Chapter 2 : From Prompts to Actions: How AI Agents Leverage Tools

The emerging vocabulary used to describe AI agents is strongly marked by anthropomorphism. The very notion of *tools*, which will be the focus of this chapter, is no exception. It echoes an old philosophical paradigm: that the civilized human being shapes and masters their universe through tools, which enable action, and through language, which enables representation.

A *tool* is most often defined as a material object employed to alter other material objects[1]. In this definition, human beings can themselves be considered material objects, which makes it straightforward to extend the notion of tool to immaterial or digital objects—such as data. Throughout history, the emergence of new tools has been decisive: it has marked the great transitions between ages and revolutions in human development.

A tool can thus be understood as any object that extends an individual's capacity to modify aspects of their environment or to accomplish a particular task. This definition maps directly onto the contemporary intention of broadening the abilities of LLMs: by equipping them with tools, one enables them to act beyond the confines of pure language.

It is no coincidence that the vocabulary of historical and anthropological studies of tool use has been borrowed in AI discourse. Terms such as *multiple tool use* or *tool substitution* now describe patterns of LLM–tool interactions, echoing the ways in which humans have, across millennia, learned to combine, replace, and refine the implements of their own agency.

While the notion of *tool* refers to a concrete object that extends the capacity of an individual, *technology* is generally understood in a broader sense. Technology encompasses not only the tools themselves but also the knowledge, methods, and organizational frameworks that make their use possible. A stone blade is a tool; the set of practices, techniques, and cultural transmission that allowed humans to produce, sharpen, and wield blades constitutes a technology.

This distinction matters for our discussion of AI agents. When we speak of "tool use" in LLMs, we are deliberately emphasizing the **extension of capacity**: an LLM calling an API to retrieve data, or executing a function to parse a file, corresponds to the anthropological sense of an agent wielding a tool. But when we speak of *technologies*, we are situating these tools within the broader engineering frameworks—workflow engines, orchestration systems, memory management—that make their repeated and reliable use possible.

At their core, **Large Language Models (LLMs)** are sequence predictors: they are trained to guess the most probable next token (word, subword, or character) given a sequence of previous ones. Over time, these sequence predictors have been turned into what many now perceive as *task completion systems*. By carefully shaping prompts, or by fine-tuning them with specialized methods, we have been able to transform raw prediction engines into tools that appear to follow instructions and complete tasks.

1 https://www.jstor.org/stable/2575191

It is essential, however, to separate what LLMs truly are from the misunderstandings that surround them. A common misconception, for example, is to treat LLMs as **search engines**. They are not. Unlike a search engine, which queries a structured index of documents, an LLM generates new text based on statistical patterns learned from its training corpus. When it "retrieves knowledge," it is in fact reproducing patterns stored in its parameters, not accessing an external database.

To use LLMs properly, one must understand both their **capabilities** and their **limitations**:

What LLMs can do, and why

- **Text completion**: this is their fundamental task, predicting the next token.
- **Instruction following**: this ability stems from the *alignment phase*, where raw models are adapted to follow human-like instructions using methods such as supervised fine-tuning (SFT), reinforcement learning with human feedback (RLHF), or more recent approaches like GRPO.
- **"State of the world" alignment**: models reflect information encoded in their training corpus, which they store in their parameters in ways we do not yet fully understand. This gives the impression of knowledge, though it is really statistical memory.
- **Code generation**: LLMs are particularly strong at writing code, largely because their training data includes vast quantities of structured, well-formatted, and repetitive code snippets, which map well to the predictive strengths of the architecture.

Despite their impressive capabilities, LLMs have well-known limitations :

- **Context size**
  LLMs are bound by their context window: the maximum number of tokens they can process at once. This is a computational constraint. While context windows have grown (from a few thousand to hundreds of thousands of tokens), they are still finite. This makes it difficult to handle very long documents or multi-step reasoning without specialized memory mechanisms.
- **Hallucinations**
  LLMs sometimes produce answers that are fluent and convincing but factually false. This occurs especially in *out-of-domain situations* or when asked about *recent events* not present in the training corpus. The deeper issue is that an LLM cannot evaluate whether a prediction is *true*—it only knows whether it is *probable*. There is therefore a fundamental tension between **probability** (what is statistically likely) and **truth** (what is factually correct).
- **Weakness with numbers**
  LLMs are notoriously bad at arithmetic and symbolic calculation. This stems from the way numbers are tokenized in training. For instance, a model might happily produce nonsense such as: « `39 + 32 = 611` » Some progress has been made (Anthropic, for example, enforced single-character tokenization of digits in their models, which improved arithmetic performance), but the weakness remains.
- **Poor symbolic reasoning**
  LLMs also struggle with abstract reasoning, logical puzzles, and symbolic manipulation.

They can simulate reasoning patterns but cannot consistently maintain the invariants required for rigorous symbolic work (e.g., algebraic transformations, formal proofs).

- **Lack of meta-awareness**
  Humans writing text have an awareness of the “meta” characteristics of their output: length, structure, style, argument flow. LLMs, by contrast, have no stable self-representation of what they have already produced. They cannot check for redundancy, coherence, or formal consistency in the same way a human writer can.
- **Code generation without verification**
  LLMs can generate large amounts of code, but they cannot execute or test their own outputs. Unlike a developer working in VS Code, who can run, debug, and iterate, an LLM cannot check whether its code compiles or runs correctly
- In the ends, « words are wind[2] » and LLM cannot achieve much without being connected to a broader infrastructure.

The shortcomings of LLMs are not insurmountable. Three broad strategies have emerged to address them, each working at a different level of the system:

1. **Improve the models themselves**
   The first approach is to change how LLMs are trained in the broadest sense. This can include enlarging and curating the training corpus, refining tokenization schemes, modifying architectures, or improving inference methods. Such changes aim to directly enhance the intrinsic performance of the models.
2. **Improve how LLMs are integrated**
   A second strategy is to design system architectures that mitigate weaknesses rather than expecting the model alone to solve them. Retrieval-Augmented Generation (RAG) is a key example, reducing hallucinations by grounding responses in external knowledge bases. Other techniques include prompt chaining, output validation, and feedback loops that structure and monitor LLM use.
3. **Extend the function signature of LLMs**
   At their core, LLMs are functions that take as input raw text (plus sampling parameters or conditions) and return a string—usually just content. By modifying this *signature*, one can expand their capabilities (we’ll explore this is section I)

One of the most striking aspects of LLMs is that some of their most useful abilities were not explicitly designed, but *emergent*. Their capacity to follow instructions without task-specific fine-tuning, for instance, was an unexpected by-product of the complexity of their training process. Once observed, these capabilities were quickly exploited to outperform the state of the art on multiple NLP benchmarks, and were then gradually integrated into industrial applications.

More recently, we are witnessing the reverse dynamic. The **limitations observed in industrial use**are now driving new advances in the models themselves. In other words, operational bottlenecks have begun to shape how LLMs are trained and fine-tuned, leading to architectural and methodological changes.

