

# Chapter 4 : Agent Architectures and Task adaptation

With simple agents (language models extended through tool use and careful data curation) we have already expanded the domain of fully and reliably automatizable tasks. As Joelle Pineau, now Chief AI Officer at Cohere, recently put it, we are “*expanding the action space of AI systems.*<sup>1</sup>” The expression, borrowed from the field of reinforcement learning, is strikingly accurate to describe the latest evolutions of both foundation models and AI agents.

The wall of automation has moved, but it is still there. The idea remains the same, deeply rooted in classical computer science: when a problem is too complex to solve directly, break it down into smaller problems that you, or here, a single agent, can handle. The fundamental question then becomes: **how can we automate a complex task that no individual agent can perform alone?**

Before we go further, it is worth clarifying what we mean by *action space* and by *automating further*. Expanding the action space means increasing the diversity and scope of the operations an AI system can perform autonomously to reach a goal. Automating further or harder means tackling tasks that require more reasoning steps, more memory, or more coordination. Shorter, bounded tasks are easier to automate than long, open-ended ones, mainly because of error propagation and goal misalignment. Each additional step introduces a new chance for drift. This defines one of the main current limits of LLM-based agents: for now, their automation abilities are mostly confined to *data-related tasks*, tasks that live in the digital world of text and code (though tooling can give LLM real world capabilities as it is currently explored in Robotics).

As discussed earlier, AI agents are pieces of software that embed language model calls in order to fully automate tasks. They are the new vessel through which automation is being explored today. The initial design of agents aimed to test the limits of what could be automated using LLMs to go beyond their intrinsic scope, which was originally confined to generating text from an instruction.

In Chapter 2, we saw how *tooling* grants LLMs new capabilities to interact with their environment and thus achieve more complex objectives. It radically extended the automation frontier: an LLM can now move from merely suggesting improvements to a piece of code to dynamically correcting it, testing it, and even closing a GitHub issue on its own. Yet this expansion also exposed new weaknesses at the very core of the agent. Some tasks remained systematically unsolved, even when adequate tools were provided, revealing that the LLM’s reasoning and control mechanisms can themselves become the limiting factor.

In Chapter 3, we examined another decisive layer: the *context*. The data exposed to the LLM inside the agent has a major impact on its performance. We studied how to optimize the agent’s productivity by ensuring that it remains aligned with its goal. This phase of *data curation* is crucial in data-intensive tasks, where the quality and structure of contextual information often determine the difference between success and failure.

If we wish to push automation further, two main avenues of progress remain. The first is to improve the models themselves; the second, to improve the way these models are integrated within the automation process, that is, to rethink the **agent architecture**.

---

1 <https://www.youtube.com/shorts/-wDbQtOOjBs>

Improving the models follows the traditional path of machine learning research: better alignment, more efficient learning algorithms, richer datasets, more compute. At inference time — or, perhaps more appropriately here, *during the agent's run*, techniques borrowed from LLM optimization can also be applied: *test-time compute, best-of-N sampling, Tree-of-Thought or Graph-of-Thought* reasoning, and so forth. However, these methods tend to be costly and yield uncertain or marginal improvements. They continue the line of classical ML research rather than redefining it.

Tool calling and context engineering have already appeared as two major engineering paradigms for integrating models into functional systems. Yet a third, less explored frontier remains, one that concerns not the model or its inputs, but the **architecture of the agent itself**.

But this work on, pardon the pun, *embedding* the model inside a more complex application is still very new. The field of agent architecture design is barely two years old. It is a direct consequence of the paradigm shift introduced by large language models.

Before 2018, the dominant logic was: *design your own model*. This was the era of **feature engineering** and **end-to-end architectures**, each tailored to a specific task. Between 2018 and 2022, with the advent of **pretrained language models (PLMs)**, the paradigm inverted: *start from an existing model and fine-tune it for your application*. The model's output was then directly consumable (classification, tagging, sentiment, translation...) each with a clear objective and a fixed architecture.

Today, with LLMs, the model is no longer the endpoint of the pipeline; it has become a **component** to be *channeled* toward a use case. Depending on the task, this channeling process naturally varies, much as fine-tuning architectures once differed between classification, event extraction, or scoring.

In classical model design, we optimized three layers:

- **Input** : the feature space, both raw and metadata (cf. input/output tags in dependency parsing). This strongly echoes what we now call *context engineering*: selecting and formatting the right information.
- **Output** : the target layer: what do we want to obtain, and how do we adapt the model's structure to yield it?
- **Intermediate operations** : the latent transformations or reasoning steps that bridge input and output (for example convolution steps in RNN)

The agentic turn stems from the same realization that once drove multimodal learning: different tasks ultimately rely on learning the same underlying *representations of meaning*. Building a sentiment classifier, an event detector, or an image captioner all required the model to internalize semantic relations. The shift, therefore, was to **learn meaning deeply once**, through heavier pretraining, and then focus on how to *leverage* it.

This paradigm opened several new paths: it made it possible to create **close variants of the same model**, first fine-tuned for a task, then specialized for a domain thus reducing overall training costs. It also facilitated **multi-task learning**, where related objectives could be learned jointly. Most importantly, it unlocked progress in fields that were previously limited by the scarcity of labelled data.

In the classical ML world, scaling performance required more *depth, compute, and data*. In the agentic world, we can still scale along these axes, but we have new levers as well: **better tools, more structured architectures, and even multiple cooperating agents**. In that sense, the contemporary challenge is one of *translation*: bridging the vocabulary and mindset of traditional machine learning with the emerging discipline of **AI engineering**.

Agentic AI thus appears as the direct continuation of the previous paradigm. It makes new tasks immediately available, lowers the entry threshold, and shifts the effort elsewhere — toward **better task framing and the search for the right architecture** to solve it.

The main difference lies in its **recursive nature**. Several model calls are now possible within the same process, something that was rather rare in the ML era. Back then, we had model *flows* and *pipelines*, but they were static and rigid. Agentic architectures introduce a new degree of **plasticity**: the capacity to adapt dynamically to intermediate results, to branch, retry, and reorganize reasoning steps on the fly.

Yet, in many ways, the core challenge remains the same. Classical machine learning sought the best trade-off between **cost** (training = data + compute + inference) and **performance**: building the most frugal and efficient model for a given task, adapting a known task to a new domain, or creating a new task altogether. Agentic AI inherits this spirit of *optimization under constraint*, but at the level of systems rather than models.

At the same time, AI engineering shares much with **software engineering** — and rightly so, since we defined agents as software entities. Automating a complex process, such as customer support, can be decomposed into smaller, manageable steps: new email triage, data retrieval, draft generation, revision, and reply. This is precisely how a programmer would structure a complex function into coherent sub-modules.

Likewise, the central question becomes: **how can we build minimal yet capable AI agents for any given task?**

Two fundamental problems immediately arise:

1. **How can we design complex agent architectures that handle complex tasks reliably**, avoiding the common pitfalls of error propagation and compounding uncertainty?
2. **How can we identify or learn the appropriate architecture for a given domain**, and ensure that multiple agents can cooperate over extended missions? Such systems must remain highly adaptive, not a sequence of predictable steps, but an evolving process.

In the end, all these questions converge toward a familiar tension, the one between **control** and **exploration**. The more we constrain an agentic system, the more reliable it becomes; yet the more we let it explore, the greater its chances of finding the truly correct or creative solution. Striking the right balance between the two is, perhaps, the defining challenge of modern AI engineering.

We'll start by revisiting the **definition of agents and the evolution of their architectures**, from the early ReAct loop to controlled automation systems and the first signs of multi-agent coordination. We then examine the engineering of **how complex agentic systems are built**.

Finally, we turn to a **concrete use case**, the design of *deep research agents*, to illustrate how these architectural principles translate into practice for long-horizon, knowledge-intensive tasks.

# I) The evolution in architecture : from simple loops to Multi Agent Architectures

Since a clear definition of what constitutes an *AI agent* has only recently emerged, and remains, to this day, somewhat elusive, a wide variety of concurrent implementations have flourished, ranging from the simple embedding of an LLM into a single feature to the construction of complex multi-agent systems.

Several interesting observations can be drawn from this rapid architectural evolution over the past two years:

- **An evolutionary process.**

The development of agent architectures followed a pattern reminiscent of biological evolution. Early designs were small and tentative, *mutations* of existing ideas, and only the viable ones survived. Many early frameworks now lie in GitHub oblivion, while a few successful design patterns became dominant. Independent experiments often converged toward the same conclusions, not necessarily the *best* solutions, but *viable* ones. The ReAct paradigm, for instance, lies at the core of most current agent frameworks. The similarities between Microsoft's Agent Framework and LangGraph are striking. Conversely, tools that proved suboptimal vanished quickly: almost no one mentions Amazon StrandAgents today, probably because Google's Agent Developer Kit is more powerful and expressive.

