
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

`CH16.` 에이전트 *(Agent)*

- `LangChain` 및 기타 `LLM` (Large Language Model) 애플리케이션에서 중요한 개념
- **`인공지능 시스템`이 `더욱 자율적`이고 `목표 지향적`으로 `작업`을 `수행`할 수 있게 해주는 `컴포넌트`**
- **주어진 `목표`를 달성하기 위해 `환경`과 `상호작용`하며 `의사 결정`을 내리고 `행동`을 취하는 `지능형 개체`로 볼 수 있음**

- **`주요 특징`**
  - **`자율성`**: `사전`에 `정의된 규칙`이나 명시적인 `프로그래밍` ❌ → **`스스로 결정`을 내리고 `행동`**
  - **`목표 지향성`**: `특정 목표` or `작업`을 `달성`하기 위해 **`설계`**
  - **`환경 인식`**: `주변 환경` or `상황`을 `인식` → **이에 따라 `적응`**
  - **`도구 사용`**: `다양한 도구` or `API` 활용 → **작업 수행**
  - **`연속성`**: `주어진 목표`를 `달성`하기 위하여 1회 수행 ❌ → **`반복 수행`** → **`목표 달성`** 추구

- **`LangChain에서의 Evaluation`**
  - *`LangChain`은 `LLM`의 애플리케이션의 평가를 위한 다양한 도구와 프레임워크를 제공함*
    - **`모듈화된 평가 컴포넌트`**: 다양한 평가방법 쉽게 구현 및 조합 가능
    - **`Chain 평가`**: 전체 `LLM` 애플리케이션 파이프라인 평가
    - **`데이터셋 기반 평가`**: 사용자 정의 데이터셋 사용 → 모델 평가
    - **`평가 자표`**: 정확성, 일관성, 관련성 등 다양한 지료 제공

- **`LangChain에서의 에이전트`**

- **Agent**: **의사 결정**을 **담당**하는 **핵심 컴포넌트**
- **Tools**: **에이전트**가 **사용**할 수 있는 **기능**들의 **집합**
- **Toolkits**: **관련된 도구**들의 **그룹**
- **AgentExecutor**: **에이전트**의 **실행**을 **관리**하는 **컴포넌트**

- **에이전트**의 **작동 방식**
  - **입력 수신**: 사용자로부터 작업 or 질문 받기
  - **계획 수립**: 주어진 작업을 완료하기 위한 단계별 계획 세우기
  - **도구 선택**: 각 단계에 적합한 도구 선택하기
  - **실행**: 선택한 도구 사용 → 작업 수행
  - **결과 평가**: 수행 결과 평가 → 필요시 계획을 조정
  - **출력 생성**: 최종 결과 or 답변을 사용자에게 제공

- **활용 사례**
  - **정보 검색 및 분석**: 웹 검색, 데이터베이스 쿼리 등을 수행
  - **작업 자동화**: 복잡한 워크플오우 → 자동으로 처리
  - **고객 서비스**: 질문에 답변 → 문제를 해결
  - **의사 결정 지원**: 데이터 분석 → 권장 사항 제공
  - **창의적 작업**: 글쓰기, 코드 생성 등의 **창의적 작업 수행**

- **장점, 한계**
  - **장점**
    - 복잡한 작업의 자동화
    - 유연성, 적응성
    - 다양한 도구와의 통합 가능성
  - **한계**
    - 제어, 예측 가능성의 어려움
    - 계산 비용, 리소스 요구 사항

- `Agent`
  - *`LangChain`, `LLM app` 에서 강력한 도구 → 인공지능 시스템의 `자율성`, `문제 해결 능력`을 크게 향상 → 지속적 연구, 개발 → `Agent` 의 능력, 응용 분야는 계속해서 확장될 것임*

---

1. `도구` *(Tools)*

1) `도구` *(Tools)*

- **Tools = `에이전트`, `체인` 또는 `LLM`이 외부 세계와 `상호작용`하기 위한 `인터페이스`**
- 방법
  - **a. `LangChain`에서 `기본 제공`하는 `도구` 사용** → 쉽게 도구를 활용할 수 있음
  - **b. `사용자 정의 도구` (Custom Tool)** 를 쉽게 구축하는 것도 가능
- 참고
  - **`LangChain 통합 도구 리스트`**

- **`환경설정`**

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv

# API 키 정보 로드
load_dotenv()                                   # True
```

```
from langsmith import Client
from langsmith import traceable

import os

# LangSmith 환경 변수 확인

print("\n--- LangSmith 환경 변수 확인 ---")
langchain_tracing_v2 = os.getenv('LANGCHAIN_TRACING_V2')
langchain_project = os.getenv('LANGCHAIN_PROJECT')
langchain_api_key_status = "설정됨" if os.getenv('LANGCHAIN_API_KEY') else "설정되지 않

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langchain_
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='{langchain_tracing_v2}')
    print(f"✅ LangSmith 프로젝트: '{langchain_project}'")
    print(f"✅ LangSmith API Key: {langchain_api_key_status}")
    print("  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.")
```