---

2 George Martin. Or, in a french version « Parole, paroles, paroles » , Dalida.

It is important to note that these innovations largely stem from **corporate research labs**. While industry actors such as OpenAI, Anthropic, Google DeepMind, or Meta regularly introduce new training methods in response to practical limitations, only a small fraction of this knowledge is formally published. Academic work on the subject remains limited, and much of the detail about how industrial LLMs overcome these challenges remains undocumented or opaque.

In this class we will focus on the development of LLM tooling capabilities, talk about their main potential and limitations and study the recent coding practices used to connect LLM and external capabilities.

I) Controled Generation : From Structured Output to Tool use

Up to now, we have looked at LLMs as text generators with remarkable but fragile capabilities. If we want to turn them into reliable components of software systems, however, we need more control. This brings us to the idea of **constrained generation**, the first step on the path from free-form text toward structured outputs, and eventually toward tool use.

a) Constraining Generation for better software integration. [3]

One of the main limitations of classical LLMs, when it comes to integrating them into downstream applications, is that they are not deterministic. You feed variables into a prompt template, obtain a text string as output, and hope it matches your needs. If the expected output is itself free text—for example, a summary—this is acceptable. But if the application requires a specific, well-formatted answer, things become more complicated.

Traditionally, models were built to behave like functions: for sentiment analysis[4], for example, one could call a dedicated classifier that returned one of a fixed set of outputs (POSITIVE, NEGATIVE, NEUTRAL). The program's logic then routed accordingly. By contrast, an LLM will return a full string—often with justifications, explanations, or additional commentary.

In the "early days" of LLM adoption[5],, developers adapted prompts to force structured outputs. They might ask explicitly: *"Answer only with POSITIVE or NEGATIVE."* They would then parse the response with a regular expression. But this was brittle: if the justification contained both words, the parser could be confused. A different approach was to request a JSON object as output. This improved matters, but older open-source models often produced malformed JSON, or ignored the schema altogether.

While these strategies worked most of the time, they introduced **non-deterministic failures** into otherwise robust software pipelines. No developer wants their program to crash because, once in a while, the model forgets a closing bracket. As a result, systems had to rely on inelegant "try-and-catch" mechanisms, adding complexity and fragility.

---

3 I recommend this course, https://learn.deeplearning.ai/courses/getting-structured-llm-output/lesson/cat89/introduction, for a well crafted different perspective.

4 Please, I beg you, do not be among these people that use LLM for trivial binary classification tasks while it has been scientifically proven that traditionnal classifiers are much more reliable, without even mentionning frugality.

5 I'm talking about 2023

This is why LLM providers began to work on ways to **constrain generation deterministically**. Fortunately, some "old-school" mathematics—long known in computational linguistics and automata theory—turned out to be extremely useful here.

A study led by Google research led accross more tha n 50 industry professionals highlighted in April 2024 that constraining LLM output was crucial[6]. The notion of structuration is granular. It could be constraints about length, simple constraint about a format (json) or harsch constraing requiring a json that exactly follows the schema given in input.
The paper distinguished between low level constraints, form based only, and high level constraint focused on style and semantics. We will mainly focus on the first category.

Category		%	Representative Examples	Precision
Low-level constraints				
Structured Output	Following standardized or custom format or template (e.g., markdown, HTML, DSL, bulleted list, etc.)	26.1%	*"Summarizing meeting notes into markdown format"* *"... the chatbot should quote dialogues, use special marks for scene description, etc."* *"I want the output to be in a specific format for a list of characteristics [of a movie] to then easily parse and train on."*	Exact
	Ensuring valid JSON object (with custom schema)	16.4%	*"... use the JSON output of the LLM to make an http request with that output as a payload."* *"I want to have the output [of the quiz] to be like a json with keys* `{"question": "...", "correct_answer": "...", "incorrect_answers": [...]}`*"*	Exact
Multiple Choice	Selecting from a predefined list of options	23.9%	*"Classifying student answers as right / wrong / uncertain...)"* *"While doing sentimental analysis, [...] restrict my output to few fixed set of classes like Positive, Negative, Neutral, Strongly Positive, etc."*	Exact
Length Constraints	Specifying the targeted length (e.g., # of characters / words, # of items in a list)	16.4%	*"... Make each summary bullet LESS THAN 40 words. If you generate a bullet point that is longer than 40 words, summarize and return a summary that is 40 words or less."* *"I want to limit the characters in the output to 100 so it is a valid YouTube Shorts title."*	Approx.
High-level constraints				
Semantic Constraints	Excluding specific terms, items, or actions	8.2%	*"Exclude PII (Personally Identifiable Information) and even some specific information..."* *"If asking for html, do not include the standard html boilerplate (doctype, meta charset, etc.) and instead only provide the meaningful, relevant, unique code."*	Exact
	Including or echoing specific terms or content	2.2%	*"... I want [the email] to include about thanking my manager and also talk about the location he is based on to help him feel relatable."* *"We want LLM to repeat input with some control tokens to indicate the mentions. e.g. input: 'Obama was born in 1961.',... , we want output to be '«Obama» was born in 1961.'"*	Exact
	Covering or staying on a certain topic or domain	2.2%	*"[The output of] a query about 'fall jackets' should be confined to clothing."* *"For ex. In India we have Jio and Airtel as 2 main telecom service provider. While building chat bot for Airtel, I would want the model to only respond [with] Airtel related topics."*	Exact
	Following certain (code) grammar / dialect / context	4.5%	*"While generating SQL,... restrict the output to a particular dialect and use the table / database name mentioned in the prompt."* *"... implement[ing] a voice assistant that calls specific methods with relevant arguments,... the output needs to be valid syntax and only call the methods specified in the context"*	Exact
Stylistic Constraints	Following certain style, tone, or persona	6.7%	*"... it is important that the [news] summary follow a style guide, ... for example, preference for active voice over passive voice."* *"Use straightforward language and avoid complex technical jargon..."*	Approx.
Preventing Hallucination	Staying grounded and truthful	8.2%	*"... we do not want [a summary of the doc] to include opinions or beliefs but only real facts."* *"If the LLM can't find a paper or peer-reviewed study, do not provide a hallucinated output."*	Exact
	Adhering to instructions (without improvising unrequested actions)	4.5%	*"For 'please annotate this method with debug statements', I'd like the output to ONLY include changes that add print statements... No other changes in syntax should be made. "* *"LLMs usually ends up including an advice associated to the summarised topic, advice we need to avoid so they are not part of the doc."*	Exact

Over time, "structured output" became a feature integrated directly into LLM inference solutions. At a macroscopic level, structured output can be defined as a feature that forces the model's response to strictly follow a schema specified by the user—most often expressed in JSON.

To the best of my knowledge, OpenAI was the precursor in this area, first introducing what they called *"JSON mode"*. In this early version, the developer had to provide a schema and also explicitly mention in the prompt that a JSON object was expected as output. However, only the later

6 https://arxiv.org/pdf/2404.07362

"structured output" feature of their API truly ensured schema adherence. This new version is significantly more powerful and more reliable.

