

WEEK 02

함께하는 AI 에이전트 개발

2주차: "웹 검색 기능을 갖춘 리서치 어시스턴트" 완성



PROJECT GOAL

리서치 어시스턴트



DURATION

180 Mins (3h)



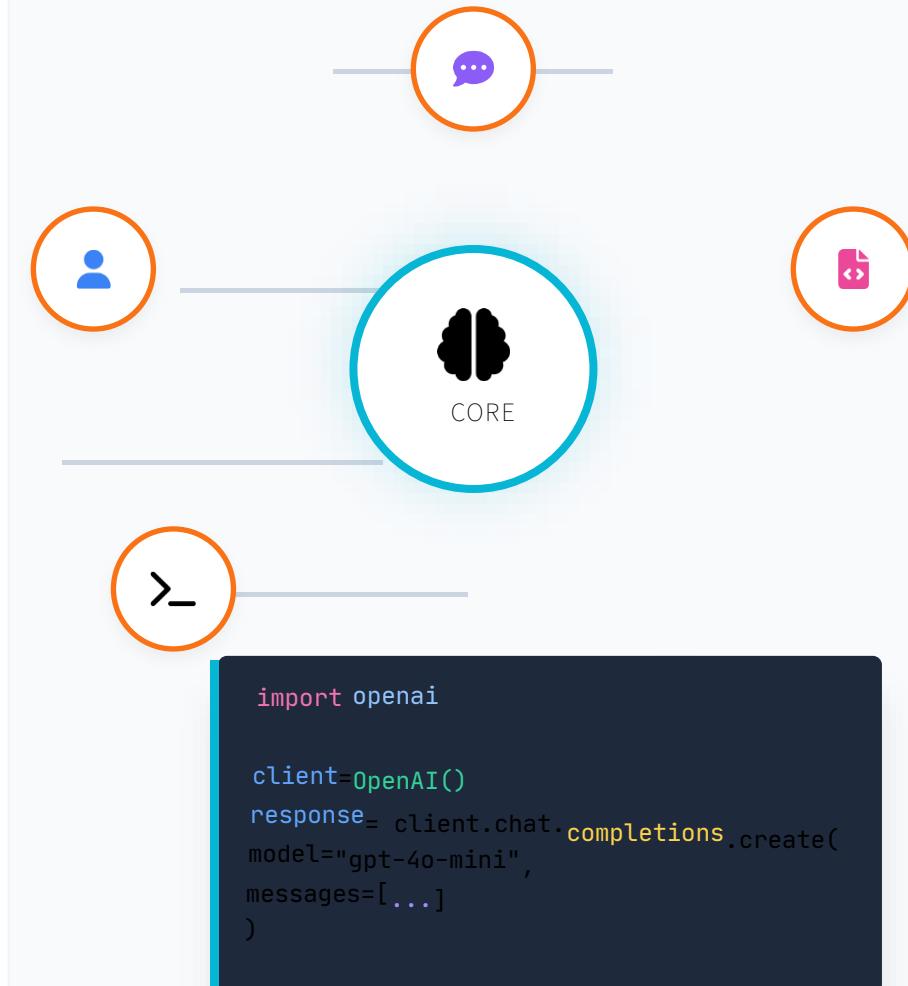
CORE STACK

Python, OpenAI API



INSTRUCTOR

Thomas Moon



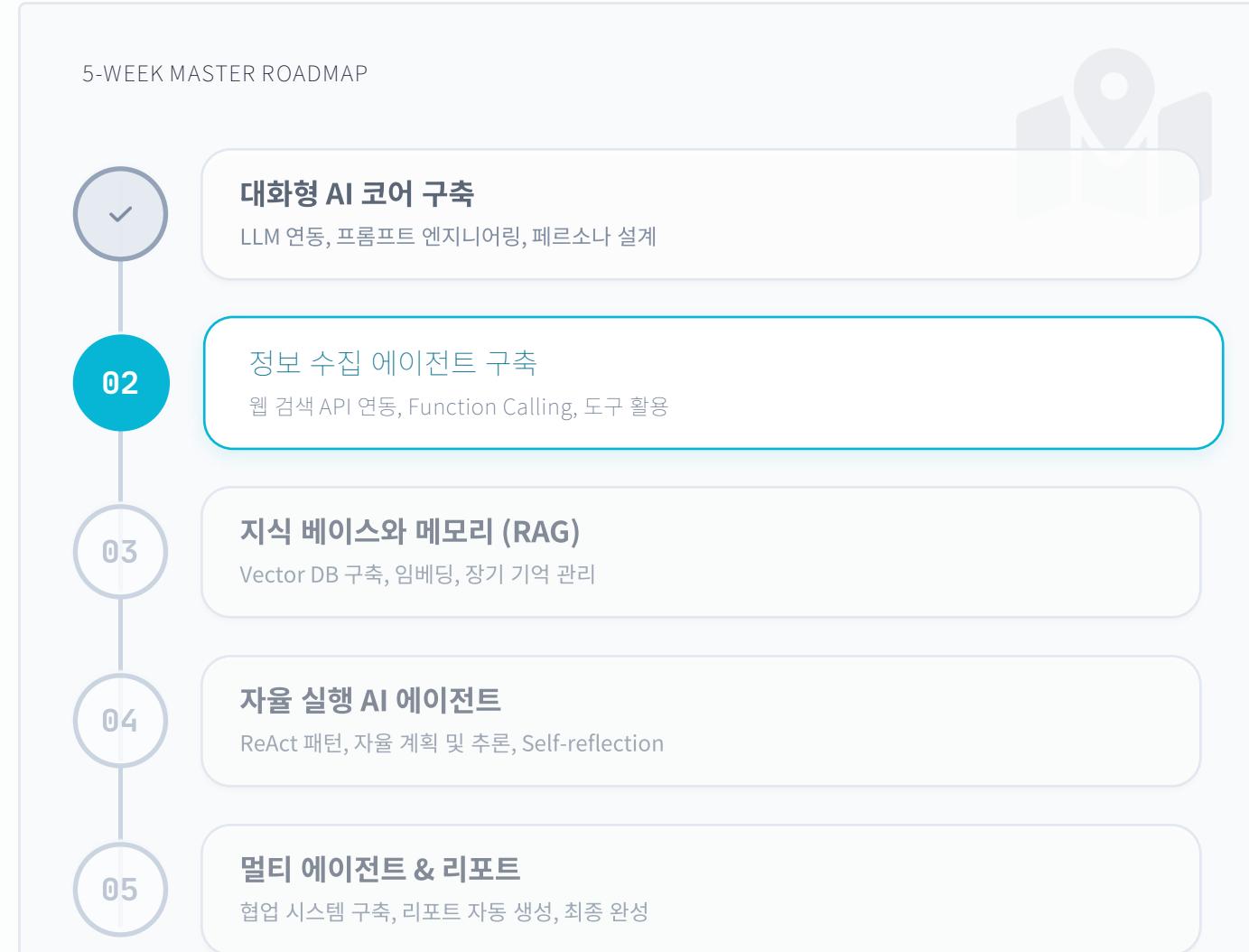
Today's Agenda & Roadmap

이번 시간의 학습 목표와 전체 과정을 조망합니다.

TODAY: WEEK 02

정보 수집 에이전트 구축

- 지난 주 복습 및 개선
- 이론: Tool-using Agent
Function Calling & Tool Use 메커니즘
- 실습 1: 웹 검색 도구 통합
Tavily Search API 연동
- 실습 2: 대화 관리자와 통합
검색 결과 파싱 및 응답 생성
- LangChain 프레임워크 소개
- 마무리 및 다음 주 예고



01

SECTION

SECTION 01

지난 주 복습 및 개선

1주차 성과 점검과 개선 포인트 확정

☰ OBJECTIVES & CHECKPOINTS

- ✓ 코어 구성요소 리뷰 (Core)
- ✓ 기능 부재 분석 (Tools)
- ✓ 한계 진단 (Knowledge Cutoff)
- ✓ 이어지는 과제 조망

KEYWORDS:

● Review

● Refactor

● Next Step

1주차 복습: 완성 시스템과 한계

지난 주 성과 점검 및 이번 주 개선 목표

핵심 구성요소 (COMPONENTS)

- ConversationManager: 대화 흐름 제어 및 히스토리 관리
- System Message: 리서치 어시스턴트 페르소나 정의
- State Management: Idle → Responding 처리

SYSTEM EVOLUTION



WK 01: CORE ENGINE

Dialogue & Intent

⚠ 현재 시스템의 한계 (LIMITATIONS)

- Knowledge Cutoff 학습 데이터 이후 정보 부재
- No Tools 실시간 웹 검색 등 외부 도구 활용 불가
- Reliability 출처(Source)가 없어 신뢰성 검증 어려움

◉ 이번 주 해결 목표 (GOAL)

"외부 검색 도구(Tavily)를 연동하여 최신 정보를 획득하고, 명확한 근거(출처)를 제시하는
에이전트 완성"



WK 02: WEB SEARCH

Real-time Information



ENHANCED RESPONSE

Answer + Citations

Troubleshooting & 개선

자주 발생하는 이슈 및 해결 방안

Q&A

02

SECTION

THEORY

AI FUNDAMENTALS

이론 : Tool-using Agent

AI가 외부 도구를 사용하여 정보 한계를 극복하는 원리

KEY TOPICS

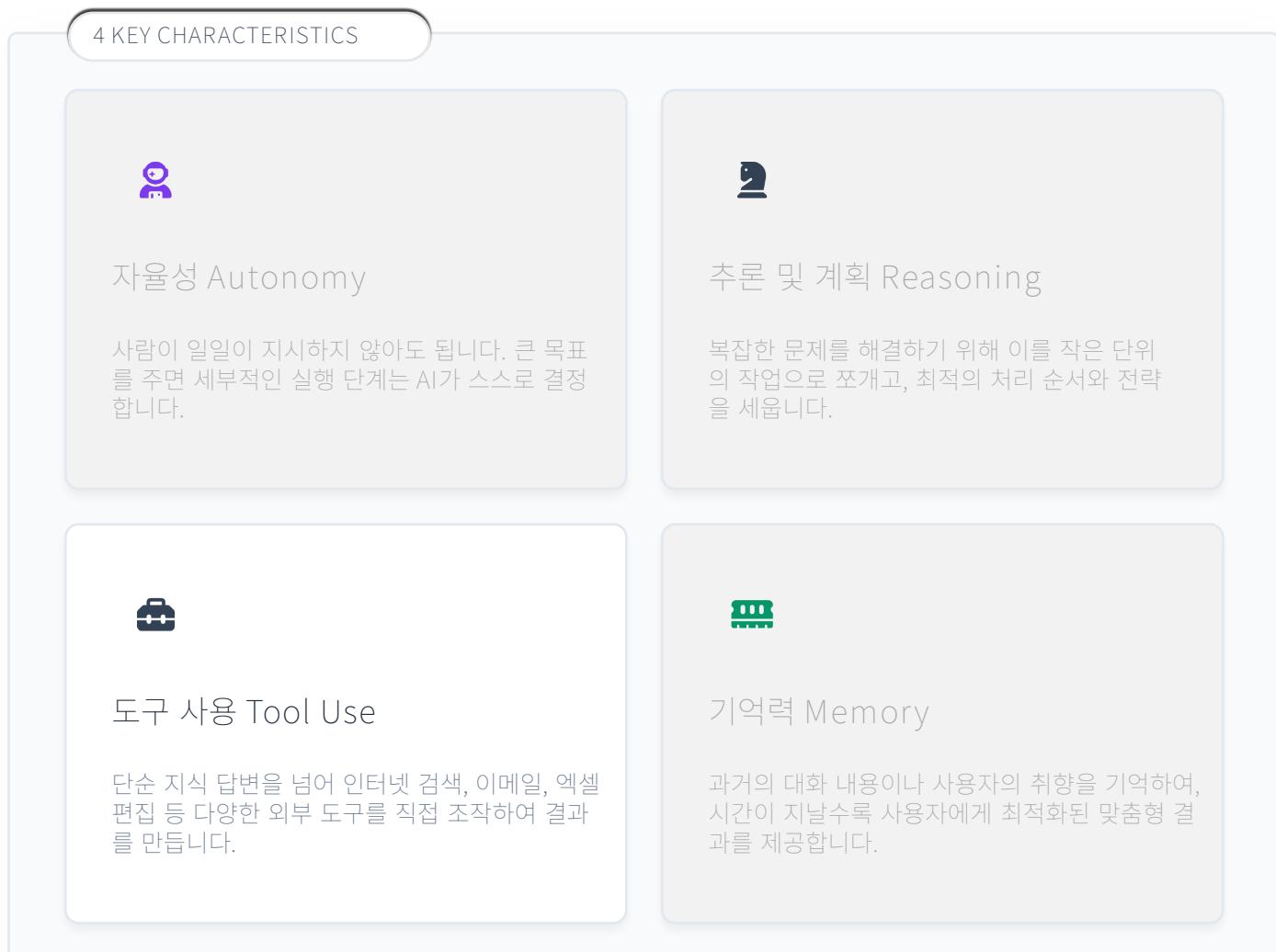
- ✓ Tool Use의 핵심 개념과 흐름
- ✓ OpenAI API 구조 및 응답 처리
- ✓ Function Calling 메커니즘
- ✓ Tavily 검색 API 소개 및 연동

AI 에이전트 특성 (복습)

단순 응답을 넘어 목표를 완수하는 시스템

행동하는 AI

일반적인 AI는 정보 제공에 그치지만, AI 에이전트는 복잡한 명령을 수행하기 위해 도구를 사용하고 스스로 행동 과정을 결정합니다.



Tool Use란 무엇인가?

AI 에이전트의 핵심: 외부 도구 활용 능력

🤖 AI가 도구를 사용한다는 것

단순히 텍스트를 생성하는 것을 넘어, 자신의 능력 한계를 인지하고 필요한 시점에 적절한 외부 도구(API, 코드 등)를 호출하여 문제를 해결하는 에이전트의 핵심 능력입니다.

↔️ 인간 VS AI 도구 사용 비교

인간 (Human)

↳ 망치로 못을 박음

↳ 계산기로 계산

↳ 인터넷 검색

↳ 이메일 전송

AI 에이전트 (Agent)

API로 데이터를 가져옴

코드 실행으로 계산

Search API 호출

Email API 호출

TOOL USE WORKFLOW

LLM (두뇌)

"이 질문에 답하려면 최신 정보가 필요해. 웹 검색 도구를 사용해야겠어."

TOOL SELECTION



Search



Email



Code



Data

사용 가능한 도구 중 가장 적합한 도구 선택



TOOL EXECUTION

```
search ("2024 AI trends")
→ [결과 반환: JSON Data]
```

Function Calling 개념

LLM이 도구를 '호출'하는 방법

“

"LLM이 사용자의 요청을 분석하여, 필요한 함수(도구)와 그 인자(파라미터)를 JSON 형식으로 출력하는 기능"

★ 핵심 포인트



구조화된 출력 (Structured Output)

모호한 자연어를 기계가 실행 가능한 정형 데이터로 변환



외부 세계 연결 (Bridge)

고립된 LLM이 API, DB, 코드 실행기 등과 상호작용 가능



결정론적 실행 (Execution)

추상적인 '의도'를 구체적인 '함수 호출'로 매핑



OpenAI Function Calling 구조

API STRUCTURE: OpenAI의 Tool Use 구현

TOOLS 파라미터 정의

type: 도구의 타입 (여기서는 함수)

name: 함수의 이름

Description: 설명

중요! LLM이 이 설명을 읽고 언제 이 도구를 사용할지 판단

parameters - 함수가 받는 인자 (JSON Schema)

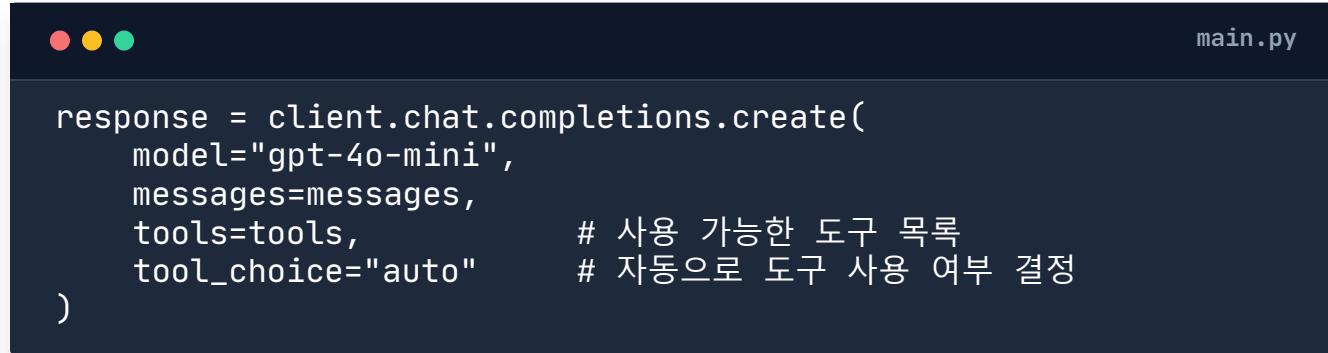
```
● ● ● tool_definitions.py

tools = [
{
    "type": "function",
    "function": {
        "name": "search_web",
        "description": "웹에서 최신 정보를 검색합니다",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "검색할 키워드나 질문"
                },
                "num_results": {
                    "type": "integer",
                    "description": "반환할 결과 수",
                    "default": 5
                }
            },
            "required": ["query"]
        }
    }
}
```