```
    else:
        print("❌ LangSmith 추적이 완전히 활성화되지 않았습니다. 다음을 확인하세요:")
        if langchain_tracing_v2 != "true":
            print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있지 않습니다 (현재: '{langch
        if not os.getenv('LANGCHAIN_API_KEY'):
            print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
        if not langchain_project:
            print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

- 셀 출력

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
-> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

```
# ========================================
# 경고 메시지 무시
# ========================================
import os
os.environ['TOKENIZERS_PARALLELISM'] = 'false'
```

---

2) 빌트인 도구 *(built-in tools)*

- 랭체인에서 제공하는 **사전에 정의된 도구(tool)** 와 **툴킷(toolkit)** 을 사용할 수 있음
  - **`tool`** = 단일 도구
  - **`toolkit`** = 여러 도구를 묶어서 하나의 도구로 사용할 수 있음
- 참고:
  - **`LangChain Tools/Toolkits`**
  - langchin tools/toolkits - old vers

---

3) `Python REPL 도구`

- Python 코드를 **`REPL`** *(Read-Eval-Print Loop)* 환경에서 실행하기 위한 클래스 제공
  - 참고
  - 구 링크
  - **`ver.1.0.`**

- **설명**
  - Python 셸 환경 제공
  - 유효한 Python 명령어를 입력으로 받아 실행
  - `print()` 함수 → 결과 출력

- **주요 특징**
  - `sanitize_input`: 입력 정제 옵션 *(기본값 = True)*
  - `python_repl`: PythonREPL 인스턴스 *(기본값 = 전역 범위에서 실행)*

- **사용 방법**
  - `PythonREPLTool` 인스턴스 생성
  - `run` or `arun`, `invoke` 메서드 사용 → Python 코드 실행

- **입력 정제**
  - 입력 문자열에서 불필요한 `공백`, `백틱`, `'python'` 키워드 등을 제거함

```
# 파이썬 코드 실행 → 결과 반환하기
print(python_tool.invoke("print(100 + 200)"))
```

- *Python REPL can execute arbitrary code. Use with caution.*
- 300

- **흐름 정리**
  - `LLM` 모델에게 특정 작업을 수행하는 `Python` 코드를 작성하도록 요청
  - 작성된 코드 실행 → 결과 얻기
  - 결과 출력하기

---

4) `검색 API 도구`

① 검색 도구

- **검색 API 도구**
  - Tavily 검색 API 활용 → 검색 기능 구현하는 도구
    - a. **TavilySearchResults**
    - b. **TavilyAnswer**

*

- **API 키 발급**: API 키 발급 링크
  - `.env` 파일에 환경변수로 설정하기

```
TAVILY_API_KEY=tvly-abc...
```

② Tavily Search Results

- **설명**
  - Tavily 검색 API 쿼리 → **JSON 형식의 결과 반환**
  - 포괄적이고 정확하며 신뢰할 수 있는 결과에 최적화된 검색 엔진
  - 현재 이벤트에 대한 질문에 답변할 때 유용

- **주요 매개변수**
  - **max_results** *(int)*: 반환할 최대 검색 결과 수 *(기본값 = 5)*
  - **search_depth** *(str)*: 검색 깊이 (*"basic"* 또는 *"advanced"*)
  - **include_domains** *(List[str])*: 검색 결과에 포함할 도메인 목록
  - **exclude_domains** *(List[str])*: 검색 결과에서 제외할 도메인 목록
  - **include_answer** *(bool)*: 원본 쿼리에 대한 짧은 답변 포함 여부
  - **include_raw_content** *(bool)*: 각 사이트의 정제된 HTML 콘텐츠 포함 여부
  - **include_images** *(bool)*: 쿼리 관련 이미지 목록 포함 여부

- **반환 값**
  - 검색 결과를 포함하는 **JSON 형식의 문자열**: *url*, *content*

```
# ------------------------------------------------------------
# 1. 환경 설정 및 API 키 로드
# ------------------------------------------------------------

from dotenv import load_dotenv

load_dotenv()

import warnings
warnings.filterwarnings("ignore")

print("환경 설정 및 라이브러리 로드 완료.")
```

```
# ------------------------------------------------------------
# 2. 도구(Tools) 정의
# ------------------------------------------------------------
from langchain_experimental.tools import PythonREPLTool
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.tools import Tool
# LCEL 체인 구성에 필요한 임포트
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import HumanMessage, ToolMessage, SystemMessage
```

```
# 2-1. Python REPL 도구 초기화 및 래핑 (***오류 수정 부분***)
# 1. PythonREPLTool 객체 생성
python_repl = PythonREPLTool()
```

```
# 2. Tool 객체로 래핑
python_repl_tool = Tool(
    name=python_repl.name,
    description=python_repl.description,
    func=python_repl.run,
)
print(f"Python REPL 도구 이름: {python_repl_tool.name}")
```

