Agent

LLM 기반 Agents

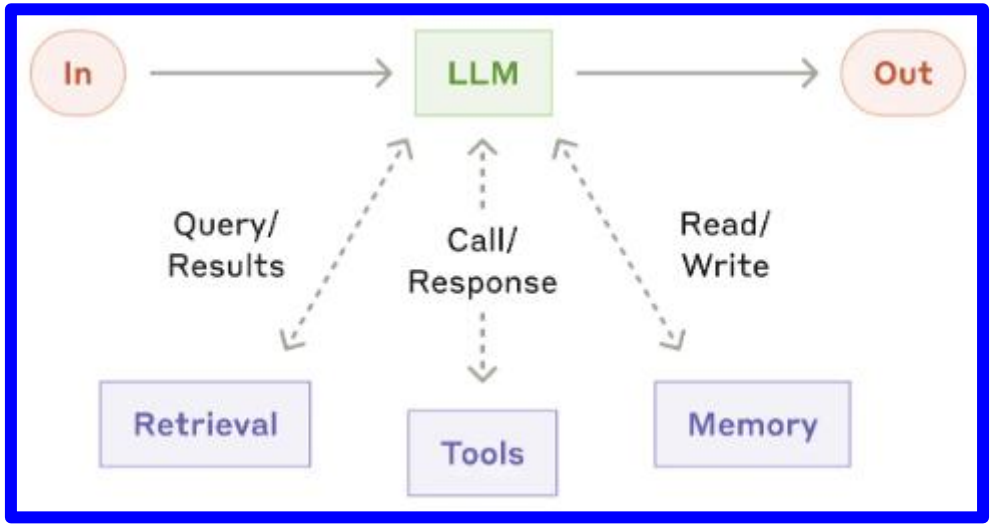
Agenda

0. 개요 – 개념 및 실습
1. Tool Use → Function Calling
2. Retrieval → RAG (Retrieval Augmented Generation)
3. Reasoning → ReAct/Plan-Execute
4. Memory 설계
5. Orchestration → LangGraph 이용
6. MCP (Model Context Protocol)
7. 기말 프로젝트

ReAct (Reason + Act) 기반의 순환(Loop) 구조

ReAct Loop

1. **Thought**: LLM이 현재 목표와 메모리(이전기록)을 바탕으로 상황 분석하고 다음으로 수행할 행동(Action)을 추론
2. **Action**: LLM이 추론한 Thought에 따라 특정 도구(Tool)를 선택하고 실행
3. **Observation**: 실행된 도구로부터 결과를 입력 받음. 이 결과는 Short-term Memory에 기록
4. **Repeat**: Agent는 이 Observation을 바탕으로 다시 새로운 Thought를 시작하며, 목표 달성 또는 최종 답변 생성까지 이 루프를 반복



어떠한 LLM이 적합할까?

구분	OpenAI	Claude 3.5	Gemini
문법 단순성	가장 간단	중간	복잡
Function Calling	안정적	자연스럽고 확장성 있음	지원 복잡
Reasoning 성능	중상	최상급	상
Multimodal	제한적	텍스트 중심	강력(이미지/오디오)
RAG 적합성	우수	긴 텍스트 수용 우수	플랫폼 통합형
학습 편의성	100%	100%	어려움

- **OpenAI는 개발자 경험(DX) 중시, API 이용 개발 편의성 높음**
- **Claude (Anthropic)은 헌법형 AI 정책에 기반한 안전 중시, 추론 기능 우수, 텍스트 위주.**
- **Gemini는 엔터프라이즈 통합형, 텍스트+이미지+오디오 질의 용이, 학습 진입 장벽 있음.**

API를 이용한 LLM 질의 간단 예제

```
from openai import OpenAI
client = OpenAI()
resp = client.chat.completions.create(
    model="gpt-5-mini",
    messages=[{"role":"user","content":"What is the capital of France?"}]
)
print(resp.choices[0].message.content)
```

- 모델은 stateless, 즉 대화 내용을 자체적으로 유지하지 않음

Function Calling 원리

- **Function Calling:** LLM이 문자열을 함수 호출 형식(JSON)"으로 변환
- LLM이 코드를 실행하는 것이 아니라, 함수를 어떻게 호출할 지를 예측하고, 실행은 호출자(Agent)가 담당한다.
- 출력 예:
  FunctionCall(arguments='{"city":"서울"}', name='get_weather')
- get_weather 함수는 agent가 마련해야 하며, arguments에서 알 수 있다.

```
client = OpenAI()

functions = [{
    "name": "get_weather",
    "description": "도시의 현재 날씨",
    "parameters": {
        "type": "object",
        "properties": {"city": {"type": "string"}},
        "required": ["city"]
    }
}]

resp = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role":"user","content":"서울의 날씨 알려줘"}],
    functions=functions,
    function_call="auto"
)
print(resp.choices[0].message.function_call)
============================================================
FunctionCall(arguments='{"city":"서울"}', name='get_weather')
```

