Function Calling: LLM이 외부 세계와 소통하는 방법 (ft. Qwen, llama, Gemma)

- based on https://devocean.sk.com/experts/techBoardDetail.do?ID=167407&boardType=experts&page=&searchData=&subIndex=&idList=&searchText=&techType=

Summary

- 에이전틱 AI는 AI가 스스로 계획을 세우고 외부 도구를 활용해 문제를 해결하는 자율성과 판단력을 가진 지능형 도구로 진화한 것을 말합니다.
- Function Calling은 LLM이 자연어 요청을 이해하고 적합한 외부 API를 호출해 실시간 정보를 가져오는 기술로, 사용자가 요청한 정보를 보다 정확하고 유연하게 제공할 수 있게 합니다.
- 오픈소스 LLM을 통해 Function Calling을 구현하는 방법도 제공되며, 프롬프트 엔지니어링을 통해 모델이 함수 호출 정보를 생성하고, 외부 애플리케이션이 이를 처리해 결과를 제공하는 구조입니다.

Introduction

- 요즘 AI 업계에서 가장 뜨거운 키워드를 하나만 꼽으라면 단연 Agentic AI일 것입니다.
- 기존 소프트웨어에서 'Agent'라는 용어는 주로 단순하고 반복적인 작업을 자동화하는 도구를 지칭했습니다.
- 그러나 LLM의 등장은 이 개념에 새로운 의미를 부여했습니다. 이제 AI Agent는 단순한 자동화를 넘어서 일정 수준의 자율성과 판단력을 갖춘 지능형 도구로 진화하고 있습니다.
- 이렇게 스스로 판단하고 행동하는 AI를 우리는 "에이전틱(agentic)하다"라고 표현하며, 이러한 시스템을 에이전틱 AI라고 부릅니다.
- 마치 영화 속 자비스처럼, 사용자의 복잡한 요구사항을 이해하고 필요한 정보를 찾아오거나 외부 시스템과 연동하여 작업을 수행하는 미래가 현실로 다가오고 있는 것이죠.
- 그렇다면 이렇게 자율적인 AI가 어떻게 현실 세계와 상호작용하며 복잡한 임무를 수행할 수 있을까요?
- 이러한 상호 작용을 가능하게 하는 기반 기술 중 하나가 바로 **Function Calling** 입니다.
- Function Calling은 LLM이 자연어 요청을 이해하고 적합한 외부 함수(API)를 호출하도록 유도하는 기술로, 사용자의 의도를 정확히 파악해 필요한 정보를 준비하는 메커니즘입니다.
- 예를 들어 사용자가 “오늘 날씨 어때?”라고 물으면, LLM이 날씨 API 호출에 필요한 정보를 생성하여 실시간 날씨 정보를 가져오게 됩니다.
- 이 덕분에 LLM은 더 이상 학습된 데이터에만 의존하지 않습니다.
- 실시간 정보에 접근하고, 반복적인 작업을 자동화하며, 복잡한 워크플로우를 조율하는 등 훨씬 더 강력하고 유연한 능력을 갖추게 되었습니다.
- 이번 글에서는 LLM의 잠재력을 한 단계 끌어올리는 Function Calling 기술에 대해 알아보겠습니다.
- 이 기술이 왜 필요하게 되었는지, 어떤 원리로 작동하는지, 그리고 실제 오픈소스 LLM에서 어떻게 활용할 수 있는지 차근차근 살펴보겠습니다.

Function Calling의 개념 등장

- 기존의 LLM은 실시간 정보를 가져오거나 복잡한 계산을 수행하거나 외부 시스템과 상호작용하는 데 한계가 있었습니다.
- 예를 들면, 특정 시점 이후의 최신 뉴스나 현재 날씨와 같은 정보는 정확히 답변하기 어려웠죠.
- 이러한 한계를 극복하기 위해, 외부 정보를 LLM의 답변 생성 과정에 통합하려는 다양한 아이디어들이 등장하기 시작했습니다.
- 그 중 대표적인 사례가 2022년에 발표된 ReAct (Reasoning and Acting) 기법입니다.

- ReAct는 모델이 질문에 대해 먼저 내부적 추론(Thought)을 수행하고 이를 바탕으로 필요한 외부 행동(Act)을 결정합니다.
- 외부 행동을 수행하면 그 결과를 관찰(Observation)하고, 이 관찰 결과를 바탕으로 다시 추론(Thought)을 반복하여 점진적으로 최적의 답변에 도달 할 수 있다는 기법입니다.
- 어찌 보면 실제 인간의 문제 해결 과정과 유사합니다.
- 예를 들어 여행 계획을 세울 때, 우리는 목적지를 정하고 교통편이나 숙소를 알아본 뒤, 필요하면 날씨나 현지 정보를 검색합니다. 그런 다음 세부 일정을 조율하고 예산을 세운 후 실제 예약을 진행하죠.
- ReAct 역시 이러한 과정을 모방하여 언어 모델이 추론과 행동을 번갈아 수행하며 문제를 해결하도록 설계되었습니다.
- 2023년 초, Meta AI에서 발표한 Toolformer가 등장하면서 한 걸음 더 나아갔습니다.
- Toolformer는 GPT 계열 모델을 기반으로, 모델 스스로 언제 어떤 API를 호출해야 하는지를 자기지도 학습(self-supervised learning) 방식으로 학습하도록 설계되었습니다.
- 이를 통해 계산기, 위키백과 검색, 달력 확인 등 다양한 외부 도구 활용이 가능해졌으며, 이후 여러 툴 기반 LLM 프레임워크 개발에 큰 영향을 주었습니다.
- 이러한 연구 흐름 속에서 2023년 6월, OpenAI가 GPT 모델에 Function Calling 기능을 공식적으로 도입하면서 이 기술은 본격적으로 대중화되었습니다.
- 특히 ChatGPT 플러그인이 등장하며 LLM이 사용자의 요청을 이행하기 위해 외부 API를 호출해야 할 필요성이 커졌고, Function Calling은 이를 위한 표준화된 인터페이스로 자리 잡게 됩니다.

