Large Language Models: Agents

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

Outline

1. What is an Agent?
2. Smolagents

Part 1: What is an agent?

What is an agent?

“An Agent is a system that leverages an AI model to interact with its environment in order to achieve a user-defined objective.

It combines reasoning, planning, and the execution of actions (often via external tools) to fulfill tasks.”

The Agent has two main parts:

- The **Brain** (AI Model). This is where all the thinking happens. The AI model handles reasoning and planning. It decides which Actions to take based on the situation.
- The **Body** (Capabilities and Tools). This part represents everything the Agent is equipped to do.

System prompts and messages

- The **system prompt** describes how the agent should behave. This information is persistent across messages

```
system_message = {
    "role": "system",
    "content": "You are a professional customer service agent. Always be polite, clear, and helpful."
}
```

- The **conversation** describes the ongoing interactions

```
conversation = [
    {"role": "user", "content": "I need help with my order"},
    {"role": "assistant", "content": "I'd be happy to help. Could you provide your order number?"},
    {"role": "user", "content": "It's ORDER-123"},
]
```

Chat templates

Each model has different conversation model and special tokens…

Solution:

- ChatML is the standard template format (see previous slide)
- Each model publishes a chat template in Jinja, which describes how to transform ChatML into text (see next slide)

Right: Jinja

```
{% for message in messages %}
{% if loop.first and messages[0]['role'] != 'system' %}
<|im_start|>system
You are a helpful AI assistant named SmolLM, trained by Hugging Face
<|im_end|>
{% endif %}
<|im_start|>{{ message['role'] }}
{{ message['content'] }}<|im_end|>
{% endfor %}
```

Left: ChatML

```
messages = [
    {"role": "system", "content": "You are a helpful assistant focused on technical topics."},
    {"role": "user", "content": "Can you explain what a chat template is?"},
    {"role": "assistant", "content": "A chat template structures conversations between users and AI model
    {"role": "user", "content": "How do I use it ?"},
]
```

Right: prompt

```
<|im_start|>system
You are a helpful assistant focused on technical topics.<|im_end|>
<|im_start|>user
Can you explain what a chat template is?<|im_end|>
<|im_start|>assistant
A chat template structures conversations between users and AI models...<|im_end|>
<|im_start|>user
How do I use it ?<|im_end|>
```

What is a tool?

A Tool is a function given to the LLM

Examples:

- Web Search: Fetch up-to-date information from the internet
- Image Generation: Create images based on text descriptions
- Retrieval:Retrieve information from an external source

A simple example

```python
@tool
def calculator(a: int, b: int) -> int:
    """Multiply two integers."""
    return a * b
```

This implementation contains four pieces of information:

- A descriptive name of what it does: “calculator”
- A longer description, provided by the function’s docstring comment: “Multiply two integers”
- The inputs and their type: the function expects two ints
- The type of the output

A MORE INTERESTING EXAMPLE

```python
@tool
def get_current_time_in_timezone(timezone: str) -> str:
    """A tool that fetches the current local time in a specified timezone.
    Args:
        timezone: A string representing a valid timezone (e.g., 'America/New_York').
    """
    try:
        # Create timezone object
        tz = pytz.timezone(timezone)
        # Get current time in that timezone
        local_time = datetime.datetime.now(tz).strftime("%Y-%m-%d %H:%M:%S")
        return f"The current local time in {timezone} is: {local_time}"
    except Exception as e:
        return f"Error fetching time for timezone '{timezone}': {str(e)}"
```

Good news! Model Context Protocol (MCP)

Model Context Protocol (MCP) is an open protocol that standardizes how applications provide tools to LLMs

MCP provides:

- A growing list of pre-built integrations that your LLM can directly plug into
- The flexibility to switch between LLM providers and vendors
- Best practices for securing your data within your infrastructure

This means that any framework implementing MCP can leverage tools defined within the protocol, eliminating the need to reimplement the same tool interface for each framework

A list of MCP reference servers

https://github.com/modelcontextprotocol/servers?tab=readme-ov-file#-reference-servers

Making the agent aware of the tools

Option 1: we simply add to the System Prompt the description of the tools (automatically generated from implementations):

```
system_message="""You are an AI assistant designed to help users efficiently and accurately. Your
primary goal is to provide helpful, precise, and clear responses.

You have access to the following tools:
Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int
"""
```

Option 2: we fine-tune a model with a dataset of problems

Thought-Action-Observation cycle (1/2)

Agents work in a continuous cycle:

- **Thought**: the agent decides what the next step should be
- **Action**: the agent takes an action, by calling the tools with the associated arguments
- **Observation**: the agent reflects on the response from the tool

Thought-Action-Observation cycle (2/2)