OpenAI Function Calling 구조

API STRUCTURE: OpenAI의 Tool Use 구현

☞ API 호출 및 제어



The screenshot shows a dark-themed code editor window titled "main.py". The code is as follows:

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
    tools=tools,          # 사용 가능한 도구 목록
    tool_choice="auto"    # 자동으로 도구 사용 여부 결정
)
```

The code uses the OpenAI API to create a completion for a GPT-4o-mini model, specifying messages, available tools, and an automatic tool selection strategy.

참고: <https://platform.openai.com/docs/guides/function-calling>

Function Calling 응답 처리

🔍 응답 객체 핵심 필드

속성 (Property)	설명 (Description)
tool_calls	LLM이 생성한 도구 호출 목록 (List) * tool_calls가 있으면 도구 실행 필요
function.name	실행할 함수의 이름 (String) 예: "search_web"
function.arguments	함수에 전달할 인자 (JSON String) 파싱(json.loads) 후 사용
tool_call_id	각 호출의 고유 ID 결과 반환 시 매핑에 필수

도구 실행 결과는 반드시 tool_call_id와 함께 LLM에 다시 전달해야 최종 답변을 생성할 수 있습니다.



response_handler.py

```
message = response.choices[0].message

if message.tool_calls:
    for tool_call in message.tool_calls:
        function_name = tool_call.function.name
        arguments = json.loads(tool_call.function.arguments)
        if function_name == "search_web":
            result = search_web(**arguments)
        messages.append({
            "role": "tool",
            "tool_call_id": tool_call.id,
            "content": json.dumps(result)
        })
final_response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages
)
```

Arguments 형태 (예시)

```
{"location": "Seoul, Korea", "units": "celsius"}
```

**은 딕셔너리 언패킹(Dictionary Unpacking)

예) Search web = (location = "Seoul, Korea", unit = "celsius")

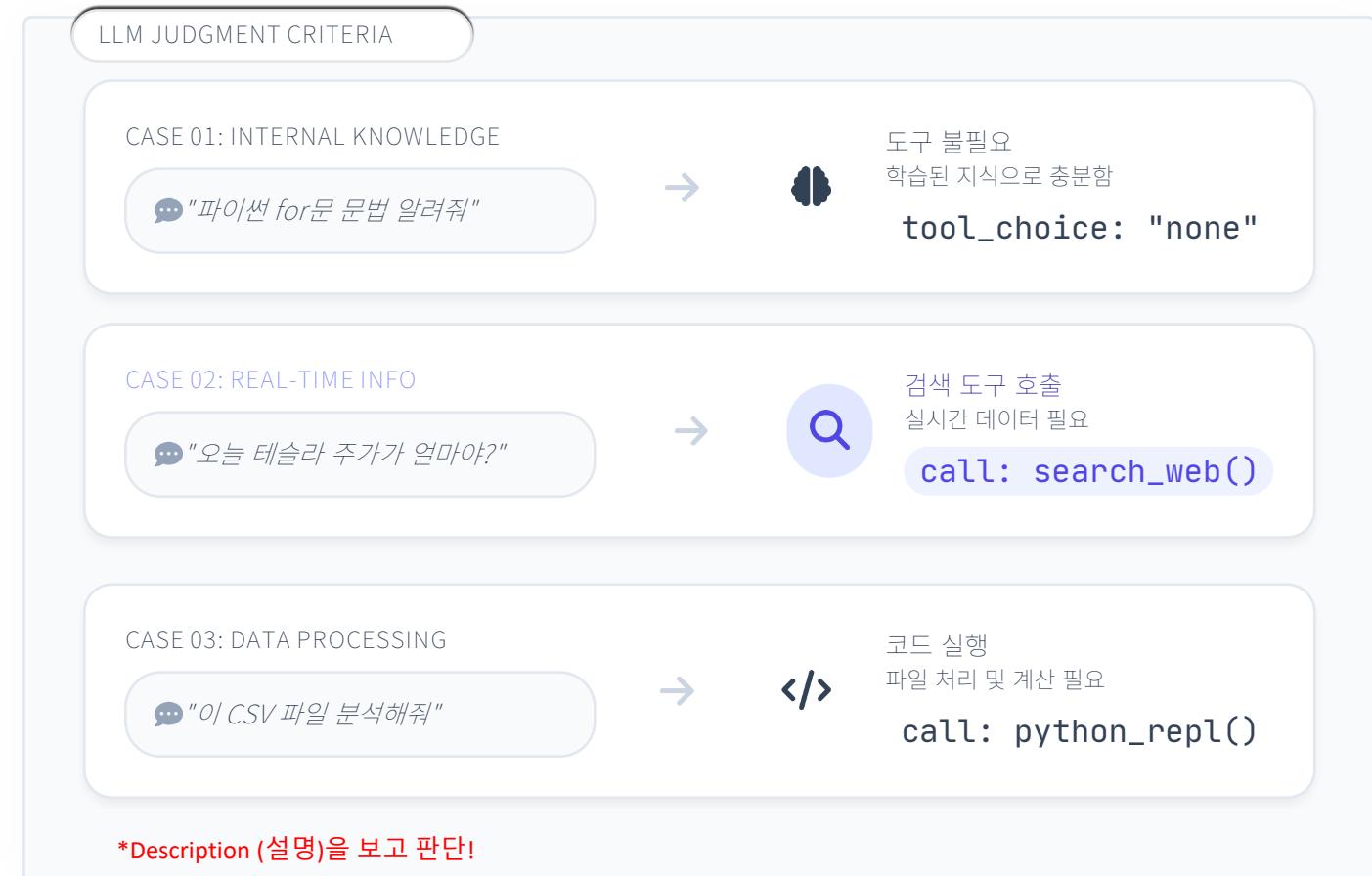
도구 선택 메커니즘

에이전트가 도구를 '선택'하는 원리

tool_choice 옵션

API 호출 시 tool_choice 파라미터를 통해 에이전트의 행동을 제어할 수 있습니다.

Option	설명 및 사용 시점
"auto"	자동 판단 (권장) LLM이 대화 맥락에 따라 도구 사용 여부를 스스로 결정합니다.
"none"	사용 안 함 도구를 강제로 끄고 일반 대화 모드로만 응답하게 합니다.
"required"	필수 사용 어떤 도구든 반드시 하나 이상 사용하도록 강제합니다



도구 설계 원칙

좋은 도구 정의를 위한 4가지 가이드라인

❶ 식별과 설명 (Identity)

1. 명확한 이름 (Clear Naming)

"do_something" 같은 모호한 이름 대신, 기능이 직관적으로 드러나는 구체적인 이름(search_web_news)을 사용하세요.

2. 상세한 설명 (Description)

LLM이 언제, 어떤 목적으로 이 도구를 사용해야 하는지 명확히 이해할 수 있도록 구체적인 가이드를 포함하세요.

3. 단일 책임 (Single Responsibility)

하나의 도구는 하나의 명확한 기능만 수행해야 합니다. 복잡한 기능은 여러 도구로 분리하세요.

✓ NAMING CONVENTION

BAD

```
"name": "do_stuff"
```

GOOD

```
"name": "search_web_news"
```

✓ DESCRIPTION DETAIL

BAD

```
"description": "검색합니다"
```

GOOD

```
"description":  
"최신 뉴스와 정보를 검색합니다. 실시간 데이터 질문에 사용하세요."
```

도구 설계 – Tools 정의 예시

```
tools = [
    {
        "type": "function",
        "name": "search_news_articles",
        "description": (
            "Searches news articles from the internal database "
            "based on keyword, time range, and language."
        ),
        "strict": True,
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "Search keyword or phrase."
                },
                "time_range": {
                    "type": "string",
                    "enum": ["day", "week", "month", "year"],
                    "description": "Search time range filter."
                },
                "required": [
                    "query",
                    "time_range"
                ],
                "additionalProperties": False
            }
        }
    }
]
```

⚙️ 명세와 구조 (Specification)

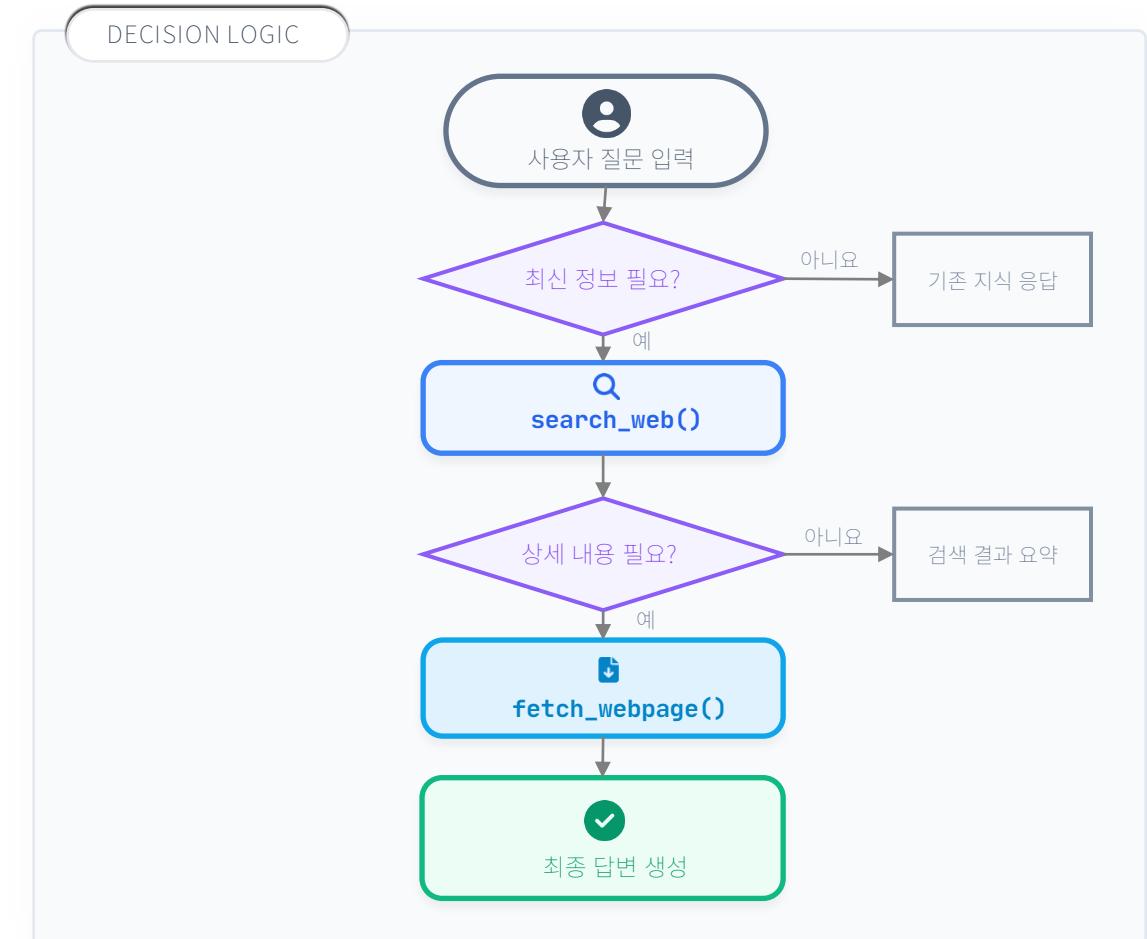
파라미터 명세 (Parameters)

인자의 타입(Type), 필수 여부(Required), 가능한 값의 범위(Enum) 등을 상세히 정의하여 오류를 방지하세요.

우리 프로젝트에 필요한 도구들

이번 주 구현 (WEEK 02)		PRIORITY: HIGH
		UPCOMING
	<code>search_web</code>	Tavily API
	<code>fetch_webpage</code>	BS4 / Tavily
향후 추가 예정 (ROADMAP)		
	<code>analyze_data</code> 데이터 분석 및 시각화	W3
	<code>save_to_knowledge</code> 벡터 DB에 정보 저장	W3
	<code>generate_report</code> 리포트 자동 생성	W5

리서치 어시스턴트를 위한 도구 설계 및 로드맵



Tavily Search API 소개

AI 에이전트를 위해 설계된 차세대 검색 도구

“WHAT IS TAVILY?”

"AI 에이전트와 LLM(Large Language Model)을 위해 특별히 설계되어, 최적화된 검색 결과와 요약 콘텐츠를 제공하는 검색 API"

KEY FEATURES

- AI-Optimized Results: LLM이 이해하기 쉬운 구조화된 JSON 응답
- Content Extraction: 웹페이지의 핵심 본문 내용 자동 추출 및 요약
- Source Attribution: 신뢰성 확보를 위한 원본 URL 및 출처 제공
- Real-time Data: 실시간 최신 뉴스 및 이벤트 정보 검색
- Simple Integration: Python/LangChain 등 간편한 SDK 연동

기존 검색 API vs Tavily 비교

특성	GOOGLE SEARCH API	TAVILY API
주요 대상	일반 사용자 / 개발자	AI 에이전트 / LLM
결과 형식	URL + 짧은 스니펫	요약 답변 + 핵심 본문
후처리	추가 스크래핑/파싱 필요	즉시 사용 가능 (Ready)
LLM 친화성	낮음 (잡음 많음)	높음 (토큰 효율적)
비용 정책	유료 (복잡한 할당량)	무료 티어 제공 (월 1,000회)

Why Tavily?

RAG(검색 증강 생성) 구현 시, 별도의 크롤러를 구축할 필요 없이 API 하나로 검색+수집+정제를 한 번에 해결할 수 있습니다.

Tavily API 응답 구조

Tavily 검색 결과 포맷 이해하기

필드 (FIELD)	설명 (DESCRIPTION)	활용 (USAGE)
answer	AI가 생성한 요약 답변	빠른 응답 생성
results	검색 결과 리스트 (Array)	상세 정보 및 참조
content	웹 페이지 핵심 본문	LLM Context 주입
score	관련성 점수 (0.0 ~ 1.0)	결과 필터링/정렬
url	원본 페이지 링크	출처(Source) 표기



개발 팁 (Tip)

LLM에 전체 HTML을 넣는 대신, Tavily가 추출한 content 필드를 사용하면 토큰 비용을 절약하고 할루시네이션을 줄일 수 있습니다.

response_example.json

```

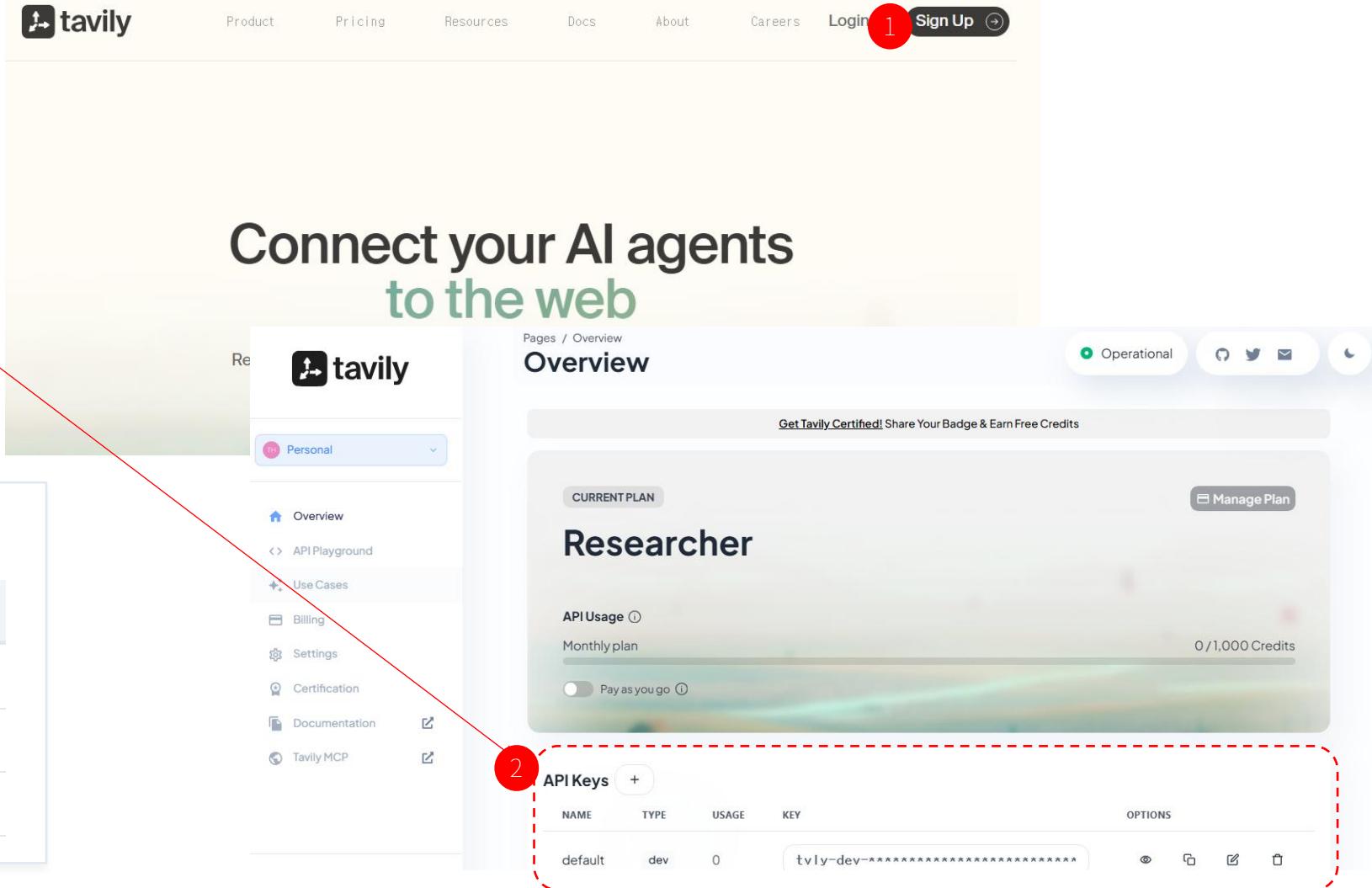
1 {
2   "query" :"AI agent trends 2024" ,
3   "answer" :"2024년 AI 에이전트 트렌드는 자율성 향상, 멀티모달..." ,
4   "results" :[
5     {
6       "title" :"The Rise of AI Agents in 2024" ,
7       "url" :"https://example.com/ai-agents-2024" ,
8       "content" :"AI agents are becoming more autonomous..." ,
9       "score" :0.95 ,
10      "published_date" :"2024-01-15"
11    },
12    {
13      "title" :"Building Effective AI Agents" ,
14      "url" :"https://example.com/building-agents" ,
15      "content" :"Key components of modern AI agents include..." ,
16      "score" :0.89 ,
17      "published_date" :"2024-01-10"
18    }
19  ],
20  "response_time" :1.23

```

Tavily API 키 발급 및 설정

다음 실습을 위한 준비 (Preparation)

1. <https://www.tavily.com> 접속
회원가입 후, 로그인



The screenshot shows the Tavily website's homepage. At the top right, there is a red circle with the number '1' over the 'Login' button. A red arrow points from this circle down to the 'API Keys' section on the right side of the page. Another red circle with the number '2' is placed over the 'API Keys' heading.

Connect your AI agents to the web

Pages / Overview

Overview

Get Tavily Certified! Share Your Badge & Earn Free Credits

CURRENT PLAN

Researcher

API Usage ⓘ

Monthly plan

0 / 1,000 Credits

API Keys +

NAME	TYPE	USAGE	KEY	OPTIONS
default	dev	0	tavly-dev-*****	

! 무료 티어 제한 (Free Tier)

항목	제한 사항
월간 요청 수	1,000회 / 월
요청당 결과	최대 10개
검색 깊이	basic / advanced

03

SECTION

 HANDS-ON  SEARCH SYSTEM

실습 Part 1 - 웹 검색 도구 통합

검색 엔진 구축 및 Tavily API 연동

HANDS-ON GOALS

- ✓ Tavily API 키 설정
- ✓ 웹 검색 및 결과 파싱 구현
- ✓ SearchAgent 클래스 설계
- ✓ 쿼리 최적화 로직 적용

KEYWORDS:

 Hands-on

 Tavily API

 Python

실습 목표 - 검색 에이전트 구축

달성 과제



Tavily Search API 클라이언트를 프로젝트에 연동하고 초기화합니다.



검색 로직 구현

사용자의 자연어 질문을 효과적인 검색 쿼리로 변환하여 실행합니다.



결과 파싱

비정형 검색 결과를 LLM이 이해하기 쉬운 구조화된 형식으로 정리합니다.



출처 관리

신뢰성 확보를 위해 참조 URL 및 메타데이터 정보를 체계적으로 관리합니다.

</> 실행 코드

main.py (Preview)

```
# SearchAgent 클래스 초기화
search_agent=SearchAgent()

# 검색 수행 (Web Search Tool)
results=search_agent.search("2024년 AI 에이전트 동향")

# 결과 및 출처 출력
print(results.summary)# 요약 내용
print(results.sources)# 출처 목록
print(results.raw_data)# 원본 데이터
```

* 이 코드를 실행하면 실제 웹 검색 결과가 반환됩니다.

프로젝트 구조 업데이트

+ 검색 에이전트 모듈 추가

search_agent.py NEW Tavily API 연동 및 검색 로직을 담당하는 핵심 클래스

src/tools/ NEW 확장성을 고려한 도구 전용 패키지 (tool_definitions, web_search)

settings.py UPDATE 환경변수 로드 및 전역 설정 관리 로직 추가

.env UPDATE Tavily API 키 추가 (TAVILY_API_KEY)

requirements.txt UPDATE 웹 검색 기능을 위한 tavily-python 라이브러리 추가

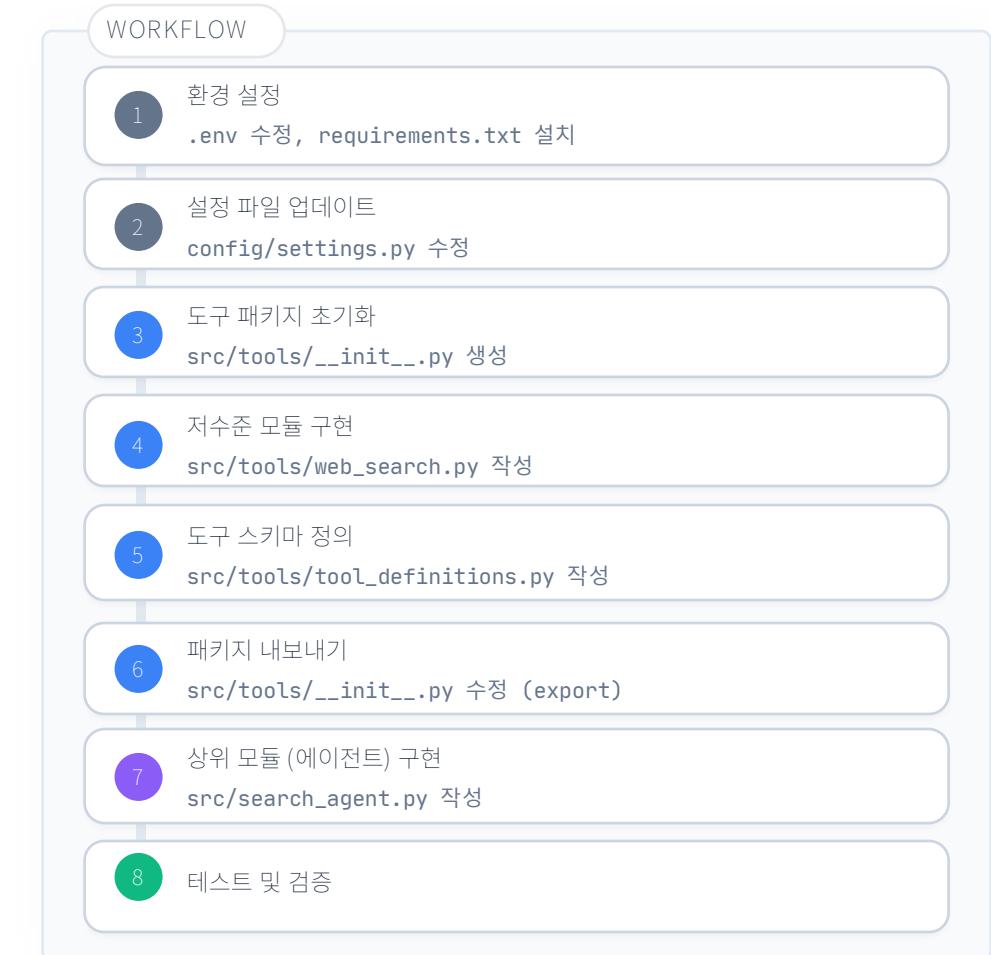
```
● ○ ● ~/ai-research-assistant/2주차 풀더 구조
ai-research-assistant/
├── venv/
├── src/
│   ├── __init__.py
│   ├── conversation_manager.py
│   └── search_agent.py  NEW
│       └── tools/  NEW
│           ├── __init__.py
│           ├── web_search.py  NEW
│           └── tool_definitions.py  NEW
└── config/
    ├── prompts.py
    └── settings.py  UPDATE
└── data/
    ├── .env  UPDATE
    └── requirements.txt  UPDATE
    └── main.py
    └── README.md
```

작업 순서 및 파일 구조

단계별 가이드

Part 1 생성/수정 파일 목록

파일 (PATH)	작업	설명
<code>src/tools/__init__.py</code>	NEW	도구 패키지 초기화 (모듈화)
<code>src/tools/web_search.py</code>	NEW	Tavily API 직접 호출 함수 구현
<code>src/tools/tool_definitions.py</code>	NEW	OpenAI Function Calling 스키마 정의
<code>src/search_agent.py</code>	NEW	SearchAgent 클래스 (오케스트레이션)
<code>config/settings.py</code>	EDIT	Tavily 관련 설정 변수 추가
<code>.env</code>	EDIT	TAVILY_API_KEY 환경변수 추가
<code>requirements.txt</code>	EDIT	tavily-python 패키지 의존성 추가



1. 환경설정 - Tavily 라이브러리 설치

설정 및 환경변수

.env UPDATE Tavily API 키 추가 (TAVILY_API_KEY)



.env

```
# Tavily API Key 추가
TAVILY_API_KEY=tvly-xxxxx...
```

의존성 패키지

requirements.txt UPDATE 웹 검색 기능을 위한 tavily-python 라이브러리 추가



requirements.txt

```
# 2주차 추가 패키지
tavily-python>=0.3.0
```

가상환경 활성화 후, 의존성 패키지 업데이트

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> pip install -r requirements.txt

```
Downloading tavily_python-0.7.19-py3-none-any.whl (18 kB)
Downloading tiktoken-0.12.0-cp314-cp314-win_amd64.whl (921 kB)
921.1/921.1 kB 10.3 MB/s 0:00:00
Downloading regex-2026.1.15-cp314-cp314-win_amd64.whl (280 kB)
Using cached requests-2.32.5-py3-none-any.whl (64 kB)
Downloading charset_normalizer-3.4.4-cp314-cp314-win_amd64.whl (107 kB)
Downloading urllib3-2.6.3-py3-none-any.whl (131 kB)
Installing collected packages: urllib3, regex, charset_normalizer, requests,
tiktoken, tavily-python
Successfully installed charset_normalizer-3.4.4 regex-2026.1.15 requests-
2.32.5 tavily-python-0.7.19 tiktoken-0.12.0 urllib3-2.6.3
```

1. 환경설정 - Tavily API 연동 테스트

```
test_tavily.py

import os
import json
from dotenv import load_dotenv
from tavily import TavilyClient

load_dotenv()
client = TavilyClient(api_key=os.getenv("TAVILY_API_KEY"))

# 검색 결과 가져오기
result = client.search("hello world test")

# json.dumps를 사용하여 들여쓰기(indent) 적용
print(json.dumps(result, indent=4, ensure_ascii=False))
```

```
(venv) PS C:\Users\asakh\Documents\GitHub\AI_agent_lecture_02> python ./test_tavily.py
{
    "query": "hello world test",
    "response_time": 0.74,
    "follow_up_questions": null,
    "answer": null,
    "images": [],
    "results": [
        {
            "url": "https://szabgab.com/testing-hello-world",
            "title": "Testing Hello World",
            "content": "# Testing Hello World. Create a file called hello\\_world.pl with the following content:. Create a file called hello\\_world.py with the following content:. Create the file hello\\_world.rb that looks like this:. Save the following code in hello\\_world.java. that should create another file called hello\\_world.class Run it as. In order to run that we would probably create an HTML file called hello\\_world.html containing the following line. ## Testing Hello World. We would like to be able to execute our hello\\_world \"application\" regardless of the language it was written in. If you are on a Linux or Unix machine what you need to do it to put the appropriate sh-bang line in the hello\\_world program. #!/usr/bin/perl print \"Hello World\\n\"; Let's create a file called hello\\_world.t and put in the following content:. use strict; use warnings; use Test::Simple tests => 1; $ENV{PATH} = '.'; my $output = qx{@ARGV}; ok($output eq \"Hello World\\n\"); perl hello_world.t hello_world.py. perl hello_world.t hello_world.rb. perl hello_world.t hello_world.php. perl hello_world.t java.sh.",
            "score": 0.99963164,
            "raw_content": null
        },
        "request_id": "3218eb32-4b4e-45b7-8167-b8dba3046ffa"
    ]
}
```

JSON 형태로 API 응답 결과 출력!

2. 설정 파일 업데이트 - Tavily 검색 설정 추가

```
# 기본 검색 결과 개수
TAVILY_DEFAULT_MAX_RESULTS: int = 5

# 기본 검색 깊이 (기본 모드: 빠르고 간단한 검색)
TAVILY_DEFAULT_SEARCH_DEPTH: str = "basic"

# 고급 검색 깊이 (더 상세하고 포괄적인 검색)
TAVILY_ADVANCED_SEARCH_DEPTH: str = "advanced"

# 검색 쿼리 최적화: 검색 시 제거할 한국어 표현 리스트
# 사용자 질문에서 불필요한 표현을 제거하여 검색 품질 향상
QUERY_REMOVE_PHRASES: list[str] = [
    "알려줘",
    "설명해줘",
    "찾아줘",
    "에 대해",
    "에 대해서",
    "에 관해서",
    "에 관해",
    "좀",
    "제발",
    "부탁해",
    "부탁드려",
]
```

```
# 검색 쿼리 최적화: 연도로 변환할 시간 표현 리스트
# 시간 관련 표현을 현재 연도로 변환하여 검색 정확도 향상
TIME_INDICATOR_PHRASES: list[str] = [
    "최신",
    "요즘",
    "현재",
    "올해",
    "이번 년도",
    "이번년도",
    "최근",
    "지금",
]
```

3. Tools 구현 - web_search.py

모듈 개요 (MODULE OVERVIEW)

목적: Tavily API를 사용하여 에이전트에 실시간 웹 검색 능력 부여

주요 기능: 검색 실행, 결과 구조화(Parsing), 쿼리 최적화, LLM용 포맷팅

★ 주요 특징 (KEY FEATURES)

에러 처리: 입력 값 검증, 예외 처리(Try-Except), 로깅(Logging) 적용

성능: 검색 소요 시간 측정, 클라이언트 캐싱(Lazy Loading)

확장성: 검색 깊이(Depth) 조절 및 도메인 필터링 옵션 지원 구조



A. 헬퍼 함수 (Helpers)

`_get_tavily_client()`
→ 클라이언트 싱글톤/캐싱
`_validate_api_key()`



C. 검색 함수 (Core Logic)

`tavily_search(query, ...)`
→ 기본 검색, 파싱, 시간 측정
`tavily_search_with_context()`
→ 컨텍스트 + Advanced 검색



B. SearchResult 클래스

Fields: `query`, `results`, `sources`
Property: `has_answer`
Method: `get_top_results()`



D. 유틸리티 함수 (Utils)

`optimize_search_query()`
→ 한국어/공백 제거, 연도 추가
`format_for_llm()`
→ 마크다운 변환 (요약/출처)

3. Tools 구현 - SearchResult 클래스

Tavily 검색 결과를 구조화하는 데이터 클래스

클래스 개요 및 필드

Python의 `@dataclass`를 사용하여 검색 결과를 불변(immutable) 객체로 관리합니다.

`query`: 사용자가 입력한 검색어

`answer`: Tavily AI가 생성한 요약 답변

`results`: 개별 검색 결과 리스트 (`List[Dict]`)

`sources`: 출처 URL 목록

`search_time`: API 응답 소요 시간

RESULTS 필드 구조 (DICT)

`title`: 웹 페이지 제목

`url`: 원본 링크 주소

`content`: 본문 핵심 내용 요약

`score`: 관련도 점수 (0.0 ~ 1.0)

CLASS 구조

`@dataclass SearchResult`

FIELDS (ATTRIBUTES)

- `query : str`
- `answer : Optional[str]`
- `results : List[Dict[str, Any]]`
- `sources : List[str]`
- `search_time : float`
- `raw_response : Optional[Dict]`

PROPERTIES

- + `result_count : int` read-only
- + `has_answer : bool` read-only

METHODS

- + `get_top_results (n: int = 3) -> List[Dict]`
Top N개의 가장 관련성 높은 결과를 반환
- + `get_sources_as_string (separator: str = "\n") -> str`
출처 목록을 포맷팅된 문자열로 변환

3. Tools 구현 - tool_definitions.py

OpenAI Function Calling 도구 정의 스키마

파일 개요 (FILE OVERVIEW)

OpenAI Function Calling 스페어를 준수하는 JSON 스키마를 정의합니다.
LLM이 도구를 선택하고 인자를 생성하는 데 필요한 메타데이터
(이름, 설명, 파라미터)를 제공합니다.

DESCRIPTION 내용

✓ 사용 해야 하는 경우

- ✓ 실시간 정보 (주가, 날씨 등)
- ✓ 최근 뉴스 및 이벤트
- ✓ 사실 관계 확인/검증
- ✓ 명시적인 검색 요청
- ✓ 2024년 이후 최신 정보
- ✓ 최신 현황, 트렌드

✗ 사용하지 않아야 하는 경우

- ✗ 일반적인 개념/정의 설명
- ✗ 프로그래밍 문법/코드 작성
- ✗ 수학 계산 및 논리 추론
- ✗ 개인적 조언이나 의견
- ✗ 창작물 작성 (시, 소설)
- ✗ 단순 번역 작업
- ✗ 확정된 역사적 사실

src/tools/tool_definitions.py

```
SEARCH_WEB_TOOL = {
    "type": "function", # 도구 타입
    "function": {
        "name": "search_web", # 함수 이름
        "description": "...", # 상세 설명
        "parameters": { # 파라미터 스키마
            "type": "object",
            "properties": { ... },
            "required": { ... }
        }
    }
}
```

PARAMETERS

query (Required): 검색 키워드/질문
 search_depth (Optional): 깊이 설정
 "basic" 또는 "advanced"

HELPER FUNCTIONS

```
get_tool_by_name(name) -> Dict
get_all_tool_names() -> List[str]
```

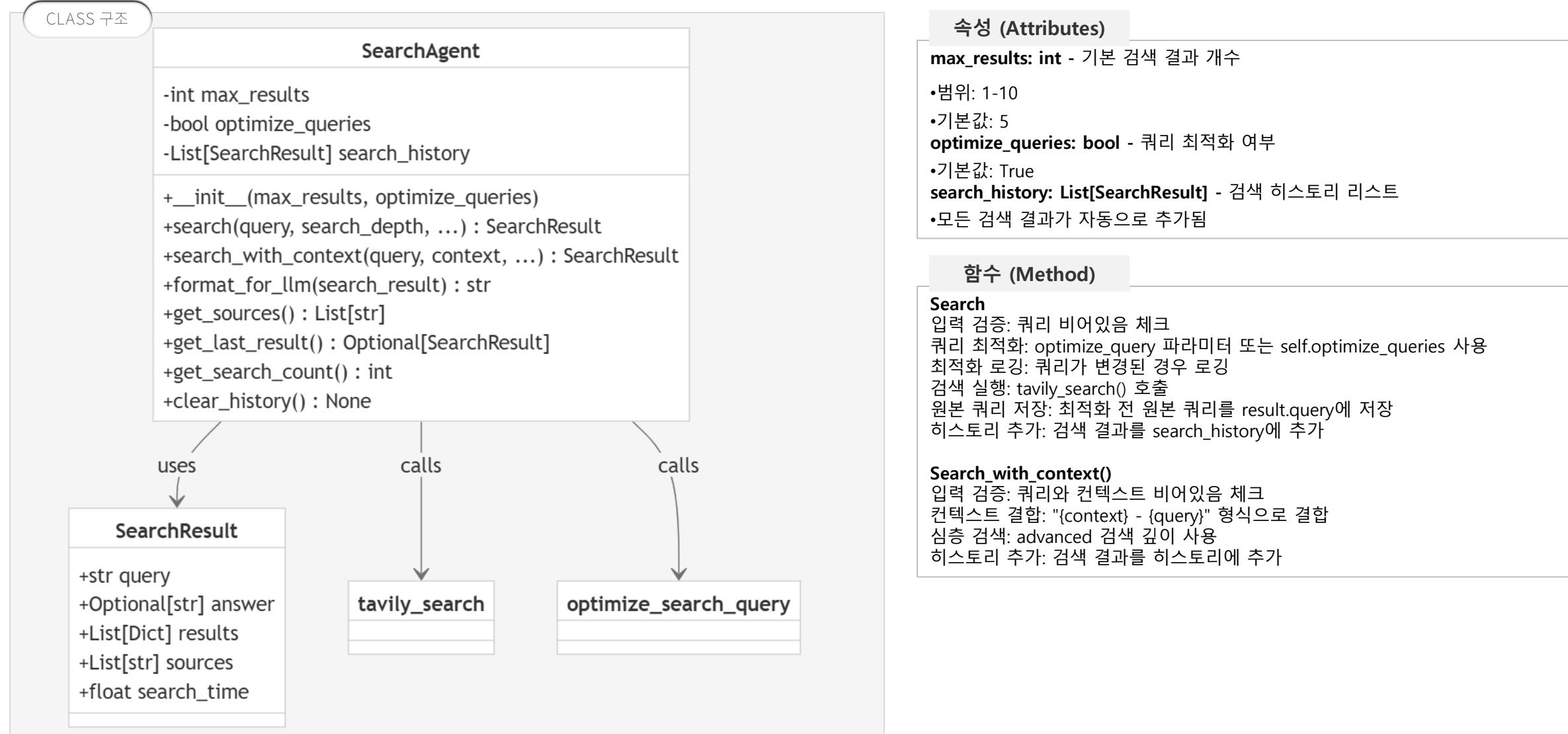
get_tool_by_name(name: str) -> Optional[Dict]

도구 이름으로 도구 정의를 찾습니다.
 파라미터: name (str) - 도구 이름
 반환값: 도구 정의 딕셔너리 또는 None

get_all_tool_names() -> List[str]

사용 가능한 모든 도구 이름 리스트를 반환합니다.
 반환값: 도구 이름 리스트

4. 검색 에이전트 구현



실습 Part1 구현 프롬프트 - 1

config/settings.py 파일에 Tavily 검색 관련 설정을 추가해주세요.

[추가할 설정]

1. Tavily 검색 기본 설정

- TAVILY_DEFAULT_MAX_RESULTS = 5
- TAVILY_DEFAULT_SEARCH_DEPTH = "basic"
- TAVILY_ADVANCED_SEARCH_DEPTH = "advanced"

2. 검색 쿼리 최적화용 상수

- QUERY_REMOVE_PHRASES: 검색 시 제거할 한국어 표현 리스트
예: "알려줘", "설명해줘", "찾아줘", "에 대해" 등
- TIME_INDICATOR_PHRASES: 연도로 변환할 시간 표현 리스트
예: "최신", "요즘", "현재", "올해" 등

[요구사항]

- 기존 설정(OpenAI 관련)은 유지
- 주석으로 섹션 구분 (# Tavily 검색 설정)
- 각 설정에 설명 주석 추가

실습 Part1 구현 프롬프트 - 2

src/tools/ 폴더를 생성하고 빈 __init__.py 파일을 만들어주세요.

[생성할 구조]

```
src/
└── tools/
    └── __init__.py
```

[__init__.py 내용]

- 모듈 설명 docstring만 포함
- "도구(Tools) 모듈 패키지" 설명
- 실제 import/export는 나중에 추가할 예정이라는 TODO 주석

실습 Part1 구현 프롬프트 - 3

src/tools/web_search.py 파일을 생성하고, 먼저 기본 구조와 SearchResult 클래스를 작성해주세요.

[파일 구조]

1. 모듈 docstring
2. Import 문
 - os, time, logging
 - typing (Optional, List, Dict, Any)
 - dataclasses (dataclass, field)
 - dotenv (load_dotenv)
3. 로깅 설정
4. 환경변수 로드
5. TavilyClient lazy import 함수 (_get_tavily_client)

[SearchResult 클래스]

@dataclass로 구현하며 다음 필드 포함:

- query: str - 검색 쿼리
- answer: Optional[str] = None - AI 요약 답변
- results: List[Dict[str, Any]] - 검색 결과 리스트 (default_factory=list)
- sources: List[str] - 출처 URL 리스트 (default_factory=list)
- search_time: float = 0.0 - 검색 소요 시간
- raw_response: Optional[Dict[str, Any]] = None - 원본 응답

[SearchResult 프로퍼티/메서드]

- result_count (property): 결과 개수 반환
- has_answer (property): answer 존재 여부
- get_top_results(n: int = 3): 상위 n개 결과 반환
- get_sources_as_string(separator: str = "\n"): 출처 문자열로 반환

[_get_tavily_client 함수]

- global 변수로 TavilyClient를 캐싱
- ImportError 시 친절한 에러 메시지

실습 Part1 구현 프롬프트 - 4

src/tools/web_search.py에 tavyly_search 함수를 추가해주세요.

[함수 시그니처]

```
def tavyly_search(
    query: str,
    api_key: Optional[str] = None,
    search_depth: str = "basic",
    include_answer: bool = True,
    include_raw_content: bool = False,
    max_results: int = 5,
    include_domains: Optional[List[str]] = None,
    exclude_domains: Optional[List[str]] = None,
) -> SearchResult:
```

[기능]

1. 입력 검증
 - query가 비어있으면 ValueError
 - query.strip() 처리
2. API 키 설정
 - 파라미터 → 환경변수 순서로 확인
 - 없으면 ValueError (친절한 메시지)
3. max_results 범위 검증 (1-10)
4. Tavyly 클라이언트 생성 및 검색 실행
 - 시간 측정 (time.time())
 - search_params 딕셔너리로 파라미터 구성
 - include_domains, exclude_domains는 있을 때만 추가
5. 결과 파싱
 - results에서 url 추출하여 sources 리스트 생성
 - SearchResult 객체 생성 및 반환
6. 로깅
 - 검색 시작: 쿼리, depth, max_results
 - 검색 완료: 결과 수, 소요 시간

[예외 처리]

- try-except로 감싸서 로깅 후 re-raise

[docstring]

Google 스타일로 Args, Returns, Raises, Example 포함

실습 Part1 구현 프롬프트 - 5

src/tools/web_search.py에 다음 함수들을 추가해주세요.

[함수 1: tavity_search_with_context]

```
def tavity_search_with_context(
    query: str,
    context: Optional[str] = None,
    api_key: Optional[str] = None,
    max_results: int = 5,
) -> SearchResult:
    - context가 있으면 쿼리 앞에 추가: f"{context} - {query}"
    - search_depth는 "advanced" 고정 (컨텍스트 검색은 심층)
    - 내부적으로 tavity_search 호출
```

[함수 2: format_search_result_for_llm]

```
def format_search_result_for_llm(search_result: SearchResult) -> str:
    - 검색 결과를 LLM 컨텍스트용 마크다운 문자열로 변환
    - 구조:
        ## 웹 검색 결과: '{query}'
        검색 시간: {time}초
        ### 📚 요약
        {answer} # has_answer일 때만
        ### 🖥️ 상세 검색 결과
        **[1] {title}**
        - 출처: {url}
        - 관련도: {score}
        - 내용: {content} # 300자 초과시 자르기
        ### 📚 참고 출처
        1. {source1}
        2. {source2}
```

[함수 3: optimize_search_query]

```
def optimize_search_query(user_input: str) -> str:
    - 사용자 입력을 검색에 최적화된 쿼리로 변환
    - 처리 내용:
        1. strip() 처리
        2. 불필요한 한국어 표현 제거 (config/settings.py의 QUERY_REMOVE_PHRASES 사용하거나 직접 정의)
            - "알려줘", "설명해줘", "찾아줘", "검색해줘", "조사해줘"
            - "뭐야", "무엇인가요", "에 대해", "관해서" 등
        3. 시간 표현 처리 (TIME_INDICATOR_PHRASES)
            - "최신", "요즘", "현재", "올해" 등이 있으면
            - 해당 표현 제거 + 현재 연도 추가
        4. 연속 공백 제거: " ".join(query.split())
```

[각 함수에 docstring과 Example 포함]

실습 Part1 구현 프롬프트 - 6

터미널에서 web_search.py 모듈을 테스트해주세요.

[테스트 방법]

파이썬 인터랙티브 셸에서:

1. 모듈 import 테스트

```
from src.tools.web_search import tavily_search, SearchResult, format_search_result_for_llm, optimize_search_query
```

2. 쿼리 최적화 테스트

```
optimize_search_query("최신 AI 트렌드에 대해 알려줘")  
# 예상: "AI 트렌드 2024" 또는 "AI 트렌드 2025"
```

3. 검색 테스트

```
result = tavily_search("AI agent trends 2024", max_results=3)  
print(f"결과 수: {result.result_count}")  
print(f"요약: {result.answer[:100]} if result.answer else 'None'...")
```

4. 포맷팅 테스트

```
formatted = format_search_result_for_llm(result)  
print(formatted[:500])
```

[확인 사항]

- ImportError 없이 모듈 로드되는지
- 검색 결과가 정상적으로 반환되는지
- 포맷팅된 문자열이 마크다운 형식인지

실습 Part1 구현 프롬프트 - 7

src/tools/tool_definitions.py 파일을 생성해주세요.

[파일 목적]

OpenAI Function Calling에 사용할 도구 정의 JSON 스키마

[내용]

- 모듈 docstring
 - 파일 목적 설명
 - description이 LLM 판단에 중요하다는 점 언급

2. SEARCH_WEB_TOOL 딕셔너리

```
{
  "type": "function",
  "function": {
    "name": "search_web",
    "description": """웹에서 최신 정보를 검색합니다."""
  }
}
```

이 도구를 사용해야 하는 경우:

- 실시간 정보 (주가, 날씨, 환율, 스포츠 결과 등)
- 최근 뉴스나 이벤트
- 특정 사실의 확인/검증
- "검색해줘", "찾아줘", "조사해줘" 등 명시적 요청
- 2024년 이후 정보
- 기업/인물/제품의 최신 현황
- 트렌드, 동향, 전망

이 도구를 사용하지 않아야 하는 경우:

- 일반 개념/정의 설명
- 프로그래밍 문법, 코드 작성
- 수학 계산, 논리적 추론
- 개인적 조언, 의견
- 창작물 (시, 소설, 이메일 등)
- 번역
- 확정된 역사적 사실""",

```
"parameters": {
  "type": "object",
  "properties": {
    "query": {
      "type": "string",
      "description": "검색할 키워드나 질문. 구체적으로 작성. 예: 'Tesla stock 2024', 'AI agent trends'"
    }
},
```

```

  "search_depth": {
    "type": "string",
    "enum": ["basic", "advanced"],
    "description": "basic: 빠른 일반 검색, advanced: 심층 검색. 기본값 basic"
  },
  "required": ["query"]
}
}
```

3. AVAILABLE_TOOLS 리스트

- SEARCH_WEB_TOOL만 포함

4. TOOLS_BY_NAME 딕셔너리

- 이름으로 도구 찾기용
- {"search_web": SEARCH_WEB_TOOL}

5. 헬퍼 함수들

- get_tool_by_name(name: str) -> dict
- get_all_tool_names() -> list

[주석]

- 향후 추가될 도구들 (FETCH_WEBPAGE_TOOL 등) 주석으로 예시

실습 Part1 구현 프롬프트 - 8

src/tools/_init__.py 파일을 완성해주세요.

[내용]

1. 모듈 docstring 업데이트

- 패키지 설명

- 포함된 모듈들 설명

2. web_search에서 import

- tavity_search

- tavity_search_with_context

- format_search_result_for_llm

- optimize_search_query

- SearchResult

3. tool_definitions에서 import

- AVAILABLE_TOOLS

- SEARCH_WEB_TOOL

- TOOLS_BY_NAME

- get_tool_by_name

- get_all_tool_names

4. __all__ 리스트 정의

- 외부에서 사용할 수 있는 모든 항목 나열

[형식]

```
from .web_search import (
```

```
    ...
```

```
)
```

```
from .tool_definitions import (
```

```
    ...
```

```
)
```

```
__all__ = [...]
```

실습 Part1 구현 프롬프트 - 9

src/search_agent.py 파일을 생성해주세요.

[파일 목적]

SearchAgent는 웹 검색 도구들을 오케스트레이션하는 상위 레벨 클래스입니다.

src/tools/web_search.py의 함수들을 조합하여 사용합니다.

[기본 구조]

1. 모듈 docstring
2. Import 문
 - logging
 - typing (Optional, List)
- src.tools.web_search에서 필요한 함수들 import:
 - SearchResult
 - tavity_search
 - tavity_search_with_context
 - format_search_result_for_llm
 - optimize_search_query
3. 로깅 설정
4. SearchAgent 클래스

[속성]

- max_results: int - 기본 검색 결과 수
- optimize_queries: bool - 쿼리 최적화 여부
- search_history: List[SearchResult] - 검색 히스토리

[__init__ 메서드]

```
def __init__(  
    self,  
    max_results: int = 5,  
    optimize_queries: bool = True  
):  
    - max_results 범위 검증 (1-10)  
    - 속성 초기화  
    - 로깅
```

[클래스 docstring]

- 클래스 설명
- Attributes 섹션
- Example 사용법

실습 Part1 구현 프롬프트 - 10

src/search_agent.py의 SearchAgent 클래스에 메서드들을 추가해주세요.

[메서드 1: search]

```
def search(
    self,
    query: str,
    search_depth: str = "basic",
    include_answer: bool = True,
    max_results: Optional[int] = None,
    optimize_query: Optional[bool] = None
) -> SearchResult:
```

- 입력 검증 (빈 쿼리 체크)

- optimize_query 처리:

- None이면 self.optimize_queries 사용
- True면 optimize_search_query() 호출

- 원본과 다르면 로깅

- tavity_search() 호출

- 원본 쿼리를 result.query에 저장 (최적화된 쿼리 대신)

- search_history에 추가

- 결과 반환

[메서드 2: search_with_context]

```
def search_with_context(
    self,
    query: str,
    context: str,
    max_results: Optional[int] = None
) -> SearchResult:
```

- tavity_search_with_context() 호출

- search_history에 추가

- 결과 반환

[메서드 3: format_for_llm]

```
def format_for_llm(self, search_result: SearchResult) -> str:
    - format_search_result_for_llm() 호출하여 반환
```

[메서드 4: get_sources]

```
def get_sources(self) -> List[str]:
    - search_history가 비어있으면 빈 리스트
    - 마지막 검색 결과의 sources 반환
```

[메서드 5: get_last_result]

```
def get_last_result(self) -> Optional[SearchResult]:
    - 마지막 검색 결과 반환 (없으면 None)
```

[메서드 6: get_search_count]

```
def get_search_count(self) -> int:
    - search_history 길이 반환
```

[메서드 7: clear_history]

```
def clear_history(self) -> None:
    - search_history 초기화
    - 로깅
```

[모든 메서드에 docstring 포함]

실습 Part1 구현 프롬프트 - 11

Part 1에서 작성한 코드 전체를 테스트해주세요.

[테스트 시나리오]

1. SearchAgent import 및 생성
from src.search_agent import SearchAgent
agent = SearchAgent()
2. 기본 검색 테스트
result = agent.search("2024년 AI 에이전트 트렌드")
print(f"쿼리: {result.query}")
print(f"결과 수: {result.result_count}")
print(f"검색 시간: {result.search_time:.2f}초")
print(f"요약: {result.answer[:200]} if result.answer else 'None'...")
3. 쿼리 최적화 확인
result2 = agent.search("최신 머신러닝에 대해 알려줘")
로그에서 쿼리 최적화 확인
4. LLM 포맷 테스트
formatted = agent.format_for_llm(result)
print(formatted[:500])
5. 출처 확인
print(f"출처: {agent.get_sources()}")
6. 히스토리 확인
print(f"총 검색 횟수: {agent.get_search_count()}")
7. 히스토리 초기화
agent.clear_history()
print(f"초기화 후 검색 횟수: {agent.get_search_count()}")

[확인 사항]

- 모든 import가 정상 동작하는지
- 검색 결과가 올바르게 반환되는지
- 쿼리 최적화가 동작하는지 (로그 확인)
- format_for_llm이 마크다운 형식인지
- 히스토리 관리가 정상인지

[문제 발생 시]

- ImportError: __init__.py의 export 확인
- API 키 오류: .env 파일 확인
- 모듈 경로 오류: 프로젝트 루트에서 실행 확인

Part1 완료 체크 리스트

8 Steps / 24 Items Completed

Step 1: 환경 설정

- ✓ .env 키 등록
- ✓ requirements.txt
- ✓ pip install
- ✓ test_tavily.py

Step 3: 폴더 구조

- ✓ src/tools/ 폴더 생성
- ✓ __init__.py 생성

Step 5: tool_definitions.py

- ✓ SEARCH_WEB_TOOL
- ✓ AVAILABLE_TOOLS
- ✓ Helper Functions

Step 7: search_agent.py

- ✓ SearchAgent Class
- ✓ Methods 구현

Step 2: 설정 파일

- ✓ config/settings.py Tavily 설정

Step 4: web_search.py

- ✓ SearchResult Class
- ✓ format_for_llm
- ✓ tavily_search
- ✓ optimize_query
- ✓ context_search
- ✓ Module Test

Step 6: __init__.py 완성

- ✓ 모듈 export 추가
- ✓ __all__ 정의

Step 8: 통합 테스트

- ✓ Agent 생성
- ✓ 포맷팅 검증
- ✓ 검색 동작 확인
- ✓ 히스토리 확인

04

SECTION

 HANDS-ON  SEARCH SYSTEM

실습 Part 2 대화 관리자와 통합

ConversationManager와 SearchAgent를 연결하여 완전한 정보 수집 에이전트 구축

KEY IMPLEMENTATION GOALS

- ✓ 시스템 통합 (Integration)
- ✓ 도구 호출 흐름 (Flow)
- ✓ 자동 판단 로직 (Logic)
- ✓ 맥락 기반 응답 (Response)

KEYWORDS:

 Integration  Orchestration  Automation

실습 Part 2 목표

◎ 달성 과제



통합 (Integration)

ConversationManager와 SearchAgent를 유기적으로 연결하여 하나의 시스템으로 동작하게 합니다.



판단 (Decision)

사용자의 질문을 분석하여 외부 검색이 필요한지 여부를 자동으로 판단하는 로직을 구현합니다.



흐름 (Flow)

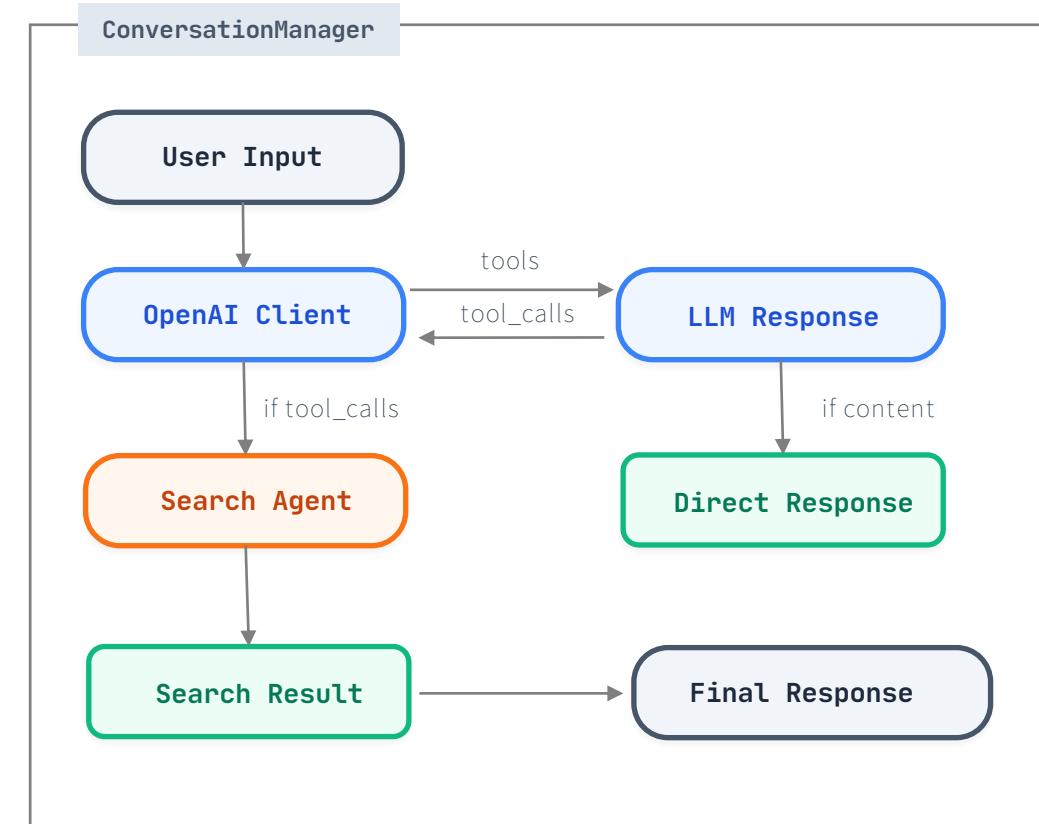
OpenAI Function Calling을 활용하여 도구 호출 및 결과 반환의 전체 사이클을 제어합니다.



응답 (Response)

검색된 정보를 바탕으로 출처가 포함된 자연스러운 답변을 생성합니다.

▣ 통합 아키텍처



2주차 전체 작업 파일

프로젝트 파일 구조 및 작업 현황 요약

작업 단계 구분

Part 1 도구 및 에이전트 구축

SearchAgent 클래스와 Tavily API 연동을 위한 기본 모듈을 생성합니다.

Part 2 통합 및 오케스트레이션

ConversationManager와 도구를 연결하고 시스템 메시지를 고도화합니다.

파일 작업 통계

10

총 파일

7

신규 생성

3

수정

2

설정 파일

Project File List

FILE NAME	ROLE	PHASE
src/tools/web_search.py NEW	Tavily API 호출 및 결과 처리	Part 1
src/tools/tool_definitions.py NEW	OpenAI Function Calling 스키마	Part 1
src/tools/__init__.py NEW	도구 패키지 초기화 및 export	Part 1
src/search_agent.py NEW	검색 에이전트 클래스	Part 1
config/settings.py MOD	Tavily API 설정 추가	Part 1
.env MOD	API Key 환경변수 추가	Part 1
requirements.txt MOD	패키지 의존성 (tavily-python)	Part 1
src/conversation_manager.py MOD	도구 실행 로직 통합	Part 2
config/prompts.py MOD	System Message 업데이트 (V2)	Part 2
main.py MOD	출처 표시 및 상태 명령어 추가	Part 2

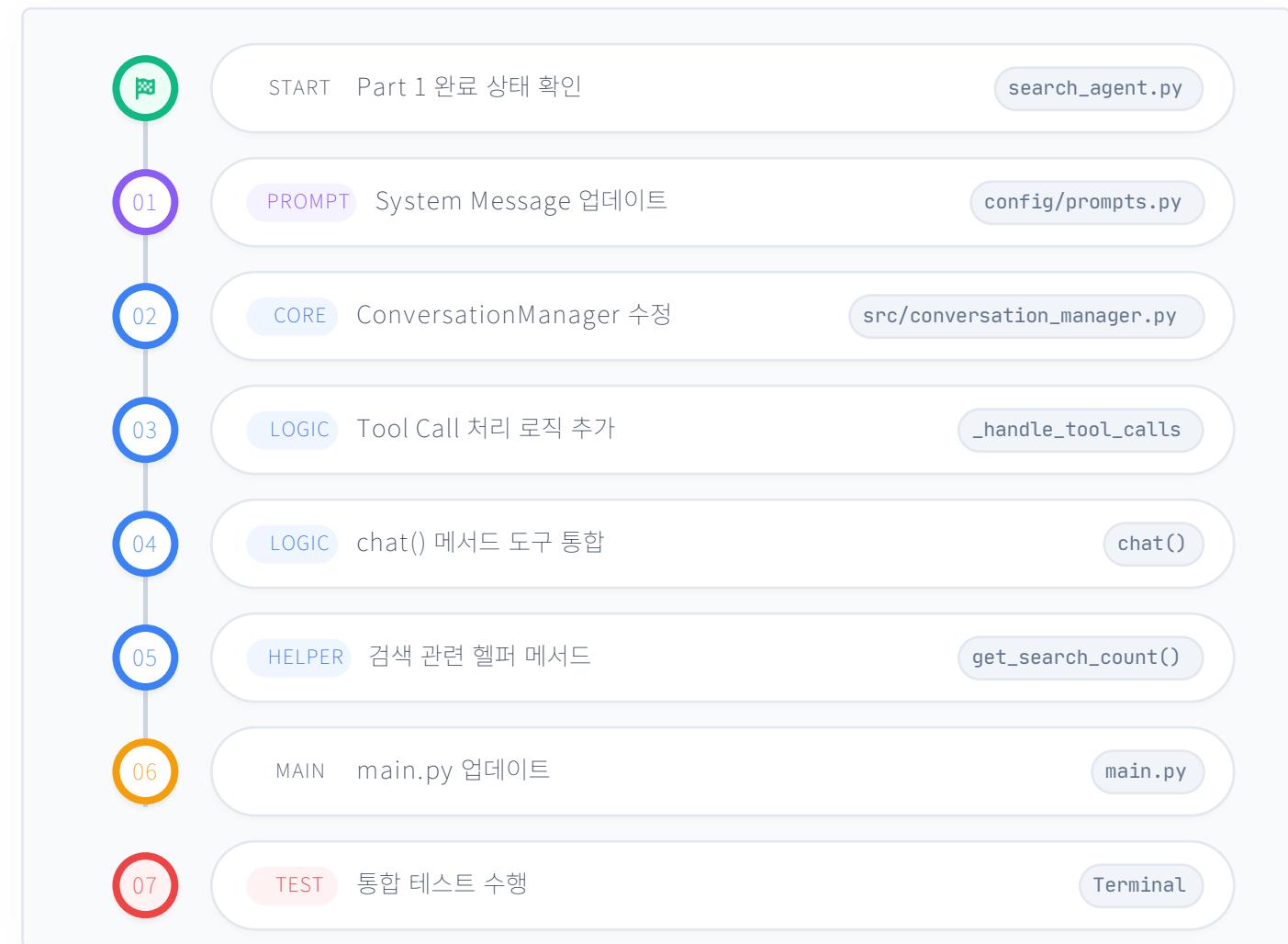
Part 2 작업 순서

✓ 전제 조건 (Prerequisites)

- ✓ Part 1 구현 완료
SearchAgent 클래스 및 웹 검색 도구 동작 확인
- ✓ 도구 정의 확인
`AVAILABLE_TOOLS`, `SEARCH_WEB_TOOL`

☰ 작업 흐름 요약

- System Message: 도구 사용 지시 추가
- Logic Update: 도구 호출 판단 및 실행
- Integration: 검색 결과를 포함한 답변 생성



RESEARCH_ASSISTANT_SYSTEM_MESSAGE_V2 주요 내용

웹 검색 도구 통합을 위한 시스템 프롬프트 고도화

🤖 핵심 역할 (CORE ROLE)

"웹 검색 기능을 갖춘 전문 리서치 어시스턴트로서, 정확하고 신뢰할 수 있는 최신 정보를 제공합니다."

☰ 응답 형식 (RESPONSE FORMAT)



⚖️ 도구 사용 원칙 (TOOL USAGE PRINCIPLES)

✓ 사용 권장 (Do)

- ✓ 실시간 정보 (주가, 날씨, 환율 등)
- ✓ 최근 뉴스 및 이벤트 ("최근", "올해")
- ✓ 특정 사실의 확인 및 검증
- ✓ 명시적 검색 요청 ("찾아줘", "검색해줘")
- ✓ 2024년 이후의 최신 정보
- ✓ 기업/인물/제품의 최신 현황
- ✓ 트렌드, 동향, 전망 조사

🚫 사용 지양 (Don't)

- ✗ 일반적인 개념이나 정의 설명
- ✗ 프로그래밍 문법 설명 및 코드 작성
- ✗ 수학 계산 및 논리적 추론
- ✗ 개인적인 조언이나 의견 제시
- ✗ 창작물 작성 (시, 소설, 이메일)
- ✗ 단순 번역 작업
- ✗ 이미 확정된 역사적 사실

↔ VERSION COMPARISON

V1 (Basic) vs V2 (Search)

항목	V1 (기본 버전)	V2 (웹 검색 버전)
핵심 기능	일반 리서치 어시스턴트	웹 검색 기능 포함 어시스턴트
도구 사용	없음	search_web 도구 사용
정보 소스	기존 학습 지식 기반	실시간 웹 검색 결과 기반
명확화	모호한 질문 시 필수 강조	도구 사용 원칙에 집중
출처 명시	가능한 경우 언급	검색 시 필수 명시
응답 형식	일반 대화 형식	구조화된 형식 (요약, 상세, 출처)
최신 정보	시점 명시 권장	웹 검색으로 최신 정보 제공
검색 제안	없음	필요 시 검색 제안 가능
제약사항	할루시네이션 주의	검색하지 않은 정보 날조 금지

ConversationManager에 SearchAgent 통합

➊ 초기화 과정 (IMPORT & SETUP)

```
try:
    from src.search_agent import SearchAgent
    from src.tools.tool_definitions import AVAILABLE_TOOLS
except ImportError:
    # Part 1 기능이 완성되지 않은 경우를 위한 풀백
    SearchAgent = None
    AVAILABLE_TOOLS = []
```

➋ 초기화 시점 (__INIT__)

```
# 검색 기능 설정
self.enable_search = enable_search
self.search_agent: Optional[SearchAgent] = None
self.tools: Optional[List[dict]] = None
if enable_search:
    try:
        if SearchAgent is not None:
            self.search_agent = SearchAgent() # SearchAgent 인스턴스 생성
            self.tools = AVAILABLE_TOOLS if AVAILABLE_TOOLS else [] # 도구 정의 설정
            logger.info("검색 기능 활성화됨")
        else:
            logger.warning("SearchAgent가 사용 불가능합니다. 검색 기능을 비활성화합니다.")
            self.enable_search = False
    except Exception as e:
        logger.warning(f"검색 기능 초기화 실패: {e}")
        self.enable_search = False
```

EXECUTION FLOW

1. Configuration Check
if self.enable_search:

2. API Call with Tools
client.chat...(tools=self.tools)

3. LLM Decision
tool_choice="auto" → tool_calls

4. Execute & Return
_execute_tool() → SearchAgent

SUMMARY

- 도구 활성화 확인
- Tools 스키마 전달
- LLM 의도 파악
- 검색 수행 및 결과

ConversationManager 추가되는 주요 함수

도구 호출(Tool Calling)과 실행(Execution)의 분리

1. _CALL_API_WITH_TOOLS()

OpenAI API 호출 시 검색 기능 활성화 여부를 확인하고, 활성화된 경우 tools 스키마를 파라미터에 포함합니다.

src/conversation_manager.py

```
def _call_api_with_tools (self):
    # ... 기본 파라미터 설정 생략 ...

    # 검색 기능 활성화 시 도구 추가
    if self.enable_search and self.tools:
        call_params[ "tools" ] = self.tools
        call_params[ "tool_choice" ] = "auto"

    # API 호출 및 응답 반환
    response = self.client.chat.completions.create(
        **call_params
    )

    return response
```

2. _EXECUTE_TOOL()

LLM이 요청한 도구 이름을 확인하여 SearchAgent를 통해 실제 검색을 수행하고, 그 결과를 문자열로 포맷팅하여 반환합니다.

src/conversation_manager.py

```
def _execute_tool (self, function_name, arguments):
    try:
        if function_name == "search_web" :
            query = arguments.get( "query" , "" )
            depth = arguments.get( "search_depth" , "basic" )

            if not query:
                return json.dumps({ "error" : "Empty query" })

            # SearchAgent로 검색 실행 및 포맷팅
            res = self.search_agent.search(query, depth)
            return self.search_agent.format_for_llm(res)
        else:
            return json.dumps({ "error" : f"Unknown tool: {function_name}" })
    except Exception as e:
        return json.dumps({ "error" : f"Error: {str(e)}" })
```

ConversationManager 추가되는 주요 함수

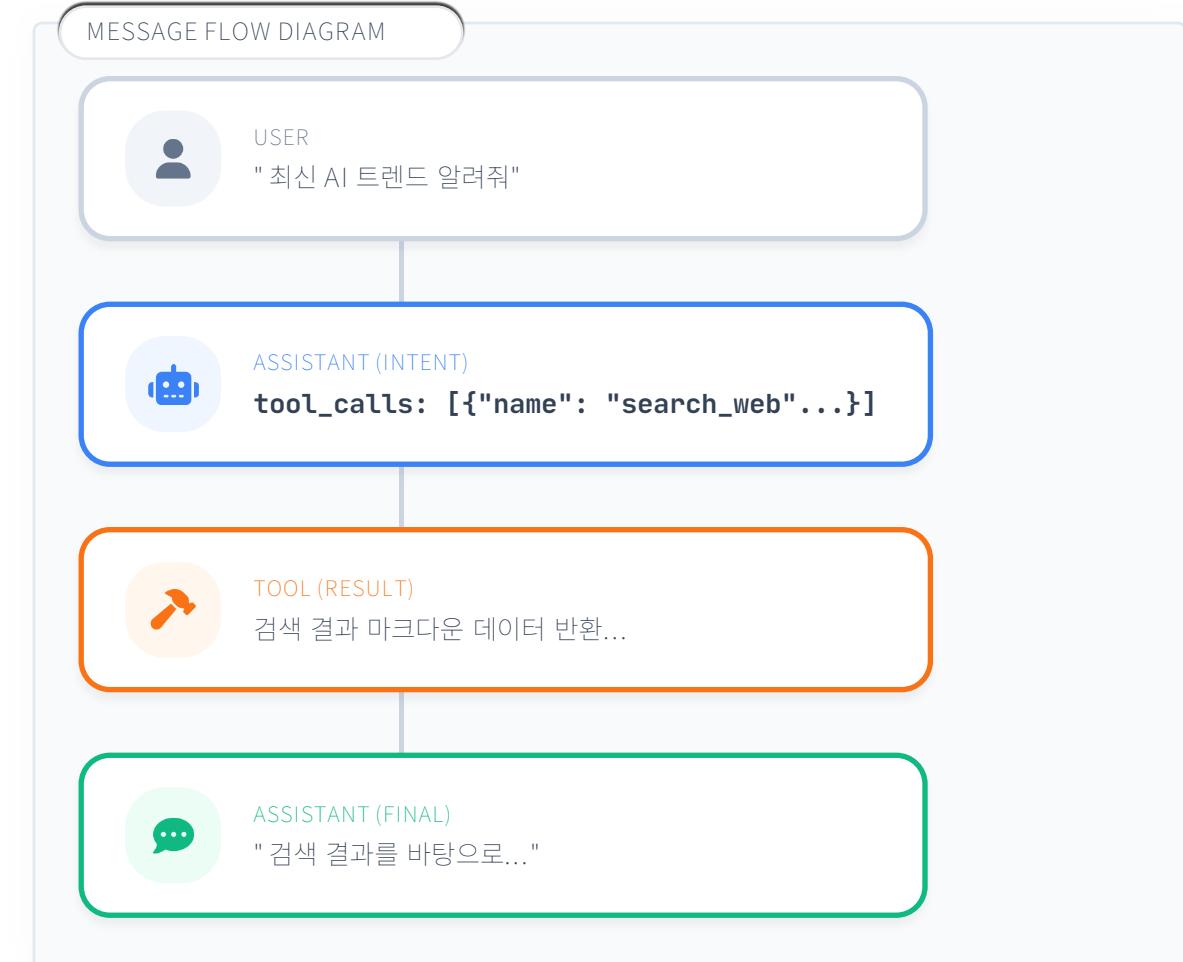
3. _handle_tool_calls()

함수 역할 (ROLE)

LLM이 요청한 도구 호출(Tool Calls)을 순차적으로 처리하고, 결과를 다시 LLM에 전달하여 최종 응답을 생성하는 오케스트레이션 메서드

동작 흐름 (EXECUTION STEPS)

- 1 Assistant의 tool_calls 요청 정보를 대화 기록(Messages)에 저장
- 2 요청된 각 도구를 실행하고, 결과를 role="tool" 메시지로 추가
- 3 도구 실행 결과(Context)를 포함하여 OpenAI API 재호출
- 4 생성된 최종 자연어 응답을 히스토리에 저장하고 반환



ConversationManager에서 Chat 함수 수정

def chat() 메서드 업데이트 및 도구 통합

1. 기능 정의

사용자의 자연어 입력을 받아, 도구 사용 여부를 포함한 적절한 AI 응답을 생성하고 반환하는 메인 메서드입니다.

1. API 호출 부분 변경

```
# 기준: 직접 API 호출 (도구 미지원)
response =self .client.chat.completions.create(
    model=DEFAULT_MODEL,
    messages=self .messages
)
```

↓

```
# 변경: 헬퍼 메서드 사용 (도구 지원)
response =self ._call_api_with_tools ()
```

```
# _call_api_with_tools() 내부에서 # enable_search 확인 및 tools
파라미터 # 자동 주입 처리
```

2. 응답 처리 로직 변경

단순 텍스트 응답 처리에서 도구 호출(tool_calls) 감지 및 분기 처리 로직으로 확장됩니다.

2. 응답 처리 및 분기

```
message = response.choices[ 0 ].message
# 기준: 무조건 텍스트 내용만 추출
content = message.content
return content
```

↓

```
# 변경: 도구 호출 여부 확인
if message .tool_calls:
    logger .info (f"도구 호출 감지: { len(message.tool_calls)}개" )
    # 도구 실행 및 최종 응답 생성 위임
    result =self ._handle_tool_calls (message)
else :
    # 일반 텍스트 응답 처리
    result = message.content
    self .messages. append ({
        "role": "assistant" ,
        "content" :result
    })
return result
```

main.py의 handle_command() 함수에 명령어 추가

사용자 편의를 위한 상태 확인 및 출처 조회 기능 구현

1. sources 명령어

가장 최근 검색 결과의 출처(URL) 목록을 표시합니다. 답변의 근거를 확인하고 싶을 때 사용합니다.

```
# NEW 출처 보기 명령어
if command == 'sources':
    sources = manager.get_last_search_sources()
    if sources:
        print("\n 마지막 검색 출처:")
        for i, source in enumerate(sources, 1):
            print(f" {i}. {source}")
        print()
    else:
        print("\n검색 기록이 없습니다.\n")
return True
```

2. status 명령어

현재 시스템의 검색 기능 활성화 여부와 사용 통계(대화/검색 횟수)를 실시간으로 확인합니다.

```
# NEW 상태 확인 명령어
if command == 'status':
    print(f"\n 현재 상태:")
    print(f" • 검색 기능: {'활성화' if manager.is_search_enabled() else '비활성화'}")
    print(f" • 대화 횟수: {manager.get_message_count()}회")
    print(f" • 검색 횟수: {manager.get_search_count()}회")
    print()
    return True
```

Tool Calling 전체 흐름 (Sequence Diagram)

사용자 질문부터 검색 실행, 최종 응답까지의 여정

Trigger

사용자가 질문을 입력하면 ConversationManager가 상태를 검증하고 API에 전달합니다.

Decision (Alt)

[검색 필요] `tool_calls` 를 반환하면, 로컬의 SearchAgent가 Tavily API를 통해 정보를 수집합니다.

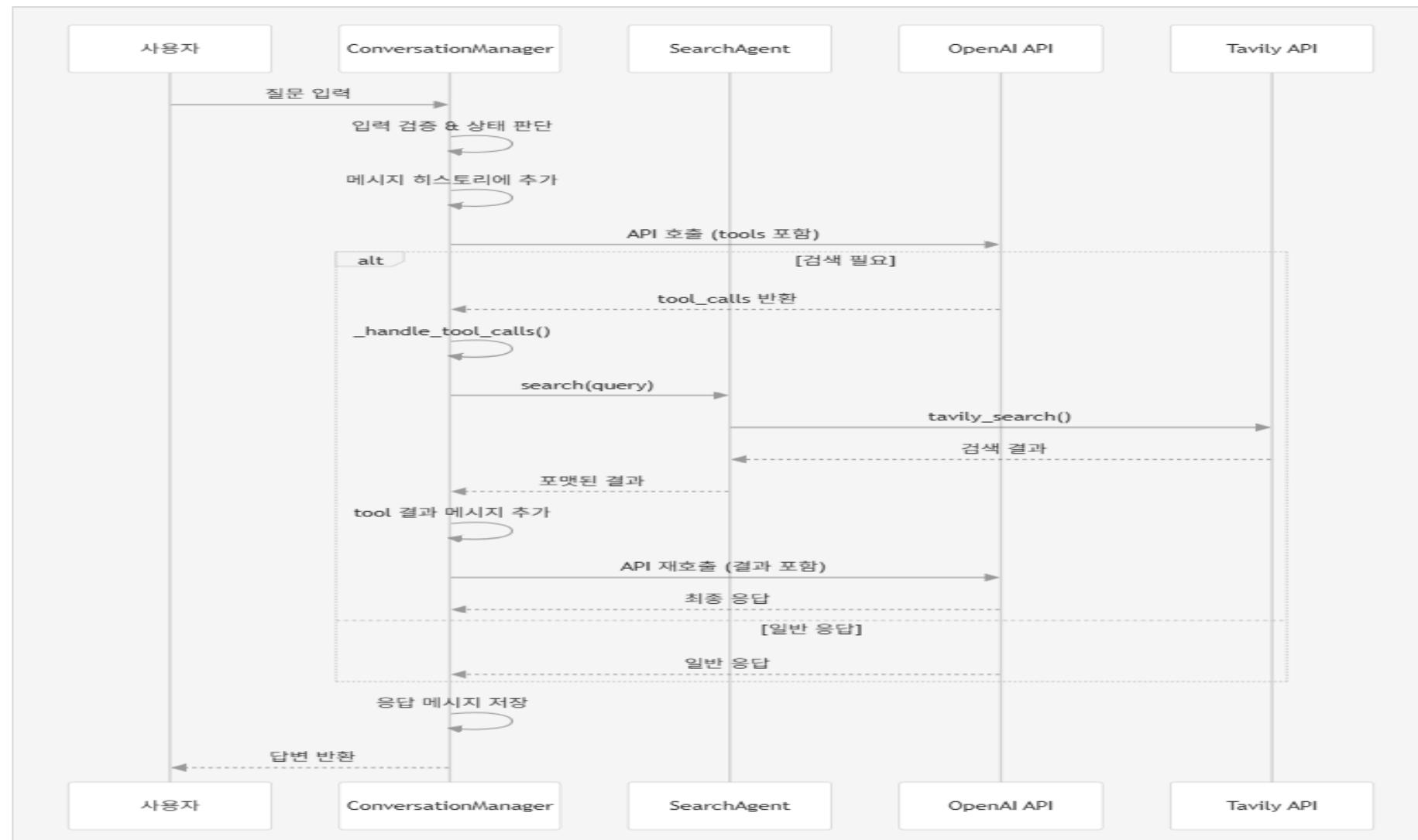
[일반] 도구 호출 없이 바로 답변합니다.

Synthesis

수집된 검색 결과(Context)를 대화 기록에 추가하여 API를 재호출하면, LLM이 최종 답변을 생성합니다.

PARTICIPANTS

- User
- Manager
- OpenAI
- SearchAgent



실습 Part2 구현 프롬프트 - 1

config/prompts.py 파일에 검색 기능을 포함한 새로운 System Message를 추가해주세요.

[추가할 변수]

RESEARCH_ASSISTANT_SYSTEM_MESSAGE_V2

[내용 구조]

1. 핵심 역할 섹션

- 웹 검색 기능을 갖춘 전문 리서치 어시스턴트
- 사용자 정보 요구 파악
- search_web 도구 사용하여 최신 정보 검색
- 검색 결과 바탕으로 정확한 정보 제공
- 항상 출처 명시

2. 도구 사용 원칙 섹션

- ### search_web 도구를 사용해야 하는 경우:
- 실시간 정보 (주가, 환율, 날씨 등)
 - 최근 뉴스, 이벤트 ("요즘", "최근", "올해", "2024년" 등)
 - 사실 확인/검증
 - 명시적 요청 ("검색해줘", "찾아줘", "조사해줘")
 - 기업/인물/제품 최신 현황
 - 트렌드, 동향, 전망

- ### search_web 도구를 사용하지 않아야 하는 경우:
- 일반 개념/정의 설명
 - 프로그래밍 문법, 코드 작성
 - 수학 계산, 논리적 추론
 - 개인적 조언, 의견
 - 창작물 (글, 시, 이메일 등)
 - 번역
 - 확정된 역사적 사실

3. 응답 형식 섹션

- ### 검색 결과 활용 시 형식:

[핵심 요약]

검색 결과 바탕 핵심 내용

[상세 내용]

- 주요 포인트 1

- 주요 포인트 2

[참고 출처]

1. 출처 URL

2. 출처 URL

일반 응답 시:
자연스럽게 대화하되, 필요시 검색 제안

4. 제약사항 섹션

- 검색 안 한 정보를 검색한 것처럼 제시 금지
- 불확실한 정보는 명시
- 검색 실패 시 사용자에게 알림
- 민감 정보의

5. 대화 스타일 섹션

- 전문적이면서 친근한 톤
- 복잡한 정보는 구조화
- 맞춤형 응답

[요구사항]

- 기존 RESEARCH_ASSISTANT_SYSTEM_MESSAGE는 유지 (V1)
- 새로운 V2는 별도 변수로 추가
- 멀티라인 문자열(triple quotes) 사용

실습 Part2 구현 프롬프트 - 2

src/conversation_manager.py 파일 상단의 import 섹션을 수정해주세요.

[추가할 import]

1. 내부 모듈 import (기존 import 아래에 추가)

```
from src.search_agent import SearchAgent  
from src.tools.tool_definitions import AVAILABLE_TOOLS
```

2. 프롬프트 import 수정

기존: RESEARCH_ASSISTANT_SYSTEM_MESSAGE

변경: RESEARCH_ASSISTANT_SYSTEM_MESSAGE_V2 (또는 둘 다 import)

[요구사항]

- 기존 import는 유지
- 내부 모듈 import는 별도 섹션으로 구분 (주석 추가)
- try-except로 ImportError 처리 (Part 1 미완료 시 대비)

실습 Part2 구현 프롬프트 - 3

src/conversation_manager.py의 `_init_` 메서드를 수정해주세요.

[변경 사항]

1. 파라미터 추가

```
def __init__(
    self,
    system_message: Optional[str] = None,
    enable_search: bool = True # NEW 추가
):
```

2. 새로운 속성 초기화 (기존 속성 초기화 후에 추가)

```
# NEW 검색 기능 설정
self.enable_search = enable_search
self.search_agent: Optional[SearchAgent] = None
self.tools: Optional[List[dict]] = None
```

if enable_search:

```
    try:
        self.search_agent = SearchAgent()
        self.tools = AVAILABLE_TOOLS
        logger.info("검색 기능 활성화됨")
    except Exception as e:
        logger.warning(f"검색 기능 초기화 실패: {e}")
        self.enable_search = False
```

3. System Message 기본값 변경

- 기존: RESEARCH_ASSISTANT_SYSTEM_MESSAGE
- 변경: RESEARCH_ASSISTANT_SYSTEM_MESSAGE_V2

4. 로깅 메시지 업데이트

```
logger.info(
    f"ConversationManager 초기화 완료 "
    f"(search={'활성화' if self.enable_search else '비활성화'})"
)
```

[요구사항]

- 기존 초기화 로직 유지
- `enable_search=False`일 때도 정상 동작해야 함
- `SearchAgent` 초기화 실패해도 대화 기능은 동작해야 함

실습 Part2 구현 프롬프트 - 4

conversation_manager 수정한 내용을 테스트해주세요.

[테스트 코드]

파이썬 인터랙티브 셀에서:

```
from src.conversation_manager import ConversationManager

# 검색 기능 활성화 테스트
manager1 = ConversationManager(enable_search=True)
print(f"검색 활성화: {manager1.enable_search}")
print(f"SearchAgent: {manager1.search_agent is not None}")
print(f"Tools: {manager1.tools is not None}")

# 검색 기능 비활성화 테스트
manager2 = ConversationManager(enable_search=False)
print(f"검색 활성화: {manager2.enable_search}")
print(f"SearchAgent: {manager2.search_agent is not None}")
```

[확인 사항]

- ImportError 없이 모듈 로드되는지
- enable_search=True일 때 SearchAgent 생성되는지
- enable_search=False일 때 SearchAgent가 None인지
- 로그 메시지가 올바르게 출력되는지

실습 Part2 구현 프롬프트 - 5

src/conversation_manager.py에 _call_api_with_tools 메서드를 추가해주세요.

[위치]
chat() 메서드 위 또는 아래 (private 메서드 섹션)

[메서드 시그니처]
def _call_api_with_tools(self) -> Any:

[기능]

1. API 호출 파라미터 딕셔너리 구성

```
call_params = {  
    "model": DEFAULT_MODEL,  
    "messages": self.messages,  
    "temperature": DEFAULT_TEMPERATURE,  
}
```

2. 검색 기능 활성화 시 도구 추가

```
if self.enable_search and self.tools:  
    call_params["tools"] = self.tools  
    call_params["tool_choice"] = "auto" # LLM이 자동 판단
```

3. API 호출 및 응답 반환

```
response = self.client.chat.completions.create(**call_params)  
return response
```

[docstring]

```
"""  
도구 정의를 포함하여 OpenAI API를 호출합니다.  
검색 기능이 활성화되어 있으면 tools 파라미터를 추가하여  
LLM이 필요시 도구를 호출할 수 있도록 합니다.  
Returns:  
    OpenAI API 응답 객체  
"""
```

[참고]

- tool_choice="auto": LLM이 도구 사용 여부를 자동 판단
- tool_choice="required": 반드시 도구 사용
- tool_choice="none": 도구 사용 안 함

실습 Part2 구현 프롬프트 - 6

src/conversation_manager.py에 _handle_tool_calls 메서드를 추가해주세요.

[메서드 시그니처]

```
def _handle_tool_calls(self, message) -> str:
```

[파라미터]

- message: OpenAI API 응답의 choices[0].message 객체
- message.content: 텍스트 응답(있을 수도 없을 수도 있음)
- message.tool_calls: 도구 호출 리스트

[기능]

1. Assistant 메시지 저장 (tool_calls 포함)

- OpenAI API 규격에 맞게 저장해야 함

```
assistant_msg = {
    "role": "assistant",
    "content": message.content, # None일 수 있음
    "tool_calls": [
        {
            "id": tc.id,
            "type": "function",
            "function": {
                "name": tc.function.name,
                "arguments": tc.function.arguments
            }
        }
    for tc in message.tool_calls
    ]
}
self.messages.append(assistant_msg)
```

2. 각 도구 호출 처리

```
for tool_call in message.tool_calls:
    function_name = tool_call.function.name
    arguments = json.loads(tool_call.function.arguments)

    logger.info(f"도구 실행: {function_name}({arguments})")

    # 도구 실행
    result = self._execute_tool(function_name, arguments)
```

결과를 메시지에 추가 (tool role)

```
self.messages.append({
    "role": "tool",
    "tool_call_id": tool_call.id,
    "content": result
})
```

3. 도구 결과 포함하여 API 재호출

```
final_response = self._call_api_with_tools()
final_content = final_response.choices[0].message.content
```

4. 최종 응답 저장 및 반환

```
self.messages.append({
    "role": "assistant",
    "content": final_content
})
return final_content
```

[docstring]

"""\nLLM이 요청한 도구 호출을 처리합니다.

1. Assistant의 도구 호출 요청을 메시지에 저장
2. 각 도구를 실행하고 결과를 메시지에 추가
3. 도구 결과를 포함하여 API를 다시 호출
4. 최종 응답을 반환

Args:

message: OpenAI API 응답의 message 객체 (tool_calls 포함)

Returns:

str: 도구 결과를 반영한 최종 AI 응답

[중요]

- tool_call_id는 반드시 원본 ID를 그대로 사용해야 함
- arguments는 JSON 문자열이므로 json.loads() 필요
- role="tool"은 OpenAI API 규격

실습 Part2 구현 프롬프트 - 7

src/conversation_manager.py에 _execute_tool 메서드를 추가해주세요.

[메서드 시그니처]

```
def _execute_tool(self, function_name: str, arguments: dict) -> str:
```

[기능]

1. search_web 도구 처리

```
if function_name == "search_web":  
    query = arguments.get("query", "")  
    search_depth = arguments.get("search_depth", "basic")
```

쿼리 검증

```
if not query:  
    return json.dumps({"error": "검색어가 비어있습니다."})
```

SearchAgent로 검색 실행

```
search_result = self.search_agent.search(  
    query=query,  
    search_depth=search_depth  
)
```

LLM용 포맷으로 변환

```
formatted = self.search_agent.format_for_llm(search_result)
```

```
logger.info(  
    f"검색 완료: {search_result.result_count}개 결과, "  
    f"{search_result.search_time:.2f}초"  
)
```

return formatted

2. 알 수 없는 도구 처리

else:

```
logger.warning(f"알 수 없는 도구: {function_name}")  
return json.dumps({"error": f"알 수 없는 도구: {function_name}"})
```

3. 예외 처리

전체를 try-except로 감싸서 도구 실행 실패 시에도
대화가 중단되지 않도록 함

except Exception as e:

```
    logger.error(f"도구 실행 실패 ({function_name}): {str(e)}")  
    return json.dumps({"error": f"도구 실행 실패: {str(e)}"})
```

[docstring]

"""

지정된 도구를 실행하고 결과를 반환합니다.

Args:

function_name: 실행할 도구 이름
arguments: 도구에 전달할 인자 딕셔너리

Returns:

str: 도구 실행 결과 (LLM이 이해할 수 있는 형식)

"""

[반환값 형식]

- 성공: format_for_llm() 결과 (마크다운 문자열)
- 실패: JSON 문자열 {"error": "에러 메시지"}

실습 Part2 구현 프롬프트 - 8

src/conversation_manager.py의 chat 메서드를 수정해주세요.

[기존 chat 메서드 구조]
def chat(self, user_input: str) -> str:
 # 입력 검증
 # 사용자 메시지 추가
 # API 호출
 # 응답 추출
 # Assistant 메시지 추가
 # 반환

[수정할 부분]

1. API 호출 부분 변경
 기존:
 response = self.client.chat.completions.create(
 model=DEFAULT_MODEL,
 messages=self.messages,
 temperature=DEFAULT_TEMPERATURE
)
 변경:
 response = self._call_api_with_tools()

2. 응답 처리 부분 변경

기존:
 assistant_message = response.choices[0].message.content
 self.messages.append({"role": "assistant", "content": assistant_message})
 return assistant_message

변경:
 message = response.choices[0].message

NEW 도구 호출 확인 및 처리
if message.tool_calls:
 logger.info(f"도구 호출 감지: {len(message.tool_calls)}개")
 result = self._handle_tool_calls(message)
else:
 # 일반 응답
 result = message.content
 self.messages.append({
 "role": "assistant",
 "content": result
 })

self.message_count += 1
self.state = "idle"
return result

[전체 수정된 chat 메서드 구조]
def chat(self, user_input: str) -> str:
 # 입력 검증
 if not user_input or not user_input.strip():
 raise ValueError("입력이 비어있습니다.")
 user_input = user_input.strip()
 self.state = "processing"
 # 사용자 메시지 추가
 self.messages.append({
 "role": "user",
 "content": user_input
 })
 try:
 # NEW 도구 포함하여 API 호출
 response = self._call_api_with_tools()
 message = response.choices[0].message

 # NEW 도구 호출 확인 및 처리
 if message.tool_calls:
 logger.info(f"도구 호출 감지: {len(message.tool_calls)}개")
 result = self._handle_tool_calls(message)
 else:
 # 일반 응답
 result = message.content
 self.messages.append({
 "role": "assistant",
 "content": result
 })
 self.message_count += 1
 self.state = "idle"
 return result
 except Exception as e:
 self.state = "error"
 logger.error(f"응답 생성 실패: {str(e)}")
 raise

[요구사항]

- 기존 예외 처리 로직 유지
- message.tool_calls가 None이거나 빈 리스트일 때 일반 응답 처리
- 로깅 유지/강화

실습 Part2 구현 프롬프트 - 9

main.py의 print_welcome() 함수를 업데이트해주세요.

[변경 내용]

```
def print_welcome():
    """환경 메시지를 출력합니다."""
    print()
    print("=" * 60)
    print("AI 리서치 어시스턴트 v2.0")
    print("웹 검색 기능이 추가되었습니다!")
    print("=" * 60)
    print()
    print("⭐ 사용 가능한 명령어:")
    print("• quit / exit / 종료 : 프로그램 종료")
    print("• save : 대화 저장")
    print("• clear : 대화 히스토리 초기화")
    print("• sources : 마지막 검색 출처 보기") # NEW
    print("• status : 현재 상태 확인") # NEW
    print()
    print("💡 검색 활용 팁:") # NEW
    print("• '~에 대해 조사해줘' → 웹 검색 실행")
    print("• '최신 ~ 알려줘' → 최신 정보 검색")
    print("• '~ 뉴스 찾아줘' → 관련 뉴스 검색")
    print()
    print("=" * 60)
    print()
```

실습 Part2 구현 프롬프트 - 10

main.py의 handle_command() 함수에 새로운 명령어를 추가해주세요.

[추가할 명령어]

1. sources 명령어
 # NEW 출처 보기 명령어
 if command == 'sources':
 sources = manager.get_last_search_sources()
 if sources:
 print("₩n 최근 검색 출처:")
 for i, source in enumerate(sources, 1):
 print(f" {i}. {source}")
 print()
 else:
 print("₩n 검색 기록이 없습니다₩n")
 return True

2. status 명령어
 # NEW 상태 확인 명령어
 if command == 'status':
 print("₩n 현재 상태:")
 print("• 검색 가능: {'활성화' if manager.is_search_enabled() else '비활성화'}")
 print("• 대화 횟수: {manager.get_message_count()}회")
 print("• 검색 횟수: {manager.get_search_count()}회")
 print()
 return True

[전체 handle_command 함수 구조]

```
def handle_command(command: str, manager: ConversationManager) -> bool:
    command = command.lower().strip()
```

```
# 종료 명령어
if command in ['quit', 'exit', '종료']:
    return handle_quit(manager)
```

```
# 저장 명령어
if command == 'save':
    # 기존 코드...
    return True
```

```
# 초기화 명령어
if command == 'clear':
    # 기존 코드...
    return True
```

```
# NEW 출처 보기 명령어
if command == 'sources':
    # 위 코드...
    return True

# NEW 상태 확인 명령어
if command == 'status':
    # 위 코드...
    return True

return False
```

실습 Part2 구현 프롬프트 - 11

main.py의 main() 함수를 수정해주세요.

[변경 사항]

1. ConversationManager 초기화 부분 수정

```
try:  
    # NEW enable_search=True 명시  
    manager = ConversationManager(enable_search=True)  
  
    # NEW 검색 기능 상태 출력  
    if manager.is_search_enabled():  
        print("✓ 검색 기능이 활성화되었습니다.\n")  
    else:  
        print("⚠️ 검색 기능이 비활성화되었습니다. (API 키 확인 필요)\n")  
  
except Exception as e:  
    logger.error(f"초기화 실패: {e}")  
    print(f"✗ 초기화 실패: {e}")  
    print("환경 설정을 확인해주세요. (.env 파일)")  
    return
```

2. 대화 루프에서 처리 중 메시지 추가 (선택사항)

```
# AI 응답 생성  
print("\n✉️ 처리 중...") # NEW 검색 시 시간이 걸릴 수 있으므로  
response = manager.chat(user_input)  
print(f"\nAI: {response}\n")
```

3. 종료 메시지 수정

```
print()  
print("=" * 60)  
print("👋 대화를 종료합니다. 안녕히 가세요!")  
print(f"총 대화: {manager.get_message_count()}회")  
print(f"총 검색: {manager.get_search_count()}회") # NEW  
print("=" * 60)  
print()
```

실습 Part2 구현 프롬프트 - 12

main.py의 main() 함수를 수정해주세요.

[변경 사항]

1. ConversationManager 초기화 부분 수정

```
try:  
    # NEW enable_search=True 명시  
    manager = ConversationManager(enable_search=True)  
  
    # NEW 검색 기능 상태 출력  
    if manager.is_search_enabled():  
        print("✓ 검색 기능이 활성화되었습니다.\n")  
    else:  
        print("⚠️ 검색 기능이 비활성화되었습니다. (API 키 확인 필요)\n")  
except Exception as e:  
    logger.error(f"초기화 실패: {e}")  
    print(f"✗ 초기화 실패: {e}")  
    print("환경 설정을 확인해주세요. (.env 파일)")  
return
```

2. 대화 루프에서 처리 중 메시지 추가 (선택사항)

```
# AI 응답 생성  
print("\n🤖 처리 중...") # NEW 검색 시 시간이 걸릴 수 있으므로  
response = manager.chat(user_input)  
print(f"\nAI: {response}\n")
```

3. 종료 메시지 수정

```
print()  
print("=" * 60)  
print("👋 대화를 종료합니다. 안녕히 가세요!")  
print(f"총 대화: {manager.get_message_count()}회")  
print(f"총 검색: {manager.get_search_count()}회") # NEW  
print("=" * 60)  
print()
```

Part 2 완료 체크리스트

9 Steps Completed

✓ 전체 조건: Part 1 (SearchAgent 구현 및 동작 확인) 및 기본 설정 완료

All Systems Go

Step 1: prompts.py

- ✓ `SYSTEM_MESSAGE_V2` 추가
- ✓ 도구 사용 원칙(Do/Don't) 포함
- ✓ 응답 형식/출처 가이드 포함

Step 2: Manager (기반)

- ✓ `SearchAgent / Tools import`
- ✓ `enable_search` 파라미터 추가
- ✓ `self.search_agent` 초기화
- ✓ 중간 로딩 테스트 성공

Step 3: _call_api_with_tools

- ✓ `call_params` 딕셔너리 구성
- ✓ `tools` 조건부 파라미터 추가
- ✓ `tool_choice="auto"` 설정
- ✓ API 호출 및 응답 반환

Step 4: _handle_tool_calls

- ✓ Assistant 메시지(tool_calls) 저장
- ✓ Tool Call 순회 및 `_execute`
- ✓ `role="tool"` 메시지 추가
- ✓ API 재호출 및 최종 응답 저장

Step 5: _execute_tool

- ✓ `search_web` 도구 분기 처리
- ✓ `SearchAgent.search()` 호출
- ✓ `format_for_llm()` 포맷팅
- ✓ 예외 처리(try-except) 구현

Step 6: chat() 수정

- ✓ `_call_api_with_tools()` 교체
- ✓ `tool_calls` 존재 여부 확인
- ✓ 분기: 핸들러 호출 vs 일반 응답
- ✓ 대화 흐름 로깅 추가

Step 7: 헬퍼 메서드

- ✓ `get_last_search_sources()`
- ✓ `get_search_count()`
- ✓ `is_search_enabled()`
- ✓ history/save 메서드 수정

Step 8: main.py

- ✓ 환경 메시지(v2.0) 업데이트
- ✓ `sources` 명령어 구현
- ✓ `status` 명령어 구현
- ✓ `enable_search=True` 설정

Step 9: 통합 테스트

- ✓ 검색 트리거/일반 질문 구분 성공
- ✓ `sources / status` 명령어 확인
- ✓ 연속 검색(Context) 동작 성공
- ✓ `clear / save / quit` 동작 확인

통합 테스트 (1/3) - 기본 동작

> 테스트 방법

```
$ python main.py
```

시나리오 1: 검색이 필요한 질문

YOU "2024년 AI 에이전트 트렌드에 대해 조사해줘"

예상 동작:

터미널에 "🔍 처리 중..." 상태 메시지 출력

로그: 도구 호출 감지: 1개 출력

로그: 도구 실행: search_web(...) 출력

로그: 검색 완료: N개 결과, X.XX초 출력

AI 응답 내용에 최신 검색 정보가 포함되어야 함

응답 하단에 [참고 출처] 목록이 명시되어야 함

시나리오 2: 검색이 불필요한 질문

YOU "Python의 for문 문법을 설명해줘"

예상 동작:

1. 도구 호출 없이 바로 응답
2. 일반적인 Python 문법 설명

시나리오 3: sources 명령어

YOU "source"

예상 동작:

시나리오 1 이후: 출처 목록 출력

시나리오 2 이후: "검색 기록이 없습니다." 출력

시나리오 4: status 명령어

YOU "status"

예상 동작:

현재 상태:

- 검색 기능: 활성화
- 대화 횟수: 2회
- 검색 횟수: 1회

통합 테스트 (2/3) - 엣지 케이스

시나리오5 연속 검색

YOU 최신 머신러닝 트렌드 알려줘

YOU 딥러닝과 차이점도 검색해줘

예상 동작:

두 번 다 검색 실행

- status에서 검색 횟수 2회 확인
- sources에서 두 번째 검색 출처 출력

시나리오6 검색 후, 일반 질문

YOU 테슬라 주가 최신 정보 알려줘

YOU 고마워

예상 동작:

- 첫 번째: 검색 실행

- 두 번째: 검색 없이 응답

시나리오7 clear 후 상태

YOU clear

YOU status

예상 동작:

대화 횟수: 0회

검색 횟수: 0회

시나리오8 save 후 파일 확인

YOU save

예상 동작:

data/conversation_YYYYMMDD_HHMMSS.json 파일 생성
파일에 search_enabled, search_count 필드 포함

통합 테스트 (3/3) - 엣지 케이스

시나리오9 종료

YOU quit

대화를 저장하시겠습니까? (y/n): n

예상 동작:

총 대화 횟수 출력

총 검색 횟수 출력

문제 해결 가이드

테스트 중 발생할 수 있는 문제들과 해결 방법

⚠ 문제 1: ImportError - SearchAgent

- ⌚ 증상 `from src.search_agent import ...` 실패
- 🔍 원인 Part 1 미완료 또는 파일 경로/파일명 오류
- ✓ 해결 Part 1 완료 여부 및 `src/search_agent.py` 존재 확인

⚠ 문제 2: 도구가 호출되지 않음

- ⌚ 증상 검색이 필요한 질문에도 도구 호출 없이 일반 응답만 출력됨
- 🔍 원인 `tools` 파라미터 미전달 또는 `tool_definitions` 설명 부족
- ✓ 해결 `_call_api_with_tools`에서 tools 전달 및 import 확인

⚠ 문제 3: tool_call_id 오류

- ⌚ 증상 OpenAI API 400 오류 (Invalid tool_call_id)
- 🔍 원인 `_handle_tool_calls` 처리 시 원본 ID와 불일치
- ✓ 해결 `tool_call.id` 값을 변형 없이 그대로 사용하는지 확인

🛠 추가 이슈 및 디버깅

데이터 처리 관련

</> 문제 4: JSON 파싱 오류

- ⌚ 증상 `json.loads(arguments)` 실행 시 Decode Error 발생
- 🔍 원인 라이브러리 버전에 따라 `arguments` 가 이미 Dict 타입인 경우
- ✓ 해결 `type()` 확인 후 문자열일 때만 파싱하도록 조건부 처리

🔗 문제 5: 검색 결과가 응답에 반영 안 됨

- ⌚ 증상 검색 로그는 뜨지만 AI 최종 응답에 정보가 포함되지 않음
- 🔍 원인 Tool 결과 메시지의 `role` 또는 `tool_call_id` 매칭 실패
- ✓ 해결 메시지 추가 시 `role="tool"` 및 올바른 ID 사용 확인

💡 Debugging Tip

문제가 지속될 경우 `logger.info()`를 사용하여 messages 리스트 전체를 출력해보세요. 대화 맥락이 API에 올바르게 전달되고 있는지 확인할 수 있습니다.

05

SECTION

LangChain 프레임워크 소개

선택 학습: 프레임워크 관점에서 에이전트 개발 방법 이해

☰ LEARNING OBJECTIVES

- ✓ LangChain 개요 이해
- ✓ 코드 비교 (Direct vs Framework)
- ✓ Agent 구조 및 구성요소
- ✓ 장단점 분석 및 참고자료

KEYWORDS:

● Framework

● Agent

● Comparison

LangChain이란?

프레임워크 개요

LangChain은 LLM 기반 애플리케이션 개발을 위한 오픈소스 프레임워크입니다. 복잡한 AI 워크플로우를 쉽게 구축할 수 있는 추상화 레이어를 제공하여 개발 생산성을 높입니다.

핵심 컴포넌트 (Core Components)

컴포넌트	설명	우리 프로젝트 대응
Models	LLM 래퍼 (Wrapper)	OpenAI Client
Prompts	프롬프트 템플릿	config/prompts.py
Chains	컴포넌트 연결	ConversationManager
Agents	자율 실행 에이전트	4주차 구현 예정
Tools	외부 도구 연동	SearchAgent (Tavily)

LLM 애플리케이션 개발을 위한 표준 프레임워크

LOW-LEVEL

Our Code

VS

HIGH-LEVEL

LangChain

OpenAI() → ChatOpenAI()

messages.append() → ConversationBufferMemory()

search_agent.search() → TavilySearchTool()

_handle_tool_calls() → create_tool_calling_agent()

LangChain은 복잡한 로직을 추상화하여 제공합니다.

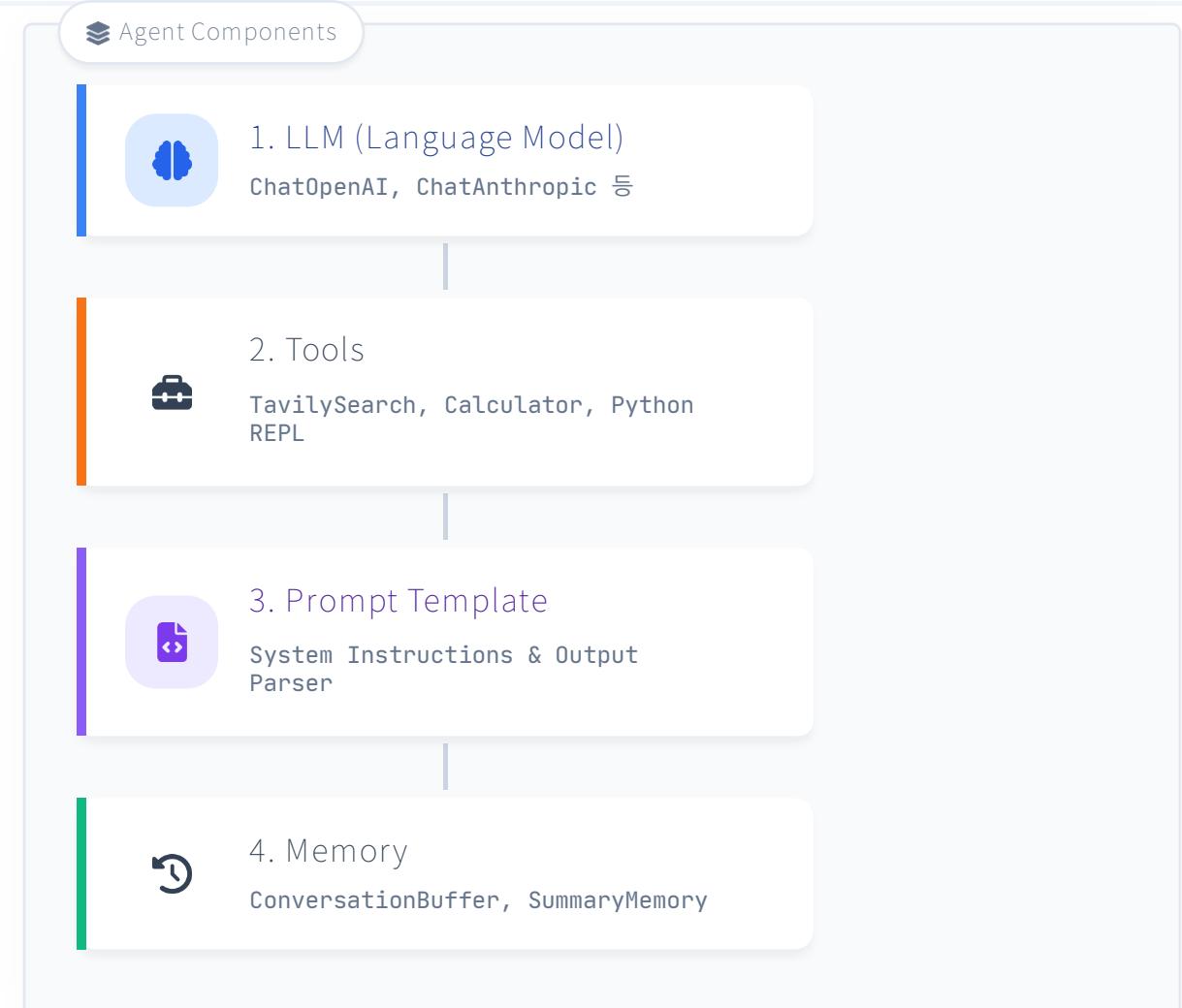
LangChain Agent 구조

🤖 Agent Types

목적에 따른 다양한 에이전트 구현 방식

타입 (Type)	설명 (Description)	사용 사례 (Use Case)
Tool Calling Agent	OpenAI Function Calling 기반의 가장 안정적인 에이전트	OpenAI 모델
ReAct Agent	Reasoning(추론) + Acting(행동) 패턴으로 문제 해결	복잡한 추론
Structured Chat	여러 입력 스키마를 가진 도구들을 구조화된 JSON으로 호출	다양한 모델

Note: 우리 프로젝트는 Function Calling 방식을 직접 구현하여 사용하고 있으며, 이는 LangChain의 Tool Calling Agent와 동일한 원리입니다.



LangChain 코드 비교

직접 구현(Low-level) vs LangChain(High-level)

</> 우리 코드 (직접 구현)

LOW-LEVEL CONTROL

```
from openai import OpenAI
from src.search_agent import SearchAgent

# 1. 초기화 & 도구 정의
client = OpenAI()
search_agent = SearchAgent()
tools = [SEARCH_WEB_TOOL] # JSON 스키마 직접 정의

# 2. API 호출 (tools 파라미터 전달)
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
    tools=tools
)

# 3. 도구 호출 처리 (수동 오케스트레이션)
if response:
    # 1) 도구 실행
    .choices[0].message.tool_calls:
    result = search_agent.search(query)
    # 2) 결과 포맷팅 및 히스토리 추가
    # 3) API 재호출하여 최종 응답 생성
```

@ LangChain 사용

HIGH-LEVEL ABSTRACTION

```
from langchain_openai import ChatOpenAI
from langchain_community.tools import TavilySearchResults
from langchain.agents import create_tool_calling_agent

# 1. 초기화 (도구 자동 로드)
llm = ChatOpenAI(model="gpt-4o-mini")
tools = [TavilySearchResults(max_results=5)]

# 2. 애이전트 생성 (프롬프트 + LLM + 도구 결합)
agent = create_tool_calling_agent(llm, tools, prompt)
executor = AgentExecutor(agent=agent, tools=tools)

# 3. 실행 (자동 오케스트레이션)
# 호출, 실행, 결과 처리, 재호출이 한 줄로 수행됨
result = executor.invoke({"input": "AI 트렌드 조사해줘"})
```



상세 제어 가능: 모든 실행 단계와 데이터 흐름을 직접 코드로 관리



높은 추상화: 복잡한 루프 로직을 캡슐화하여 생산성 향상

직접 구현 vs LangChain 장단점

</> 직접 구현 (CUSTOM)



학습 효과

LLM API 작동 원리와 Function Calling 메커니즘을 깊이 있게 이해할 수 있습니다.



완전한 제어

모든 비즈니스 로직을 직접 관리하며, 문제 발생 시 디버깅이 매우 용이합니다.



경량화

프로젝트에 꼭 필요한 기능만 구현하여 불필요한 의존성을 최소화합니다.



커스터마이징



이번 과정에서는 직접 구현을 통해 원리를 확실히 이해하고, 이후 실무 프로젝트에서는 LangChain을 적절히 병행 활용하여 생산성을 높이는 것이 좋습니다.

LANGCHAIN



빠른 프로토타이핑

적은 코드로 복잡한 기능을 구현할 수 있으며, 다양한 내장 도구를 즉시 활용 가능합니다.



풍부한 생태계

수많은 통합 모듈과 활발한 커뮤니티 지원으로 문제 해결이 빠릅니다.



높은 추상화

복잡한 로직을 캡슐화하여 사용하기 쉬우며, 다양한 LLM 모델 간 전환이 자유롭습니다.



문제점

LangChain 참고 자료

추가 학습을 위한 공식 문서 및 설치 가이드

공식 문서 (Official Docs)

LangChain Documentation	python.langchain.com/docs/
LangChain Agents	.../modules/agents/
Tool Calling	.../concepts/tool_calling/

추천 튜토리얼 (Tutorials)

자료명	설명	링크
LangChain Quickstart	프레임워크 기본 사용법 및 개념	DOCS
Building Agents	에이전트 구축 가이드 (심화)	DOCS
LangSmith	LLM 앱 디버깅 및 모니터링 도구	WEB

> Installation Guide

BASH

1. 기본 패키지 설치 (Core + OpenAI)

```
pip install langchain langchain-openai
```

2. Tavily 검색 도구 통합

```
pip install langchain-community tavily-python
```

3. 전체 패키지 (선택 사항)

모든 통합 기능을 포함하여 설치합니다.

```
pip install langchain[all]
```

설치 확인

```
python -c "import langchain; print(langchain.__version__)"
```

💡 실습 환경에 따라 가상환경(venv) 활성화 후 설치를 권장합니다.

06

SECTION

마무리 및 다음 주 예고

2주차 학습 정리 및 3주차 준비

☰ OBJECTIVES & PREVIEW

- ✓ 완성 기능 요약 (Summary)
- ✓ 3주차 예고: RAG (Preview)
- ✓ 핵심 학습 내용 리뷰 (Review)
- ✓ 3주차 예고: 벡터 DB (Vector)

KEYWORDS:

● Summary

● Preview

● RAG

2주차 완성 기능

이번 주 구현한 핵심 기능 및 시스템 완성도

☰ 구현 완료 항목 (COMPLETED ITEMS)

기능	설명	상태
SearchAgent	Tavily API 기반 검색 에이전트	DONE
Tool Definition	OpenAI Function Calling 도구 정의	DONE
통합 (Integration)	ConversationManager + SearchAgent	DONE
출처 관리	검색 결과 기반 출처 명시	DONE
에러 처리	검색 실패 및 예외 상황 대응	DONE

🏆 주요 성과 (KEY ACHIEVEMENT)

"에이전트가 실시간 웹 정보를 스스로 수집하고 활용하여, 사용자에게 근거 있는(Grounded) 답변을 제공할 수 있게 되었습니다."

SYSTEM STATUS



AI Research Assistant v2.0

ONLINE



자연스러운 대화 인터페이스

WEEK 01



전문 페르소나 (Research Assistant)

WEEK 01



웹 검색 및 정보 수집

WEEK 02



출처 기반 신뢰성 있는 응답

WEEK 02



Function Calling 기반 도구 사용

WEEK 02

2주차 핵심 학습 내용 정리

정보 수집 에이전트 구축 과정 요약



1. 이론 (TOOL-USING AGENT)

Function Calling 개념과 작동 원리 이해

도구 정의 (Name, Description, Parameters)

LLM의 도구 선택 메커니즘 (Reasoning)

효과적인 도구 설계 원칙



3. 실습 PART 2 (시스템 통합)

ConversationManager 확장 및 통합

Tool Calling 처리 흐름 (Handle → Execute)

도구 실행 결과의 재주입(Re-injection)

출처(Source)를 포함한 신뢰성 있는 응답



2. 실습 PART 1 (웹 검색)

Tavily Search API 연동 및 설정

SearchAgent 클래스 설계 및 구현

검색 결과 파싱 및 Markdown 포맷팅

사용자 의도에 맞는 쿼리 최적화 기법



4. LANGCHAIN 소개

LLM 애플리케이션 프레임워크 개념

직접 구현 vs LangChain 장단점 비교

핵심 컴포넌트: Agent, Tool, Chain

추상화를 통한 생산성 향상 이해



3주차 예고: 지식 베이스와 메모리

기억하고 학습하는 에이전트 구축 (RAG)

3주차 핵심 주제

주제	설명
RAG	검색 증강 생성 (Retrieval-Augmented Generation) 아키텍처
Vector DB	ChromaDB를 활용한 대규모 지식 저장 및 검색
Embedding	텍스트를 의미론적 벡터로 변환하는 기술
Memory	세션 간 대화 맥락 유지 및 장기 기억 구현

학습 목표

- RAG 개념 이해: LLM의 지식 한계를 극복하는 아키텍처 원리
- 벡터 DB 구축: 텍스트 데이터를 벡터화하여 저장 및 검색
- 임베딩 구현: OpenAI Embeddings API 활용 실습
- 영구 저장: 검색 결과와 대화 내용을 DB에 저장하여 재활용
- 히스토리 관리: 긴 대화 내용을 요약하고 문맥을 유지하는 기법

