

# agent ai

tool usage가 활성화 되어있어야함.

아니라면 단순 프롬프트 체이닝() 혹은 수동 툴-콜링(manual tool-use) 패턴 사용

| LangChain 기능              | Tool Usage 필요 여부 | 설명          |
|---------------------------|------------------|-------------|
| LLMChain , PromptTemplate | ✗                | 단순 프롬프트 체이닝 |
| RetrievalQA , Vectorstore | ✗                | 문서 검색형 QA   |
| Agent (ZeroShot, ReAct 등) | ✓                | 외부 도구 호출 필요 |
| Tool , StructuredTool     | ✓                | 함수/모듈 호출용   |

[사용자]

↓

[모델 A (OpenRouter free 모델)]

→ "다음 작업들을 JSON 리스트로 반환" (예: 어떤 툴을 써야 하는지)

↓

[너의 Python 코드]

→ JSON 파싱 → 그대로 실행 or 전달

↓

[모델 B 또는 외부 API 호출]

```
system_prompt = """
```

너는 도구 실행 계획만 세우는 **assistant**이다.

결과를 반드시 아래 JSON 형식으로만 출력하라.

형식:

```
{
```

```
    "actions": [
        {"tool": "summarize_schedule", "args": {"range": "tomorrow"}},
        {"tool": "send_kakao_message", "args": {"recipient": "me", "content": "회의
알림"}}
    ]
}
```

```
"""
```

```
user_prompt = "내 일정 정리하고, 카톡으로 내일 회의 알림 보내줘"
```

-

```

{
  "actions": [
    {"tool": "summarize_schedule", "args": {"range": "tomorrow"}},
    {"tool": "send_kakao_message", "args": {"recipient": "me", "content": "회의
알림"}}
  ]
}

```

-

```

reply = res.choices[0].message.content
actions = json.loads(reply)[ "actions"]

for act in actions:
  if act["tool"] == "summarize_schedule":
    result = summarize_schedule(**act[ "args"])
  elif act["tool"] == "send_kakao_message":
    send_kakao_message(**act[ "args"])

```

tool usage가 활성화 되어있어있더라도 n8n langchain langgraph 같은 걸 쓰지 않는다면 수동으로 위와같이 만들어 줘야함.

## LLM

간단히 말하면 다음 단어(토큰) 맞추기

예를 들어 문장이 이렇게 있다고 해 보자:

| 오늘 날씨가 너무 좋아서 공원에 ...

여기서 ... 자리에 뭐가 올지 맞추는 모델

- “갔다”
- “산책하러”
- “나왔다”
- “가고 싶다”

이런 후보들 중에,

**“이 앞의 문맥을 봤을 때, 어떤 단어가 가장 자연스러울까?”**

→ 그 확률을 계산해서 제일 그럴듯한 걸 고르는 거지.

| 이 “다음 단어 맞추기”를 잘하는 모델 = LLM

컴퓨터는 글자를 직접 이해 못 하니까 숫자로 바꿔야 해.

## 1. 토큰(token)

- 문장을 잘게 쪼갠 단위
- 단어일 수도 있고, 단어 조각(subword)일 수도 있음
- 예:
  - “안녕하세요” → [ “안”, “녕”, “하세요” ] 같은 식으로 쪼개지기도 하고
  - “computer” → [ “com”, “put”, “er” ] 처럼 쪼개지기도 함

## 2. 숫자로 매핑 (토큰 ID)

- 각 토큰마다 번호를 붙임
- 예: “안” → 1052, “녕” → 9843 이런 식

## 3. 임베딩(embedding)

- 토큰 ID 하나를 긴 벡터(예: 길이 4096짜리 실수 배열)로 바꿈
- 예:
  - “고양이” → [0.12, -0.8, 0.03, ...]
  - “강아지” → [0.11, -0.79, 0.02, ...]
- 비슷한 의미의 단어는 비슷한 방향의 벡터를 가지도록 학습됨

여기까지:

문장 → 토큰 → 숫자 ID → 고차원 벡터(임베딩)

이제 이 벡터들을 보고 “다음 단어 확률”을 계산해야 하는데,  
여기서 쓰이는 구조가 **트랜스포머(Transformer)**야.

핵심 아이디어는 **Attention(어텐션)** 하나로 정리할 수 있어.

## 🔍 Attention: “어디를 얼마나 볼지” 정하는 메커니즘

예를 들어 문장:

어제 친구랑 고기를 먹었는데, 너무 맛있어서 또 가고 싶다.

여기서 “맛있어서”라는 단어를 이해하려면:

- 앞에 나온 “고기”, “먹었는데” 같은 단어들이 중요하지?
- “어제”, “친구랑”은 상대적으로 덜 중요하고.

Attention은 “현재 위치에서 문장 전체를 훑어보면서,  
어떤 단어에 집중할지 가중치를 주는 연산”이야.

느낌적으로:

- “맛있어서” → (고기: 0.7, 먹었는데: 0.6, 어제: 0.2, 친구랑: 0.3 ...)

이렇게 문장 안의 단어들끼리 서로를 바라보면서 의미를 주고받는 구조가 트랜스포머의 핵심.

트랜스포머는 이런 블록을 수십 개, 수백 개 쌓아서:

- 단어 간 관계
  - 문장 구조
  - 긴 맥락 정보
- 를 점점 더 풍부하게 섞음

## 4. 학습: 엄청나게 많이 “맞추기 문제”를 풀어본 상태

LLM이 “똑똑한 척” 할 수 있는 이유는 단 하나야.

인터넷 텍스트로 만든 거대한 코퍼스에서  
‘다음 단어 맞추기’ 문제를 기가 막히게 많이 풀어봤기 때문.

학습 과정 아주 단순화하면:

1. 랜덤한 파라미터(가중치)로 시작
2. 문장 하나 뽑음:
  - “나는 오늘 학교에 갔다.”
3. 앞에서부터 토큰 잘라가면서:
  - “나는” → 다음 단어 “오늘” 맞추기
  - “나는 오늘” → 다음 단어 “학교에” 맞추기
  - “나는 오늘 학교에” → “갔다” 맞추기
4. 매번 모델이 예측한 확률 분포 vs 실제 정답 비교해서
  - 오차(loss)를 계산
5. 그 오차를 줄이도록 가중치(파라미터)를 조금씩 수정
  - 이게 역전파(backpropagation) + 경사 하강법(gradient descent)

이걸:

- 수십억~수조 개 토큰에 대해
- 수천~수만 번 반복

→ 그러다 보면 모델 파라미터(수십억~수천억 개)가  
“언어의 통계 패턴”을 몸에 익히게 되는 거지.

## 5. 추론(inference): 지금 우리가 대화할 때 벌어지는 일

우리가 지금 대화하는 시점에는 학습은 이미 끝난 상태고,  
모델은 이렇게 동작해.

