

Vizuara AI Agents Bootcamp Day 5

Smolagents: The simplest AI Agent coding library



VIZUARA AI
JUN 26, 2025



6



1



Share

In this post, we'll break down the key takeaways from Day 5 of the Vizuara AI Agents Bootcamp. Day 5 was all about SmolAgents, a new framework called “*the simplest agents library*.”



The banner features a dark green background. At the top center is the Vizuara AI logo, a stylized flower with pink, blue, and yellow petals. To the right, a white 'LIVE' badge is positioned above a circular video feed showing a man with dark hair and a beard. The main text 'Smolagents: Simplest Agent Coding Library' is displayed in large white and yellow font. Below this, a purple pill-shaped button contains the text 'BOOTCAMP DAY 5'. At the bottom, a row of logos includes LangGraph, smolagents (a yellow robot head), n8n (a pink pill with a white logo), crewAI (a red and white pill with a logo), and OpenAI (a blue pill with a white logo).

Table of contents

1. *SmolAgents at a Glance: Many Agents, One Framework*

2. *Code vs JSON: SmolAgents' Code-First Philosophy*
3. *Inside the CodeAgent Loop: How Does It Work?*
4. *ToolCallingAgent vs CodeAgent: A Quick Comparison*
5. *Vision Agents*
6. *LangFuse*
7. *Wrapping up Day 5*

(1) SmolAgents at a Glance: Many Agents, One Framework

SmolAgents keeps things simple but versatile. It provides a range of agent types and capabilities under one roof:

- **Vision + Browser Agents:** Agents that can interpret visual data (images/screenshots) and even navigate the web. *Imagine an agent that “sees” a webpage or an image and interacts with it like a human would!* This opens up possibilities for UI automation and image understanding.
- **Code Agents:** Agents that **write Python code** as actions to use tools and solve tasks. These agents leverage a language model to produce code (for example, calling a function) which the framework executes. Code Agents are powerful because they can implement complex logic and multi-step operations within a single thought.
- **Tool-Calling Agents:** Agents that invoke tools through structured **JSON-like** or text instructions. This is the more classical approach where the AI outputs an action in a predefined format (like a JSON specifying which tool to use and with what input). It's more constrained than code execution, but easier to predict and parse.
- **Multi-Agents:** Support for orchestrating **multiple agents** together. One agent can manage or call other agents as tools (hierarchical agents). This enables complex

workflows where specialized sub-agents handle parts of a task, coordinated by manager agent.

- **Retrieval Agents:** Agents augmented with **retrieval abilities** – for example, pulling in information from a knowledge base or the web to ground their answers. This is essentially *RAG (Retrieval-Augmented Generation)* in agent form: the agent has a tool to search documents or vector databases and uses those results in its reasoning.
- **Built-in Tools:** A collection of handy **tools** that come with SmolAgents to get you started. These include things like web search, a Python code interpreter, and even a speech-to-text transcriber. Tools are basically functions an agent can call – and you can easily add custom ones. By having a good set of tools, SmolAgents ensure your agents can take actions in the world (like fetching web content or doing calculations).

(2) Code vs JSON: SmolAgent's Code-First Philosophy

CodeAct: LLM Agent using [Code] as Action



Think

I should calculate the phone price in USD for each country, then find the most cost-effective country.



Action

```
countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
    exchange_rate, tax_rate = lookup_rates(country)
    local_price = lookup_phone_price("xAct 1", country)
    converted_price = convert_and_tax(
        local_price, exchange_rate, tax_rate
    )
    shipping_cost = estimate_shipping_cost(country)
    final_price = estimate_final_price(converted_price, shipping_cost)
    final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)
```

Control & Data Flow of Code
Simplifies Complex Operation



s Required!



Environment 1.1, 0.19

Re-use `min` Function from Existing
Software Infrastructures (Python library)



Response

The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

A standout feature of SmolAgents is that it lets the AI agent call tools by writing code, not just by filling out a JSON schema. Traditional agent frameworks often have the model output a structured plan or JSON indicating something like: {"action": "Search", "input": "Python tutorials"}. SmolAgents takes a different route and asks the model to produce a snippet of Python code that directly calls the tool functions needed.