While all major LLM providers now offer some form of structured output, very little is explained about how these results are achieved. The official documentation, however, hints at interesting possibilities. For instance, for fields that are strings, one can specify a `pattern`—a regular expression that the output must match—or a required format. For numbers, one can impose thresholds or ranges. These constraints give developers new levels of control that were previously absent.

The limitation, however, is clear: when relying on proprietary APIs, we do not know what is happening behind the scenes. The mechanisms ensuring schema adherence remain opaque, and this lack of transparency complicates both reproducibility and research.

Constrained text generation is not a new subject in NLP. It was already a popular research topic in the transformer era, with important contributions from scholars such as Claire Gardent[7]. With the advent of LLMs, these older ideas have resurfaced, and several techniques are now used to obtain exploitable results.

1. Reprompting approaches

The most straightforward method is *reprompting*. The model first generates output as usual. A validator then checks whether the output conforms to the expected schema. If it does not, the system diagnoses why it failed, reformulates the prompt to include the error, and retries. While simple and effective in many cases, this approach has two major drawbacks:

- Each retry increases computational cost.
- There is no guarantee of eventual success, particularly if the model persistently misunderstands the schema.

2. Logic-based approaches

By contrast, logic-based methods—sometimes called *structured generation*—intervene directly at the point of token generation. Instead of letting the model freely sample from the entire vocabulary, these methods dynamically modify the logits: invalid tokens are suppressed, and the probabilities of valid ones are renormalized. This ensures that the model can only produce outputs consistent with the defined structure.

Libraries such as **Outlines**, **SGLang**, or **Guidance** implement this approach. The idea is not new—similar techniques have been explored since 2019—but LLMs have renewed interest in them. The advantages are significant:

- Robust adherence to schemas, outperforming methods based on supervised fine-tuning on JSON corpora.
- Flexibility to enforce not only JSON structures but also more complex constraints, such as regular expressions or custom grammars.

However, there are limitations:

- Because this method requires tampering with logits at each decoding step, it is only possible with **open models** where the decoding process is fully accessible.

7 https://members.loria.fr/CGardent/newpublications.html

- The model will adhere to the schema even when the input is entirely unrelated, which can produce *schema-conforming hallucinations*.
- Results can sometimes be inconsistent, and the overall impact on the quality of generated text is not yet fully understood.

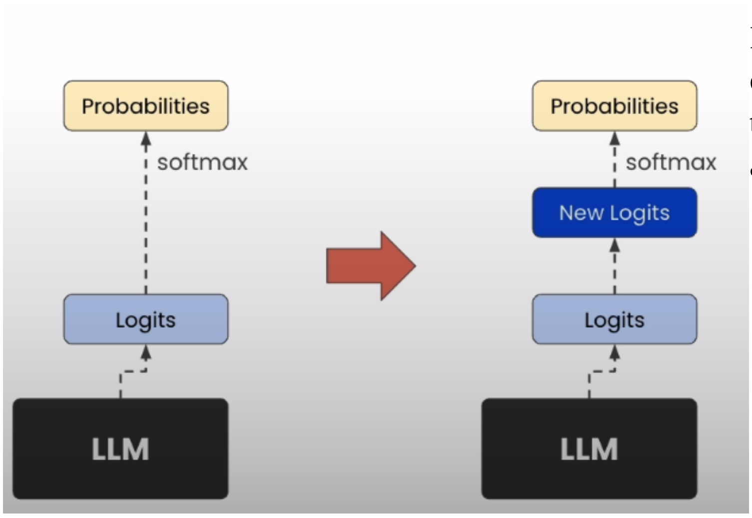
To give you a better understanding on how constrained generation works, best is to present it as a decoding strategy.

Decoding Strategies in LLMs

Strategy	How it works	Advantages	Drawbacks / Limits
Greedy decoding	Always selects the most probable next token.	Deterministic, simple, very fast.	Produces repetitive, dull text; no diversity.
Beam search	Keeps several candidate sequences ("beams") at once, expands them step by step.	More diverse than greedy; often higher-likelihood outputs.	Computationally expensive; still tends toward generic/safe completions.
Temperature sampling	Scales logits by a *temperature* parameter before sampling.	Allows tuning between determinism (low T) and creativity (high T).	High T can cause incoherence; low T collapses back to greedy.
Speculative decoding	Uses a smaller "draft" model to propose tokens, then verifies them with the main model.	Greatly speeds up inference; reduces latency for large models.	More complex pipeline; requires two cooperating models.
Constrained generation (regex, grammars, JSON schemas)	Invalid tokens are suppressed at decoding (logits modified to enforce structure).	Guarantees schema adherence; enables structured output, tool use, valid JSON.	Only possible with open models or special APIs; risk of schema-conforming hallucinations.

It is worth noting that **Top-k** and **Top-p (nucleus) sampling,** that you probably know by now, are already forms of *constrained generation*. At each decoding step, these methods restrict the candidate tokens to a subset—either the *k* most probable tokens (Top-k) or the smallest set whose cumulative probability is above *p* (Top-p). In both cases, the model's logits are modified to remove invalid options and renormalize the probabilities of the remaining ones.

This makes Top-k/Top-p a lighter, probabilistic form of constrained generation. By contrast, modern structured decoding (using regexes, grammars, or JSON schemas) enforces *hard constraints* on output structure, guaranteeing validity at the cost of flexibility.



In **constrained generation**, a more complex algorithm is applied directly to the logits at each decoding step. The algorithm takes as input three elements:

1. the **constraints** specified by the user (for example a JSON schema, a regular expression, or a grammar),
2. the **logits** produced by the model for the next token,

3. and the **prefix already generated**.

Algorithm 1 Constrained Decoding

Require: Constraint C, LLM f, Prompt x
Ensure: Output o adhering to C

```
o ← []
loop
    C.update(o)            ▷ advance state of C
    m ← C.mask()           ▷ compute mask
    v ← f(x + o)           ▷ compute logits
    v′ ← m ⊙ v′
    t ← decode(α′)         ▷ sample
    if t = EOS then
        break
    end if
    o.append(t)
end loop
return o                   ▷ output
```

It then computes a new probability distribution by **removing all tokens that would violate the constraints** and **renormalizing the remaining logits** so that they sum to one. This algotirhm changes the logits themselves, not the sampling process.

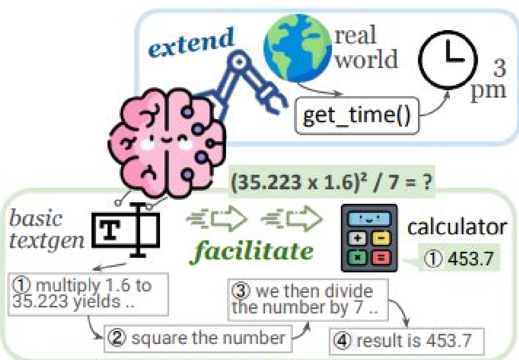
There are deep connections between regular expressions and finite-state machines (FSMs[8]). A regular expression is essentially a compact, declarative way of describing the same set of sequences that can be recognized (or generated) by a finite-state automaton. This equivalence means that any regex constraint we impose on an LLM's output can be compiled into a finite-state machine that guides token generation.

