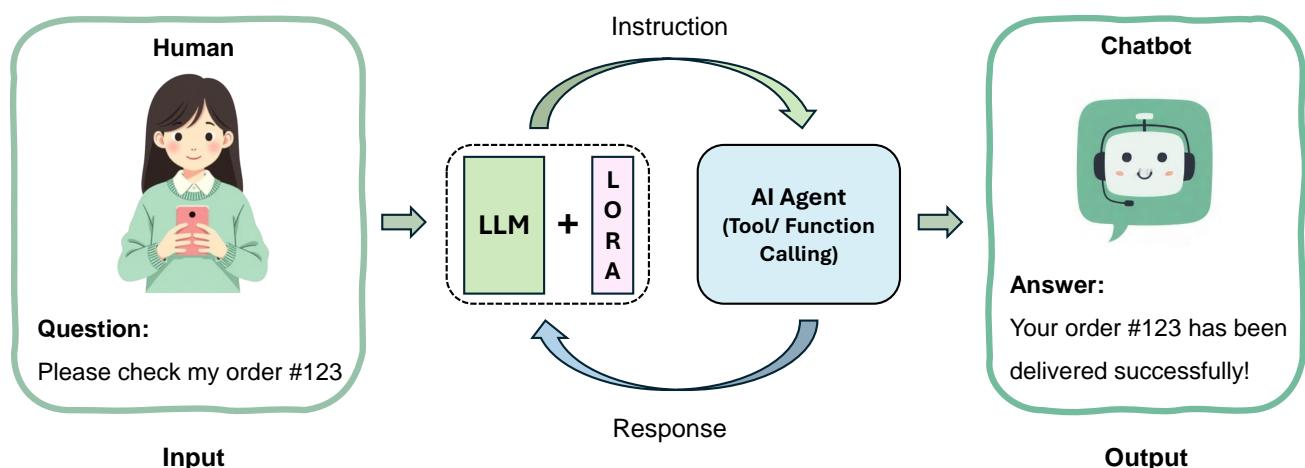


# Tutorial: Solving Computer Vision Tasks with LLMs Tool Calling

Yen-Linh Vu, Anh-Khoi Nguyen, Nguyen-Thuan Duong, Quang-Vinh Dinh

## I. Giới thiệu

Trong quá trình phát triển các mô hình ngôn ngữ lớn (LLMs), chúng ta ngày càng kỳ vọng nhiều hơn, không chỉ ở khả năng hiểu và sinh ngôn ngữ, mà còn ở việc hỗ trợ giải quyết các tác vụ thực tế. Tuy vậy, sẽ có những lúc mô hình gặp phải những giới hạn: nó không thể tự mình truy cập dữ liệu thời gian thực, thực hiện tính toán chính xác, hay tương tác với hệ thống bên ngoài chỉ bằng những gì đã học. Khi đó, Tool Calling đóng vai trò như một cầu nối quan trọng để mở rộng năng lực của mô hình, giúp mô hình thật sự hành động và phản hồi dựa trên thực tại.



Hình 1: Minh họa bài toán QA sử dụng chatbot và AI Agent.

Chúng ta có thể thấy tổng quan cách hoạt động của hệ thống Tool Calling qua Hình 1. Khi người dùng gửi một câu hỏi, mô hình ngôn ngữ (LLM) sẽ không trả lời ngay lập tức, mà trước tiên phân tích yêu cầu của người dùng xong mới quyết định có cần gọi đến một công cụ bên ngoài hay không (chẳng hạn như API kiểm tra đơn hàng, dịch vụ thời tiết, hay công cụ tìm kiếm), nhận kết quả từ công cụ đó, rồi mới sinh ra phản hồi cuối cùng dựa trên dữ liệu đã cập nhật. Hệ thống hoạt động theo chu trình: Suy nghĩ → Gọi công cụ → Quan sát kết quả → Trả lời.

Trong bài viết này, chúng ta sẽ tìm hiểu cách giúp mô hình ngôn ngữ lớn (chỉ xử lý được văn bản) có khả năng tương tác với các công cụ bên ngoài có sẵn nhằm giải quyết một số tác vụ về thị giác máy tính. Theo đó, chương trình sau khi xây dựng có Input/Output như sau:

- **Input:** Một văn bản mô tả tác vụ về thị giác máy tính mà chương trình có hỗ trợ.
- **Output:** Câu trả lời cho tác vụ thị giác tương ứng.

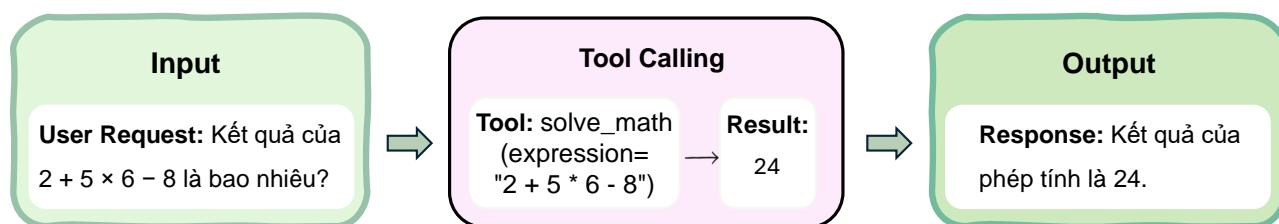
Trước khi đi vào nội dung chính của chương trình, chúng ta sẽ cùng tìm hiểu từ khái niệm tool calling cơ bản cho đến kỹ thuật tinh chỉnh mô hình nhằm cải thiện hiệu suất trong việc sử dụng tool.

**Bài viết sẽ có bối cảnh như sau:**

- **Phần I:** Giới thiệu nội dung bài viết.
- **Phần II:** Bàn luận về Tools/Functions Calling trong LLMs.
- **Phần III:** Cải thiện khả năng Tool Calling của LLMs thông qua Instruction Fine-tuning.
- **Phần IV:** Thực hiện các Computer Vision task thông qua Tool Calling.
- **Phần V:** Câu hỏi trắc nghiệm.
- **Phần VI:** Tài liệu tham khảo.
- **Phần VII:** Phụ lục.

## II. Function/Tool Calling trong LLMs

**Tool Calling là gì?** Tool Calling là khả năng để một mô hình ngôn ngữ lớn (LLM) gọi và thực thi một công cụ hoặc hàm bên ngoài trong quá trình phản hồi người dùng. Điểm mấu chốt không nằm ở việc mô hình mô phỏng câu trả lời, mà ở việc mô hình chủ động đề xuất hành động gọi công cụ, dựa trên khả năng suy luận và kiến thức hiện có.



Hình 2: Minh họa quá trình Tool Calling mô hình sử dụng công cụ bên ngoài để giải bài toán.

Khác với việc đoán mò đáp án, mô hình có thể đưa ra suy nghĩ như: “Tôi không thể tính phép toán này, nhưng tôi biết công cụ nào có thể.” Chẳng hạn, thay vì trả lời ngay “ $2 + 5 \times 6 - 8$  có kết quả là bao nhiêu”, mô hình sẽ tạo ra một lệnh gọi công cụ tính toán, ví dụ như sau:

```

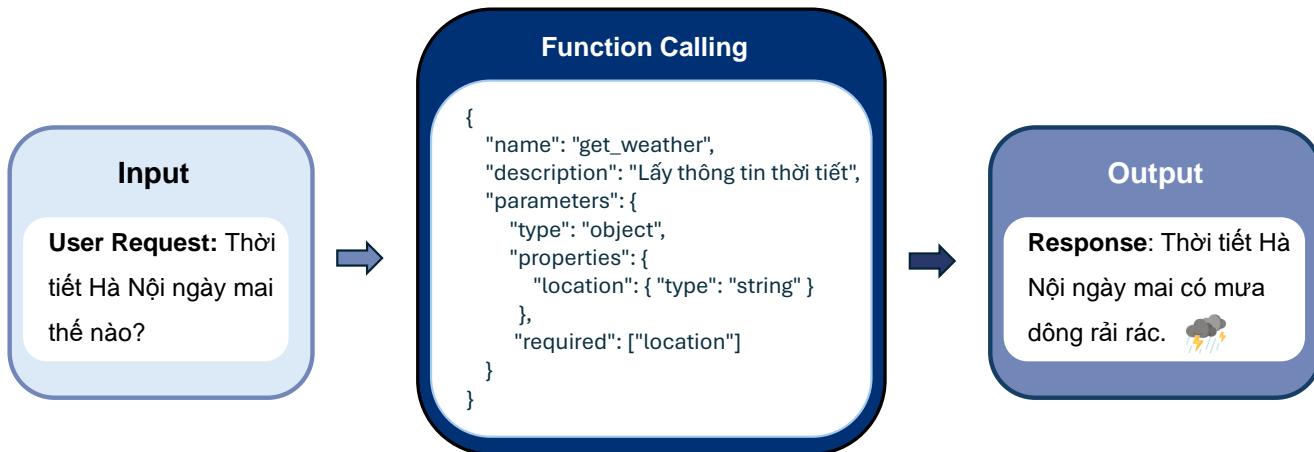
1 # Gọi công cụ tính toán
2 Tool: solve_math(expression="2 + 5 * 6 - 8")
3 # Kết quả: 24
  
```

Sau khi công cụ thực thi, kết quả (24) sẽ được trả về và mô hình có thể phản hồi là “Kết quả của phép tính là 24.” Cách tiếp cận này giúp LLM trở nên giống như một trợ lý thực sự: biết cách khai thác công cụ để tìm câu trả lời.

**Vậy Function Calling là gì và có gì khác biệt?** Nói một cách đơn giản, Tool Calling là khái niệm chung để chỉ khả năng của mô hình khi tương tác với các công cụ bên ngoài. Trong khi đó, Function Calling là một cách cụ thể để thực hiện điều đó – đặc biệt phổ biến trong hệ sinh thái của OpenAI. Với Function Calling, các hàm mà mô hình có thể gọi sẽ được mô tả theo định dạng JSON schema. Nhờ vậy, mô hình có thể “nhìn thấy” được tên hàm, các tham số cần truyền vào, và kiểu dữ liệu tương ứng, giúp nó gọi đúng hàm với đúng thông tin. Ví dụ như trong hình:

```

1 {
2   "name": "get_weather",
3   "description": "Lấy thông tin thời tiết",
4   "parameters": {
5     "type": "object",
6     "properties": {
7       "location": { "type": "string" }
8     },
9     "required": ["location"]
10   }
11 }
  
```



Hình 3: Minh họa quá trình sử dụng Function Calling.

Không chỉ dừng lại ở việc hiểu hàm, mô hình còn có thể tự suy luận ra hành động cần làm: khi nào nên gọi hàm, cần truyền vào những tham số nào, và xử lý kết quả trả về sao. Toàn bộ quá trình gồm ba bước chính:

- **Thought (Suy nghĩ):** phân tích yêu cầu đầu vào.
- **Act (Hành động):** sinh lời gọi hàm.
- **Observe (Quan sát):** nhận và xử lý kết quả trả về.

Function Calling tạo ra một “ngôn ngữ hành động” có cấu trúc, từ đó giúp LLM hành xử như một thành phần biết tương tác logic trong hệ thống lớn hơn. Tuy nhiên có một điều quan trọng ta cần lưu ý, mô hình không trực tiếp thi hàm. Nó chỉ sinh lệnh gọi hàm (function call) dưới dạng JSON. Việc thực thi hàm và trả kết quả thuộc về hệ thống bên ngoài, sau đó kết quả này mới được mô hình sử dụng để tạo ra phản hồi hoàn chỉnh.

