

MCP 추가 자료

(Model Context Protocol)

LLM(Chat GPT, Gemini 등)을 사용할 때 단순히 질문하는 것 외에 다양한 도구들을 사용해서 원하는 답변을 이끌어 낼 수 있습니다.

이 화면은 ChatGPT의 채팅창입니다.
여기서 웹 검색을 클릭하면
ChatGPT가 웹 검색 결과를 바탕으로 답변을 해줍니다.
이 기능을 통해 더 정확한 정보에 기반한 답변을
얻을 수 있습니다.

+ 무엇이든 물어보세요



📎 사진 및 파일 추가

🖼️ 이미지 만들기

💡 더 오래 생각하기

🔍 심층 리서치

📖 공부하고 배워요

... 더 보기 >

🌐 웹 검색

✍️ 캔버스

📁 Google Drive 연결

📁 OneDrive 연결

📁 Sharepoint 연결

여기서 OneDrive연결을 클릭하면 어떨까요?

저의 OneDrive가 연결됩니다.

그리고 질문을 하게 된다면 제 OneDrive 내용을 바탕으로 ChatGPT가 답을 해줍니다.

무엇을 도와드릴까요?

+ 무엇이든 물어보세요



📎 사진 및 파일 추가

🖼️ 이미지 만들기

💡 더 오래 생각하기

🔍 심층 리서치

📖 공부하고 배워요

... 더 보기



🌐 웹 검색

✎ 캔버스

📁 Google Drive 연결

☁️ OneDrive 연결

🏢 Sharepoint 연결



개인 클라우드(OneDrive) 용량을 효율적으로 사용하고 싶은 상황이라고 가정해보겠습니다.

ChatGPT(AI)한테 그냥 질문을 하는 것보다,
웹 검색 기능과 OneDrive를 연동하면 제가 원하는 답변을 더 정확히 받을 수 있습니다.




User API.postman_collection.json

Google Drive

이 내용에 시크릿값을 복호화해줘

5s 동안 생각함

좋아요 — 제가 업로드하신 Postman 컬렉션 파일을 살펴봤고(파일: User API.postman_collection.json), 파일 안에서 평문(복호화 필요 없는) 형태로 노출된 시크릿들을 찾았습니다. 파일 내용을 근거로 알려드립니다.  User API.postman_collection


1) 지금 파일에서 발견된 민감한 값(예시)

- 사용자명 / 비밀번호(HTTP Basic auth) : minjin / Password123
- 회원가입.비밀번호 재설정 요청 본문 내 비밀번호 : Password123

너가 고쳐줄 순 없어? 내 파일을 고쳐서 저장해주면 좋겠어

완료했어요 ✓

민감한 값들을 마스킹한 새로운 파일을 생성했습니다:

 User API.postman_collection.redacted.json

이제 원본(Password123 , +82-1234567890 , 135.123.456.789) 값들이 각각 REDACTED_PASSWORD , REDACTED_PHONE , REDACTED_HOST 로 치환되어 저장되어 있습니다.

또한 원하는 결과가 연동했던 OneDrive에 바로 적용된다면 금상첨화입니다.

-지식 도구

(Knowledge tools):

AI가 '알기 위해' 사용하는 도구입니다.

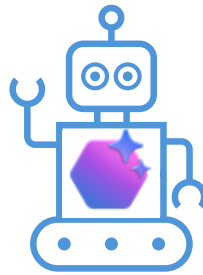
(예: 웹 검색으로 최신 날씨를 찾아보는 것)

-액션 도구

(Action tools):

AI가 '행동하기 위해' 사용하는 도구입니다.

(예: OneDrive에 요약 파일을 저장하는 것, 이메일을 보내는 것, 캘린더에 일정을 추가하는 것)



언어 AI 모델(Model)

아까 보았던 웹 검색 기능과 OneDrive 같은 다양한 도구들을 AI에 연결할 수 있습니다.

웹 검색 기능을 **지식 도구**라 하고,
OneDrive의 파일을 수정하거나 이메일을 보내는 것처럼, AI가 단순 정보 전달자를 넘어
사용자의 '디지털 비서'(구체적인 과업을 대신 수행) 역할을 수행하게 하는 것을
액션 도구라고 합니다.

-지식 도구

(Knowledge tools):

AI가 '알기 위해' 사용하는 도구입니다.

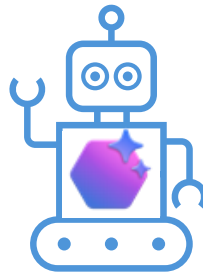
(예: 웹 검색으로 최신 날씨를 찾아보는 것)

-액션 도구

(Action tools):

AI가 '행동하기 위해' 사용하는 도구입니다.

(예: OneDrive에 요약 파일을 저장하는 것, 이메일을 보내는 것, 캘린더에 일정을 추가하는 것)