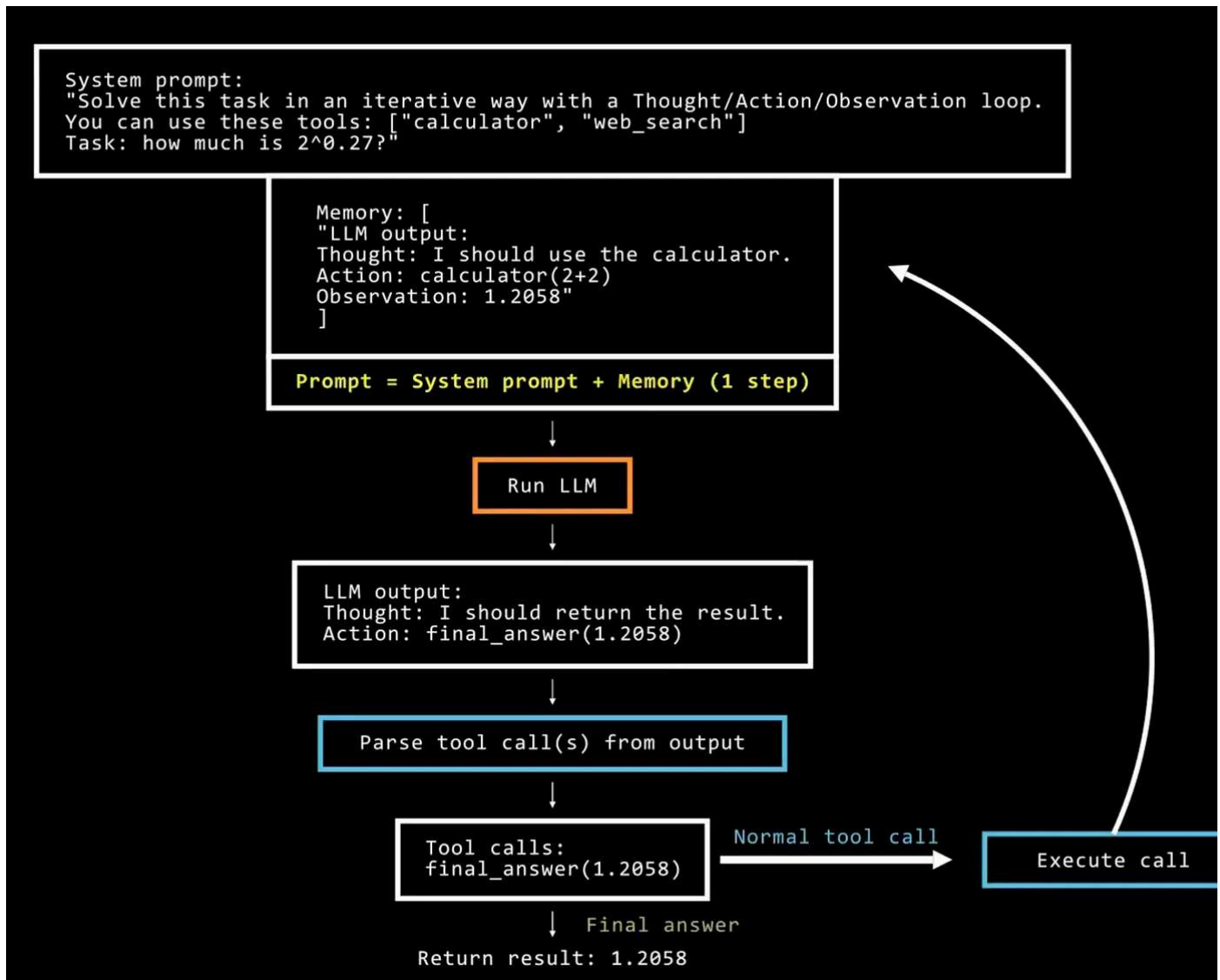
Why code? Writing actions as code has some big advantages:

- **More Flexibility:** The agent can use the full flexibility of Python for logic. It can loop, branch, or combine multiple tool calls in one go. For example, the agent could write a loop to perform several web searches and filter results – tasks that would be clunky as separate JSON calls.

- **Fewer Steps:** Because the agent can do more in one action, it often needs fewer iteration cycles to solve a problem. In fact, early benchmarks showed that code based agents can use ~30% fewer reasoning steps (and thus fewer LLM calls) while achieving better results.
- **Human-Like Thought Process:** When the AI writes code, it's almost like seeing its thought process in an executable form. It can be easier to trace what the agent is trying to do by reading the code, which can aid debugging or understanding agent's reasoning.
- **JSON Option Remains:** Of course, SmolAgents still supports the classic JSON-style tool calls via the **ToolCallingAgent**, which you might use for simpler or more constrained tasks. But the default and the star of the show is the **CodeAgent**.