- `Python REPL` 도구 이름: `Python_REPL`

```
# 2-2. Tavily 검색 도구 초기화 및 래핑

# ① TavilySearchResults 객체 생성
tavily_search = TavilySearchResults(
    max_results=5,
    include_answer=True,
    search_depth="advanced"
)

# ② Tool 객체로 래핑
tavily_tool = Tool(
    name=tavily_search.name,
    description=tavily_search.description,
    func=tavily_search.run,
)
```

```
print(f"Tavily 검색 도구 이름: {tavily_tool.name}")
```

- Tavily 검색 도구 이름: `tavily_search_results_json`

```
# ------------------------------------------------------------
# 3. Gemini LLM 및 LCEL 체인 설정
# ------------------------------------------------------------

# 3-1. LLM 모델 초기화
import os

tools = [python_repl_tool, tavily_tool]

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,
    google_api_key=os.getenv("GOOGLE_API_KEY"))

# LLM에 도구 정보를 바인딩
llm_with_tools = llm.bind_tools(tools)

print(f"LLM 모델 초기화 및 도구 바인딩 완료: {llm.model}")
```

```
# 3-2. 프롬프트 정의
prompt = ChatPromptTemplate.from_messages(
    [
        SystemMessage(
            content=(
                "You are a helpful and powerful AI assistant named Gemini. "
                "You have access to the following tools: {tool_names}. "
                "You should use them when appropriate."
            )
        ),
        ("human", "{input}"),
    ]
)
```

```
# 3-3. 도구 호출 및 실행을 처리하는 함수
def _handle_tool_calls(model_output):
    """모델 출력에서 도구 호출을 확인하고 실행합니다. (인수 전달 로직 개선)"""

    if not model_output.tool_calls:
        return model_output.content

    tool_messages = []

    for tool_call in model_output.tool_calls:
        tool_name = tool_call["name"]
        tool_args = tool_call["args"]
        tool_call_id = tool_call["id"]

        print(f"\n[Tool Call] 도구 이름: {tool_name}, 인수: {tool_args}")
```

```
            tool_to_run = next(
                (tool for tool in tools if tool.name == tool_name), None
            )

            if tool_to_run:
                try:
                    # TavilySearch는 Tool 객체로 래핑되었을 때, .run이 쿼리 문자열을 기대함
                    if tool_name == "tavily_search_results_json":
                        # TavilyResultsTool의 run 메서드는 'query' 인수를 기대
                        result = tool_to_run.func(tool_args.get("query"))
                    # Python REPL 도구는 'code' 인수를 기대
                    elif tool_name == "python_repl_tool":
                        # PythonREPLTool의 run 메서드는 'code' 인수를 기대
                        result = tool_to_run.func(tool_args.get("code"))
                    else:
                        # 기타 도구는 args 전체를 전달 (이는 도구 정의에 따라 달라질 수 있음)
                        result = tool_to_run.func(**tool_args)

                except Exception as e:
                    result = f"Error executing tool {tool_name}: {e}"

                tool_messages.append(
                    ToolMessage(
                        content=str(result),
                        tool_call_id=tool_call_id,
                    )
                )
            else:
                tool_messages.append(
                    ToolMessage(
                        content=f"Error: Tool '{tool_name}' not found.",
                        tool_call_id=tool_call_id,
                    )
                )

        return [model_output] + tool_messages
```

```
    # 3-4. LCEL 기반 체인 구성

    def tool_calling_chain(input_text):
        """사용자 입력에 대해 도구 호출을 반복적으로 처리하는 체인입니다."""

        tool_names = ", ".join([tool.name for tool in tools])

        full_chain = prompt | llm_with_tools

        first_response = full_chain.invoke({"input": input_text, "tool_names": tool_na

        # 도구 호출이 없다면 여기서 종료
        if not first_response.tool_calls:
            return {"output": first_response.content}

        current_messages = [
            HumanMessage(content=input_text),
            first_response,
        ]
```

```
    # 최대 2회 추가 반복 (총 3단계)
    for _ in range(2):
        tool_messages = _handle_tool_calls(current_messages[-1])

        current_messages.extend(tool_messages)

        next_response = llm_with_tools.invoke(current_messages)

        current_messages.append(next_response)

        # 만약 다음 응답이 최종 응답이라면 (도구 호출이 없다면) 종료
        if not next_response.tool_calls:
            # 최종 응답이 빈 문자열일 경우, AI 메시지 객체 자체를 반환하기
            return {"output": next_response.content if next_response.content else

    # 최대 반복 횟수를 초과하면, 마지막 LLM 응답을 반환하기
    return {"output": current_messages[-1].content if current_messages[-1].content
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트_1.1
# ------------------------------------------------------------

print("\n" + "="*50)
print("TEST 1: Python REPL 도구 사용 (코드 실행)")
print("="*50)

result_code = tool_calling_chain("파이썬으로 12345와 67890을 곱한 결과를 계산해 줘.")
print(f"\n[Gemini 최종 응답]: {result_code['output']}")
```