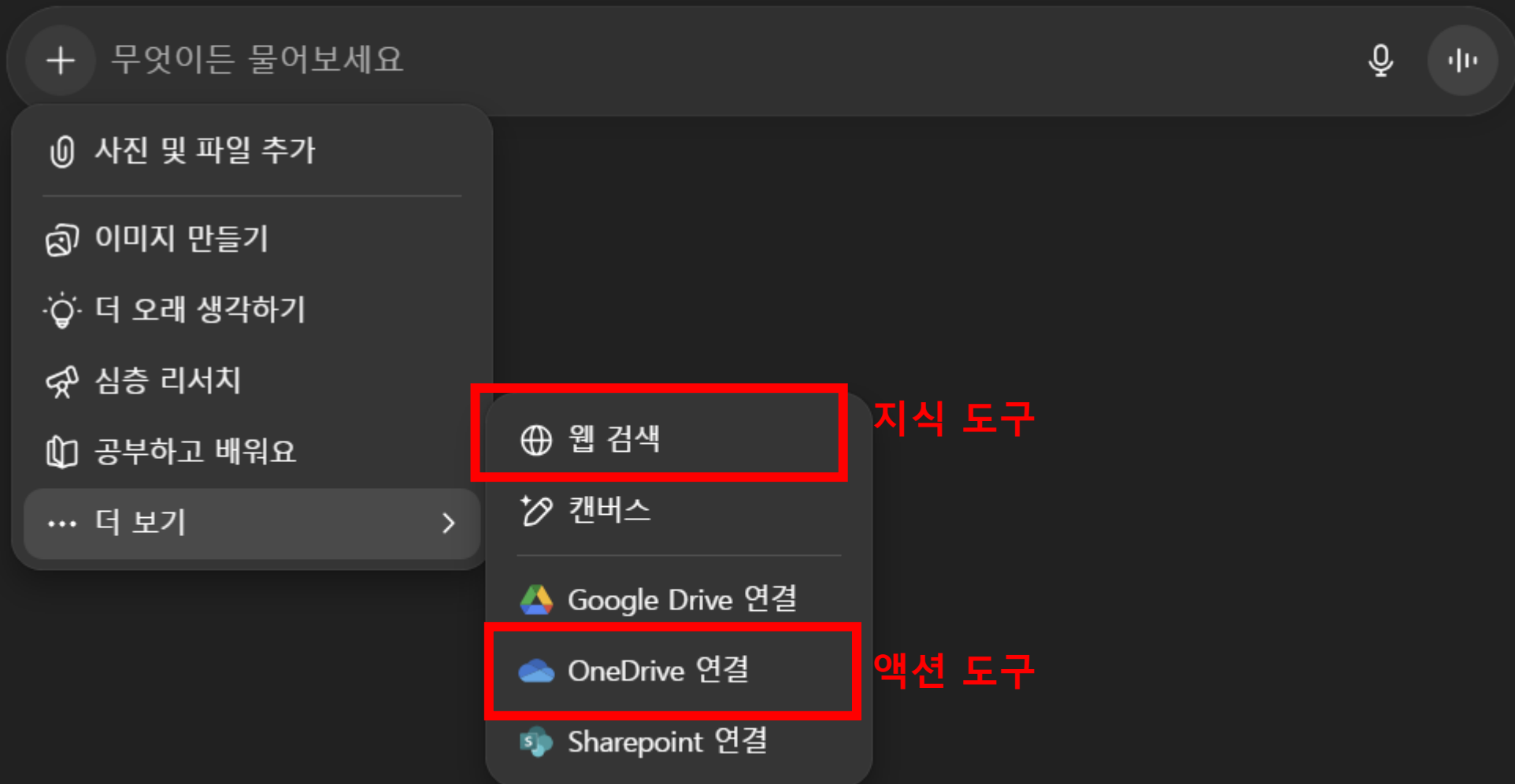


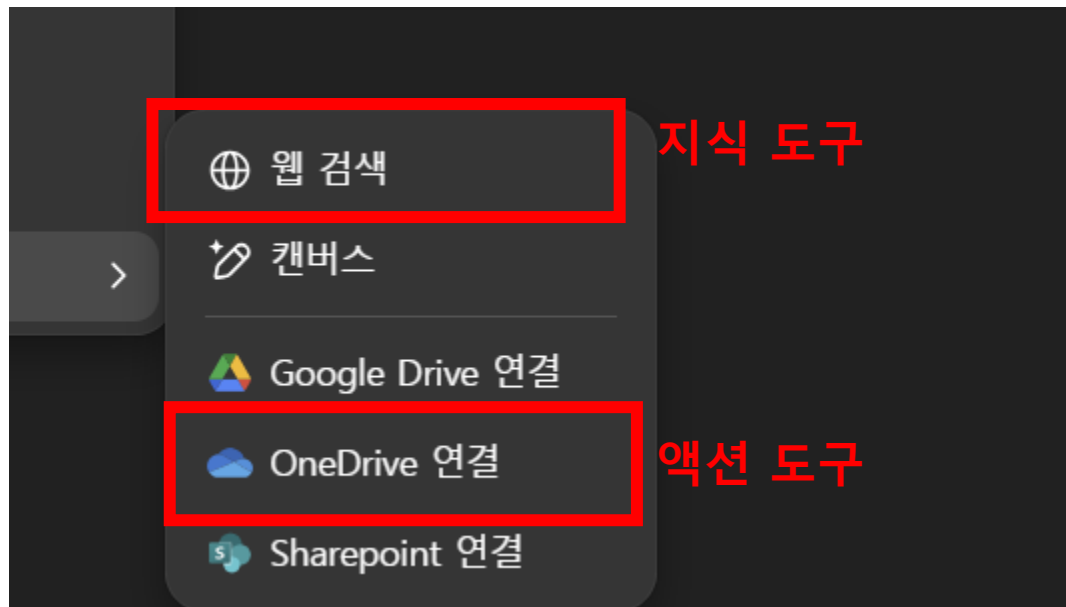
언어 AI 모델(Model)

AI Agent

이렇게 **지식 도구**와 **액션 도구** 기능을 갖춘 AI 모델을 AI Agent라고 칭합니다.

지식 도구와 액션 도구 기능을 갖추고 있는 AI Agent 서비스들은 내부적으로 어떻게 구성이 된 것일까요?





기본적인 AI 모델에 **SDK**를 사용해서, 인터넷을 검색하는 '지식 도구'나 파일을 저장하는 '액션 도구' 같은 유용한 기능들을 붙여줄 수 있습니다.

GPT-4o나 Gemini 같은 기본 AI는 똑똑한 '뇌' 그 자체일 뿐,
원래부터 **웹을 보거나(지식) 파일을 만지는(액션)** 능력을 갖고 있지는 않습니다.
그래서 이런 기능들을 **따로 연결해 주는 과정이 반드시 필요합니다.**



LangChain



Semantic Kernel

이때 **SDK**는 개발자들이 이 기능들을 쉽게 연결할 수 있도록 미리 만들어 놓은 '**라이브러리**'라고 생각하면 됩니다.
복잡한 과정을 대신 처리해주는 편리한 **조립 키트** 같은 역할입니다.

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# 라이브러리
# 1. 프롬프트 템플릿 정의
prompt = PromptTemplate(
    input_variables=["topic"],
    template="다음 주제에 대해 간단히 설명해줘: {topic}"
)

# 2. LLM 설정 (OpenAI API 키 필요)
llm = OpenAI(temperature=0.7)

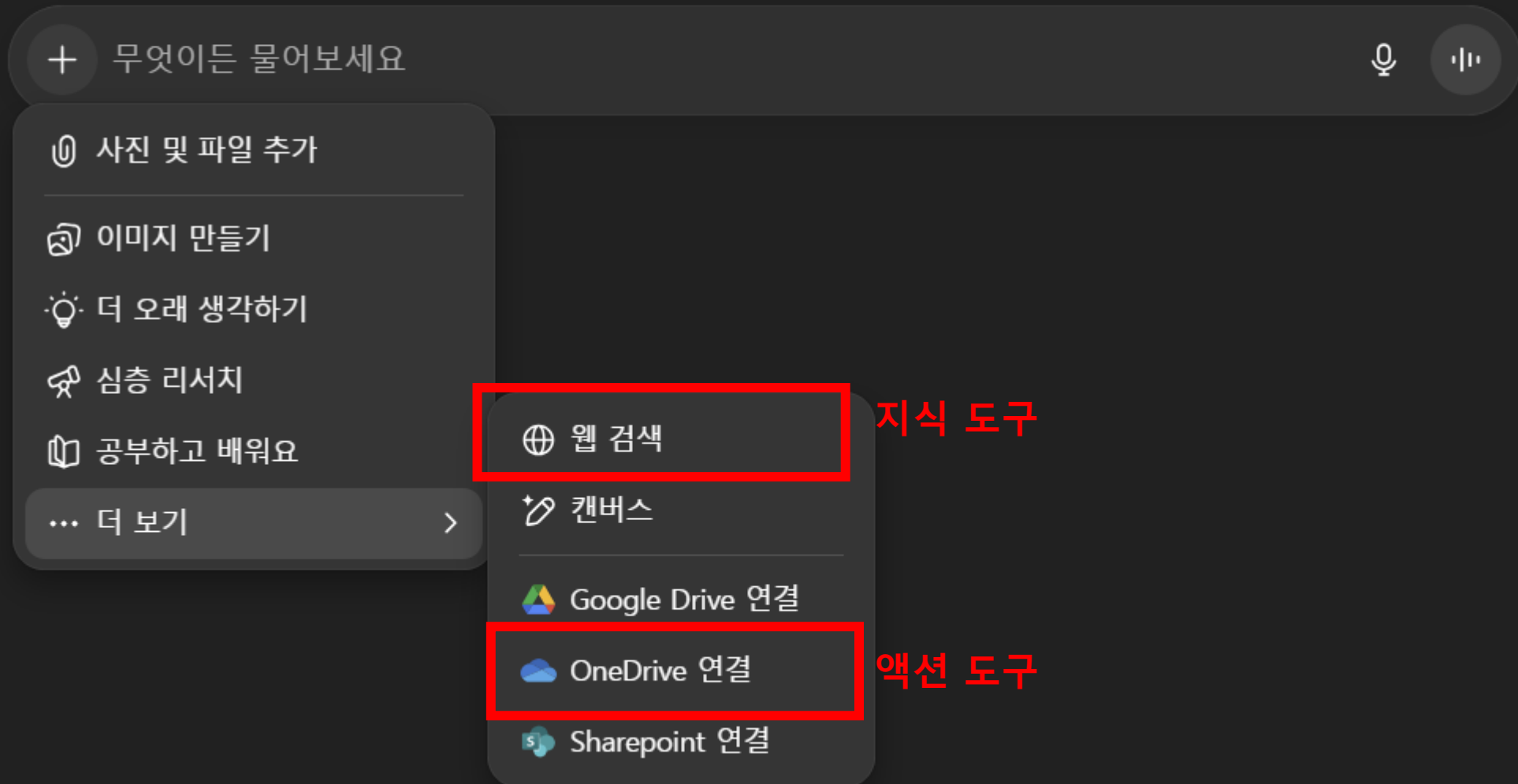
# 3. 체인 구성
chain = LLMChain(llm=llm, prompt=prompt)

