AI Agent 과정

Function Calling & Structured Output

목차 Function Calling & Structured Output

1. LLM의 Function Calling 원리 이해

Function Calling 이란?

Function Calling

- 사용자의 자연어 입력에서 API 호출에 필요한 인자들을 추출하는 기능
- 미리 정의된 외부 Tool 중에서 어떤 함수와 인자(arguments)로 API 호출해야 하는지 파악
- 구조화된 데이터(JSON) 형식으로 추출

주의:

- LLM은 함수를 직접 실행(execute)하지 않음
- LLM의 역할은 사용자의 의도를 파악하여 함수 호출 명세서를 생성하는 것까지
- 생성된 명세서를 받아 실제 함수를 실행하는 주체는 애플리케이션 코드(Agent Executor)
- Agent가 수행할 최종 행동에 대한 통제권은 항상 개발자의 코드에 있음

Function Calling 이란?

Function Calling:

- 의도 분석 (Intent Classification):
  - ("오늘 서울 날씨 어때?") 정보 질의인지, 특정 기능을 요구하는지 판단
  - 의도에 가장 부합하는 함수 결정 (get_current_weather 선택)
- 인자 추출 (Argument Extraction):
  - 호출 결정된 함수 실행에 필요한 입력값 파라미터를 사용자 질문에서 추출
  - location="서울", date="오늘" 이라는 인자로 정확히 매핑
- 구조화된 출력 생성 (Structured Output Generation):
  - 일반적인 텍스트 답변 대신, 호출할 함수의 이름과 추출된 인자를 담은 JSON 형식 문자열을 리턴

```
{"tool_calls": [{
    "function": {
      "name": "get_current_weather",
      "arguments": "{ \"location\": \"서울\", \"date\": \"오늘\" }"}}]}
```

Function Calling 2-Step Workflow

Function Calling 2-Step:

- Step 1: LLM 제안 (Suggestion)
  - 질문 ("서울 날씨 알려줘")
  - Agent는 등록된 Tool Schema와 질문 모두 LLM에게 전달
  - LLM은 질문 분석 후, 호출할 함수와 해당 함수에 쓸 파라미터를 결정
  - 함수와 파라미터 정보를 담은 JSON 객체를 Agent에게 반환

Function Calling 2-Step Workflow

Function Calling 2-Step:

- Step 2: 코드 실행 (Execution)
  - LLM이 반환한 JSON을 Agent가 파싱
  - JSON에 명시된 get_current_weather 함수를 location="서울" date="오늘" 이라는 파라미터와 함께 실행 (날씨 API 서버와 통신)
  - ("오늘 서울의 날씨는 맑고, 기온은 25도 입니다.")
  - LLM에게 함수 실행 결과를 전달하며 최종 답변 요청

Gemini vs OpenAI vs LangChain

Google Gemini (Native SDK):

- FunctionDeclaration 객체를 사용하는 명시적이고 구조적인 방식

OpenAI (Native SDK):

- JSON 딕셔너리 형태로 Tool 명세를 직접 정의하는 방식 (업계 표준의 시초)

LangChain (Abstraction Framework):

- @tool 데코레이터와 .bind_tools()를 사용하여 과정을 추상화하고 간소화한 방식

Google Gemini Native SDK

Gemini:

- 명시적인 객체 지향 선언
- FunctionDeclaration이라는 클래스 사용
- Tool의 명세를 코드 레벨에서 정의
- 코드 자동 완성, 타입 체킹 등 개발자 도구의 지원을 받기 용이하며, 구조가 명확

```python
from google.generativeai.types import Tool, FunctionDeclaration

get_current_weather_func = FunctionDeclaration(
    name="get_current_weather",
    description="특정 도시의 현재 날씨 정보를 가져옵니다.",
    parameters={
        "type": "object",
        "properties": {
            "location": {"type": "string", "description": "날씨를 조회할 도시의 영어 이름"},
        },
        "required": ["location"]
    }
)

# Tool 객체로 묶어서 모델에 전달
agent_tools = Tool(function_declarations=[get_current_weather_func])
```

OpenAI Native SDK

OpenAI:

- 유연한 JSON 딕셔너리 선언
- Tool의 명세를 Python 객체가 아닌, JSON Schema 형식의 딕셔너리로 직접 작성
- REST API와 직접 통신하는 환경 등에서 더 유연하게 스키마를 동적으로 생성할 수 있음
- Gemini의 FunctionDeclaration과 구조적으로 동일

