Apr 13, 2025

Implementing ReAct Agentic Pattern From Scratch

AI Agents Crash Course—Part 10 (with implementation).

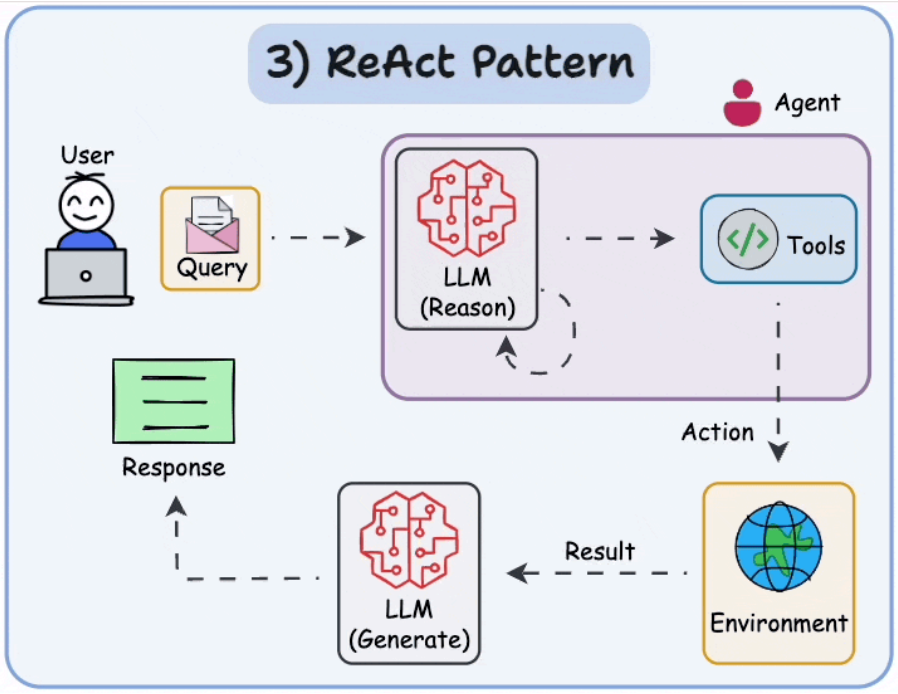
Avi Chawla, Akshay Pachaar

Introduction

Building advanced AI agents has been greatly simplified by high-level frameworks like LangChain and CrewAI.

However, these frameworks often hide the underlying logic that makes an AI agent tick.

More specifically, under the hood, many such frameworks use the ReAct (Reasoning and Acting) pattern to let large language models (LLMs) think through problems and use tools to act on the world.



For example, an agent in CrewAI typically alternates between reasoning about a task and acting (using a tool) to gather information or execute steps, following the ReAct paradigm.

This approach enhances an LLM agent's ability to handle complex tasks and decisions by combining chain-of-thought reasoning with external tool use.

While frameworks like CrewAI implement this logic for you and you should also keep using them in production systems (also because they are built by experts), often times, relying on pre-packaged libraries without understanding their inner workings can lead to confusion.

You might use an agent framework and wonder how it decides to, say, perform a web search or a calculation as part of answering a question.

An example is shown below where the Agent is going through a series of thought activities before producing a response.

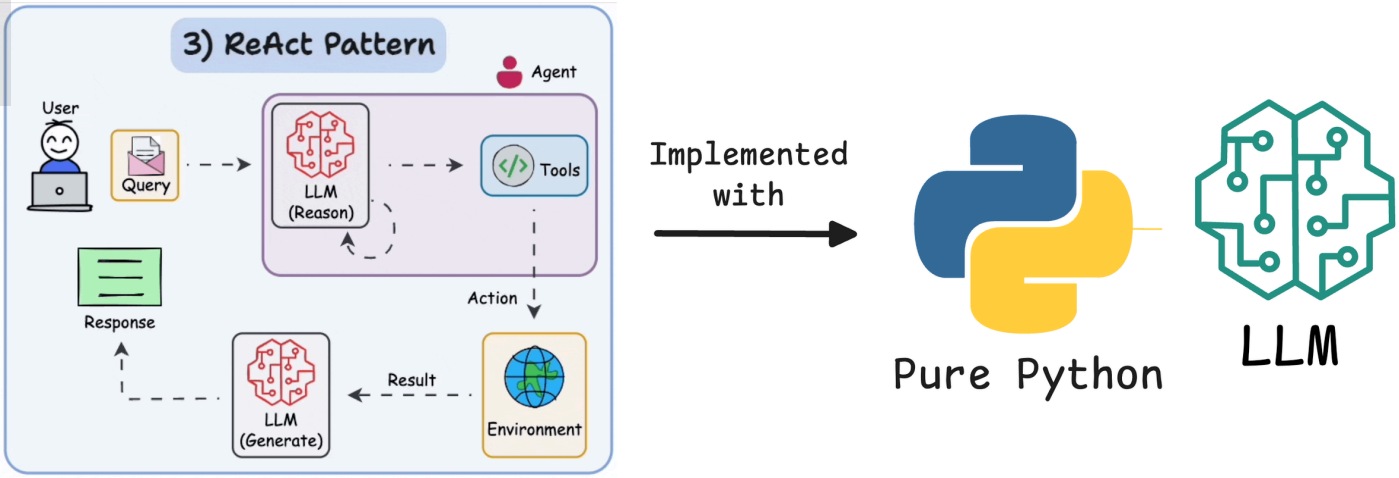
```
# Agent: News Collector
## Task: Search for the latest news on Agent2Agent Protocol


# Agent: News Collector
## Final Answer:
1. **"Agent2Agent Protocol Expands Blockchain Communication" - November 10, 2023**
   The Agent2Agent Protocol has announced its latest advancements in facilitating seamless communication between decentralized agents on various bl

2. **"Agent2Agent Protocol Announces Strategic Partnership" - November 7, 2023**
   A significant partnership has been formed between Agent2Agent Protocol and a leading technology firm to broaden their capabilities in artificial
```

```
# Agent: News Reporter
## Task: Create a news headline on Agent2Agent Protocol from the News Collector. Also validate the news is relevant to Agent2Agent Protocol using t

Thought

# Agent: News Reporter
## Thought: I need to validate the relevance of the news related to the Agent2Agent Protocol by searching the internet for recent developments.
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol news November 2023\"}"
## Tool Output:
```

```
Thought

# Agent: News Reporter
## Thought: Thought: The search results indicate that the Agent2Agent Protocol is currently being discussed in various contexts, especially in rela
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol latest news\"}"
## Tool Output:

Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
Snippet: Today, we're launching a new, open protocol called Agent2Agent (A2A), with support and contributions from more than 50 technology partners
---
```

```
Thought

# Agent: News Reporter
## Thought: The Agent2Agent Protocol has made significant strides recently with new announcements and partnerships involving AI and blockc
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol news updates\"}"
## Tool Output:

Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
Snippet: Today, we're launching a new, open protocol called Agent2Agent (A2A), with support and contributions from more than 50 technology partners
---
```

In this article, we'll demystify that process by building a ReAct agent from scratch using only Python and an LLM.



By doing so, we gain full control over the agent's behavior, making it easier to optimize and troubleshoot.

We'll use OpenAI, but if you prefer to do it with Ollama locally, an open-source tool for running LLMs locally, with a model like Llama3 to power the agent, you can do that as well.

Along the way, we'll explain the ReAct pattern, design an agent loop that interleaves reasoning and tool usage, and implement multiple tools that the agent can call.

The goal is to help you understand both the theory and implementation of ReAct agents.

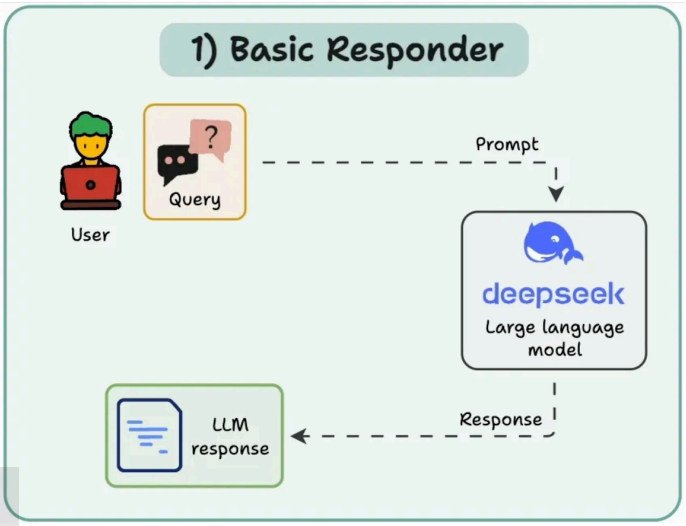
By the end, you'll have a working agent and a clear picture of how frameworks like CrewAI leverage ReAct internally.

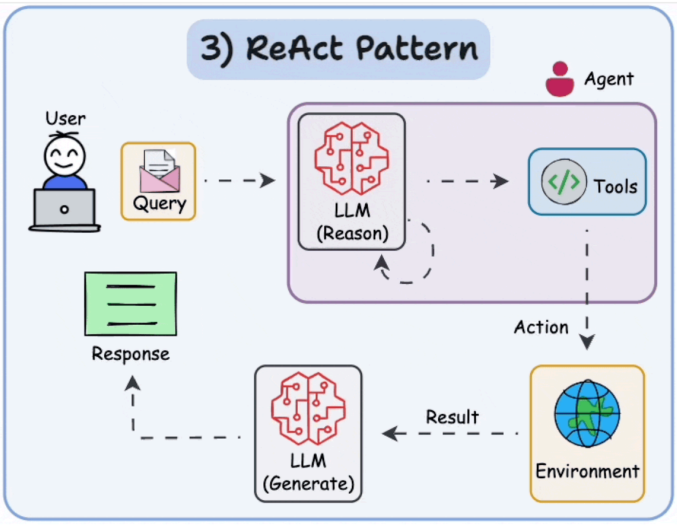
Let's begin!

What is ReAct pattern?

ReAct (short for Reasoning and Acting) is a paradigm for AI agent design where an agent uses chain-of-thought reasoning and tool-using actions in aggregation.

Instead of generating a direct answer in one step, a ReAct agent thinks step-by-step and can perform intermediate actions (like looking something up or calculating a value) before finalizing its answer.





To get a clear perspective on this, Like think about how ReAct pattern works.