Function Calling의 원리

- 일반적으로 소프트웨어 개발에서 서비스 간 통신 규약을 API(Application Programming Interface)라고 부릅니다.
- 개발자는 필요에 따라 API를 통해 외부 시스템과 데이터를 주고받을 수 있는데, 이때 주로 사용하는 데이터 형식 중 하나가 JSON 입니다.
- **Function Calling의 핵심 아이디어는 LLM이 자연어 답변을 생성하는 대신, 특정 상황에서 소프트웨어가 이해하고 처리할 수 있는 구조화된 데이터(예: JSON 형식)를 출력하도록 유도하는 것입니다.**
- 예를 하나 들어보죠. 사용자가 "오늘 서울 날씨 어때?"라고 질문했을 때, 기존 LLM은 학습된 데이터에 기반하기 때문에 과거의 날씨 정보나 잘못된 정보를 제공했을 것입니다. 하지만 Function Calling이 적용된 모델은 사용자의 요청을 분석하여 "정확한 최신 날씨 정보는 외부 날씨 API를 통해 가져오는 것이 좋겠다"라고 판단하고, API 호출에 필요한 정보를 다음과 같은 JSON 형식으로 생성합니다.

In [ ]:
```
{
  "name": "get_current_weather",
  "arguments": {
    "location": "Seoul"
    }
}
```

이렇게 모델이 생성한 JSON 데이터를 사용자 애플리케이션이 받아 처리하고, 이 정보를 바탕으로 실제 날씨 API를 호출하여 최신 날씨 정보를 가져옵니다.

API로부터 받은 응답 데이터를 다시 LLM에게 전달하면, 모델은 이 정보를 활용하여 최종 답변을 정확하고 자연스럽게 생성할 수 있습니다.

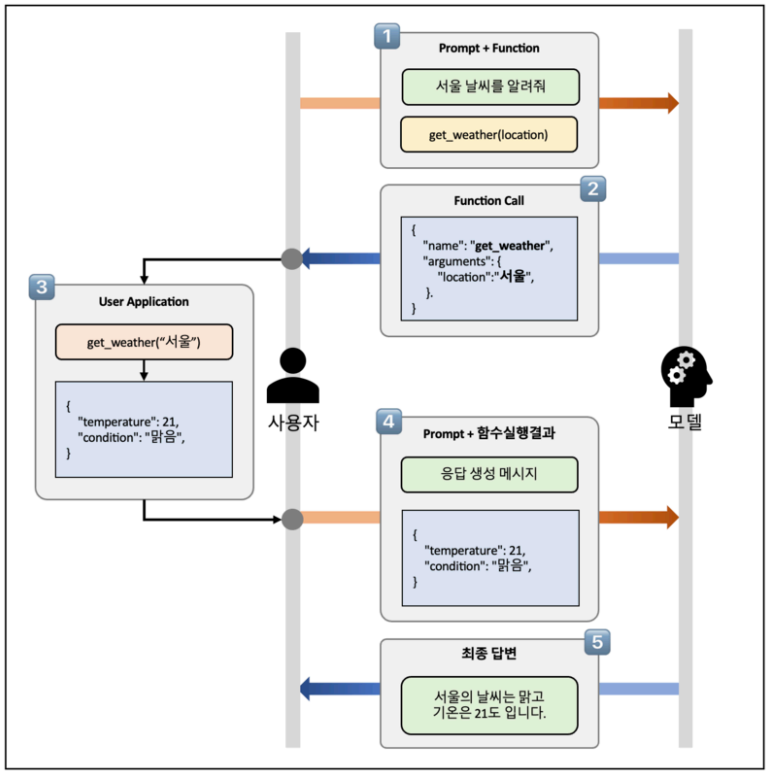
Function Calling의 주요 특징 중 하나는 이를 구현하기 위해 모델의 기본 구조를 변경할 필요가 없다는 점입니다.

이는 모델이 특정 조건(예: 함수 호출이 필요하다고 판단될 때)에서 **JSON 형식의 출력을 생성하도록 파인튜닝 하거나, 프롬프트 엔지니어링을 통해 유도**하는 방식으로 구현됩니다.