RAG (Retrieval Augmented Generation)

- LLM 모델이 외부 문서를 검색하여 정보를 프롬프트에 주입
- 구조: Query → Retriever → Context (Prompt) → Generator
- 아래 예제는 corpus에서 query와 관련된 문장을 retrieve 하는 간단한 예제이다.

```
from sentence_transformers import SentenceTransformer
import numpy as np


model = SentenceTransformer("all-MiniLM-L6-v2")
corpus = ["파리는 프랑스의 수도다.", "도쿄는 일본의 수도다."]
emb = model.encode(corpus, normalize_embeddings=True)
query = model.encode(["프랑스의 수도는?"], normalize_embeddings=True)
sims = np.dot(query, emb.T)
print(corpus[np.argmax(sims)])  # 파리는 프랑스의 수도다.
```

Memory 개념

- **단기 메모리**: 현재 세션의 대화 요약
- **장기 메모리**: 과거 대화/사용자 프로필 벡터화 저장

```
from collections import deque
buffer = deque(maxlen=5)

def add(role, msg):
    buffer.append((role, msg))
    return "\n".join(f"{r}: {m}" for r,m in buffer)

for i in range(3):
    print(add("user", f"message {i}"))
```

ReAct / Reasoning 개요

- **ReAct = Reason + Act**
- Thought: 무엇을 해야 하는가?
- Action: 도구 실행
- Observation: 결과 확인
- Thought 반복

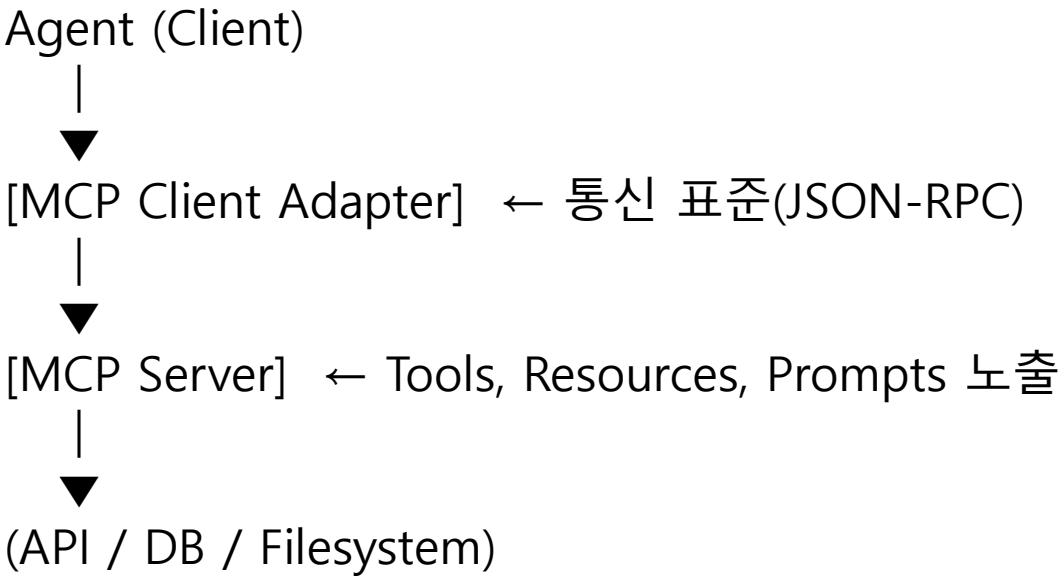
- **LangGraph 노드 / 에지 구조로 발전**

```
for step in range(3):
    thought = f"생각 {step}: 사용자 의도 분석"
    if "날씨" in question:
        action = "weather_tool"
        obs = weather_tool("서울")
    else:
        action = "none"
        obs = "대화만 진행"
    print(thought, action, obs)
```

MCP (Model Context Protocol) 개념

- **Agent가 도구를 직접 연결 → 중복 / 비표준 문제 발생**
- **도구와 리소스 연결을 표준화한 프로토콜 계층**

- **MCP Server: 툴/리소스를 노출하는 표준 엔드포인트**
- **MCP Client: Agent에서 여러 MCP 서버를 호출**
- **LangGraph는 이를 위한 공식 어댑터(langchain-mcp-adapters) 지원**



구현 전략

- 초기 학습-핵심 로직은 순수 Python으로 구현
- 그래프형 오케스트레이션(분기/재시도/Human-In-The-Loop)이 필요해지는 시점에서 LangGraph(+MCP 어댑터) 활용

① 순수 Python으로 정확도/성능 지표 확보

② 플로우가 커지면 LangGraph로 그래프화

③ 사내/로컬/클라우드 리소스 표준화가 필요해지면 MCP(+ Adapter) 도입

④ LangChain은 Tool/Retriever-Adapter 등 선택적 도입