Sample 1

Formally, an LLM following ReAct generates reasoning traces (the “Thoughts”) and task-specific actions (calls to tools) in an interleaved manner. This means the model’s output might look something like:

- Thought: I should calculate the total.
- Action: `Calculator("123 + 456")`
- Observation: `579`
- Thought: Now I have the sum; next, I need to multiply it.
- Action: `Calculator("579 * 789")`
- Observation: `456,831`.
- Thought: I have the final result.
- Final Answer: `456,831`.

The reasoning traces (the chain of thoughts) help the model plan and keep track of what to do next, while the actions let it consult external sources or perform calculations to gather information it otherwise wouldn’t have inherent access to.

In effect, the model isn't limited to its internal knowledge; it can reach out to tools, databases, or the internet as needed and then reason about the results.

This significantly enhances what the agent can do.

IBM describes ReAct agents as ones that use an LLM "brain" to coordinate reasoning and action, enabling interactions with the environment in a structured but adaptable way.

Unlike a basic chatbot that answers with whatever is in its static knowledge, a ReAct agent can think, search, calculate, and then combine the results into an answer.

Sample 2

Consider the output of an Agent I built in a multi-agent system (we'll get to the code shortly):

```
# Agent: News Reporter
## Task: Create a news headline on Agent2Agent Protocol from the News Collector. Also validate the news is relevant to Agent2Agent Protocol using t


# Agent: News Reporter
## Thought: I need to validate the relevance of the news related to the Agent2Agent Protocol by searching the internet for recent developments.
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol news November 2023\"}"
## Tool Output:
```

In the example shown above, we see a live trace of an AI News Reporter Agent executing a task using the ReAct paradigm. The agent has been asked to create a news headline related to the "Agent2Agent Protocol". However, rather than jumping to conclusions, it reasons step by step, as seen in its structured trace.

Let's break this down:

```
# Agent: News Reporter
## Task: Create a news headline on Agent2Agent Protocol from the News Collector. Also validate the news is relevant to Agent2Agent Protocol using t


# Agent: News Reporter
## Thought: I need to validate the relevance of the news related to the Agent2Agent Protocol by searching the internet for recent developments.
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol news November 2023\"}"
## Tool Output:
```

- Agent Role: `News Reporter` – This agent specializes in curating and validating news content.
- Task: The agent has been instructed to generate a news headline and ensure it's relevant to the Agent2Agent Protocol.
- Thought: The agent first reasons internally that it should validate the information by performing a quick search for any recent updates about the protocol. This is the reasoning part of the ReAct cycle.
- Action: It proceeds to use a tool labeled `Search the internet`, passing in a structured JSON input with the query: `"Agent2Agent Protocol news November 2023"`. This is the acting step where the agent leverages external tools to gather real-world data.

```
# Agent: News Reporter
## Thought: I need to validate the relevance of the news related to the Agent2Agent Protocol by searching the internet for recent developments.
## Using tool: Search the internet
## Tool Input:
"{\"search_query\": \"Agent2Agent Protocol news November 2023\"}"
## Tool Output:

Search results: Title: Announcing the Agent2Agent Protocol (A2A)
Link: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
Snippet: The A2A protocol will allow AI agents to communicate with each other, securely exchange information, and coordinate actions on top of variou
---
Title: #108 Google's Agent2Agent Protocol: The New Standard for AI ...
Link: https://sidsaladi.substack.com/p/108-googles-agent2agent-protocol
Snippet: Google's Agent2Agent (A2A) protocol is an open communication standard designed to enable independent AI agents to interoperate seamlessly.
---
Title: AI Agents and Automation: Multiagent Communication Protocols
Link: https://jingdongsun.medium.com/ai-agents-and-automation-multiagent-communication-protocols-940281ccc259
Snippet: Just a few days ago, another headline caught my attention — this time from Google — about the Agent2Agent Protocol (A2A). This news sparked
---
Title: Model Context Protocol + Agent2Agent | by Gregory Zem - Medium
Link: https://medium.com/@mne/model-context-protocol-agent2agent-ad25a1260b7a
Snippet: The good news in this whole story with A2A and MCP is the unification and standardization of interfaces. The bad news is that the hysteria
---
```

- Tool Output: It contains the results retrieved by the search tool—potentially news snippets, article summaries, or relevant URLs.

This illustrates the power of combining structured thought and external actions: rather than relying on the model's internal knowledge alone, the agent cross-checks

facts using tools.

It's an example of how the ReAct pattern encourages transparency, accuracy, and verifiability in agent behavior—an essential feature for any system tasked with real-world information synthesis.

You can imagine scaling this further with multi-agent setups: a News Collector gathers raw feeds, a Fact Verifier checks reliability, and this News Reporter constructs the headline—all coordinating asynchronously using reasoning and tool-based actions.

Here's the implementation if you want to replicate the same output above.

- Make sure you have created a `.env` file with the `OPENAI_API_KEY` specified in it. It will make things much easier and faster for you. Also, add these two lines of code to handle asynchronous operations within a Jupyter Notebook environment, which will allow us to make asynchronous calls smoothly to your Crew Agent.

- If you don't want to use OpenAI and stick to an open-source LLM, you can use Ollama as well.

  Follow these steps:
  - Download Ollama from https://ollama.com.
  - Run the Ollama app.
  - Pull an open-source LLM.

- Define the LLM as shown below.
- Pass the LLM as the `llm` parameter to the Agent.

Command line

```
ollama pull llama3.2:1b
```

```
from crewai import LLM

llm = LLM(
    model="ollama/llama3.2:1b",
    base_url="http://localhost:11434"
)
```

We begin by importing the essential classes from `crewai` and a useful tool: `SerperDevTool`. This tool wraps a real-time web search capability (via serper.dev) and allows our agents to fetch live information from the internet.

notebook.ipynb

```
from crewai import Agent, Task, Crew, Process
from crewai_tools import SerperDevTool
```

Also, get a Serper Dev API key from serper.dev and store it in the `.env` file created earlier:

.env

```
SERPER_API_KEY="..."
OPENAI_API_KEY="sk-...."
```

Next, initialize the Web Search Tool, which the Agents can invoke when they need web results:

notebook.ipynb

```
news_search_tool = SerperDevTool()
```

Moving on, define the First Agent—The News Collector:

notebook.ipynb

```
latest_news_agent = Agent(
    role="News Collector",
    goal="Get the latest news on a given topic and gather insights",
    backstory="""You are a reporter who is responsible for
                 getting the latest news on a given topic""",
    verbose=True,
)

news_search_task = Task(
    description="Search for the latest news on {topic}",
    expected_output="A list of the latest news on a given topic",
    tool=news_search_tool,
    agent=latest_news_agent
)
```

- This agent is designed to behave like a digital journalist. Its responsibility is to gather news stories related to a given topic using the Serper tool. The `verbose=True` flag ensures we get detailed logging—this is what creates the transparent ReAct-style trace showcased earlier.
- Also, the task instructs the `News Collector` to actively search for the latest information on the specified `{topic}`. The `tool` parameter links the task to `SerperDevTool`, enabling it to actually perform the search rather than hallucinating.

Next, define the Second Agent—The News Reporter:

notebook.ipynb

```
news_reporter_agent = Agent(
    role="News Reporter",
    goal="""Use the latest news from the News Collector
            and create a news headline.""",
    backstory="""You are a reporter who is uses the latest news on
                 a given topic and creating a headline.""",
    verbose=True,
)

news_headline_task = Task(
    description="""Create a news headline on {topic} from the News
                   Collector. Also validate the news is relevant to
                   {topic} using the web search tool.""",
    expected_output="A news headline on a given topic",
    tools=[news_search_tool],
    agent=news_reporter_agent
)
```

- This agent is the headline writer. It consumes the output from the previous task and crafts a concise headline. Like the collector, it's also verbose—meaning we'll see its reasoning steps, tool calls, and decisions in the logs.
- This Agent's task is particularly interesting because it challenges the reporter agent to do two things:
    - Use the prior output (collected news articles).

    - Perform its own validation using the search tool again—double-checking that the news is relevant.

- That's a ReAct pattern:
    - First, the agent reasons: "Do I have enough information? Is this valid?"
    - Then, it acts: makes a tool call to confirm.

Moving on, we connect our agents and tasks in a sequential crew pipeline:

notebook.ipynb

```
news_reporter_crew = Crew(
    agents=[latest_news_agent, news_reporter_agent],
    tasks=[news_search_task, news_headline_task],
    process=Process.sequential,
    verbose=True
)
```

- The Collector performs the initial search.
- The Reporter builds and validates the headline.

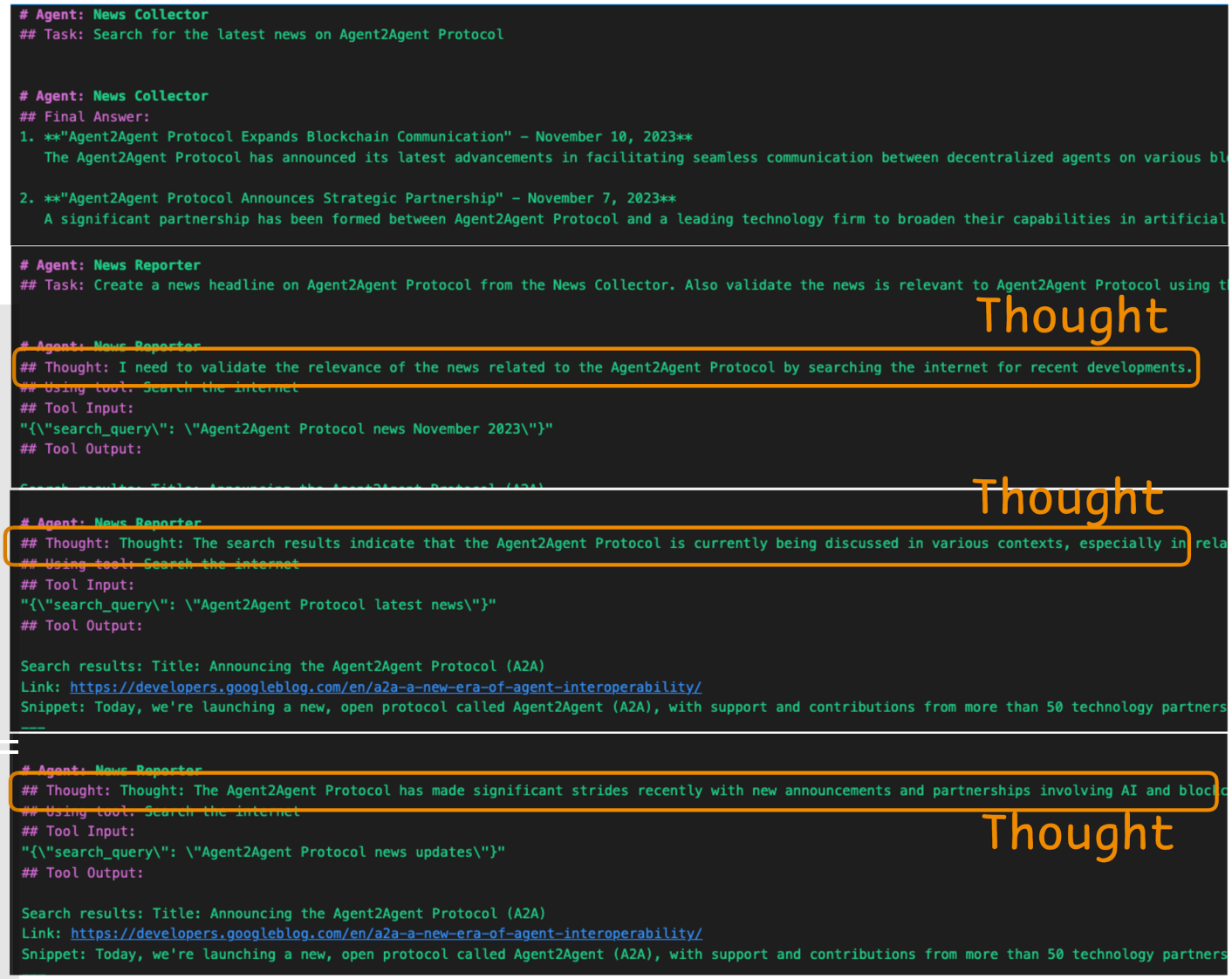
Finally, we start the workflow by passing a topic: Agent2Agent Protocol. The agents will dynamically process this input, use external tools, think through their next move, and generate a validated news headline.