이러한 접근 방식 덕분에 기존 LLM 기술 위에 유연하게 Function Calling 기능을 추가할 수 있다는 장점이 있습니다.

Function Calling의 단계적 작동방식

- 아래 다이어그램은 Function Calling의 일반적인 처리 흐름을 단계별로 보여줍니다.



1단계: 사용자 요청 및 함수 준비

- 함수 정의 첫번째로 해야 할일은 LLM이 사용할 수 있는 함수를 미리 정의 하는 것입니다. 함수 정의는 일반적으로 함수 이름(name), 기능 설명(description), 필요한 입력값(parameters) 등을 포함하며, 명확한 구조 전달을 위해 JSON 스키마 형식으로 작성됩니다.
- 예를 들면 특정 도시의 현재 날씨를 조회 하는 함수 get_weather는 아래와 같은 형식으로 정의할 수 있을 것입니다.

In [ ]:
```json
{
  "name": "get_weather",
  "description": "특정 도시의 현재 날씨 정보를 조회합니다.",
  "parameters": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "날씨를 조회할 도시 이름 (예: 서울, 부산, New York)"
      },
    },
    "required": ["location"]
  }
}
```

- 함수 정의 형식을 JSON 스키마로 사용하는 이유는 LLM 에게 함수를 전달 할 때 함수가 어떠한 역활을 하는지 자연어로 주저리 주저리 설명하는 것보다 함수의 구조와 요구사항을 LLM에게 보다 명확하고 체계적으로 전달 수 있기 때문입니다.

사용자 요청

- 이제 정의된 함수 목록(Tools)과 함께 사용자의 질문(예: "서울 날씨 알려줘")을 모델 API에 전달합니다.

Function 에서 TOOLS로

- 초기에는 OpenAI API 등에서 functions라는 파라미터를 사용해 호출 가능한 함수 목록을 지정했지만, 이후에는 이를 **tools**라는 더 확장성 있는 구조로 대체하게 되었습니다.
- tools는 단순한 함수 호출 만을 위한 것이 아니라, 검색, 코드 실행, 데이터베이스 질의 등 다양한 종류의 외부 도구를 포괄할 수 있도록 설계된 구조입니다.

- 즉, 함수 호출이라는 좁은 개념에서 벗어나, LLM이 외부 세계와 더 유연하고 풍부하게 상호작용할 수 있도록 진화한 것입니다.

2단계: 함수 호출 생성

- 모델이 함수 목록과 사용자의 요청을 받으면 다음 과정을 수행하게 됩니다.
- 요청에서 '날씨 조회'라는 의도와 '서울'이라는 위치 정보를 추출합니다.
- get_weather 함수의 설명("특정 도시의 현재 날씨 정보를 조회합니다.")과 파라미터(location)가 사용자의 의도 및 추출된 정보와 가장 잘 맞는다고 판단합니다.
- get_weather 함수의 필수 파라미터인 location에 추출된 정보 '서울'을 매핑합니다.
- 최종적으로 다음과 같은 함수 호출 정보를 JSON 형식으로 생성하여 반환합니다.

In [ ]:
```
{
  "name": "get_weather",
  "arguments": {
    "location": "서울"
  }
}
```

3단계: 외부 시스템에서 함수 실행

- LLM이 생성한 JSON 형식의 함수 호출 정보는 개발자가 구축한 애플리케이션 코드로 전달됩니다.
- 여기서 핵심은 LLM은 이 호출 정보를 생성할 뿐, 스스로 외부 함수나 API를 직접 실행하지는 않는다는 것입니다.
- Function Calling 개념을 처음 접할 때 LLM이 함수 실행까지 모두 처리하는 것으로 오해하기 쉽습니다. ("LLM이 호출 정보를 만들고 직접 함수를 실행해서 결과까지 가져오는 것 아닌가?")
- 하지만 실제 역할은 명확히 나뉩니다. LLM은 사용자의 요청 의도를 파악하고, 가장 적합한 함수를 선택하여 '어떻게 호출해야 하는지'에 대한 구조화된 지시(JSON)를 만드는 역할에 집중합니다.
- 실제 함수의 실행 책임은 애플리케이션에게 있습니다.
- 애플리케이션은 LLM으로부터 이 지시(JSON)를 받아, 자신의 코드 환경 내에서 해당 함수를 직접 실행하고 그 결과를 다시 LLM에게 전달합니다.
- 즉, **LLM은 '지시 생성자', 애플리케이션은 '실행자'** 로 역할이 분담된 구조로 이해해야 합니다.
- 따라서 애플리케이션 코드는 전달받은 JSON을 파싱하여 함수 이름(get_weather)과 인자({"location": "서울"})를 얻고, 이를 바탕으로 실제 get_weather 함수 로직을 실행하는 것입니다.

4단계: 실행 결과 전달

- 애플리케이션 코드가 get_weather 함수를 실행하고 날씨 API로부터 결과를 받으면, 이를 다시 LLM에게 전달합니다.
- 이 결과 역시 구조화된 형태(주로 JSON)로 전달하는 것이 일반적입니다.

