Introduction to Agentic AI

-- LLM Agent Application Basics

Instructor: Guangjing Wang

guangjingwang@usf.edu

LLM Agent: Next Token Generation

How to make agents build a web browser?

Building an agent for browser interaction ›　　Answer now

2. **Vision-based**: The agent interacts with screenshots for more complex UIs (like OpenAI CUA), but it's slower and riskier.

The thinking process, tool calling process are inherently a next-token generation process: Previous generated tokens impact the next-token.

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, \ldots, w_{i-1})$$

Activity ×

Thinking

- Explaining web browser agent creation approaches
  When a user asks how to make agents build a web browser, they could mean either controlling a browser with automation tools like Selenium or developing a new browser UI from scratch. The more likely scenario involves using agents to operate a browser. I'll provide guidance on automation tools, architecture, and considerations.
- Explaining agent interaction with web browsers
  To help the user with building agents that interact with a web browser, I'll provide a Python example of using Playwright to control a browser. I'll mention common actions like navigate, click, type, and screenshot, and highlight safeguards like rate limits, sandboxing, and testing frameworks.
- Browsing for web automation agent frameworks and tools
  www.reuters.com　apnews.com　www.wired.com　10 more
- Building an agent for browser interaction
  To build a browsing agent, we can follow two approaches:
  1. **DOM-driven**: Using tools like Playwright or Selenium, the agent interacts with the browser via the DOM, offering reliability and predictability.
  2. **Vision-based**: The agent interacts with screenshots for more complex UIs (like OpenAI CUA), but it's slower and riskier.
  I'll include a step-by-step approach for both, focusing on tool selection, best practices, and defining APIs.

Decoding for Language Generation

- **Assumption:** the probability distribution of a token sequence can be decomposed into the product of conditional next token distributions:

$$P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_1, ..., w_{i-1})$$

- **Decoding Methods:**
  - **Greedy search**: select the token with the highest probability as its next token
  - **Beam search**: keep the most likely n_beam of candidates at each step and eventually choosing the sequence that has the overall highest probability.
  - **Sampling**: Temperature-based, Top-k-based, and Top-p-based

Sampling

$$\sigma(z_i/T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

- **T→0:** Pushing model toward "greedy" selection (choosing the highest probability).
- **T<1 (Low Temperature):** The distribution becomes sharper and increases the likelihood of high probability words and decreasing the likelihood of low probability words.
- **T > 1(High Temperature):** The distribution flattens, increasing entropy and making the output more diverse and less predictable.

Sampling when Setting Temperature as 0

- Way 1: Adding a small number, e.g., 1e-8

$$P_i = \frac{\exp(z_i/(T+\epsilon))}{\sum_j \exp(z_j/(T+\epsilon))}$$

- Way 2: triggering "greedy" selection (choosing the highest probability), resulting highly deterministic outputs.
- However, the output is still not deterministic when temperature is 0
  - Models adopt MOE: which expert processes which token during inference is unknown. This is generally decided in a batch level. We can mostly maintain batch level-consistency if you have the same batch every time.
  - Hardware issues: **butterfly effect -- a small change in one state of a deterministic nonlinear system can result in large differences in a later state. (A torando was influenced by a butterfly flapping its wings)**

Hardware Issues Causing the Uncertainty

- **Floating-Point Precision:** GPU operations (e.g., matrix multiplication) involving floating-point numbers can produce slightly different results due to rounding errors, which can change the highest-probability token.
- **Parallel Computation (Nondeterminism):** Operations in GPUs are often parallelized for speed, which can cause variations in the order of calculations. (1+1e16-1e16 = 0)
- **Tie-Breaking:** If multiple tokens have nearly identical top probabilities, minor numerical differences can cause the model to pick a different "most likely" token.
- **Backend Infrastructure:** Requests might be routed to different hardware instances or server that, due to differing load balancing, produce slightly different outputs.