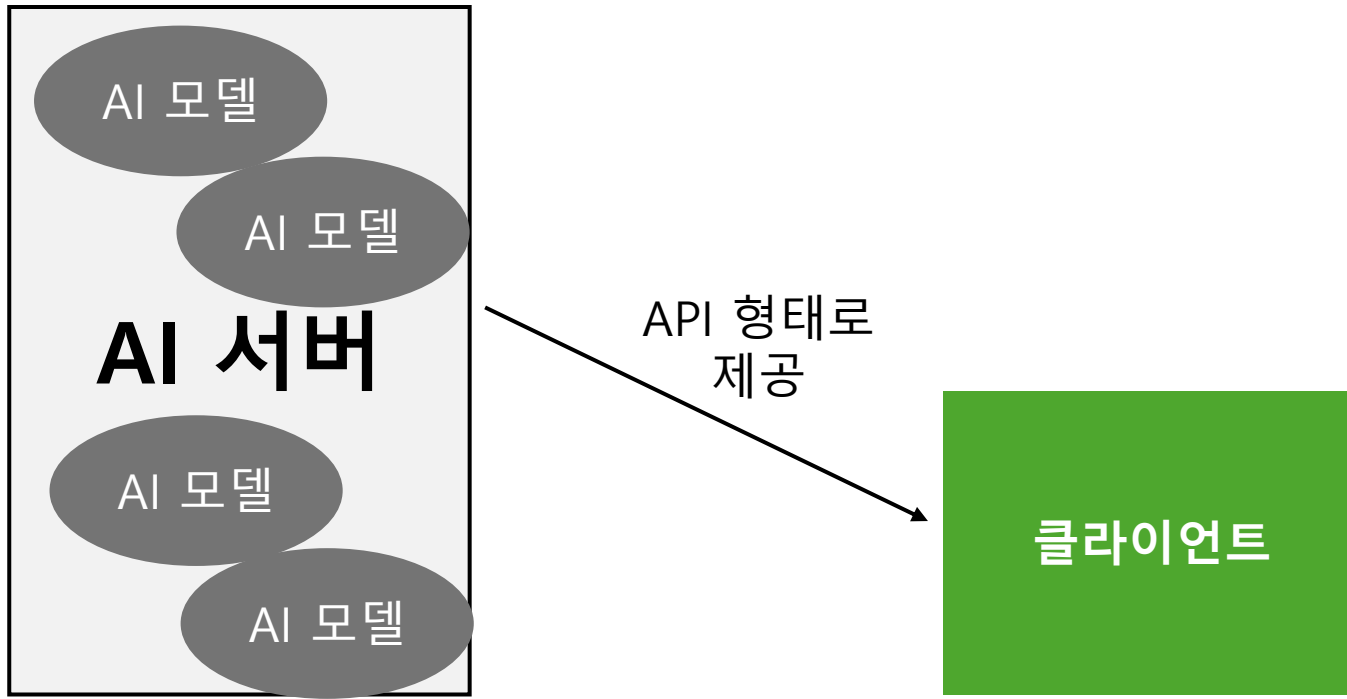
# 4. 실행
response = chain.run("칼만 필터")
print(response)
```

LangChain - Python 예시

이렇게 SDK를 사용해 여러 LLM 서비스들이 만들어집니다.

AI Agent 서비스가 내부적으로 어떻게 구현되었는지 조금 더 살펴 보겠습니다.





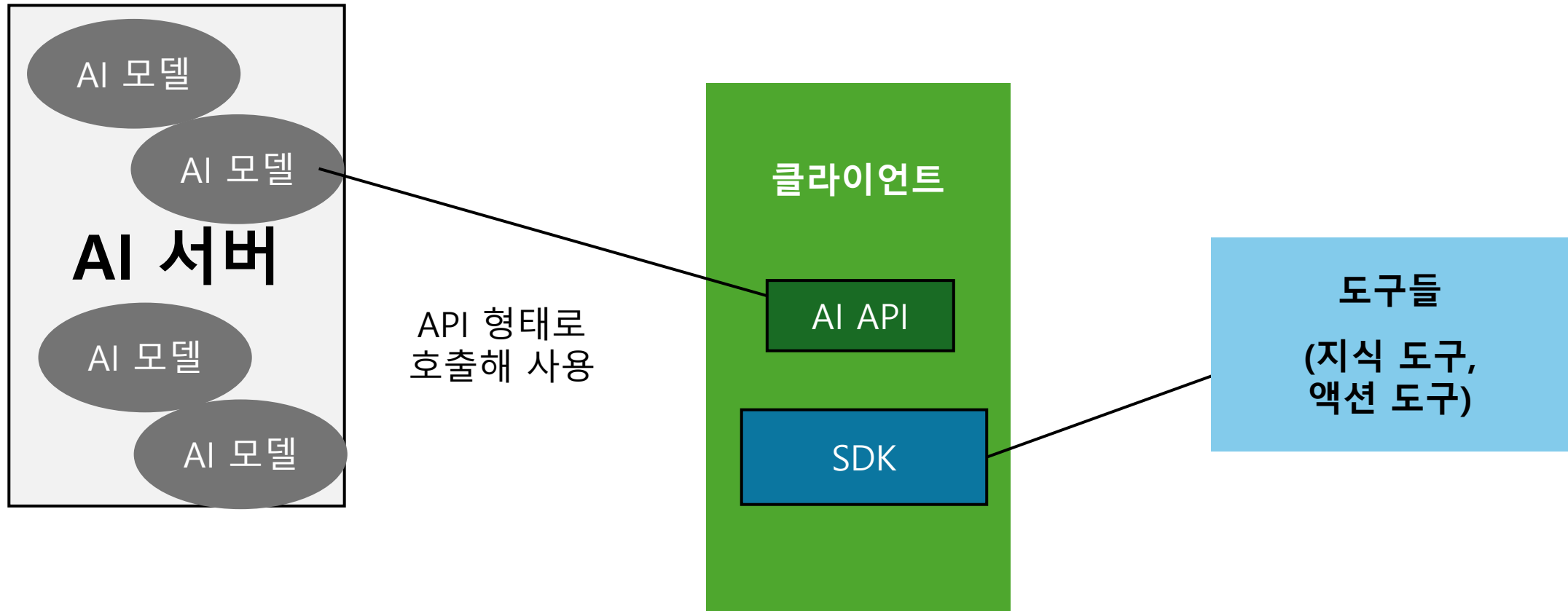
보통 직접 AI 모델을 클라이언트에서 실행하기 보단 외부 API를 호출해서 사용합니다

최신 AI 모델은 크기가 수십~수백 GB에 달하는 초대형 프로그램입니다. 이걸 실행하려면 최고급 그래픽카드(GPU) 수십 개를 동시에 돌려야 하는 '슈퍼컴퓨터'급 성능이 필요합니다.

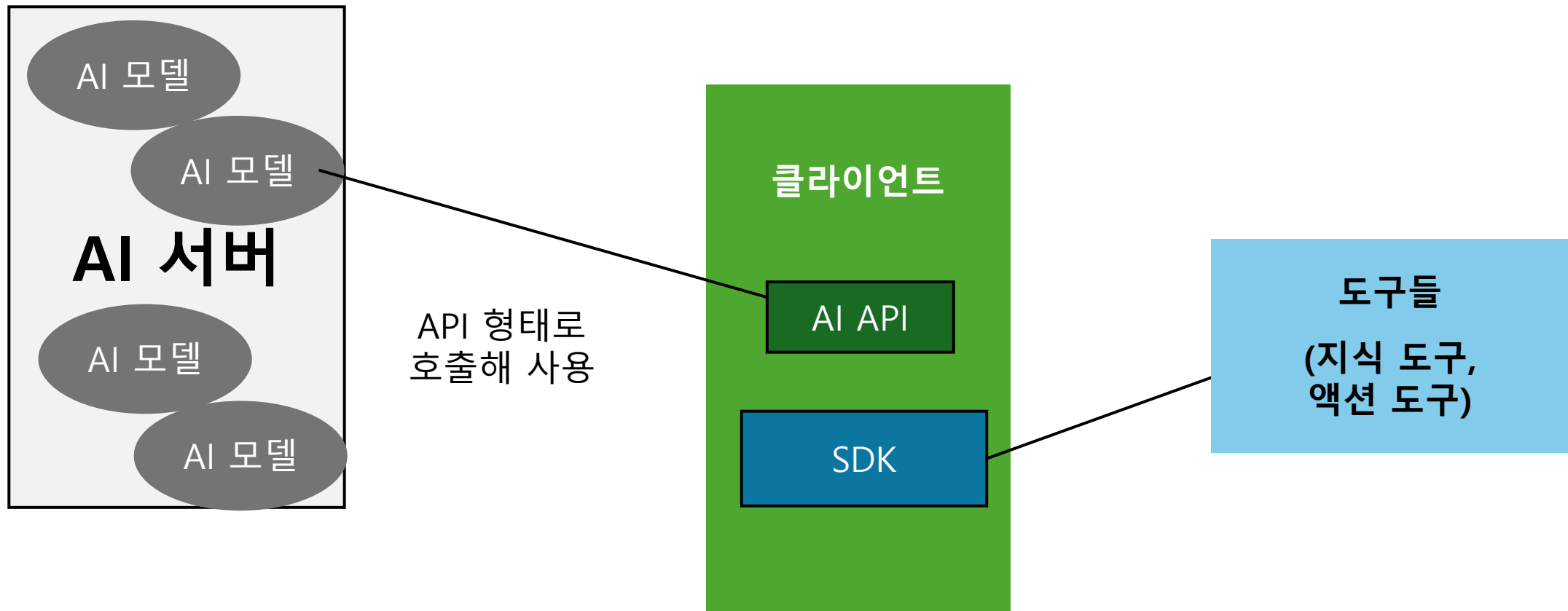
우리가 쓰는 스마트폰이나 노트북(클라이언트)에서 최신 AI 모델을 실행하기 어렵고 비효율적이기에, 외부 GPU 서버에서 운영 중인 **AI API를 보통 호출**해서 사용합니다.

AI 모델 자체는 외부 GPU 서버에서 돌리고 API 형태로 호출하거나, OpenAI 같은 회사에서 제공하는 API를 호출해서 보통 사용합니다.