1. 너가 보낸 메시지 전체를 토큰화
2. 트랜스포머에 넣고 “다음 토큰 확률 분포”를 계산
3. 그 중에서
  - 가장 확률 높은 토큰을 고르거나
  - 샘플링(temperature, top-p, top-k 등)으로 살짝 랜덤 섞기도 하고
4. 그 토큰을 문장 뒤에 붙임
5. 새로 확장된 문장을 다시 넣고, 또 다음 토큰 예측
6. 이걸 멈출 때까지 반복 → 문장 한 줄 생성

즉, 한 글자 한 글자(정확히는 한 토큰씩) 찍어내는 방식이야.

## 6. “이해한다”는 말의 진짜 의미

LLM이 사람처럼 생각한다기보단, 솔직히 말하면:

“패턴 매칭 + 통계적 직감”

이게 쌓여서

- 논리적으로 말하는 것처럼 보이고
- 수학 증명을 쓰고
- 코드도 짜고
- 너의 의도도 어느 정도 파악해.

근데 주의할 점:

- 진짜 세계 모델(물리적으로 세상이 어떻게 돌아가는지에 대한 명시적 시뮬레이터)이 아니라
- 학습 데이터에서 본 텍스트의 패턴 기반이란 점에서 한계가 있음
  - 헛소리(환각)
  - 숫자 계산 실수
  - 사실관계 틀리기도 함

## 7. 요약하면

한 줄로 요약하면 LLM은:

“많은 양의 텍스트를 먹고,  
‘다음에 올 단어를 맞추는’ 일을 반복 학습해서  
인간처럼 말하는 흉내내는 함수”

조금 더 기술적으로는:

“토큰 시퀀스를 입력받아,  
다음 토큰의 확률 분포를 출력하는 거대한 트랜스포머 신경망”

# attention

다음 단어 예측 Loss를 최소화하는 방향으로 학습됨.

문장 예시:

| 나는 어제 친구랑 고기를 먹었다.

여기서 “먹었다”라는 단어가 “고기”에 중요한 attention을 두는 건  
사람은 직관적으로 알지만  
모델은 이걸 스스로 “숫자 점수”로 만들도록 학습한다.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class TinySelfAttention(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.d_model = d_model

        # Q, K, V: 각각 (d_model → d_model)
        self.Wq = nn.Linear(d_model, d_model)
        self.Wk = nn.Linear(d_model, d_model)
        self.Wv = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x shape = (N, d_model)
        Q = self.Wq(x)    # shape: (N, d_model)
        K = self.Wk(x)    # shape: (N, d_model)
        V = self.Wv(x)    # shape: (N, d_model)

        # Attention = softmax(QKT / sqrt(d))
        attn_scores = (Q @ K.transpose(0, 1)) / (self.d_model ** 0.5)
        attn_probs = F.softmax(attn_scores, dim=-1)  # (N, N)

        # Weighted sum
        out = attn_probs @ V  # (N, d_model)
        return out

class TinyTransformerBlock(nn.Module):
    def __init__(self, d_model, hidden=1024):
        super().__init__()
        self.attn = TinySelfAttention(d_model)
        self.ffn = nn.Sequential(
```

```

        nn.Linear(d_model, hidden),
        nn.ReLU(),
        nn.Linear(hidden, d_model)
    )

    def forward(self, x):
        # x: (N, d_model)
        x = x + self.attn(x)      # residual
        x = x + self.ffn(x)       # residual
        return x

# -----
# 사용 예시
# -----
d_model = 32
model = TinyTransformerBlock(d_model)

# 10개의 토큰 (N=10)
x1 = torch.randn(10, d_model)
out1 = model(x1)
print(out1.shape)    # torch.Size([10, 32])

# 150개의 토큰 (N=150)
x2 = torch.randn(150, d_model)
out2 = model(x2)
print(out2.shape)    # torch.Size([150, 32])

```

## 2. Attention 점수는 어떻게 만들어짐

토큰마다 임베딩이 있지?

예:

- “나는” → 벡터 A
  - “고기” → 벡터 B
  - “먹었다” → 벡터 C
- 이런 식.

Attention은 이 벡터들에서  
3개의 새로운 벡터를 만든다:

- Query(q)
- Key(k)
- Value(v)

이 세 가지는 전부 학습 가능한 선형변환(**weight matrix**)로 만든다:

```
q = Wq * embedding
k = Wk * embedding
v = Wv * embedding
```

즉:

**Attention에서 중요한 건 Query와 Key의 관계 (q·k 점수)**

- Query: 지금 보고 있는 단어가 “무엇을 찾으려고 하는지”
- Key: 각 단어가 “나는 이런 정보를 가지고 있어요”라고 내놓는 명함 같은 것

q와 k를 내적하면 **두 단어의 관련성 점수 (similarity)**가 나와.