- **`test_1`** - (`5.6s`)

```
==================================================
TEST 1: Python REPL 도구 사용 (코드 실행)
==================================================

[Tool Call] 도구 이름: Python_REPL, 인수: {'__arg1': 'print(12345 * 67890)'}

[Tool Call] 도구 이름: Python_REPL, 인수: {'__arg1': 'print(12345 * 67890)'}

[Gemini 최종 응답]: content='' additional_kwargs={'function_call': {'name': 'Python_
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트_1.2
# ------------------------------------------------------------
print("\n" + "="*50)
print("TEST 5: Python REPL 도구 사용 (대규모 리스트 처리)")
print("="*50)
```

```
# Python REPL을 유도하는 질문 (복잡한 데이터 처리)
result_repl_2 = tool_calling_chain("다음 리스트 [389, 452, 102, 578, 931, 224, 765] 으
print(f"\n[Gemini 최종 응답]: {result_repl_2['output']}")
```

- **`test_1.2`** - (`5.9s`)

```
==================================================
TEST 5: Python REPL 도구 사용 (대규모 리스트 처리)
==================================================

[Tool Call] 도구 이름: Python_REPL, 인수: {'code': '(389 + 452 + 102 + 578 + 931 + 22

[Tool Call] 도구 이름: Python_REPL, 인수: {'code': '(389 + 452 + 102 + 578 + 931 + 22

[Gemini 최종 응답]: content='' additional_kwargs={'function_call': {'name': 'Python_
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트_2.1
# ------------------------------------------------------------

print("\n" + "="*50)
print("TEST 2: Tavily 검색 및 계산 도구 사용 (복합 질문)")
print("="*50)

result_tavily_1 = tool_calling_chain("최근 2년간의 대한민국 인구 증가율이 어떻게 돼? 그리고 그
print(f"\n[Gemini 최종 응답]: {result_tavily_1['output']}")
```

- **`test_2`** - (`21.4s`)

```
==================================================
TEST 2: Tavily 검색 및 계산 도구 사용 (복합 질문)
==================================================

[Tool Call] 도구 이름: tavily_search_results_json, 인수: {'query': '대한민국 인구 증가율 2

[Tool Call] 도구 이름: tavily_search_results_json, 인수: {'query': '대한민국 인구 증가율 2

[Gemini 최종 응답]: 2022년 대한민국의 인구 증가율은 -0.1%이며, 이 값에 100을 곱하면 -10이 됩니다.
2023년 대한민국의 인구 증가율은 0.2%이며, 이 값에 100을 곱하면 20이 됩니다.
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트_2.2
```

```
# ------------------------------------------------------------
print("\n" + "="*50)
print("TEST 4: Tavily 검색 도구 사용 (최신 정보 확인)")
print("="*50)

# Tavily 검색을 유도하는 질문 (오늘의 이슈)
result_tavily_2 = tool_calling_chain("오늘(현재 날짜 기준) 전 세계적으로 가장 큰 이슈가 되는 뉴
print(f"\n[Gemini 최종 응답]: {result_tavily_2['output']}")
```

- **test_2.2** - (19.0s)

```
==================================================
TEST 4: Tavily 검색 도구 사용 (최신 정보 확인)
==================================================

[Tool Call] 도구 이름: tavily_search_results_json, 인수: {'query': '오늘 전 세계적으로 가장

[Gemini 최종 응답]: 죄송합니다. 현재 검색된 뉴스 기사들의 날짜가 2025년으로 표시되어 있어, 오늘(현재 날

하지만 제공된 검색 결과들을 바탕으로 주요하게 언급되는 국제 이슈들을 정리해 드리자면 다음과 같습니다. (다시

1.  **러시아-우크라이나 전쟁 관련 소식:** 러시아와 우크라이나 간의 평화 회담 가능성, 우크라이나의 러시아
2.  **미국 정치 및 국제 외교 (특히 트럼프 관련):** 트럼프 대통령의 우크라이나 전쟁 관련 중요 결정 시사,
3.  **중동 지역 분쟁:** 이스라엘의 예멘 수도 사나 에너지 인프라 공습, 팔레스타인 가족 사망 소식 등 중동
```

---

5) 이미지 생성해보기

```
# ------------------------------------------------------------
# 1. 환경 설정 및 API 키 로드
# ------------------------------------------------------------
from dotenv import load_dotenv
load_dotenv()

import warnings
warnings.filterwarnings("ignore")

print("환경 설정 및 라이브러리 로드 완료")
```