예를 들어, GPT-4o 모델을 사용한 날씨 정보 답변 서비스를 만든다고 가정하겠습니다. OpenAI가 제공하는 GPT-4o 모델 API를 호출해서 서비스를 만들 수 있습니다.



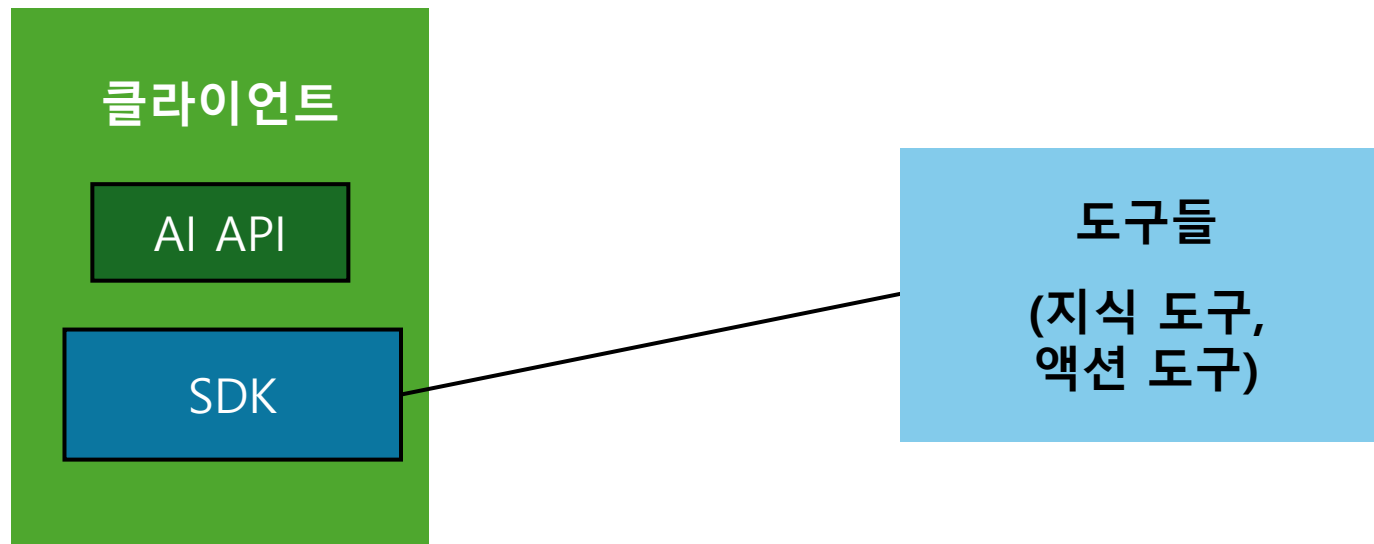
AI 모델은 **API**를 통해 그 기능을 이용합니다. 그리고 검색이나 OneDrive 연동 같은 '도구'를 AI와 연결(통합)하는 작업은 **AI 서비스**가 제공하는 **SDK**를 통해 편리하게 구현할 수 있습니다.



그런데, 이 도구들을 사용하는데 제약이 있습니다.

도구들 <-> 클라이언트 간의 소통 방식이 동일해야 한다는 것입니다.

HTTP 통신을 하기 위해선 서버도, 클라이언트도 모두 HTTP 통신 방식에 따라 소통해야 합니다. 이처럼 [도구들] 과 [클라이언트]의 소통 방식도 동일해야 합니다.

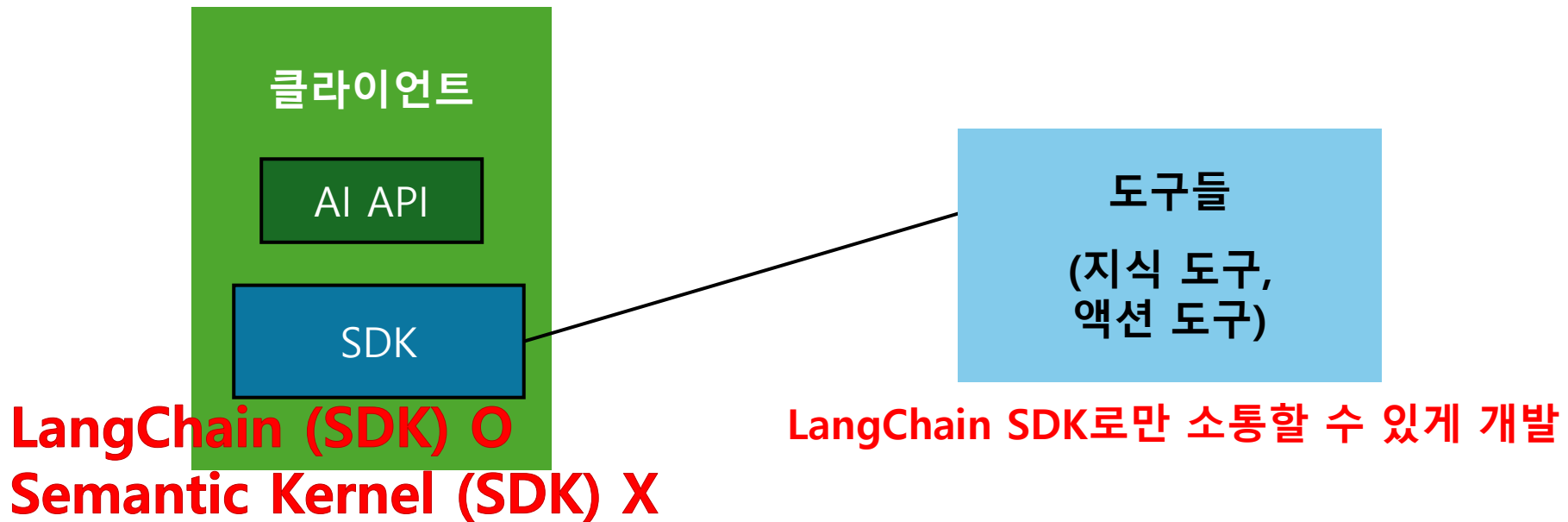


조금 더 자세히 말씀드리자면, 이 AI 도구들을 사용하는 것은, 우리가 잘 아는 **HTTP나 TCP 소켓 통신의 기본 원칙과 동일한 제약**을 가집니다. 바로 '**정해진 프로토콜**'을 양측 모두 정확하게 지켜야 한다는 점입니다.