In [ ]:
```
{
  "location": "서울",
  "temperature": 21,
}
```

5단계: 최종 응답 생성

- LLM은 4단계에서 전달받은 함수 실행 결과를 활용하여, 사용자 질문("서울 날씨 알려줘")에 대한 최종 답변을 자연어로 생성합니다.
- 예를 들어 최종적으로 사용자는 다음과 같은 답변을 받게 되는 것이죠.
- “현재 서울의 기온은 21도 입니다. "

오픈소스 모델로 Function Calling 구현하기

- Function Calling의 동작 방식을 살펴보았으니, 실제 오픈소스 LLM을 사용하여 Function Calling을 구현해보겠습니다.
- 여기서는 Qwen 2.5, Llama 3.1, Gemma 3 모델을 예시로 사용할텐데, 각 모델이 Function Calling을 처리하는 방식이 조금씩 차이가 있습니다.
- 우선 Qwen2.5 모델을 대상으로 전체 플로우를 살펴본 후 Llama 3.1과 Gemma3의 차이점에 대해 알아보겠습니다.
- 예제 코드는 사용자가 특정 KOSPI 주식 정보를 요청하면, Function Calling을 통해 관련 정보를 조회하여 답변하는 시나리오를 기반으로 합니다.

Qwen 2.5 Function Calling 예제

In [ ]:
```
!pip install yfinance transformers vllm
```

In [ ]:
```
import json
import re
import yfinance as yf
from vllm import LLM, SamplingParams
from transformers import AutoTokenizer
from datetime import datetime
```

모델 및 토크나이저 초기화

- 추론 속도를 높이기 위해 vLLM 으로 모델을 로드 합니다.

In [ ]:
```
# 모델 및 토크나이저 초기화
MODEL_ID = "Qwen/Qwen2.5-7B-Instruct"
#model = LLM(MODEL_ID, tensor_parallel_size=4, max_model_len=2048, max_num_seqs=4)
model = LLM(MODEL_ID, tensor_parallel_size=1)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
```

샘플링 파라메터 설정

- vLLM으로 텍스트를 생성할 때, 출력의 특성을 조절하기 위해 디코딩 파라메터를 조절합니다.
- 함수 호출 생성 시에는 정확성을 위해 temperature를 낮추고, 최종 답변 생성 시에는 자연스러움을 위해 약간 높입니다.

In [ ]:
```
# 샘플링 파라미터 설정 - Qwen 2.5에 맞게 stop token 변경
sampling_params_func_call = SamplingParams(
    max_tokens=256, temperature=0.0, stop=["<|im_end|>"], skip_special_tokens=False
)
sampling_params_text = SamplingParams(
    max_tokens=512, temperature=0.1, top_p=0.95, stop=["<|im_end|>"], skip_special_to
)
```

TOOL의 정의

- 주식 정보를 가져오는 함수 get_kospi_stock_info 에 대해 JSON 스키마 형식으로 함수를 정의합니다.

In [ ]:
```python
# KOSPI 주식 정보
KOSPI_TICKER_MAP = {
    "SK텔레콤": "017670.KS", "삼성전자": "005930.KS", "SK하이닉스": "000660.KS",
    "현대차": "005380.KS", "기아": "000270.KS", "LG에너지솔루션": "373220.KS",
    "NAVER": "035420.KS", "카카오": "035720.KS",
}
```

In [ ]:
```python
# 도구(함수) 정의
TOOLS = [{
    "type": "function",
    "function": {
        "name": "get_kospi_stock_info",
        "description": "특정 KOSPI 주식의 현재 가격 및 기본 정보를 가져옵니다.",
        "parameters": {
            "type": "object",
            "properties": {
                "stock_name_or_code": {
                    "type": "string",
                    "description": "주식 이름(예: 'SK텔레콤') 또는 종목 코드(예: '017670')"
                }
            },
            "required": ["stock_name_or_code"]
        }
    }
}]
```

주식 정보 조회 함수 (사용자 애플리케이션)

- TOOLS에 정의된 get_kospi_stock_info 함수의 실행 코드 입니다. 이 코드는 사용자 애플리케이션의 일부이며, yfinance 라이브러리를 사용해 주식 정보를 조회합니다.
- 함수 실행의 결과 즉 주식명, 현재가, 전일 종가, 통화단위는 JSON 코드로 반환됩니다.
- 다시 한번 상기하지만, 이 함수는 LLM이 실행하는 것이 아니라 사용자 애플리케이션으로 실행됩니다.
- 즉 LLM이 이 함수를 호출하는 것이 아니라 LLM이 생성한 함수 호출 정보를 바탕으로 사용자의 코드에서 이 함수를 호출 하고, 결과 값을 리턴 받게 됩니다.