- 환경 설정 및 라이브러리 로드 완료

```
# ------------------------------------------------------------
# 2. 도구(Tools) 정의 (공식 모듈 사용 + 사용자 정의 도구)
# ------------------------------------------------------------
```

```
import os
from typing import Dict, Any

from langchain_experimental.tools import PythonREPLTool
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.tools import Tool

# LCEL 체인 구성에 필요한 임포트
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import HumanMessage, ToolMessage, SystemMessage
```

```
# *** LangChain 모듈 오류 회피를 위해 'GoogleImageGenerationTool' 대신
# *** 사용자 정의 이미지 생성 도구 구현하기
def generate_image_by_imagen(prompt: str, api_key: str = None) -> str:
    """
    Google Imagen 모델을 사용하여 이미지를 생성하고 결과를 반환합니다.
    이 함수는 LangChain 모듈 임포트 오류를 우회하기 위한 사용자 정의 도구입니다.
    """
    try:
        # langchain_google_genai에 포함된 Image API 사용하기
        # 이 함수가 직접 API 키와 프롬프트를 사용하여 이미지 생성 논리 수행하기
        from google import genai

        # API 키는 환경 변수에서 가져오는 것이 가장 좋음
        # 여기서는 ChatGoogleGenerativeAI가 내부적으로 사용할 수 있도록 API 키를 설정하기
        if not api_key:
            api_key = os.getenv("GOOGLE_API_KEY")

        if not api_key:
            return "ERROR: GOOGLE_API_KEY 환경 변수가 설정되지 않았습니다. 이미지 생성을 위해 ㅊ

        client = genai.Client(api_key=api_key)

        print(f"-> Imagen 모델로 이미지 생성 요청: '{prompt[:50]}...'")

        # Imagen 모델 호출
        result = client.models.generate_images(
            model='imagen-3.0-generate-002',
            prompt=prompt,
            config=dict(
                number_of_images=1,
                output_mime_type="image/jpeg",
                aspect_ratio="1:1"
            )
        )

        if result.generated_images:
            # 생성된 이미지의 URI 또는 기타 유용한 정보를 반환하기
            image_uri = result.generated_images[0].uri

            # 사용자에게 이미지가 생성되었고, URI가 반환되었음을 알리는 메시지를 반환
            return f"Image successfully generated. (URI: {image_uri}). Please info
        else:
            return "Image generation failed. No images were returned."
```

```
        except ImportError:
            return "ERROR: google-genai 라이브러리를 임포트할 수 없습니다. 'pip install google-
        except Exception as e:
            return f"ERROR: 이미지 생성 중 오류 발생: {e}"
```

```
# 2-1. Python REPL 도구 초기화 및 래핑
# 1. PythonREPLTool 객체 생성
python_repl = PythonREPLTool()

# 2. Tool 객체로 래핑
python_repl_tool = Tool(
    name=python_repl.name,
    description="Python 코드를 실행하여 수학적 계산이나 데이터 처리를 수행합니다. 'code' 인수를 사
    func=python_repl.run,
)

print(f"Python REPL 도구 이름: {python_repl_tool.name}")
```

```
# 2-2. Tavily 검색 도구 초기화 및 래핑
# 1. TavilySearchResults 객체 생성
tavily_search = TavilySearchResults(
    max_results=5,
    include_answer=True,
    search_depth="advanced"
)
```

```
# 2. Tool 객체로 래핑
tavily_tool = Tool(
    name=tavily_search.name,
    description="인터넷에서 최신 정보를 검색할 때 사용합니다. 'query' 인수를 사용합니다.",
    func=tavily_search.run,)
print(f"Tavily 검색 도구 이름: {tavily_tool.name}")
```

- `Tavily` 검색 도구 이름: `tavily_search_results_json`

```
# 2-3. 사용자 정의 Gemini Image Generation 도구 래핑
gemini_image_tool = Tool(
    name="imagen_generator",                                    # 도구 이름을 명시적으로 지정
    description="주어진 'prompt'를 기반으로 Google Imagen 모델을 사용하여 이미지를 생성합니다.",
    func=generate_image_by_imagen,
)
print(f"Gemini 이미지 생성 도구 이름 (사용자 정의): {gemini_image_tool.name}")
```

- `Gemini` 이미지 생성 도구 이름 (사용자 정의): `imagen_generator`

```
# ------------------------------------------------------------
# 3. Gemini LLM 및 LCEL 체인 설정
# ------------------------------------------------------------

# 3-1. LLM 모델 초기화
# 이미지 생성 도구 (gemini_image_tool) 추가하기
```

```
tools = [python_repl_tool, tavily_tool, gemini_image_tool]
llm2 = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.0)

# LLM에 도구 정보를 바인딩
llm_with_tools = llm2.bind_tools(tools)
print(f"LLM 모델 초기화 및 도구 바인딩 완료: {llm2.model}")
```