```
score = qT * k
```

이걸 softmax로 정규화하면

각 단어에 대한 **Attention weight (가중치)**가 된다.

### ➡ 3. 그럼 이 q, k, v를 만드는 Wq, Wk, Wv는 어디서 나옴?

바로 학습으로 만든다.

이 세 개의 행렬(Wq, Wk, Wv)은 수십억 개의 파라미터 중 일부로서  
다음 단어 예측 Loss를 최소화하는 방향으로 학습됨.

즉:

- “고기”와 “먹었다”가 중요한 관계인데
- Attention이 엉뚱하게 “나는”에 집중했다?

그러면 모델 예측이 틀릴 확률이 커져서  
**Loss가 크게 나오고 → 역전파 → Wq/Wk/Wv 수정됨**

반대로,

- “먹었다”가 “고기” 또는 “친구랑”에 적절히 집중하면

- 다음 단어 예측이 잘 맞아 → Loss 작아짐  
→ 가중치가 안정적으로 수렴함
- 

## 트랜스포머

### Autoregressive LM:

- attention: 지금 처리 중인 단어가 문장의 어떤 단어에 집중 해야 하는가?
- input -> encoder(vectorization attention) -> 문맥이 반영된 표현으로 바뀜
- -> decoder -> 이전 생성된 단어와 encoder를 참고해서 다음단어 예측
- -> output
- “문장 속에서 다음 단어를 맞히는 게임”을 반복하면서 언어의 문법, 문맥, 의미를 자연스럽게 배우는 모델이에요.

### BERT:

- 문장 중간에 빈칸을 추론.

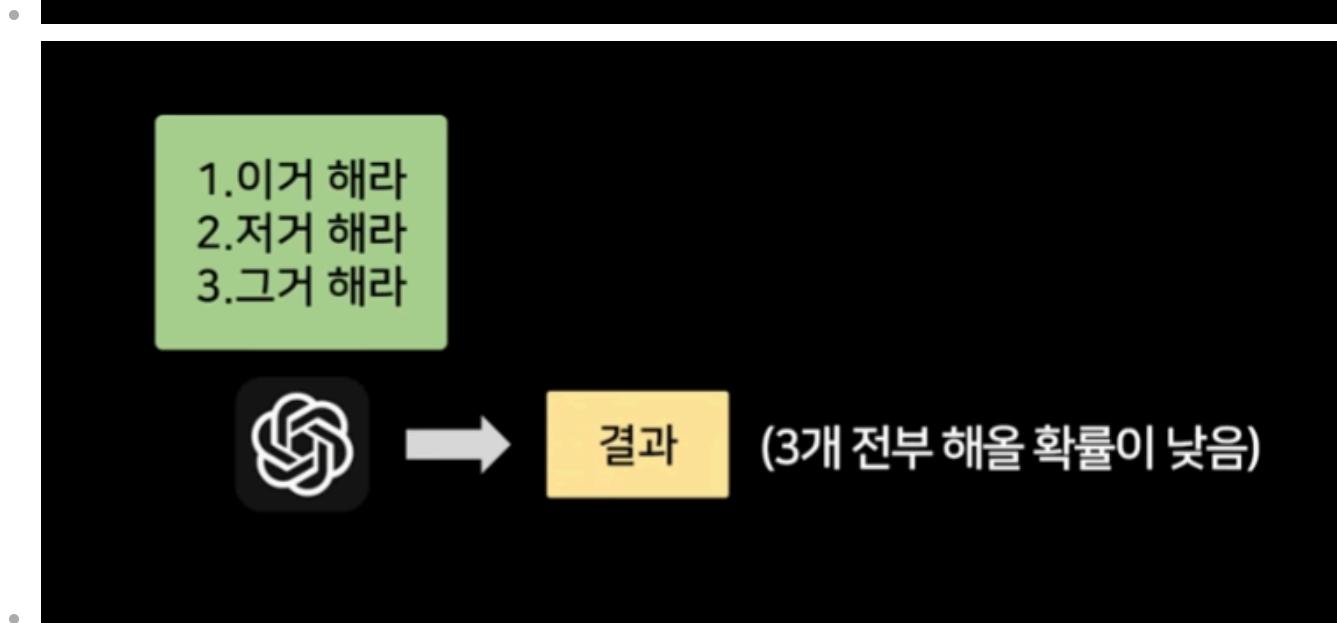
그래서 LLM이 뭐냐