Controlling Output

- **Maximum length**: The maximum length is the total # of tokens the AI is allowed to generate.
- **Stop sequences/Stop token**: These sequences tell the model when to stop generating output and helps to control content length and structure. Prompting an LLM to write an email using "Best regards," or "Sincerely," as a stop sequence tells the model to stop before the closing salutation.
- **Frequency penalty**: A frequency penalty is a setting that discourages repetition in the generated text by penalizing tokens proportionally to how frequently they appear. The more often a token is used in the text, the less likely the LLM is to use it again.
- **Presence penalty**: The presence penalty is similar to the frequency penalty, but penalizes tokens based on whether they have occurred or not rather than penalizing them proportionally.

LLM API Call

https://www.youtube.com/watch?v=pjH7fsXYh4w
https://www.youtube.com/watch?v=9ZyHckE3iIo&list=PLi7jtY2ZZqRZ_8uawO4GUx_yi_7-Psejl

LLM and Agent Framework

- unSloth: LLM Fine-tuning Framework
- LLAMA Factory: LLM Fine-tuning Framework based on TRL, Transformers libraries

- LangChain/LangGraph: Build LLM Agent Applications
- Dify: Agentic Workflow Builder
- LLamaIndex: Build LLM-powered Agents
- SmolAgents: A barebones library for agents that think in code

When to Use Agentic Framework

- An agentic framework is **not always needed when building an application around LLMs**.
- If the approach to build an agent is simple, like a chain of prompts, using plain code may be enough.
- Plain code makes you have full control and understanding of their system without abstractions.
- When the workflow becomes more complex, such as letting an LLM call functions or using multiple agents, these abstractions start to become helpful.

Key Components of Agentic Framework

- An LLM engine that powers the system.
- A list of tools the agent can access.
- A parser for extracting tool calls from the LLM output.
- A system prompt synced with the parser.
- A memory system.
- Error logging and retry mechanisms to control LLM mistakes.

Framework Example: Smolagents

- CodeAgents are the primary type of agent in smolagents;
- ToolcallingAgents: rely on JSON/text blobs that the system must parse and interpret to execute actions;
- RetrievalAgents: leverage vector stores for efficient retrieval and implement **Retrieval-Augmented Generation (RAG)** patterns;

CodeAgents in Smolagents Framework

```python
from smolagents import CodeAgent, InferenceClientModel
import numpy as np
import time
import datetime

agent = CodeAgent(tools=[], model=InferenceClientModel(), additional_authorized_imports=['datetime'])
agent.run("Give me the best playlist for a party at Wayne's mansion.
```

```
New run

Give me the best playlist for a party at the Wayne's mansion. The party idea
is a 'villain masquerade' theme

InferenceClientModel - Qwen/Qwen3-Next-80B-A3B-Thinking
Step 1
Output message of the LLM:
Thought: I need to find a playlist for a villain masquerade party at Wayne's
mansion. I'll start by searching for relevant playlists using web_search.

<code>
search_results = web_search(query="villain masquerade party playlist")
print(search_results)

— Executing parsed code:
search_results = web_search(query="villain masquerade party playlist")
print(search_results)

Execution logs:
## Search Results

[you're at a masquerade and unknowingly dancing with the
villain](https://open.spotify.com/playlist/4zxYvQs9gByjRhAn5ujSfQ)
you're at a masquerade and unknowingly dancing with the villain · Playlist · 89
```