The code-first approach does introduce new considerations: executing arbitrary code means you have to think about safety (SmolAgents includes options like secure interpreters or sandboxes for running code safely). But for many cases, the trade-off is worth it because of the boost in capability and efficiency. Next, let's peek into *how* a CodeAgent actually operates step by step.

(3) Inside the CodeAgent Loop: How Does It Work?



So, what happens under the hood when you run a SmolAgents CodeAgent? It's an iterative loop that follows the **ReAct paradigm** (Reason + Act) with a twist that actions are code. Here's the typical flow in a CodeAgent's cycle:

1. **Task Input & Setup:** You give the agent a task or question. This is added to the agent's memory (which stores the conversation and results so far). A system prompt is initialized, describing the agent's role and available tools.
2. **Thought/Action Generation:** The agent (via the LLM) generates the next action or code snippet, formatted as a Python code snippet. For example, it might produce code calling a `web_search("keyword")` or doing a calculation. This is essentially the "thought" turned into an executable action.

3. **Tool Execution:** The generated code is then executed by the agent. If the code calls a tool (like a search), that tool runs and returns output. The code could also contain simple logic (e.g., loops, if-statements) combining tool results.
4. **Observation & Memory Update:** The result of the tool call (the observation) is captured. The agent logs this outcome and appends it to its memory (so the transcript of what's happened so far is stored). The memory now includes: the initial question, the action taken, and the observation obtained.
5. **Repeat Loop:** With the updated context, the agent goes back to step 2. The LLM gets to see the conversation history including the latest result, and decides on the *next* action (again, as code). This loop of thought → code action → execution → observation continues, allowing the agent to make multi-step reasoning progress.
6. **Final Answer:** Eventually, the agent determines it has enough information or has completed the task. At that point, instead of calling a regular tool, it calls a special tool named `final_answer(answer_text)`. This is a signal that the agent is done. The argument to `final_answer` is the answer it wants to give. When the framework sees `final_answer` called, it ends the loop and returns that answer to you.

Throughout this process, the **agent's memory** ensures continuity – it “remembers” previous actions and their outcomes. This prevents it from repeating work or contradicting itself. If the agent hits an error (say the code had a bug or a tool failed) that error message can also be logged to memory so the AI can reconsider in the next step.

The result is an agent that incrementally reasons and gathers info, much like a person would solve a problem step-by-step. It's quite fascinating to watch a CodeAgent in action, because you see it writing and executing code in real-time to figure things out.

(4) ToolCallingAgent vs CodeAgent: A Quick Comparison

Instruction: Determine the most cost-effective country to purchase the smartphone model "CodeAct 1". The countries to consider are the USA, Japan, Germany, and India.

Available APIs

```

[1] lookup_rates(country: str) -> (float, float)
[2] convert_and_tax(price: float, exchange_rate: float, tax_rate: float) -> float
[3] estimate_final_price(converted_price: float, shipping_cost: float) -> float
[4] lookup_phone_price(model: str, country: str) -> float
[5] estimate_shipping_cost(destination_country: str) -> float

```

LLM Agent using [Text/JSON] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action `Text: lookup_rates, Germany`
`JSON: {"tool": "lookup_rates", "country": "Germany"}`

Environment 1.1, 0.19

Action `Text: lookup_phone_price, CodeAct 1, Germany`
`JSON: {"tool": "lookup_phone_price", "model": "CodeAct 1", "country": "Germany"}`

Environment 700

Action `Text: convert_and_tax, 700, 1.1, 0.19`
`JSON: {"tool": "convert_and_tax", "price": 700, "exchange_rate": 1.1, "tax_rate": 0.19}`

Environment 916.3

[... interactions omitted (look up shipping cost and calculate final price) ...]

Action `Text: lookup_rates, Japan`
`JSON: {"tool": "lookup_rates", "country": "Japan"}`

[... interactions omitted (calculate final price for all other countries) ...]