예를 들어, **HTTP 통신**에서 서버가 `Content-Type: application/json` 헤더와 `{ "id": "..."}` 형태의 JSON 본문을 요구한다면, 클라이언트는 이 형식을 글자 하나 틀리지 않고 그대로 보내야 합니다. 형식이 다르면 서버는 요청을 해석하지 못하고 거부합니다.

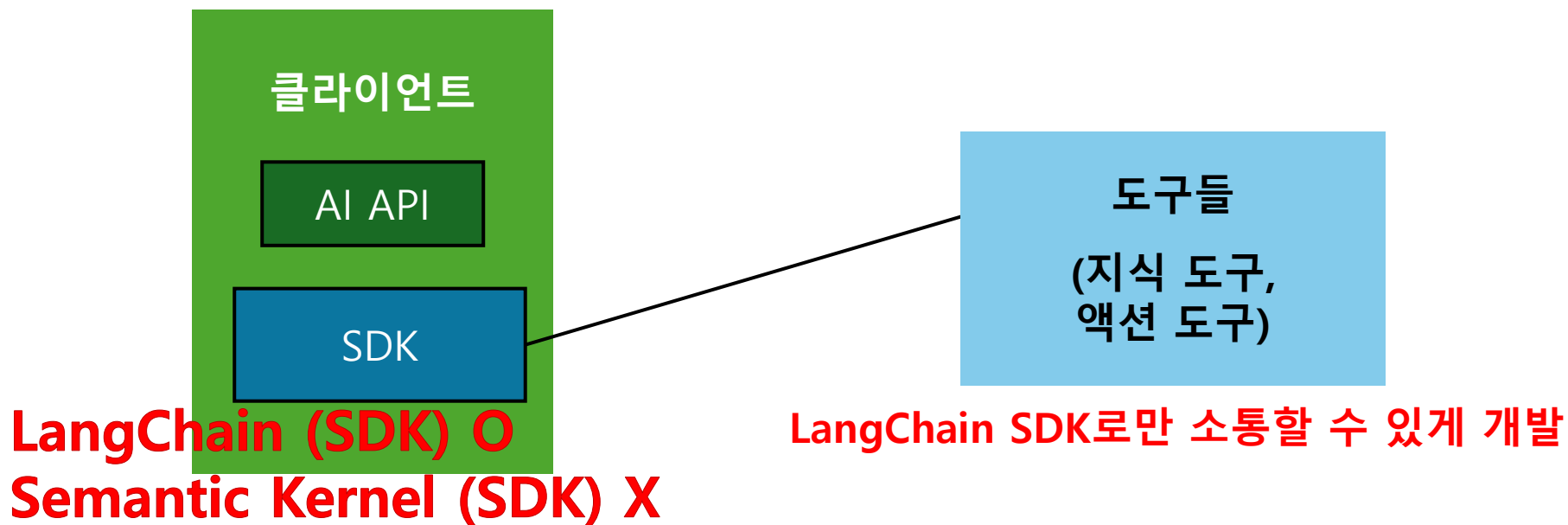
AI의 Tool Calling도 마찬가지입니다. 우리가 AI에게 '이런 도구가 있다'고 알려주는 **JSON 스키마**부터, AI가 우리에게 '이 함수를 실행해달라'고 요청하는 **JSON 구조**, 그리고 우리가 그 실행 결과를 다시 AI에게 돌려주는 **JSON 형식**까지, 이런 '**프로토콜**'에 따라야 합니다.

그리고 이런 프로토콜은 **SDK에 따라 각각 다릅니다**.



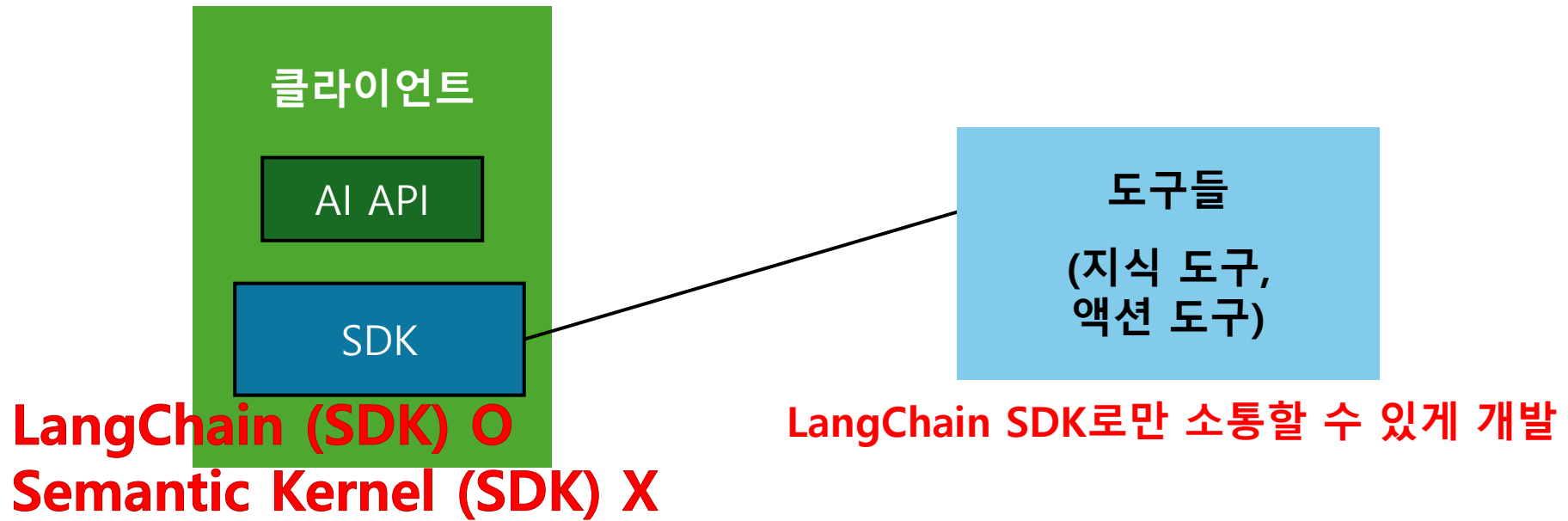
만약 [도구들]이 **LangChain SDK**에 따라 소통할 수 있게 개발되었다면, 이 [도구들]을 호출해 사용하는 AI 어플리케이션(클라이언트)를 만든다고 할 때, **LangChain SDK**를 사용해야 합니다.

만약 클라이언트에서 Semantic Kernel SDK를 사용해서 이 [도구들]을 호출해도 사용할 수가 없습니다.
