** with **ChatGPT as a product**. This will allow us to distinguish between:

- **A model exposed as an API** (pure LLM calls, stateless, where everything depends on what you put in the prompt and the parameters you configure),
- and **a conversational interface** (ChatGPT), where orchestration, memory, and sometimes tool use are layered on top of the same models.

The goal of the activity is to make explicit what is “just the model,” and what already belongs to the *agentic layer* built around it (cf notebook 1)

In 2025, many people still imagine that using ChatGPT means “talking directly to a large language model.” Yet a closer look shows this is no longer the case. What appears in the browser is not the raw model, but a conversational agent built on top of it — an assistant that wraps, steers, and extends the underlying LLM in ways that are invisible to most users. One way to see this distinction is to compare the ChatGPT web application with the OpenAI API, where the model can be accessed in its most direct form.

When working through the API, you can control every aspect of the interaction. For example, if you fix the temperature to zero and repeat the same call twice with an identical system prompt, you obtain the same result word for word. The process is deterministic, reproducible, and therefore testable. By contrast, trying the same comparison in ChatGPT often yields different formulations across attempts. The difference is not because the model has changed, but because the application adds hidden instructions and state management that you do not control. What looks like variability is, in fact, the product intervening to make the conversation feel more natural.

The same principle appears when looking at memory. In the API, you must pass the whole conversation history with every request; if you omit it, the model has no recollection of prior turns. In ChatGPT, however, the assistant “remembers” by design. It integrates the past exchanges automatically, sometimes even across sessions, and adapts its answers accordingly. This memory is not an intrinsic property of the model; it is the scaffolding that transforms a sequence of isolated calls into what feels like a coherent dialogue.

Another telling experiment concerns identity and personality. Through the API, the model has no default persona: you have to write a system message such as “You are a meticulous lawyer” or “You speak like a poet,” and only then does it adopt the desired style. In ChatGPT, the assistant already comes with a voice, a name, and an adaptable personality profile. The difference illustrates a fundamental shift: with the API, the engineer defines the assistant; with ChatGPT, the product defines it for you.

Structured outputs are not the only place where the difference between the raw model and the product becomes evident. Another revealing aspect is the way code execution is handled. When you interact with the API directly, the model can generate code — a Python snippet to calculate statistics, or a script to plot a graph — but nothing is executed unless you, as the engineer, take the output and run it in your own environment. The model produces text, and responsibility for running it lies with you. In ChatGPT, by contrast, there is an integrated code interpreter. When you ask it to draw a heatmap, the assistant does not simply print Python code; it actually executes it in a sandboxed environment provided by the platform. The output is then returned as an image seamlessly embedded in the conversation. From the outside, it looks as if the model has drawn the heatmap itself, but in truth the product has captured the generated code, executed it in a controlled environment with restricted resources and permissions, and then piped the result back into the dialogue. This invisible mediation is precisely what marks ChatGPT as an agent: it delegates to an execution tool and integrates the result, giving the user the illusion of a unified intelligence.

A similar logic underlies what OpenAI calls “deep research” or “guided workflows.” If you ask the API directly to perform a complex investigation — for example, to compare several scientific positions and write a structured report — the model will do its best in a single shot, but often the answer is shallow, fragmented, or inconsistent. Through the ChatGPT assistant, however, the same task is handled in stages. The system first drafts a plan, then queries sources, then generates sections, and finally refines the whole into a polished document. Each of these steps corresponds to an internal prompt-and-response loop, orchestrated by the agent layer rather than by the user. What you receive is not one model output but the result of a miniature workflow: an internal research assistant guiding itself through a pipeline. Again, the effect is that the assistant seems more capable than the model alone, because the product has transformed raw generation into a process with memory, structure, and self-correction.

Taken together, these comparisons show that ChatGPT today is not simply a model exposed through a chat interface. It is an engineered agent: it manages state, steers personality, decides when to call tools, and presents itself as a coherent assistant. The OpenAI API demonstrates what the raw model can do, but the product goes further by wrapping it in layers that create the illusion of a single intelligent partner. Understanding this distinction is essential. It reveals why ChatGPT feels so fluid and capable, and it clarifies what it means to design your own agents: you are not building a new model, but building the scaffolding — the prompts, the memory, the tool connections — that turn a model into an assistant.

b) The steps of Gen AI integration

Most users experience ChatGPT or similar large language models through a web application provided by the vendor. A smaller group of technically skilled users with access to powerful GPUs (on the order of several thousand euros) may download models and run them locally using inference libraries such as vLLM, in the same way that smaller models like BERT have traditionally been deployed. However, this is not the way LLMs are generally used in production. In professional settings, they are almost always integrated into applications via API calls, or occasionally through a locally hosted inference server that exposes an API.