In practice, constrained generation systems often exploit this: the regex provided by the user is first transformed into an automaton, which can then be used to filter the model's logits step by step. This ensures that the sequence of tokens generated is always a path through the automaton—that is, always matches the regex. The same logic can be extended beyond regular languages. For more complex structures such as **context-free grammars (CFGs)**, which are essential for coding tasks or generating structured documents like JSON, XML, or programming langages. This is sometimes referred to as **grammar-constrained decoding** or **highly optimized grammar decoding .T**his theoretical link—regex → finite-state machines → grammars—provides the formal backbone for modern structured generation, which in turn enables robust integration of LLMs into downstream applications requiring deterministic, well-formed outputs.

b) from Structure Output to Tool Use

LLMs are capable of generating text and performing a number of tasks correctly, such as summarization and, for some languages, translation. But they quickly reach their limits on subjects such as mathematical reasoning or answering questions that require knowledge beyond their training.

This is where the idea arises of complementing generative capabilities with other modules. For example, if an LLM struggles to solve complex calculations, a calculator or a Python program can handle them with no difficulty. The concept of a *tool* therefore simply consists in interfacing a text-



8 Yes, this M1 class everybody feared

generating LLM with other computational functions that can accomplish what the model itself cannot do reliably.

And the easiest way to run capabilities on a computer is to run code. A code function takes as input a list of parameters (well typed and described if the programmer did a good job) and turns it into an expected output. Coding functions are efficient for many tasks such as computation , information retrieval… To sum things up all the capabilities one would like to give to an LLM.

Category	Example Tools

Knowledge access	`sql_executor(query: str) -> answer: any`
	`search_engine(query: str) -> document: str`
	`retriever(query: str) -> document: str`

Computation activities	`calculator(formula: str) -> value: int | float`
	`python_interpreter(program: str) -> result: any`
	`worksheet.insert_row(row: list, index: int) -> None`

Interaction w/ the world	`get_weather(city_name: str) -> weather: str`
	`get_location(ip: str) -> location: str`
	`calendar.fetch_events(date: str) -> events: list`
	`email.verify(address: str) -> result: bool`

Non-textual modalities	`cat_image.delete(image_id: str) -> None`
	`spotify.play_music(name: str) -> None`
	`visual_qa(query: str, image: Image) -> answer: str`

Special-skilled LMs	`QA(question: str) -> answer: str` `translation(text: str, language: str) -> text: str`

Table 1: Exemplar tools for each category.

The notion of *LLM tool usee*[9] was initially little more than a trick derived directly from structured output. A programming function has a name and requires a set of arguments to run correctly. If the LLM is given, in its system prompt, some information about these functions (for example, their docstrings), it can be prompted to output a dictionary containing the information required to execute the function, such as:

```
{
  "function_name": "...",
  "kwargs": { }
}
```

The LLM’s answer can then be parsed, the arguments extracted, and the function executed. The real challenge lies in *making the LLM aware* that it has tools at its disposal and in guiding it to use them effectively. The tools themselves may serve a wide variety of purposes, though the most common are those that allow LLMs to query information—whether on the web or in external databases.

Tool learning refers to the process that “aims to unleash the power of LLMs to effectively interact with various tools to accomplish complex tasks” [10]. In other words, it is the systematic way of teaching LLMs not only to generate text but to plan, select, and call external functions in a purposeful manner.

The process of tool learning is often described in four stages:

1. **Task planning** – The task is analyzed to better clarify user intent and propose a workable plan.

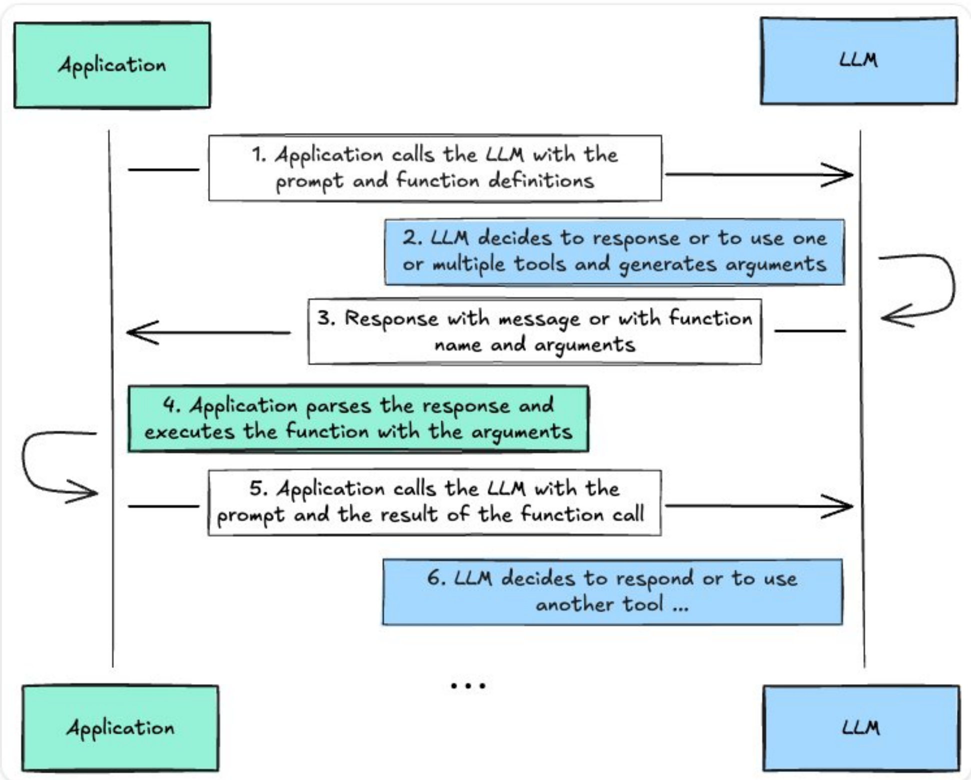
9 https://arxiv.org/pdf/2405.17935
10 https://arxiv.org/pdf/2307.16789

2. **Tool selection –** The most relevant tool is chosen. This can be done by the LLM itself, but in early experiments the number of tools was often too large, so retrieval-based approaches were used to narrow the set of candidates.
3. **Tool calling –** The chosen tool is executed with arguments generated by the LLM.
4. **Answer generation –** A final response is produced that integrates the tool's output into the conversational context.

In 2023 lots of different variants variants of this architecture were proposed. This was only made possible by robust schema adhering generation. Many works focused on how to improve LLM Tool use capabilities and efforts focused on all of these four stages yielded benefits. However it was at the cost of expensive fine tunings and adaptations only reproductible on specific use cases. Moreover, the whole mechanic was heavy to implement. It is only when tool calling was incorporating during LLM training that real standardization happened.