**Khi nào nên dùng Tool hoặc Function Calling?** Tool/Function Calling thường được sử dụng khi ta muốn LLM giải quyết những tác vụ mà bản thân nó không thể hoặc rất khó tự giải quyết chỉ thông qua ngôn ngữ tự nhiên. Những tình huống này phổ biến trong các tác vụ thực tế như:

- Truy vấn cơ sở dữ liệu hoặc hệ thống quản lý thông tin nội bộ.
- Tìm kiếm thông tin mới nhất (giá cổ phiếu, thời tiết, tin tức).
- Thực hiện các phép toán phức tạp hoặc thao tác logic.
- Tương tác với các API hoặc hệ thống phần mềm khác.
- Triển khai công việc chuyên môn: phân tích y học, tìm kiếm học thuật, truy xuất bản đồ.

Với các tình huống này, việc tích hợp công cụ bên ngoài giúp mô hình mở rộng năng lực xử lý vượt ra khỏi giới hạn chỉ dựa vào văn bản huấn luyện.

**Các loại công cụ trong Tool Calling?** Dựa trên cách tích hợp và ngữ cảnh sử dụng, các công cụ trong Tool Calling có thể chia thành ba loại chính:

- **Built-in Tool:** là các công cụ được tích hợp sẵn trong mô hình như Wolfram Alpha, Python, Brave Search trong Llama 3.
- **JSON-based Tool:** thường gặp trong hệ sinh thái OpenAI, nơi function được định nghĩa bằng schema JSON rõ ràng.
- **Custom Tool:** do người dùng định nghĩa và kiểm soát, chẳng hạn như các hàm Python tùy biến hoặc endpoint API riêng.

Ta có thể thấy ví dụ về việc sử dụng công cụ toán học Wolfram để giải một biểu thức:

```

1 # Lời gọi công cụ Wolfram
2 Tool: wolfram_alpha.calculate("2*y^2 + x^2")
3 # Trả về biểu thức được rút gọn hoặc hình ảnh biểu diễn

```

Mỗi loại công cụ phù hợp với một mục tiêu và môi trường triển khai khác nhau, việc lựa chọn phù hợp sẽ giúp hệ thống hoạt động ổn định và hiệu quả.

**Tool Calling có phải là AI Agent?** Không hoàn toàn. Khái niệm *agent* là một trong những nền tảng quan trọng nhất trong lĩnh vực Trí tuệ nhân tạo (AI). Trong giáo trình kinh điển *Artificial Intelligence: A Modern Approach* của Stuart Russell và Peter Norvig đã định nghĩa như sau:

*“An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. The agent function maps percept histories to actions.”*

— Stuart Russell & Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Chapter 2, p. 34.

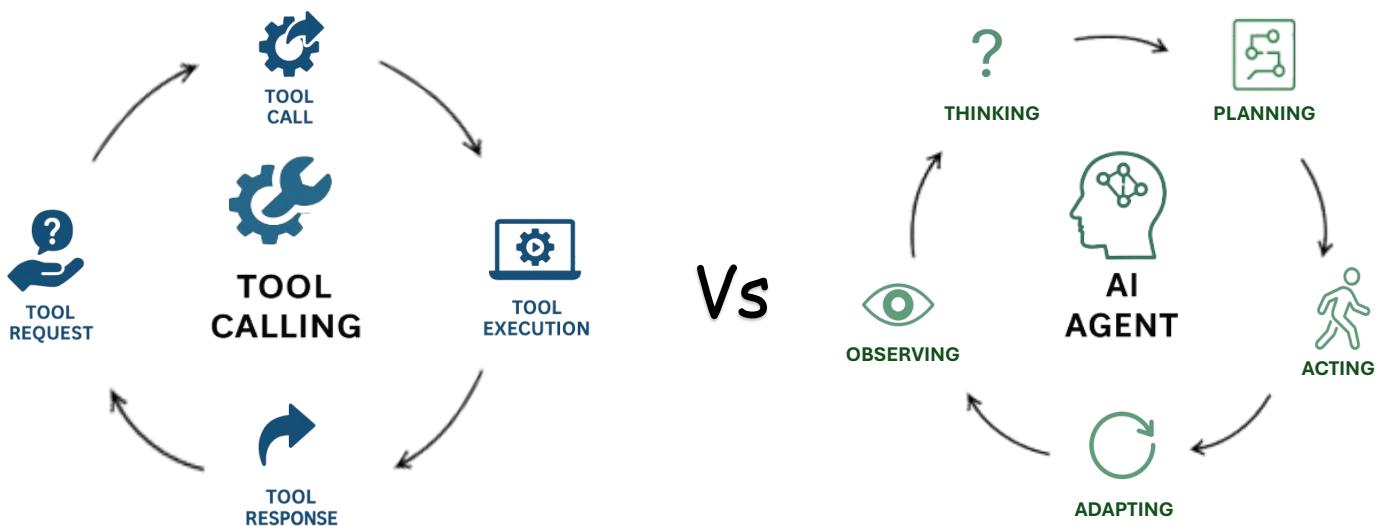
(**Tạm dịch:**) “Một agent là bất kỳ thực thể nào có thể được xem là có khả năng cảm nhận môi trường xung quanh thông qua các cảm biến, và đưa ra hành động tác động lại lên môi trường thông qua các bộ phận điều khiển. Hàm agent ánh xạ lịch sử nhận thức thành các hành động.”

Theo định nghĩa trên, một **AI Agent** là hệ thống có khả năng cảm nhận môi trường, lập kế hoạch và thực hiện hành động dựa trên thông tin nhận được. Chu trình hành vi của một agent không dừng lại ở phản ứng tức thì, mà bao gồm một vòng lặp liên tục:

**Observing → Thinking/Planning → Acting → Adapting → Observing**

Điều này có nghĩa rằng một agent cần có khả năng:

- **Quan sát (observing):** thu nhận thông tin từ môi trường.
- **Tư duy / Lập kế hoạch (thinking/planning):** phân tích bối cảnh và xây dựng chiến lược.
- **Hành động (acting):** thực thi quyết định.
- **Điều chỉnh (adapting):** điều chỉnh hành vi dựa trên kết quả.



Hình 4: So sánh Tool Calling và AI Agent.

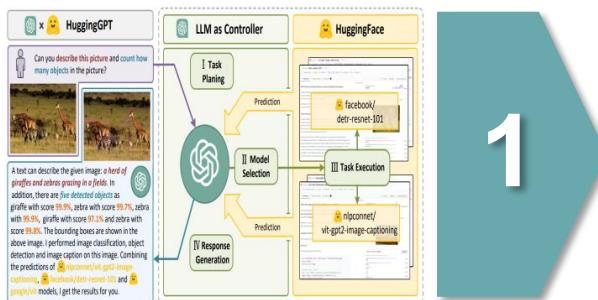
Như vậy, ta có thể hiểu là một mô hình ngôn ngữ có thể biết cách gọi công cụ (tool calling), nhưng nếu thiếu các bước như lập kế hoạch, quan sát phản hồi từ môi trường, hoặc điều chỉnh hành vi, thì nó vẫn chỉ là một hệ thống phản hồi theo lượt (*turn-based reactive system*), chưa đạt đến mức của một *AI Agent* thực thụ. Ngược lại, một agent có thể hoạt động mà không cần gọi công cụ, miễn là nó có chu trình hành vi đầy đủ như trên. Do đó:

- **Tool Calling** là một *kỹ năng* (gọi đúng công cụ vào đúng thời điểm.)
- **AI Agent** là một *hệ thống hành vi tự chủ* – có khả năng quan sát, suy luận, hành động và học hỏi từ môi trường.

Một khi phân biệt rõ hai khái niệm này ta có thể tránh sự hiểu nhầm giữa “một mô hình biết sử dụng công cụ” và “một hệ thống có khả năng tư duy độc lập, hành động có chiến lược và thích nghi với thay đổi.”

Sau khi đã tìm hiểu qua các khái niệm Tool Calling, Function Calling và AI Agent cũng như sự khác biệt giữa chúng, ta có thể cùng nhìn lại hành trình phát triển của công nghệ này qua các cột mốc nổi bật.

Hình 5 dưới đây sẽ giúp chúng ta hình dung rõ hơn về quá trình phát triển của Tool/Function Calling trong các mô hình ngôn ngữ lớn (LLMs). Từ những bước khởi đầu với HuggingGPT, đến sự xuất hiện của Function Calling API của OpenAI, và gần đây là các chuẩn giao tiếp đa mô hình như Model Context Protocol của Anthropic. Mỗi bước tiến đều đánh dấu một thay đổi quan trọng trong cách LLMs được thiết kế để không chỉ phản hồi bằng ngôn ngữ, mà còn biết hành động thông qua công cụ, xử lý logic và tương tác với môi trường như một agent thực sự.

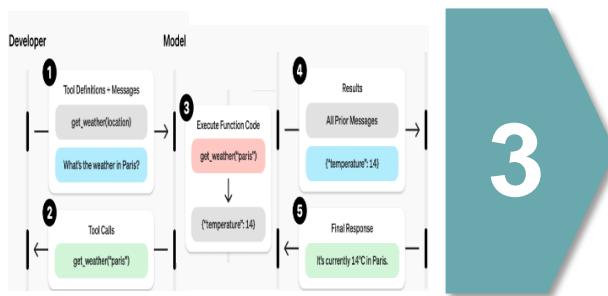


## HuggingGPT 03/2023

LLM lần đầu đóng vai "bộ não" điều phối: phân tích yêu cầu, lập kế hoạch, và kích hoạt các mô hình AI chuyên sâu (text, image, speech...) từ Hugging Face. Đây là nền tảng cho việc tích hợp công cụ (tool use) vào LLM.

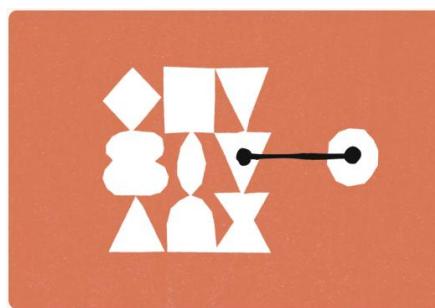
## GPT4Tools 05/2023 LLMs as Tool Makers

Phát triển từ ý tưởng trên, GPT-4 không chỉ gọi công cụ (tool calling) mà còn tự động tạo lệnh (function generation) dựa trên mô tả. Bước nhảy vọt này giúp LLM chuyển từ thao tác thủ công sang tự học cách ứng dụng công cụ.



## Function Calling in AI Agents (Mistral 7B) 10/2023

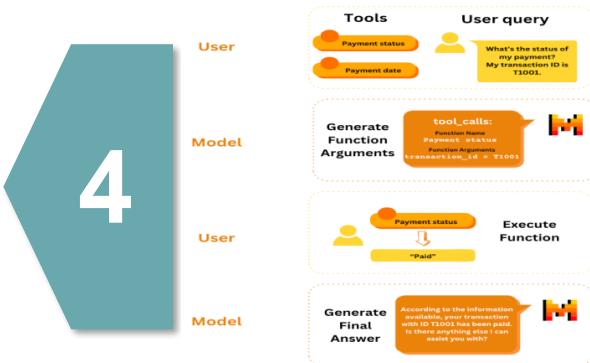
Tiến thêm một bước, Function Calling trở thành phần cốt lõi trong vòng lặp agent (agent loop): ví dụ Mistral 7B thực hiện chuỗi suy nghĩ → chọn công cụ → thực thi → cải thiện (*planning → acting → observing → adapting*). Quy trình cho phép xử lý tác vụ đa bước (multi-step tasks) trong môi trường phản hồi.



5

## OpenAI Function Calling API 06/2023

Ké thừa xu hướng, OpenAI ra mắt API cho phép LLM sinh lệnh gọi hàm (function calls) dạng JSON. Công cụ này kết nối mô hình với hệ thống backend, cơ sở dữ liệu... một cách chính xác và an toàn, mở đường cho tích hợp đa nền tảng.