What this means in practice is that an application embedding GenAI features must treat the model as a component in a larger pipeline. The application typically:

- Stores the prompts it uses, so that requests are consistent and reproducible.
- Parses the LLM's responses, often enforcing a structured output format (e.g. JSON) to guarantee machine-readability.
- Injects the parsed content as variables into the next stage of the program, whether that stage is a database update, a function call, or a user-facing feature.

If the application is intended to act as a conversational assistant, there is an additional requirement: the application itself must store the chat history. This context is what allows the assistant to produce coherent multi-turn interactions. The LLM provider does not persist your conversation state across calls — unless you use a specific productized assistant layer like ChatGPT, where memory is managed on the platform side. In an integrated system, preserving and managing dialogue history is therefore the developer's responsibility, and it becomes one of the essential steps in building an agentic conversational experience.

This represents a profound shift in the way programmers are accustomed to working. Traditionally, when a piece of code was written, one expected it to behave deterministically: the same input would always produce the same output (unless randomness was explicitly introduced with a library such as `random`). Determinism was the norm, and it allowed developers to reason about programs, debug them systematically, and trust that once a function worked, it would continue to do so under identical conditions.

By contrast, when GenAI models are integrated into applications, this assumption breaks down. The same prompt given twice may yield different outputs, and the notion of "correctness" is no longer binary. Developers are forced to manage this inherent non-determinism. They do so by carefully crafting prompts to restrict the space of possible answers, by lowering the temperature to reduce variability, or by adding parsing and validation mechanisms to enforce structure on the model's output. Instead of absolute guarantees, one now works with probabilities, constraints, and guardrails.

Another major change is that the successful execution of the program is no longer entirely under the developer's control. The application depends on the availability of an external API. Latency, rate limits, or outages can all disrupt execution. Moreover, if the model version is not fixed, its behavior may change silently between runs, introducing new forms of instability. What once was a closed, self-contained codebase has become dependent on external services whose behavior may evolve unpredictably. Taming this instability becomes part of the new craft of programming with GenAI.

In this context, the role of LLMs inside applications can be understood in several modes. They act as content generators, producing summaries, drafts, or answers. They act as **content extractors**, parsing unstructured text and transforming it into variables that downstream functions can consume. And increasingly, they replace traditional control structures such as nested `if/else` or rules engines, making decisions about which path to follow in a workflow. In other words, LLMs are not just another library — they change the very texture of programming. The discipline evolves from writing deterministic functions to orchestrating probabilistic behaviors.

This explains why, at some point, generative AI crossed paths with workflow automation. One can see the creation of software features as a way to automate tasks: the task is broken down into smaller subtasks, and the different resources required are identified. Traditionally, this automation process was a shared mission, where product managers understood customer needs and developers implemented them.

Over time, several observations were made:

- Many of the tasks performed by a feature are common (opening a file, parsing it for information, etc.).
- Many of the external resources that programs use are also common (mailboxes, SQL databases).
- The switching, orchestration, and logic of most features are often the same.

This led to the idea of writing these “software bricks” once and for all, in a customizable way, and letting people assemble workflows from these pieces of software—much like building a Lego construction. These low-code and no-code approaches enabled non-technical users to build features without heavy development. If a new component was needed, it could be coded independently and added to the library.

The notion of workflow building relies on the fact that each step of a workflow can be understood as the transformation of a certain input into a certain output. The exact same can be said of a block of code. There is therefore, with workflows, a complete morphism between the steps of a workflow and the code that is executed.

Software capable of executing pre-constructed workflows are called **workflow engines**. These programs can run instances of workflows. The notion originally stemmed from robotics, where the main robot needed to be able to execute several tasks simultaneously, but was later extended to define low-code tools that make it possible to create “software without coding.” These products became particularly successful in enterprises for repetitive tasks such as email classification and handling, or advertisement workflows.

Several programs were developed, among them **n8n** and others. These systems allow users to build workflows via drag-and-drop interfaces: selecting workflow inputs (user input, emails, database events), choosing among connectors to load or transform data, and exporting the final result. More recently, these platforms have integrated generative AI—specifically, LLM calls at certain steps of the workflow to transform data.