Code Execution Using Local Python Executor in Smolagents Framework

Code execution in smolagents is a rebuilt secure LocalPythonExecutor

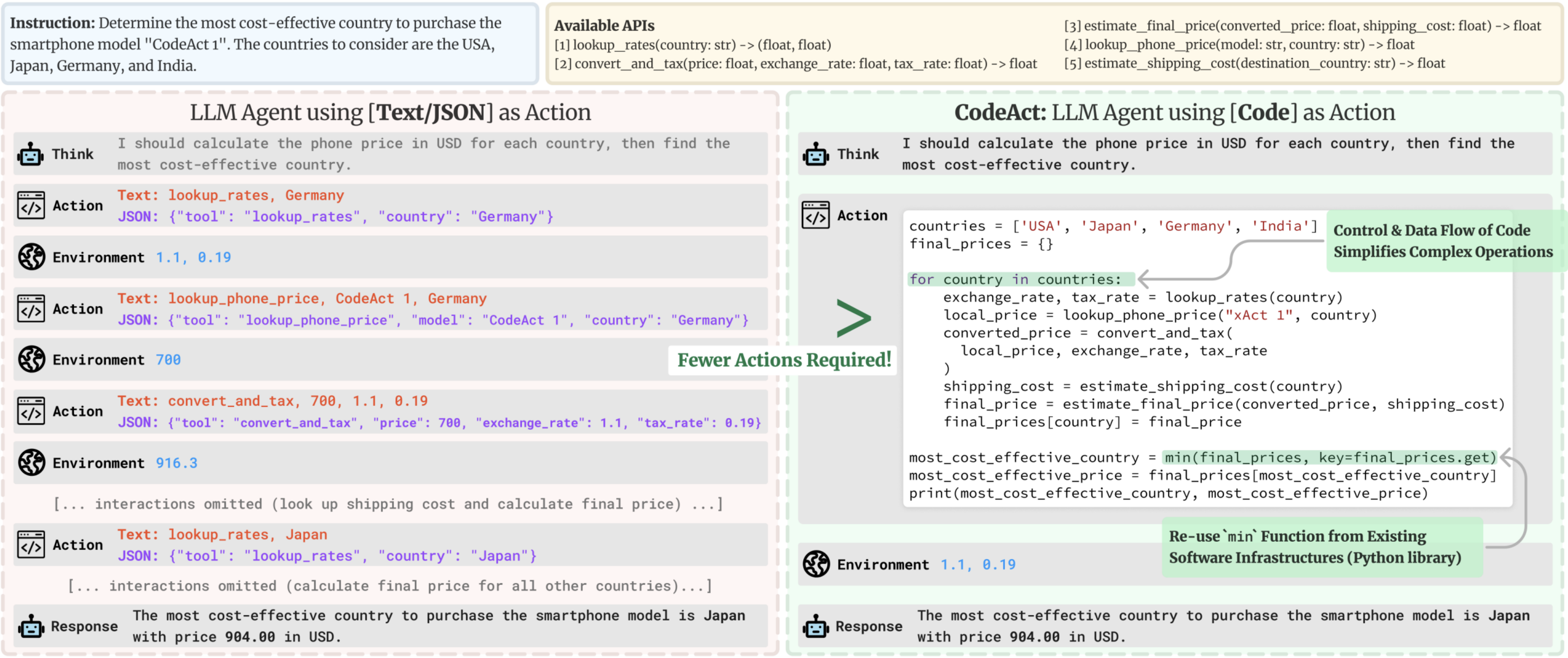
- This interpreter works by loading the Abstract Syntax Tree (AST) from Code and executes it operation by operation;
- Imports are disallowed unless they have been explicitly added to an authorization list by the user;
- Access to submodules is disabled by default, and each must be explicitly authorized in the import list.
- The total count of elementary operations processed is capped to prevent infinite loops and resource bloating.
- Any operation that has not been explicitly defined in the custom interpreter will raise an error.

Example of the Local Python Executor

```python
from smolagents.local_python_executor import LocalPythonExecutor

# Set up custom executor, authorize package "numpy"
custom_executor = LocalPythonExecutor(["numpy"])

# Utility for pretty printing errors
def run_capture_exception(command: str):
    try:
        custom_executor(harmful_command)
    except Exception as e:
        print("ERROR:\n", e)

# Undefined command just do not work
harmful_command="!echo Bad command"
run_capture_exception(harmful_command)
# >>> ERROR: invalid syntax (<unknown>, line 1)


# Imports like os will not be performed unless explicitly added to `additional_authorized_imports`
harmful_command="import os; exit_code = os.system('echo Bad command')"
run_capture_exception(harmful_command)
```

https://huggingface.co/docs/smolagents/tutorials/secure_code_execution
https://github.com/huggingface/smolagents/blob/v1.8.1/src/smolagents/local_python_executor.py#L1152

ToolCalling Method (Left) VS Direct Code Generation (Right)



ToolCallingAgents in SmolAgents Framework

- ToolCallingAgents are the second type of agent available in smolagents.
- Unlike CodeAgents that use Python snippets, these agents **use the built-in tool-calling capabilities of LLM providers** to generate tool calls as **JSON structures**.
- This is the standard approach used by OpenAI, Anthropic, and many other providers.
- LLM will simply generate an output in JSON format, like: {"function": "get_flight_details", "arguments": {"source": "Source", "destination": "Dest", "date": "date"}}.