## Model Context Protocol (MCP) – Anthropic 11/2024

Đến cuối năm 2024, MCP cho phép nhiều LLM chia sẻ ngữ cảnh (context sharing) và phối hợp như một hệ đa agent (multi-agent). Đây là bước tiến lớn cho các hệ thống AI phức tạp và linh hoạt hơn.

Hình 5: Khảo sát các mốc phát triển chính của Tool/Function Calling trong LLMs (2023–2024).

## II.1. Tool Calling Built-in với Llama3

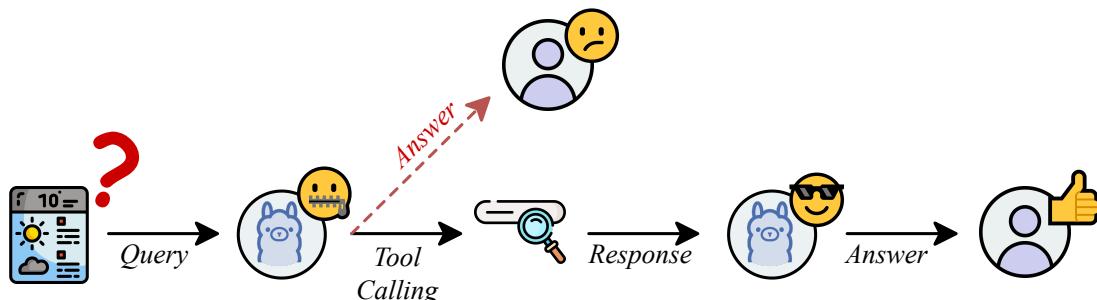
Tool Calling là tính năng nổi bật được tích hợp sẵn trong mô hình ngôn ngữ Llama3, giúp mô hình có thể chủ động sử dụng các công cụ hỗ trợ để hoàn thành những tác vụ cụ thể mà bản thân mô hình gặp khó khăn. Có ba hình thức Tool Calling chính được hỗ trợ:

- **Built-in Tools** (công cụ tích hợp sẵn như Brave Search và Wolfram Alpha).
- **JSON-based Tool Calling** (gọi công cụ dựa trên cấu trúc JSON).
- **User-defined Custom Tool Calling** (gọi công cụ do người dùng tự định nghĩa).

Mục này sẽ minh họa về **built-in tools**. Cụ thể, các phiên bản mô hình Llama3.1-8B/70B/405B có khả năng sử dụng trực tiếp hai công cụ tích hợp dựa trên Python như sau:

- **Brave Search**: Giúp mô hình thực hiện các tìm kiếm trên web.
- **Wolfram Alpha**: Cho phép mô hình thực hiện các phép tính toán phức tạp.

Trong đó, đầu vào của mô hình sẽ một câu hỏi để truy vấn thông tin (như là "Thời tiết ở Hồ Chí Minh hôm nay như thế nào?") và mô hình sẽ tạo ra câu trả lời bằng cách tổng hợp các thông tin được cung cấp từ công cụ (Brave Search). Các hành động được minh họa như hình bên dưới.



Hình 6: Minh họa quá trình truy vấn thông tin về thời tiết. Trường hợp mô hình không sử dụng công cụ được tô màu đỏ và thường trả về kết quả không như kỳ vọng của người dùng.

Để kích hoạt việc gọi các công cụ tích hợp này, chúng ta sử dụng một đoạn mã khai báo đặc biệt gọi là *System Prompt*, chẳng hạn như:

```

1<|begin_of_text|><|start_header_id|>system<|end_header_id|>
2
3Environment: ipython
4Tools: brave_search, wolfram_alpha
5Cutting Knowledge Date: December 2023
6Today Date: 23 July 2024
7
8You are a helpful assistant<|eot_id|>
9<|start_header_id|>user<|end_header_id|>
10
11What is the current weather in Menlo Park, California?<|eot_id|>
12<|start_header_id|>assistant<|end_header_id|>
  
```

Một số điểm cần lưu ý về cách thức mô hình thực hiện Tool Calling:

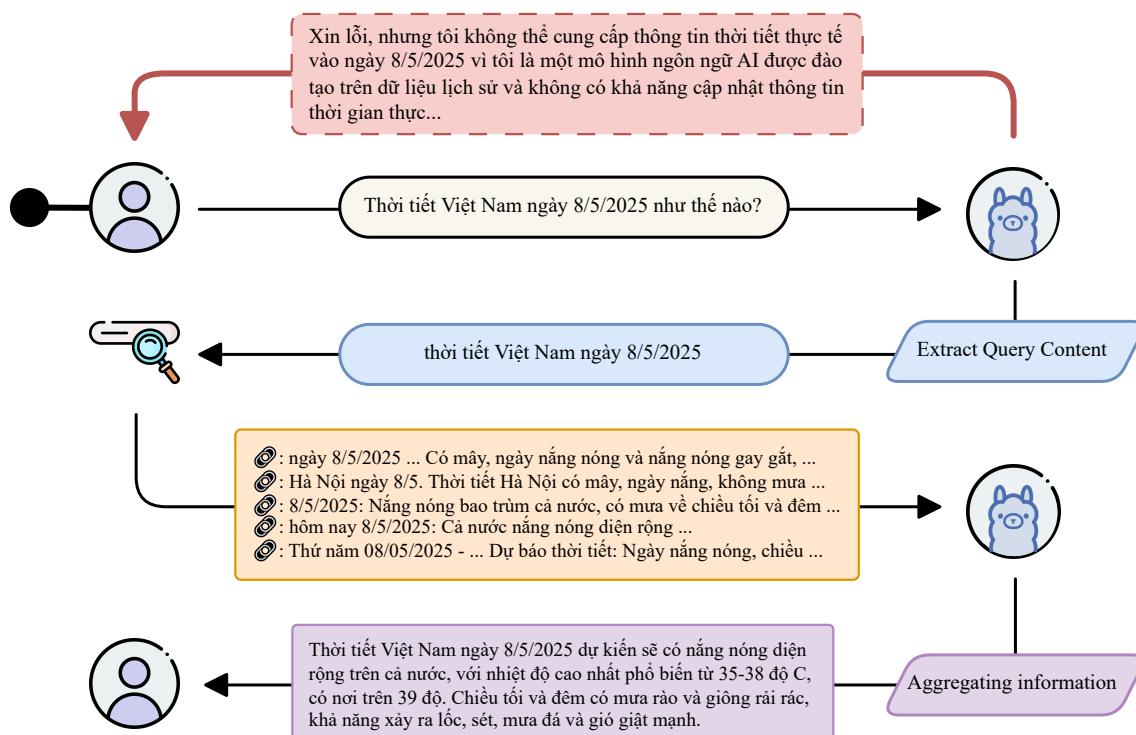
- Việc khai báo Environment: ipython tự động kích hoạt tính năng Code Interpreter mà không cần ghi rõ trong mục Tools.
- Khi trả lời, nội dung do mô hình tạo ra sẽ bắt đầu với một tag đặc biệt <|python\_tag|>.
- Trong chế độ này, mô hình có thể sử dụng tag <|eom\_id|> để biểu thị quá trình suy luận nhiều bước nhưng chưa kết thúc, khác với <|eot\_id|> là dấu hiệu cho thấy lượt trả lời đã hoàn tất.

Các công cụ tích hợp được gọi bằng cú pháp Python cụ thể như sau:

```

1 # Brave Search
2 <|python_tag|>
3 brave_search.call(query="...") 
4 <|eom_id|>
5
6 # Wolfram Alpha
7 <|python_tag|>
8 wolfram_alpha.call(query="...") 
9 <|eom_id|>
```

Để đơn giản hóa, việc triển khai minh họa này sử dụng mô hình Llama3.1-8B thông qua API của Groq và công cụ tìm kiếm Brave Search được thay thế bằng DuckDuckGo Search, trong khi giữ nguyên công cụ tính toán phức tạp là Wolfram Alpha.



Hình 7: Minh họa quy trình xử lý và phản hồi từ Llama3.1 với công cụ tìm kiếm. Với đường màu đỏ là phản hồi khi không sử dụng công cụ và ngược lại cho đường màu đen.

### II.1.1. Thiết lập môi trường, mô hình và công cụ

Trước khi bắt đầu, cần cài đặt và nhập (import) các thư viện cần thiết như sau:

```
1 !pip install duckduckgo-search groq wolframalpha
```

```
1 import os
2 import re
3 import json
4 import asyncio
5 import aiohttp
6 import requests
7 import nest_asyncio
8 import urllib.parse
9 from groq import Groq
10 from google.colab import userdata
11 from duckduckgo_search import DDGS
12
13 nest_asyncio.apply()
14
15 os.environ["GROQ_API_KEY"] = "your_groq_api_key"
16 APP_ID = "your_wolfram_app_id"
17
18 client = Groq()
```

Trong đó, từng bước lấy **API Key** của Groq gồm:

1. Đăng nhập (hoặc đăng ký) tại [đây](#).
2. Tại thanh điều hướng (góc trên bên phải), chọn "API Keys" và đợi chuyển trang, sau đó, nhấn "Create API Key" để tạo API Key.
3. Sao chép "API Key" và thay vào your\_groq\_api\_key trong đoạn mã trên.

Với Wolfram, lấy **App ID** theo các bước sau:

1. Đăng nhập (hoặc đăng ký) tại [đây](#).
2. Chọn "Get an App ID" và điền các trường Name, Description tùy ý và chọn **Simple API** cho mục API, sau đó nhấn "Submit" (nếu không thấy trang như mô tả, từ thanh điều hướng chọn "API ACCESS").
3. Sao chép đoạn ký tự bên dưới "App ID" (ví dụ: JH5XYZ-ABCDE9XU5W) và thay thế your\_wolfram\_app\_id trong đoạn mã trên.

### II.1.2. Xây dựng các hàm xử lý

Sau khi thiết lập được mô hình và công cụ, để xử lý việc gọi và nhận kết quả từ API, cần bổ sung thêm một số hàm như sau:

- **model\_response(prompt)**: Gửi prompt đến mô hình ngôn ngữ LLaMA-3.1-8B và trả về phản hồi của mô hình.

- `extract_query(tool_call_text)`: Trích xuất tên công cụ và nội dung truy vấn từ chuỗi đầu vào.
- `format_search_results(results)`: Định dạng danh sách kết quả tìm kiếm thành chuỗi văn bản dễ đọc, bao gồm tiêu đề, mô tả và liên kết.
- `wolfram_short_answer(query)`: Gửi truy vấn đến Wolfram Alpha API và trả về câu trả lời ngắn dưới dạng văn bản, xử lý dưới dạng bất đồng bộ.

```

1 def model_response(prompt):
2     response = client.chat.completions.create(
3         messages=[
4             {
5                 "role": "user",
6                 "content": prompt,
7             }
8         ],
9         # model="llama-3.3-70b-versatile", # for search
10        model="llama-3.1-8b-instant", # for calculation
11    )
12
13    return response
14
15 def extract_query(tool_call_text):
16     brave_match = re.search(r'brave_search.call\((query="(.*?)"\)', tool_call_text)
17     if brave_match:
18         return "brave_search", brave_match.group(1)
19     wolfram_match = re.search(r'wolfram_alpha.call\((query="(.*?)"\)', tool_call_text)
20     if wolfram_match:
21         return "wolfram_alpha", wolfram_match.group(1)
22     return None, None
23
24 def format_search_results(results):
25     lines = []
26     for i, item in enumerate(results, start=1):
27         title = item.get("title", "No title")
28         href = item.get("href", "No link")
29         body = item.get("body", "No description")
30         lines.append(f"{i}. {title}\n  Description: {body}\n  Link: {href}\n")
31     return "\n".join(lines)
32
33 async def wolfram_short_answer(query):
34     encoded_query = urllib.parse.quote(query)
35     url = f"https://api.wolframalpha.com/v1/result?i={encoded_query}&appid={APP_ID}"
36     async with aiohttp.ClientSession() as session:
37         async with session.get(url) as response:
38             if response.status == 200:
39                 return await response.text()
40             else:
41                 return f"[Error {response.status}] {await response.text()}"

```

### II.1.3. Xây dựng LLM Tool Calling

Nhìn chung, quy trình Tool Calling gồm 5 bước rõ ràng:

1. Người dùng cung cấp yêu cầu (prompt) cùng với System Prompt hoặc một đoạn prompt mặc định.
2. Mô hình quyết định công cụ cần gọi.
3. Công cụ thực thi và trả kết quả về (ví dụ Wolfram Alpha thực hiện phép tính).
4. Kết quả công cụ được gửi trở lại cho mô hình.
5. Mô hình tổng hợp và gửi phản hồi cuối cùng tới người dùng.