- `LLM` 모델 초기화 및 도구 바인딩 완료: `models/gemini-2.5-flash`

```
# 3-2. 프롬프트 정의 (참고 사이트 에이전트 구조를 모방)
prompt = ChatPromptTemplate.from_messages(
    [
        SystemMessage(
            content=(
                "You are a helpful and powerful AI assistant named Gemini. "
                "You have access to the following tools: {tool_names}. "
                "You should use them when appropriate."
            )
        ),
        ("human", "{input}"),
    ]
)
```

```
# 3-3. 도구 호출 및 실행을 처리하는 함수

def _handle_tool_calls(model_output):
    """모델 출력에서 도구 호출을 확인하고 실행합니다. (인수 전달 로직 개선)"""

    if not model_output.tool_calls:
        return model_output.content

    tool_messages = []

    for tool_call in model_output.tool_calls:
        tool_name = tool_call["name"]
        tool_args = tool_call["args"]
        tool_call_id = tool_call["id"]

        print(f"\n[Tool Call] 도구 이름: {tool_name}, 인수: {tool_args}")

        tool_to_run = next(
            (tool for tool in tools if tool.name == tool_name), None
        )

        if tool_to_run:
            try:
                # 사용자 정의 Imagen 도구 처리 (prompt 인수를 기대)
                if tool_name == gemini_image_tool.name:
                    # 사용자 정의 함수에 필요한 인수를 전달함
                    # API 키는 환경 변수에서 가져오므로, 여기서는 prompt만 전달함
                    result = tool_to_run.func(prompt=tool_args.get("prompt"))
                # TavilySearch 도구 처리 (query 인수를 기대)
                elif tool_name == tavily_tool.name:
                    # TavilyResultsTool의 run 메서드는 'query' 인수를 기대
```

```
                    result = tool_to_run.func(tool_args.get("query"))
                # Python REPL 도구 처리 (code 인수를 기대)
                elif tool_name == python_repl_tool.name:
                    # PythonREPLTool의 run 메서드는 'code' 인수를 기대
                    result = tool_to_run.func(tool_args.get("code"))
                else:
                    # 기타 도구는 args 전체를 전달
                    result = tool_to_run.func(**tool_args)

            except Exception as e:
                # API 키 누락 오류 등에 대비
                error_message = f"Error executing tool {tool_name}: {e}"
                if "api_key" in str(e).lower() or "not authorized" in str(e).lower
                    error_message += "\n(Note: GOOGLE_API_KEY 환경 변수가 유효한지 확인하
                result = error_message

            tool_messages.append(
                ToolMessage(
                    content=str(result),
                    tool_call_id=tool_call_id,
                )
            )
        else:
            tool_messages.append(
                ToolMessage(
                    content=f"Error: Tool '{tool_name}' not found.",
                    tool_call_id=tool_call_id,
                )
            )

    return [model_output] + tool_messages
```

```
# 3-4. LCEL 기반 체인 구성
def tool_calling_chain(input_text):
    """사용자 입력에 대해 도구 호출을 반복적으로 처리하는 체인입니다."""

    tool_names = ", ".join([tool.name for tool in tools])

    full_chain = prompt | llm_with_tools

    first_response = full_chain.invoke({"input": input_text, "tool_names": tool_na

    # 도구 호출이 없다면 여기서 종료
    if not first_response.tool_calls:
        return {"output": first_response.content}

    current_messages = [
        HumanMessage(content=input_text),
        first_response,
    ]

    # 최대 2회 추가 반복 (총 3단계)
    for _ in range(2):

        tool_messages = _handle_tool_calls(current_messages[-1])
```

```
            current_messages.extend(tool_messages)

            next_response = llm_with_tools.invoke(current_messages)

            current_messages.append(next_response)

            # 만약 다음 응답이 최종 응답이라면 (도구 호출이 없다면) 종료
            if not next_response.tool_calls:
                # 최종 응답이 빈 문자열일 경우, AI 메시지 객체 자체를 반환함
                return {"output": next_response.content if next_response.content else

        # 최대 반복 횟수를 초과하면, 마지막 LLM 응답을 반환함
        return {"output": current_messages[-1].content if current_messages[-1].content
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트
# ------------------------------------------------------------
print("\n" + "="*50)
print("TEST 3: 이미지 생성 도구 사용 (Gemini Imagen)")
print("="*50)

# 이미지 생성 도구를 사용하도록 유도하는 질문
result_dalle = tool_calling_chain("귀여운 아기 고양이가 우주복을 입고 토성 근처에 떠 있는 이미지를
print(f"\n[Gemini 최종 응답]: {result_dalle['output']}")
```

- **이미지 생성 시도** - (2.7s)

```
==================================================
TEST 3: 이미지 생성 도구 사용 (Gemini Imagen)
==================================================
Both GOOGLE_API_KEY and GEMINI_API_KEY are set. Using GOOGLE_API_KEY.

[Tool Call] 도구 이름: imagen_generator, 인수: {'prompt': 'A cute baby cat in a space
-> Imagen 모델로 이미지 생성 요청: 'A cute baby cat in a spacesuit floating near Satur.

[Gemini 최종 응답]: 죄송합니다. 현재로서는 이미지 생성 기능을 사용할 수 없습니다.
```