Define a Tool in SmolAgents Framework

```
class HFModelDownloadsTool(Tool):
    name = "model_download_counter"
    description = """
    This is a tool that returns the most downloaded model of a given task on the Hugging Face Hub.
    It returns the name of the checkpoint."""
    inputs = {
        "task": {
            "type": "string",
            "description": "the task category (such as text-classification, depth-estimation, etc)",
        }
    }
    output_type = "string"

    def forward(self, task: str):
        from huggingface_hub import list_models

        model = next(iter(list_models(filter=task, sort="downloads", direction=-1)))
        return model.id

model_downloads_tool = HFModelDownloadsTool()
```

Key Info:

- name,
- tool description,
- input types,
- input descriptions,
- output type.

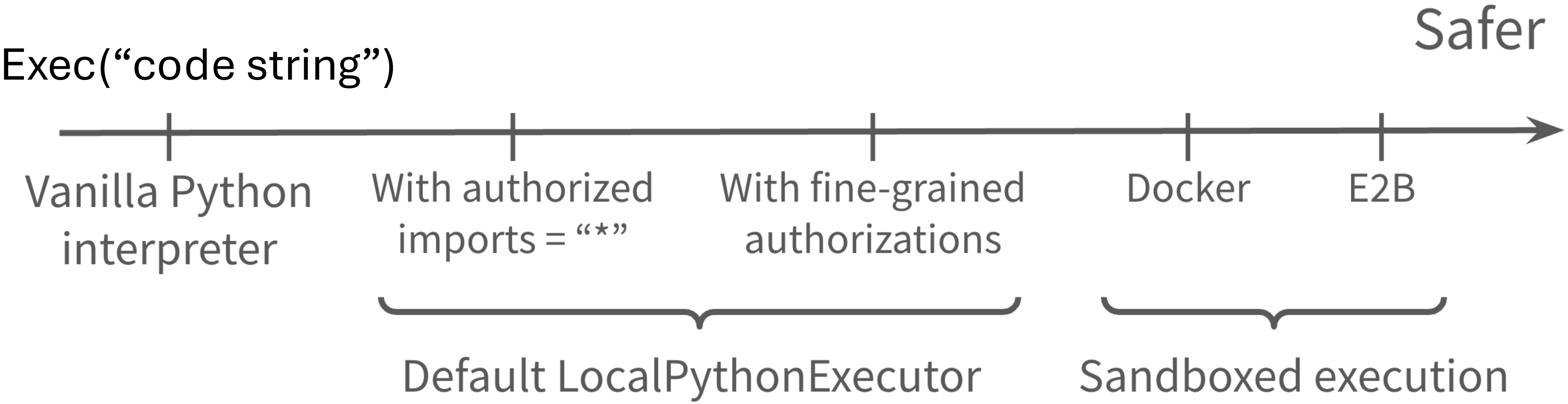
The Process of Running a Tool (1)

If a function relies on external functions, does the LLM execute the function within my environment, or do I need to explicitly execute it in my backend and return the result to the model?

- LLMs are "simple" text-in-text-out.
- The LLM will create the tool call but not execute it.
- For example, if you have a tool called "getCustomers" the LLM will generate the parameters (e.g., X, Y and Z) that are needed to call the tool.
- The actual tool call will be done by **you, or the framework** used to build the AI application.

Environments in Tool Execution

Exec("code string")



Vanilla Python Interpreter: Dynamic Code Execution

```python
>>> code = """
... numbers = [2, 3, 7, 4, 8]
...
... def is_even(number):
...     return number % 2 == 0
...
... even_numbers = [number for number in numbers if is_even(number)]
...
... squares = [number**2 for number in even_numbers]
...
... result = sum(squares)
...
... print("Original data:", numbers)
... print("Even numbers:", even_numbers)
... print("Square values:", squares)
... print("Sum of squares:", result)
... """

>>> exec(code)
Original data: [2, 3, 7, 4, 8]
Even numbers: [2, 4, 8]
Square values: [4, 16, 64]
Sum of squares: 84
```

https://realpython.com/python-exec/

Potential Security Issues

- **Plain LLM error:** LLMs may unintentionally generate harmful commands while attempting to be helpful.
- **Supply chain attack:** Running an untrusted or compromised LLM could expose a system to harmful code generation.
- **Prompt injection:** An agent browsing the web could arrive on a malicious website that contains harmful instructions, thus injecting an attack into the agent’s memory
- **Exploitation of publicly accessible agents:** Agents exposed to the public can be misused by malicious actors to execute harmful code. It can damage the file system, exploit local or cloud-based resources, abuse API services, and even compromise network security.