Các bước trên ứng với từng bước trong đoạn mã bên dưới:

```

1  async def llama_with_tool(query):
2      PROMPT = f"""
3 {<|begin_of_text|><|start_header_id|>system<|end_header_id|>
4
5 Environment: ipython
6 Tools: brave_search, wolfram_alpha
7 Cutting Knowledge Date: December 2023
8 Today Date: 23 July 2024
9
10 You are a helpful assistant specialized in performing web and computation searches. You have
    access to two tools:
11     1. **brave_search** for general web searches
12     2. **wolfram_alpha** for computation and factual queries
13 Always try to use these tools for informational and computational query questions.<|eot_id|><
    |start_header_id|>user<|end_header_id|>
14 {query}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
15 """
16
17     response = model_response(PROMPT)
18     tool_call = response.choices[0].message.content.strip()
19
20     print("== Model Tool Call Output ==\n")
21     print(tool_call)
22
23     tool_name, query_content = extract_query(tool_call)
24     print("\n== Extracted Query Content ==\n")
25     print(query_content)
26
27     if tool_name == "brave_search":
28         tool_response = DDGS().text(query_content, max_results=5)
29         formatted_results = format_search_results(tool_response)
30     elif tool_name == "wolfram_alpha":
31         res = await wolfram_short_answer(query_content)
32         formatted_results = "The exact value of the query is " + res
33     else:
34         formatted_results = "No tool could be identified."
35
36     print("\n== Tool Responses ==\n")
37     print(formatted_results)
38
39     follow_up_prompt = f"Tool responses:\n{formatted_results}\nBased on these responses,
        provide a concise result for this query.: {query}"

```

```

40 follow_up_response = model_response(follow_up_prompt)
41
42 print("\n==== Final Answer ===\n")
43 return follow_up_response.choices[0].message.content

```

Nhờ tích hợp Tool Calling, Llama3.1-8B có thể giải quyết hiệu quả hơn những vấn đề phức tạp, giúp người dùng nhận được các câu trả lời chính xác và hữu ích hơn. Ví dụ như khi mô hình dùng Brave Search (DuckDuckGo Search) như Hình 7 sau khi chạy đoạn mã bên dưới:

```

1 search_question = "Thời tiết Việt Nam ngày 8/5/2025 như thế nào?"
2 print(await llama_with_tool(query=search_question))

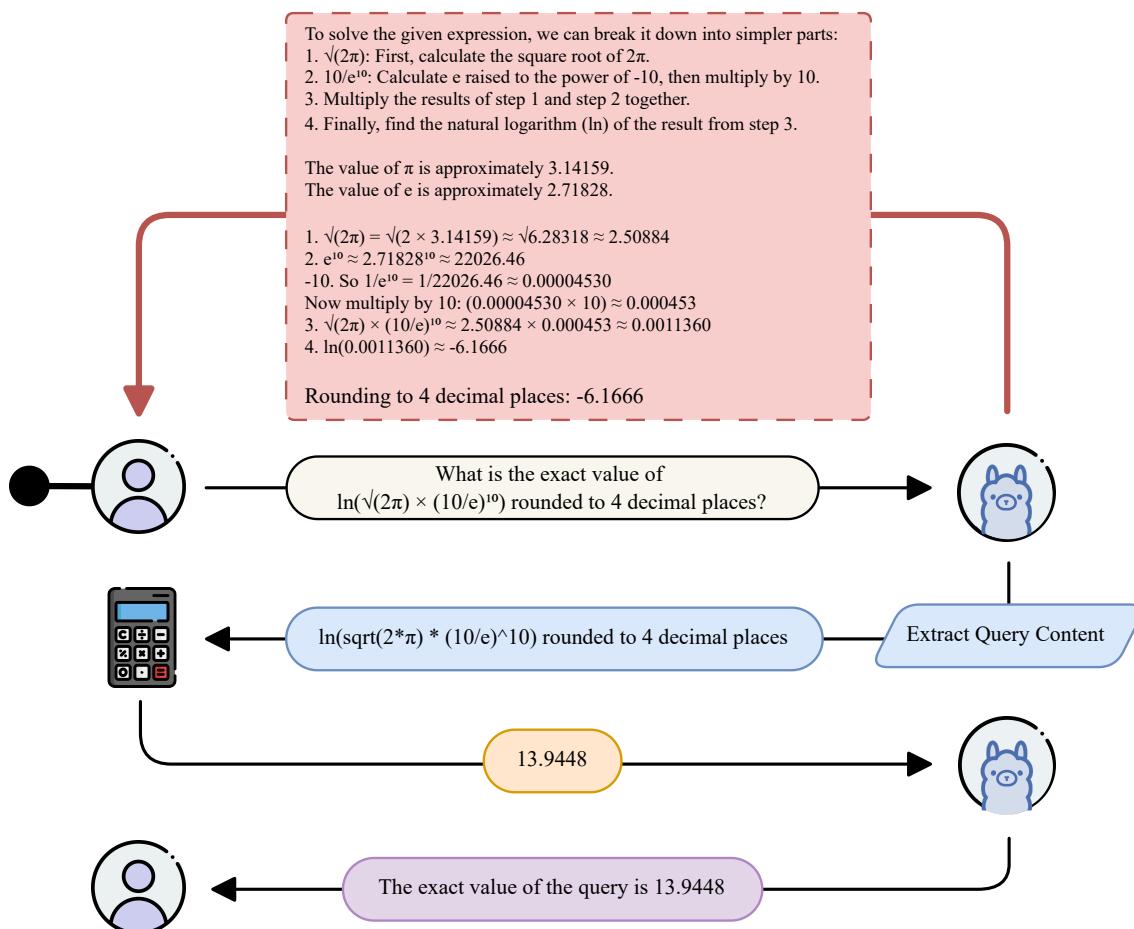
```

Hoặc dùng công cụ Wolfram khi chạy đoạn mã sau:

```

1 math_question = "What is the exact value of ln(sqrt(2pi) * (10/e)^10) rounded to 4 decimal
                  places?"
2 print(await llama_with_tool(query=math_question))

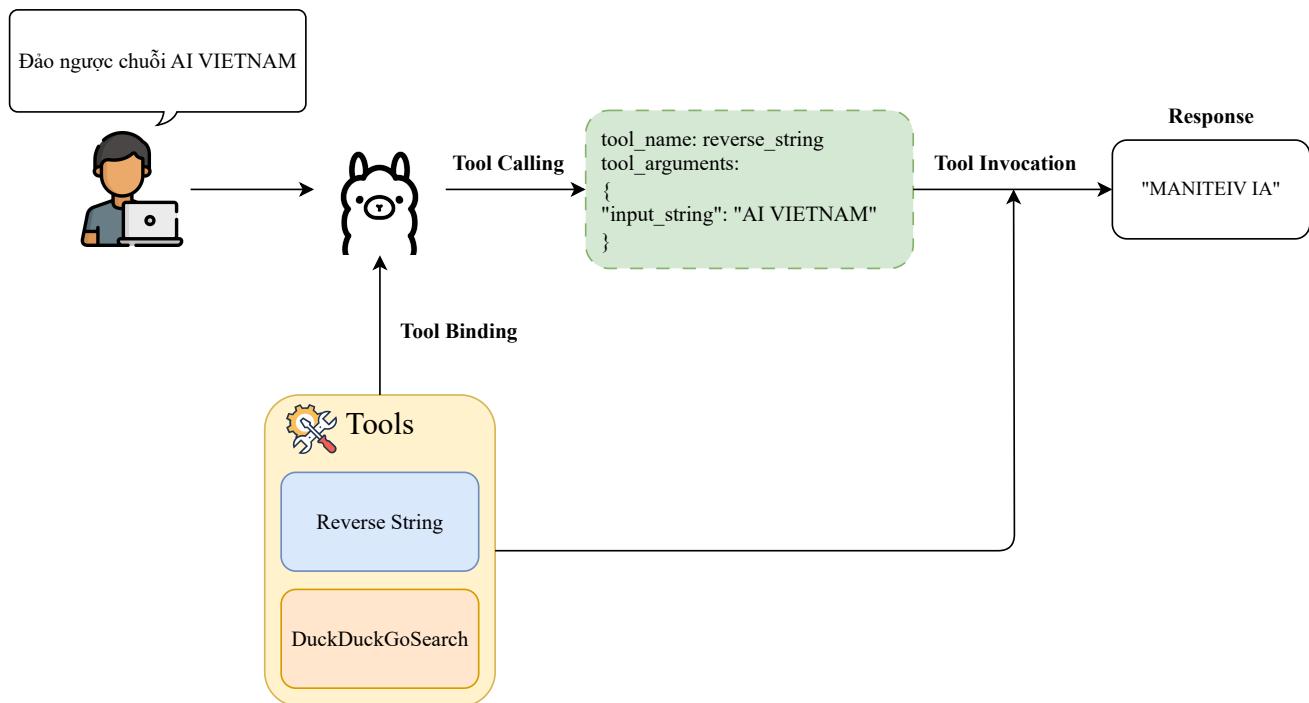
```



Hình 8: Quy trình xử lý và phản hồi từ Llama3.1 với công cụ tính toán phức tạp. Trong đó, phản hồi không sử dụng công cụ cho kết quả sai và ngược lại khi có dùng công cụ.

## II.2. Tool Calling với thư viện LangChain

Ở thời điểm hiện tại, việc cài đặt các tool cho mô hình lớn được hỗ trợ rất nhiều trên những thư viện chuyên về mô hình lớn như [LangChain](#), [crewAI](#), [LlamaIndex](#)... Ở phần này, chúng ta sẽ sử dụng thư viện LangChain và cùng tìm hiểu xem việc trang bị tool được thực hiện như thế nào với thư viện này.



Hình 9: Mô tả LLMs Tool Calling với thư viện LangChain.