notebook.ipynb

```
topic = "Agent2Agent Protocol"

response = news_reporter_crew.kickoff(inputs={"topic": topic})
```

This produces the following verbose output, which shows how your agent "thinks" in natural language, plans its next move, and uses external tools to carry out

actions.



But that's not all. Look at one more thing towards the end of this verbose output:

```
You ONLY have access to the following tools, and should NEVER make up tools that are not listed here:

Tool Name: Search the internet
Tool Arguments: {'search_query': {'description': 'Mandatory search query you want to use to search the internet', 'type': 'str'}}
Tool Description: A tool that can be used to search the internet with a search_query.

IMPORTANT: Use the following format in your response:

```
Thought: you should always think about what to do
Action: the action to take, only one name of [Search the internet], just the name, exactly as it's written.
Action Input: the input to the action, just a simple JSON object, enclosed in curly braces, using " to wrap keys and values.
Observation: the result of the action
```

Once all necessary information is gathered, return the following format:

```
Thought: I now know the final answer
Final Answer: the final answer to the original input question
```
```

Under the hood, this ReAct-style behavior is governed by a very specific format template—what we call the action protocol. When using tools like `SerperDevTool`, CrewAI instructs the LLM to follow a rigid response schema. This helps ensure agents interact with tools in a safe, deterministic, and interpretable manner.

First, we have the Agent tool prompt format:

Copy

```
"""
You ONLY have access to the following tools,
and should NEVER make up tools that are not listed here:

Tool Name: Search the internet
Tool Arguments: {'search_query': {'description':
                 'Mandatory search query you want to use
                 to search the internet', 'type': 'str'}}
Tool Description: A tool that can be used to
                  search the internet with a search_query.

"""
```

This is part of the tool context injected into the LLM prompt. It tells the agent:

- What tools are available.

- What arguments are required.
- That it must not invent tools or go off-protocol.

This creates strong constraints around agent behavior—which is important when you want to avoid hallucinations or misuse of capabilities.

The prompt also includes this critical instruction:

```
"""
IMPORTANT: Use the following format in your response:

```
Thought: you should always think about what to do
Action: the action to take, only one name of
[Search the internet], just the name, as it's written.

Action Input: the input to the action, just a
simple JSON object, enclosed in curly braces,
using " to wrap keys and values.

Observation: the result of the action
```
"""
```

This is the *reasoning* + *acting* loop spelled out in literal terms:

1. Thought: The agent expresses its internal reasoning.
2. Action: The agent picks the tool to use—verbatim.
3. Action Input: Arguments for the tool, formatted as strict JSON.
4. Observation: What the tool returned (i.e., the raw output).

This schema ensures that tools can be executed reliably, logged consistently, and traced clearly. And crucially—it's LLM-friendly. Everything is written in natural

language that the model understands and responds to.

Once all information is gathered, the agent is told to conclude like this:

```
"""
Once all necessary information is gathered,
return the following format:

```
Thought: I now know the final answer
Final Answer: the final answer to
the original input question
```
"""
```

This signals the end of the reasoning chain. At this point, the agent has completed its research and can produce a confident, top-level answer.

That's the essence of ReAct, which is implemented seamlessly using CrewAI.

But why does this matter?

ReAct was introduced as a major step in making LLM-based agents more reliable and powerful.

By having the model explain its thought process and check facts via tools, we reduce problems like hallucination and error propagation.

The original ReAct research by Yao et al. showed that this approach can overcome hallucination in question answering by letting the model retrieve real information (e.g. querying Wikipedia) to verify its facts.

Computer Science > Computation and Language

[Submitted on 6 Oct 2022 (v1), last revised 10 Mar 2023 (this version, v3)]

ReAct: Synergizing Reasoning and Acting in Language Models

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, Yuan Cao

While large language models (LLMs) have demonstrated impressive capabilities across tasks in language understanding and interactive decision making, their abilities for reasoning (e.g. chain-of-thought prompting) and acting (e.g. action plan generation) have primarily been studied as separate topics. In this paper, we explore the use of LLMs to generate both reasoning traces and task-specific actions in an interleaved manner, allowing for greater synergy between the two: reasoning traces help the model induce, track, and update action plans as well as handle exceptions, while actions allow it to interface with external sources, such as knowledge bases or environments, to gather additional information. We apply our approach, named ReAct, to a diverse set of language and decision making tasks and demonstrate its effectiveness over state-of-the-art baselines, as well as improved human interpretability and trustworthiness over methods without reasoning or acting components. Concretely, on question answering (HotpotQA) and fact verification (Fever), ReAct overcomes issues of hallucination and error propagation prevalent in chain-of-thought reasoning by interacting with a simple Wikipedia API, and generates human-like task-solving trajectories that are more interpretable than baselines without reasoning traces. On two interactive decision making benchmarks (ALFWorld and WebShop), ReAct outperforms imitation and reinforcement learning methods by an absolute success rate of 34% and 10% respectively, while being prompted with only one or two in-context examples. Project site with code: this https URL

It also improves the transparency of the agent's decision-making, as we can inspect the chain of thought for debugging or trustworthiness.

Overall, the ReAct pattern turns a passive LLM into an active problem solver that can break down complex tasks and interact with external data sources, much like an autonomous assistant.

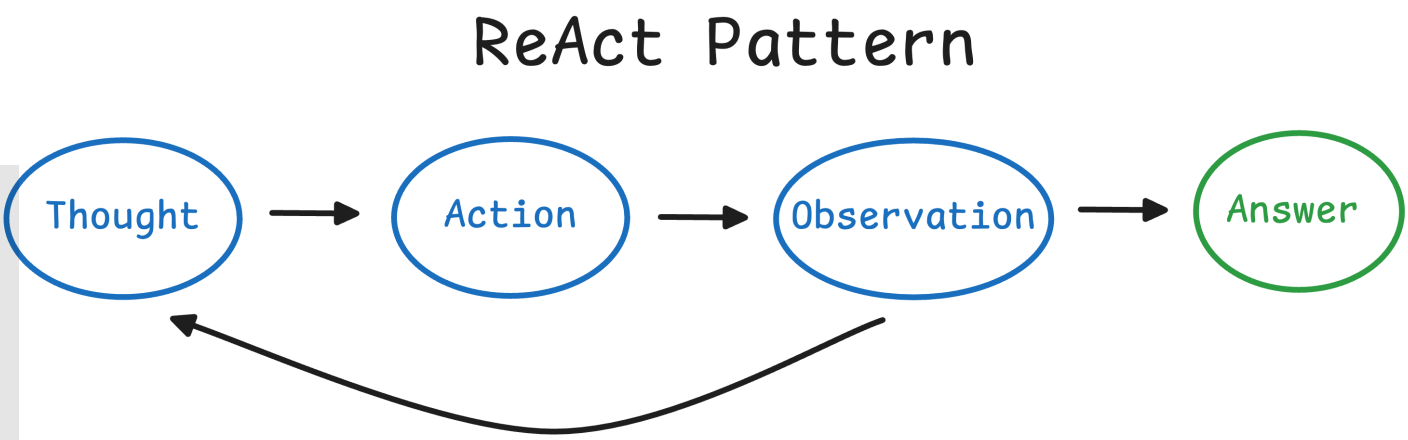
This also explains why it is widely used in almost Agentic frameworks. The actual implementation could vary but everything does connect back to something that's derived from a ReAct pattern.

To reiterate, this format:

- Forces your LLM to operate step by step,
- Clearly separates thinking from acting,
- Guarantees deterministic input-output behavior for tools,
- And produces traceable reasoning chains you can inspect or debug.

Reasoning + Acting: How ReAct Agents Work

A ReAct agent operates in a loop of Thought → Action → Observation, repeating until it reaches a solution or a final answer.