3) Function calling as a new LLM capability

In June 2023, OpenAI introduced a new function-calling[11]) capability in the Chat Completions API, as an extension of the structured output. This was an applicative formalization of the endeavours in scientific literature to connect LLM and tools. But how does it work exactly ?



11 https://platform.openai.com/docs/guides/function-calling?api-mode=chat désolé je n'arrive pas à retrouver le lien de l'illustration que j'ai utilisée.

The user must add to the completion code a new parameter that corresponds to the **docstring** of the functions he wants the LLM to have access to. This has to be given as a correct json/pydantic schema :

```
tools = [
    {
        "type": "function",
        "name": "get_horoscope",
        "description": "Get today's horoscope for an astrological sign.",
        "parameters": {
            "type": "object",
            "properties": {
                "sign": {
                    "type": "string",
                    "description": "An astrological sign like Taurus or
Aquarius",
                },
            },
            "required": ["sign"],
        },
    },
]
```

The idea is to make the model aware of the list of tools available to it by explicitly providing them in the system prompt. One generally gives the equivalent of the function's docstring (a description, the list of arguments and their types, and the type of the expected output). It is then automatically added to the system prompt between specific tokens, unbeknowst to the user.

The model then decides whether it needs to use one of these tools or simply respond normally.

For example, if an LLM has access to tools named `get_weather(location)` and `play_song(song_name)`, and the user asks "What is the weather in Paris?", the model should decide to execute `get_weather(Paris)`. On the other hand, if the user asks for the founding date of INALCO, it should be able to answer normally without calling a tool.

For the function calling mechanism to work, the model undergoes specific fine-tuning that trains it to choose the right tool in a variety of scenarios. If the model decides to use a tool, it usually returns a JSON object indicating the name of the tool to be used and the arguments to pass, or sometimes uses special tags—though the exact format varies from one model to another.

Thus, when calling an augmented LLM capable of function calling, several things happen:

- If the LLM decides not to use a tool, it answers in the usual free-text format.
- If it decides to use a tool, then (depending on the model) it responds in a structured format that specifies the tool name and any arguments inferred from the context.

- On the application side, the system that invoked the LLM must parse the response to determine whether a tool call was chosen. If so, the call is executed by the application, not by the LLM itself.
- The result of the tool call is then added to the context as an *observation* and fed back to the LLM. The LLM is called a second time, and may either choose to call another tool or provide a final answer based on the observation.

One of the prerequisites for function calling is that the LLM model must be fine-tuned to detect when a function needs to be called, What we call *function calling*, and which allows LLMs to be augmented with tools, is therefore nothing more than a mechanism that effectively transforms the LLM into a **router**, deciding *when* a tool should be executed. The execution itself is handled by the application, not by the model.

The reliability of this mechanism was made possible by specific fine-tuning of the models. This implies that:

- The formatting conditions must be strictly respected (presentation of tools in the system prompt with specific tags), otherwise the mechanism fails.
- Each model has used a different format, which means prompts must be adapted for each model. This is inconvenient, and utilities such as those provided by Hugging Face are often used to automate format conversion as much as possible

It was **not a new capability baked into the base GPT-4 or GPT-3.5 models themselves**, but rather an **API-level innovation**. The base models already had the ability to **emit JSON-like structures** if prompted correctly. What OpenAI added in June 2023 was an **API wrapper around the models** that let developers declare a set of available functions (name, description, argument schema), instructed the model (through hidden system prompting and specific tags) to output only a JSON object matching one of these schemas, and automatically validated the output against the schema before returning it to the developer.

So the "function calling" feature was, rather than a fundamental change to the neural weights of GPT-4/3.5. at the start :

Prompt engineering + schema enforcement + API orchestration,

Later, providers extended this idea into **structured outputs** (forcing adherence to JSON schemas, regexes, etc.), sometimes with deeper integration into the decoding process. But the first widely-used "function calling" in GPT-4 was **an API-level feature built on top of the existing models**.

The base models hadn't changed — they were simply very good at emitting structured text when prompted correctly. What made the feature robust was a validation phase that happened just after generation.

So from a chonological point of view, structured output happend after (late 2023) function calling, as an extension of the same intuition. Other providers quickly adopted and extended the idea in early 2024 and even refined it (guidance and grammar constrained generation). Newer models (Anthropic's Claude 3.5, OpenAI's GPT-4.5, Google Gemini 1.5) show signs that structured output is now trained into the models themselves, not just bolted on via API wrappers. Providers mention “native JSON mode” and “guaranteed schema adherence”, suggesting that during training or fine-tuning, models were now explicitly optimized to generate valid structured outputs.

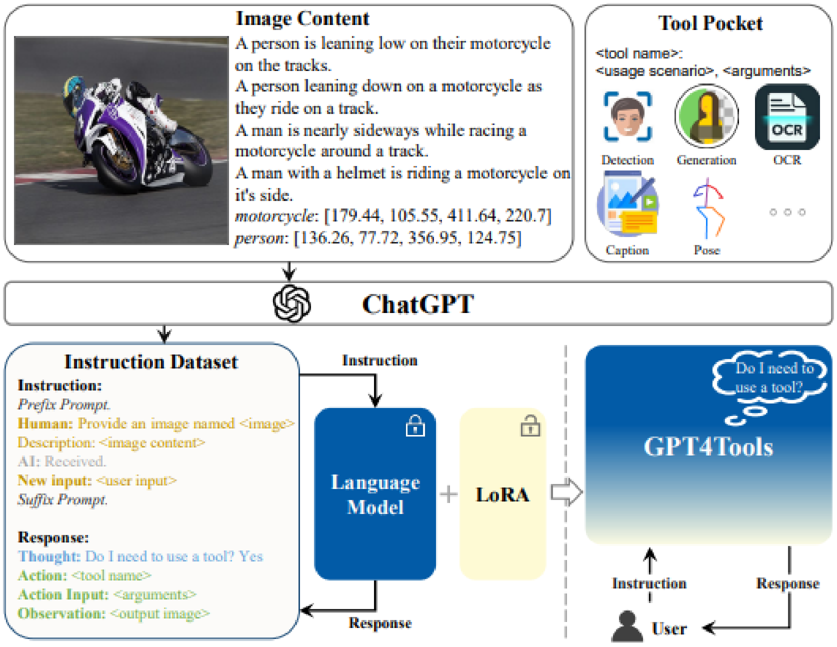


Figure 1: Diagram of the GPT4Tools. We prompt the ChatGPT with image content and definition of tools in order to obtain a tool-related instruction dataset. Subsequently, we employ LoRA [38] to train an open source LLM on the collected instruction dataset, thus adapting the LLM to use tools.