### II.2.1. Tải và import các thư viện cần thiết

Chúng ta cần cài đặt thư viện cũng như các module liên quan đến thư viện LangChain để có chạy được đoạn code này. Sau đó, các bạn import một số module như sau:

```

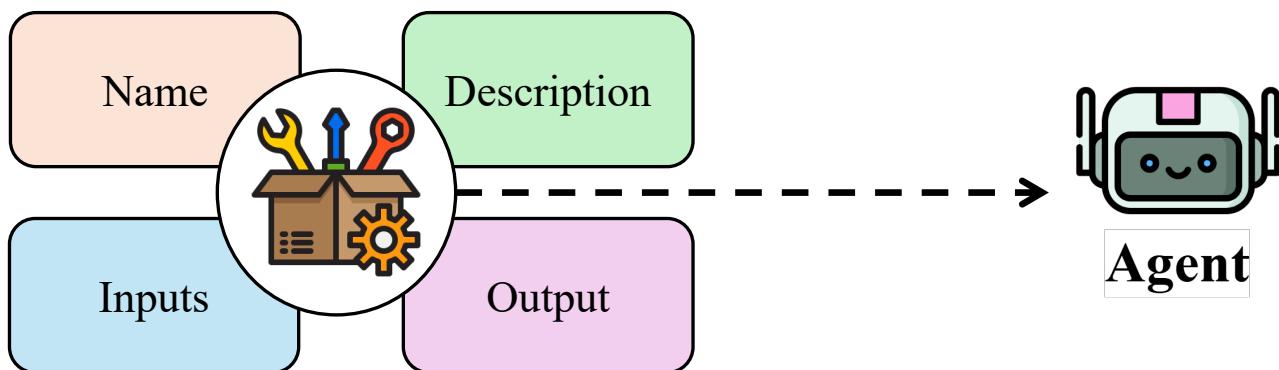
1 import torch
2
3 from langchain.chat_models import init_chat_model
4 from langchain_core.tools import tool
5 from langgraph.prebuilt import create_react_agent
6 from langchain_community.utilities import DuckDuckGoSearchAPIWrapper
7 from langchain_community.tools import DuckDuckGoSearchResults
8 from langchain.tools import Tool

```

### II.2.2. Định nghĩa các tool trong LangChain

Đối với LangChain, để định nghĩa các tool, chúng ta sẽ cần định nghĩa hàm Python với đầy đủ thông tin từ input đầu vào đến output đầu ra đúng với công năng mà ta muốn cho công cụ cần cài. Theo đó, với một tool, ta cần định nghĩa các thông tin như sau:

1. **@tool decorator:** Để LangChain có thể hiểu cũng như đưa hàm Python trở thành công cụ cho LLM, ta cần định nghĩa hàm với decorator `@tool`.
2. **Tên hàm:** Tên hàm Python cần phải thể hiện rõ chức năng hoặc công dụng hoặc mục đích của công cụ mà chúng ta muốn tạo.
3. **Các tham số đầu vào đầu ra kèm type hint:** Khi định nghĩa các biến đầu vào cho hàm, chúng ta lưu ý cần phải cung cấp cả thông tin về kiểu dữ liệu tương ứng cho từng biến. Làm như vậy, mô hình sẽ có thể hiểu và đưa thông tin chính xác hơn khi gọi hàm.
4. **Docstring:** Một dòng mô tả ngắn cho chức năng hoặc nội dung chính của hàm. Đây là một thao tác clean code thông dụng nhưng với LangChain đây sẽ là thứ giúp mô hình có thể hiểu hơn về hàm, công cụ mà ta muốn nó sử dụng.



Hình 10: Minh họa một số thông tin mà Agent cần để hiểu ý nghĩa của một tool do ta tự định nghĩa.

```

1 @tool
2 def reverse_string(input_string: str) -> str:
3     """Reverse a string"""
4     reversed_string = input_string[::-1]
5
6     return reversed_string
7
8 @tool
9 def search(query: str, top_k=3) -> str:
10    """Search DuckDuckGo for the given query and return the first result."""
11    wrapper = DuckDuckGoSearchAPIWrapper(region="vn-vi", time="d", max_results=top_k)
12    search = DuckDuckGoSearchResults(api_wrapper=wrapper, source="news")
13    results = search.run(query)
14
15    return results

```

Ở trên, ta định nghĩa hai tool rất đơn giản cho mô hình bao gồm một có chức năng đảo ngược chuỗi và một tool dùng để search một nội dung bất kì trên internet như minh họa ở hình 9.

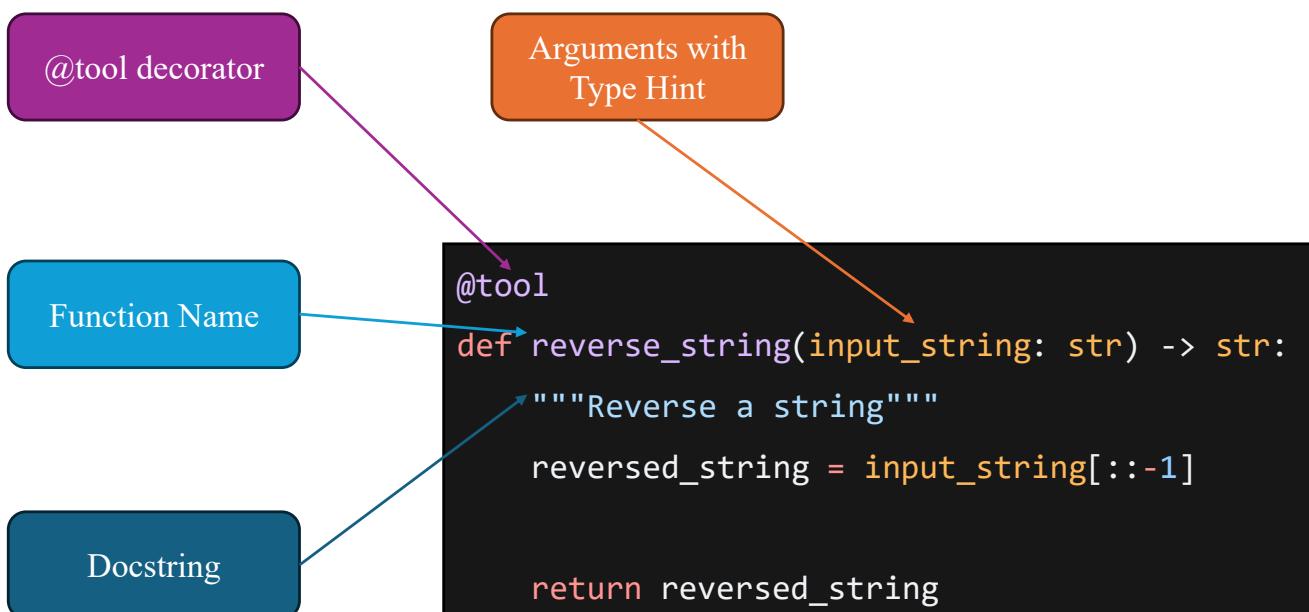
### II.2.3. Khai báo mô hình

Ta khai báo một mô hình lớn để sử dụng. Tại đây, mô hình Llama 3.2 3B được chọn để làm mô hình lớn trong ứng dụng này. Lưu ý rằng, mô hình được khởi tạo từ nguồn là Ollama. Vì vậy, các bạn cần phải cài đặt và triển khai Ollama trước khi chạy đoạn code này.

```

1 model_id = "llama3.2:3b"
2 temperature = 0.5
3 max_output_tokens = 512
4
5 llm = init_chat_model(
6     model=model_id,
7     model_provider="ollama",
8     temperature=temperature,
9     max_tokens=max_output_tokens,
10    format="json"
11)

```



Hình 11: Minh họa các thành phần cần thiết để định nghĩa một tool trong LangChain.

### II.2.4. Khai báo danh sách các tools sử dụng và khởi tạo agent với tools

Cuối cùng, tổng hợp các thông tin trên, chúng ta sẽ tạo một danh sách các tool và đưa vào module tạo một ReAct Agent (một loại Agent) đã được hỗ trợ sẵn từ LangChain để tiến hành việc lắp đặt các công cụ đã định nghĩa vào mô hình lớn. Thực hiện như sau:

```

1 tools = [reverse_string, search]
2 agent_executor = create_react_agent(llm, tools)

```

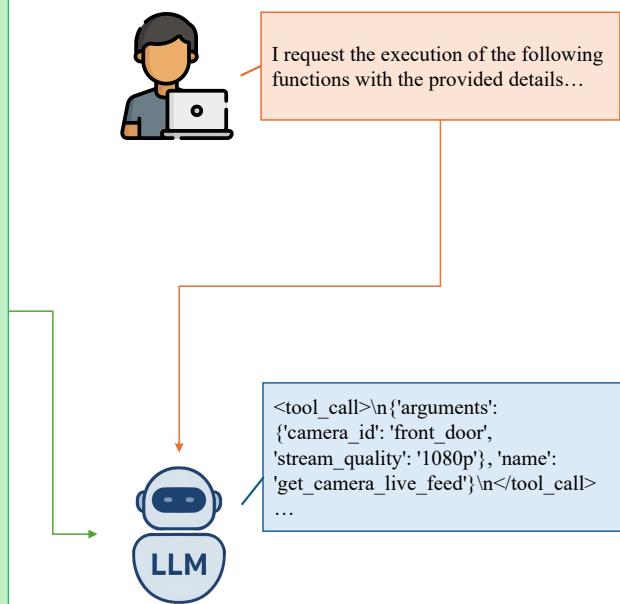
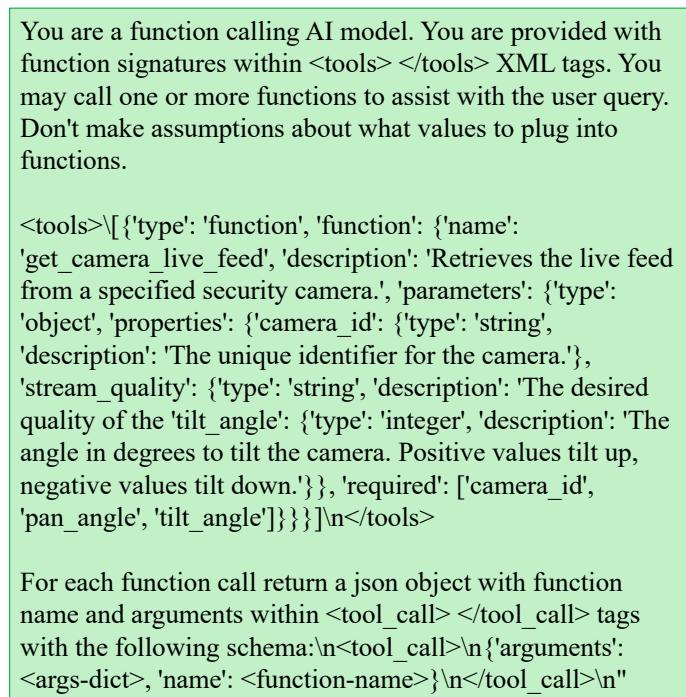
Khi muốn thực hiện inference mô hình này, các bạn chỉ cần thực thi đoạn code:

```
1 agent_executor.invoke({"input": "your input"})
```

là đã có thể sử dụng được mô hình. Việc sử dụng tool sẽ được mô hình tự động hiểu và sử dụng nếu cần để trả lời câu hỏi.

### III. Cải thiện khả năng gọi tool của mô hình với Instruction Fine-tuning

Một số mô hình lớn, đặc biệt là các mô hình có số lượng tham số thấp, việc hiểu và đưa ra các lời gọi hàm còn hạn chế do không được quá chăm chút cho phần đề này ở bước pre-training. Vì vậy, một cách có thể áp dụng để giải quyết vấn đề nêu trên đó là áp dụng kỹ thuật Instruction Fine-tuning trên một bộ dữ liệu với các tác vụ kêu gọi mô hình lớn đưa ra các lời gọi hàm phù hợp với một yêu cầu và một danh sách hàm định nghĩa sẵn trong prompt.



Hình 12: Minh họa về bài toán huấn luyện mô hình với dữ liệu là các yêu cầu gọi hàm để giải quyết một yêu cầu từ người dùng.