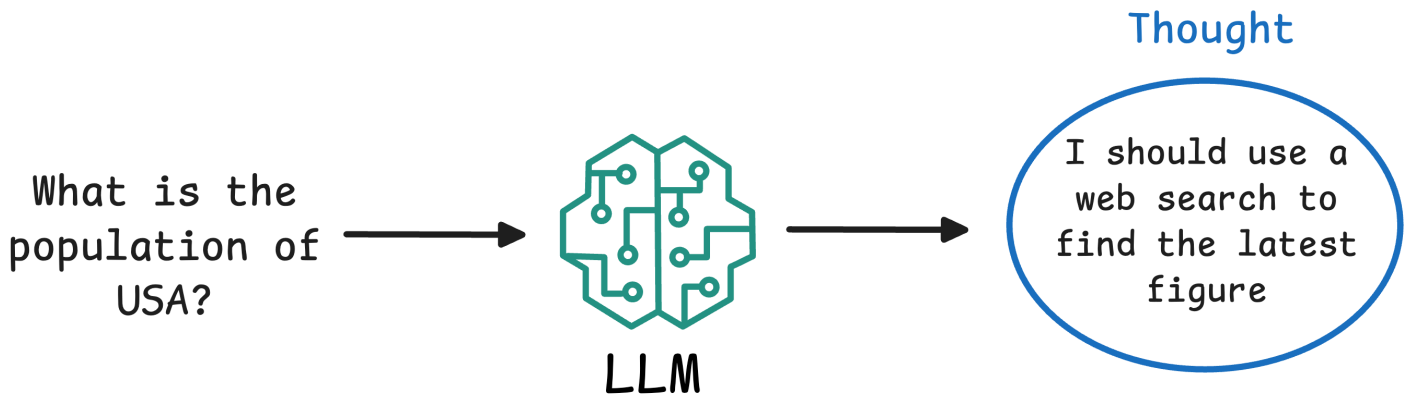
This is analogous to how humans solve problems:

- we think about what to do
- perform an action (like looking something up or doing a calculation),
- observe the result
- and then incorporate that into our next thought.

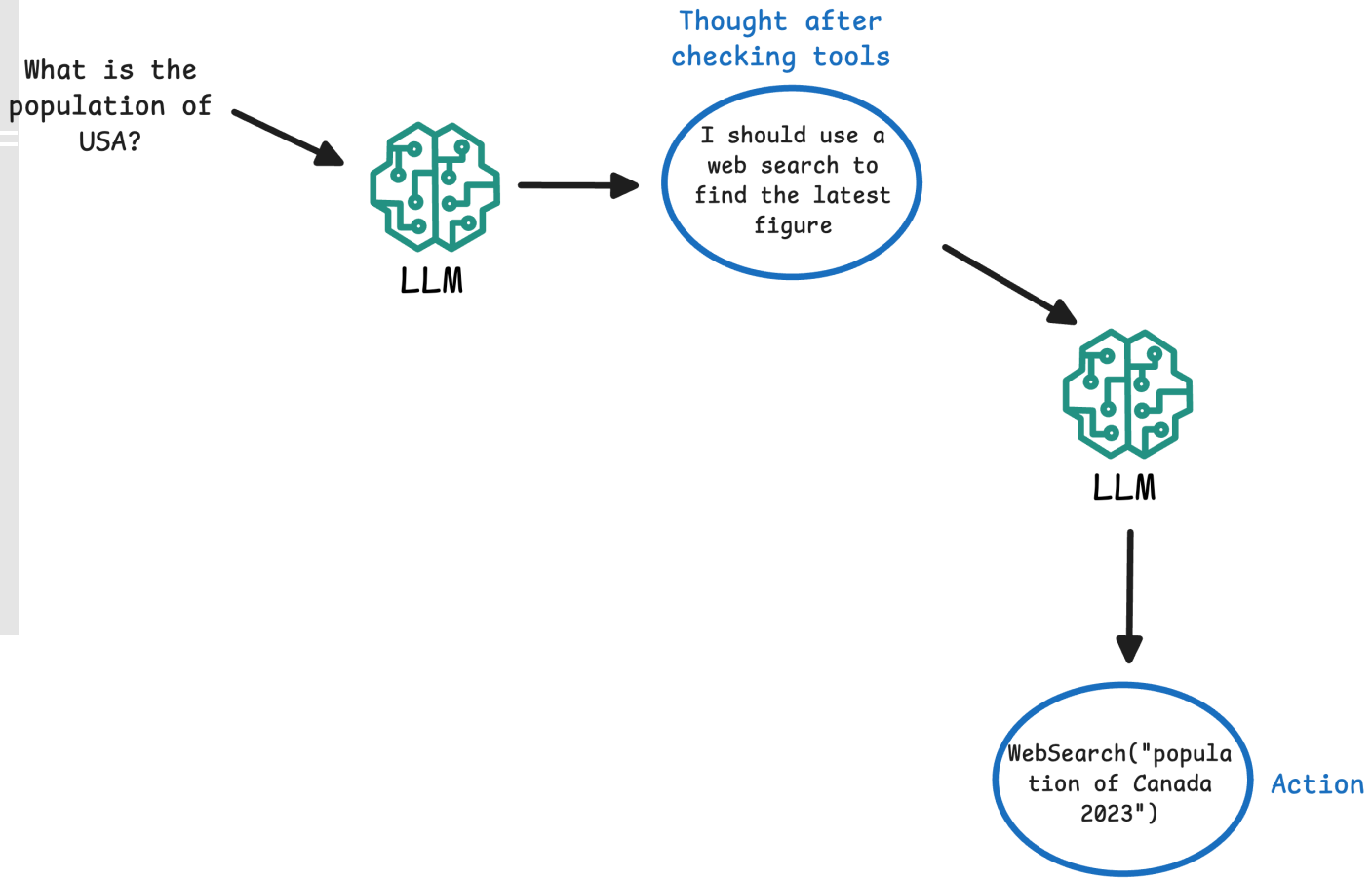
The ReAct framework uses prompt engineering to enforce this structured approach, alternating the model's thoughts and actions/observations.

Here's a step-by-step breakdown of the ReAct cycle in an AI agent:

- Thought: The Agent (powered by an LLM) analyzes the user's query and internal context, and produces a reasoning step in natural language. This is typically not shown to the end user but is part of the agent's self-talk. For example: "The question asks for the population of a country; I should use a web search to find the latest figure."

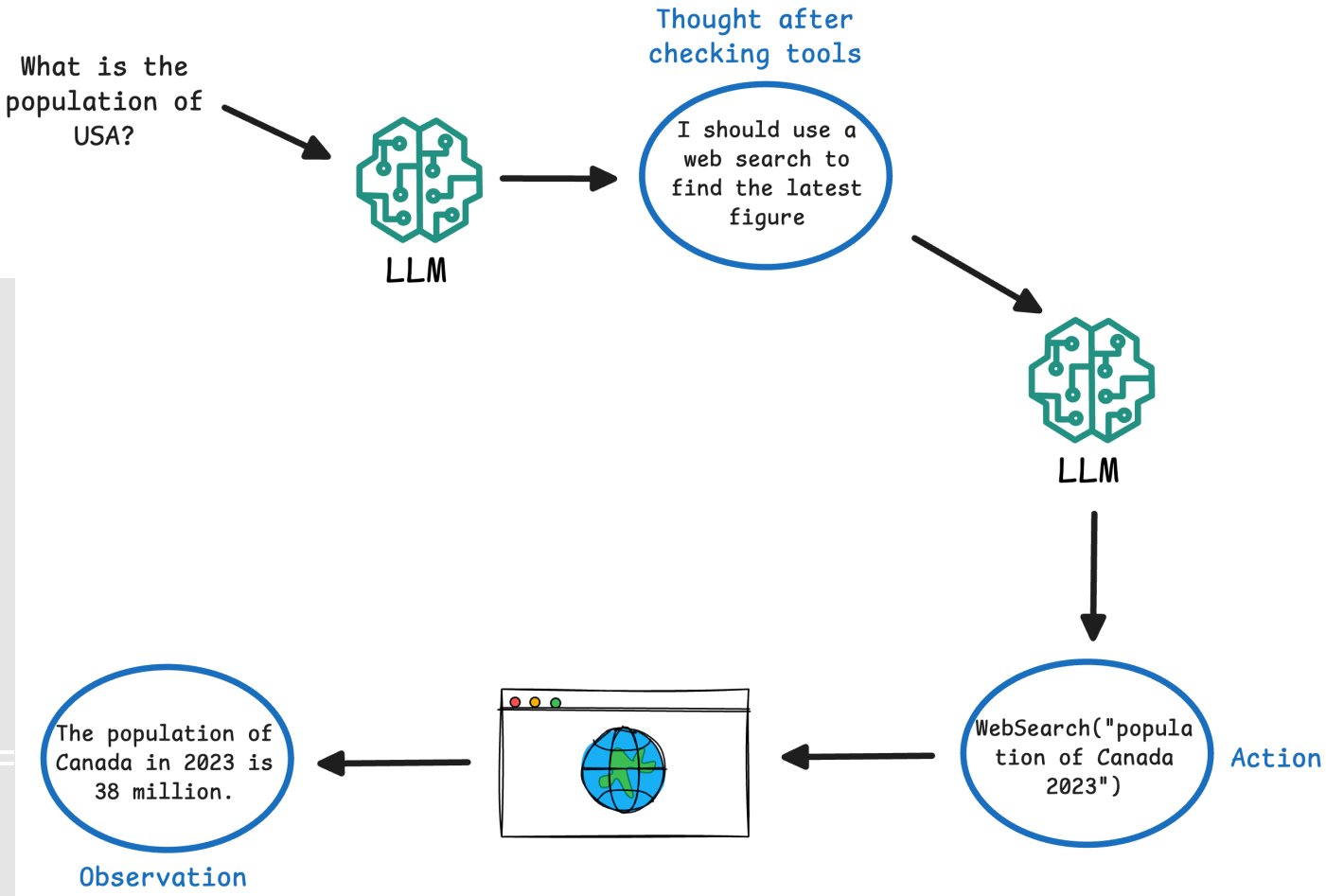


- Action: Based on the thought, the agent decides on an external tool or operation to perform. It outputs a prescribed format indicating the action. For instance: `Action: WebSearch("population of Canada 2023")`. The agent essentially "calls" a function (tool) by name, often with some input parameters.