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_current_weather",
            "description": "특정 도시의 현재 날씨 정보를 가져옵니다.",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {"type": "string", "description": "날씨를 조회할 도시의 영어 이름"},
                },
                "required": ["location"],
            },
        },
    }
]
```

스키마 설명(Description)의 중요성

Tool 성능의 90%는 설명:

- LLM은 오직 Tool과 파라미터의 description을 읽고 그 기능을 파악
- LLM의 Tool 선택 성능을 높이려면 Tool 기능을 세분화하여 여러개로 분할해야함
- 나쁜 설명의 예:
  - description="데이터를 찾는다." (-> 무슨 데이터?)
  - parameter="search_term" (-> 무엇을 검색하는 용어인가?)
- 좋은 설명의 예:
  - description="arXiv.org에서 특정 키워드가 포함된 최신 AI 관련 논문을 검색합니다.“
  - parameter="query: str, description="논문 검색에 사용할 핵심 키워드 ('Large Language Model’)
- Agent가 Tool을 제대로 사용하지 않는다면, 가장 먼저 Tool의 설명을 의심하고 개선해야 함

병렬 호출(Parallel Calling)

최신 LLM(Gemini 2.5 Pro, GPT-o3 등)은 동시에 여러 개의 독립적인 Tool 호출 가능

- 여러 API 요청을 병렬로 처리할 수 있어, 전체 응답 시간을 크게 단축시킬 수 있음
- 애플리케이션 단에서 tool_calls 리스트를 순회하며 모든 함수를 실행
- 그 결과들을 모아 2단계 호출에 전달
- 사용자 프롬프트: "서울 날씨랑 삼성전자 주가 둘 다 알려줘

```
"tool_calls": [
  {"name": "get_current_weather", "args": {"location": "Seoul"}},
  {"name": "get_stock_price", "args": {"symbol": "005930.KS"}}
]
```

보안 및 안정성 (Guardrails)

Agent 안전장치:

- 사용자 승인 (Human-in-the-Loop):
  - 파일 삭제, 데이터베이스 수정 등 파괴적이거나 되돌릴 수 없는 행동을 하는 Tool에게
  - LLM이 해당 Tool 호출을 제안하면, 즉시 실행하지 않게 함
  - 사용자에게 " 정말로 이 작업을 실행하시겠습니까? (y/n) " 라고 확인을 받게끔 설계
  - LangGraph의 interrupt 기능을 사용하면 쉽게 구현 가능

보안 및 안정성 (Guardrails)

Agent 안전장치

- 입력값 검증 (Input Validation):
  - Tool 함수 내부에서, LLM이 넘겨준 인자값이 유효한지 검증하는 로직을 추가
  - 숫자가 들어와야 할 곳에 문자가 오는 상황 등 방지
- 권한 제어 (Permission Control):
  - 사용자의 등급이나 역할에 따라 사용할 수 있는 Tool을 제한
  - 일반 사용자는 데이터 조회(GET) Tool만, 관리자는 데이터 수정(POST) Tool까지

2. Structured Output을 통한 출력 형식 관리

LLM 응답의 불안정성

자유로운 LLM 응답의 문제점:

- “사용자 리뷰에서 제품명과 평점을 추출해줘.”
  - 응답 A: 제품명은 ' 갤럭시 Z 폴드 6 ' 이고, 평점은 5점입니다
  - 응답 B: 사용자는 갤럭시 Z 폴드 6에 대해 별 5개를 주었습니다
  - 응답 C: { " product " : " Galaxy Z Fold 6 " , " rating " : 5}
  - 응답 D: 평점: ★★★★★, 제품: 갤럭시 Z 폴드 6

- 위 응답들을 처리하기 위해 수많은 파싱(Parsing) 규칙과 예외 처리 코드 필요
- LLM의 응답 형식이 조금만 바뀌어도 시스템 전체가 멈출 수 있는 '깨지기 쉬운(Fragile)' 구조

Structured Output

Structured Output(구조화된 출력):

- 개발자가 미리 정의한 데이터 구조(JSON)를 따르도록 강제하는 기술
- 출력 형식을 통제하여 예측 가능하고 코드가 즉시 처리할 수 있게 함
- Pydantic과 Instructor를 사용하여 구현

```
{
  "product": "Galaxy Z Fold 6",
  "rating": 5,
  "review_summary": "화면 크기와 멀티태스킹 기능이 뛰어남",
  "sentiment": "positive",
  "recommend": true
}
```

데이터 설계도

Pydantic:

- Python의 타입 힌트(Type Hint)를 사용
- 데이터의 유효성을 검사하고 설정을 관리
- LLM으로부터 받고 싶은 데이터 구조의 설계도를 Python 클래스로 정의
- 데이터 스키마 정의:
  - 어떤 데이터 필드가(name, age), 어떤 타입으로(str, int) 존재해야 하는지 정의
- 데이터 유효성 검사 (Validation):
  - LLM이 생성한 데이터가 우리가 정의한 타입과 규칙에 맞는지 자동으로 검증

Pydantic 모델 정의

BaseModel을 상속받고, 클래스 변수에 타입 힌트를 사용하여 데이터 구조 정의

Field를 사용하여 각 필드에 대한 상세 설명 추가 가능