Trong phần này, chúng ta sẽ cùng tìm hiểu các bước áp dụng Instruction Fine-tuning mô hình Llama 3.2 1B Instruct trên bộ dữ liệu về function calling. Các bước thực hiện như sau:

#### III.1. Tải và import các thư viện cần thiết

```
1 !pip install unsloth transformers trl torch
```

```

1 import unsloth
2 import os, torch, random, json
3 from datasets import load_dataset
4 from transformers import (
5     TrainingArguments,
6     set_seed
    
```

```

7 )
8 from trl import SFTTrainer, SFTConfig
9 from unsloth import FastLanguageModel

```

## III.2. Khai báo mô hình

Ta khai báo mô hình pre-trained để thực hiện huấn luyện như sau:

```

1 model_id = "meta-llama/Llama-3.2-1B-Instruct"
2 max_seq_length = 2048
3 model, tokenizer = FastLanguageModel.from_pretrained(
4     model_id,
5     max_seq_length = max_seq_length,
6     load_in_4bit = True,
7 )

```

## III.3. Khai báo cài đặt và áp dụng LoRA vào mô hình

Để viết huấn luyện ít tốn kém hơn và vẫn đảm bảo tính hiệu quả, ta ứng dụng kỹ thuật LoRA như sau:

```

1 model = FastLanguageModel.get_peft_model(
2     model,
3     r = 16,
4     lora_alpha = 32,
5     lora_dropout = 0.05,
6     target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
7                       "gate_proj", "up_proj", "down_proj"],
8     bias = "none"
9 )

```

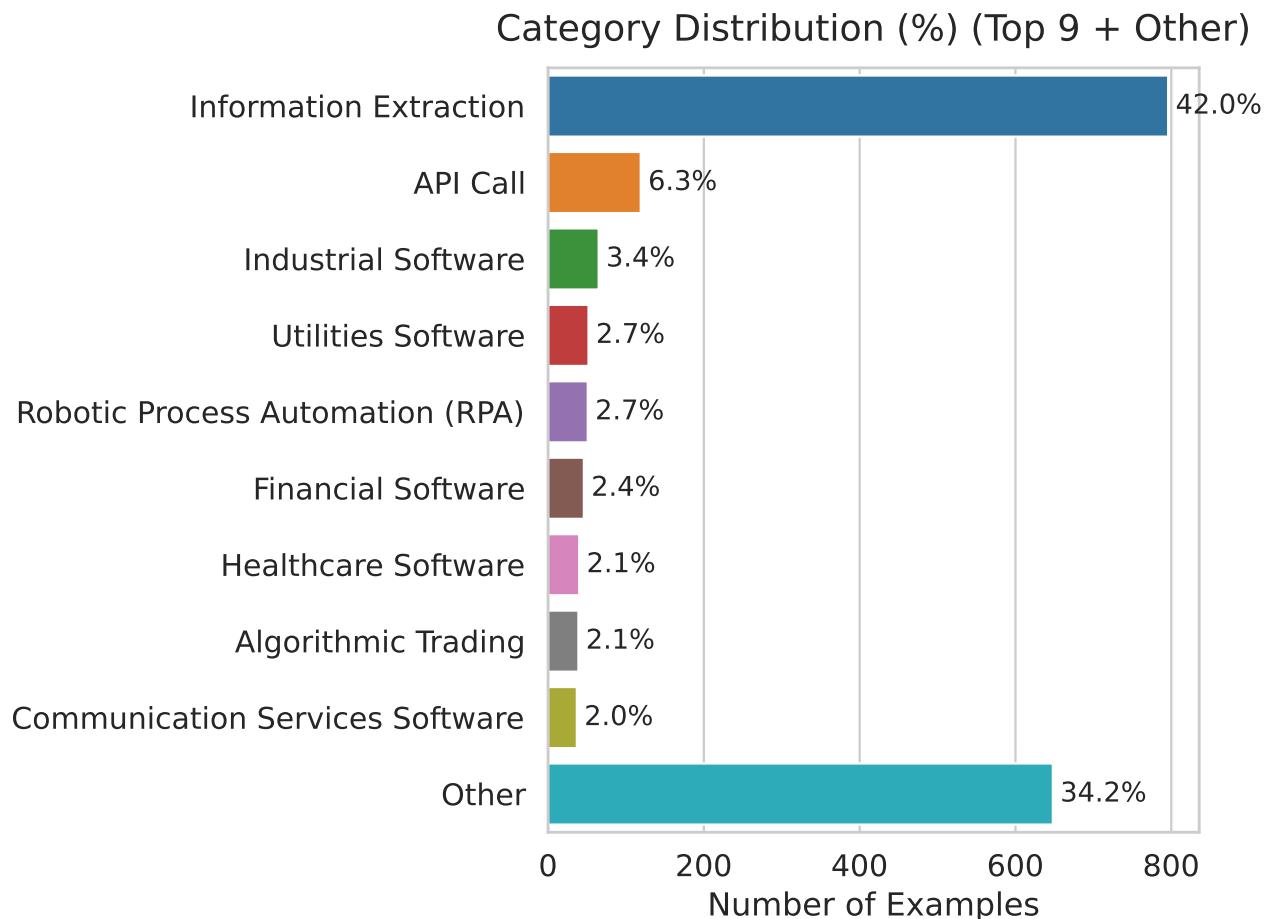
## III.4. Tải bộ dữ liệu

Trong bài này, ta sử dụng bộ dữ liệu hermes-function-calling-v1 từ NousResearch để thực hiện tinh chỉnh mô hình. Về tổng quan, đây là một bộ dữ liệu instruction với nội dung về tool calling trên nhiều ngữ cảnh khác nhau. Để tải bộ dữ liệu này, các bạn thực thi đoạn code sau:

```

1 dataset_id = "NousResearch/hermes-function-calling-v1"
2 raw_ds = load_dataset(dataset_id, split="train")

```



Hình 13: Thống kê sơ bộ về các topic tool calling trong bộ dữ liệu Hermes Function Calling V1.

Với bộ dữ liệu này, bởi sự đa dạng về mặt ngữ cảnh gọi tool, ta kỳ vọng rằng sau khi huấn luyện trên bộ này mô hình sẽ có thể cải thiện được khả năng in-context learning với bất kì nội dung về tool calling.

### III.5. Tiền xử lý bộ dữ liệu về format conversation của Llama

```

1 SYS_PROMPT = (
2     "Cut-off knowledge: Dec 2023\n"
3     "Today: 23 July 2024\n\n"
4     "You are a helpful assistant with tool-calling capabilities."
5 )
6
7 def to_chat(example):
8     msgs = [{"role": "system", "content": SYS_PROMPT}]
9     role = {"system": "system", "human": "user", "gpt": "assistant"}
10    for turn in example["conversations"]:
11        msgs.append({"role": role[turn["from"]]}, "content": turn["value"])
12    chat_str = tokenizer.apply_chat_template(

```

```

13     msgs,
14     tokenize=False,
15     add_generation_prompt=False
16   )
17   return {"text": chat_str}
18
19 ds = raw_ds.map(to_chat).remove_columns(
20   [c for c in raw_ds.column_names if c != "text"]
21 )

```

### III.6. Khai báo các cài đặt huấn luyện và thực hiện huấn luyện mô hình

Vì bộ dữ liệu Hermes có một format conversation khác nhau, cụ thể là về tên các role trong hội thoại. Ta cần định nghĩa một hàm đơn giản để chuyển đổi các vai trò trên về định dạng của mô hình mà chúng ta đang sử dụng, trong trường hợp này là Llama 3.2:

```

1 training_args = TrainingArguments(
2   output_dir = "./llama3-toolcall-lora",
3   per_device_train_batch_size = 16,
4   gradient_accumulation_steps = 2,
5   num_train_epochs = 1,
6   learning_rate = 2e-4,
7   lr_scheduler_type = "cosine",
8   bf16 = torch.cuda.is_bf16_supported(),
9   logging_steps = 25,
10  save_strategy = "epoch",
11  push_to_hub = False,
12)
13
14 sft_cfg = SFTConfig(dataset_text_field="text",
15                      max_seq_length=max_seq_length)
16
17 trainer = SFTTrainer(
18   model      = model,
19   tokenizer  = tokenizer,
20   train_dataset = ds,
21   args       = training_args,
22   **sft_cfg.to_dict(),
23)
24 trainer.train()

```

### III.7. Lưu mô hình đã huấn luyện

Cuối cùng, ta lưu lại các thông tin mô hình sau huấn luyện để có thể tái sử dụng như sau:

```

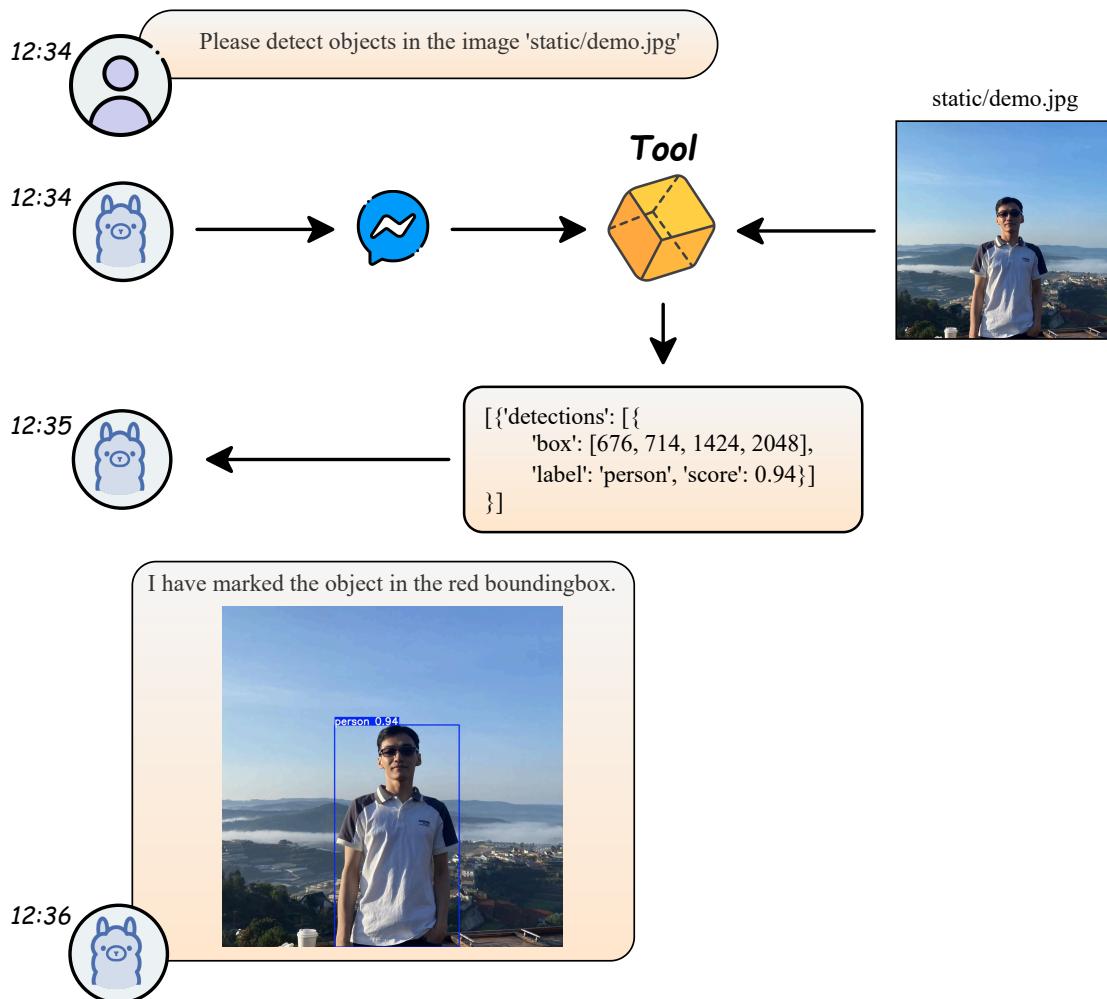
1 trainer.model.save_pretrained("./llama3-toolcall-lora")
2 tokenizer.save_pretrained("./llama3-toolcall-lora")

```

## IV. Giải quyết Computer Vision task với Tool Calling

Trong phần này, chúng ta sẽ cùng tìm hiểu cách ứng dụng kỹ thuật Tool Calling vào mô hình lớn (chỉ xử lý text) để giải các bài toán liên quan đến thị giác máy tính như ở phần I. đã đề cập. Cụ thể, ta sẽ xây dựng một chương trình có khả năng thực hiện các tác vụ Object Detection Và Object Segmentation trên một ảnh. Như vậy, Input/Output của bài này là:

- **Input:** Một ảnh có sẵn trong máy tính và một câu mô tả bài toán (object detection hoặc object segmentation) cho mô hình.
- **Output:** Giải quyết thành công yêu cầu đầu vào, có thể gồm trả về tọa độ bounding box hoặc mask của vật thể trong ảnh.