- **A response to limitations — expanding the action space.**

Each major architectural innovation was driven by the limitations of the previous generation and by the desire to *push the action space further*. The introduction of while-loops, for example, came from the need to solve tasks requiring multiple reasoning steps — such as multi-hop questions (“Who is the son of the woman who married the actor of the latest *Pirates of the Caribbean* movie?”). Likewise, the proliferation of specialized sub-agents arose because no single agent can viably handle dozens of tools at once.

- **A software-engineering mindset.**

Most frameworks were conceived by software engineers rather than AI researchers, and this background left a strong imprint on the agentic landscape. The *tool* often influenced the *function*, rather than the other way around. Familiar software design patterns reappeared under new names — *handoff*, for instance, is the agentic equivalent of spawning a new *thread*. This cross-pollination produced clever ideas but also led at times to over-engineering and unnecessary complexity which, fortunately, natural selection eventually eliminated. It also introduced a touch of anthropomorphism: we now speak of *manager agents*, *planning agents*, or *supervisors*, as if software components had careers.

In short, the development of agent architectures has been a form of **technological syncretism**, blending influences from AI research, software engineering, and evolutionary tinkering. This process has directly shaped how practitioners build and use agents today (most developers rely on a handful of mainstream frameworks), which in turn define what agents are capable of.

Understanding the intricacies of these architectures is therefore essential if we want to leverage agents efficiently to solve real-world tasks. In retrospect, this accelerated evolution also reveals a

deeper question: are these architectures the product of genuine innovation, or of our attempt to reproduce familiar patterns of control in a new paradigm

We will now trace this evolution step by step from the simplicity of early loops to the *software cathedrals* of modern multi-agent systems.

### a) The start of the ReAction

AI agents emerged from a broader question: **how can we effectively leverage LLMs inside real applications?**

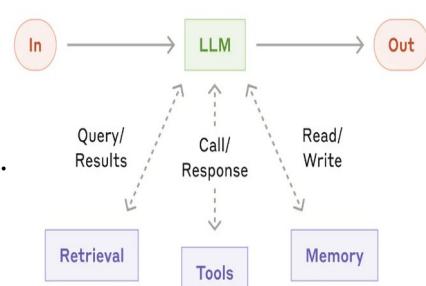
Large Language Models have reached state-of-the-art performance on many NLP tasks such as summarization or classification, and can now directly replace previous generation ML models within products. **But beyond replacing older systems, they also enable new kinds of tasks, or familiar ones applied to new domains**, where the LLM acts as a general-purpose model instead of training a specialized one.

Take sentiment analysis in a new language, for instance, say, Japanese. Instead of collecting a domain-specific corpus, fine-tuning a multilingual model, and repeating the whole training recipe that worked for English, one can simply query an LLM. There is a trade-off between training and inference cost, but it exemplifies the growing tendency to integrate LLMs directly into operational contexts rather than building new models from scratch.

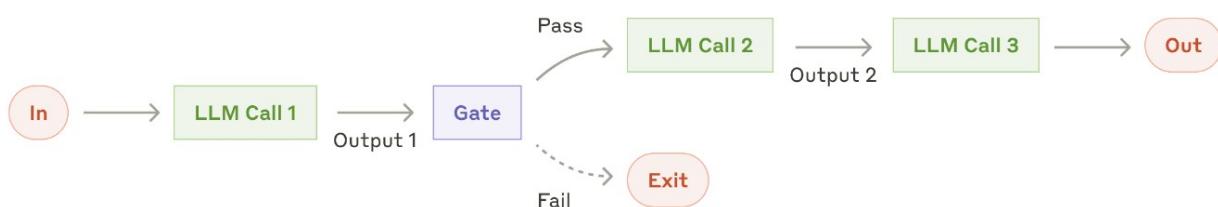
At some point, as this integration becomes more complex — especially when *tools* or *decision loops* are added. people start calling the resulting system an *agent*. In a seminal blog post published by Anthropic in December 2024<sup>2</sup>, the authors summarized what they observed across dozens of real-world collaborations: *the most successful implementations were usually the simplest ones*. They identified three design patterns that captured the majority of effective LLM integrations:

#### \* The Augmented LLM (the focus of Chapter 2).

Here, the model gains access to external tools and can use them at will. This architecture suits chatbots specialized in a product or domain, or tasks where the LLM must search, retrieve, or manipulate specific data.



#### The Workflow.



Several LLMs, each with different instructions or tools, are called in a fixed sequence. The models

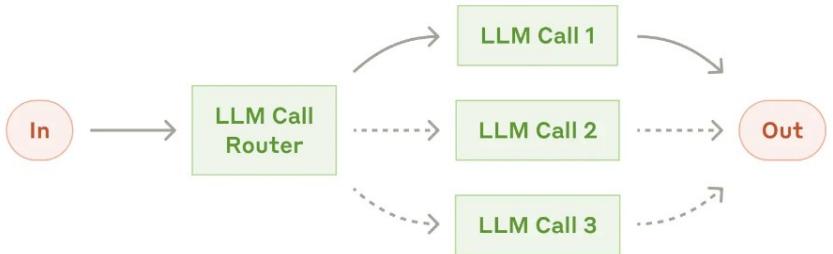
2 <https://www.anthropic.com/engineering/building-effective-agents>

do not choose which tool to use; rather, they act as *transformers* — for example, one generating a mail reply, another translating text, another summarizing results. This approach is powerful when a task can be decomposed into smaller, well-defined subtasks, and when this decomposition remains valid even in edge cases. In fact, many systems described as “agents” today are, in reality, **workflows**: deterministic pipelines where LLMs have no agency over the next step but are causal actions of the system .

### Routing.

In traditional programming, conditions are used to decide which block of code to execute based on a variable’s value, for example: “if

more than five documents are retrieved, send them by email to the customer.” But when dealing with unstructured data like text or images, defining such conditions becomes far less trivial. LLMs can serve as **semantic routers**, evaluating complex criteria on complex inputs, e.g., deciding whether a text speaks negatively about a company. Of course, one could train a classifier for that, but it would require labeled data and model maintenance.



All these patterns represent clever ways of integrating LLMs — but, *strictly speaking*, they are **not yet agents**. Their execution remains predictable: we can list all possible paths through the program. Even with tool use, the overall structure is deterministic — user message → (optional loop of tool call + execution) → assistant message.

A system truly becomes **agentic** when the LLM itself decides what to do next, when the flow of execution can no longer be enumerated in advance, and when determinism gives way to autonomy. And the most fundamental structure that enables this autonomy is one that programmers know all too well: **the loop**.

Most of the time, *instruct* LLMs were initially used in a **single-turn setting**, with follow-up interactions only meant to refine or improve the model’s first answer.

Yet, even with the right tools, many complex and data-intensive tasks simply cannot be completed within a single turn.

Take the example of **customer support automation**.

To fully handle an incoming request, the system must:

1. Analyze the email,
2. Extract the user’s main intention and relevant data,
3. Retrieve contextual information about the order,
4. Check corporate policies, and
5. Decide whether the case is simple enough to handle autonomously or whether it should be escalated to a human operator.

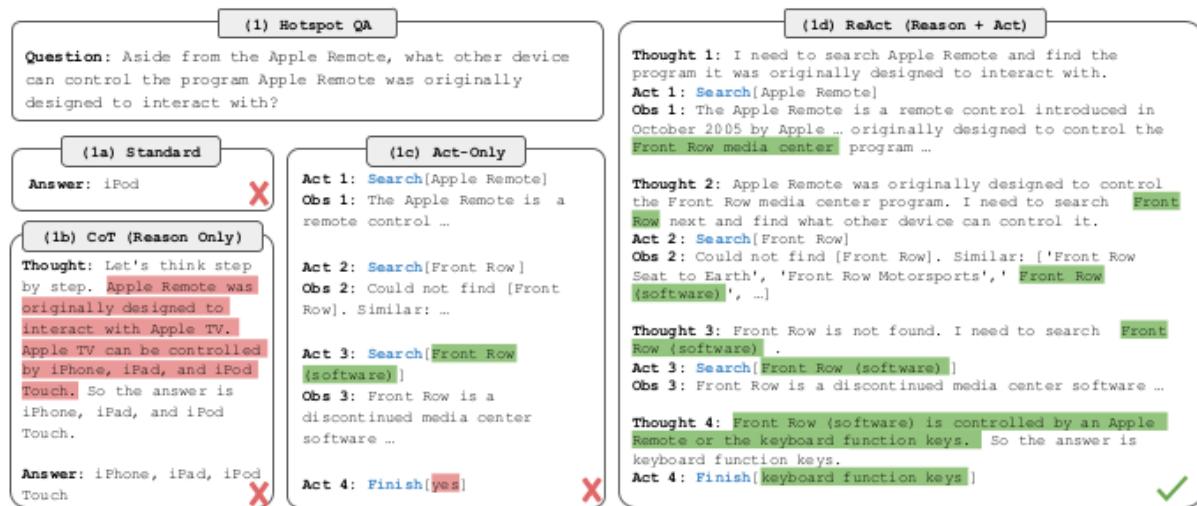
To achieve *true automation*, there must be **no human in the loop** between each model call. The agent must be able to perform actions *sequentially and autonomously* until its role is fulfilled. This behaviour can be elicited through the system prompt and maintained by **automatically routing each LLM output back to itself** until an *escape criterion* is met — for example, a maximum number of iterations, the generation of a special token such as <END> (the agentic equivalent of an *end-of-sequence* token), or the successful completion of a predefined action.