Semantic Kernel에 의한 호출 내용을 [도구들]이 인식하지 못하기 때문입니다.

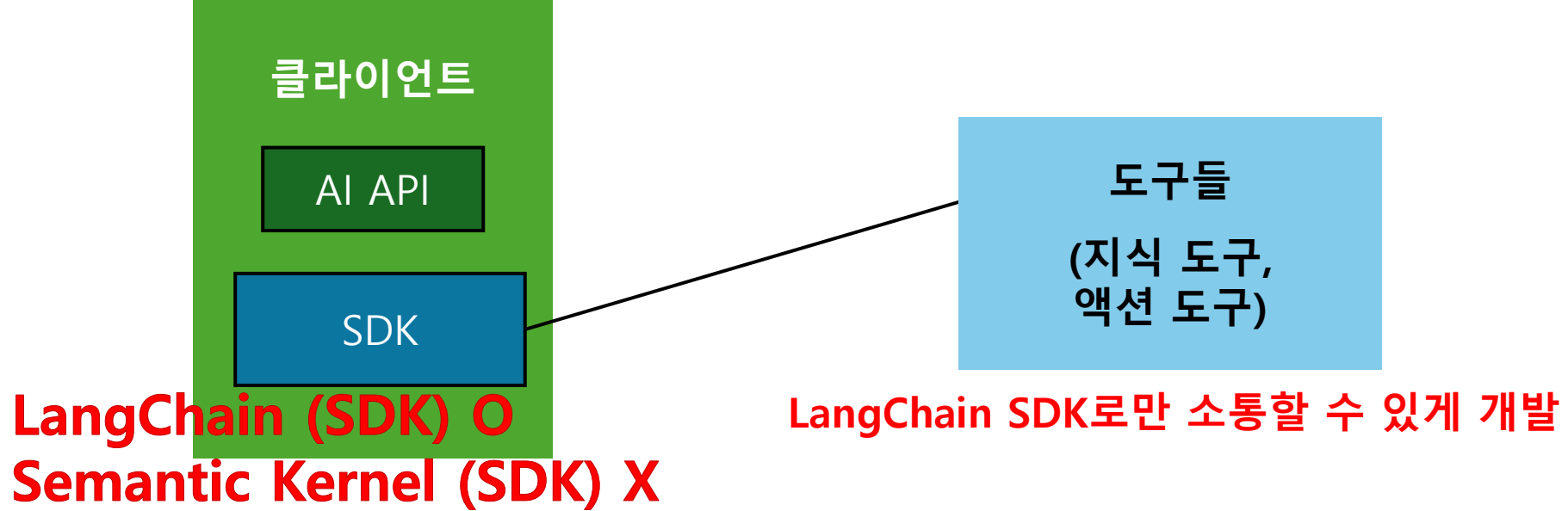


이처럼 LangChain SDK 환경에 맞춰 개발된 [도구들]은 오직 LangChain을 사용하는 애플리케이션에서만 호출할 수 있는 것과 같은 프레임워크(SDK) 종속성이 있습니다.

그리고 동일한 기능을 가진 도구라도 각 프레임워크(SDK)에 맞춰 중복 개발해야 하고, 다른 프레임워크의 유용한 도구를 가져다 쓸 수 없게 된다는 점에서 파편화 문제도 있습니다.



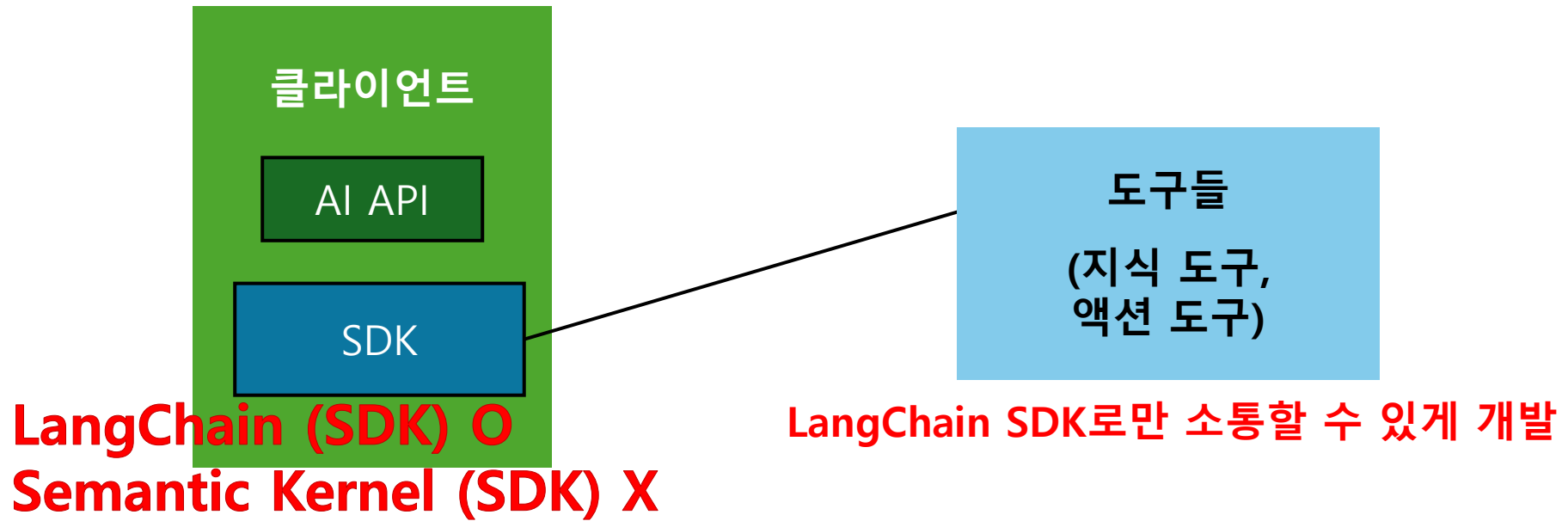
바로 이 프레임워크 종속성과 파편화 문제를 해결하기 위해 제시된 개념이 바로 모델 컨텍스트 프로토콜(Model Context Protocol, MCP) 입니다.



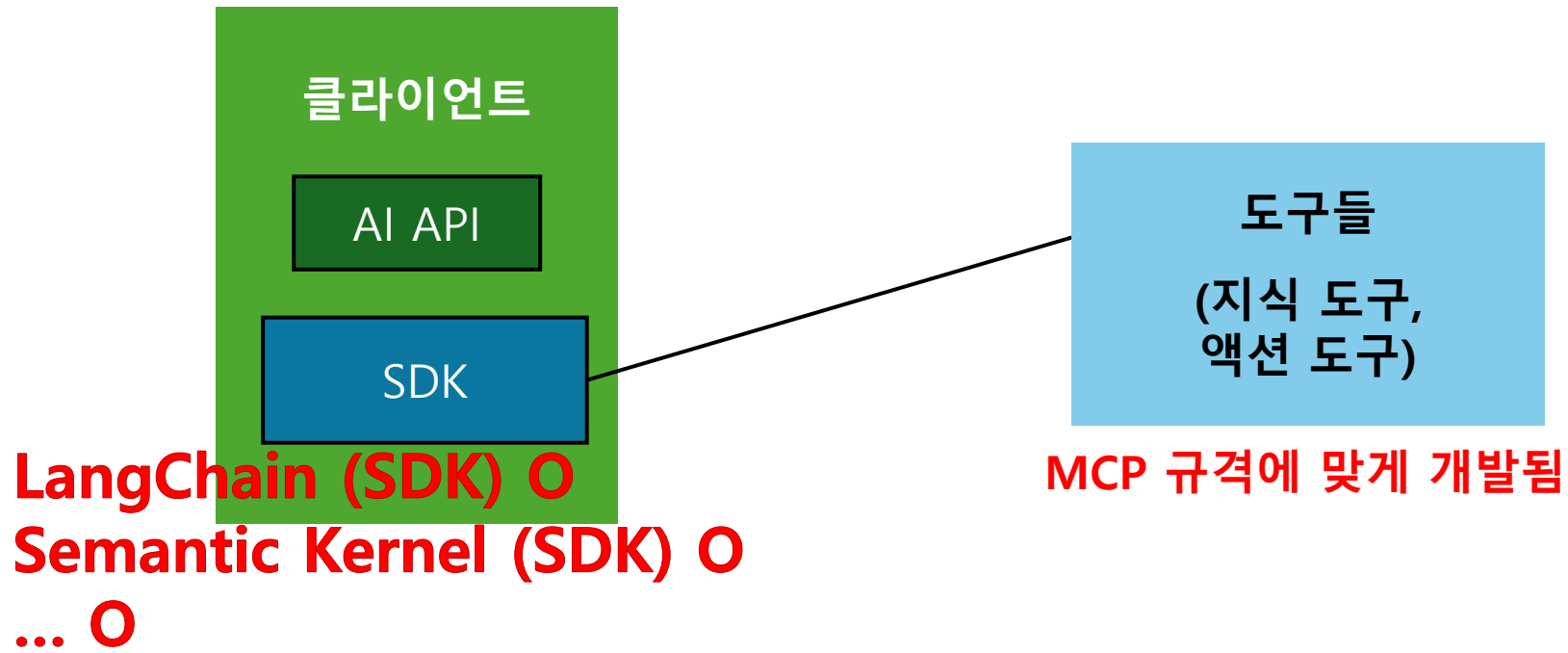
MCP란 한마디로, 'AI 컴포넌트들을 위한 공용 통신 규약(프로토콜)'입니다. 즉, 어떤 프레임워크를 쓰든 '대화 문맥(Context)' 정보를 항상 동일한 JSON 구조로 주고 받자는 사회적 약속과 같습니다.

이는 마치 모든 웹 브라우저와 서버가 HTTP 명세를 따르기 때문에 서로 통신이 가능한 것과 같은 원리입니다. MCP는 바로 AI 생태계의 'HTTP'와 같은 역할을 목표로 하는 셈입니다.