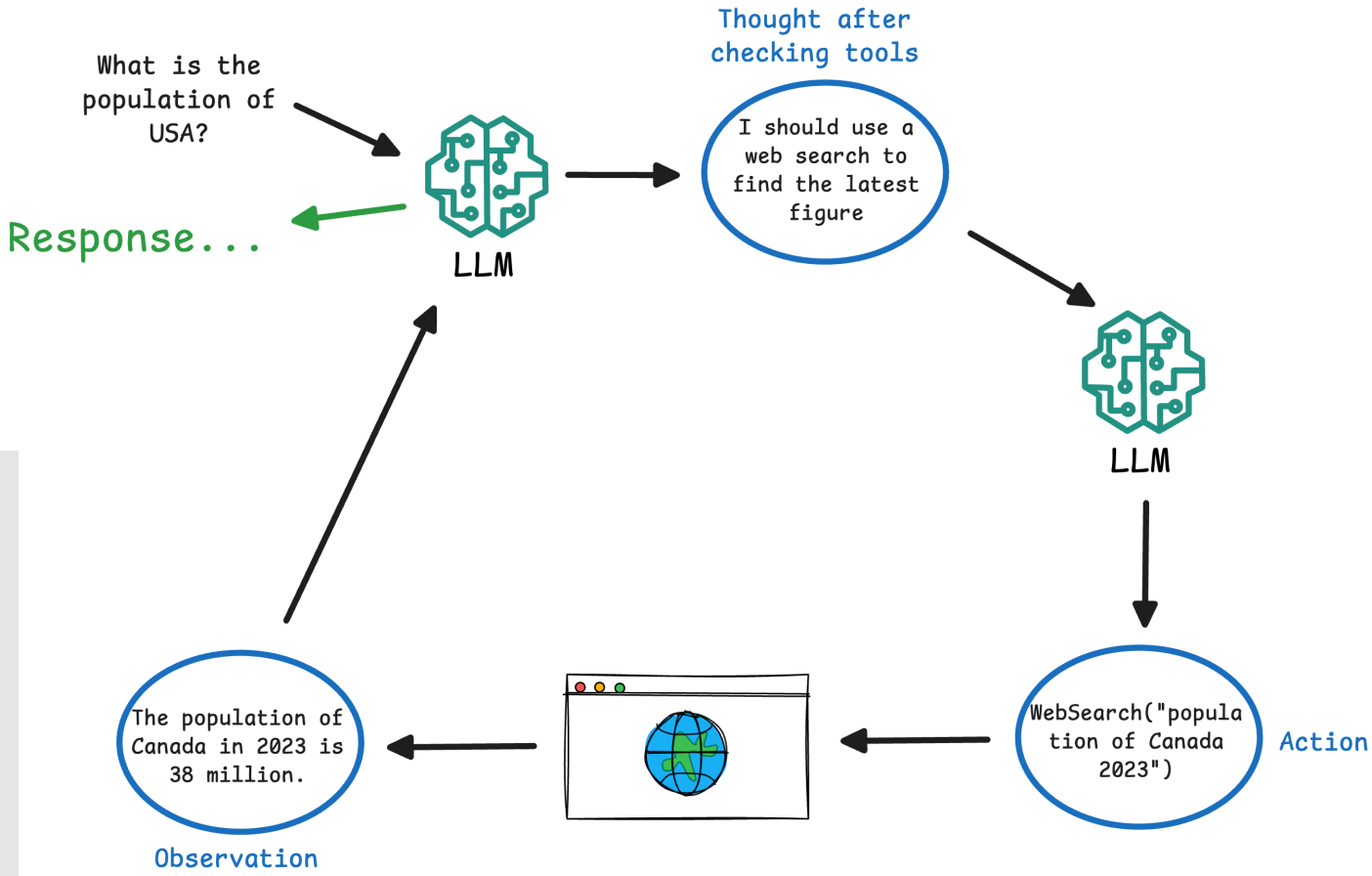


- Observation: The agent's environment (our code) executes the requested action and returns the result (observation) back to the agent. For example, the

web search tool might return: "`Observation: The population of Canada in 2023 is 38 million.`" This observation is fed into the agent's context.



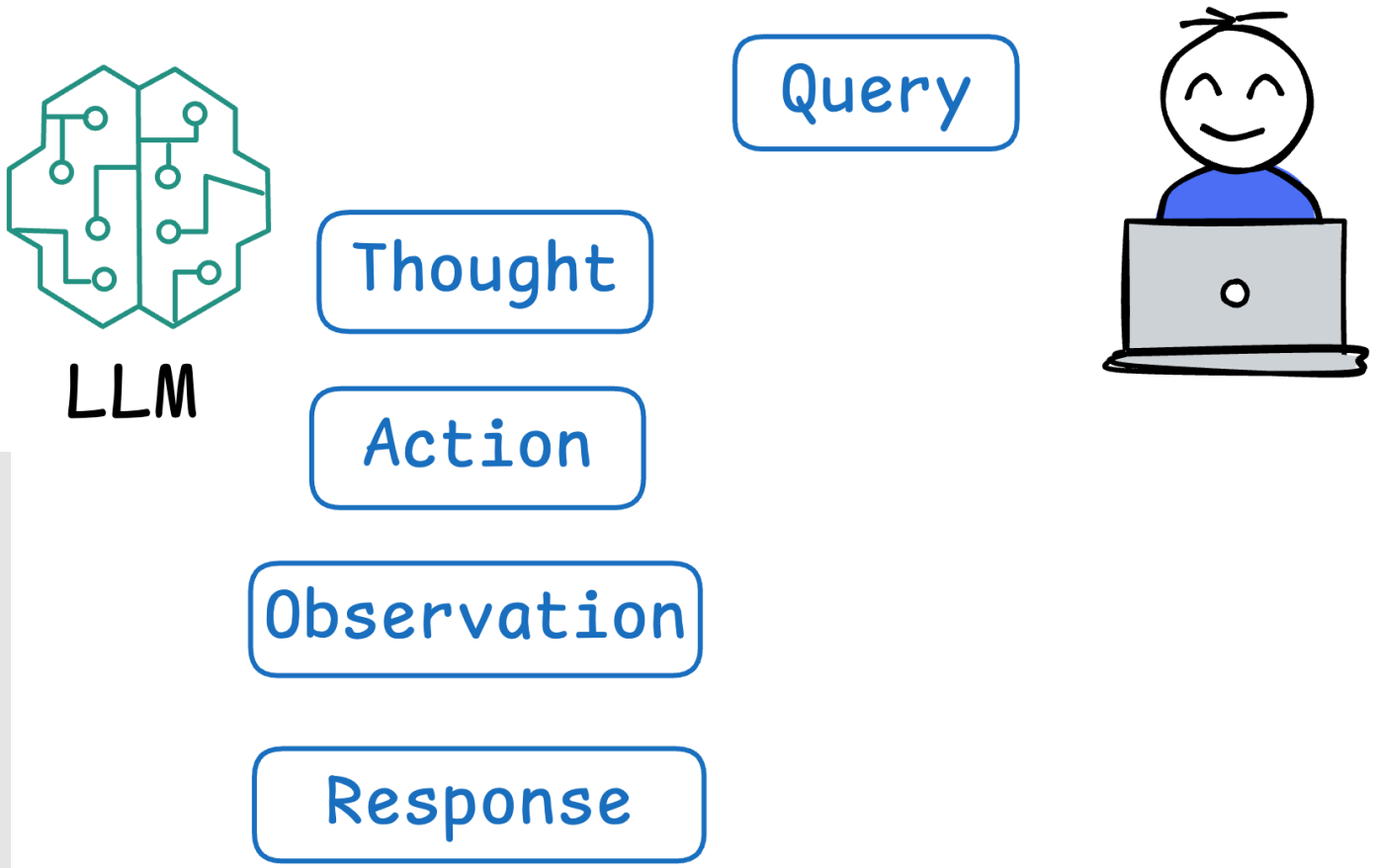
- The agent takes the new information and goes back to step 1 (another Thought). It will reason with the fresh data. In our example, it might think: "Now I have the population figure; I can answer the question."



1. This Thought/Action/Observation cycle repeats, allowing the agent to chain multiple tool uses if needed (search, then maybe a calculation, then another search, etc.). Eventually, the agent decides it can answer the user. At that point, instead of an Action, it outputs a Final Answer (sometimes marked as `Answer:` or `Final Answer:` in the format).

As we shall see shortly in the implementation from scratch, throughout this process, the agent maintains the conversation and its own intermediate steps.

Each Thought and Observation can be appended to the dialogue context so the LLM remembers what it has done so far.



This is crucial for coherence. The end result is that the agent effectively plans its approach on the fly, mixing reasoning and acting.

This dynamic approach is much more adaptable than a rigid script or a single-turn response. It allows handling unforeseen sub-tasks, similar to how humans adjust plans when new information comes up.

It's important to note that all these "Thought" and "Action" annotations are not magical features of the LLM—they come from how we prompt the model.

As we shall see below, we explicitly instruct the model to format its responses in this structured way. In other words, ReAct is implemented via carefully crafted prompt templates and parsing logic, not via any built-in LLM ability.

The LLM is guided to behave like an agent that reasons and acts, through the examples and instructions we give it.

Now that we understand the ReAct pattern conceptually, we can start building our own agent that follows this logic. We'll need a language model to serve as the agent's brain, some tools the agent can use, and a loop that ties them together.

In the next section, we'll step away from CrewAI and build a ReAct agent from scratch—in pure Python, using only local LLMs and a simple set of tool definitions. You'll see that everything we covered here is not magic—it's just smart prompt design combined with controlled I/O.

> 💡 Make sure you have created a `.env` file with the `OPENAI_API_KEY` specified in it. It will make things much easier and faster for you. Also, add these two lines of code to handle asynchronous operations within a Jupyter Notebook environment, which will allow us to make asynchronous calls smoothly to your Crew Agent.

ReAct Implementation from Scratch

Below, we shall implement a ReAct Agent in two ways:

- Manually executing each step for better clarity.
- Without manual intervention to fully automate the Reasoning and Action process.

You can download the code below:

react_notebook

react_notebook.ipynb • 12 KB

Let's look at the manual process first!

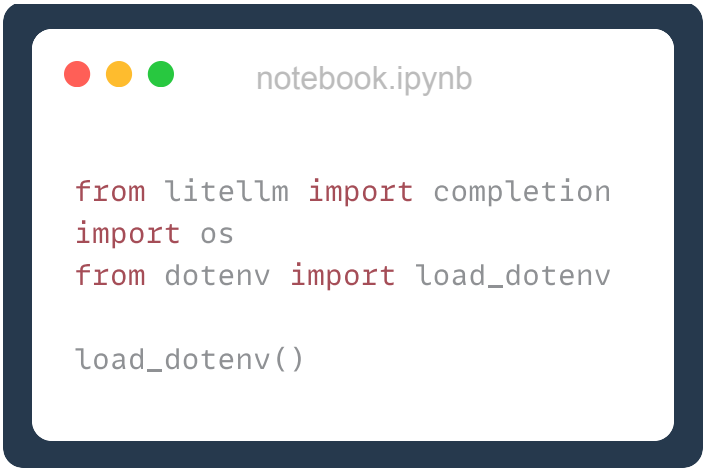
#1) ReAct with manual execution

In this section, we'll implement a lightweight ReAct-style agent from scratch, without using any orchestration framework like CrewAI or LangChain.