The first academic work that clearly hinted at this direction appeared in late 2022 (and have now been quoted 5166 times up to this day (please update) : ReAct: Synergizing Reasoning and Acting in Language Models<sup>3</sup>

It was the first to formalize the idea of **interleaving reasoning and acting** by allowing a language model to proceed autonomously for several rounds of reasoning-action loops to solve a problem. The authors demonstrated that this approach outperformed both of its predecessors:

- *Chain-of-Thought* prompting (which focused on reasoning across multiple LLM calls), and
- *Tool-Augmented Acting* (single-turn tool use).

In other words, the *agentic revolution* began with a classical research intuition: if two methods outperform a baseline independently, combining them will likely yield even better results<sup>4</sup>. Simple logic, but transformative consequences.



The formulation in the paper may appear abstract — “We augment the agent’s action space to  $\hat{A} = A \cup L$ , where  $L$  is the space of language.”

Yet the underlying idea is straightforward: **reasoning itself becomes an action**.

The intermediate textual traces that the LLM produces (its “thoughts”) are treated as concrete actions that can, in turn, trigger other actions — until the task is completed.

In the original paper, the authors fine-tuned a relatively small pretrained language model to demonstrate the benefits of this architecture (GPT-3.5 was not yet publicly available when the study was released in November 2022).

3 <https://arxiv.org/pdf/2210.03629>

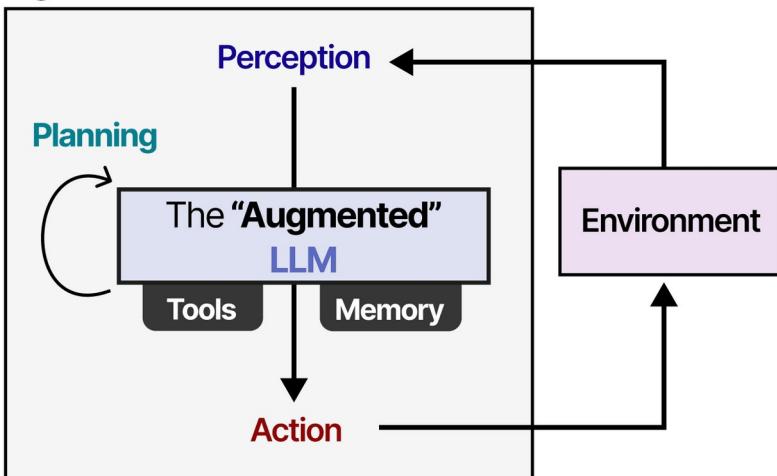
4 This combinatory does not always work and is often not worth the complexity tradeoff

Later, when applied to modern LLMs, this approach scaled dramatically and directly informed subsequent work on **tool use** and **autonomous agents**.

The strength of the ReAct framework lies in its balance: it is **intuitive**, **explainable**, and **effective**. Its traces provide a readable reasoning chain — allowing one to understand *what* the system did at each step, even if *how* it decided remains opaque.

It is more performant than simpler architectures and easily adaptable to diverse tasks.

## Agent



This first breakthrough **extended the model’s action space**, literally and conceptually, by transforming reasoning itself into a form of action. From that point on, the LLM was no longer just answering; it was *acting through thinking*. From this point LLM could gain real agency.

Interestingly enough, authors did not provide any schema of the ReAct pipeline<sup>5</sup> in their paper, but I cannot resist the urge to share Marteen Grotendorst’s illustration<sup>6</sup>.

This paper had a tremendous impact, not only on performance metrics, but on the very way people **conceived what an “agent” is**. Today, *ReAct agents* represent the **accepted frontier** between what qualifies as an agent and what remains a simple LLM integration.

Several frameworks, such as **Hugging Face Smolagents**, **LangChain**, or **PydanticAI**, essentially provide infrastructures to create *ReAct-style agents*: single LLMs capable of reasoning, acting, and looping through tool calls.

Most of the others ,including **Microsoft Agent Framework**, **OpenAI Swarm**, or **Google’s Agent Development Kit (ADK)** , present themselves as **orchestration libraries**, whose main function is to coordinate and combine multiple ReAct agents into larger, distributed systems.

In that sense, the *action space* of LLMs was indeed expanded.

Yet, despite their elegance and influence, **ReAct agents still inherit several structural limitations** that constrain their autonomy and robustness limitations that have driven the next wave of research in agent architectures :

### Cognitive bandwidth and specialization.

A single LLM has a fixed internal representation space and limited context window. It cannot simultaneously reason, store long-term information, and maintain fine domain expertise.

As tasks grow broader, specialization becomes necessary.. A ReAct agent cannot have a system prompt that is too broad, and the quantity of tools that we can use simultaneously is limited (around 10).

5 It is in fact very logical because in their perspective and timeline, they were building a different LLM **prompting strategy**,(same level as CoT) Agents did not exist yet and most agents were based on the work they did !

6 <https://newsletter.maartengrootendorst.com/>

## Conflicting objectives and modular reasoning.

Many complex tasks involve internal trade-offs (speed vs. precision, creativity vs. factuality, exploration vs. Safety.) A single agent cannot easily arbitrate between such conflicting goals within one reasoning loop.

## Parallelization of reasoning.

The ReAct loop is fundamentally sequential: one thought, one act, one observation at a time. Which means information stored at the beginning of the generation will at some point be lost due to context limitations.

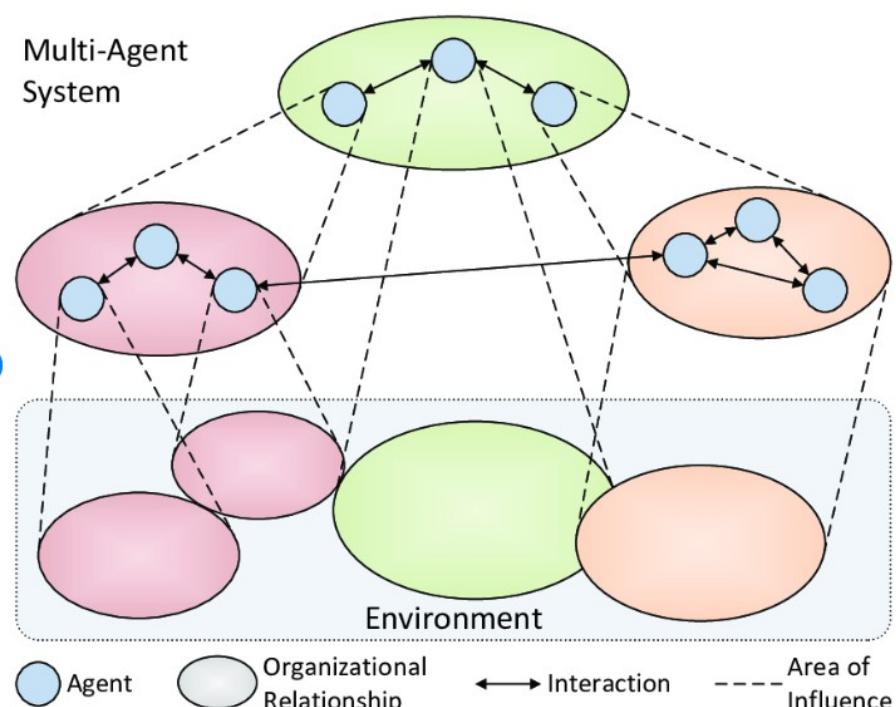
To automate complex, long-horizon, or open-ended processes, we must expand the action space horizontally, allowing several agents, each with distinct competences or perspectives, to collaborate, coordinate, and self-regulate. That is where modern multi-agent architectures begin: not as larger loops, but as ecosystems of reasoning.

## b) When does it start to be multi agent ?

The idea of **multi-agent systems** does not come from the LLM era, it originates in the long-standing field of **autonomous agents in embedded software and industrial control**.

There, the principle was already clear: since it is impossible to design a truly *general-purpose* agent, one should instead build **swarms of specialized agents** and develop methods to coordinate their actions to ensure that each agent knows what others are doing and to prevent concurrency conflicts. In such systems, every autonomous agent perceives only a **partial view of the environment** and is responsible for a **scoped and limited set of actions**. Coordination mechanisms, shared data structures, and supervisory layers are what turn this collection of narrow agents into a functioning whole.