- 디버깅

```
# *** LangChain 모듈 오류 회피를 위해 'GoogleImageGenerationTool' 대신
# *** 사용자 정의 이미지 생성 도구로 구현해보기
def generate_image_by_imagen(prompt: str, api_key: str = None) -> str:
    """
    Google Imagen 모델을 사용하여 이미지를 생성하고 결과를 반환합니다.
    이 함수는 LangChain 모듈 임포트 오류를 우회하기 위한 사용자 정의 도구입니다.
```

```
    """
    try:
        # langchain_google_genai에 포함된 Image API를 사용하기
        # 이 함수가 직접 API 키와 프롬프트를 사용하여 이미지 생성 논리를 수행하기
        from google import genai

        # API 키는 환경 변수에서 가져오는 것이 가장 좋음
        # 여기서는 ChatGoogleGenerativeAI가 내부적으로 사용할 수 있도록 API 키를 설정하기
        if not api_key:
            api_key = os.getenv("GOOGLE_API_KEY")

        if not api_key:
            return "ERROR: GOOGLE_API_KEY 환경 변수가 설정되지 않았습니다. 이미지 생성을 위해 ㅊ

        client = genai.Client(api_key=api_key)

        print(f"-> Imagen 모델로 이미지 생성 요청: '{prompt[:50]}...'")

        # Imagen 모델 호출
        result = client.models.generate_images(
            model='imagen-3.0-generate-002',
            prompt=prompt,
            config=dict(
                number_of_images=1,
                output_mime_type="image/jpeg",
                aspect_ratio="1:1"
            )
        )

        if result.generated_images:
            # 생성된 이미지의 URI 또는 기타 유용한 정보를 반환하기
            image_uri = result.generated_images[0].uri

            # 사용자에게 이미지가 생성되었고, URI가 반환되었음을 알리는 메시지를 반환
            return f"Image successfully generated. (URI: {image_uri}). Please info
        else:
            return "Image generation failed. No images were returned."

    except ImportError:
        return "ERROR: google-genai 라이브러리를 임포트할 수 없습니다. 'pip install google-
    except Exception as e:
        # **수정된 부분: API 오류를 더 자세히 출력**
        print(f"\n[IMAGE GENERATION API ERROR]: {e}")
        return f"ERROR: 이미지 생성 중 API 오류 발생. 상세 정보는 콘솔을 확인하거나, GOOGLE_API
```

```
# 2-1. Python REPL 도구 초기화 및 래핑
# 1. PythonREPLTool 객체 생성
python_repl = PythonREPLTool()
```

```
# 2. Tool 객체로 래핑
python_repl_tool = Tool(
    name=python_repl.name,
    description="Python 코드를 실행하여 수학적 계산이나 데이터 처리를 수행합니다. 'code' 인수를 사
    func=python_repl.run, # .run 메서드는 문자열 인수를 받습니다.
```

```
)
print(f"Python REPL 도구 이름: {python_repl_tool.name}")
```

```
# 2-2. Tavily 검색 도구 초기화 및 래핑
# 1. TavilySearchResults 객체 생성
tavily_search = TavilySearchResults(
    max_results=5,
    include_answer=True,
    search_depth="advanced"
)
# 2. Tool 객체로 래핑
tavily_tool = Tool(
    name=tavily_search.name,
    description="인터넷에서 최신 정보를 검색할 때 사용합니다. 'query' 인수를 사용합니다.",
    func=tavily_search.run, # .run 메서드는 문자열 인수를 받습니다.
)
print(f"Tavily 검색 도구 이름: {tavily_tool.name}")

# 2-3. 사용자 정의 Gemini Image Generation 도구 래핑
gemini_image_tool = Tool(
    name="imagen_generator", # 도구 이름을 명시적으로 지정
    description="주어진 'prompt'를 기반으로 Google Imagen 모델을 사용하여 이미지를 생성합니다.",
    func=generate_image_by_imagen,
)
print(f"Gemini 이미지 생성 도구 이름 (사용자 정의): {gemini_image_tool.name}")
```

```
# ------------------------------------------------------------
# 3. Gemini LLM 및 LCEL 체인 설정
# ------------------------------------------------------------

# 3-1. LLM 모델 초기화
# 이미지 생성 도구 (gemini_image_tool)를 추가하기
tools = [python_repl_tool, tavily_tool, gemini_image_tool]
llm3 = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.0)

# LLM에 도구 정보를 바인딩
llm_with_tools = llm3.bind_tools(tools)
print(f"LLM 모델 초기화 및 도구 바인딩 완료: {llm3.model}")
```

```
# 3-2. 프롬프트 정의 (참고 사이트 에이전트 구조를 모방)
prompt = ChatPromptTemplate.from_messages(
    [
        SystemMessage(
            content=(
                "You are a helpful and powerful AI assistant named Gemini. "
                "You have access to the following tools: {tool_names}. "
                "You should use them when appropriate."
            )
        ),
        ("human", "{input}"),
    ]
)
```

```python
# 3-3. 도구 호출 및 실행을 처리하는 함수
def _handle_tool_calls(model_output):
    """모델 출력에서 도구 호출을 확인하고 실행합니다. (인수 전달 로직 개선)"""

    if not model_output.tool_calls:
        return model_output.content

    tool_messages = []

    for tool_call in model_output.tool_calls:
        tool_name = tool_call["name"]
        tool_args = tool_call["args"]
