Introduction to Agents

🤔 What are agents?

Any efficient system using AI will need to provide LLMs some kind of access to the real world: for instance the possibility to call a search tool to get external information, or to act on certain programs in order to solve a task. In other words, LLMs should have ***agency***. Agentic programs are the gateway to the outside world for LLMs.

> AI Agents are **programs where LLM outputs control the workflow**.

Any system leveraging LLMs will integrate the LLM outputs into code. The influence of the LLM's input on the code workflow is the level of agency of LLMs in the system.

Note that with this definition, "agent" is not a discrete, 0 or 1 definition: instead, "agency" evolves on a continuous spectrum, as you give more or less power to the LLM on your workflow.

See in the table below how agency can vary across systems:

Agency Level	Description	Short name	Example Code
☆☆☆	LLM output has no impact on program flow	Simple processor	`process_llm_output(llm_response)`
★☆☆	LLM output controls an if/else switch	Router	`if llm_decision(): path_a() else: path_b()`
★★☆	LLM output controls function execution	Tool call	`run_function(llm_chosen_tool, llm_chosen_args)`

Agency Level	Description	Short name	Example Code
★★☆	LLM output controls iteration and program continuation	Multi-step Agent	`while llm_should_continue(): execute_next_step()`
★★★	One agentic workflow can start another agentic workflow	Multi-Agent	`if llm_trigger(): execute_agent()`
★★★	LLM acts in code, can define its own tools / start other agents	Code Agents	`def custom_tool(args): ...`

The multi-step agent has this code structure:

```
memory = [user_defined_task]
while llm_should_continue(memory): # this loop is the multi-step part
    action = llm_get_next_action(memory) # this is the tool-calling part
    observations = execute_action(action)
    memory += [action, observations]
```

This agentic system runs in a loop, executing a new action at each step (the action can involve calling some pre-determined *tools* that are just functions), until its observations make it apparent that a satisfactory state has been reached to solve the given task. Here's an example of how a multi-step agent can solve a simple math question:

✅ When to use agents / ⛔ when to avoid them

Agents are useful when you need an LLM to determine the workflow of an app. But they're often overkill. The question is: do I really need flexibility in the workflow to efficiently solve the task at hand? If the pre-determined workflow falls short too often, that means you need more flexibility. Let's take an example: say you're making an app that handles customer requests on a surfing trip website.

You could know in advance that the requests will belong to either of 2 buckets (based on user choice), and you have a predefined workflow for each of these 2 cases.

1. Want some knowledge on the trips? ⇒ give them access to a search bar to search your knowledge base
2. Wants to talk to sales? ⇒ let them type in a contact form.

If that deterministic workflow fits all queries, by all means just code everything! This will give you a 100% reliable system with no risk of error introduced by letting unpredictable LLMs meddle in your workflow. For the sake of simplicity and robustness, it's advised to regularize towards not using any agentic behaviour.

But what if the workflow can't be determined that well in advance?

For instance, a user wants to ask: `"I can come on Monday, but I forgot my passport so risk being delayed to Wednesday, is it possible to take me and my stuff to surf on Tuesday morning, with a cancellation insurance?"` This question hinges on many factors, and probably none of the predetermined criteria above will suffice for this request.

If the pre-determined workflow falls short too often, that means you need more flexibility.

That is where an agentic setup helps.

In the above example, you could just make a multi-step agent that has access to a weather API for weather forecasts, Google Maps API to compute travel distance, an employee availability dashboard and a RAG system on your knowledge base.

Until recently, computer programs were restricted to pre-determined workflows, trying to handle complexity by piling up if/else switches. They focused on extremely narrow tasks, like “compute the sum of these numbers” or “find the shortest path in this graph”. But actually, most real-life tasks, like our trip example above, do not fit in pre-determined workflows. Agentic systems open up the vast world of real-world tasks to programs!

Why smolagents ?

For some low-level agentic use cases, like chains or routers, you can write all the code yourself. You’ll be much better that way, since it will let you control and understand your system better.

But once you start going for more complicated behaviours like letting an LLM call a function (that’s “tool calling”) or letting an LLM run a while loop (“multi-step agent”), some abstractions become necessary:

- For tool calling, you need to parse the agent’s output, so this output needs a predefined format like “Thought: I should call tool ‘get_weather’. Action: get_weather(Paris).”, that you parse with a predefined function, and system prompt given to the LLM should notify it about this format.
- For a multi-step agent where the LLM output determines the loop, you need to give a different prompt to the LLM based on what happened in the last loop iteration: so you need some kind of memory.