Even though LLMs are arguably the most **multipurpose computational systems** ever created, they too face similar constraints on their **action space**. An LLM agent can only manage a limited number of tools (no more than ten, I beg you), and beyond a certain point, domain specialization becomes inevitable. What is particularly striking is that this logic was already formalized long before the current wave of agentic AI. The diagram I use here comes from a **2018 paper**, *Self-Reconfigurable Manufacturing Control based on Ontology-Driven Automation Agents* — and yet it perfectly depicts the structure of some of today's most advanced agentic systems.



The same problems , limited perception, coordination, concurrency, and specialization, have simply reappeared, reframed by the arrival of language models. The main technical problem is to build an efficient orchestration layer that ensures the global task of the agent is achieved, that is to say to ensure some determinism in growingly complex architectures.

First of all, **ReAct agents** can naturally replace simple LLM components in the integration patterns discussed earlier — *routing*, *workflows*, and *tool-augmented tasks* — allowing systems to **scale horizontally** by dedicating a specialized agent to each subtask.

Take again the example of a **customer-support workflow**:

a *triage agent* could classify the email and identify its intent, then forward it to the appropriate downstream agent — one handling general inquiries, another processing order-related issues, and a third managing disputes.

A *retrieval agent* could then gather all relevant information before passing it to a *writing agent* responsible for generating the final reply.

These new workflows, composed of **simple ReAct agents**, are significantly more powerful than classical deterministic pipelines, because **tool use is now dynamic rather than predefined**.

The agent decides in real time which tools to invoke and in what order.

As a result, we can no longer predict the full list of operations that will occur during execution — the workflow becomes *non-deterministic*, unlike traditional ones where every LLM call is pre-scribed.

Most complex “agents” deployed through low-code automation platforms such as **N8N** or **Dify** in fact belong to this intermediate category.

Fortunately, the mechanism of **tool use** also introduces a key property that enables true scalability: **compositionality**.

An **agent**, after all, is itself a complex piece of software that transforms certain inputs into structured outputs — exactly what a *tool* does.

It follows that specialized agents can, in turn, be **used as tools** by other agents.

This principle is implemented natively in several modern frameworks (for instance in **OpenAI’s Swarm** or **LangGraph**) through a mechanism known as **handoff**.

Handoffs allow an agent to **delegate a task** to another agent.

This can be achieved either by automatically generating a dedicated tool for each specialized agent at initialization (assuming each agent is properly described, with declared inputs and outputs), or by creating a **generic handoff tool** that enables the main agent to call any registered sub-agent type.

At runtime, this delegation works as follows: the main agent invokes the specialized agent, a new instance of it is created, executes its task, and returns the result to the caller.

The main agent waits for the completion of this sub-process before resuming.

In theory — and increasingly in practice — the output obtained this way is *qualitatively superior* to what a single, general-purpose ReAct agent could have achieved by juggling all tools within one reasoning loop. This compositionnality anables to extend the action space vertically by making the tools more and more advanced. It also makes the general orchestration of a single Agent run more tricky (cf part II). This **handoff mechanism** is not limited to individual ReAct agents.

More complex structures — such as **workflows embedding multiple ReAct agents** — can themselves be **encapsulated and called as single tools**.

In other words, entire multi-step processes can appear to another agent as a simple callable function, enabling deeper layers of abstraction.

This mechanism is also **recursive**.

A specialized agent designed for one main task can, in turn, include as tools even more specialized sub-agents. This is precisely how most modern **agent-building platforms** manage what is now known as *tool overload*. When users connect too many tools from an MCP (Model Context Protocol) server, these platforms automatically create **sub-agents** specialized by tool family — for instance, a *Gmail agent*, a *GitHub agent*, or a *Slack agent*. A higher-level, generalist agent then routes between its core reasoning capabilities and these domain-specific agents, each responsible for executing the right tool calls.

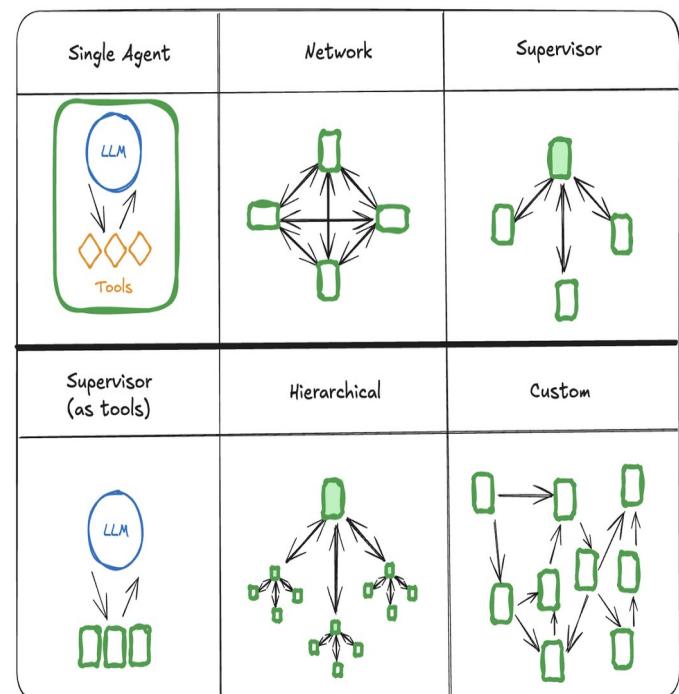
While this hierarchical delegation pattern prevents immediate cognitive overload for the main agent, it introduces a new cost: **token overconsumption**. Since information must propagate down the hierarchy — from the main agent to the most deeply nested specialized agent — the same contextual data is repeatedly transmitted, reformulated, and reinterpreted at each level. This recursive design expands the action space

structurally but does so at a growing computational and economic expense.

If **handoffs** provide a technical solution for getting several specialized agents to collaborate, they still do not offer a **clear way for agents to cooperate toward a single, shared goal**.

This challenge has given rise to several distinct **multi-agent design patterns**<sup>7</sup>. The first, which we have already described, is the **supervisor or hierarchical pattern**. In this setup, a main agent — often referred to as a *manager* or *controller* — delegates subtasks to subordinate agents.

While this structure works and is conceptually simple, it suffers from several notable limitations.



Beyond the problem of **token overconsumption**, hierarchical systems lack a **shared communication space** between agents. Each sub-agent operates in isolation, with no direct access to the reasoning or results of its peers. As a consequence, similar operations are often repeated by multiple subordinate agents, leading to **redundancy** and **inefficiency**. Meanwhile, the supervisor accumulates large quantities of **heterogeneous data** in its context and faces the difficult task of orchestrating all these fragments into a coherent final output. In response to these issues, other architectures have emerged — particularly those based on **graph structures**. In such systems, agents are modeled as interconnected nodes capable of exchanging information directly with one another rather than exclusively through a central controller. Conceptually, this design is far more elegant and closer to genuine cooperation. However, it also proves **much harder to monitor and control**, as information flows become non-linear, asynchronous, and often unpredictable.

<sup>7</sup><https://docs.langchain.com/oss/python/langgraph/graph-api>

## Where Does the Multi-Agent System Begin?

Defining the exact boundary between a **single-agent** and a **multi-agent** system is not as straightforward as it may seem.

It is not simply a matter of the number of models involved, nor of how many tools are available within a loop.

A single ReAct agent can already display *online specialization* — dynamically selecting the right tools or reasoning path within a single reasoning loop — without ever ceasing to be a single agent.

Likewise, even **complex workflows** that sequence several LLM calls may still not qualify as multi-agent systems if their execution remains **deterministic** — if every call and transition is predefined, and if the system's global behavior can be fully predicted from its structure.

The real threshold appears when **different instances of LLMs, each with their own objectives or system prompts, interact in a way that is not predetermined**.

A multi-agent system is one where these agents — possibly including the main supervisor itself — can **delegate, exchange, or negotiate** tasks with others (even with temporary copies of themselves) in a manner that cannot be fully inferred from the initial setup or from the output alone.

In short, **multi-agency begins where execution ceases to be deterministic**.

When the flow of reasoning and action depends on emergent interactions between agents rather than on a predefined sequence, we move from orchestration to **true cooperation** — from architecture to ecosystem.

### c) And beyond : self evolving agents

Even if it was not possible to cover every aspect in this limited space, it is clear that **orchestrating multiple agents into reliable agentic systems** remains the current technological bottleneck — and the main focus of companies seriously investing in Agentic AI today.

As Anthropic observed, the most successful use cases so far have been the simplest ones: the more complex the system, the greater the responsibilities and the stronger the need for a robust orchestration layer.

Yet, while most current research concentrates on **controlling and coordinating ReAct-based agents**, a growing number of studies now aim to design **agents capable of modifying their own structure**. The generic term *self-evolving agents* is often used to describe these endeavors — though it in fact covers a wide range of heterogeneous approaches. This new direction is the direct consequence of the broader **shift in focus from model architectures to agent architectures**.

As one recent survey puts it<sup>8</sup>:

“This paradigm shift — from scaling static models to developing self-evolving agents — has sparked growing interest in architectures and methods enabling continual learning and adaptation from data, interactions, and experience.”