The Process of Running a Tool (2)

Does the LLM only suggest which function to call and with what arguments, leaving all execution to my system?

- Yes, that's actually what happens.
- The execution system could be anything from custom implementation of the SDK of your LLM provider (OpenAI, Anthropic etc) to a full-fledged agentic framework.

The Process of Running a Tool (3)

How do function-calling workflows handle complex cases like async API calls, multi chain calls or when the LLM generates malformed arguments?

- Most agentic frameworks have a way to help you construct tools [1] and help the LLM understand better how to use these tools.
- Things like async API calls will be covered within the framework, and the definition of arguments is defined in a typesafe way [2].

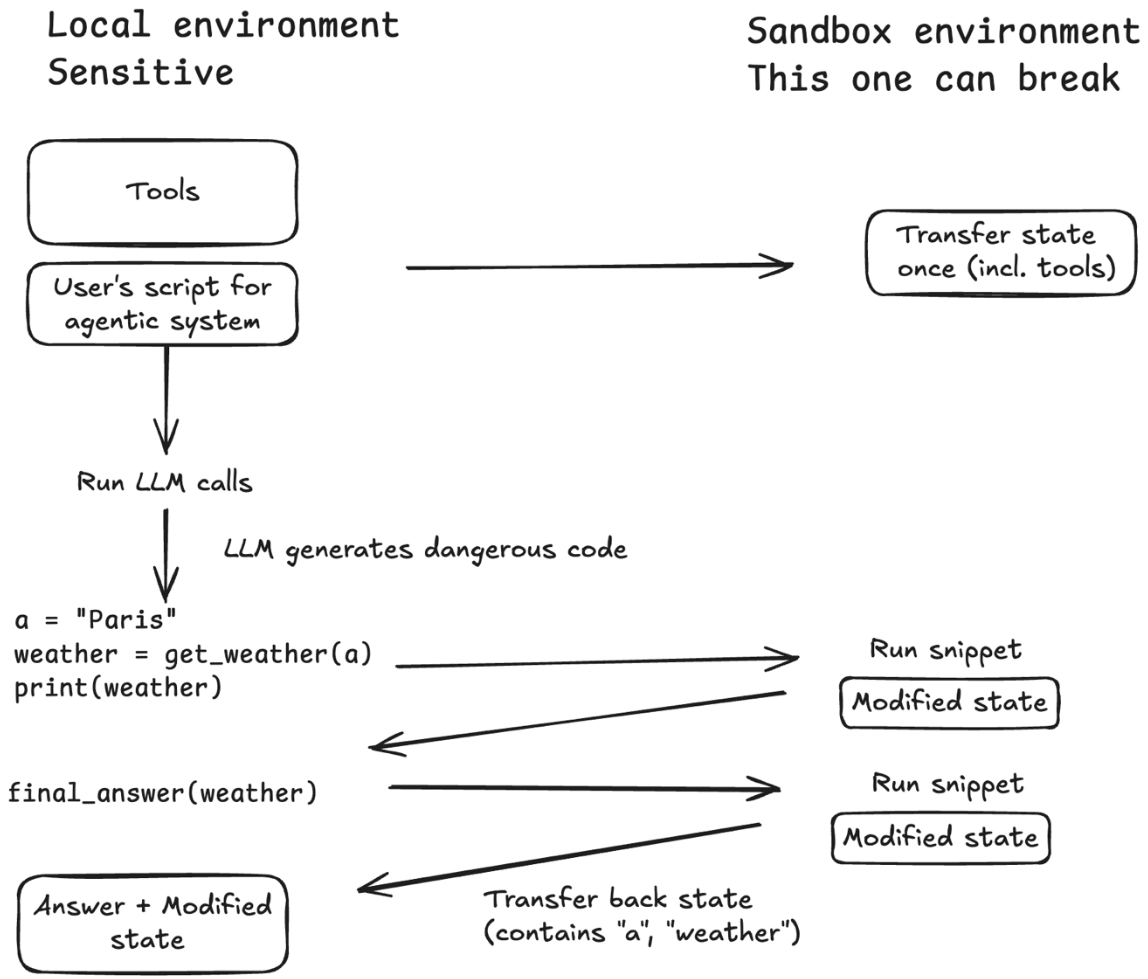
[1] Build, run & deploy Tools for AI Agents: https://github.com/IBM/wxflows
[2] Structured Output: https://docs.langchain.com/oss/javascript/langchain/structured-output (defining and validating data schemas)