We'll manually simulate each round of the agent's reasoning, pausing, acting, and observing—exactly as a ReAct loop is meant to function.

By running the logic cell-by-cell, we will gain full visibility and control over the thinking process, allowing us to debug and validate the agent's behavior at each step.

To begin, we load the environment variables (like your LLM API key) and import completion from LiteLLM (also install it first– `pip install litellm`), a lightweight wrapper to query LLMs like OpenAI or local models via Ollama.



Next, we define a minimal Agent class, which wraps around a conversational LLM and keeps track of its full message history—allowing it to reason step-by-step, access system prompts, remember prior inputs and outputs, and produce multi-turn interactions.

Here's what it looks like:

notebook.ipynb

```
class MyAgent:
    def __init__(self, system = ""):
        self.system = system
        self.messages = []
        if self.system:
            self.messages.append({"role": "system", "content": system})
```

- `system` (str): This is the system prompt that sets the personality and behavioral constraints for the agent. If passed, it becomes the very first message in the conversation—just like in OpenAI Chat APIs.
- `self.messages`: This list acts as the conversation memory. Every interaction—whether it's user input or assistant output—is appended to this list. This history is crucial for LLMs to behave coherently across multiple turns.
- If `system` is provided, it's added to the message list using the special `"role": "system"` identifier (we learned about this in **Part 5 of RAG crash course**). This ensures that every completion that follows is conditioned on the system instructions.

Next, we define a `complete` method in this class:

notebook.ipynb

```
class MyAgent:
    def __init__(self, system = ""):
        self.system = system
        self.messages = []
        if self.system:
            self.messages.append({"role": "system", "content": system})

    def complete(self, message=""):
        if message:
            self.messages.append({"role": "user", "content": message})
        result = self.invoke()
        self.messages.append({"role": "assistant", "content": result})
        return result
```

This is the core interface you'll use to interact with your agent.

- If a `message` is passed:
  - It gets appended as a `"user"` message to `self.messages`.
  - This simulates the human asking a question or giving instructions.
- Then, `self.invoke()` is called (which we will define shortly). This method sends the full conversation history to the LLM.
- The model's reply (stored in `result`) is then appended to `self.messages` as an `"assistant"` role.
- Finally, the reply is returned to the caller.

This method does three things in one call:

1. Records the user input.
2. Gets the model's reply.
3. Updates the message history for future turns.

Finally, we have the `invoke` method below:

notebook.ipynb

```
class MyAgent:
    def __init__(self, system = ""):
        self.system = system
        self.messages = []
        if self.system:
            self.messages.append({"role": "system", "content": system})

    def complete(self, message=""):
        if message:
            self.messages.append({"role": "user", "content": message})
        result = self.invoke()
        self.messages.append({"role": "assistant", "content": result})
        return result

    def invoke(self):
        llm_response = completion(model="openai/gpt-4o", messages=self.messages)
        return llm_response.choices[0].message.content
```

This method handles the actual API call to your LLM provider—in this case, via LiteLLM, using the `"openai/gpt-4o"` model.

- `completion()` is a wrapper around the chat completion API. It receives the entire message history and returns a response.
- We assume `completion()` returns a structure similar to OpenAI's format: a list of choices, where each choice has a `.message.content` field.
- We extract and return that content—the assistant's next response.

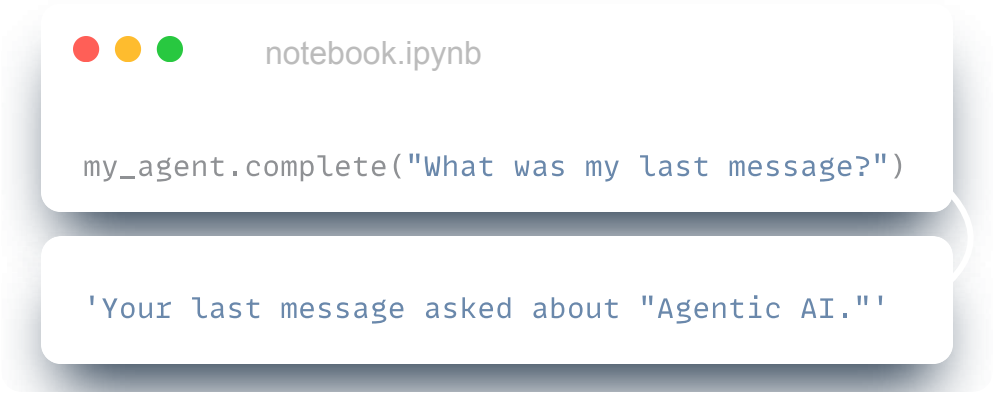
As a test, we can quickly run a simple interaction below:

notebook.ipynb

```
my_agent = MyAgent(system="You are a helpful assistant.")

my_agent.complete("What is Agentic AI?")
```

```
"""
Agentic AI refers to artificial intelligence systems that
exhibit behaviors or characteristics typically associated
with agency, which is the capacity to act autonomously and
make decisions.
"""
```

At this stage, if we ask it about the previous message, we get the correct output, which shows the assistant has visibility on the previous context:



It correctly remembers and reflects!

Now that our conversational class is setup, we come to the most interesting part, which is defining a ReAct-style prompt.

Before an LLM can behave like an agent, it needs clear instructions—not just on what to answer, but how to go about answering. That's exactly what this `system_prompt` does, which is defined below:

notebook.ipynb

```
system_prompt = """
You run in a loop and do JUST ONE thing in a single iteration:

1) "Thought" to describe your thoughts about the input question.
2) "PAUSE" to pause and think about the action to take.
3) "Action" to decide what action to take from the list of actions available to you.
4) "PAUSE" to pause and wait for the result of the action.
5) "Observation" will be the output returned by the action.

At the end of the loop, you produce an Answer.

The actions available to you are:

math:
e.g. math: (14 * 5) / 4
Evaluates mathematical expressions using Python syntax.

lookup_population:
e.g. lookup_population: India
Returns the latest known population of the specified country.

Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:
Thought: I need to find the population of Japan first.

Iteration 2:
PAUSE

Iteration 3:
Action: lookup_population: Japan

Iteration 4:
PAUSE

(you will now receive an output from the action)

Iteration 5:
Observation: 125,000,000

Iteration 6:
Thought: I now need to multiply it by 2.

Iteration 7:
Action: math: 125000000 * 2

Iteration 8:
PAUSE

(you will now receive an output from the action)

Iteration 9:
Observation: 250000000

Iteration 10:
Answer: Double the population of Japan is 250 million.
```

```
Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:
""".strip()
```

This isn't just a prompt. It's a behavioral protocol—defining what structure the agent should follow, how it should reason, and when it should stop.

Let's break it down line by line.

> 👉 You run in a loop and do JUST ONE thing in a single iteration:

This is the framing sentence. It tells the LLM not to rush toward an answer. Instead, it should proceed step by step, following a defined pattern in a *loop*—mirroring how a ReAct agent works.

> 👉 1) "Thought" to describe your thoughts about the input question.
> 2) "PAUSE" to pause and think about the action to take.
> 3) "Action" to decide what action to take from the list of actions available to you.
> 4) "PAUSE" to pause and wait for the result of the action.
> 5) "Observation" will be the output returned by the action.

Here, we give the LLM a reasoning template. These are the same primitives found in all ReAct-style agents.

Let's break each down:

- Thought: The agent's internal monologue. What is it currently thinking about?
- PAUSE (1): Instead of jumping to action, this forces the model to take a breath —simulating asynchronous steps in a multi-agent environment.

- Action: The agent picks from the list of tools it is given.
- PAUSE (2): Wait again, this time for the actual tool result.
- Observation: This will be injected into the prompt by *you* (the controller or human), after the tool runs.

By splitting this into explicit parts, we avoid hallucinations and ensure the agent works in a controlled loop.

👉 At the end of the loop, you produce an Answer.

This tells the agent: once it has all the required information—break the loop and give the final answer. No need to keep reasoning indefinitely.

👉 The actions available to you are:

math:
e.g. math: (14 * 5) / 4
Evaluates mathematical expressions using Python syntax.

lookup_population:
e.g. lookup_population: India
Returns the latest known population of the specified country.

This is a mini API reference for the agent. We show:

- The name of each tool.
- How to invoke it.
- What kind of output it produces.

This is critical. Without a clear spec, the LLM might:

- Invent non-existent tools.
- Use incorrect syntax.
- Misinterpret what the tool is supposed to do.

By using clear formatting and examples, we teach the model how to interface with tools in a safe, predictable way.

👉 Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:
Thought: I need to find the population of Japan first.

Iteration 2:
PAUSE

...

Iteration 9:
Observation: 250000000

Iteration 10:
Answer: Double the population of Japan is 250 million.

This worked-out example gives the LLM a pattern to follow. Even more importantly, it provides the developer (you) a way to intervene at each step—injecting tool results or validating whether the flow is working correctly.

With this sample trace:

- The agent knows how to think.
- The agent knows how to act.
- The agent knows when to stop.

👉 Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:

These closing lines are essential.

Without this explicit stop signal, the LLM might continue indefinitely. You're telling it: "When you have all the puzzle pieces, just say the answer and exit the loop."

The power of this `system_prompt` lies in its structure:

- It models intelligent behavior, not just question answering.
- It imposes strong constraints: think before acting, act within defined bounds, and wait for observations.
- It separates reasoning from execution, mimicking how humans operate.
- It creates a feedback-friendly iteration loop for multi-step problems.

Now that the prompt is defined, we implement the tools

notebook.ipynb

```
def math(expression: str):
    return eval(expression)

def lookup_population(country: str):
    populations = {
        "India": 1_400_000_000,
        "Japan": 125_000_000,
        "United States": 330_000_000,
        "Brazil": 210_000_000,
        "Indonesia": 270_000_000,
        "Mexico": 126_000_000,
        "Russia": 145_000_000,
        "United Kingdom": 67_000_000
    }
    return populations.get(country, "Country not found")
```

Finally, we begin a manual ReAct session:

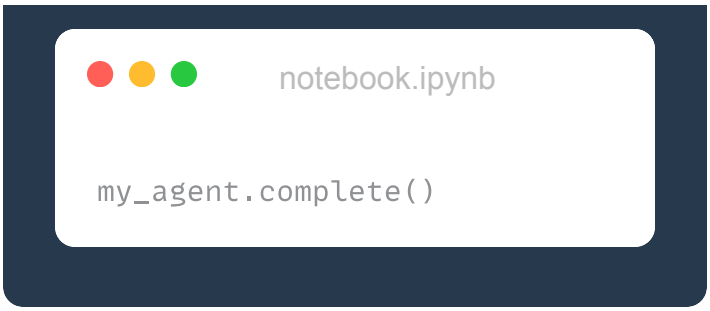
notebook.ipynb

```
my_agent = MyAgent(system=system_prompt)

my_agent.complete("""What is the population of India
                     plus the population of Japan?""")
```

This produces the following output:

👉 Iteration 1:
Thought: I need to find the population of India first.

