

LangChin AI Agent 만들기2

오산개발자모임

오늘 진행 할 사항

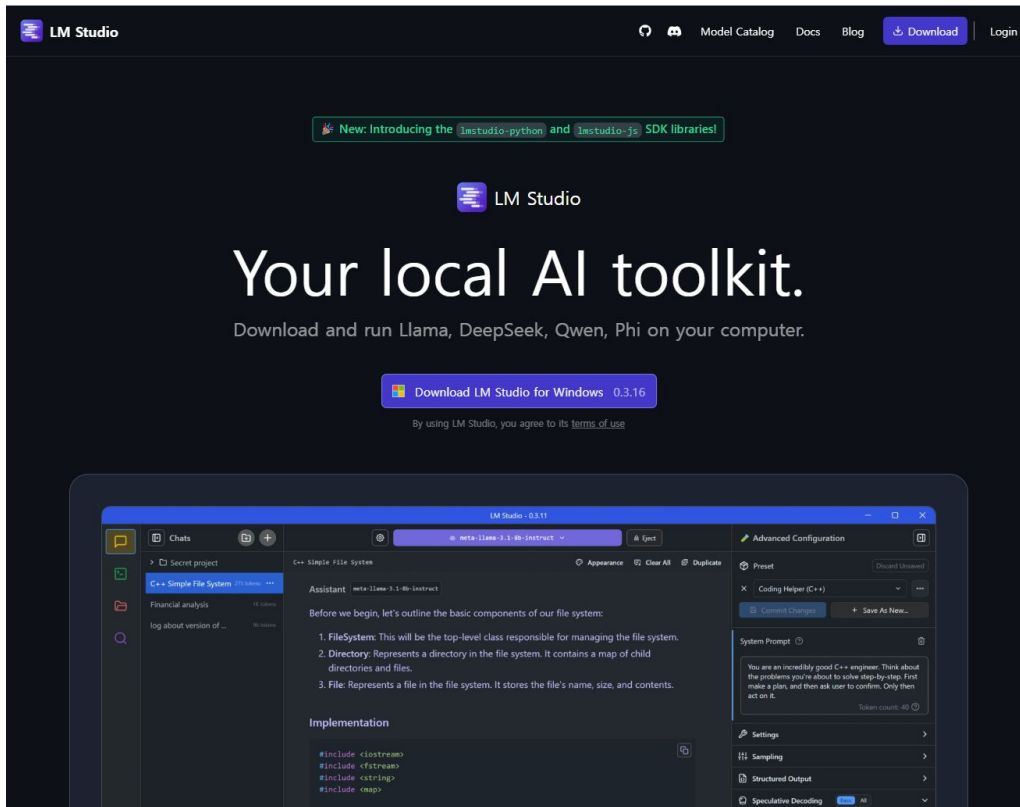
1. langchain을 이용하여 local llm 사용

- a. local llm - LM Studio / gemma 3 - 1b
- b. IDE - cursor ide
- c. language - python
- d. python package install - uv