Indeed, by using LLM fine tuning techniques, Low Rank Adaptation (LoRA), researchers managed[12] to significantly boost the performance of « small » 8B langage models (such as the ones derived from the Llama familly using an instruction corpus.

Tool use is an intersting example on how an applicative strategies can in the long term affect LLM capabilities (from hidden application layer to core model feature that influenced training procedure).

II) Exposing Tools, Understanding and Limitations

We have seen how LLMs can be taught to recognize when tools would help them accomplish their assigned tasks, and how to trigger these tools through function calling. But so far, the tools we have considered were hypothetical. Function calling itself was trained and perfected on tool *descriptions*—JSON or Pydantic objects, akin to docstrings—summarizing the transformation (inputs and outputs) that each tool is supposed to realize. In this setting, the LLM chooses the tool, while the responsibility for executing it lies with the application.

Yet this seemingly simple “ping-pong” communication that we call function calling is based on many assumptions:

- That the tool in question actually exists and is available to be executed by the application (in practice, one can test function calling with purely fictitious functions).
- That the tool, once called, behaves exactly as expected.
- That the LLM has enough information to infer the correct arguments and use the tool appropriately.

12 https://arxiv.org/pdf/2305.18752

- That the tool offered to the LLM is indeed relevant to the task at hand (the LLM chooses *which* tool to call, but the engineer decides *which* tools the LLM has access to in the first place).

Developing and exposing tools that meaningfully correspond to the tasks the LLM must accomplish quickly becomes the new bottleneck in expanded GenAI applications. It is exciting to be able to execute tools—but this presupposes that relevant tools actually exist and are well designed.

This points us toward the notion of **tool design**, which in my opinion remains severely underexplored in the industry. Tool design raises several key questions:

- What exactly counts as an *LLM tool*, as opposed to other types of software functions or APIs?
- How should tools be designed to be robust and reliable when called automatically by a model?
- How can tools and their descriptions be optimized to improve LLM results (that is, through extrinsic evaluation of tool capabilities)?

As the saying goes: *with rusty tools, even the best carpenter builds poor roofs.*

1) Controversy Over Code

At the beginning, tools were conceived very much in the image of **code**: a series of machine-readable instructions designed to transform an input into an output for further processing. This coding heritage is visible in the structure of tool descriptions, standardized early by OpenAI, which closely resemble the docstring of a well-written function.

To be valid, a tool must contain at least three elements:

- a **name**,
- a **description**,
- and a list of **inputs**, each with a type and its own description.

The initial idea behind tool use was therefore straightforward: expose a collection of coded functions from the codebase as "tools." Developers would manually convert the function's docstring into the expected JSON format, then write a few dozen lines of code to inject this tool metadata into the model's `messages` list. After parsing the LLM's response, executing the tool was trivial using basic functional programming patterns:

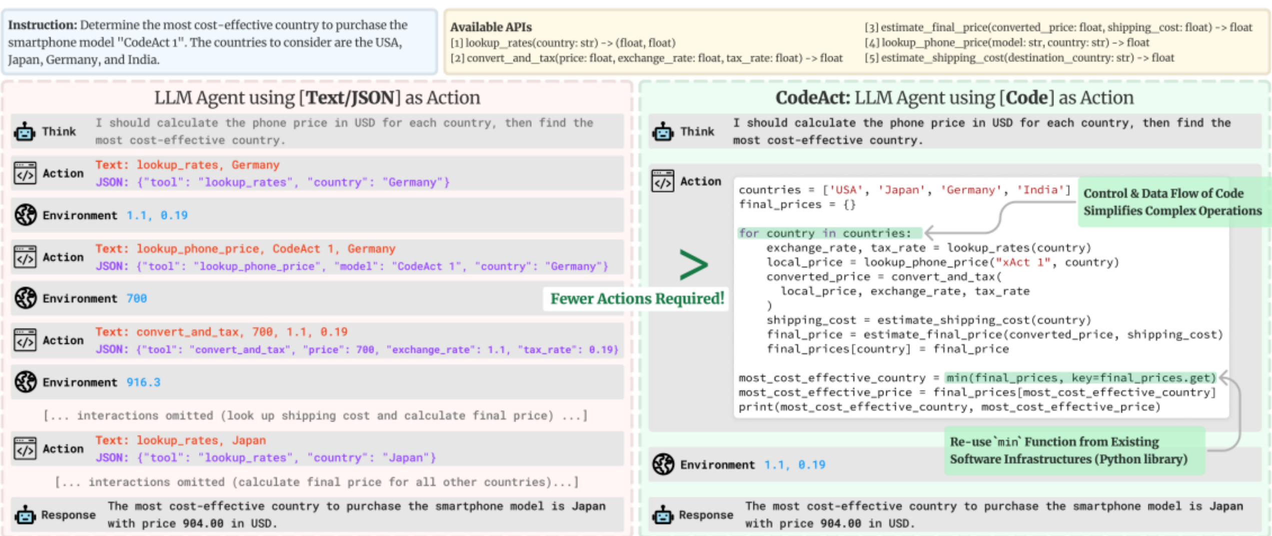
```
response = llm_client.generate(messages, tools=tools)
if response.function_calls:
    for tool_call in response.function_calls:
        tool_called = tool_call["function_name"]
        tool_result = name_2_tools[tool_called](**tool_call["arguments"])
```

Early AI agent frameworks built on this intuition by making the process easier. Instead of manually crafting JSON schemas, they allowed developers to turn ordinary Python functions into **Tool objects** with a simple decorator. Frameworks like *smolagents*, for example, automated the mechanics of function calling, freeing developers from boilerplate and letting the model focus on deciding *when* to invoke the tool.

On top of several limitations we will explore later, a comment arise : if the tool calling mechanism in the end connects LLM to code execution, why do we need to use a structured format as intermediary since LLM show very good capabilities in code generation ? It is indeed possible to ask the LLM, when it needs to use a tool, to generate executable Python code. This strategy—adopted by Hugging Face[13], for example—has several advantages:

- It allows multiple tool calls (e.g. `get_weather(Paris)` and `get_weather(Marseille)`) to be made in a single LLM call.
- The total latency of the system can be reduced by up to 30%.
- It avoids errors in JSON parsing and inconsistencies in argument names.
- It increases the success rate by limiting the risk of error propagation across intermediate steps.
- Tool use is no longer limited to the few tools explicitly defined with their expected format; instead, Python libraries can be flexibly used to optimize input transformations for tools.
- Using code makes it easier to store intermediate steps (in variables) and pass them from one stage of reasoning to the next.
- There is a much larger amount of training data available for examples of code accomplishing tasks than for examples of LLMs generating JSON for tool calls.
- The traceback from code execution makes it possible to correct potential errors without human supervision.

While this approach is promising for creating AI agents that are more flexible and adaptable, one must take into account the risks of executing code generated by an LLM and implement safeguards (e.g. limiting access to certain libraries, executing the code in a sandbox environment without access to the file system).