An important distinction is emerging between **foundation agents** (ReAct-style systems with fixed behavior) and **self-learning agents**, whose adaptability lies outside the model itself.

8 <https://arxiv.org/pdf/2507.21046>

=“Unlike previous paradigms that primarily focused on updating model parameters, self-evolving agents expand the scope of adaptation to include non-parametric components such as context (prompts and memory) and toolsets.” This implies that many different techniques can be explored to reach agents capable of **dynamic adaptation** learning new tasks, integrating new tools, or adjusting to changing environments without retraining the underlying model.

The first strategies for agent evolution naturally focus on the **models themselves**, which remain at the heart of every agent. A *self-evolving* agent would ideally be able to **adjust its own parameters** to improve its performance on a given task. Current methods rely on **data augmentation** (generating synthetic datasets of test cases and expected results for new objectives) combined with **reinforcement learning** or **supervised fine-tuning** guided by an evaluation of the agent’s behavior (most often using another LLM as a judge). However, these techniques have clear limitations. They depend heavily on the **quality of the underlying models**, both for generating the training data and for assessing performance. If the model lacks accuracy or reliability, the entire feedback loop degrades. Moreover, this training pipeline is typically **external** to the agent: it adapts the model to a task from the outside rather than enabling the agent to evolve autonomously during its own runs. Strictly speaking, this is not *self-evolution* yet, but rather *assisted adaptation*. Agents, like model will experience data drifts once in production (the data they will be exposed to will slightly differ from the ones the agent was tuned on, which will cause a drop in agent performance) and these tuned agents, such as fine tuned models will not be able to dynamically adapt.

A second, more endogenous approach focuses on **prompt optimization** — allowing agents to **modify their own prompts** over time. This builds directly on existing research in prompt optimization for single LLM calls and extends it to multi-step or multi-agent settings. The principle is analogous to **hyperparameter search** in machine learning: given an evaluation set and performance metrics, the agent (or an auxiliary model) generates and tests multiple prompt variants, selecting or recombining the most successful ones. In multi-agent systems, several prompts may need to be optimized simultaneously, reinforcing the analogy with **grid search across interacting parameters**. Yet here again, the process requires **external data and evaluation** and remains largely **outside the agent’s own runtime**. It enables adaptation, but not yet continuous, autonomous evolution.

A further category of approaches aims to make agents capable of **learning from their own experience**. By analogy with *in-context learning*, successful interaction traces can be **stored and retrieved** based on their similarity to new inputs, thereby optimizing the agent’s future trajectories. For instance, when facing a new query, the agent can recall a previously successful reasoning path or workflow and reuse it during the planning phase. “Rather than only retrieving exact past instances, advanced agents distill experiences into more general guidance.” This process gradually enriches the agent’s **episodic memory**, allowing it to refine its behavior over time without retraining.

Another line of research explores agents that can **discover or even generate new tools** when needed. Creating new tools is a direct way to **expand the action space**, as it allows the agent to execute operations that were not initially part of its capabilities. This, however, raises major challenges: each tool must be **tested and validated** to ensure that it truly accomplishes its intended purpose rather than merely producing plausible outputs. Moreover, the ability to autonomously write and execute code introduces serious **safety and security risks**.

Finally, some researchers treat **agent architecture itself as a search problem**, essentially a form of **architectural grid search**. Beyond prompt tuning, many other parameters can be optimized: model selection, tool sets, context composition, or even node interconnections in multi-agent graphs. While most current work focuses on improving a predefined architecture, a growing body of research seeks to **design optimal agent architectures for specific tasks**. This exploration is particularly difficult due to the vastness of the **design space**, which is effectively infinite. To navigate it, methods such as **Monte Carlo Tree Search** or **genetic algorithms** are often employed to iteratively propose, test, and refine agent configurations.

The notion of a *singularity*, the moment when an AI system becomes capable of autonomously improving itself without human intervention, remains, for now, a powerful metaphor rather than an imminent reality. The main limitation lies not in model capacity, but in the **architecture of agents themselves**. Contemporary AI agents are still bounded by their design: they lack persistent self-awareness of their structure, long-term control over their own codebase, and the ability to safely validate or deploy self-generated improvements. In practice, what we are building today are not *self-autonomous agents*, but rather **agent-building machines**, systems that can help us design, test, and iterate over new agents more efficiently. They represent a significant step toward autonomy, but one that remains deeply **engineering-centric**, not evolutionary.

## II) Engineering the Agentic Stack: bottlenecks & challenges

The emergence of the **AI engineer**, for agents what the ML engineer once was for models, and the gradual disappearance of the **data scientist** from these projects both signal a profound industrial shift.

The focus has moved from *understanding data* to *building complex software systems* robust enough to process vast streams of information while continuously calling **high-latency, high-cost models**.

This “**second rise of the software engineer**,” often portrayed as the age of unstoppable agents assembled through low-code or “vibe-coding” tools, is largely a **myth**, one fueled by over-enthusiastic executives and self-proclaimed experts. In reality, the **engineering challenges** underlying agent architectures are substantial, intricate, and remain one of the main **bottlenecks slowing their large-scale adoption**. We’ll focus on some selected topics among (many!) other interesting ones.

### 1) Parallelization and scheduling

AI agents are designed to automate increasingly complex tasks, and the **time a human professional would need** to complete such a task is often used as a measure of its complexity. Using agents is therefore not only a way to **compress operational costs**, but also to **dramatically reduce task duration** — a potential game-changer in critical domains such as **finance, healthcare, or defense**.

The **bottleneck** shifts: there is no longer a need to wait for a human’s availability to trigger an action once sufficient data has arrived (for instance, as soon as a customer email is received).

An agent can begin working **immediately**, provided enough memory and compute resources are available. Ideally, the agent should also run **as fast and efficiently as possible** — often meaning *using as few tokens as possible* (see Chapter V on observability and optimization).

From an engineering standpoint, the **execution of a modern LLM-based agent is inherently distributed**.

Its components follow a **microservice architecture**:

- LLMs are invoked through **external APIs** or from **dedicated GPU servers**;
- many tools make **HTTP or database calls** to external services;
- while the **main orchestration layer** runs locally, typically on a **CPU-bound process**.

This distributed nature introduces **scheduling challenges**.

To execute a single step (such as a tool or LLM call), an agent may have to **wait for the availability of external resources**.

As a result, the orchestration code must be written to handle **asynchronous execution** — waiting for specific processes to complete without freezing the entire system.

However, resources may sometimes become **unavailable** altogether (for instance, when an API behind a tool goes offline), causing asynchronous calls to hang indefinitely.

For this reason, robust systems implement:

- **Timeout mechanisms**, to stop waiting after a reasonable delay;
- **Error handling and structured logging**, to trace and diagnose failures;
- **Retry policies**, allowing the system to reattempt failed operations a limited number of times;
- **Checkpointing**, to save successful intermediate results and resume execution later once the external resource is restored.

These mechanisms, though classical in distributed software systems, are now **essential for AI agents** expected to handle large volumes of data and orchestrate high-latency components reliably.

Agents, like any program, are launched within their **own execution threads**.

This means that a given CPU can only run a **limited number of agents simultaneously**, depending on its available cores and resources. Each agent should operate within its **own execution space**, complete with its working directory and storage environment, for example, to create or modify files



independently without interfering with others<sup>9</sup>.

The concept of an *execution thread* is therefore fundamental to understanding how AI agents and workflows actually run.

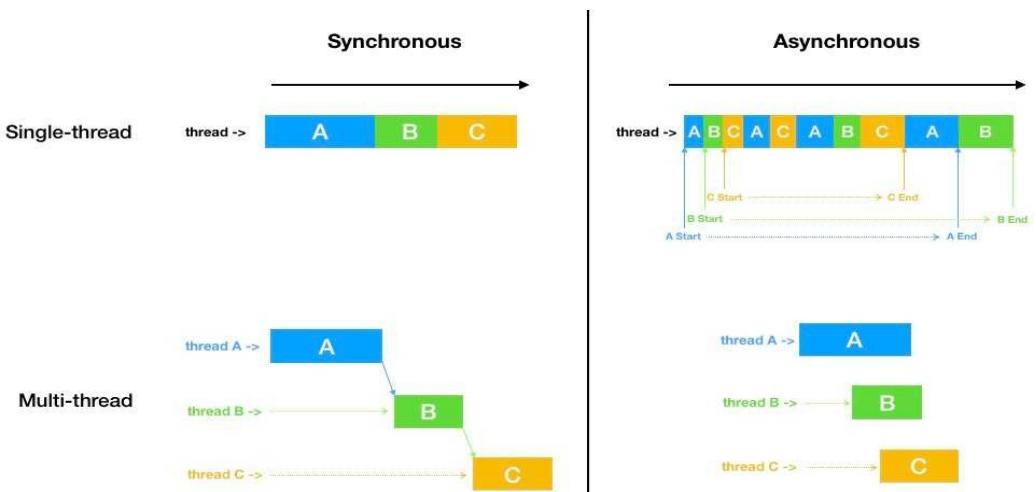
As we have seen, there are many points during an agent's lifecycle where execution pauses, waiting for **external resources** such as an API or an LLM response. During these waiting periods, the processor remains idle unless the system is designed to handle **asynchronous or concurrent execution**. From an efficiency standpoint, it would be wasteful to let computation power sit unused while one agent is waiting on I/O.