Response The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

CodeAct: LLM Agent using [Code] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action

```

countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
    exchange_rate, tax_rate = lookup_rates(country)
    local_price = lookup_phone_price("xAct 1", country)
    converted_price = convert_and_tax(
        local_price, exchange_rate, tax_rate
    )
    shipping_cost = estimate_shipping_cost(country)
    final_price = estimate_final_price(converted_price, shipping_cost)
    final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)

```

Environment 1.1, 0.19

Response The most cost-effective country to purchase the smartphone model is Ja with price 904.00 in USD.

Fewer Actions Required!

Control & Data Flow of C Simplifies Complex Oper

Re-use 'min' Function from Existing Software Infrastructures (Python library)

Both ToolCallingAgents and CodeAgents can accomplish complex tasks, but they have different strengths. Here's a side-by-side comparison of these two agent styles with SmolAgents:

Feature	ToolCallingAgent	CodeAgent
Complexity	Lower complexity	Higher complexity
Flexibility	Structured and predictable	Highly flexible and dynamic
Arbitrary code	No	Yes
Typical tasks	Simple, structured tasks	Complex, unstructured tasks
Error risk	Lower (predictable execution)	Higher (can have runtime errors)
Security implications	Safer (API calls only)	Needs caution (executes Python)

Both agents use the same underlying logic (they both are multi-step ReAct agents), in fact SmolAgents lets you choose whichever style suits your use case.

If you want quick and safe prototyping, a ToolCallingAgent might suffice. If you need the agent to truly *think in code* and handle complex sequences in one shot, the CodeAgent is your friend.

Often, the extra power of code comes with the responsibility to handle it safely – but SmolAgents provides tools to help, like secure execution modes.

(5) Vision Agents

Finally, we hinted at **Vision Agents** in the session – agents that can see. This means the agent can accept image inputs or use a visual perception tool as part of its toolbox. For instance, a vision-enabled agent could analyze a chart image you give it, or control a web browser by interpreting screenshots of a page (just like a human scanning a webpage).

SmolAgents supports vision models, which allows agents to tackle tasks that involve images or a visual environment.

(6) LangFuse

Trace692c45d35699879a05b439808f25c305

^K

▼J

↕↗

Search (ty)Timeline

ToolCallingAgent.run

10.89s

↔

ToolCallingAgent.run

ERROR

10.89s

✚

InferenceClientModel.generate

0.63s1188 → 31 (Σ 1219)

↔

DuckDuckGoSearchTool

ERROR

1.16s

✚

InferenceClientModel.generate

0.72s1273 → 29 (Σ 1302)

↔

DuckDuckGoSearchTool

ERROR

1.62s

✚

InferenceClientModel.generate

0.64s1358 → 29 (Σ 1387)

ToolCallingAgent.runID

+ Add to dataset

✎ Annotate

2025-06-26 15:01:04.896

Env: defaultLatency: 10.89s5,262 → 175 (Σ 5,437) ⓘ

Preview

Input

```
{
  task: "Search for the best book recommendations for implementing AI in the workplace."
  stream: false
  reset: true
  images: null
  additional_args: null
  max_steps: null
}
```

Output

```
null
```

Metadata

```
{
  attributes: {
```

We wrapped up Day 5 by exploring **Langfuse**, a powerful observability platform tailored for LLM agents. Using Langfuse, participants learned to track, visualize, and debug agent outputs effortlessly. By logging agent actions, tool calls, and reasoning steps, Langfuse provided deep insights into the behavior of our SmolAgents—helping us understand performance bottlenecks, logic flow, and areas for improvement. It's an invaluable companion when building robust, transparent AI agents.

(7) Wrapping up Day 5

The simplest AI Agents coding library: Smolagents



That's a wrap for Day 5.

SmolAgents shows that sometimes less is more – with a lean setup and letting the AI do the heavy lifting by writing code, we can build very capable agents. We covered the core idea of code vs JSON tool use, walked through how an agent reasons in loops, and compared the two agent styles.

Feel free to experiment with SmolAgents on your own – it's open-source and quite “smol” (the core is under 1,000 lines of code!)

See you in the next lecture!

Subscribe to Vizudara's Substack

Launched 7 months ago

My personal Substack

akshayredekar4441@gmai

Subscribe

By subscribing, I agree to Substack's [Terms of Use](#), and acknowledge its [Information Collection Notice](#) and [Privacy Policy](#).



6 Likes

Discussion about this post

Comments

Restacks



Write a comment...



Shanmukha Sasidhar Shanmukha Sasidhar 3m

pls provide colab link for running code



LIKE



REPLY

1