In [ ]:
```python
# 주가 조회
def get_kospi_stock_info(stock_name_or_code: str) -> str:

    name_or_code = stock_name_or_code.strip()
    ticker_symbol = None

    if re.fullmatch(r'\d{6}', name_or_code):
        ticker_symbol = name_or_code + ".KS"
    elif name_or_code in KOSPI_TICKER_MAP:
        ticker_symbol = KOSPI_TICKER_MAP[name_or_code]
    else:
        ticker_symbol = name_or_code + ".KS"

    stock = yf.Ticker(ticker_symbol)
    stock_info = stock.info

    current_price = stock_info.get('currentPrice')
    previous_close = stock_info.get('previousClose')

    price_to_use = current_price if current_price is not None else previous_close
    price_display = round(price_to_use, 2) if price_to_use is not None else "정보 없음"
    previous_close_display = round(previous_close, 2) if previous_close is not None e

    result = {
        "ticker": ticker_symbol,
        "stock_name": stock_info.get('shortName', name_or_code),
        "current_price": price_display,
        "previous_close": previous_close_display,
        "currency": stock_info.get('currency', 'KRW')
```

```
    }

    return json.dumps(result, ensure_ascii=False)
```

함수 호출 파싱

- LLM이 함수를 호출하기로 결정하면, Qwen 모델은 특정 형식(...)으로 응답을 생성합니다.
- 때문에 이 응답에서 실제 호출할 함수 이름과 인자를 추출하는 함수가 필요합니다.
- llama나 gemma가 응답 형식이 다르기 때문에 추후 이들 모델을 사용한 예제에서는 parse_tool_calls 함수를 해당 모델의 출력 형식에 맞게 수정해야 합니다.

In [ ]:

```
# 함수 호출 파싱
# Qwen LLM 출력에서 <tool_call>...</tool_call> 형식의 함수 호출을 파싱
def parse_tool_calls(content: str):

    tool_calls = []
    pattern = r"<tool_call>(.*?)</tool_call>"
    matches = re.finditer(pattern, content, re.DOTALL)

    last_match_end = 0
    parsed_calls = []

    for match in matches:
        tool_call_content = match.group(1).strip()
        func_data = json.loads(tool_call_content)

        if isinstance(func_data.get("arguments"), str):
            func_data["arguments"] = json.loads(func_data.get("arguments"))

        parsed_calls.append({
            "type": "function",
            "function": {
                "name": func_data.get("name"),
                "arguments": func_data.get("arguments", {})
            },
            "id": f"call_{match.start()}"
        })
        last_match_end = match.end()

    first_match_start = content.find("<tool_call>")
    prefix_text = content[:first_match_start].strip() if first_match_start != -1 else
    if not parsed_calls:
        prefix_text = re.sub(r"<\|im_end\|>\s*$", "", prefix_text).strip()

    assistant_message = {"role": "assistant"}
    if prefix_text:
        assistant_message["content"] = prefix_text
    if parsed_calls:
        assistant_message["tool_calls"] = parsed_calls
    if not prefix_text and not parsed_calls:
         assistant_message["content"] = ""

    if "content" in assistant_message and assistant_message["content"]:
        assistant_message["content"] = re.sub(r"<\|im_end\|>\s*$", "", assistant_mess

    return assistant_message
```

Function Calling 메인 함수

- 이제 정의된 요소들을 사용하여 실제 Function Calling 과정을 실행하는 메인 함수 query_kospi_info를 살펴보겠습니다.
- 예제 코드에서는 전체 동작 방식을 이해하도록 단계별로 결과를 출력하도록 하였습니다.
- 지면상 전체 코드 는 생략하고 출력 결과를 통해 단계별 동작을 확인해보겠습니다.

1단계 초기 메시지 작성

- 가장 먼저 LLM과의 대화를 위한 메시지 스택을 구성합니다.

In [ ]:
```
[{'role': 'system', 'content': 'You are a helpful assistant. Current Date: 2025-04-20
```

2단계 함수 선택/응답 생성을 위한 프롬프트 구성

- 1단계에서 만든 메시지를 포함해서 LLM에게 어떤 도구를 사용할 수 있고, 사용하려면 이런 형식으로 응답해야 한다라는 지침을 만들어 프롬프트화 합니다.
- 특히 Qwen 모델은 아래와 같은 형식을 사용합니다.
- ... 태그를 통해 정의된 함수 목록을 모델에게 전달하고,
- .. 태그를 통해 모델이 함수 호출 방법을 생성하도록 유도 합니다.

In [ ]:
```
<|im_start|>system
You are a helpful assistant. Current Date: 2025-04-20

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "get_kospi_stock_info", "description": "특정
</tools>

For each function call, return a json object with function name and arguments within
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call><|im_end|>
<|im_start|>user
SK텔레콤의 주가를 알려줘<|im_end|>
<|im_start|>assistant
```

3단계 함수 호출/초기 응답을 위한 LLM 응답

- 2단계에서 만든 프롬프트를 Qwen에 입력으로 넣으면 모델이 이 프롬프트 형식을 이해하고 특정 함수를 호출해야겠다고 판단하게 됩니다.

- 그러면 태그 형식으로 호출할 함수 이름과 필요한 인자를 JSON 형식으로 만들어 출력합니다

In [ ]:
```
<tool_call>
{"name": "get_kospi_stock_info", "arguments": {"stock_name_or_code": "SK텔레콤"}}
</tool_call>
```

4단계 함수 호출 내용 파싱 및 메시지 스택에 추가

- 모델의 응답은 태그에 감싸져 있으므로 이를 파싱해야 실제 함수 실행에 사용할 수 있습니다.

