생성형 AI 기반 RAG 개발자 과정

Tool 실전 — Search · Wiki · Math · arXiv · Custom

Session 26 / 33 — Day 4

DuckDuckGo · Tavily · Wikipedia · LLM-Math · arXiv API

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면 (1/2)

도구 호출의 원리 (먼저 짚고 넘어갈 것)

- ✅ **LLM은 도구를 직접 실행하지 않는다** — `tool_call` (JSON) 요청만 생성, 실행은 코드
- ✅ **bind_tools** 로 도구를 LLM 에 알리고, **ToolMessage** 로 결과를 회신
- ✅ **@tool 데코레이터** — 함수 이름·docstring·타입힌트가 곧 LLM 명세

도구별 깊이

- ✅ **Web Search** 3종 비교 — DuckDuckGo (무료) · Tavily (LLM 친화) · SerpAPI (정확)
- ✅ **Wikipedia** 한국어/영어 동시 활용
- ✅ **LLM-Math** — Python REPL 안전 실행
- ✅ **arXiv** API — 논문 검색 → 다운로드 → 요약 파이프라인
- ✅ **커스텀 도구** 5가지 패턴 — async · 에러처리 · 캐싱 · structured input/output

> 📂 소스: `2.langchain/8.agents/2~10.*`

이번 시간을 마치면 (2/2)

응용

- ✅ 도구 3개 조합 — "최신 LLM 논문 5개를 찾아 한국어 요약" 자동화
- ✅ 도구 에러 처리 — 외부 API 실패 시 graceful fallback
- ✅ Rate limit 대응 — exponential backoff in tools
- ✅ 회사 API 를 도구로

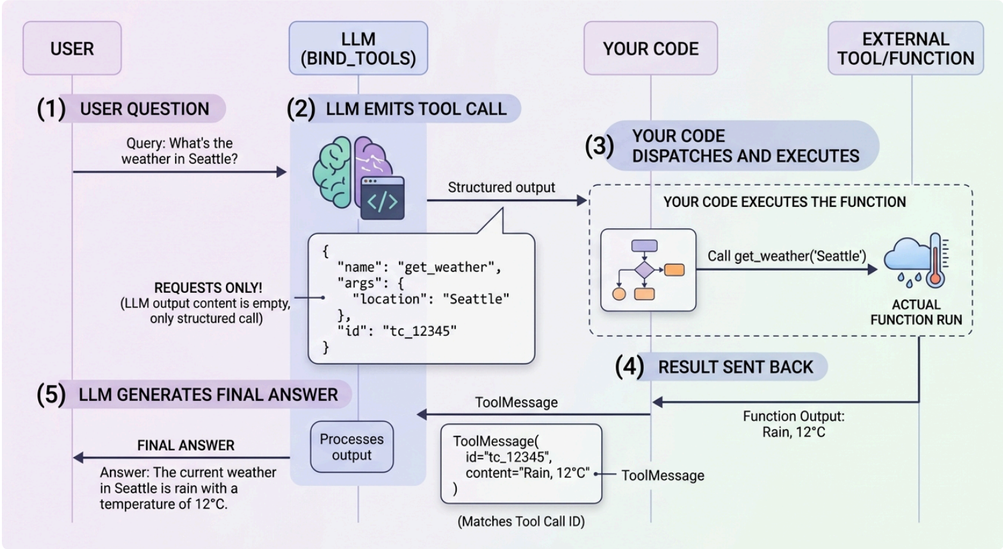
PART 0

도구 호출의 원리 — LLM은 실행하지 않는다

> 도구 5종에 앞서 도구 호출의 한 사이클 — 약 10분

tool_call(JSON)만 생성 — 실행은 코드의 몫

가장 먼저 바로잡을 오해: **LLM은 함수를 직접 실행하지 못한다.** LLM이 생성하는 것은 "이 함수를 이 인자로 호출하라"는 구조화된 요청(`tool_call`)뿐이며, 실제 실행은 코드가 담당한다.