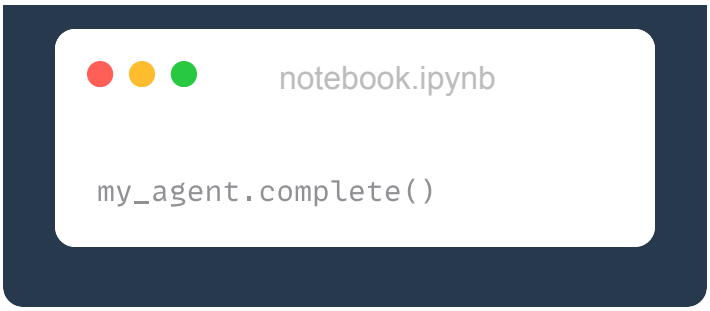
We, as a user, don't have any input to give at this stage so we just invoke the `complete()` method again:



This produces the following output:

👉 Iteration 2:
PAUSE

Yet again, we, as a user, don't have any input to give at this stage so we just invoke the `complete()` method again:

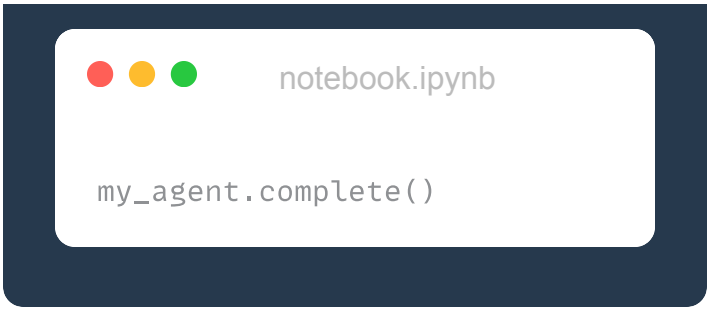


This produces the following output:

👉 Iteration 3:
Action: lookup_population: India

Now it wants to act.

We still don't have any input to give at this stage so we just invoke the `complete()` method again:



This produces the following output:

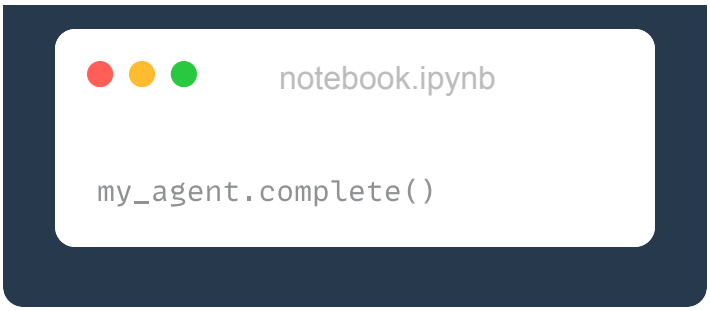
> 👉 Iteration 4:
> PAUSE

At this stage, it's needs to get the tool output in the form of an observation. Here, let's intervene and provide it with the observation:

This produces the following output:

👉 Iteration 5:
Thought: Now I need to find the population of Japan.

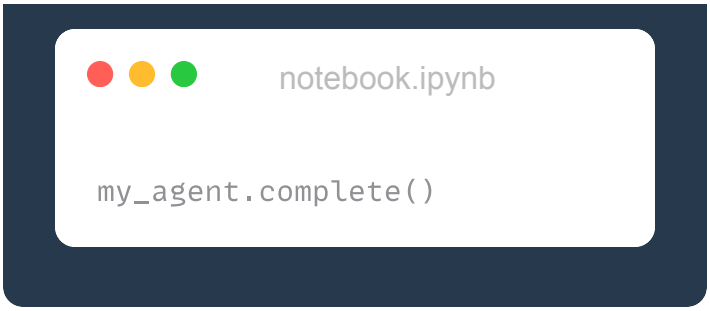
We let it continue its execution:



This produces the following output:

👉 Iteration 6:
PAUSE

We again let it continue its execution:



We get the following output:

👉 Iteration 7:
Action: lookup_population: Japan

At this stage, it's needs to get the tool output in the form of an observation. Here, let's again intervene and provide it with the observation:

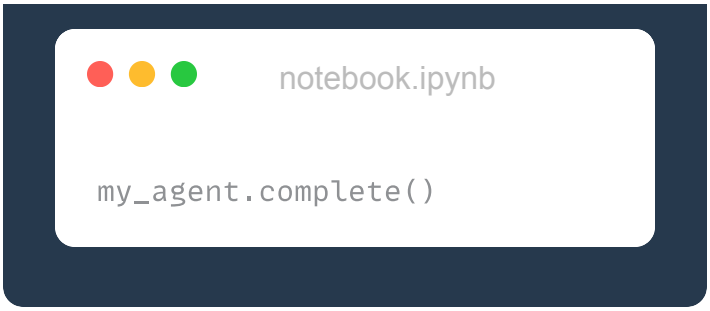
notebook.ipynb

```
my_agent.complete(f"Observation: {lookup_population('Japan')}")
```

This produces the following output:

👉 Iteration 8:
Thought: I now have the populations of both India and Japan. I need to add them together.

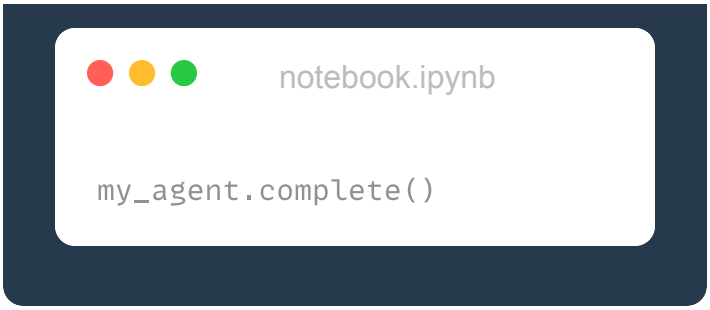
We again let it continue its execution:



We get the following output:

👉 Iteration 9:
Action: math: 1400000000 + 125000000

Now we should expect a pause according to the pattern specified:



👉 Iteration 10:
PAUSE

It is again seeking an observation, which is the sum of Japan's population and India's population. To do this, we again manually intervene and provide it with the output:

notebook.ipynb

```
my_agent.complete(f"Observation: {math('125000000 + 1400000000')}")
```

Finally, in this iteration, we get the following output:

> 👉 Iteration 11:
> Answer: The sum of the population of India and the population of Japan is 1,525,000,000.

Great!!

With this process:

- The LLM thought about what steps to take.
- It chose actions to execute.
- We manually injected tool outputs like real-world observations.
- It looped until it had enough information to generate a final answer.

This gives us an explicit understanding of how reasoning and actions come together in ReAct-style agents.

In the next part, we'll fully automate this—no manual calls required—and build a full controller that simulates this entire loop programmatically.

#2) ReAct without manual execution

Now that we have understood how the above ReAct execution went, we can easily automate that to remove our interventions.

In this section, we'll create a controller function that:

- Sends an initial question to the agent,
- Reads its thoughts and actions step-by-step,
- Automatically runs external tools when asked,
- Feeds back observations to the agent,
- And stops the loop once a final answer is found.

This is the entire code that does this:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):

    my_agent = MyAgent(system=system_prompt)

    available_tools = {"math": math,
                       "lookup_population": lookup_population}

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:
        llm_response = my_agent.complete(current_prompt)
        print(llm_response)

        if "Answer" in llm_response:
            break

        elif "Thought:" in llm_response:
            previous_step = "Thought"
            current_prompt = ""

        elif "PAUSE" in llm_response and previous_step == "Thought":
            current_prompt = ""
            previous_step = "PAUSE"

        elif "Action:" in llm_response:
            previous_step = "Action"
            pattern = r"Action:\s*(\w+):\s*(.+)"

            match = re.search(pattern, llm_response)

            if match:
                chosen_tool = match.group(1)
                arg = match.group(2)

                if chosen_tool in available_tools:
                    observation = available_tools[chosen_tool](arg)
                    current_prompt = f"Observation: {observation}"

                else:
                    current_prompt = f"Observation: Tool not available. Retry the action."

            else:
                observation = "Observation: Tool not found. Retry the action."
```

Let’s break down the full loop.

We begin by defining the `agent_loop()` function:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
```

It takes:

- `query`: the user's natural language question.
- `system_prompt`: the same ReAct system prompt we explored earlier (defining the behavior loop).

Next, inside this function, wnitialize the Agent and available tools:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }
```

- Create a new `MyAgent` instance, using the structured ReAct prompt.
- Define the dictionary of callable tools available to the agent. These names must match exactly what the agent uses in its `Action:` lines.

Moving on, we defined some state variables:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""
```

- `current_prompt` stores the next message to be sent to the LLM.
- `previous_step` helps track the last stage (e.g., Thought, Action) for better control flow.

Next, we run the reasoning loop, which continues until the agent produces a final answer. The answer is expected to be marked with `Answer:` based on our prompt design:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:
```

Next, we feed the `current_prompt` into the agent.

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        llm_response = my_agent.complete(current_prompt)
        print(llm_response)
```

The `current_prompt` could be:

- The initial user query,
- A blank string to let the agent continue reasoning,
- An observation from a tool.

We then print the agent's output—so we can inspect each iteration.

Next, if the agent produces a final answer, we break the loop.

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        llm_response = my_agent.complete(current_prompt)
        print(llm_response)

        if "Answer" in llm_response:
            break
```

In another case, if the response includes a `Thought:` line, we:

- Record the step type as "Thought".

- Set `current_prompt` to an empty string to continue to the next stage (a PAUSE).

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        llm_response = my_agent.complete(current_prompt)
        print(llm_response)

        if "Answer" in llm_response:
            break

        elif "Thought:" in llm_response:
            previous_step = "Thought"
            current_prompt = ""
```

Next, we catch the first PAUSE right after the Thought. Nothing else needs to be done here—we just move to the next step.