See? With these two examples, we already found the need for a few items to help us:

- Of course, an LLM that acts as the engine powering the system
- A list of tools that the agent can access
- A parser that extracts tool calls from the LLM output
- A system prompt synced with the parser
- A memory

But wait, since we give room to LLMs in decisions, surely they will make mistakes: so we need error logging and retry mechanisms.

All these elements need tight coupling to make a well-functioning system. That's why we decided we needed to make basic building blocks to make all this stuff work together.

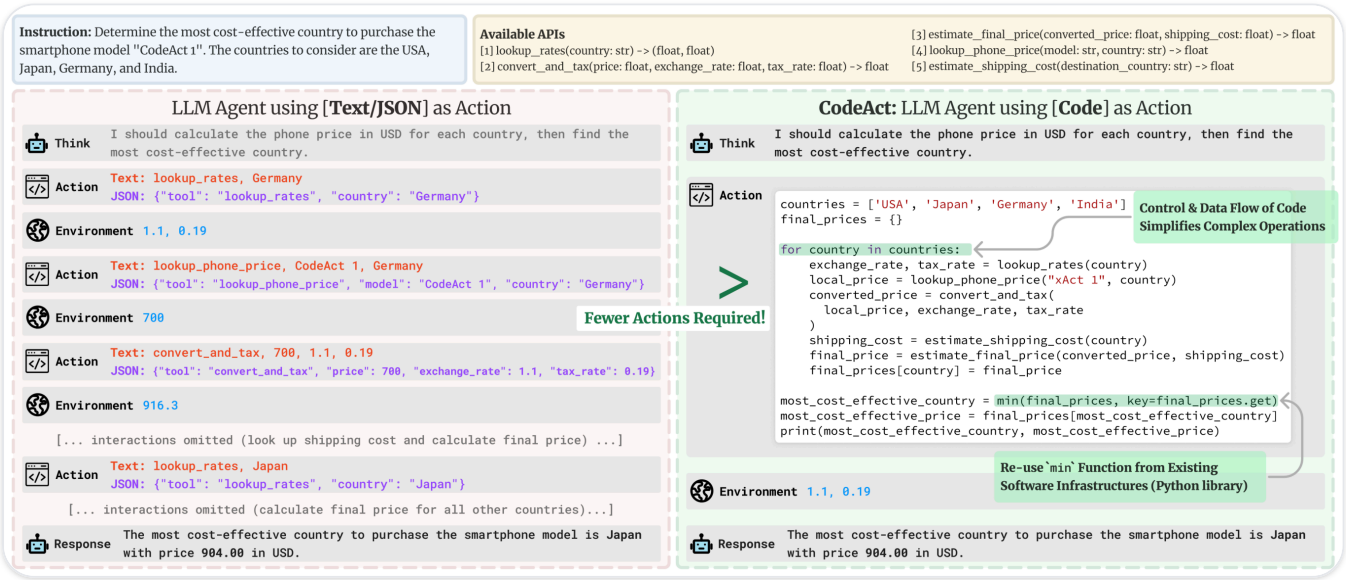
Code agents

In a multi-step agent, at each step, the LLM can write an action, in the form of some calls to external tools. A common format (used by Anthropic, OpenAI, and many others) for writing these actions is generally different shades of "writing actions as a JSON of tools names and arguments to use, which you then parse to know which tool to execute and with which arguments".

Multiple research papers have shown that having the tool calling LLMs in code is much better.

The reason for this simply that *we crafted our code languages specifically to be the best possible way to express actions performed by a computer*. If JSON snippets were a better expression, JSON would be the top programming language and programming would be hell on earth.

The figure below, taken from Executable Code Actions Elicit Better LLM Agents, illustrates some advantages of writing actions in code:



Writing actions in code rather than JSON-like snippets provides better:

- **Composability:** could you nest JSON actions within each other, or define a set of JSON actions to re-use later, the same way you could just define a python function?
- **Object management:** how do you store the output of an action like `generate_image` in JSON?
- **Generality:** code is built to express simply anything you can have a computer do.
- **Representation in LLM training data:** plenty of quality code actions are already included in LLMs' training data which means they're already trained for this!

<> Update on GitHub

← 📚 Manage your agent's memory

🤔 How do Multi-step agents work? →