The Process of Running a Tool (4)

Given that code generation is no longer a significant challenge for LLMs, wouldn't it be more efficient to provide an execution environment (like SmolAgents) rather than relying on external tools?

- Code generation is still not perfect. Even with the very best models at programming like Claude, it is hard to automatically go to nail the exact code every time.
- Tools can take less time rather than having to write a new program for everything.
- It is much easier to make individual functions secure.

Sandbox Approaches for Code Execution



https://huggingface.co/docs/smolagents/en/tutorials/secure_code_execution

Sandboxing code execution in smolagents

- **Running individual code snippets in a sandbox**:
  - Only executes the agent-generated Python code snippets in a sandbox while keeping the rest of the agentic system in your local environment
  - It still requires passing state data between your environment and the sandbox.
- **Running the entire agentic system in a sandbox**
  - Running the entire agentic system, including the agent, model, and tools, within a sandbox environment.
  - This provides better isolation but requires more manual setup and may require passing sensitive credentials (like API keys) to the sandbox environment.

LangGraph: A Brief Introduction

https://www.youtube.com/watch?v=9dXp5q3OFdQ&list=PLfaIDFEXuae16n2TWUkKq5PgJ0w6Pkwtg&index=2

Example of LangGraph with Self-defined Tool (Code Execution Ability)

```python
# Handle tools execution
def execute_tools(messages, tool_map) -> List[RichToolMessage]:
    tool_messages = []
    for tool_call in messages[-1].tool_calls:
        tool = tool_map[tool_call["name"]]
        if tool_call["name"] == CodeInterpreterFunctionTool.tool_name:
            output = tool.invoke(tool_call["args"])
            message = CodeInterpreterFunctionTool.format_to_tool_message(
                tool_call["id"],
                output,
            )
            tool_messages.append(message)
        else:
            content = tool.invoke(tool_call["args"])
            tool_messages.append(RichToolMessage(content, tool_call_id=tool_call["id"]))
    return tool_messages
```

https://e2b.dev/blog/langgraph-with-code-interpreter-guide-with-code

Deep into Tool Execution with Invoke() in LangGraph

- The .invoke() method comes from the parent class BaseTool (which inherits from Runnable). The invoke method performs a specific **sequence of operations** before actual logic runs.

1. Input Validation (Pydantic)

- Every tool in LangChain has an associated **Pydantic Schema**
- The invoke method takes the raw dictionary of arguments provided by the LLM (e.g., {"location": "London"}).
- It validates these arguments against the tool's schema.
- **If validation fails:** It raises a ValidationError immediately (which ToolNode can catch if handle_tool_errors=True).
- **If validation passes:** It converts the raw dict into the validated python objects.

https://github.com/langchain-ai/langgraph/blob/main/libs/prebuilt/langgraph/prebuilt/tool_node.py (line 930)

Deep into Tool Execution with Invoke() in LangGraph

2. Method Routing (Sync vs Async)

- The BaseTool checks if it is running in a sync or async context.
- If you are running the graph with app.invoke(...), it routes to the _run method [1].
- If you are running the graph with app.ainvoke(...), it routes to the _arun method.

3. Running Your Code

- Finally, it executes the actual logic you defined.

[1] https://github.com/langchain-ai/langchain/blob/master/libs/core/langchain_core/tools/base.py

References

- https://huggingface.co/learn/agents-course/unit2/