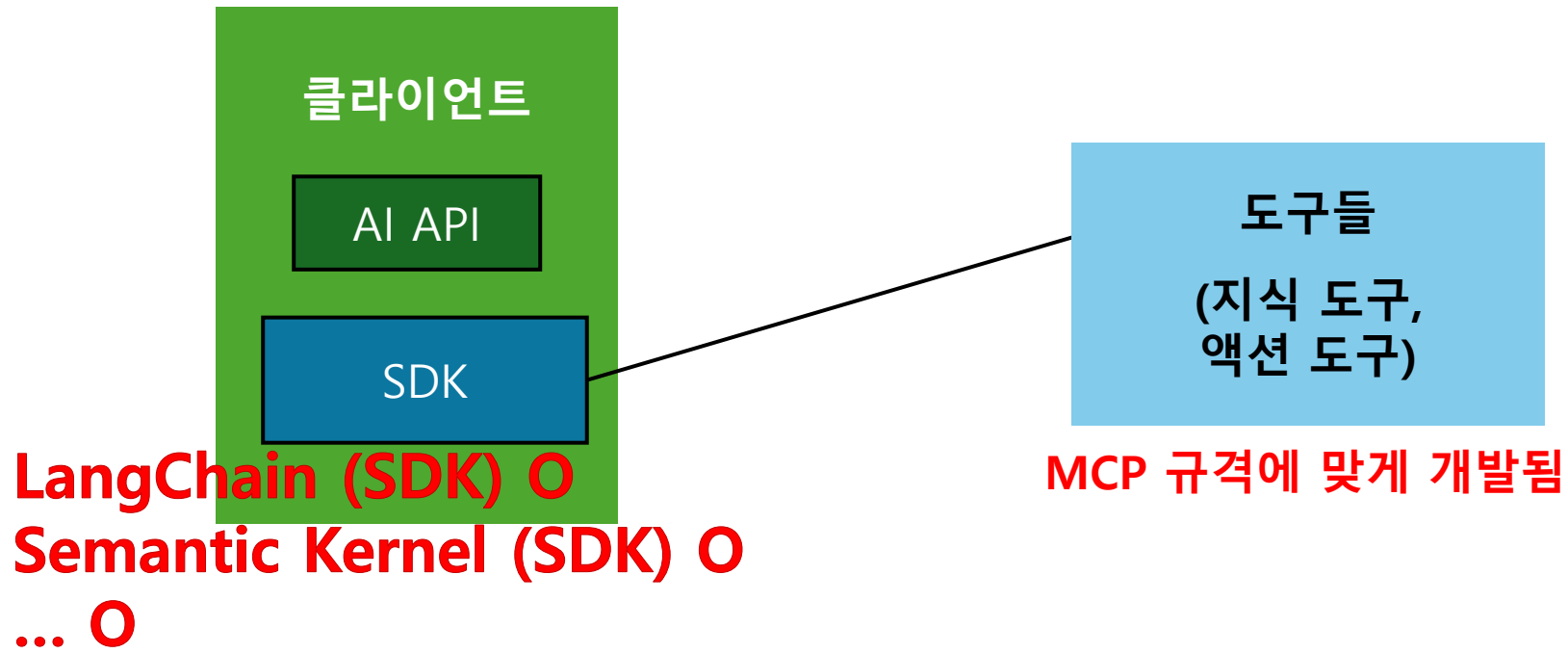
따라서 MCP가 제시하는 비전은 명확합니다. 바로 MCP 규격에 맞춰 개발된 도구는 '어떤 프레임워크에든 구애받지 않고 자유롭게 연결(Plug-in)하여 사용할 수 있게' 하는 것입니다.



기존에는 LangChain SDK로만 소통할 수 있는 [도구들]이 있을 때,
클라이언트는 무조건 LangChain SDK를 사용해야 했습니다.



그러나 MCP 규격에 맞게 개발된 [도구들]이 있다면,
클라이언트에서 LangChain SDK를 사용하든, Semantic Kernel SDK를 사용하든
이 [도구들]과 소통할 수 있습니다.



그리고 이 MCP 규격에 맞춰 도구들을 개발하는 방법은 간단합니다.

@Tool 과 같이 어노테이션을 붙여서 만들 수 있습니다.

더하기 도구, 빼기 도구를 MCP 규격에 맞춰 어떻게 개발할 수 있을까요?

우측에 더하기 기능(add)을 가진 메서드 위에 @Tool이 작성된 것을 볼 수 있습니다.

이처럼 AI가 도구로서 사용하면 좋겠는 기능을 개발한 후, **@Tool과 설명(description..)**만 붙여 주면 됩니다.