2. 첫번째 예제 - duckduckgo-search를 이용한 검색 툴 사용

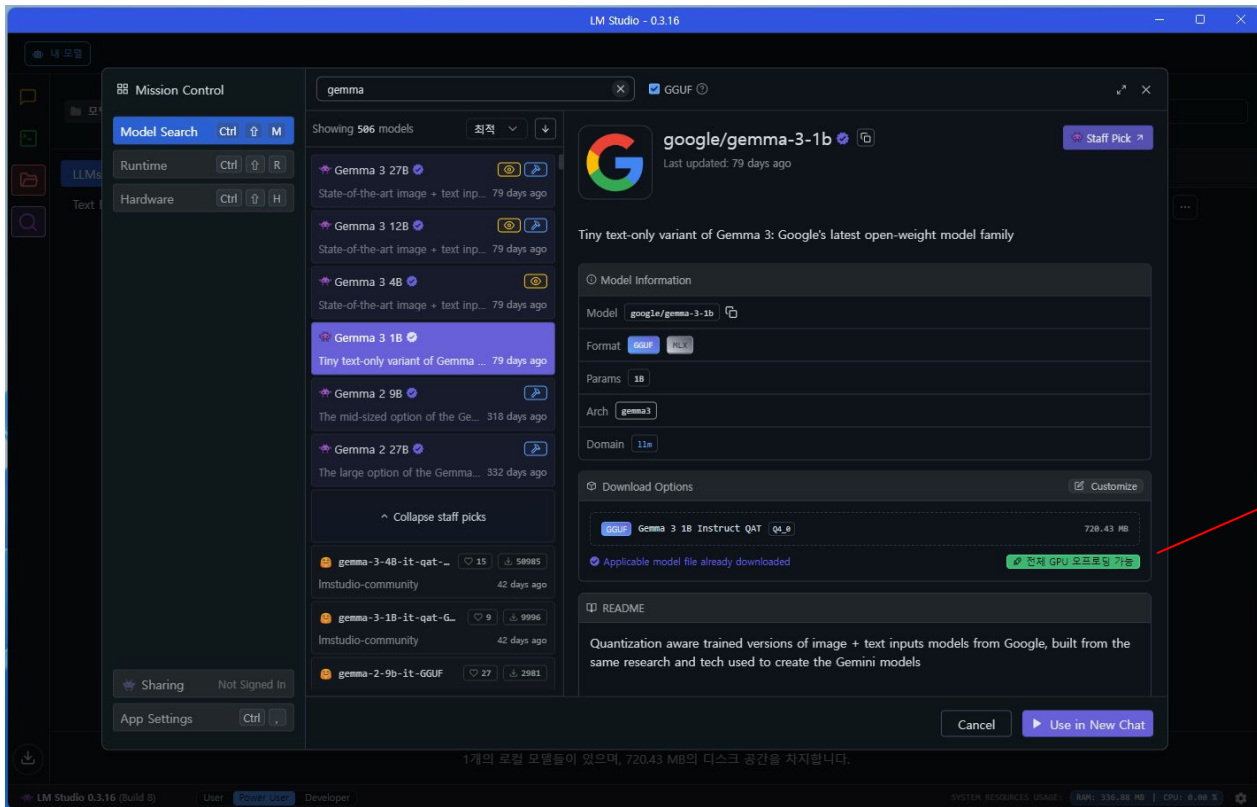
3. 두번째 예제 - prompt template을 이용한 마케팅 문구 작성

Local LLM 설치 - LM Studio 설치

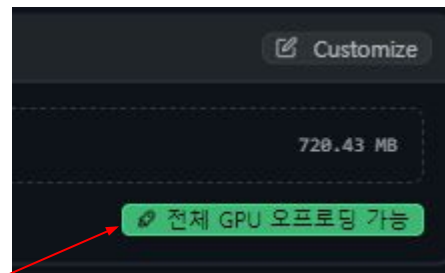


<https://lmstudio.ai/>

Local LLM 설치 - LLM Model 설치



gemma 3 : 1b



다운로드 실행

모델 로드

The screenshot shows the LM Studio application window. At the top, the title bar reads "LM Studio - 0.3.16 (Build 8)". The main interface is divided into several sections:

- Top Bar:** Displays the current model as "google/gemma-3-1b (Ctrl + L)".
- Start Server Section:** A red box highlights the "Start server (Ctrl + R)" button. Below it, the status is "Status: Stopped" with a toggle switch and a "Settings" icon. To the right, it says "Server not running".
- Model Information Panel:** Located on the right, it shows details for the selected model:
 - Model: google/gemma-3-1b
 - File: gemma-3-1b-it-QAT-Q4_0.gguf
 - Format: GGUF
 - Quantization: Q4_0
 - Arch: gemma3
 - Domain: 11m
 - Size on disk: 720.43 MB
- Developer Logs Panel:** At the bottom, it shows API endpoints (GET /v1/models, POST /v1/chat/completions, etc.) and a log entry indicating the server is not running:


```

동 가능 강조:** "[제품명] 스마트워치: 정확한 운동
데이터 분석으로 당신의 운동 목표 달성을 돕겠습니다.
데이터 기반 맞춤 운동 계획으로 최고의 결과를 얻으세
요."** 강조 시간 배터리 사용 강조:** "\배터리 걱정
은 이제 그만! [제품명] 스마트워치는 하루 종일 최적
의 성능을 유지하며, 당신의 활동을 끊임없이 지원합니
다.\*\*\n\n--\n\n\n**추가적으로 고려할 사항:**\n\n\n
**타겟 고객:** 제품을 누가 구매할 것인지에 따라 문

```
- System Resources Usage:** At the bottom right, it shows "RAM: 1.38 GB | CPU: 0.00 %".