These mechanisms of tool use—enabled by adapting models to encourage tool invocation, and especially by agent architectures that actually execute the tools—allow agents to observe and interact with their environment. However, it is still necessary to give the LLM visibility over the actions already carried out, which implies carefully managing the context provided to the model for generation.

b) On Tool Clarity

In theory, it is possible—using either the function-calling mechanism or code execution proxies—to expose any function written in the programming language of your choice as a tool accessible to the LLM. This makes it very tempting for software engineers to unleash the model on their entire codebase, allowing the LLM to orchestrate functions that were originally written for earlier workflows.

In practice, however, this approach is almost guaranteed to fail. **LLM tools differ in nature from ordinary code functions.** A function written for human developers is not necessarily a good tool for an LLM. Several important principles should be kept in mind when designing and exposing tools for LLMs:

Limit the number of tools available.

It has been shown that LLMs cannot handle too many tools at once; in practice, ten tools is often a hard limit. This constraint initially came from context window size, but it also reflects semantic fuzziness: when too many tools overlap in purpose (for example, multiple retrieval tools), it becomes much harder for the model to select the correct one.

Make tool descriptions explicit.
Tool design is very similar to prompt engineering. The tool's name, its description, and the argument names must be explicit enough for the LLM to clearly understand what the tool does and how to fill in the arguments. Providing short examples of usage (*few-shot tooling*) can greatly improve reliability. Of course, this requires a level of documentation that few developers apply to their own code.

Provide all necessary information for execution.
Many code functions rely on hidden environment variables (such as URLs, API keys, or workflow-specific parameters). But the LLM does not—and should not—have direct access to these. The more parameters the model has to guess or fill in, the higher the risk of error propagation. Robust tools should minimize unnecessary parameters and abstract away environment-specific details.

Design for error handling and feedback.
Tools should return errors in a structured and informative way, rather than crashing silently or producing ambiguous outputs. These error messages can be re-injected into the LLM's context, allowing it to retry with corrected arguments or to gracefully report failure. Good error handling is essential to prevent cascading failures across an agent's workflow

Tool design is a key step in ensuring Ai Agents will run smoothly and yield the expected outcome. Since frameworks now can handle most of the difficult backend intricacies, tool is what necessitate most work from AI Agents developpers. They need careful design in description and bencharking, as highlighted by Anthropic[14].

❌ **Bad Tool Design**

```
json<br>{<br> "name": "weather",<br> "description": "Gets weather.",<br> "parameters": {<br> "loc": "string"<br> }<br>}
```

✅ **Good Tool Design**

```
json<br>{<br> "name": "get_weather",<br> "description": "Retrieve the current weather for a given city. Use this tool when the user asks about weather conditions.",<br> "parameters": {<br> "location": { "type": "string" "description": "Name of the city (e.g. 'Paris')" },<br> "unit": { "type": "string", "enum": ["celsius", "fahrenheit"], "description": "Temperature unit, default is celsius" }<br> },<br> "errors": {<br> "404": "City not found",<br> "503": "Weather service unavailable"<br> }<br>}
```

c) Where the hell do we run this ?

Another important limitation of treating tools purely as code functions appears at the moment of deployment. Frameworks make it relatively easy to build agents and expose tools on a single development machine. But deploying an agent in production requires that the **tool code be shipped alongside the agent code**.

This becomes problematic when tools depend on heavy or sensitive parts of a proprietary codebase. For example, a retrieval tool in an Agentic RAG system may need direct access to the company's internal database. Packaging that tool with the agent can lead to duplication of proprietary code or sensitive parameters across multiple instances, and can also create **complex dependency management issues** (for example when using Docker or other containerized environments). It gets even more tricky when you want to share tools accross organization or division. For instance, a company conversationnal agent could have access to some HR documents, each user's special information, his mailbox… Thus tooling requires desiloing data.

This forces us to take a step back and reflect on the **architecture of AI services**.

- At the most basic level, an AI agent is simply a **computer program** that runs somewhere on a machine. For reliability and security, it is often launched inside a virtual environment or a Docker container to ensure proper isolation.
- The agent will frequently need to call a **Large Language Model**, which can either be a proprietary model accessed through an external API, or a locally hosted model—sometimes itself running inside a separate container.
- **Tools** occupy a special status within this architecture. Although they are executed during the agent's runtime, tools can often be the *submerged part of the iceberg*: what the agent directly calls is just a lightweight wrapper, while the actual computation may be delegated to much heavier machinery (for example, a retrieval pipeline, a database query engine, or a GPU-intensive ML model).

14 https://www.anthropic.com/engineering/building-effective-agents?utm_source=chatgpt.com

III) Exposing Tools to the World : Towards new standards

The architectural issues we have highlighted show that, in the long run, it is not feasible for all tools to be written, stored, and hosted within the same application that supports the agent. While it makes sense for some tools to remain tightly coupled to their local environment, this approach quickly runs into several limitations:

- **Scalability –** It is simply impossible to concentrate every possible functionality in a single codebase.
- **Redundancy –** Many tools end up being rewritten multiple times by different teams, even when they could have been implemented once and reused by all.
- **Security and governance –** Certain information or operations should only be exposed and stored in a single secure location, not duplicated across environments.

In response, we are witnessing the emergence of a **global tool ecosystem**. On one end of the spectrum, we see open-source packages providing LLMs with common capabilities (such as web browsing, database access, or YouTube retrieval). On the other, more ambitious initiatives are appearing to define **complex communication standards** that allow tools to be shared, orchestrated, and even billed across multiple institutions.

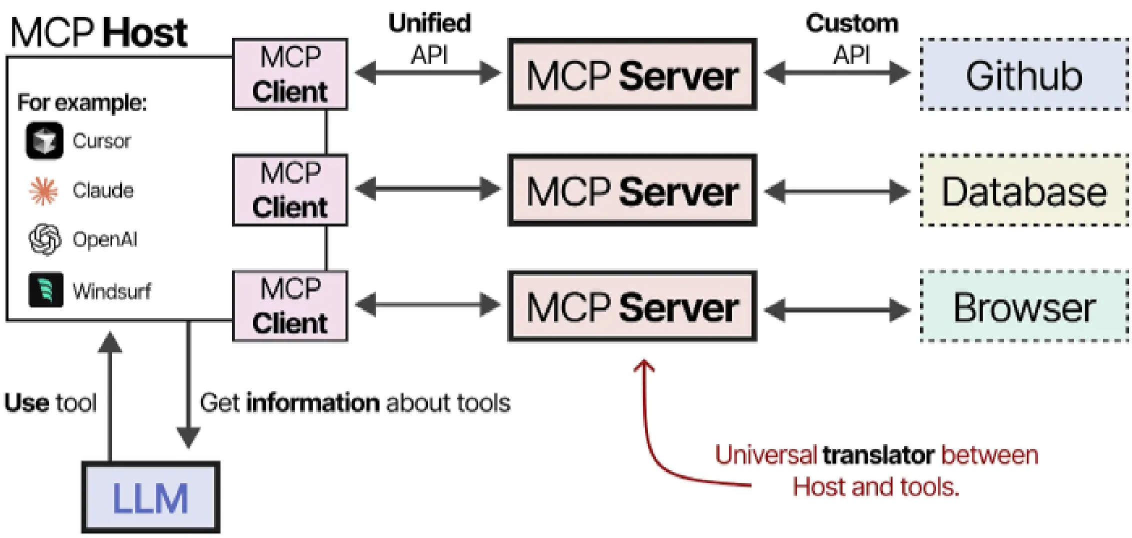
a) API and the microservice architecture :