내 입력을 이해해서 답변을 출력하는 모델인데, 어떻게 동작하냐면, 처음에 인터넷의 모든 자료를 가져와서 학습을 진행함, 어떤식의 학습이냐면, 문장을 순차적으로 한 단어씩 이어붙이면서 모델에 넣으면 다음 단어를 예측 할 수 있게끔 그러니까 문장을 넣으면 다음에 올 단어를 예측 하는 모델을 만드는 거지, 그러면 답변생성은 내질문을 LLM에 넣으면 다음 단어가 나오겠지? 그러면 그 단어를 output에 추가하고 다음으로 내답변 + output을 LLM에 넣으면 또 다음 단어가 나오겠지? 이를 반복하면 문장이 나오는데 그걸 출력하는 모델이다, 이게 왜 잘 동작하는가? 학습을 진행할때 질문하고 답변하는 식의 문장을 다수 학습데이터로 사용함. 따라서 잘 동작한다. 또, attention이라는 테크닉을 사용하여, 내가 lm에 보낸 문장에서 각 단어가 이전 어떤단어와 얼마나 상관관계가 있는지 판단하여 그걸 반영하여 문맥을 더 잘 알 수 있게 함. attention도 파라메터를 학습하여 학습을 함에 따라 더 성능이 좋아진다. 학습할 때 혹은 결과를 얻을 때 문장을 어떻게 넣는가? 문장을 단어단위 혹은 토큰단위로 나눈뒤 그걸 임베딩 즉 실수의 의미가 있는 배열로 바꾸어 실수 배열형태로 집어 넣는다. 그럼 문장을 어떻게 실수집합으로 바꾸는가? lm이 보기에 처음 나온 토큰이라면 임베딩 테이블에 집어넣고 랜덤 실수 배열을 같이 적어 둠. 이는 lm이 학습할 때 같이 학습되어 실제 의미있는 단어 사전으로 바뀜. 그럼 토큰단위로 어떻게 뽑냐? 그걸 토크나이저가 하는데 토크나이저는 기존에 누가 만들어둔걸 사용한다. 