        tool_call_id = tool_call["id"]

        print(f"\n[Tool Call] 도구 이름: {tool_name}, 인수: {tool_args}")

        tool_to_run = next(
            (tool for tool in tools if tool.name == tool_name), None
        )

        if tool_to_run:
            try:
                # 사용자 정의 Imagen 도구 처리 (prompt 인수를 기대)
                if tool_name == gemini_image_tool.name:
                    # 사용자 정의 함수에 필요한 인수를 전달합니다.
                    # API 키는 환경 변수에서 가져오므로, 여기서는 prompt만 전달하기
                    result = tool_to_run.func(prompt=tool_args.get("prompt"))
                # TavilySearch 도구 처리 (query 인수를 기대)
                elif tool_name == tavily_tool.name:
                    # TavilyResultsTool의 run 메서드는 'query' 인수를 기대
                    result = tool_to_run.func(tool_args.get("query"))
                # Python REPL 도구 처리 (code 인수를 기대)
                elif tool_name == python_repl_tool.name:
                    # PythonREPLTool의 run 메서드는 'code' 인수를 기대
                    result = tool_to_run.func(tool_args.get("code"))
                else:
                    # 기타 도구는 args 전체를 전달
                    result = tool_to_run.func(**tool_args)

            except Exception as e:
                # API 키 누락 오류 등에 대비
                error_message = f"Error executing tool {tool_name}: {e}"
                if "api_key" in str(e).lower() or "not authorized" in str(e).lower
                    error_message += "\n(Note: GOOGLE_API_KEY 환경 변수가 유효한지 확인하
                result = error_message

            tool_messages.append(
                ToolMessage(
                    content=str(result),
                    tool_call_id=tool_call_id,
                )
            )
        else:
            tool_messages.append(
                ToolMessage(
                    content=f"Error: Tool '{tool_name}' not found.",
                    tool_call_id=tool_call_id,
```

```
                )
            )
    return [model_output] + tool_messages
```

```
# 3-4. LCEL 기반 체인 구성
def tool_calling_chain(input_text):
    """사용자 입력에 대해 도구 호출을 반복적으로 처리하는 체인입니다."""

    tool_names = ", ".join([tool.name for tool in tools])

    full_chain = prompt | llm_with_tools

    first_response = full_chain.invoke({"input": input_text, "tool_names": tool_na

    # 도구 호출이 없다면 여기서 종료
    if not first_response.tool_calls:
        return {"output": first_response.content}

    current_messages = [
        HumanMessage(content=input_text),
        first_response,
    ]

    # 최대 2회 추가 반복 (총 3단계)
    for _ in range(2):
        tool_messages = _handle_tool_calls(current_messages[-1])

        current_messages.extend(tool_messages)

        next_response = llm_with_tools.invoke(current_messages)

        current_messages.append(next_response)

        # 만약 다음 응답이 최종 응답이라면 (도구 호출이 없다면) 종료함
        if not next_response.tool_calls:
            # 최종 응답이 빈 문자열일 경우, AI 메시지 객체 자체를 반환함
            return {"output": next_response.content if next_response.content else
    # 최대 반복 횟수를 초과하면, 마지막 LLM 응답을 반환함
    return {"output": current_messages[-1].content if current_messages[-1].content
```

```
# ------------------------------------------------------------
# 4. 에이전트 실행 테스트_2
# ------------------------------------------------------------
print("\n" + "="*50)
print("TEST 3: 이미지 생성 도구 사용 (Gemini Imagen)")
print("="*50)

# 이미지 생성 도구를 사용하도록 유도하는 질문
result_dalle = tool_calling_chain("귀여운 아기 고양이가 우주복을 입고 토성 근처에 떠 있는 이미지를
print(f"\n[Gemini 최종 응답]: {result_dalle['output']}")
```

- **이미지 생성 시도_2** - (5.5s)

```
==================================================
TEST 3: 이미지 생성 도구 사용 (Gemini Imagen)
==================================================
Both GOOGLE_API_KEY and GEMINI_API_KEY are set. Using GOOGLE_API_KEY.

[Tool Call] 도구 이름: imagen_generator, 인수: {'prompt': '귀여운 아기 고양이가 우주복을 입고
-> Imagen 모델로 이미지 생성 요청: '귀여운 아기 고양이가 우주복을 입고 토성 근처에 떠 있는 모습...'

[IMAGE GENERATION API ERROR]: 400 INVALID_ARGUMENT. {'error': {'code': 400, 'messa

[Gemini 최종 응답]: 죄송합니다. 이미지 생성 중 오류가 발생했습니다. 다시 시도해 주시거나, 나중에 다시 시
```

- 해당 이미지 생성 `API` 생성 권한 ❌ → 추후 다시 시도 예정

- next: ***`01_Tools_2`***