1. Server 실행
2. port 확인
3. model 이름 확인

uv 설치 - python package manager

rust로 만들어 짐. pip보다 10배 ~ 100배 빠름

<https://github.com/astral-sh/uv>

Installation

Install uv with our standalone installers:

On macOS and Linux.

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```



On Windows.

```
powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | iex"
```



작업 플로우

1. 작업폴더를 만든다.
2. 가상환경을 만든다.
3. 가상환경을 활성화 한다.
4. 필요한 패키지를 설치한다.
5. 프로그램을 작성한다.
6. LM Studio의 서버를 실행한다.
7. 작성한 프로그램을 실행한다.

```
md my-first-ai-agent
```

```
cd my-first-ai-agent
```

```
uv venv .venv
```

```
.venv\Scripts\activate
```

```
uv pip install langchain-community
```

```
uv pip install langchain_openai
```

첫 번째 예제 - duckduckgo-search 도구를 이용한 검색

가상 환경에 패키지 설치 (방법1)

```
uv pip install langchain-community
```

```
uv pip install langchain
```

```
uv pip install duckduckgo-search
```

가상 환경에 패키지 설치 (방법2)

```
uv pip install -r requirements.txt
```

☞ 실습환경의 쉬운 배포를 위해 방법2
선호

첫 번째 예제 - duckduckgo-search 도구를 이용한 검색

```
1  from langchain_community.chat_models import ChatOpenAI
2  from langchain.agents import initialize_agent, Tool
3  from langchain.agents.agent_types import AgentType
4  from langchain_community.tools import DuckDuckGoSearchRun
5
6
7  llm = ChatOpenAI(openai_api_base="http://localhost:1234/v1",
8                  openai_api_key="lmstudio",
9                  model_name = "google/gemma-3-1b"
10                 )
11
```

- LM Studio가 openAI 형식의 API를 지원하므로 ChatOpenAI를 사용
- port 뒤에 v1을 붙이는 이유는 openAI 인터페이스 이기 때문
- model_name은 LM Studio에 설치되어 있는 model 명을 그대로 사용

첫 번째 예제 - duckduckgo-search 도구를 이용한 검색

```
11
12 # DuckDuckGo 검색 툴 추가
13 search = DuckDuckGoSearchRun()
14
15 tools = [
16     Tool(
17         name="Search",
18         func=search.run,
19         description="웹에서 검색할 수 있습니다."
20     )
21 ]
22
23 # 에이전트 초기화
24 agent = initialize_agent(
25     tools=tools,
26     llm=llm,
27     agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
28     verbose=True,
29     handle_parsing_errors=True # ✅ 이 옵션 추가
30 )
31
32 # 명령어 실행
33 response = agent.run("대한민국 경기도 오산시 시청의 주소를 알려주세요.")
34 print(response)
```

- initialize_agent를 이용하여 LLM과 도구들을 기반으로 agent를 생성.
- 생각(Thought)→행동(Action)
→관찰(Obseration)
→최종답변이라는 일련의 과정이 포함된 에이전트 생성
- DuckDuckGo 검색 도구 추가
- verbose=True 에이전트의 실행 과정을 콘솔에 출력
- handle_parsing_errors=True
LLM이 올바르게 않은 형식의 응답을 줄 경우, 에러를 무시하고 다시 시도. Action과 Final Answer가 동시에 있을 때 생기는 오류를 자동으로 처리