Hình 14: Minh họa bài toán gọi tool là các mô hình vision để giải quyết các tác vụ thị giác máy tính với LLMs Tool Calling.

Chúng ta sẽ cùng cài đặt theo các bước như sau:

## IV.1. Tải các thư viện cần thiết

Trong phần bài này, chúng ta cần cài đặt không chỉ các thư viện liên quan đến việc huấn luyện mô hình lớn như các phần trước mà còn phải tải toàn bộ các thư viện có liên quan đến các công cụ vision mà chúng ta sẽ triển khai. Trong trường hợp này, ta cần tải thêm thư viện Ultralytics để có thể sử dụng mô hình Detection từ đây.

```
1 !pip install torch transformers Pillow opencv-python ultralytics
```

## IV.2. Import các thư viện cần thiết

```
1 import os
2 import torch
3 import json
4 import ast
5 import sys
6 from transformers import SamModel, SamProcessor
7 from transformers import AutoModelForCausalLM, AutoTokenizer
8 from transformers import GenerationConfig
9 from PIL import Image
10 from ultralytics import YOLO
```

## IV.3. Định nghĩa Tools

Tới đây, ta định nghĩa một số tool dưới dạng là các hàm Python. Mỗi hàm này sẽ có chức năng giải quyết một bài toán thị giác máy tính nào đó. Trong trường hợp này, chúng ta sẽ tạo hai hàm dùng để giải bài detection và bài segmentation thông qua pre-trained models có sẵn.

### IV.3.1. Hàm detection

Hàm này gói toàn bộ quy trình YOLO v11 dưới một giao diện thống nhất: nhận đường dẫn ảnh, suy luận, chuyển kết quả sang định dạng JSON và tùy chọn trực quan hóa. Nhờ đó, LLM chỉ cần gọi một tên hàm duy nhất mà không phải quan tâm tới nội bộ mô hình thị giác.

```
1 detection_model_id = "yolo11n.pt"
2 detection_model = YOLO(detection_model_id)
3 def run_detection(image_path: str, is_visualize: bool = False):
4     """YOLOv11: return list of {box, label, score} for a single image."""
5     results = detection_model(image_path)
6     r = results[0]
7
8     detections = []
9     for box in r.boxes:
10         coords = box.xyxy.cpu().numpy().flatten().tolist()
11         score = float(box.conf.cpu().numpy().item())
12         cls_id = int(box.cls.cpu().numpy().item())
```

```

13     detections.append({
14         "box": coords,
15         "label": r.names[cls_id],
16         "score": score,
17     })
18
19     if is_visualize:
20         r.show()
21
22     return {"detections": detections}

```

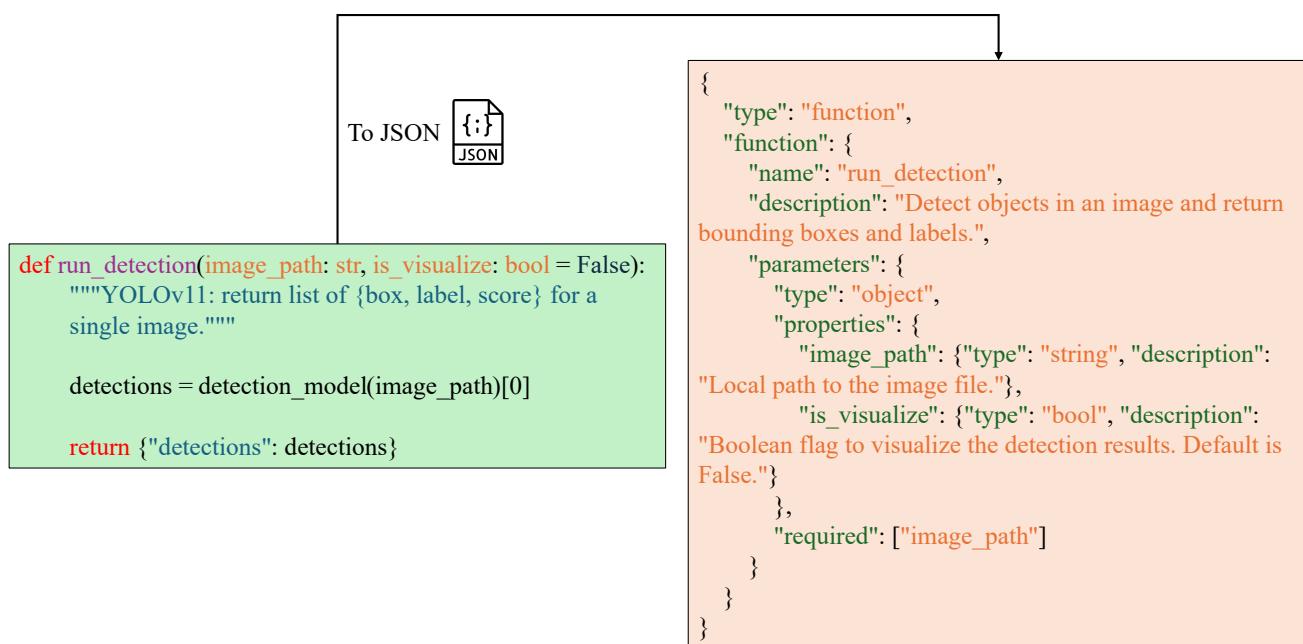
#### IV.3.2. Hàm segmentation

Tương tự hàm detection, hàm segmentation được thiết kế để trích xuất trực tiếp các binary masks từ SAM và chuyển đổi về kiểu dữ liệu thuần Python. Cách đóng gói này hỗ trợ việc truyền-nhận dữ liệu giữa các thành phần một cách trơn tru.

```

1 segmentation_model_id = "facebook/sam-vit-base"
2 sam_processor = SamProcessor.from_pretrained(segmentation_model_id)
3 sam_model = SamModel.from_pretrained(segmentation_model_id)
4 def run_segmentation(image_path: str):
5     """SAM: return binary masks as nested lists"""
6     img = Image.open(image_path).convert("RGB")
7     inputs = sam_processor(images=img, return_tensors="pt")
8     outputs = sam_model(**inputs)
9
10    masks = outputs.pred_masks.squeeze(0).cpu().detach().numpy().tolist()
11    return {"masks": masks}

```



Hình 15: Minh họa việc đưa một số thông tin của hàm mà chúng ta muốn mô hình sử dụng thành một dạng JSON.

#### IV.3.3. Định nghĩa danh sách tools

```

1 tool_functions = [
2     "run_detection": run_detection,
3     "run_segmentation": run_segmentation
4 ]

```

Để giúp mô hình nắm được danh sách các hàm mà nó có thể được sử dụng, ta định nghĩa một list các JSON với nội dung gồm các tên hàm và các thông tin chi tiết có liên quan. Theo đó, ta khai báo biến với nội dung như sau:

```

1 functions_lst = [
2     {
3         "type": "function",
4         "function": {
5             "name": "run_detection",
6             "description": "Detect objects in an image and return bounding boxes and labels."
7                 ,
8             "parameters": {
9                 "type": "object",
10                "properties": {
11                    "image_path": {"type": "string", "description": "Local path to the image
12                                    file."},
13                    "is_visualize": {"type": "bool", "description": "Boolean flag to
14                                    visualize the detection results. Default is
15                                    False."}
16                },
17                "required": ["image_path"]
18            }
19        }
20    },
21    {
22        "type": "function",
23        "function": {
24            "name": "run_segmentation",
25            "description": "Segment objects in an image and return binary masks.",
26            "parameters": {
27                "type": "object",
28                "properties": {
29                    "image_path": {"type": "string", "description": "Local path to the image
30                                    file."}
31                },
32                "required": ["image_path"]
33            }
34        }
35    }
36 ]

```

#### IV.4. Định nghĩa system prompt

Khi đã có danh sách các thông tin hàm dưới dạng một danh sách json (dict), chúng ta sẽ định nghĩa system prompt nhằm đưa vào mô hình để mô hình có thể hiểu được tổng quan nhiệm vụ

cần làm cũng như danh sách các công cụ mà nó có thể được sử dụng. Nội dung system prompt như sau:

```

1 SYSTEM_PROMPT = """
2 You are an expert in composing functions. You are given a question and a set of possible
   functions.
3 Based on the question, you will need to make one or more function/tool calls to achieve the
   purpose.
4 If none of the function can be used, point it out. If the given question lacks the parameters
   required by the function,
5 also point it out. You should only return the function call in tools call sections.
6
7 If you decide to invoke any of the function(s), you MUST put it in the format of [func_name1(
   params_name1=params_value1, params_name2=
   params_value2...), func_name2(params)]\n
8 You SHOULD NOT include any other text in the response.
9
10 Here is a list of functions in JSON format that you can invoke.\n\n{functions}\n""".format(
   functions=functions_lst)

```

## IV.5. Khai báo mô hình

Tương tự như các bài trước, ở đây ta cũng sử dụng mô hình Llama 3.2 1B Instruct bởi tính gọn nhẹ của nó, phù hợp để huấn luyện trên Google Colab:

```

1 model_id = "meta-llama/Llama-3.2-1B-Instruct"
2 llm = AutoModelForCausalLM.from_pretrained(model_id,
3                                             device_map=device,
4                                             torch_dtype=torch.bfloat16)
5 tokenizer = AutoTokenizer.from_pretrained(model_id)

```

## IV.6. Khai báo generation config và hội thoại

Tương tự như các đoạn code liên quan đến việc sử dụng các mô hình lớn trong thư viện transformers, chúng ta cũng cần phải định nghĩa một list hội thoại mở đầu gồm có system role và user role. Trong đó system role chứa nội dung system prompt và user role chứa yêu cầu đầu vào. Ta đồng thời định nghĩa một số tham số liên quan đến tạo sinh, các bạn có thể điều chỉnh cho phù hợp với mong muốn của mình:

```

1 messages = [
2     {
3         "role": "system",
4         "content": SYSTEM_PROMPT
5     },
6     {
7         "role": "user",
8         "content": "Please detect objects in the image 'static/demo.jpg'."
9     }
10 ]
11 generation_config = GenerationConfig(
12     max_new_tokens=512,

```

```

13     temperature=1.0,
14     top_p=0.98,
15     top_k=50,
16     repetition_penalty=1.2,
17     do_sample=True,
18     num_return_sequences=1,
19     bos_token_id=tokenizer.bos_token_id,
20     eos_token_id=tokenizer.eos_token_id,
21     pad_token_id=tokenizer.eos_token_id,
22 )