```python
from pydantic import BaseModel, Field
from typing import List

class UserInfo(BaseModel):
    # Field의 description은 LLM이 이 필드의 의미를 파악하는 데 결정적인 역할을 합니다.
    name: str = Field(description="사용자의 전체 이름")
    age: int = Field(description="사용자의 나이")
    email: str = Field(description="사용자의 이메일 주소")
    interests: List[str] = Field(description="사용자의 관심사 목록")
```

- name(문자열), age(정수), email(문자열), interests(문자열 리스트)
- 4개의 필드를 가진 객체를 만들어줘 라고 지시하는 것과 같음

LLM과 Pydantic의 통역

Instructor:

- LLM을 Pydantic에 맞게 패치(patch)
- LLM이 Pydantic 모델을 직접 이해하고 그에 맞는 출력을 생성하도록 만듦
- 자동 프롬프트 엔지니어링:
  - LLM에게 JSON 형식으로 응답하도록 유도하는 시스템 프롬프트를 내부적으로 자동 생성
- 출력 파싱 및 유효성 검사:
  - LLM이 생성한 텍스트(JSON String)를 파싱하여 Pydantic 객체로 변환 및 유효성 검사
- 자동 재시도 (Self-Correction):
  - 만약 LLM의 출력이 유효하지 않은 경우(예: JSON 문법 오류, 타입 불일치)
  - 내부적으로 오류를 수정하여 다시 LLM에게 요청하는 재시도 로직 포함

Pydantic 객체는 왜 String보다 월등한가?

Instructor를 통해 Pydantic 객체를 반환받는 것의 장점:

- 타입 안정성 및 자동 완성 (Type Safety & Autocomplete):
  - Before (Dict): user_info['naem'] -> 오타를 찾아주지 못하고 런타임 에러 발생.
  - After (Pydantic): user_info.naem -> IDE가 즉시 오타를 지적하고, user_info. 입력 시 name, age 등 사용 가능한 속성을 추천

Pydantic 객체는 왜 String보다 월등한가?

Instructor를 통해 Pydantic 객체를 반환받는 것의 장점:

- 자동 유효성 검사 (Automatic Validation):
  - LLM이 age에 "서른살"이라는 텍스트 반환 시, instructo가 이를 int 타입 위반으로 감지
  - 자동으로 재시도를 요청하거나, ValidationError를 발생
- 쉬운 데이터 변환 (Serialization):
  - .model_dump_json(): 객체를 즉시 JSON 문자열로 변환하여 API 응답으로 보낼 수 있음
  - .model_dump(): 객체를 Python 딕셔너리로 변환하여 다른 라이브러리와 연동 가능

프롬프트와 Pydantic 모델 설계의 중요성

Pydantic 필드 설명은 프롬프트:

- LLM은 오직 Field(description="...")만 보고 그 필드 의미를 파악
- 최대한 명확하고 상세해야 함
- (예: description="사용자의 전체 이름 (성 제외)" vs description="사용자의 이름")

복잡성은 LLM의 적:

- Pydantic 모델이 깊게 중첩되거나(nested) 구조가 복잡해지면, LLM이 올바른 형식으로 출력 실패
- 모델을 여러 개의 단순한 모델로 분리, 구조를 평탄하게(flat) 설계

Few-shot 예제 제공:

- 복잡한 구조를 추출해야 할 경우, 프롬프트에 한두 개의 완성된 JSON 예시를 함께 제공

예외 처리의 중요성

Instructor의 자동 재시도 기능이 강력하긴 하지만,

LLM이 끝내 유효한 데이터를 생성하지 못하는 경우에 대비:

- try...except 블록으로 감싸기:
  - instructor 클라이언트 호출은 항상 try...except pydantic.ValidationError 블록으로 감싸서 유효성 검사 실패에 대응
- 실패 시 대응 전략 (Fallback Logic):
  - 복잡한 UserInfo 모델에 실패했다면, name 필드 하나만 있는 더 단순한 모델로 다시 요청
  - 사용자에게 추가 정보를 요청하는 로직 추가
  - 실패한 입력과 출력을 로그로 기록하여 추후 모델이나 프롬프트를 개선하는 데 사용