Despite advertising themselves as AI agent platforms, their agentic capabilities remain limited: weak context handling, inability to create collaborative agent spaces, hierarchical architectures only,

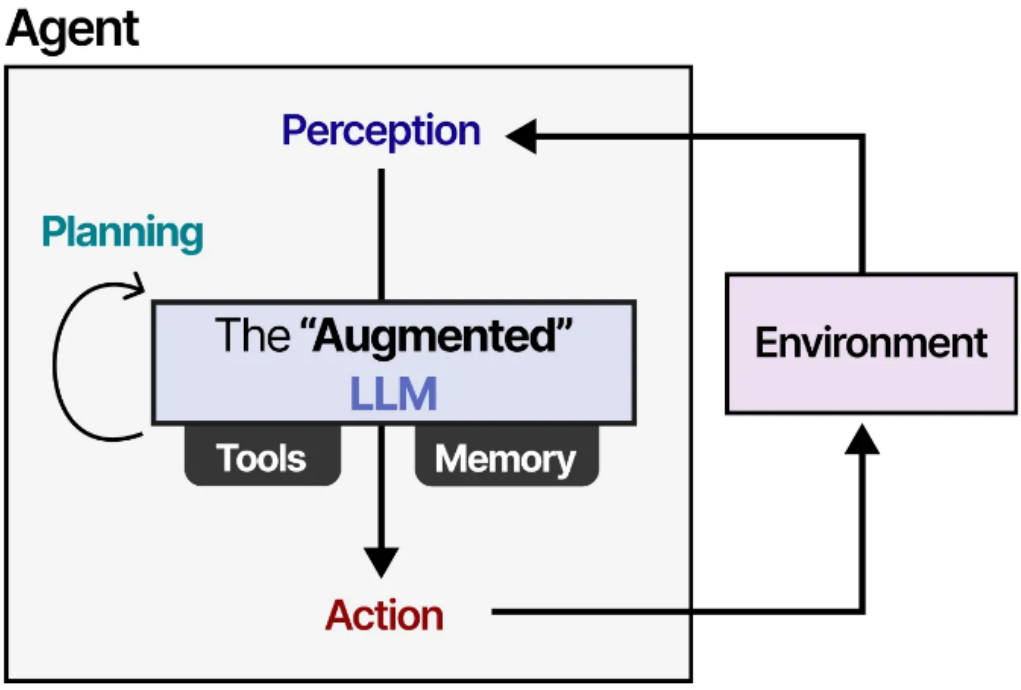
and so on. While workflows incorporating AI are indeed an important step toward agency, they do not represent the endgame toward which the field is now heading.

c) AI Agents : the final recipee

The implementation of AI agents is more a matter of software engineering than of artificial intelligence in the strict sense. A number of technical questions must be resolved if we want agents to be capable of carrying out complex tasks:

- How can we give the agent visibility over its environment, and how can we allow it to act dynamically upon it? This problem has been addressed through the notion of *tool use*.
- How can an agent keep track of the progress of a complex mission? This requires some notion of persistence and of the effects of previous actions.
- What should be done when the complexity of the task increases? How can the implementation of the agent be modified so that it is better able to handle this difficulty?

From the initial intuition of an “autonomous agent” presented in the 2023 **ReAct** paper, the complexity of agentic systems has only grown. Yet the fundamental principles behind them have not changed much.



For the purposes of this course, we will define an **AI Agent** as a computer program that has the ability to call a Large Language Model in order to autonomously solve a complex task without external guidance or supervision.

In the next classes, we will examine the essential components of such programs:

- To carry out its tasks effectively, the program (let us call it the agent) must have access to external resources. Unlike workflows, where access to resources is scheduled and predefined, the agent must be able to request them dynamically, depending on what the LLM determines is necessary. To bridge this gap, the notion of *tooling* was developed. In Chapter II we will study its evolution and possibilities.
- Many tasks that agents need to handle are knowledge-intensive, while LLMs are limited by their context windows. In an anthropomorphic metaphor, developers introduced the idea of *memory* to describe information persistence. In Chapter III, we will study how to optimize context management in order to improve agent performance.
- As tasks grow more complex, simple ReAct agents with a limited number of tools fail. More advanced (and often more costly) architectures have been designed where multiple agents cooperate, with varying degrees of success. We will examine some of these systems with a particular focus on *Deep Research Agents*, which are capable of producing full reports from a single initial query.
- After so much theory, it will be time to see how agents can be applied to truly interesting—or at least profitable—problems. But not before you are tested on these theoretical foundations, of course.
- Finally, with a stronger grasp of the architecture of AI agents, we will conclude the semester with a study of their performance and potential. The more complex an AI pipeline becomes, the more it is prone to error propagation. We will discuss both the traditional benchmarks used to evaluate agents and a set of key performance indicators that bring the notion of *observability* to the forefront. Understanding the inner functioning of agents is the only way to use them successfully in practice.