> 도구 호출 흐름 · 질문→LLM이 tool_call(name·args·id) 생성→코드가 실행→ToolMessage 회신→LLM 최종 답변 (LLM은 호출만 요청, 실행은 코드)

4단계 한 사이클 — ① `bind_tools` 로 도구 알리기 → ② LLM이 `.tool_calls` 로 호출 요청 → ③ 코드가 실행하고 `ToolMessage` 로 회신 → ④ LLM이 결과 보고 최종 답. 이 사이클을 "더 부를 도구가 없을 때까지" 반복하면 그것이 ReAct 루프이자 Agent.

자동 루프를 걷어내고 직접 따라가기

```
# 2.langchain/8.agents/4.internals/4.1_bind_tools.py
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, ToolMessage

@tool
def add(a: int, b: int) → int:
    """두 정수 a 와 b 를 더한다."""
    return a + b

@tool
def multiply(a: int, b: int) → int:
    """두 정수 a 와 b 를 곱한다."""
    return a * b

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
llm_with_tools = llm.bind_tools([add, multiply])   # ① 도구 목록 알리기
```

⬇️ 코드가 길어 다음 장에 이어집니다.

자동 루프를 걷어내고 직접 따라가기 (이어서)

```
# ② 1차 호출 — LLM은 답 대신 "어떤 도구를 부를지" 결정
question = "3 더하기 5 를 계산하고, 그 결과에 7 을 곱해줘."
response = llm_with_tools.invoke([HumanMessage(content=question)])
print(response.content)        # 보통 빈 문자열 — 도구를 호출하므로
for call in response.tool_calls:
    print(f"{call['name']}({call['args']})  [id={call['id']}]")
    # → add({'a': 3, 'b': 5})  [id=call_abc...]

# ③ 코드가 실행하고 ToolMessage 로 회신 (tool_call_id 로 짝짓기)
tool_map = {"add": add, "multiply": multiply}
messages = [HumanMessage(content=question), response]
for call in response.tool_calls:
    result = tool_map[call["name"]].invoke(call["args"])   # 실제 실행
    messages.append(ToolMessage(content=str(result), tool_call_id=call["id"]))

# ④ 2차 호출 — 결과를 보고 최종 답변
final = llm_with_tools.invoke(messages)
print(final.content)   # → "8에 7을 곱하면 56입니다."
```

🎯 2차 응답에 또 `tool_calls` 가 있으면 ③~④를 반복 — 이 "없을 때까지 반복"이 ReAct 루프이고, 뒤에 나올 `create_agent` 가 이 반복을 자동으로 처리한다.

docstring·타입힌트는 LLM이 읽는 명세

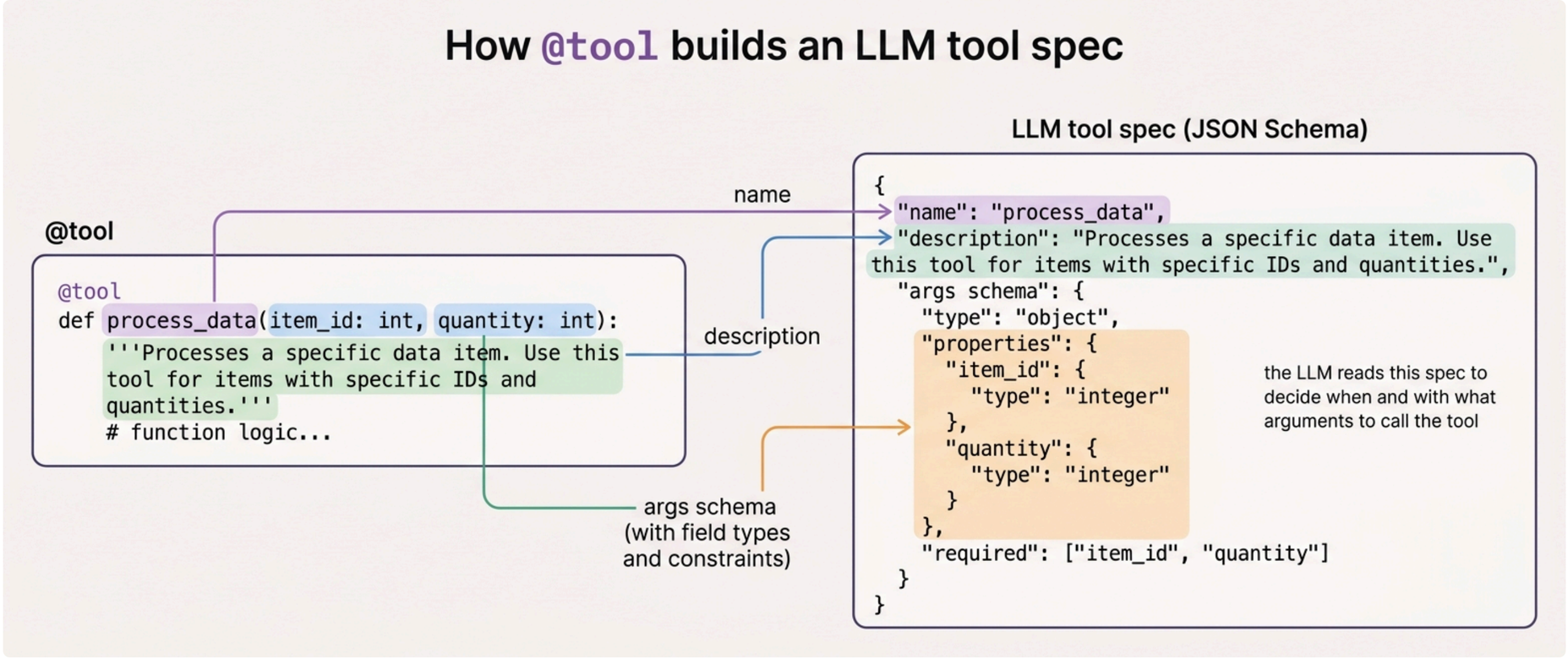
```
# 2.langchain/8.agents/2.custom_tools/2.2_at_tool_basic.py
from langchain_core.tools import tool

@tool
def get_word_length(word: str) → int:
    """단어의 글자 수를 센다."""        # ← LLM이 보는 "무엇을 하는가"
    return len(word)

# @tool 이 함수에서 자동으로 뽑아낸 명세 확인
print(get_word_length.name)         # get_word_length     ← 함수 이름
print(get_word_length.description)  # 단어의 글자 수를 센다.  ← docstring
print(get_word_length.args)         # {'word': {'type': 'string', ...}}  ← 타입힌트
```

LLM은 이 명세(JSON Schema)만 보고 **언제·어떤 인자로** 도구를 호출할지 판단한다. 따라서 **함수 이름·docstring·타입힌트의 설계가 곧 프롬프트 엔지니어링**이다. docstring이 모호하면 도구 선택 정확도가 떨어진다.

@tool 데코레이터 해부



@tool 데코레이터 · 함수 이름→도구 이름, docstring→설명, 타입힌트/Pydantic Field→인자 스키마(JSON Schema)

description이 도구 선택·인자 정확도를 좌우

```
# 2.langchain/8.agents/2.custom_tools/2.4_pydantic_args.py
from typing import Literal
from pydantic import BaseModel, Field
from langchain_core.tools import tool

class SendEmailInput(BaseModel):
    """이메일 전송 도구의 인자."""
    to: str = Field(description="수신자 이메일 주소 (반드시 유효한 이메일 형식)")
    subject: str = Field(description="이메일 제목 (50자 이내, 간결하게)")
    body: str = Field(description="이메일 본문 (반드시 한국어로 작성)")
    priority: Literal["low", "normal", "high"] = Field(
        default="normal",
        description="우선순위. urgent 한 경우에만 high 사용",
    )

@tool(args_schema=SendEmailInput)
def send_email(to: str, subject: str, body: str, priority: str = "normal") → str:
    """사용자가 요청하면 이메일을 보낸다."""
    return f"이메일이 {to} 에게 전송되었습니다 (priority={priority})."

# "alice@example.com 에게 '회의 일정 변경' 제목으로 보내줘. 긴급해."
#   → priority="high"   ("긴급해" 를 Literal enum 으로 인식)
```

description이 인자 정확도를 좌우

Pydantic이 주는 무기 3가지

- `Field(description=...)` — 인자마다 가이드 → LLM이 정확히 채움
- `Literal[...]` — 값을 enum 으로 강제, 정해진 값 외 거부
- `Field(ge=1, le=20)` — 범위·검증 자동 적용 (예: max_results 1~20)

> 🎯 도구가 자주 틀린 인자를 받는다면, 함수 본문보다 **description 부터** 손보라.

PART A

PART A
Web Search — 3종 비교

> LLM 에 연결할 검색 API 선택 기준 — 약 20분

DuckDuckGo · Tavily · Serper(SerpAPI)

도구	API Key	가격	결과 품질	LLM 친화도	권장
DuckDuckGo	❌ 필요 없음	무료	보통	보통	학습·실험
Tavily	필요	무료 1000회/월	좋음	★★★★★ (요약 포함)	**본 코스 권장**
SerpAPI / Serper	필요	월 $50+	매우 좋음	좋음	운영·정확도 우선
Brave Search	필요	무료 2000회/월	좋음	좋음	프라이버시 중시
Bing Search	필요	종량제	매우 좋음	좋음	MS 생태계
Google CSE	2개 키	무료 100회/일	좋음	보통 (raw HTML)	Google 결과 필요 시

Tavily 가 LLM 친화적인 이유

- 검색 결과 + **각 결과의 자동 요약** 포함
- "answer" 필드 — 검색 결과 종합한 LLM 답변 제공
- 토큰 절약 + 추가 LLM 호출 ↓

```
# 본 코스 권장 설치
pip install langchain-community duckduckgo-search tavily-python
```

같은 질의, 3가지 API

```
# ═══ 1. DuckDuckGo (무료, 키 X) ═══
from langchain_community.tools import DuckDuckGoSearchRun
ddg = DuckDuckGoSearchRun()
result = ddg.invoke("Llama 4 release date")
print(result)
# → 검색 결과 텍스트 문자열 (HTML snippet 들)

# === 2. Tavily ═══
# pip install langchain-tavily
from langchain_tavily import TavilySearch
tavily = TavilySearch(
    max_results=5,
    api_key=os.environ["TAVILY_API_KEY"],
    search_depth="advanced",   # basic | advanced
    include_answer=True,       # LLM 친화적 요약 답변 포함
)
result = tavily.invoke("Llama 4 release date and benchmark scores")
print(result["answer"])         # 종합 답변
print(result["results"])        # [{url, title, content, score}, ...]
```

> ⬇️ 코드가 길어 다음 장에 이어집니다.

같은 질의, 3가지 API (이어서)

```
# ═══ 3. Serper (SerpAPI 의 저렴 대안) ═══
# pip install google-search-results
from langchain_community.utilities import GoogleSerperAPIWrapper
serper = GoogleSerperAPIWrapper(serper_api_key=os.environ["SERPER_API_KEY"])
result = serper.run("Llama 4 release date")
print(result)

# ═══ @tool 로 감싸서 Agent 에 ===
from langchain_core.tools import tool

@tool
def web_search(query: str) → str:
    """최신 정보 (뉴스·릴리스·실시간 데이터) 가 필요할 때 사용.
    Wikipedia 로는 못 찾는 2024년 이후 정보 위주."""
    return tavily.invoke(query)["answer"]
```

본 코스 권장 패턴

- **학습용**: DuckDuckGo (키 없이 즉시)
- **본격 실습**: Tavily (월 1000회 무료, LLM 친화)
- **운영 정확도**: Serper / SerpAPI

PART B

PART B
Wikipedia — 사실 확인의 표준

한국어/영어 동시 활용 — 약 10분

2.2_wikipedia1_ko.py 패턴

```
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper
from langchain_core.tools import tool

# ═══ 기본 — 영어 위주 ===
wiki_en = WikipediaQueryRun(
    api_wrapper=WikipediaAPIWrapper(lang="en", top_k_results=3, doc_content_chars_max=2000)
)

# ═══ 한국어 ===
wiki_ko = WikipediaQueryRun(
    api_wrapper=WikipediaAPIWrapper(lang="ko", top_k_results=3, doc_content_chars_max=2000)
)

# ═══ 다국어 — fallback 패턴 ===
@tool
def wikipedia_search(query: str) → str:
    """위키피디아에서 인물·역사·과학·예술 등 사실 정보 조회.
    먼저 한국어 검색, 없으면 영어 검색."""
    ko_result = wiki_ko.invoke(query)
    if "검색 결과 없음" in ko_result or len(ko_result) < 100:
        # 영어로 fallback
        return f"[en] {wiki_en.invoke(query)}"
    return f"[ko] {ko_result}"

# 사용 — Agent 에서 자동 호출
result = wikipedia_search.invoke({"query": "Claude Shannon"})
print(result)
```

Wikipedia 도구 — 운영 시 유의점

Wikipedia 사용 시 한 줄 주의

- 정보 누락 흔함 — 작은 인물·최신 사건 미수록
- 페이지 변경 시 결과 다름 — 운영에서는 캐싱 권장
- 본 코스 권장 조합: **Wikipedia (사실) + Web Search (최신) 둘 다**

PART C

PART C
Math — 안전한 계산

LLM 직접 계산의 한계와 위임 — 약 8분

1.1_llmmath.py 패턴

```
from langchain_community.agent_toolkits.load_tools import load_tools
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
tools = load_tools(["llm-math"], llm=llm)
# → LLMMathChain — LLM 이 수식을 Python 코드로 변환 → eval 실행

# 또는 더 안전한 PythonREPLTool (sandboxed 권장)
from langchain_experimental.tools import PythonREPLTool

@tool
def safe_calc(expression: str) → str:
    """수학 계산이 필요할 때 사용. Python 수식 그대로 (예: '2 ** 10 + 3').
    절대 LLM 이 직접 계산하지 말 것 — 산술 오류 흔함."""
    try:
        result = eval(expression, {"__builtins__": {}}, {})   # 제한된 eval
        return f"= {result}"
    except Exception as e:
        return f"계산 오류: {e}"
```

계산 위임의 근거 + Python REPL 대안

LLM 직접 계산의 위험성

```
LLM 단독: "12345 * 6789 = ?" → "약 8천 3백만" (오답, 정답: 83,810,205)
LLM-Math: "12345 * 6789" → Python eval → "83810205" (정답)
```

대안 — Python REPL 도구 (더 강력)

```
from langchain_experimental.tools import PythonREPLTool
# LLM 이 임의의 Python 코드 실행 — numpy, pandas 까지
# ⚠️ 보안 위험 — 운영에서는 컨테이너 sandbox 필수
```

> 💡 운영: **수식만 받는 안전한 도구** + 별도 **sandboxed Python** (`docker run --rm`) 분리.

PART D

PART D
arXiv — 논문 자동화

> 검색 → 다운로드 → 요약 한 흐름 — 약 20분

3.1_arxiv_thesis.py — arXiv API 직접

```
from langchain_core.tools import tool
import arxiv   # pip install arxiv

@tool
def search_arxiv(query: str, max_results: int = 5) → list[dict]:
    """arXiv 에서 학술 논문 검색. 최신 ML/AI 연구 정보가 필요할 때.

    Args:
        query: 검색어 (영문 권장)
        max_results: 반환할 논문 수 (기본 5)
    """
    client = arxiv.Client()
    search = arxiv.Search(
        query=query,
        max_results=max_results,
        sort_by=arxiv.SortCriterion.SubmittedDate,
        sort_order=arxiv.SortOrder.Descending,
    )
    results = []
    for paper in client.results(search):
        results.append({
            "id":       paper.entry_id.split("/")[-1],      # "2406.12345v1"
            "title":    paper.title,
            "authors":  [a.name for a in paper.authors],
            "summary":  paper.summary[:400] + "…",
            "pdf_url":  paper.pdf_url,
            "published": paper.published.strftime("%Y-%m-%d"),
            "categories": paper.categories,
```

`search_arxiv` 호출 결과

```
# 사용
papers = search_arxiv.invoke({"query": "RAG retrieval augmented generation 2024", "max_results": 3})
for p in papers:
    print(f"[{p['published']}] {p['title']}")
    print(f"  → {p['pdf_url']}")
```

3.2~3.4 — PDF 다운로드 · 한국어 요약 도구

```
from langchain_core.tools import tool
from langchain_community.document_loaders import PyPDFLoader
from langchain_openai import ChatOpenAI
import arxiv, tempfile
from pathlib import Path

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

@tool
def download_arxiv_paper(arxiv_id: str) → str:
    """arXiv ID 로 PDF 다운로드 후 본문 텍스트 반환. 약 30초 소요."""
    paper = next(arxiv.Client().results(arxiv.Search(id_list=[arxiv_id])))
    tmpdir = Path(tempfile.mkdtemp())
    pdf_path = tmpdir / f"{arxiv_id}.pdf"
    paper.download_pdf(dirpath=str(tmpdir), filename=pdf_path.name)
    pages = PyPDFLoader(str(pdf_path)).load()
    return "\n\n".join(p.page_content for p in pages[:10])   # 처음 10페이지

@tool
def summarize_paper_ko(text: str) → str:
    """긴 영문 논문을 한국어로 5문단 요약 (배경·문제·방법·결과·의의)."""
    prompt = f"""다음 영문 논문을 한국어로 5단락 요약 (배경·문제·방법·결과·의의):

{text[:8000]}
```

> 다음 슬라이드에서 이 두 도구를 `search_arxiv` 와 묶어 Agent 에 연결한다.

도구 3개를 묶어 Agent 가 순차 호출

```
from langchain.agents import create_agent

# search_arxiv(앞 슬라이드) + download + summarize 를 한 Agent 에
agent = create_agent(llm, [search_arxiv, download_arxiv_paper, summarize_paper_ko])

result = agent.invoke({
    "messages": [{
        "role": "user",
        "content": "RAG 평가에 관한 2024년 이후 논문 1개를 찾아 한국어로 요약해줘."
    }]
})
```

Agent 가 자동으로 수행

1. `search_arxiv("RAG evaluation 2024")` → 5개 후보
2. 가장 관련 높은 ID 선택 (예: `2407.12345`)
3. `download_arxiv_paper("2407.12345")` → PDF 본문
4. `summarize_paper_ko(text)` → 5단락 한국어 요약
5. 최종 답변

> 🎯 **이 패턴이 Day 4 의 핵심** — 도구 N개를 LLM 이 순차 호출해 복잡한 작업을 자동화한다.

PART E

PART E
커스텀 도구 — 5가지 패턴

내 도메인 도구 작성의 베스트 프랙티스 — 약 20분

I/O 많은 도구 = async

```
from langchain_core.tools import tool
import httpx
import asyncio

@tool
async def fetch_url_async(url: str) → dict:
    """주어진 URL 의 본문을 가져온다. 비동기 (병렬 호출에 유리)."""
    async with httpx.AsyncClient(timeout=10) as client:
        r = await client.get(url)
        return {"status": r.status_code, "content": r.text[:5000]}
```

동기 vs 비동기 — 선택 기준

작업	권장
빠른 계산 (math, regex)	동기
외부 API 호출	**비동기** (특히 여러 도구 병렬)
DB 쿼리	비동기 (sqlalchemy async, asyncpg)
파일 I/O	동기 OK (보통 빠름)
LLM 안에서 LLM 호출	비동기

> 💡 **Tip**: 모든 도구를 비동기로 만들면 호환성 ↓. **명백히 I/O 많은 것만 async**.

외부 API 실패 시 graceful fallback

```
from langchain_core.tools import tool

@tool
def get_weather(city: str) → str:
    """도시의 날씨 정보."""
    try:
        r = requests.get(f"https://api.weather.example/v1/{city}", timeout=5)
        r.raise_for_status()
        data = r.json()
        return f"{city}: {data['temp']}°C, {data['condition']}"

    except requests.Timeout:
        return f"날씨 API 시간 초과. 다른 도구나 추론으로 답변하거나 사용자에게 다시 시도하라고 안내하세요."

    except requests.HTTPError as e:
        if e.response.status_code == 404:
            return f"도시 '{city}' 정보를 찾을 수 없습니다. 영문 도시명으로 재시도 권장."
        return f"날씨 서비스 오류 ({e.response.status_code}). 잠시 후 재시도."

    except Exception as e:
        return f"날씨 조회 실패: {type(e).__name__}. 사용자에게 알리세요."
```

에러도 LLM 이 보는 Observation

핵심 — Agent 에게 "다음에 무엇을 하라" 힌트 주기

잘못된 에러 처리	좋은 에러 처리
`raise Exception`	에러 메시지 반환 + 다음 액션 힌트
빈 문자열 반환	"정보 없음. 다른 도구를 시도하세요."
예외를 삼킴	사용자가 알아야 할 정보를 메시지에

> 🎯 **도구 반환값은 LLM 이 보는 Observation**. 에러도 LLM 이 다음 결정에 활용할 수 있도록 정보를 충분히 담아라.

같은 호출 반복 방지 + API 한도 보호

```
from langchain_core.tools import tool
from functools import lru_cache
import time, random
import requests

# ═══ 캐싱 — 같은 호출 메모이즈 ===
@lru_cache(maxsize=1000)
def _cached_lookup(query: str) → str:
    # 실제 외부 호출
    return requests.get(f"https://api.example/{query}").text

@tool
def expensive_lookup(query: str) → str:
    """비싼 외부 API. 같은 query 는 캐시."""
    return _cached_lookup(query)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

같은 호출 반복 방지 + API 한도 보호 (이어서)

```
# ═══ Rate Limit — Exponential Backoff ═══
def with_backoff(fn, max_retries=5):
    def wrapper(*a, **kw):
        delay = 1.0
        for attempt in range(max_retries):
            try:
                return fn(*a, **kw)
            except requests.HTTPError as e:
                if e.response.status_code ≠ 429:
                    raise
                if attempt ═ max_retries - 1:
                    raise
                sleep_s = delay * (1 + random.random() * 0.3)
                time.sleep(sleep_s)
                delay *= 2
    return wrapper

@tool
@with_backoff
def rate_limited_api(query: str) → str:
    """Rate limit 가 엄격한 외부 API. 429 시 자동 재시도."""
    r = requests.get(f"https://strict-api.example/{query}")
    r.raise_for_status()
    return r.text
```

`lru_cache` 는 만료 없는 영구 캐시 — 시간 만료가 필요하면 다음 슬라이드의 TTL 캐시.

1시간 후 자동 만료되는 캐시

```
# pip install cachetools
from cachetools import TTLCache, cached
import requests

# maxsize=1000 개까지, 각 항목 3600초(1시간) 후 만료
_cache = TTLCache(maxsize=1000, ttl=3600)

@cached(_cache)
def _ttl_lookup(q: str) → str:
    return requests.get(f"https://api.example/{q}").text
```

`lru_cache` vs `TTLCache` 선택 기준

상황	권장
결과가 거의 안 변함 (단어 길이, 환율 표 등)	`lru_cache` (영구)
시간 지나면 낡는 데이터 (뉴스·날씨·재고)	`TTLCache` (ttl 설정)
프로세스 재시작 후에도 유지 필요	Redis 등 외부 캐시

> 🎯 **도구 캐싱의 효과는 비용·속도 둘 다** — 같은 질의 반복이 잦은 Agent 일수록 체감이 크다.

Pydantic 으로 도구 인터페이스 명시

```
from pydantic import BaseModel, Field
from typing import Literal
from langchain_core.tools import tool

class SearchInput(BaseModel):
    """Web search 도구 입력 스키마."""
    query: str = Field(description="검색어 (영문 권장)")
    region: Literal["us", "kr", "jp", "global"] = Field(
        default="global",
        description="검색 지역. 한국 뉴스는 'kr'.",
    )
    max_results: int = Field(default=5, ge=1, le=20)
    date_range: Literal["d", "w", "m", "y", None] = Field(
        default=None,
        description="d=하루, w=주, m=월, y=년, None=전체",
    )
```

⬇️ 코드가 길어 다음 장에 이어집니다.

Pydantic 으로 도구 인터페이스 명시 (이어서)

```
class SearchResult(BaseModel):
    """Web search 결과 스키마."""
    title: str
    url: str
    snippet: str
    published_date: str | None = None

@tool("web_search", args_schema=SearchInput)
def web_search(query: str, region: str = "global",
               max_results: int = 5,
               date_range: str | None = None) → list[dict]:
    """웹 검색. 최신 정보·뉴스·릴리스가 필요할 때."""
    # ... 실제 검색
    return [SearchResult(...).model_dump() for r in results]
```

효용 4가지

1. LLM 이 정확한 형식으로 호출 (`region: "kr"`)
2. 잘못된 값 (`region: "korea"`) 자동 거부 → 재시도 유도
3. 자동 문서화 — Pydantic 스키마 = JSON schema
4. IDE 자동완성 — 타입 힌트 그대로

문서 검색 + Agent 의 결합

```
from langchain_core.tools import tool
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings

# 색인해 둔 내 문서 로드
vectorstore = Chroma(
    persist_directory="./chroma_db",
    collection_name="my_library",
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-small"),
)

@tool
def search_my_docs(question: str, k: int = 4) → str:
    """**내 문서**(PDF·노트)에서 관련 내용을 검색.
    회사 위키 / 학과 노트 / 책 챕터가 들어있음.

    외부 정보 (Wikipedia, 뉴스) 는 다른 도구 사용.
    수학 계산은 calc 도구."""
    docs = vectorstore.similarity_search(question, k=k)
    return "\n\n".join(
        f"[{d.metadata.get('doc_id')} p.{d.metadata.get('page')}]\n{d.page_content}"
        for d in docs
    )
```

⬇️ 코드가 길어 다음 장에 이어집니다.

문서 검색 + Agent 의 결합 (이어서)

```
# Agent 에 통합 — 5개 도구
from langchain.agents import create_agent
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")
agent = create_agent(llm, [
    search_my_docs,       # 내 문서 검색
    web_search,           # 최신 정보
    wikipedia_search,     # 사실 확인
    search_arxiv,         # 논문
    safe_calc,            # 수학
])

# 복합 질의
result = agent.invoke({"messages": [{"role": "user", "content":
    "내 노트의 RAG 평가 메트릭이랑 2024년 이후 arXiv 논문의 RAG 평가 방법을 비교해줘"
}]})
```

> 🎯 **Day 2 + Day 4 의 가장 강력한 조합** — 내부 지식과 외부 정보를 LLM 이 한데 모아 답한다.

PART F

PART F
미니 실습

> 워크플로 자동화 — 약 30분

실습 과제 (30분) — 1·2단계

1단계 — 워크플로 1개 선정 (5분)

일주일에 1~2회 반복 수행하는 작업 1개.

예시: - "분야별 최신 논문 5개 → 한국어 요약 → 노션에 정리" - "경쟁사 신규 제품 뉴스 모니터링 → Slack 알림" - "코드베이스 + arXiv 검색 → 새 기법 적용 가능성 평가"

2단계 — 도구 4~5개 조합 설계 (10분)

```
tools = [
    search_my_docs,       # 내 문서 검색
    web_search,           # 최신 뉴스
    wikipedia_search,
    search_arxiv,
    download_arxiv_paper,
    summarize_paper_ko,
]

agent = create_agent(llm, tools)
```

실습 과제 (30분) — 3·4단계

3단계 — 실행 + Trace 분석 (10분)

- Agent invoke
- LangSmith 또는 verbose 로그로 trace 캡처
- 도구별 호출 순서·인자 도식화

4단계 — 실패 패턴 1건 찾고 수정 (5분)

- 도구 description 수정으로 해결
- 또는 system prompt 강화

결과 보고

- 워크플로 자동화 시간 — 사람 vs Agent
- 신뢰할 수 없는 결과 — 어떤 검증이 추가로 필요한가

핵심 8가지

1. ✅ Search 3종 비교 — **Tavily 본 코스 권장** (LLM 친화)
2. ✅ Wikipedia 다국어 fallback 패턴
3. ✅ LLM-Math = Python REPL 위임 — LLM 직접 계산 X
4. ✅ arXiv 자동화 — 검색→다운로드→요약 파이프라인
5. ✅ 커스텀 도구 5 패턴 — async · 에러 · 캐싱/RL · structured · RAG 도구화
6. ✅ 에러 메시지도 **LLM 이 보는 Observation** — 다음 액션 힌트 포함
7. ✅ Pydantic 스키마로 도구 인터페이스 명시
8. ✅ **RAG 를 도구로** 변환 = 검색·계산·문서를 아우르는 강력한 조합

실습 과제 — 직접 해보기

필수

- [ ] 워크플로 자동화 1개 — 도구 4+ 조합
- [ ] Trace 1건 캡처

옵션

- [ ] Tavily API Key 발급 (월 1000 무료) — 무료 검색 도구로 업그레이드
- [ ] 회사 내부 API 1개를 `@tool` 로 감싸기
- [ ] **LangSmith** 가입 + trace 자동 수집 설정

📤 제출

- 자동화 시나리오 1개 + 도구 코드 + trace 스크린샷