영어는 BPE, WordPiece -> (알고리즘임) 등을 사용하고 한글은 비슷하게 바이트단위로 나눈다. 나누고 나서 BPE같은 경우 학습하는 과정에서(lm과는 별개) 알아서 병합을 진행한다. 그래서 우리는 이미 학습된 토크나이저를 사용.

agent ai

chaining

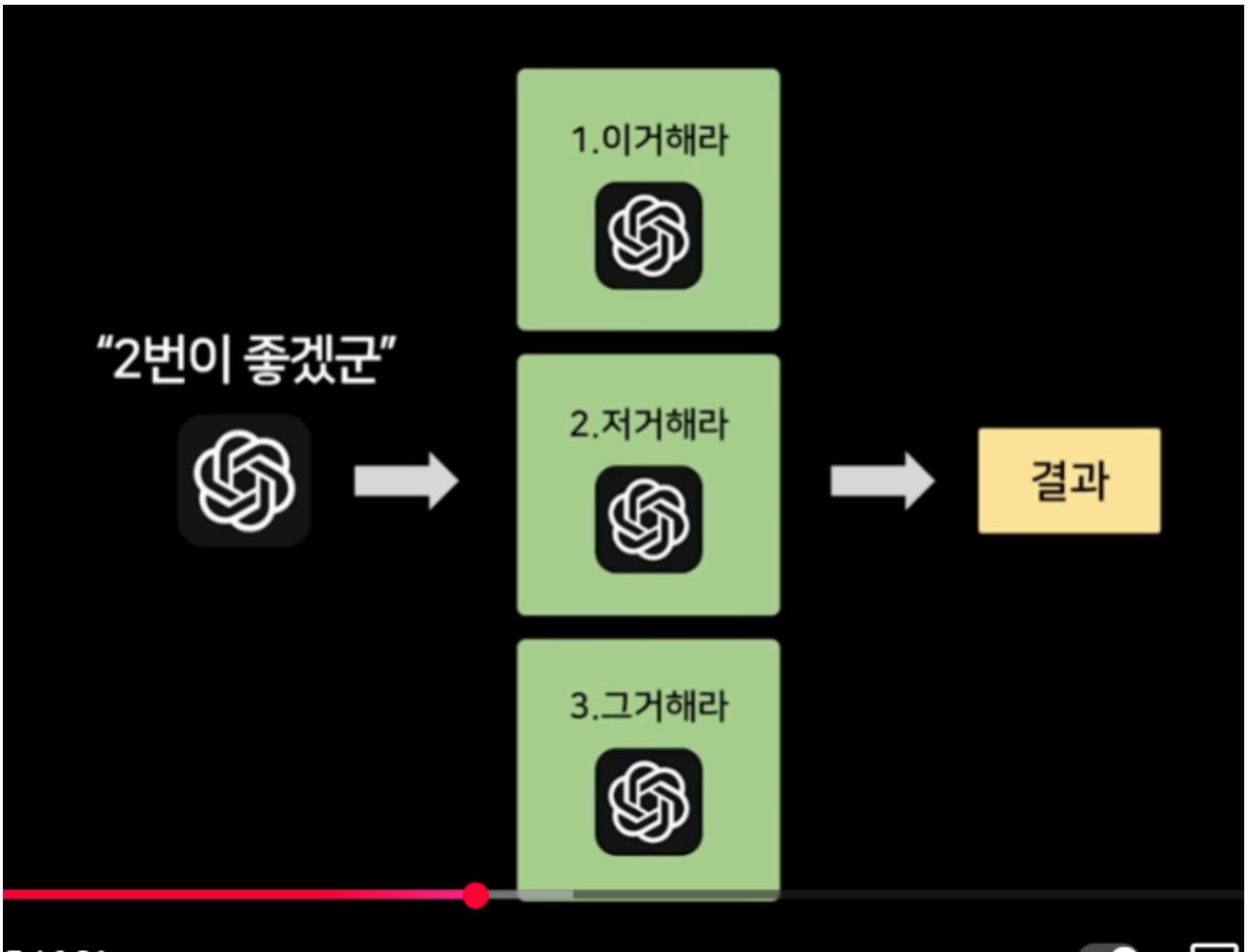
- 순차적으로 gpt(gpt(gpt(A),B),C)해서 순차적으로 답변을 매개변수로 답을 구하는 형식



결과1 = GPT에API요청("1번과정 해와라")  
결과2 = GPT에API요청(결과1, "2번과정 해와라")

## route

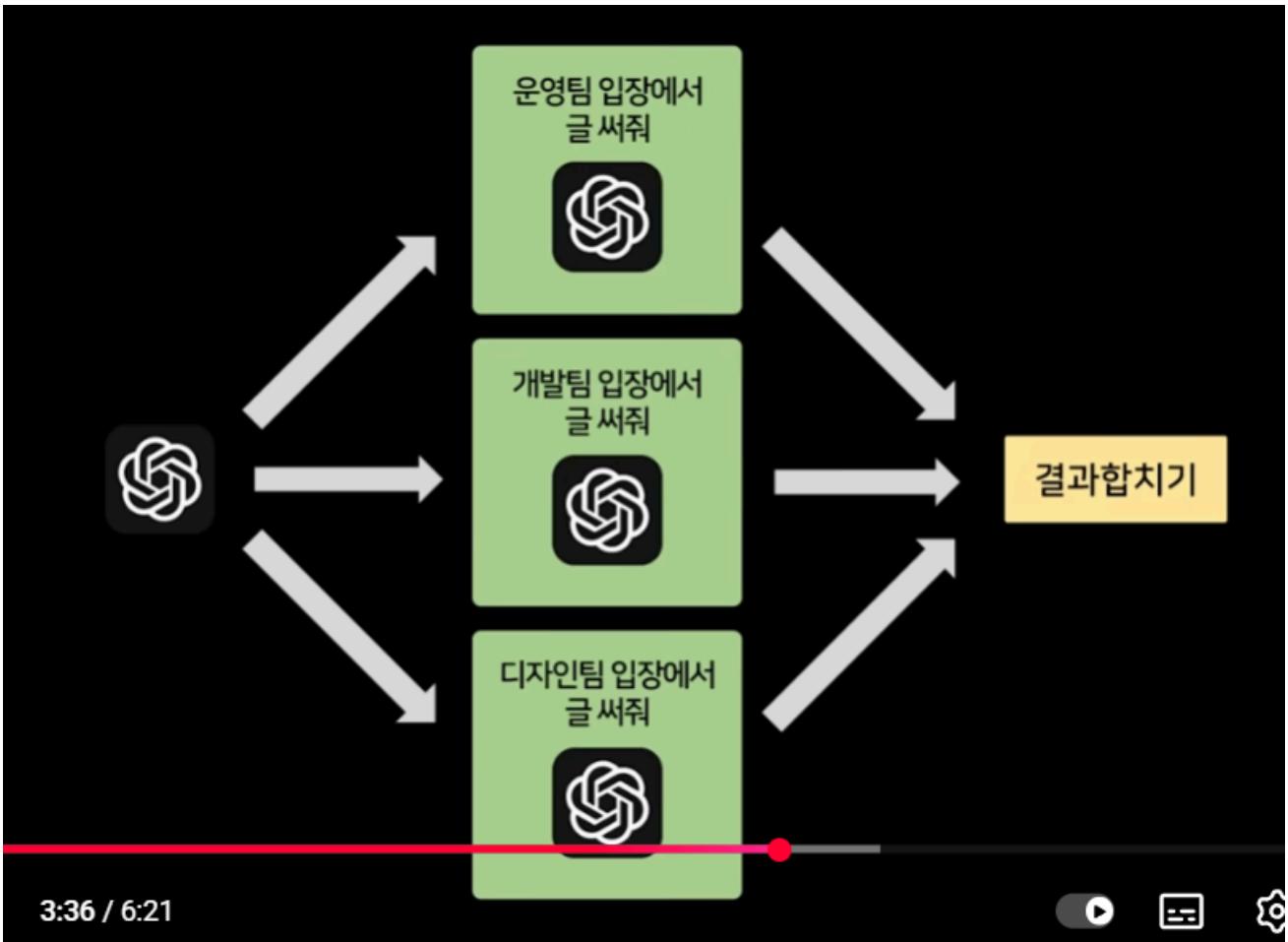
- 가이드 만들
- 운영팀, 개발팀, 디자인팀 등
- 문장이 오면 어느 부서로 전달 할 지 정하고 실제로 거기에 맞게 보내서 추론시키기



- 트 만드는 기초 기술

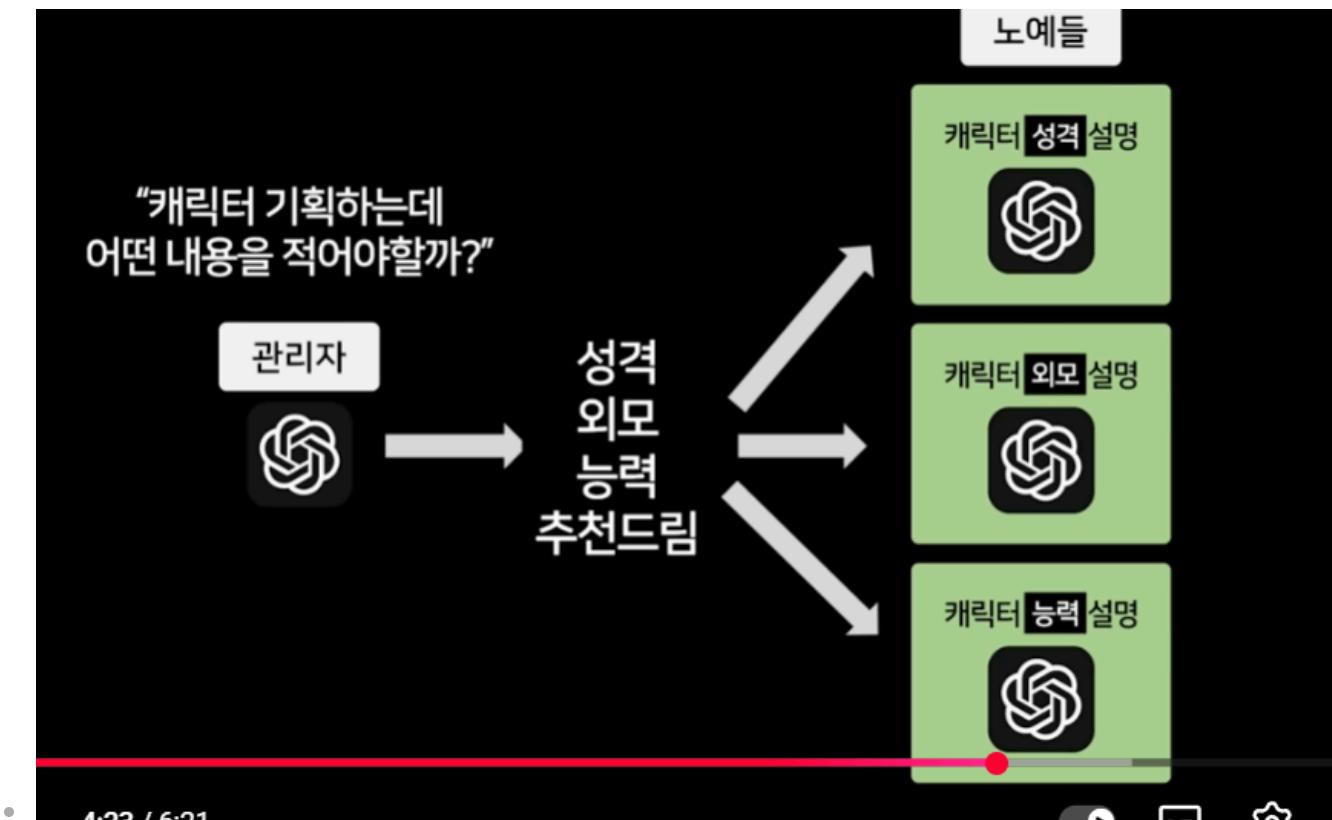
## parallelization

- 여러 부서의 답변을 전부 받아 짜집기하기



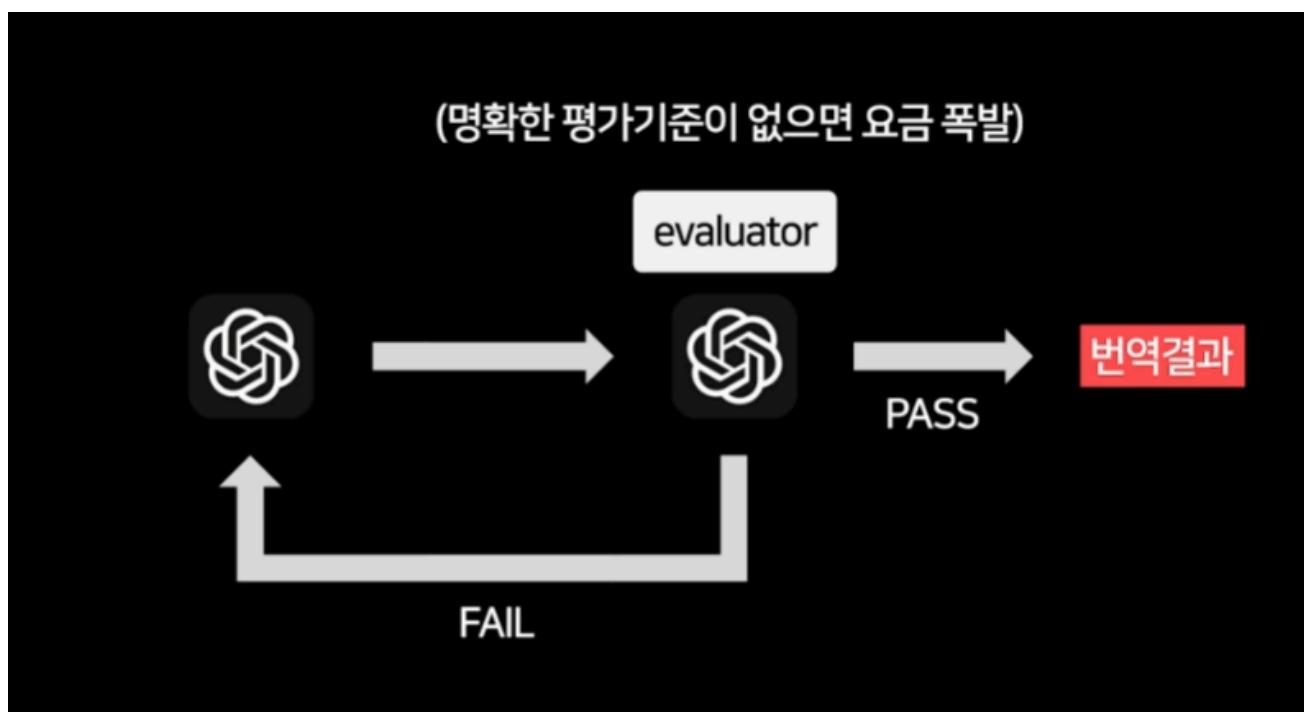
## 관리자 - 노예 orchestrator worker

- 관리자가 - 필요한 내용들 추론
- 노예 - 각각의 내용들을 각각 수행해서 추론
- 짜집기



## evaluation

- 기준을 정하고 기준에 부합하는지 true false 뺑뺑이



+db출력, db저장, 검색, 메일보내기, json csv로 출력  
crewai