This cycle is explained in the System Prompt:

```
system_message="""You are an AI assistant designed to help users efficiently and accurately. Your
primary goal is to provide helpful, precise, and clear responses.

You have access to the following tools:
Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int

You should think step by step in order to fulfill the objective with a reasoning divided into
Thought/Action/Observation steps that can be repeated multiple times if needed.

You should first reflect on the current situation using `Thought: {your_thoughts}`, then (if
necessary), call a tool with the proper JSON formatting `Action: {JSON_BLOB}`, or print your final
answer starting with the prefix `Final Answer:`
"""
```

Step 1: Thought, the ReAct prompting technique

ReAct is the concatenation of “Reasoning” (Think) with “Acting” (Act)

Simplest idea ever: **ReAct** is a prompting technique that appends “Let’s think step by step” to the prompt

It encourages the Agent to decompose the problem into sub-tasks

Step 2: action

A very important distinction: how do agents take action?

- **Function-calling agents**: the agent specifies which tool to use and what are the arguments
- **Code agents**: the agent writes a piece of code that is interpreted internally
- **JSON agents**: the action specifies a tool for manipulating data in JSON files

Code agents >> Json agents?

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.

>

Fewer Actions Required!

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 Operations

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

Environment 1.1, 0.19

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

Step 3: observation

In the observation phase, the agent:

- **Collects feedback**: Receives data or confirmation that its action was successful (or not)
- **Appends results**: Integrates the new information into its existing context, effectively updating its memory
- **Adapts its strategy**: Uses this updated context to refine subsequent thoughts and actions

Part 2: Smolagents

Generalities

Many, many frameworks. Three mainstream examples:

- Smolagents (by Hugging Face)
- LangGraph (by LangChain)
- LlamaIndex (by Meta)

Smolagents is simple, lightweight, code-based

It supports both code agents and JSON agents

Architecture

There are typically three files:

- *prompts.yaml* which defines the System Prompt
- *tools.py* which contains all tools used by the Agent
- *app.py* which defines the Agent

The file prompts.yaml

- ***system_prompt***: describe the Thought, Code, and Observation cycle, ReAct, the tools, and possibly some examples (few shot)

- ***planning***: prompts for each step of the planning, encouraging to explicitly write and update facts and plans (*initial_facts*, *initial_plan*, *update_facts_pre_messages*, *update_facts_post_messages*, *update_plan_pre_messages*, *update_plan_post_messages*)

The file prompts.yaml

Two remarks:

- Typically use the generic prompts.yaml files that one can find online
- adding tasks using the concrete tools of the Agent helps

The file tools.py

Write your own

```python
@tool
def get_current_time_in_timezone(timezone: str) -> str:
    """A tool that fetches the current local time in a specified timezone.
    Args:
        timezone: A string representing a valid timezone (e.g., 'America/New_York').
    """
    try:
        # Create timezone object
        tz = pytz.timezone(timezone)
        # Get current time in that timezone
        local_time = datetime.datetime.now(tz).strftime("%Y-%m-%d %H:%M:%S")
        return f"The current local time in {timezone} is: {local_time}"
    except Exception as e:
        return f"Error fetching time for timezone '{timezone}': {str(e)}"
```

The file tools.py

Load from the Hub

```python
from smolagents import Tool

image_generator = Tool.from_space(
    space_id="black-forest-labs/FLUX.1-schnell",
    name="ImageGenerator",
    api_name="/infer",
    description="Generate an image from a prompt"
)
```

The file tools.py

Other options:

- from_gradio
- from_MCP
- from_langchain

The file app.py

This is just an example, without any tool

Agents need a solid LLM, Qwen-32B is a good starting point

```python
from smolagents import CodeAgent
import yaml

model = "Qwen/Qwen2.5-Coder-32B-Instruct"

with open("prompts.yaml", 'r') as stream:
    prompt_templates = yaml.safe_load(stream)

agent = CodeAgent(
    model=model,
    tools=[], # your list of tools
    max_steps=6,
    verbosity_level=1,
    grammar=None,
    planning_interval=None,
    name=None,
    description=None,
    prompt_templates=prompt_templates
)

agent.run("What time is it in Sydney?")
```

Memory

By default, the Agent forgets everything after each run, because this would be very expensive in terms of tokens

```
agent.run("What time is it in Sydney?")

agent.run("What was the last question I asked you?", reset = False)
```

Multi-agents

Multi-agent structures allow to separate memories between different sub-tasks, with two great benefits:

- Each agent is more focused on its core task, thus more performant
- Separating memories reduces the count of input tokens at each step, thus reducing latency and cost

```
manager_agent = CodeAgent(
    model="deepseek-ai/DeepSeek-R1",
    managed_agents=[web_agent],
)
```

A great reference

https://huggingface.co/learn/agents-course