Depending on the **architecture** of the agent system, it is often possible, and desirable, to launch **multiple tasks in parallel**. Consider, for instance, a **search agent**: a *manager agent* might orchestrate three *sub-agents*, each responsible for investigating a different topic.

If these sub-agents were executed one after another, the total runtime would scale linearly with the number of subtasks. By contrast, running the three search agents **concurrently**, each in its own thread or process, allows them to work independently while the manager waits for all results to complete before synthesizing them into a final answer. This pattern—*parallel delegation followed by synchronized aggregation*—illustrates one of the key advantages of agentic systems when properly scheduled. It enables greater **throughput** and **responsiveness**, particularly in domains such as search, data extraction, or multi-document analysis, where tasks can be decomposed into independent subtasks and recombined at the end of execution.

This is why it is essential for AI agents to evolve from purely sequential execution to true parallelism<sup>10</sup>. In many existing frameworks, parallelization is achieved by simply launching subagents in separate execution threads. While this already improves efficiency, more refined architectures are emerging.

In graph-based frameworks, for instance, each agent's operations are represented as nodes in a computational graph. At runtime, all nodes that are ready for execution, meaning that their dependencies have been satisfied, can be stored in a queue and executed as soon as a thread becomes available, regardless of their nature (LLM call, tool invocation, or subagent handoff) or their position in the broader multi-agent structure. This dynamic scheduling strategy ensures that no computational resource remains idle while pending tasks are waiting to run.



9[https://cookbook.openai.com/examples/codex/codex\\_mcp\\_agents\\_sdk/](https://cookbook.openai.com/examples/codex/codex_mcp_agents_sdk/)

building\_consistent\_workflows\_codex\_cli\_agents\_sdk see this cookbook for how to safely design multi agent systems ;

10<https://www.youtube.com/watch?v= PgOfuJ5s7A>

## 2) Handling data flow : the limits of blind patching

With parallel execution and the growing complexity of agent architectures, **new challenges arise in keeping the data flow clear and robust**. In a simple ReAct agent, the flow of information is easy to follow: a linear exchange between user, assistant, function calls, and tool results.

The reasoning path unfolds as a sequence, much like a conversation. However, in systems with **parallelized subagents**, each agent follows its own local flow, and the global data graph becomes multidimensional. It is therefore crucial to maintain **explicit control over which agent has access to which data**, especially when variables share the same name or refer to overlapping entities.

Asynchronous execution adds another layer of complexity.

When several threads or agents operate at the same time, they may attempt to **access or modify the same resource**, such as a file, a database entry, or a shared memory block.

Without proper safeguards, this can lead to **inconsistent states**, duplicated writes, or even data corruption. To avoid such issues, developers must implement **locks or transactional controls** on key resources, ensuring that only one process can write or update them at a time. These precautions are fundamental to maintain data integrity in systems where many agents run concurrently, and they highlight the delicate balance between **parallel performance** and **execution consistency** that every agent architecture must achieve.

Keeping the data flow stable is also complicated by the **non-deterministic nature of LLM components**.

An agentic workflow connects multiple software bricks, many of which we did not design ourselves, and they must be “glued” together through a **common language of data exchange**.

In practice, three main representations dominate:

1. **JSON**, the most common choice, since LLMs were often post-trained to follow its syntax and it is easy to parse.

However, in parallelized environments, **duplicate field names** or **conflicting key updates** can appear, forcing the orchestration layer to explicitly track variable lineage and state history.

2. **Markdown**, a structured yet human-readable text format, is often used to store task lists, planning notes, or agent instructions.

It provides clarity and hierarchy, making it well-suited for mixed human-AI collaboration.

3. **Executable code**, which combines data and instruction in one concise and expressive representation.

While powerful, it introduces serious **safety and validation challenges**, as code must be both syntactically correct and secure to execute.

Beyond data serialization, **tuning the non-determinism of heterogeneous components** remains a major engineering challenge.

External APIs may **change their interface** without warning, breaking existing workflows.

Tools imported through an MCP server may **require more parameters** than intended or expose sensitive data that the LLM should not access for privacy or compliance reasons.

These practical problems show that building reliable data pipelines for agentic systems is less about

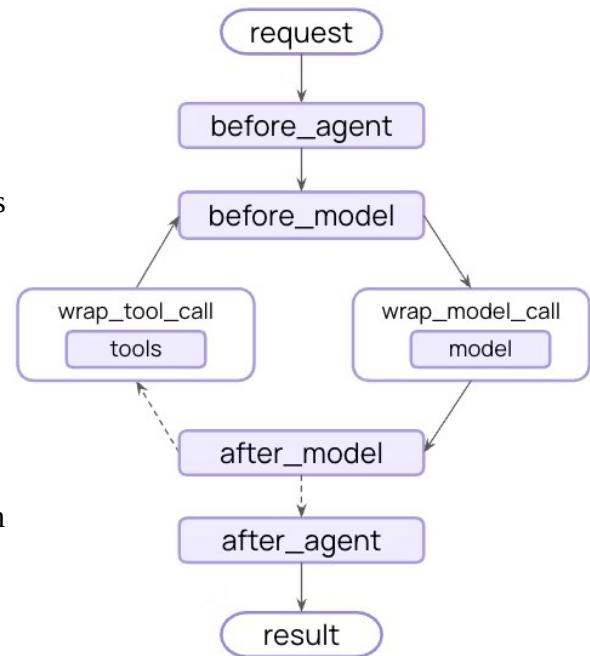
patching errors as they appear, and more about **designing explicit, verifiable communication contracts** between every component of the workflow.

To manage these heterogeneous and sometimes unpredictable components, developers increasingly rely on **middlewares** or *intergiciels* in French. Middlewares recently gained traction in several major agent frameworks such as **LangGraph**<sup>11</sup> and **Microsoft Agent Framework**, precisely because developers realized that they were not optional but **necessary**.

A *middleware* is a software layer that sits **between two components** of a system and manages or transforms the **information exchanged** between them. Its role is to ensure that communication follows certain rules, formats, and constraints, without requiring each component to know the internal logic of the others.

In the context of **AI agents**, middlewares act as **filters inserted into the data pipeline**.

They can intervene **before or after** any major action such as a tool call, an LLM call, or a handoff between agents. By doing so, they allow developers to **regain control over external components** filtering or reformatting outputs, validating incoming data, injecting missing context, or masking fields that should remain private.



Many **middlewares** are now provided **by default** in modern agent frameworks, and they can serve very different purposes.

Some common examples include:

- **Summarization middleware**, which automatically compresses the agent's context once a certain token limit is reached (see Chapter 3). This acts as a form of *short-term memory*, preventing uncontrolled context growth.
- **Human-in-the-loop middleware**, which requests explicit confirmation before executing potentially sensitive actions, such as sending an email or publishing a file.
- **Context editing or tool selection middleware**, which dynamically rewrites prompts, filters tools, or adjusts parameters to improve efficiency and safety.

In practice, middlewares are an **architectural choice** that allows frameworks to **factorize operations of various kinds** (logging, context management, retry policies, data validation, or even monitoring) under a single, uniform mechanism.

But at their core, a middleware is often **no more than a few lines of code** inserted between each major step of the agent's run. They are simple in form, yet essential in function, serving as the **nervous system** that keeps complex agentic workflows coherent, observable, and controllable.

11 <https://docs.langchain.com/oss/python/langchain/middleware> the illustration is from their documentation.

## c) from AI Agents to Agentic AI

Building a single, working agent is already an achievement (and more than most people manage) but it is not enough to ensure **successful deployment**.

In practice, multiple agent runs must be launched each time an event triggers the system, which requires implementing **multithreading** and **parallel orchestration** in the software layer that handles agent execution. This orchestration layer is **external to the agent's own code**, and designing it correctly is yet another exercise in high-precision software engineering, but it is also a **necessary condition** for the large-scale adoption of AI agents.

This broader shift is often described as the transition **from “AI Agents” to “Agentic AI.”** As one forthcoming O'Reilly book I'm looking forward to reading aptly puts it:

“With the rise of autonomous agents, the question is no longer *How do we build agents?* but rather, *How do we manage an entire ecosystem of them?*”<sup>12</sup>

Reaching this level of maturity raises several **technical and infrastructural challenges** that must be addressed before true Agentic AI can emerge:

- **Infrastructure scalability**

All components of an agent (its LLM, tools, APIs, and even the agent instances themselves) must be clearly managed from a **readable configuration**. This configuration should link directly to **hardware availability** (CPU threads for orchestration, GPU allocation for hosted models). To achieve this, many teams are now exploring **container orchestration systems**, such as *Kubernetes*, which can host microservices and maintain smooth communication between them. An **infrastructure control layer** of this kind is essential to scale agent ecosystems safely and efficiently.