- 이 파싱 결과는 messages 리스트에 추가되어 모델과의 대화 흐름에 "어떤 도구를 호출할 예정이다" 라는 맥락을 명확히 추가해줍니다. 이 스택은 나중에 최종 응답 생성을 위한 프롬프트로 활용됩니다.

In [ ]:
```
{'role': 'assistant', 'tool_calls': [{'type': 'function', 'function': {'name': 'get_k
```

5단계 함수 실행 결과

- 모델이 호출하라고 지시한 함수를 실제로 실행합니다. 이때 내부적으로 get_kospi_stock_info("SK텔레콤")을 호출하며, Yahoo Finance API를 통해 실제 주식 정보를 가져옵니다.

In [ ]:
```
{"ticker": "017670.KS", "stock_name": "SKTelecom", "current_price": 57700.0, "previou
```

6단계 최종 응답 생성을 위한 프롬프트 작성

- 지금까지의 모든 대화 흐름(system → user → assistant의 함수 호출 → tool의 응답)을 기반으로, 모델에게 자연어 응답을 요청할 차례입니다. 이때 messages를 그대로 사용해 최종 프롬프트를 구성합니다.

In [ ]:
```
<|im_start|>system
You are a helpful assistant. Current Date: 2025-04-20<|im_end|>
<|im_start|>user
SK텔레콤의 주가를 알려줘<|im_end|>
<|im_start|>assistant
<tool_call>
{"name": "get_kospi_stock_info", "arguments": {"stock_name_or_code": "SK텔레콤"}}
</tool_call><|im_end|>
<|im_start|>user
<tool_response>
{"ticker": "017670.KS", "stock_name": "SKTelecom", "current_price": 57700.0, "previou
</tool_response><|im_end|>
<|im_start|>assistant
```

In [ ]:
```
# 메인 쿼리 처리 함수
def query_kospi_info(query: str) -> str:
    current_date = datetime.now().strftime('%Y-%m-%d')
    messages = [
        {"role": "system", "content": f"You are a helpful assistant. Current Date: {c
        {"role": "user", "content": query}
    ]
    print(f"\n### 1단계 초기 메시지 작성:\n{messages}")

    prompt = tokenizer.apply_chat_template(
        messages, tools=TOOLS, add_generation_prompt=True, tokenize=False
    )
    print(f"\n### 2단계 함수 선택/응답 생성을 위한 프롬프트 구성:\n{prompt}")

    first_output = model.generate([prompt], sampling_params_func_call)[0].outputs[0].
    print(f"\n### 3단계 함수 호출/초기 응답을 위한 LLM 응답:\n{first_output}")

    assistant_msg = parse_tool_calls(first_output)
    messages.append(assistant_msg)
    print(f"\n### 4단계 함수 호출 내용 파싱 및 메시지 추가:\n{assistant_msg}")

    if assistant_msg.get("tool_calls"):
        for call in assistant_msg["tool_calls"]:
            fn = call["function"]["name"]
            args = call["function"]["arguments"]
            if fn == "get_kospi_stock_info":
                result = get_kospi_stock_info(args["stock_name_or_code"])
            else:
                result = json.dumps({"error": "지원하지 않는 함수"}, ensure_ascii=False)
            print(f"\n### 5단계 함수 실행 결과 ({fn}):\n{result}")
            messages.append({
                "role": "tool",
                "tool_call_id": call["id"],
```

```
                "name": fn,
                "content": result
            })

        final_prompt = tokenizer.apply_chat_template(
            messages, add_generation_prompt=True, tokenize=False
        )
        print(f"\n### 6단계 최종 응답 생성을 위한 프롬프트:\n{final_prompt}")

        final_output = model.generate([final_prompt], sampling_params_text)[0].output
        final_response = final_output.strip().rstrip("<|im_end|>")
        print(f"\n### 7단계 최종 LLM 응답 (정리 후):\n{final_response}")
        return final_response
    else:
        print("LLM이 함수를 호출하지 않았습니다. 초기 응답을 반환합니다.")
        content = assistant_msg.get("content", "").strip()
        return content or "응답 내용을 찾을 수 없습니다."
```

실행 및 결과 확인

이제 작성된 코드로 "SK텔레콤의 주가르 알려줘"라는 실제 질문으로 테스트 해보죠.

