How do multi-step agents work?

The ReAct framework (Yao et al., 2022) is currently the main approach to building agents.

The name is based on the concatenation of two words, “Reason” and “Act.” Indeed, agents following this architecture will solve their task in as many steps as needed, each step consisting of a Reasoning step, then an Action step where it formulates tool calls that will bring it closer to solving the task at hand.

All agents in `smolagents` are based on singular `MultiStepAgent` class, which is an abstraction of ReAct framework.

On a basic level, this class performs actions on a cycle of following steps, where existing variables and knowledge is incorporated into the agent logs like below:

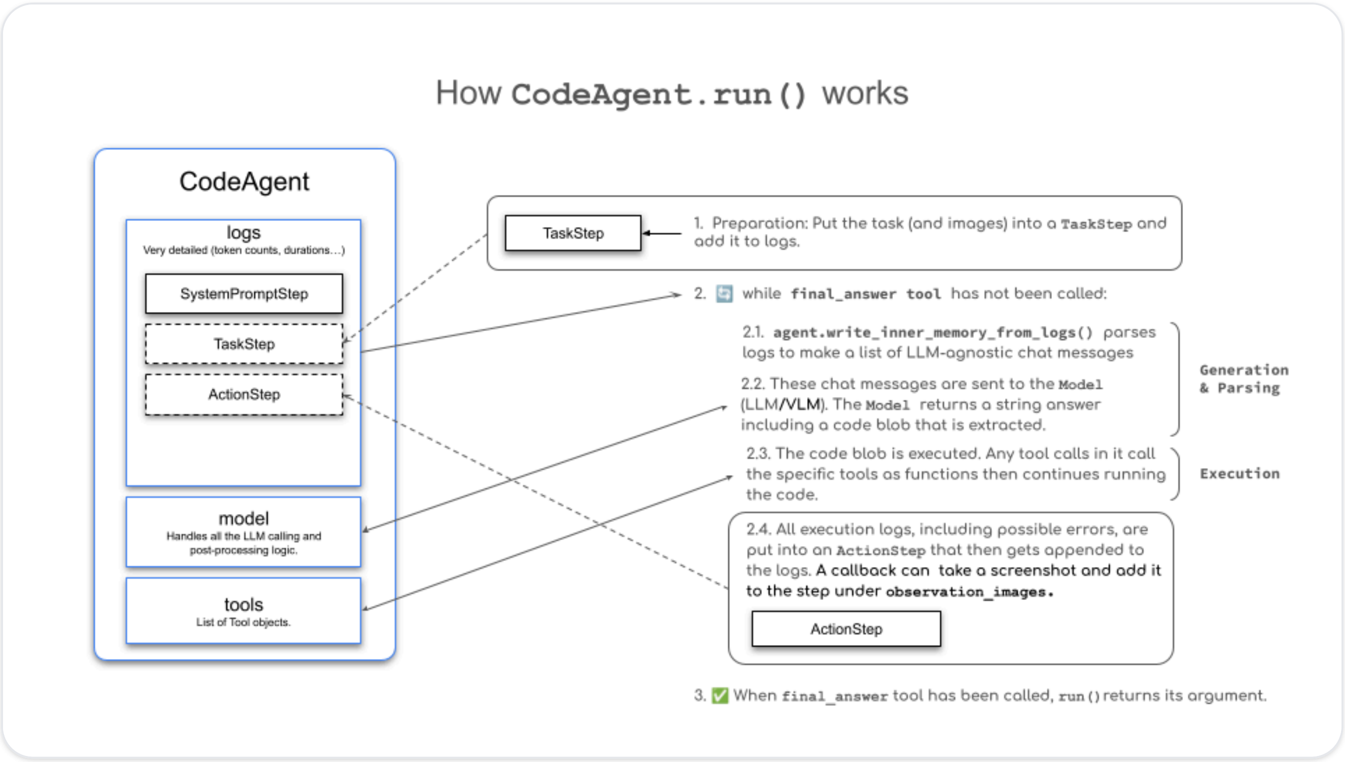
Initialization: the system prompt is stored in a `SystemPromptStep`, and the user query is logged into a `TaskStep` .

While loop (ReAct loop):

- Use `agent.write_memory_to_messages()` to write the agent logs into a list of LLM-readable chat messages.
- Send these messages to a `Model` object to get its completion. Parse the completion to get the action (a JSON blob for `ToolCallingAgent`, a code snippet for `CodeAgent`).
- Execute the action and logs result into memory (an `ActionStep`).
- At the end of each step, we run all callback functions defined in `agent.step_callbacks` .

Optionally, when planning is activated, a plan can be periodically revised and stored in a `PlanningStep` . This includes feeding facts about the task at hand to the memory.

For a `CodeAgent`, it looks like the figure below.



1. Preparation: Put the task (and images) into a `TaskStep` and add it to logs.
2. 🔄 while `final_answer tool` has not been called:
   - 2.1. `agent.write_inner_memory_from_logs()` parses logs to make a list of LLM-agnostic chat messages
   - 2.2. These chat messages are sent to the `Model` (LLM/VLM). The `Model` returns a string answer including a code blob that is extracted.
   - 2.3. The code blob is executed. Any tool calls in it call the specific tools as functions then continues running the code.
   - 2.4. All execution logs, including possible errors, are put into an `ActionStep` that then gets appended to the logs. A callback can take a screenshot and add it to the step under `observation_images`.
3. ✅ When `final_answer` tool has been called, `run()` returns its argument.

Here is a video overview of how that works:

We implement two versions of agents:

- CodeAgent is the preferred type of agent: it generates its tool calls as blobs of code.
- ToolCallingAgent generates tool calls as a JSON in its output, as is commonly done in agentic frameworks. We incorporate this option because it can be useful in some narrow cases where you can do fine with only one tool call per step: for instance, for web browsing, you need to wait after each action on the page to monitor how the page changes.

> Read Open-source LLMs as LangChain Agents blog post to learn more about multi-step agents.

<> Update on GitHub

← 🤖 An introduction to agentic systems Self-correcting Text-to-SQL →