- **Data governance**<sup>13</sup>

AI agents inevitably process **sensitive data**, from corporate knowledge bases to private user information. Some of this data must remain local or be stored under specific **compliance constraints** (for example, GDPR). This is particularly challenging when using **outsourced LLMs**, since external calls can expose private data to third parties. Robust data governance therefore requires **fine-grained access controls** to ensure that each agent or tool only accesses what it strictly needs. Mechanisms for **data anonymization** and **context filtering** can further strengthen compliance and privacy guarantees.

- **Security**

Beyond governance, the system must protect itself from **unauthorized access**, **data poisoning**, and **exfiltration** attempts. External actors should not be able to inject malicious data or exploit vulnerabilities in the agent's environment. This can be achieved by filtering the agent's inputs, testing intermediate outputs through **guardrails**, and enforcing **multi-layer authentication** (for tools, models, and agents). **Encryption** and **audit logs** also play a critical role in securing both the agent's behavior and the data it handles.

---

12 <https://www.oreilly.com/library/view/agentic-mesh/9798341621633/>

13 <https://learn.deeplearning.ai/courses/governing-ai-agents/lesson/tqxyfo/introduction> see the course on that topic

These questions usually arise later in the development process.

Most agent projects (including your final course project!) start as **proofs of concept**, where *feasibility takes priority over scalability and governance*.

Yet, once the prototype works, rigorous engineering practices become indispensable to ensure a **safe, secure, and maintainable deployment**. Agentic AI is not just about building intelligent agents; it is about building **ecosystems of agents that can be trusted to operate in the real world**.

## III) Use case : Building a deep research agent

Now that the main principles and bottlenecks of agent architectures are clear, it is time to focus on how to **build an agent for a concrete task**: the generation of a **search report** on a particular topic. We focus on this use case because it combines many capabilities discussed so far, reasoning, planning, tool use, and information synthesis, and because it has numerous extensions and applications. Historically, it is also one of the first **multi-agent systems** that was exposed to end users by major LLM providers, and a vast literature now surrounds it.

### a) Deep Research: A Perspective

What exactly does *deep research* mean?

The term refers to **AI-powered applications that automate complex research workflows** by integrating three key ingredients: large language models, advanced information retrieval, and autonomous reasoning capabilities.

However, the name itself is somewhat **spurious**.

“Search” naturally evokes search engines, yet in this context it refers not only to web queries but to any process that retrieves information from a **knowledge base**, be it online data, corporate repositories, or internal document stores. And “deep”? It does not refer to deep learning itself, nor to the *DeepMind* lineage, but rather to the **depth of reasoning** involved, going beyond a single query-response exchange to extract, analyze, and connect information from multiple sources.

In other words, *deep research* contrasts with the **shallow or immediate answers** produced by standard LLM prompting. It is about **digging deeper**, structuring knowledge, and building informed syntheses on complex questions. Typical examples include literature reviews, market or economic reports, and long-form analyses, tasks that require both **retrieval** and **discursive organization**. Because of this iterative reasoning and multi-step planning, a deep research run usually lasts several minutes rather than seconds.

The development of deep research agents can be divided into three main phases:

- **2023–2024: Early explorations.**

In December 2024, Google Gemini pioneered the first integrated *Deep Research* feature, focusing on basic multi-step reasoning and knowledge integration.

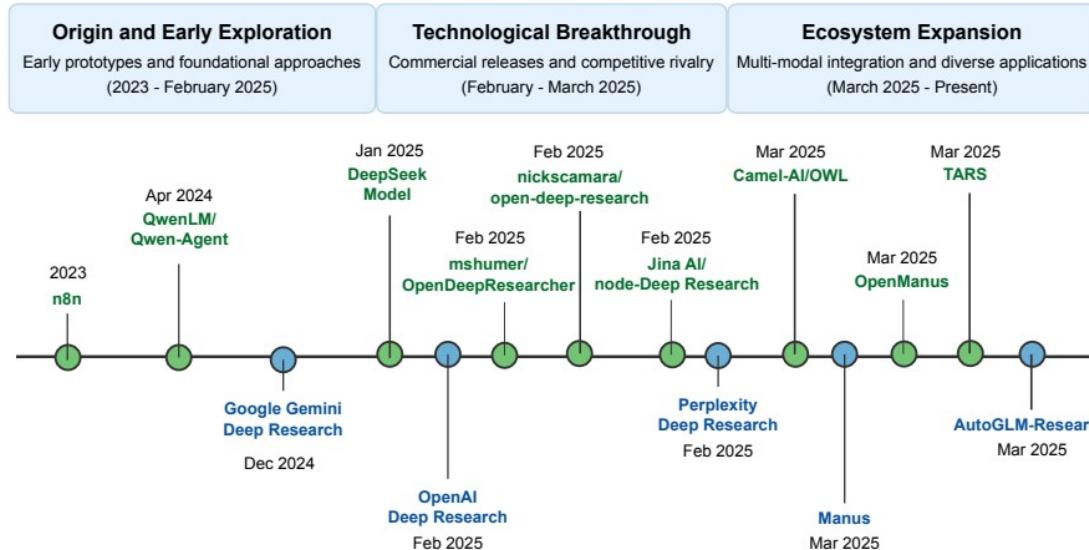
- **2025: Technological rivalry.**

In February 2025, OpenAI released its own version of *Deep Research*, marking a significant leap forward.

This implementation leveraged **reasoning-optimized models** and strong **parallelization capabilities**, inspiring a wave of open-source equivalents such as **Hugging Face's OpenDeepResearch**, **LangChain Deep Research Toolkit**, and others.

- **2025–present: Ecosystem extension.**

What began as a proprietary feature has now evolved into an **independent class of agentic systems**.



Despite being marketed as new and revolutionary, Deep Research was in fact the **first mature application of agent-engineering principles** that had already started to stabilize through frameworks such as LangChain or N8N.

Its success lies not in conceptual novelty, but in **efficient task adaptation** and **broad diffusion**, which made the approach accessible and widely reusable.

## b) Cookbook :Comparing Deep Research Architectures

We have now seen how Deep Research (*DR*) agents emerged from the convergence of reasoning-oriented LLMs, retrieval systems, and orchestration frameworks.

But how do we move from the **concept** of a research agent to a **functional architecture** that can truly perform end-to-end investigation tasks?

This activity invites you to explore and compare several existing Deep Research implementations, both proprietary and open source , to identify the **architectural principles, design patterns, and technical innovations** that make them effective.

## Objective

Understand how different implementations of Deep Research agents (OpenAI, Google, Hugging Face, and LangChain) translate the same conceptual goal — automated reasoning, retrieval, and synthesis — into distinct architectural choices.

### Instructions :

With your group, study the documentation associated with a deep research implementation and understand how it was built :

*System architecture:* is it a monolithic ReAct-like loop, a pipeline, or a multi-agent graph?

*Implementation approach:* how are retrieval, planning, and writing orchestrated?

*Functional capabilities:* what can the system actually do (parallel browsing, document reading, evidence

synthesis, uncertainty handling...)?  
*Limitations:* what still requires human supervision or cannot be generalized?

*OpenAI – Deep Research (February 2025)*

<https://openai.com/index/introducing-deep-research/> OpenAI+1

System card (PDF): <https://cdn.openai.com/deep-research-system-card.pdf> OpenAI

*Google – Gemini Deep Research (December 2024 → 2025 updates)*

<https://blog.google/products/gemini/gemini-deep-research/> (feature overview)

Technical overview: <https://ai.google.dev/gemini/2.5>

*Hugging Face – Open Deep Research (2025, Open Source)*

Blog: <https://huggingface.co/blog/open-deep-research>

Repo: [https://github.com/huggingface/smolagents/tree/main/examples/open\\_deep\\_research](https://github.com/huggingface/smolagents/tree/main/examples/open_deep_research)

*LangChain – LangGraph / Open Deep Research (2025)*

<https://blog.langchain.com/open-deep-research/> LangChain Blog

[https://github.com/langchain-ai/open\\_deep\\_research?utm\\_source=chatgpt.com](https://github.com/langchain-ai/open_deep_research?utm_source=chatgpt.com)

*Anthropic – Claude Deep Research (2025)*

Blog: <https://www.anthropic.com/news/clause-deep-research>

<https://www.anthropic.com/research/clause-deep-research-system-overview>

### Partial conclusions :

On the model side, Deep Research systems increasingly rely on models specifically tuned for web search and information processing. Think of the shift from GPT-4 to the o3 lineage, where post-training explicitly targets browsing fluency, multi-step planning, and self-reflection. In practice, the planner will invoke more calls to outline and revise a plan, and evaluative steps are inserted to check coverage or consistency. Some advanced variants integrate uncertainty estimation to decide when to branch, retry, or stop, although true confidence scoring generally requires access to model logits and is therefore not always available in hosted settings.