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        llm_response = my_agent.complete(current_prompt)
        print(llm_response)

        if "Answer" in llm_response:
            break

        elif "Thought:" in llm_response:
            previous_step = "Thought"
            current_prompt = ""

        elif "PAUSE" in llm_response and previous_step == "Thought":
            current_prompt = ""
            previous_step = "PAUSE"
            continue
```

If we detect an `Action:` line, we:

- Note that we're in the action step.
- Use a regex to extract the tool name and its argument.

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        ...

        elif "Action:" in llm_response:
            previous_step = "Action"
            pattern = r"Action:\s*(\w+):\s*(.+)"

            match = re.search(pattern, llm_response)
```

For example, in: `Action: lookup_population: India`, the regex pulls out:

- `lookup_population` as the tool.
- `India` as the argument.

Moving on, we execute the tool and capture the observation:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt: str = ""):
    my_agent = MyAgent(system=system_prompt)

    available_tools = {
        "math": math,
        "lookup_population": lookup_population
    }

    current_prompt = query
    previous_step = ""

    while "ANSWER" not in current_prompt:

        ...

        elif "Action:" in llm_response:
            previous_step = "Action"
            pattern = r"Action:\s*(\w+):\s*(.+)"

            match = re.search(pattern, llm_response)

            if match:
                chosen_tool = match.group(1)
                arg = match.group(2)

                if chosen_tool in available_tools:
                    observation = available_tools[chosen_tool](arg)
                    current_prompt = f"Observation: {observation}"

                else:
                    current_prompt = f"Observation: Tool unavailable. Retry."
```

- If the tool name is valid, we call it like a Python function and capture the result.
- We format the output into `Observation: ...` so the agent can use it in the next step.
- If the tool doesn't exist, we ask the agent to retry.

This mimics tool execution + response injection.

Done!

Now we can run this function as follows:

notebook.ipynb

```
agent_loop("""What is the population of India
              plus the population of Japan?""",
           system_prompt)
```

This produces the following output, which is indeed correct:

```
agent_loop("What is the population of India plus the population of Japan?", system_prompt)
✓ 7.9s

Iteration 1:
Thought: I need to find the population of India first.
Iteration 2:
PAUSE
Iteration 3:
Action: lookup_population: India
tool found!!
Iteration 4:
Thought: I have the population of India. Now I need to find the population of Japan.
Iteration 5:
PAUSE
Iteration 6:
Action: lookup_population: Japan
tool found!!
Iteration 7:
Thought: I now have the populations of both India and Japan. I need to add these two numbers together to get
Iteration 8:
Action: math: 1400000000 + 125000000
tool found!!
Iteration 9:
Answer: The combined population of India and Japan is 1,525,000,000.
```

You now have a fully working ReAct loop—without needing any external framework.

Of course, In this implementation, we're using regex matching and hardcoded conditionals to parse the agent's actions and route them to the correct tools.

This approach works well for a tightly controlled setup like this demo. However, it's brittle:

- If the agent slightly deviates from the expected format (e.g., adds extra whitespace, uses different casing, or mislabels an action), the regex could fail to match.
- We're also assuming that the agent will never call a tool that doesn't exist, and that all tools will succeed silently.

In a production-grade system, you'd want to:

- Add more robust parsing (e.g., structured prompts with JSON outputs or function calling).
- Include tool validation, retries, and exception handling.
- Use guardrails or output formatters to constrain what the LLM is allowed to emit.

But for the purpose of understanding how ReAct-style loops work under the hood, this is a clean and minimal place to start. It gives you complete transparency into what's happening at each stage of the agent's reasoning and execution process.

This loop demonstrates how a simple agent can think, act, and observe, all powered by your own Python + local LLM stack.

Conclusion

In this tutorial, we went beyond using frameworks and built a fully autonomous ReAct-style agent from scratch using plain Python and an LLM backend.

We covered:

- The ReAct loop pattern (Thought → Action → Observation → Answer), which powers intelligent decision-making in many agentic systems.

- How to structure a system prompt that teaches the LLM to think step-by-step and call tools deterministically.
- How to implement a lightweight agent class that keeps track of conversations and interfaces with the LLM.
- A fully manual ReAct loop for transparency and debugging.
- A fully automated `agent_loop()` controller that parses the agent's reasoning and executes tools behind the scenes.

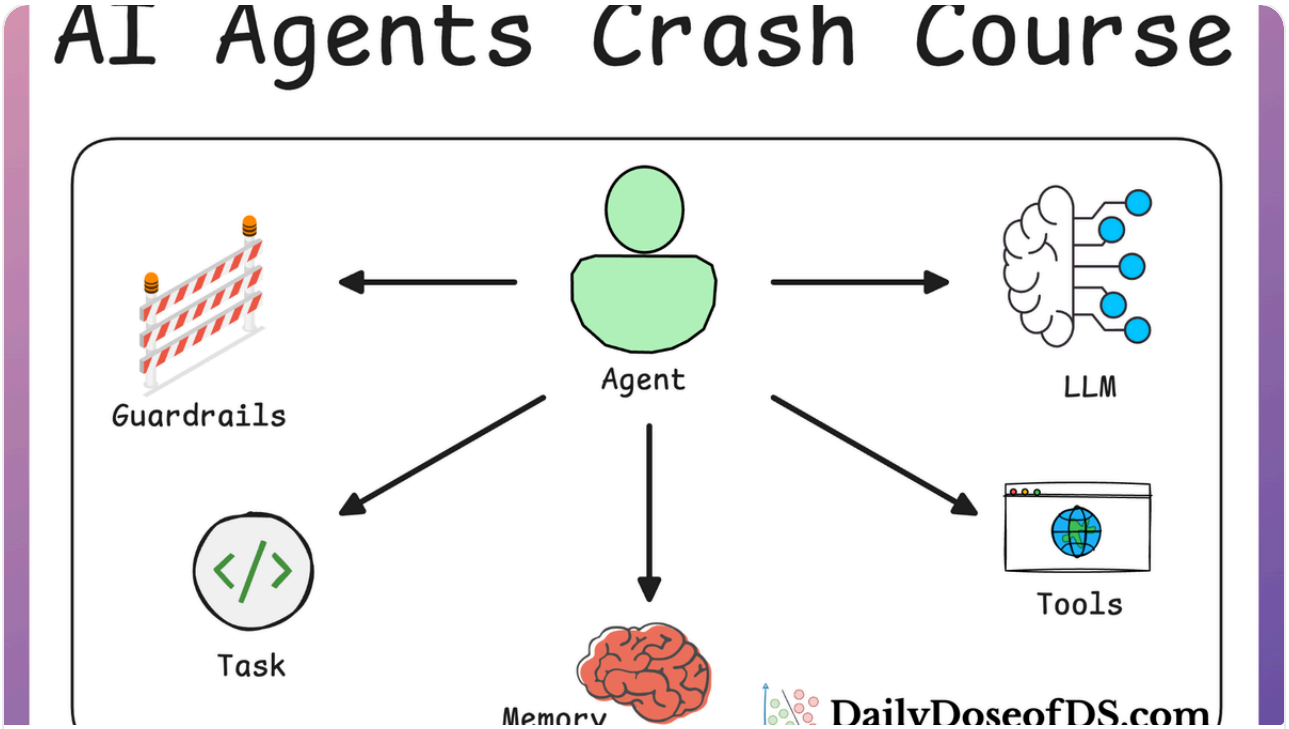
We also saw how you can plug in custom tools, run each iteration, and observe the inner workings of your agent—all without relying on any orchestration library like CrewAI, LangChain, or LlamaIndex.

And while this controller uses a few brittle shortcuts like regex parsing and hardcoded tool maps, it's an excellent foundation for building:

- Research agents
- Data wranglers
- Local retrieval systems
- Autonomous assistants

Once you're comfortable here, you'll be able to port this logic into larger projects or LLM frameworks—but with a much deeper understanding of how they actually work under the hood.

Read the next part of this crash course here:



Implementing Planning Agentic Pattern From Scratch

AI Agents Crash Course—Part 11 (with implementation).

Daily Dose of Data Science • Avi Chawla

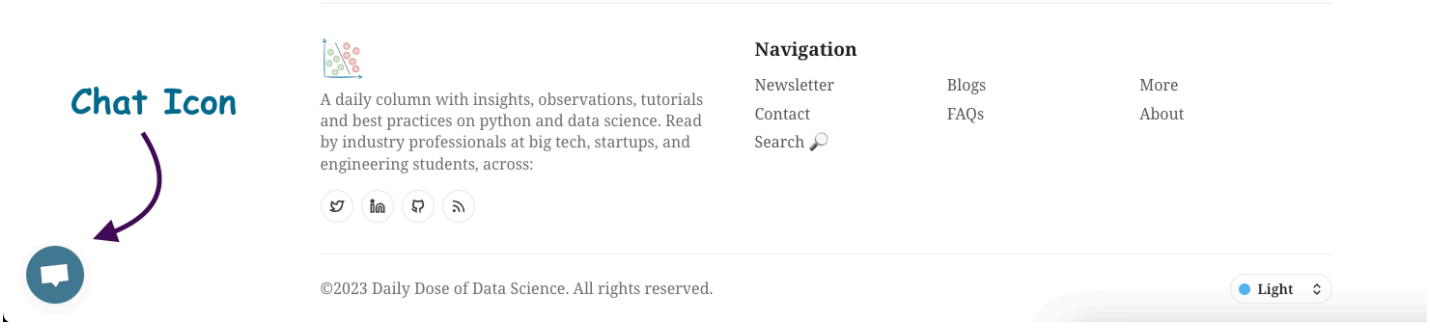
As always, thanks for reading!

Any questions?

Feel free to post them in the comments.

Or

If you wish to connect privately, feel free to initiate a chat here:



Connect via chat

Agents LLMs AI Agent Crash Course

Share this article

Read next

MCP Jun 15, 2025 • 22 min read

The Full MCP Blueprint: Building a Full-Fledged MCP Workflow using Tools, Resources, and Prompts

Model context protocol crash course—Part 4.

Avi Chawla, Akshay Pachaar

Agents Jun 8, 2025 • 20 min read

The Full MCP Blueprint: Building a Custom MCP Client from Scratch

Model context protocol crash course—Part 3.

Avi Chawla, Akshay Pachaar

Agents Jun 1, 2025 • 22 min read

The Full MCP Blueprint: Background, Foundations, Architecture, and Practical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar

Navigation