```
@Service
public class CalculatorService {

    /**
     * Add two numbers
     * @param a The first number
     * @param b The second number
     * @return The sum of the two numbers
     */
    @Tool(description = "Add two numbers together")
    public String add(double a, double b) {
        double result = a + b;
        return formatResult(a, "+", b, result);
    }

    /**
     * Subtract one number from another
     * @param a The number to subtract from
     * @param b The number to subtract
     * @return The result of the subtraction
     */
    @Tool(description = "Subtract the second number from the first number")
    public String subtract(double a, double b) {
        double result = a - b;
        return formatResult(a, "-", b, result);
    }
}
```

참고)

사실 이 `@Tool` 어노테이션은 컴파일 또는 애플리케이션 시작 시점에 **OpenAPI 명세(Specification)** 문서를 자동으로 생성하는 역할을 합니다.

클라이언트 SDK가 이 자동 생성된 명세서를 보고 '아, 이런 도구(API)가 있구나' 하고 인식하게 되는 것입니다.

그러면 클라이언트에서
LangChain4j든, Semantic Kernel을
쓰던 아까 @Tool로 정의한
도구들을 가져와 쓸 수 있습니다.

```
import dev.langchain4j.model.openai.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

**MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출**

다음 슬라이드에서는

- 클라이언트 <-> AI 모델,
- 클라이언트 <-> MCP 규격에 맞게 정의된 도구 서버

의 동작 과정에 대해 다루고 있습니다.

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();

        // SDK를 통해 AI 연결 및 호출
        // MCP 규칙에 맞게 정의된 지식/액션 도구 연결 및 호출
    }
}
```

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```



내부 동작

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

채팅을 입력받았을 때
어떻게 클라이언트, AI 모델, 도구들
이 동작할까요? 차례대로 살펴보겠습니다

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openai.OpenAiChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {
```

SDK를 통해

(2) AI 모델의 판단 및 실행 요청

- AI 모델은 채팅으로 입력된 "계산기 사용해서 3 더하기 4 결과 알려줘"라는 요청을 분석하고, 클라이언트로부터 전달받은 도구 설명문(description = "Add two numbers together")을 참고함.

- 자신의 능력만으로는 계산을 못 하지만, Add라는 도구가 이 요청을 해결할 수 있음을 파악함.

- AI 모델은 최종 답변을 하는 대신, a=3, b=4라는 인자와 함께 Add 함수를 호출하라는 **실행 요청(Tool Call)**을 클라이언트 앱으로 다시 보냅니다.

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

AI 모델

내부 동작

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

AI 모델

내부 동작

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

(3) 클라이언트 앱 → 도구 서버로 실행 요청

• AI로부터 Add 함수 실행 요청을 받은 클라이언트 앱은, 자신이 직접 함수를 실행하는 것이 아닙니다.

• 대신, MCP 규칙에 따라 '**도구 서버**' (<http://host.docker.internal:8080>)로 "Add 함수를 a=3, b=4 인자로 실행해줘" 라는 요청을 보냄.

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

AI 모델

(4) 도구 서버의 함수 실행 및 응답

요청을 받은 도구 서버는 자신이 가지고 있는 Add(3, 4) 코드를 실행합니다.

계산 결과인 '7'을 클라이언트 앱으로 다시 응답해줍니다.

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

AI 모델

채팅(계산기 사용해서 3 더하기 4 결과

(5) 클라이언트 앱 → AI 서버로 결과 보고

클라이언트 앱은 도구 서버로부터 '7'이라는 결과를 받았습니다.

이제 이 결과를 다시 AI 서버에게 전달하며 "아까 요청했던 계산 결과는 '7'이었어" 라고 보고합니다.

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

AI 연결 및 호출

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```

AI 모델

AI 서버

AI 모델

(6) 최종 답변 생성 및 표시

AI 서버는 최종 결과를 바탕으로

"3 더하기 4의 결과는 7입니다."와 같은 자연스러운 답변을 만들어 클라이언트 앱에 전달하고, 클라이언트 앱은 이를 사용자에게 보여줍니다.

채팅(계산기 사용해서 3 더하기 4 결과)

(1) 클라이언트 앱이 채팅을 입력 받음

```
import dev.langchain4j.model.openaiofficial.OpenAiOfficialChatModel;
import dev.langchain4j.service.AiServices;
import dev.langchain4j.service.tool.ToolProvider;

import java.time.Duration;
import java.util.List;

public class LangChain4jClient {

    public static void main(String[] args) throws Exception {

        ChatLanguageModel model = OpenAiOfficialChatModel.builder()
            .isGitHubModels(true)
            .apiKey(System.getenv("GITHUB_TOKEN"))
            .timeout(Duration.ofSeconds(60))
            .modelName("gpt-4.1-nano")
            .build();

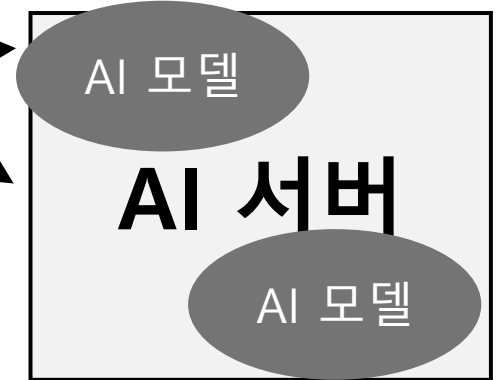
        McpTransport transport = new HttpMcpTransport.Builder()
            .sseUrl("http://host.docker.internal:8080/sse")
            .timeout(Duration.ofSeconds(60))
            .logRequests(true)
            .logResponses(true)
            .build();
```

SDK를 통해

결과:
"3 더하기 4의 결과는 7입니다."

MCP 규칙에 맞게 정의된
지식/액션 도구 연결 및 호출

```
* @return The sum of the two numbers
*/
@Tool(description = "Add two numbers together")
public String add(double a, double b) {
    double result = a + b;
    return formatResult(a, "+", b, result);
}
```



내부 동작

채팅(계산기 사용해서 3 더하기 4 결과 알려줘)

이러한 과정들을 통해 답을 받을 수 있습니다

요약

문제: AI가 사용할 수 있는 도구 개발은 LangChain, Semantic Kernel 등 프레임워크별로 파편화되어 있었습니다. 도구가 LangChain만 호환된다면 도구를 사용하는 클라이언트도 LangChain으로 개발되어야 했습니다.

해결책: MCP는 HTTP처럼 '공통 통신 규약'을 제공하여 이 문제를 해결합니다.

구현: OpenAPI 기반의 표준 덕분에 @Tool 어노테이션만으로 쉽게 개발할 수 있습니다.

효과: '한 번 개발하면 어디서든 실행되는' 재사용성 높은 AI 도구 생태계가 가능해집니다.

감사합니다

문의 및 질문: mjpark@ikss.kr