```
File Edit Selection View Go Run Terminal Help crew.py - testdev - Visual Studio Code
EXPLORER ... crew.py x main.py
TESTDEV
> .venv
> knowledge
src\testdev
> __pycache__
config
! agents.yaml
! tasks.yaml
> tools
__init__.py
crew.py
main.py
tests
env
.gitignore
pyproject.toml
README.md
report.md
tester1.py
uv.lock
OUTLINE
TIMELINE
```

```
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
from crewai_tools import SerperDevTool

tool = SerperDevTool()

@CrewBase
class Testdev():

    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @agent
    def researcher(self) -> Agent:
        return Agent(
            config=self.agents_config['researcher'],
            verbose=True
        )

    @agent
```

## Model Context Protocol

그래서 agent ai가 뭐냐

입력을 넣으면 출력을 하거나 뭔가 기능을 수행한다.

그럼 llm과 다른점이 뭐냐 여러개의 llm과 mcp를 사용하여 각 기능별로 다른 ai 혹은 mcp에 맡겨서 일을 처리한다. 중앙 llm이 어떤 llm 혹은 mcp를 사용해야하는지 정하고 해당 기능을 수행한뒤 결과를 받아 다시 또 어떤 llm 혹은 mcp를 사용해야하는지 정하고 호출하고를 반복하여 하나의 작업을 여러개의 기능별로 나눠 더 자세하게 답변을 받을 수 있다. mcp가 뭐냐

mcp 서버는 LLM이 사용할 외부 도구/데이터를 제공하는 서버다 mcp는 그냥 프로토콜 즉 규칙임.

서버는 뭐냐? 요청을 받으면 그에 맞는 작업을 해주는 컴퓨터

## 웹 서버

- 브라우저가 “홈페이지 보여줘!” → 요청
- 서버가 “여기 HTML 코드!” → 응답

## 게임 서버

- 플레이어 위치, 점수, 매칭 정보 관리

## 파일 서버

- “파일 주세요” 요청 → 파일 전달

## 서버(Server)

= 요청(request)을 받고

= 처리해서

= 응답(response)을 돌려주는

= 항상 켜져 있는 컴퓨터/프로그램

클라이언트(client)

= 요청을 보내는 쪽

## ✓ 1) 리소스(Resource) 규칙

LLM이 읽을 수 있는 외부 데이터를 표준 구조로 정의하는 규칙이야.

리소스는 반드시 이 형태를 만족해야 함:

```
{  
  "uri": "resource://calendar/events",  
  "type": "calendar.events",  
  "description": "User calendar events",  
  "mimeType": "application/json"  
}
```

LLM에게 제공되는 동작도 표준화됨:

- `list_resources` — 어떤 리소스들이 있는지
- `read_resource` — 리소스의 실제 데이터 읽기
- `subscribe` — 변경될 때 자동 업데이트 받기

📌 즉, 모든 데이터는 ‘리소스’라는 이름 아래 동일한 포맷/동작으로 제공.

## ✓ 2) 도구(Tool) 규칙

LLM이 실행할 수 있는 기능(명령)을 표준화한 구조.

모든 AI 도구는 동일한 JSON Schema를 따라야 하고,

올바른 입력값/출력값 스키마도 명확함.

예시:

```
{  
  "name": "file.write",  
  "description": "Write a file to disk",  
  "inputSchema": {  
    "type": "object",  

```

```
}
```

## LLM은 이 규칙을 사용해서:

- 입력값 형식을 자동으로 맞추고
- 잘못된 파라미터를 정정하고
- 안전하게 도구를 호출할 수 있음

☞ 즉, 도구의 형식·입력·출력 스키마가 모두 표준화되어 LLM이 실수 없이 사용 가능.

---

## ✓ 3) 프롬프트(Prompt) 규칙

LLM이 재사용할 수 있는 템플릿 프롬프트를 표준 구조로 제공하는 규칙.

```
{
  "name": "summarize",
  "description": "Summarize text",
  "arguments": ["text"]
}
```

이 규칙 덕분에:

- LLM 에이전트가 특정 역할을 안정적으로 수행
- 프롬프트를 자율적으로 재사용
- 팀 간 프롬프트 공유 표준화

☞ 즉, 프롬프트도 API처럼 정의해서 LLM이 안정적으로 호출할 수 있게 함.

---

## ✓ 4) 세션(Session) 및 메시지 통신 규칙

LLM과 MCP 서버가 어떻게 통신하는지에 대한 규칙.

### 통신 방식:

- JSON-RPC 기반
- 양방향(LLM ↔ MCP 서버)
- 모든 요청은 `id`, `method`, `params` 를 가짐

예:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "list_tools"  
}
```

## 데이터 업데이트 구독 규칙도 포함:

- LLM은 리소스 변경 이벤트를 실시간으로 받을 수 있음
- 📌 즉, AI가 안정적으로 서버와 대화할 수 있는 표준 메시지 프로토콜.

## 💡 요약하면, MCP의 규칙은 이 4가지야

| 규칙             | 설명                       |
|----------------|--------------------------|
| 리소스 규칙         | 외부 데이터를 표준형식으로 제공        |
| 도구 규칙          | AI가 실행 가능한 기능을 표준화       |
| 프롬프트 규칙        | 재사용 가능한 작업 템플릿 표준화       |
| 통신(Session) 규칙 | LLM과 MCP 서버가 주고받는 메시지 구조 |

## ✨ 쉽게 비유하면

MCP는 AI에게 이렇게 말하는 셈이야:

“모든 데이터는 이 형식으로 줘.  
모든 기능은 이 스키마로 정의해.  
모든 프롬프트는 이 규칙을 따라.  
그리고 우리 통신은 JSON-RPC로 할게.”

즉, AI 에이전트가 여러 서버·시스템과 안정적으로 연결될 수 있도록 표준을 만드는 것이 MCP.

필요하면:

- MCP 규칙 전체 구조
- 실제 JSON-RPC 예시 흐름
- MCP 서버 직접 만드는 코드
- REST API와 MCP 스키마 차이

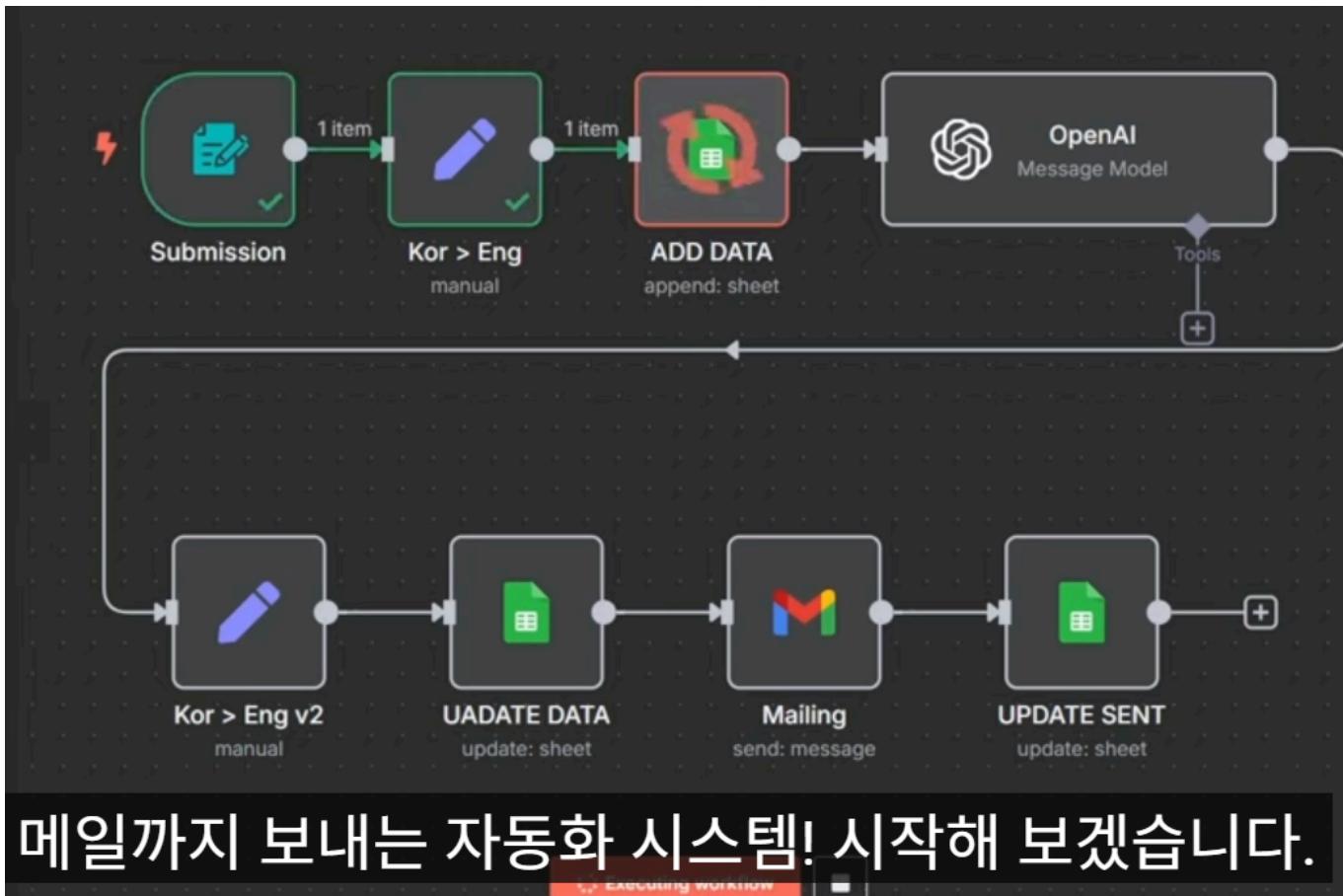
도 설명해줄게!

## n8n

- langchain과 비슷하지만 얘는 AI가 아니고 순서 flow를 명시해줘야함
- n8n
  - [Trigger: Schedule (매일 9시)]
  - ↓
  - [HTTP Request: Weather API 호출]
  - ↓
  - [Function: 메시지 포맷팅]
  - ↓
  - [Slack Node: 메시지 전송]

화살표를 만들어 줘야함.
- langchain  
agent.run("회의 요약해서 Notion에 정리해줘")
- lang graph
- **공통점:** 노드(작업)와 엣지(연결)를 명시적으로 설계해야 함.
- **차이점:** LangGraph는 상태(State)를 들고 있고, LLM이 다음 스텝을 고르게 하는 라우터/슈퍼바이저를 넣을 수 있어 → “고정 플로우 + 유연한 분기” 둘 다 가능.
- 갈수 있는 길을 여러가지 중에 선택할 수 있게 함.
- AI 모델을 불러서 쓰는 자동화 허브 역할
- 워크 플로우 자동화

| 구분 | n8n                   | AI 모델 (LLM, Agent 등) |
|----|-----------------------|----------------------|
| 역할 | 작업 자동화, 연결            | 텍스트/지능 처리            |
| 초점 | 워크플로우, API 호출, 데이터 이동 | 이해, 추론, 생성           |
| 학습 | ✗ 학습하지 않음             | ✓ 사전학습된 언어모델         |
| 예시 | “매일 요약문 생성하고 메일 보내기”  | “요약문 자체를 작성하기”       |



## 랭체인 랭그래프

- n8n처럼 하는데 체인형태 혹은 그래프 형태

```

import os
import requests
import json

def call_openai(messages, tools=None, tool_results=None):
    resp = requests.post(url, headers=headers, data=json.dumps(payload))
    return resp.json()

def get_weather(city: str) -> str:
    # 진짜로는 날씨 API 호출
    return f"{city}의 날씨는 맑음, 20도입니다."

def send_email(to: str, subject: str, body: str) -> str:
    # 진짜로는 이메일 전송 로직
    print(f"[메일 전송] to={to}, subject={subject}\n{body}")
    return "OK"

```

```

# ===== 툴 스펙을 LLM에게 알려줄 JSON =====
tool_specs = [
    {
        "type": "function",
        "function": {
            "name": "get_weather",
            "description": "도시 이름을 받아 현재 날씨를 알려준다.",
            "parameters": {
                "type": "object",
                "properties": {
                    "city": {"type": "string"}
                },
                "required": ["city"],
            },
        },
    },
    {
        "type": "function",
        "function": {
            "name": "send_email",
            "description": "이메일을 보낸다.",
            "parameters": {
                "type": "object",
                "properties": {
                    "to": {"type": "string"},
                    "subject": {"type": "string"},
                    "body": {"type": "string"},
                },
                "required": ["to", "subject", "body"],
            },
        },
    },
],
]

def run_agent(user_input: str):
    messages = [
        {"role": "system", "content": "당신은 유저의 요청을 도와주는 비서입니다."},
        {"role": "user", "content": user_input},
    ]

    tool_results_messages = []

    while True:
        resp = call_openai(messages!!!, tools=tool_specs!!!!,

```

```
tool_results=tool_results_messages!!!
choice = resp["choices"][0]["message"]

# 툴 호출이 필요한지 확인
tool_calls = choice.get("tool_calls")
if not tool_calls:
    # 최종 답변
    print("Assistant:", choice["content"])
    break

# 여러 개의 tool_call을 순차 실행
tool_results_messages = []
for tc in tool_calls:
    fn_name = tc["function"]["name"]
    args = json.loads(tc["function"]["arguments"])

    if fn_name == "get_weather":
        result = get_weather(**args)
    elif fn_name == "send_email":
        result = send_email(**args)
    else:
        result = f"Unknown tool: {fn_name}"

    # tool 결과를 LLM에게 다시 전달
    tool_results_messages.append({
        "role": "tool",
        "tool_call_id": tc["id"],
        "name": fn_name,
        "content": result,
    })

# messages에 방금 assistant의 tool_calls 메시지도 추가
messages.append({
    "role": "assistant",
    "content": choice.get("content") or "",
    "tool_calls": tool_calls,
})
# 그리고 tool 결과도 같이 보내면서 다시 호출 (위에서 처리)

if __name__ == "__main__":
    run_agent("서울 날씨 확인해서, test@example.com 으로 메일 보내줘.")
```

```
import os
from typing import Optional

from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain.agents import create_tool_calling_agent, AgentExecutor

OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]

# ===== 1) LLM 생성 =====
llm = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0,
    openai_api_key=OPENAI_API_KEY,
)

# ===== 2) LangChain의 “툴” 정의 =====
@tool
def get_weather(city: str) -> str:
    """도시 이름을 받아 현재 날씨를 알려준다."""
    return f"{city}의 날씨는 맑음, 20도입니다."

@tool
def send_email(to: str, subject: str, body: str) -> str:
    """이메일을 보낸다."""
    print(f"[메일 전송] to={to}, subject={subject}\n{body}")
    return "OK"

tools = [get_weather, send_email]

# ===== 3) 프롬프트 템플릿 =====
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "당신은 유저를 도와주는 비서입니다."),
        ("user", "{input}"),
        ("assistant", "필요하다면 반드시 툴을 사용해서 작업을 수행하세요."),
    ]
)

# ===== 4) Agent 만들기 =====
agent = create_tool_calling_agent(llm, tools, prompt)

agent_executor = AgentExecutor(
    agent=agent,
```

```

        tools=tools,
        verbose=True, # 내부 진행 상황 출력
    )

def run_agent_langchain(user_input: str):
    result = agent_executor.invoke({"input": user_input})
    # result는 {"input": ..., "output": "..."} 형태
    print("Assistant:", result["output"])

if __name__ == "__main__":
    run_agent_langchain("서울 날씨 확인해서, test@example.com 으로 메일 보내줘.")

```

예시

request

```
{
    "model": "gpt-4o-mini",
    "messages": [
        { "role": "system", "content": "You are a helpful assistant." },
        { "role": "user", "content": "서울 날씨 알려줘" }
    ],
    "tools": [
        {
            "type": "function",
            "function": {
                "name": "get_weather",
                "description": "도시의 날씨를 알려준다.",
                "parameters": {
                    "type": "object",
                    "properties": { "city": { "type": "string" } },
                    "required": [ "city" ]
                }
            }
        }
    ]
}
```

```
{
    "id": "chatcmpl-abc123",
    "object": "chat.completion",
    "choices": [
        {

```

```
"index": 0,
"message": {
    "role": "assistant",
    "content": null,
    "tool_calls": [
        {
            "id": "call_1",
            "type": "function",
            "function": {
                "name": "get_weather",
                "arguments": "{\"city\": \"서울\"}"
            }
        }
    ]
},
"finish_reason": "tool_calls"
}
]
```

```
tool_result = get_weather(city="서울")
```

```
{
"model": "gpt-4o-mini",
"messages": [
    { "role": "system", "content": "You are a helpful assistant." },
    { "role": "user", "content": "서울 날씨 알려줘" },
    {
        "role": "assistant",
        "content": null,
        "tool_calls": [
            {
                "id": "call_1",
                "type": "function",
                "function": {
                    "name": "get_weather",
                    "arguments": "{\"city\": \"서울\"}"
                }
            }
        ]
    },
    {
        "role": "tool",
        "tool_call_id": "call_1",
        "content": null
    }
]
```

```
        "content": "서울의 날씨는 맑고 20도입니다."
    }
]
}
```

```
{
  "id": "chatcmpl-final123",
  "object": "chat.completion",
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "서울의 날씨는 맑고 20도입니다!"
      },
      "finish_reason": "stop"
    }
  ]
}
```

```
[USER]
| "서울 날씨 알려줘"
▼
[LLM 1차 호출]
| → tool_call(JSON): get_weather("서울")
▼
[서버에서 직접 get_weather 실행]
| → "서울 맑음 20도"
▼
[LLM 2차 호출]
| tool 결과 전달
▼
[LLM 최종 답변]
| → "서울은 맑고 20도입니다!"
▼
[USER]
```

## 결론

agent ai는 ai 여러대를 사용하는 방식인데, 미리 뭘 호출해야 하고 그 다음 뭘 호출해야하고 이런걸 정하지 않고 main ai가 필요한 툴한테 request하여 결과를 받는 연쇄작용으로 동작

| 포인트   | 설명                                                           |
|-------|--------------------------------------------------------------|
| 기억 주체 | LLM 자체가 아니라, <b>messages</b> 리스트                             |
| 기억 내용 | 이전 assistant(tool_calls) + tool 결과 메시지                       |
| 기억 방식 | 매번 <code>chat.completions.create(messages=전체 이력)</code> 로 전달 |
| 한계    | 토큰 증가 → 요약 or 세션 메모리 필요                                      |
| 대안 도구 | LangChain, OpenAI ReAct, CrewAI 등에서 자동 관리 가능                 |

```

from langchain.chains import SimpleSequentialChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

llm = OpenAI(model="gpt-5")

summarize_prompt = PromptTemplate.from_template("Summarize this
text:\n{text}")
translate_prompt = PromptTemplate.from_template("Translate to
Korean:\n{summary}")

summary_chain = LLMChain(llm=llm, prompt=summarize_prompt)
translate_chain = LLMChain(llm=llm, prompt=translate_prompt)

chain = SimpleSequentialChain(chains=[summary_chain, translate_chain])
result = chain.run("LangChain is a framework that simplifies building LLM
applications.")
print(result)

```

핵심

```
chain = SimpleSequentialChain(chains=[summary_chain, translate_chain])
```

chain에 넣으면 알아서 해준다.

## MCP (Model Context Protocol)

- LLM(모델)과 외부 도구·데이터 소스가 상호작용하는 표준 프로토콜
- ai 아님
- LLM이 이미 “이해한 자연어 요청”을 MCP가 “기계 명령(API 호출)”로 바꿔 실행



```
|  
|   └── Weather API  
|   └── Calendar API  
└── Custom Database
```

## ☰ 예시 구조

### (1) MCP Tool 정의

```
{  
  "tools": [  
    {  
      "name": "get_weather",  
      "description": "주어진 지역의 날씨를 반환합니다",  
      "input_schema": {  
        "type": "object",  
        "properties": {  
          "location": { "type": "string" },  
          "date": { "type": "string" }  
        },  
        "required": ["location"]  
      }  
    }  
  ]  
}
```

### (2) LLM이 Tool 호출 요청 (요청 메시지)

```
{  
  "type": "tool_use_request",  
  "tool_name": "get_weather",  
  "arguments": { "location": "부산", "date": "2025-11-08" },  
  "request_id": "abc123"  
}
```

### (3) MCP 서버가 Tool 실행 후 응답

```
{  
  "type": "tool_use_result",  
  "request_id": "abc123",  
  "output": { "temperature": 18.2, "condition": "맑음" }  
}
```

## Qwen AI

- 예컨대 Model Studio 국제(싱가포르) 리전 기준으로 “qwen2.5-omni-7b” 모델의 텍스트 입력 토큰 요금이 약 **US\$0.10/백만 토큰** 수준이라는 정보가 있습니다. [AlibabaCloud+1](#)
- 중국(베이징) 리전 기준으로도 입력/출력 토큰당 요금이 공개되어 있습니다  
<https://github.com/QwenLM>  
<https://huggingface.co/Qwen>

```
{  
  "model": "qwen2-vl-plus",  
  "input": [  
    {  
      "role": "user",  
      "content": [  
        {  
          "type": "image",  
          "image": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUg..."  
        },  
        {  
          "type": "text",  
          "text": "이 이미지에서 텍스트만 OCR 해줘."  
        }  
      ]  
    }  
  ]  
}
```

OCR은 **Optical Character Recognition**의 약자야.

쉽게 말하면 **이미지 안에 있는 글자를 컴퓨터가 자동으로 읽어서 텍스트로 변환하는 기술**.

이전 CNN or resnet 그런거 지금 -> 트랜스포머

로컬에서 해야함.

ollama

sonnet