Retrieval also evolved. Early systems leaned on naïve search via a SERP API, which returns ranked links but leaves the heavy lifting to the model. Modern agents perform dynamic web exploration: they open pages, follow links inside them, read structured and unstructured content including PDFs,

and maintain a running cache of extracted facts. The loop is now search, read, pivot, and aggregate, repeated until a coverage or confidence criterion is met rather than a fixed number of hops.

**Table — Comparative Overview of Deep Research Architectures (2025)**

Provider	System Architecture	Implementation Approach	Functional Capabilities	Notable Innovations / Limitations
<b>OpenAI – Deep Research</b>	Multi-step orchestration built around a reasoning-tuned <i>o3</i> model; browsing-based agent with dynamic plan-execute-reflect loop.	Monolithic agent orchestrating multiple internal reasoning passes; web search + page reading + synthesis; runs in a managed sandbox.	Advanced web exploration (opens pages, follows links, interprets text, tables, and images); self-reflective reasoning; citation-based report generation.	Strong vertical integration between model and browser; limited transparency; relies on OpenAI infrastructure; closed-source but stable.
<b>Google – Gemini 2.5 Deep Research</b>	Integrated within Gemini ecosystem; modular components (planner, retriever, summarizer, writer) orchestrated in a workflow.	Multi-agent logic under the hood, especially for private corpus retrieval (Drive, Gmail); hybrid pipeline + dynamic browsing.	Access to both web and private data silos; deep reasoning over heterogeneous corpora; parallel search threads; strong summarization and attribution.	Tight integration with Google Workspace; less openness; retrieval quality depends on ecosystem access and permissions.
<b>Hugging Face – Open Deep Research</b>	Open-source pipeline based on <i>smolagents</i> ; modular ReAct-inspired chain with explicit steps for search, read, and write.	Deterministic workflow with optional parallel subcalls; retrieval via SERP APIs or connectors; emphasis on transparency and extensibility.	Baseline Deep Research loop: search, parse, summarize, synthesize; easily customizable; educational and inspectable.	Serves as a reference for reproducibility; limited scalability; relies on open web APIs; no autonomous planning beyond ReAct loop.
<b>LangChain – LangGraph Deep Research</b>	<b>Explicit graph-based orchestration</b> ; nodes represent planner, retriever, critic, writer; edges define dynamic flow.	Uses <i>LangGraph</i> for stateful, event-driven scheduling; supports branching, waiting, and joining of tasks (true parallelism).	Fine-grained control, observability, checkpointing; supports hybrid workflows (ReAct + graph); strong composability.	Higher engineering complexity; requires explicit graph definition; dependent on LangGraph ecosystem.
<b>Anthropic – Claude Deep Research</b>	<b>Multi-agent cooperative system</b> : several Claude agents explore, verify, and synthesize results in parallel, coordinated by a meta-agent.	Hierarchical orchestration; emphasis on reliability and factual consistency; internal evaluation and redundancy between agents.	Deep parallel exploration; cross-agent consistency checks; confidence estimation; human-readable synthesis reports.	Most advanced architecture; demonstrates effective parallel coordination; proprietary; orchestration logic not yet public.



Taken together, these changes mark an evolution from monolithic ReAct-like or linear pipelines to architectures that are partially multi-agent. Parallel readers, critics, planners, and writers can be composed and scheduled, as in the multi-agent patterns described by Anthropic and echoed in open implementations such as LangGraph and smolagents. In practice, most production systems are hybrid: they keep a simple loop at the node level, but add orchestration, parallelism, and shared state at the system level.

## c) The Struggle for Domain Adaptation

As Deep Research architectures mature, the focus is now shifting from general-purpose search and synthesis to **specialized domains**.

This evolution explains why such agents continue to attract attention today: they are **useful** and can meaningfully automate complex intellectual activities, from dissertation writing and systematic literature reviews to financial consulting, legal analysis, or strategic monitoring.

The goal is not merely to retrieve information, but to adapt reasoning, structure, and tone to the **epistemic norms of a field**.

Adapting a Deep Research agent to a specific domain rests on three main pillars:

- **Topical adaptation** — the agent must become familiar with the knowledge space and terminology of the field to avoid hallucinations, approximations, or anachronisms. A

biomedical research agent must, for instance, distinguish between experimental evidence, meta-analyses, and clinical trials; a legal one must reason through the hierarchy of norms.

- **Format adaptation** — different fields expect specific document structures: an academic paper, a market brief, and a technical report obey distinct conventions. The agent's writing and reasoning must align with these expectations.
- **Intent adaptation** — the same dataset can serve different goals. A historian seeks synthesis and interpretation; an analyst, actionable insight. Domain adaptation thus requires aligning not only the content but also the rhetorical stance and evaluation criteria of the agent's output.

Several complementary techniques can be leveraged to reach effective domain specialization.

### 1. Specialized tooling.

The most direct and robust method remains connecting the agent to **domain-specific retrieval tools**. For instance, a biomedical agent may use PubMed APIs, a legal one may query Legifrance or LexisNexis, while a geopolitical analysis agent may access internal OSINT databases.

The quality of these interfaces largely determines how well the agent can ground its reasoning in verified, up-to-date sources.

### 2. Reinforcement Learning as an Agent Run Optimizer.

Beyond simple retrieval, some research prototypes explore *adaptive control* of agent behavior through reinforcement learning. As illustrated by **Agent-RL/ReSearch** (2025), such systems learn from past runs how to **optimize execution strategies**, adjusting dynamically to intermediate results and recovering from partial failures. This approach highlights how the performance of a Deep Research system depends as much on the *control layer*, its capacity to plan, revise, and adapt, as on the model itself.

### 3. Domain-specific prompting.

Finally, a crucial yet often underestimated lever is the **prompting layer**.

Well-engineered prompts, including style guidelines, reference examples, and domain taxonomies, act as lightweight adapters that frame the reasoning process.

Prompt templates can impose academic style conventions, citation rules, or disciplinary reasoning heuristics (“always cross-check dates,” “differentiate hypotheses from observations,” etc.).

This method mirrors traditional *prompt tuning* or *context conditioning*, but applied at the **agentic orchestration level** rather than within a single LLM call.

### 4. Knowledge base integration and retrieval-augmented memory.

Rather than relying only on external APIs, some systems embed curated **domain knowledge bases** directly into the agent's memory.

This may take the form of vector-indexed corpora (for semantic retrieval), structured ontologies, or even fine-grained fact stores.

Agents can then ground their reasoning in verified information without calling external services.

The key challenge is **governance**: maintaining freshness and preventing the propagation of stale or contradictory facts.

Examples include private graph databases in corporate intelligence and medical-knowledge embeddings in health-tech agents.

## **5. Lightweight fine-tuning and adapters.**

For organizations controlling their own models, **parameter-efficient fine-tuning** (LoRA, QLoRA, adapters) provides a middle ground between pure prompting and full retraining.

A domain-specific adapter can endow the same base model with different expertise profiles — one for legal drafting, another for financial analysis, while sharing most parameters.

This makes multi-domain deployment practical and frugal, and it aligns well with the “agentic layer” paradigm where one model supports many specialized agents.

## **6. Example-based and in-context learning loops.**

Instead of retraining, an agent can **learn from its own past successful runs**.

By storing effective trajectories, reasoning chains, and validated outputs, the system builds an evolving **episodic memory**.

When facing a similar query later, it retrieves relevant exemplars to guide planning and tone. This blends retrieval-augmented generation with self-distillation and yields natural domain adaptation over time — the more the agent works in one area, the better it becomes at it.

## **7. Hierarchical composition of specialized sub-agents.**

Another path is structural: rather than adapting one model, design **ensembles of specialized agents** one for planning, one for retrieval, one for synthesis , each optimized for a specific reasoning skill or data type.The “domain” then emerges from their coordinated expertise.

This approach, visible in Anthropic’s multi-Claude architecture and in open projects like LangGraph, turns adaptation into a question of **organization** rather than fine-tuning.

## **8. User-feedback alignment and continual evaluation.**

Finally, domain robustness depends on feedback.

Some platforms integrate **human-in-the-loop evaluation** directly in agent workflows: users validate intermediate results, rank alternative outputs, or flag misconceptions.

Collected feedback becomes training or calibration data, feeding back into reinforcement loops or prompting adjustments.

This continuous supervision is particularly valuable in professional or regulated fields where human oversight remains mandatory.

## Conclusion :

We are entering a moment where the question is no longer *what* LLMs can do, but *how* we organize them.Agent architectures have become our new programming language , a grammar for building systems that think, act, and adapt. Building agents became a new way to program, that has to handle non determinism and be highly data centric.

We are still far from autonomous intelligence, but closer than ever to a form of engineered cognition systems that remember, delegate, and improve themselves over time.