In [ ]:
```
if __name__ == "__main__":
    queries = [
#         "SK텔레콤의 주가를 알려줘",
        "카카오의 주가 얼마야?"
    ]

    for query in queries:
        print(f"\n========================================")
        print(f" 질문: {query}")
        print(f"========================================")
        response = query_kospi_info(query)
        print(f"\n 답변: {response}")
```

Llama 3.1 Function Calling 예제

In [ ]:
```
import json
import re
import yfinance as yf
from vllm import LLM, SamplingParams
from transformers import AutoTokenizer
from datetime import datetime
```

In [ ]:
```
# 모델 및 토크나이저 초기화
MODEL_ID = "meta-llama/Llama-3.1-8B-Instruct"
model = LLM(MODEL_ID, tensor_parallel_size=1)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
```

In [ ]:
```
# 샘플링 파라미터 설정 - llama3.1 에 맞게 stop token 변경
sampling_params_func_call = SamplingParams(
    max_tokens=256, temperature=0.0, stop=["<|im_end|>"], skip_special_tokens=False
)
sampling_params_text = SamplingParams(
    max_tokens=512, temperature=0.1, top_p=0.95, stop=["<|im_end|>"], skip_special_to
)
```

In [ ]:
```
# KOSPI 주식 정보
KOSPI_TICKER_MAP = {
    "SK텔레콤": "017670.KS", "삼성전자": "005930.KS", "SK하이닉스": "000660.KS",
    "현대차": "005380.KS", "기아": "000270.KS", "LG에너지솔루션": "373220.KS",
    "NAVER": "035420.KS", "카카오": "035720.KS",
}
```

In [ ]:
```python
# 도구(함수) 정의
TOOLS = [{
    "type": "function",
    "function": {
        "name": "get_kospi_stock_info",
        "description": "특정 KOSPI 주식의 현재 가격 및 기본 정보를 가져옵니다.",
        "parameters": {
            "type": "object",
            "properties": {
                "stock_name_or_code": {
                    "type": "string",
                    "description": "주식 이름(예: 'SK텔레콤') 또는 종목 코드(예: '017670')"
                }
            },
            "required": ["stock_name_or_code"]
        }
    }
}]
```

In [ ]:
```python
# 주가 조회
def get_kospi_stock_info(stock_name_or_code: str) -> str:

    name_or_code = stock_name_or_code.strip()
    ticker_symbol = None

    if re.fullmatch(r'\d{6}', name_or_code):
        ticker_symbol = name_or_code + ".KS"
    elif name_or_code in KOSPI_TICKER_MAP:
        ticker_symbol = KOSPI_TICKER_MAP[name_or_code]
    else:
        ticker_symbol = name_or_code + ".KS"

    stock = yf.Ticker(ticker_symbol)
    stock_info = stock.info

    current_price = stock_info.get('currentPrice')
    previous_close = stock_info.get('previousClose')

    price_to_use = current_price if current_price is not None else previous_close
    price_display = round(price_to_use, 2) if price_to_use is not None else "정보 없음"
    previous_close_display = round(previous_close, 2) if previous_close is not None e

    result = {
        "ticker": ticker_symbol,
        "stock_name": stock_info.get('shortName', name_or_code),
        "current_price": price_display,
        "previous_close": previous_close_display,
        "currency": stock_info.get('currency', 'KRW')
    }

    return json.dumps(result, ensure_ascii=False)
```

In [ ]:
```python
# 함수 호출 파싱
# JSON 유효성 검사
def parse_tool_calls(output_text: str):

    cleaned_output = output_text.strip()
    json.loads(cleaned_output)

    return {
        "role": "assistant",
        "content": cleaned_output
    }
```

In [ ]:
```python
# IPython 메시지 생성 함수
def create_ipython_message(tool_result_dict: dict):
    return {
```

```
        "role": "ipython",
        "content": {
            "output": tool_result_dict
        }
    }
```

In [ ]:

```
# 메인 쿼리 처리 함수
def query_kospi_info(query: str):

    current_date = datetime.now().strftime('%Y-%m-%d')
    system_message = f"When you receive a tool call response, use the output to forma

    messages = [
        {"role": "system", "content": system_message},
        {"role": "user", "content": query}
    ]
    print(f"\n### 1단계 초기 메시지 작성:\n{messages}")

    prompt = tokenizer.apply_chat_template(messages, tools=TOOLS, add_generation_prom
    cleaned_prompt = "\n".join(
        line for line in prompt.splitlines() if "Environment: ipython" not in line
    )
    print(f"\n### 2단계 함수 선택/응답 생성을 위한 프롬프트 구성:\n{cleaned_prompt}")

    outputs = model.generate([cleaned_prompt], sampling_params=sampling_params_func_c
    func_call_response_text = outputs[0].outputs[0].text
    print(f"\n### 3단계 함수 호출/초기 응답을 위한 LLM 응답:\n{func_call_response_text}")

    assistant_msg = parse_tool_calls(func_call_response_text)
    messages.append(assistant_msg)
    print(f"\n### 4단계 어시스턴트 함수 호출 메시지 추가 후:\n{messages}")

#    tool_call_content = assistant_msg['content']
    tool_call_content = func_call_response_text
    print(f"$$$ check: {tool_call_content}")
    python_tag = "<|python_tag|>"
    if tool_call_content.startswith(python_tag):
        tool_call_content = tool_call_content[len(python_tag):].strip()

    tool_data = json.loads(tool_call_content)
    tool_params = tool_data.get("parameters")

    if tool_params is None or "stock_name_or_code" not in tool_params:
         raise ValueError("LLM response did not contain valid 'parameters' or 'stock_

    stock_name_or_code = tool_params["stock_name_or_code"]
    function_result_json_str = get_kospi_stock_info(stock_name_or_code)

    function_result_dict = json.loads(function_result_json_str)
    print(f"\n### 5단계 함수 실행 결과:\n{function_result_dict}")

    ipython_msg = create_ipython_message(function_result_dict)
    messages.append(ipython_msg)
    print(f"\n### 5.1단계 Tool 결과('ipython' role) 메시지 추가 후:\n{messages}")

    final_prompt = tokenizer.apply_chat_template(messages, tools=TOOLS, add_generatio

    print(f"\n### 6단계 최종 응답 생성을 위한 프롬프트:\n{final_prompt}")

    outputs = model.generate([final_prompt], sampling_params=sampling_params_text)
    final_response = outputs[0].outputs[0].text

    return final_response
```