One way to overcome the limitations of embedding all tools directly into the agent's codebase is to treat **APIs as tools**. An API (Application Programming Interface) can be defined as a standardized contract that allows one piece of software to request a service or data from another without needing access to its internal implementation. Rather than duplicating code or embedding fragile connectors inside every agent, the functionality is externalized and exposed through a stable, well-documented interface. This is precisely the logic behind the **REST standard—**resources are identified by URLs, manipulated through universal methods such as GET or POST, and returned in structured formats like JSON. From the perspective of an LLM, an API call thus becomes analogous to using a tool: the agent is told what the tool does, what inputs it requires, and what form its output will take. Crucially, APIs solve several of the bottlenecks of internal tool design highlighted earlier. They prevent redundancy by allowing a capability to be written once and reused across many agents. They mitigate security risks by ensuring sensitive functionality remains in a single controlled service, rather than scattered across instances. They align with the **microservice architecture**, where each service encapsulates one piece of functionality that can be independently deployed, scaled, and maintained. Yet APIs also bring their own constraints: tool descriptions must translate the often technical API documentation into a form that an LLM can reliably interpret, error codes must be made explicit, and authentication must be handled with care. In this sense, exposing an API to an agent is not just connecting software to software—it is an act of **tool design**, where the developer transforms an abstract service into something the LLM can meaningfully wield. Thinking of APIs as tools thus keeps the notion of tooling at the heart of the architecture: they are modular, shareable, and secure instruments that extend the agent's capabilities beyond its immediate environment.

b) MCP, the new standard ?

APIs are a very practical way to expose and share functionalities across applications, but their direct integration with LLMs is not as straightforward as one might expect. Even though there are best practices for documenting APIs—such as the OpenAPI specification—there is still no single, universal standard. Many APIs are already available in the wild, yet, as we have seen in section II.b, APIs written for developers are not always designed in a way that makes them usable as tools for language models. Their documentation may be too vague, their parameters too implicit, or their outputs too unstructured. In addition, connecting to most APIs in practice requires **non-trivial authentication steps** (API keys, OAuth tokens, access scopes), which are cumbersome for agents to manage and often outside the context the LLM itself can handle.

It is in this landscape that new initiatives such as the **Model Context Protocol (MCP)** have appeared. Model Context Protocol[15] started as an internal Anthropic side project when their engineers were facing the same tooling difficulties as everyone else. Inside of the company, several teams were developping the exact same tools for web access or File System exploration. They therefore decided to create a unified framework not only to expose capabilities, as API do, but to also unify the way LLMs.



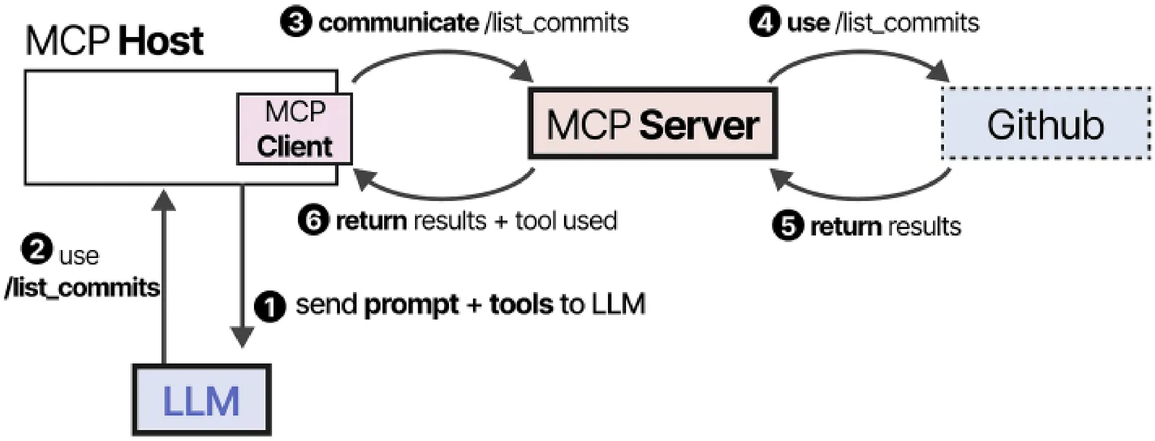
MCP proposes a standard way to expose tools, making them discoverable, documented, and callable by agents in a consistent manner. Unlike a raw API, an MCP-compliant service is described in a format designed to be read and used by LLMs. It defines not only the endpoints but also the semantics of tool use, authentication, and error handling. In other words, MCP seeks to do for agentic tools what REST once did for distributed services: provide a shared convention that makes them interoperable across systems. MCP forces people to better document the tools they expose (description and typed inputs at least, outputs is, unfortunately, optionnal and not enforced…).

* talk about the client server architecture

* // with http and C, history

15 https://www.anthropic.com/news/model-context-protocol

* the online running



BUT an unstandardized standard,

Activity :

What are the pros and cons of MCP ? Look on the web / linkedIn for information about how people percieve MCP.

not certain in its core (see, streamable http more recent but not fully supported, safety, authentication...

While still young, MCP signals a possible shift from the fragmented world of ad-hoc APIs towards a **tooling ecosystem** built on common ground. The open question, of course, is whether it will succeed in becoming *the* standard, or whether the industry will continue to see a proliferation of incompatible approaches.

c) Tools from an Economic Perspective

Seen from an economic perspective, the emergence of standards such as MCP points toward the rise of a new market dynamic. Just as access to APIs has long been monetized—whether through subscription tiers or per-request billing—access to MCP-exposed tools is likely to follow the same path. We are entering an ecosystem in which different actors occupy complementary positions: some provide the **models**, others develop and sell **tools**, while still others focus on the **orchestration layers** and agentic capabilities that connect everything together. Yet this neat division of labor is already threatened by competition, since the companies that control the most powerful models are also seeking to dominate the downstream market for tools and orchestration.

This creates a landscape reminiscent of existing commercial APIs such as SerpAPI or data brokers, where customers are charged per request for access to curated information or functionality. In the same way, we should expect the development of **"Tool-as-a-Service"** offerings, where access to specialized tools—retrieval engines, analytic modules, financial APIs, or proprietary databases—is packaged, priced, and sold as an independent service. Whether this leads to a flourishing marketplace of interoperable tools, or to a more monopolistic capture of the agentic ecosystem by a few large model providers, remains one of the key open questions shaping the future of AI agents.