두번째 예제 - prompt template를 이용한 마케팅 문구 생성

02_hello_langchain.py 1

02_hello_langchain.py > ...

```
1  #from langchain_community.chat_models import ChatOpenAI
2  from langchain_openai import ChatOpenAI # ✅ 권장 방식
3  from langchain.prompts import PromptTemplate
4  from langchain.chains import LLMChain
5
6  llm = ChatOpenAI(openai_api_base="http://localhost:1234/v1",
7                  openai_api_key="lmstudio",
8                  model_name="google/gemma-3-1b"
9                  )
10
11  prompt = PromptTemplate(
12      input_variables=["product"],
13      template="다음 제품에 대한 마케팅 문구를 작성해주세요: {product}"
14  )
15
16  chain = LLMChain(llm=llm, prompt=prompt)
17
18  result = chain.run("스마트워치")
19  print(result)
```

1. openAI의 API
인터페이스 사용
2. 프롬프트를 형식화하기
위해서 PromptTemplate
사용
3. llm과 prompt를 조합한
매우 단순한 LLM
워크플로우 실행을 위해
LLMChain 사용
4. 0.1.x버전까지 대표체인.
0.2+버전부터 LCEL로
대체됨.

LLMChain 추가 설명

1. LLMChain이란?

- LangChain 0.0.x ~ 0.1.x까지의 대표 체인.
- 내부적으로 **PromptTemplate + LLM 모델 + (선택) Memory · OutputParser** 등을 하나의 파이프라인으로 실행합니다.
- 0.1.17부터는 “**deprecated(사용 중단 예정)**” 표기가 붙었고, LangChain 0.2+에서는 **LCEL(LangChain Expression Language)**의 **prompt | llm** 파이프 패턴으로 대체됩니다. [LangChainLangChain](#)
- LLMChain은 단계적으로 폐기 예정

✅ 왜 알아야 하나요?

기존 튜토리얼·예제 코드가 여전히 **LLMChain**을 사용하므로, 레거시 코드를 읽고 새로운 LCEL로 이식할 때 구조를 이해해야 합니다.

LLMChain 추가 설명

2. 구성 요소

구성 요소	역할
PromptTemplate	변수(placeholders)를 포함한 프롬프트 템플릿 정의
LLM	OpenAI, Ollama, HuggingFace 등 실제 모델 객체
Memory(선택)	대화 히스토리 관리(예: <code>ConversationBufferMemory</code>)
OutputParser(선택)	모델 응답을 JSON·Pydantic 등 구조화

LLMChain 추가 설명

3. 0.2+ 버전부터는 아래와 같이 변경

LCEL(LangChain Expression Language)

python

복사

편집

```
chain = prompt | llm  
result = chain.invoke({"text": "Hello, how are you?"})
```

AgentType 추가 설명

```
23 # 에이전트 초기화
24 agent = initialize_agent(
25     tools=tools,
26     llm=llm,
27     agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
28     verbose=True,
29     handle_parsing_errors=True # ✅ 이 옵션 추가
30 )
```

AgentType (열거 값)	요약	내부 전략	주 사용처
<code>ZERO_SHOT_REACT_DESCRIPTION</code>	설명만 주면 ReAct 패턴으로 추론, 행동	입력 → (Thought/Action) 반복 → Answer	사전 지식 없는 간단한 작업
<code>CHAT_ZERO_SHOT_REACT_DESCRIPTION</code>	위와 동일하지만 Chat 모델 최적화	동일	챗 기반 LLM (OpenAI GPT-4o 등)
<code>CONVERSATIONAL_REACT_DESCRIPTION</code> (또는 <code>CHAT_CONVERSATIONAL_REACT_DESCRIPTION</code>)	대화 기록(메모리)을 고려하는 ReAct	대화 context + ReAct	챗봇, 고객지원
<code>SELF_ASK_WITH_SEARCH</code>	"질문을 나눠서 스스로 다시 물어보기" 패턴	Self-Ask + 검색 툴	복잡한 지식 탐색
<code>OPENAI_FUNCTIONS</code>	Function-calling 전용 라우터	함수 매핑 테이블	OpenAI 함수 호출
<code>STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION</code>	구조화된 역할/도구 지시어 + ReAct	시스템/도구 프롬프트 → ReAct	명확한 역할 구분 필요
(참고) <code>REACT_DOCSTORE</code> 등	문서 저장소 전용, 현재는 Deprecated	-	-