In [ ]:

```
if __name__ == "__main__":
    queries = [
        "Tell me the stock price of SK텔레콤",
#        "현대차 주가 알려줘"
```

```
    ]

    for query in queries:
        print(f"\n========================================")
        print(f" 질문: {query}")
        print(f"========================================")
        response = query_kospi_info(query)
        print(f"\n 답변: {response}")
```

Gemma3 Function Calling 예제

In [ ]:
```
import json
import re
import yfinance as yf
from vllm import LLM, SamplingParams
from transformers import AutoTokenizer
from datetime import datetime
```

In [ ]:
```
# 모델 및 토크나이저 초기화
MODEL_ID = "google/gemma-3-12b-it"
model = LLM(MODEL_ID, tensor_parallel_size=1)
tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
```

In [ ]:
```
# 샘플링 파라미터 설정 - Gemma 3에 맞게 stop token 변경
sampling_params_func_call = SamplingParams(
    max_tokens=256, temperature=0.0, stop=["<end_of_turn>"], skip_special_tokens=Fals
)
sampling_params_text = SamplingParams(
    max_tokens=512, temperature=0.1, top_p=0.95, stop=["<end_of_turn>"], skip_special
)
```

In [ ]:
```
# KOSPI 주식 정보
KOSPI_TICKER_MAP = {
    "SK텔레콤": "017670.KS", "삼성전자": "005930.KS", "SK하이닉스": "000660.KS",
    "현대차": "005380.KS", "기아": "000270.KS", "LG에너지솔루션": "373220.KS",
    "NAVER": "035420.KS", "카카오": "035720.KS",
}
```

In [ ]:
```
# 도구(함수) 정의
TOOLS = [{
    "type": "function",
    "function": {
        "name": "get_kospi_stock_info",
        "description": "특정 KOSPI 주식의 현재 가격 및 기본 정보를 가져옵니다.",
        "parameters": {
            "type": "object",
            "properties": {
                "stock_name_or_code": {
                    "type": "string",
                    "description": "주식 이름(예: 'SK텔레콤') 또는 종목 코드(예: '017670')"
                }
            },
            "required": ["stock_name_or_code"]
        }
    }
}]
```

In [ ]:
```
# 도구(함수) 정의
TOOLS = [{
    "type": "function",
    "function": {
        "name": "get_kospi_stock_info",
        "description": "특정 KOSPI 주식의 현재 가격 및 기본 정보를 가져옵니다.",
        "parameters": {
            "type": "object",
```

```
            "properties": {
                "stock_name_or_code": {
                    "type": "string",
                    "description": "주식 이름(예: 'SK텔레콤') 또는 종목 코드(예: '017670')"
                }
            },
            "required": ["stock_name_or_code"]
        }
    }
}]
```

In [ ]:
```
# 함수 호출 파싱
def parse_tool_calls(output_text: str):

    call = re.search(r'\[(\w+\(.*?\))\]', output_text)
    if not call:
        print("함수 호출 문자열 '[]'을 찾을 수 없음")
        return None, None

    func_match = re.match(r'(\w+)\((.*?)\)', call.group(1))
    if not func_match:
        print("함수 형식이 올바르지 않음")
        return None, None

    func_name, params_str = func_match.groups()

    params = dict(re.findall(r"(\w+)='([^']*)'", params_str))
    return func_name, params
```

In [ ]:
```
# 함수 호출 파싱
def parse_tool_calls(output_text: str):

    call = re.search(r'\[(\w+\(.*?\))\]', output_text)
    if not call:
        print("함수 호출 문자열 '[]'을 찾을 수 없음")
        return None, None

    func_match = re.match(r'(\w+)\((.*?)\)', call.group(1))
    if not func_match:
        print("함수 형식이 올바르지 않음")
        return None, None

    func_name, params_str = func_match.groups()

    params = dict(re.findall(r"(\w+)='([^']*)'", params_str))
    return func_name, params
```

In [ ]:
```
if __name__ == "__main__":
    queries = [
        "SK텔레콤의 주가를 알려줘",
#         "삼성전자 현재가 얼마야?",
    ]

    for query in queries:
        print(f"\n========================================")
        print(f" 질문: {query}")
        print(f"========================================")
        response = query_kospi_info(query)
        print(f"\n 답변: {response}")
```