```

## IV.7. Thực thi pipeline giải task vision thông qua lời gọi hàm của LLM

Cuối cùng, ta hoàn tất pipeline gọi tool với các mô hình vision đã định nghĩa theo đoạn code sau:

```

1 messages = [
2     {
3         "role": "system",
4         "content": SYSTEM_PROMPT
5     },
6     {
7         "role": "user",
8         "content": "Please detect objects in the image 'static/demo.jpg' ."
9     }
10 ]
11 generation_config = GenerationConfig(
12     max_new_tokens=512,
13     temperature=1.0,
14     top_p=0.98,
15     top_k=50,
16     repetition_penalty=1.2,
17     do_sample=True,
18     num_return_sequences=1,
19     bos_token_id=tokenizer.bos_token_id,
20     eos_token_id=tokenizer.eos_token_id,
21     pad_token_id=tokenizer.eos_token_id,
22 )
23
24 tokenized_chat = tokenizer.apply_chat_template(messages,
25                                                 tokenize=True,
26                                                 add_generation_prompt=True,
27                                                 return_attention_mask=True,
28                                                 return_tensors="pt").to(device)
29
30 outputs = llm.generate(
31     tokenized_chat,
32     generation_config=generation_config)
33 raw = tokenizer.decode(outputs[0], skip_special_tokens=True)
34 tool_calls_lst_str = raw.split("assistant")[-1]
35
36 try:

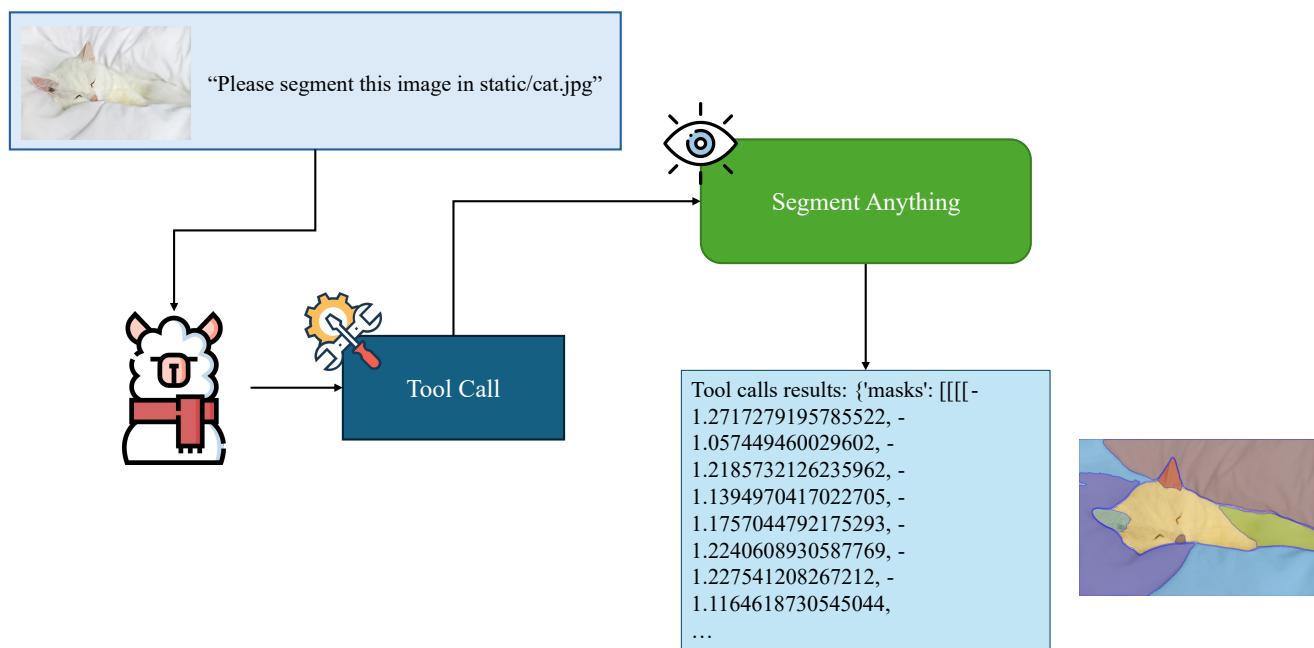
```

```

37     tree = ast.parse(tool_calls_lst_str, mode='eval')
38     call_nodes = tree.body.elts
39 except SyntaxError as e:
40     print("Cannot parse the function call. Try again.")
41     sys.exit(1)
42
43 tool_calls_result = []
44 for call in call_nodes:
45     function_name = call.func.id
46     parameters = {
47         kw.arg: ast.literal_eval(kw.value)
48         for kw in call.keywords
49     }
50
51     result = tool_functions[function_name](**parameters)
52     tool_calls_result.append(result)
53
54 print("Tool calls results:", tool_calls_result)

```

Khi thực thi đoạn code trên, ta sẽ nhận được kết quả là các lời gọi hàm để giải quyết bài toán về thị giác máy tính đầu vào. Tùy thuộc vào cách định nghĩa hàm mà ta sẽ có các chức năng khác nhau để sử dụng.



Hình 16: Minh họa việc giải bài toán segmentation thông qua lời gọi hàm có chứa model SAM (Segment Anything Model).

# V. Câu hỏi trắc nghiệm

1. Function Calling trong LLMs được hiểu là phương thức nào dưới đây?
  - (a) Kỹ thuật cho phép LLM tự động sinh thêm hàm nội bộ để mở rộng khả năng suy luận.
  - (b) Cơ chế tích hợp LLM với các hàm/công cụ bên ngoài để thực hiện các tác vụ vượt quá giới hạn bản thân mô hình.
  - (c) Phương pháp huấn luyện LLM bằng cách chèn các hàm toán học phức tạp trực tiếp vào kiến trúc Transformer.
  - (d) Hình thức tinh chỉnh mô hình ngôn ngữ lớn thông qua việc nhúng mã Python vào prompt.
2. Mục tiêu chính của Function Calling trong khuôn khổ LLMs là gì?
  - (a) Tăng tốc độ xử lý token và giảm độ trễ phản hồi.
  - (b) Bổ sung thông tin thời gian thực và dữ liệu đa phương thức mà mô hình chưa được đào tạo.
  - (c) Nén kích thước mô hình để triển khai trên các thiết bị có tài nguyên hạn chế.
  - (d) Cải thiện độ chính xác ngôn ngữ pháp trong quá trình sinh văn bản.
3. Khi LLM quyết định thực hiện một lời gọi hàm (function call), nó sẽ xuất ra gì để client có thể thực thi hàm tương ứng?
  - (a) Một đối tượng JSON chứa tên hàm cùng tham số kèm kiểu dữ liệu.
  - (b) Một file script Python hoàn chỉnh bao gồm hàm và các thư viện liên quan.
  - (c) Một URI trả đến endpoint của hàm được định nghĩa trên server.
  - (d) Một đoạn văn bản mô tả chức năng và cách sử dụng hàm.
4. Hạn chế nổi bật của các LLM phiên bản trước mà Function Calling hướng tới khắc phục là gì?
  - (a) Khả năng sinh văn bản không vượt quá 512 token.
  - (b) Không thể truy xuất hoặc khai thác thông tin thời gian thực.
  - (c) Không hỗ trợ xử lý đa ngôn ngữ trong cùng một đầu vào.
  - (d) Không thể phân tích dữ liệu hình ảnh hoặc âm thanh đầu vào.
5. Dataset “hermes-function-calling-v1” của NousResearch chủ yếu được sử dụng để:
  - (a) Đánh giá khả năng dịch ngôn ngữ qua API function calling.
  - (b) Huấn luyện LLM cải thiện kỹ năng tóm tắt văn bản.
  - (c) Fine-tuning LLM để thực hành Function Calling.
  - (d) Triển khai chatbot đa kênh với giao diện người dùng phong phú.

6. Khi fine-tuning LLaMA 3.2 trên bộ dữ liệu Function Calling, mục tiêu chính là gì?
  - (a) Giảm độ trễ khi triển khai trên môi trường sản xuất.
  - (b) Nâng cao khả năng sinh mã JSON hoặc lệnh gọi hàm với cú pháp chính xác.
  - (c) Tăng cường năng lực sáng tác thơ văn, tiểu thuyết.
  - (d) Cải thiện khả năng tổng hợp và sinh ảnh AI.
7. Sau khi client nhận được phản hồi JSON chứa lệnh gọi hàm từ LLM, bước tiếp theo thường là gì?
  - (a) Gửi JSON đó ngược lại cho LLM để xác nhận cú pháp.
  - (b) Phân tích (parse) JSON rồi gọi hàm tương ứng với tên và tham số đã chỉ định.
  - (c) Hiển thị JSON cho người dùng để họ quyết định hàm nào được thực thi.
  - (d) Lưu JSON vào cơ sở dữ liệu để sử dụng cho quá trình huấn luyện tiếp theo.
8. Trong ngữ cảnh Function Calling, định nghĩa “function schema” thường được biểu diễn dưới dạng nào sau đây?
  - (a) Tập tin YAML schema mô tả tham số và kiểu dữ liệu.
  - (b) JSON Schema định nghĩa tên hàm, tham số và kiểu dữ liệu tương ứng.
  - (c) Tệp ProtoBuf (\*.proto) dùng để serialize tham số hàm.
  - (d) Định nghĩa XML tương tác qua giao thức SOAP API.
9. Trong ngữ cảnh Function Calling, khi mô hình gọi một hàm thông qua JSON Schema, điều gì sẽ xảy ra?
  - (a) Mô hình xác định các tham số cần thiết và đảm bảo các giá trị đầu vào hợp lệ.
  - (b) Mô hình trực tiếp xử lý dữ liệu đầu vào và trả về kết quả mà không cần gọi hàm.
  - (c) Mô hình lưu trữ kết quả trả về trong cơ sở dữ liệu để huấn luyện lại sau.
  - (d) Mô hình yêu cầu người dùng cung cấp các giá trị đầu vào theo định dạng XML.
10. Khi soạn prompt cho Function Calling, yếu tố quan trọng nhất để đảm bảo LLM sinh ra lệnh gọi hàm hợp lệ là gì?
  - (a) Mô tả chi tiết thuật toán bên trong hàm.
  - (b) Định nghĩa rõ ràng tên hàm và kiểu dữ liệu của từng tham số.
  - (c) Cung cấp ví dụ về file đầu vào và đầu ra mong muốn.
  - (d) Mô tả môi trường thực thi (environment) của hàm.

## VI. Tài liệu tham khảo

- [1] Z. et al., *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face*, <https://arxiv.org/abs/2303.17580>, 2023.
- [2] OpenAI, *LLMs as Tool Makers*, <https://openai.com/blog/function-calling>, OpenAI Blog, 2023.
- [3] OpenAI, *Function Calling*, <https://platform.openai.com/docs/guides/function-calling>, OpenAI Docs, 2023.
- [4] OpenAI, *Tool Calling Example*, [https://github.com/openai/openai-cookbook/blob/main/examples/How\\_to\\_call\\_functions\\_with\\_chat\\_models.ipynb](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb), OpenAI Cookbook.
- [5] *JSON Schema Specification*, <https://json-schema.org/>.
- [6] LangChain, *Function Calling in AI Agents (Mistral 7B use case)*, <https://blog.langchain.dev/function-calling-agentic-patterns>.

## VII. Phụ lục

1. **Solution:** Các file code cài đặt hoàn chỉnh và phần trả lời nội dung trắc nghiệm có thể được tải tại [đây](#) (**Lưu ý:** Sáng thứ 2 khi hết deadline phần nội dung này, ad mới copy các tài liệu bài giải nêu trên vào đường dẫn).
2. **Demo:** Web demo và mã nguồn của ứng dụng có thể được truy cập tại [đây](#).
3. **Rubric:**

Mục	Kiến Thức	Đánh Giá
I.	<ul style="list-style-type: none"> <li>- Kiến thức về mô hình lớn (LLMs).</li> <li>- Kiến thức về AI Agents.</li> <li>- Kiến thức về cách hoạt động khả năng tương tác với các hàm/công cụ bên ngoài của LLMs.</li> <li>- Kiến thức về việc cải thiện khả năng sử dụng hàm/công cụ của LLMs thông qua Instruction Tuning.</li> </ul>	<ul style="list-style-type: none"> <li>- Nắm được lý thuyết về mô hình lớn (LLMs) và AI Agents.</li> <li>- Nắm được lý thuyết về việc trang bị khả năng sử dụng function/tool cho LLMs nhằm giải quyết các tác vụ mà mô hình gốc sẽ gặp khó khăn hoặc không làm được.</li> <li>- Có khả năng triển khai một mô hình LLMs Function Calling để giải quyết một số tác vụ liên quan đến thị giác máy tính.</li> </ul>