선택 가이드

필요 조건	추천 AgentType
코드 한 줄로 빠르게 실험	<code>ZERO_SHOT_REACT_DESCRIPTION</code>
대화 이력을 유지해야 함	<code>CONVERSATIONAL_REACT_DESCRIPTION</code>
OpenAI 함수 호출 API 활용	<code>OPENAI_FUNCTIONS</code>
복잡한 지식 탐구(검색)	<code>SELF_ASK</code>
Role / Tool을 구조적으로 제어	<code>STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION</code>

책에서는

1. Zero-short ReAct
2. Conversational ReAct
3. Self-ask with Search
4. ReAct docstore

AgentType 추가 설명

4. AgentType 별 작동 흐름 살펴보기

4-1. ZERO_SHOT_REACT_DESCRIPTION

1. **Prompt**: "도구 설명 + 최종 답변 지시"
2. **LLM**이 *Thought* → *Action* → *Observation* 루프 실행
3. 올바른 톨-순서 자동 결정

```
agent = initialize_agent(tools, llm,  
                        AgentType.ZERO_SHOT_REACT_DESCRIPTION)  
print(agent.run("서울 오늘 날씨 알려줘."))
```

4-2. CONVERSATIONAL_REACT_DESCRIPTION

- 위 흐름에 **Memory**(예: `ConversationBufferMemory`)를 추가

```
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)  
agent = initialize_agent(tools, llm,  
                        AgentType.CONVERSATIONAL_REACT_DESCRIPTION,  
                        memory=memory)
```


AgentType 추가 설명

4-3. SELF_ASK

- ⚙ 내부적으로 "질문 분해 → 검색 → 조합"을 반복
- 자체 검색 툴(예: `ArxivSearchTool`, `SerpAPI`) 필요

2. 검색 툴 준비

```
search_tool = DuckDuckGoSearchRun()  
tools = [search_tool]
```

3. SELF_ASK_WITH_SEARCH 에이전트 생성

```
agent = initialize_agent(  
    tools=tools,  
    llm=llm,  
    agent=AgentType.SELF_ASK_WITH_SEARCH,  
    verbose=True           # Thought / Action 로그 확인용  
)
```

모던 스타일

```
from langchain_openai import ChatOpenAI
from langchain_community.tools import DuckDuckGoSearchRun
from langchain_community.utilities import DuckDuckGoSearchAPIWrapper
from langchain.agents import create_self_ask_with_search_agent, AgentExecutor
from langchain import hub
```

```
llm = ChatOpenAI(model="gpt-3.5-turbo")
```

```
# ✅ RateLimit 회피를 위해 api 백엔드 사용
```

```
search_tool = DuckDuckGoSearchRun(
    name="Intermediate Answer",
    api_wrapper=DuckDuckGoSearchAPIWrapper(backend="api")
)
```

```
prompt = hub.pull("hwchase17/self-ask-with-search")
```

```
agent = create_self_ask_with_search_agent(llm, [search_tool], prompt) ✓
```

```
agent_executor = AgentExecutor(
    agent=agent,
    tools=[search_tool],
    verbose=True,
    handle_parsing_errors=True
)
```

```
question = "한국 최초의 우주인은 누구이고, 현재 하는 일은 무엇인가요?"
```

```
result = agent_executor.invoke({"input": question}) ✓
```

```
print(result["output"])
```

1. initialize_agent 함수 대신 생성 함수 사용
2. 프롬프트 허브를 결합하여 사용자의 프롬프트를 고도화 함. 사용자는 “한국 최초의 우주인은 누구이고, 현재 하는 일은 무엇인가요?”라고 입력하였지만 LLM에 전달 될때는 아래와 같은 형태로 전달 됨.

▶ LLM에 전달될 최종 프롬프트:

You are a smart AI assistant that answers multi-hop questions by reasoning step-by-step.

Question: 한국 최초의 우주인은 누구인가요?

Thought: