생성형 AI 기반 RAG 개발자 과정

AI Agent 입문

Session 25 / 33 — Day 4

Agent · ReAct · Tool Calling · Function Calling

박수현 · (사)한국정보공학기술사회 · 2026

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

이번 시간을 마치면 — 원리

원리

- ✅ **Agent** 의 정의 — 단순 LLM 호출과의 차이
- ✅ **ReAct 패턴** — Reason → Act → Observe 루프
- ✅ **Tool Calling** = OpenAI Function Calling 의 표준화
- ✅ Tool 선택 알고리즘 — LLM 의 도구 선택 메커니즘
- ✅ `create_agent` — 현행 표준 API (옛 `AgentExecutor` 와의 차이)

> 📂 소스: `2.langchain/8.agents/0~2.*`

이번 시간을 마치면 — 실습

실습

- ✅ 첫 Agent — `create_agent` 로 Wikipedia + LLM-Math 조합
- ✅ ReAct trace 직접 보기 — `result["messages"]` 출력
- ✅ OpenAI Function Calling 직접 — `tools` 파라미터
- ✅ LangChain `@tool` 데코레이터로 커스텀 도구
- ✅ 에이전트가 실패하는 패턴 5종 (관찰 누락·무한루프 등)

> 📂 소스: `2.langchain/8.agents/0~2.*`

PART A

PART A
Agent — 일반 LLM 과의 차이

> 도구를 쓰는 LLM — 약 15분

일반 LLM 호출 vs Agent

일반 LLM 호출 (Session 1~3)

```
질문 → [LLM] → 답변
```

- 1회 호출, 1회 응답
- 모델 학습 시점 이후 정보 없음
- 계산·검색·실시간 정보 불가
- "2026년 1월의 KOSPI 종가는?" → 모름

Agent

```
질문 → [LLM 사고]
         ↓
       "도구 필요?"
         ↓
      [Tool 호출] → 도구 결과
         ↓
       [LLM 사고] (결과 보고)
         ↓
       "더 필요?" → [Tool 호출] (반복)
         ↓
       [LLM 답변]
```

- 다회 호출, 여러 도구 사용
- 외부 정보·계산·실행 결과 활용
- "2026년 KOSPI?" → 검색 도구 호출 → 답변

Agent 로 가능해지는 작업

가능해지는 작업 5가지

작업	사용 도구
실시간 정보	Web Search, News API
수학·통계	Python Calculator
사실 확인	Wikipedia, arXiv
코드 실행	Python REPL, shell
내 분야 지식	RAG (Day 2!), 회사 API

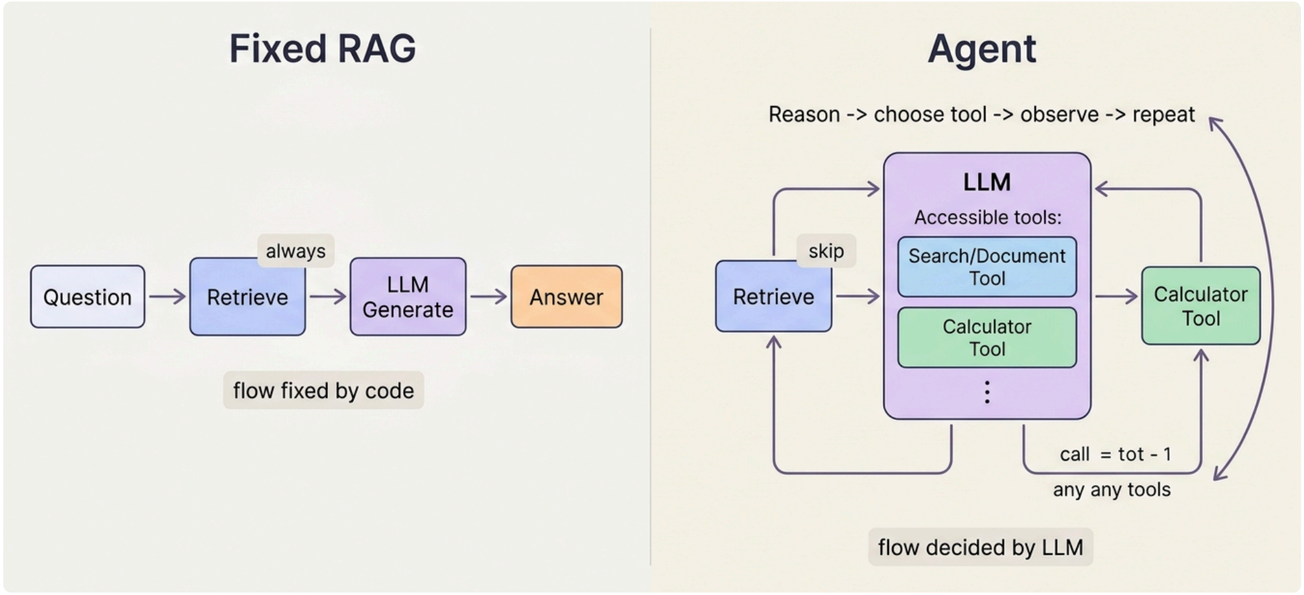
한 줄 정의

> **Agent = LLM + Tool Loop + Decision**
>
> 🎯 Day 4 의 Agentic RAG 도 결국 Agent 의 한 형태 — "검색이 필요한가" 를 판단하는 단순한 agent.

고정 RAG 의 숨은 전제

앞서 다룬 RAG 는 **어떤 질문이 들어와도 항상 "검색 → 생성"** 이라는 같은 순서를 밟는다. "안녕하세요" 같은 인사에도, "2 더하기 3은?" 같은 계산에도 무조건 벡터 DB 를 검색한다.

```
# 고정 RAG — 질문 종류와 무관하게 항상 같은 순서
```



> 고정 RAG(질문→항상 검색→생성) vs 에이전트 루프(LLM이 도구를 골라 반복 호출) · 흐름 결정 주체가 코드 → LLM으로 이동

흐름 결정을 코드에서 LLM 으로

고정 흐름의 한계 — 검색이 불필요한 질문(인사·단순 계산)에도 비용·지연 / 한 번 검색으로 부족해도 재시도 못 함 / 검색 외 능력(계산·날씨·최신 뉴스)에 무력

> 🎯 **에이전트** 는 이 순서를 뒤집는다. **무엇을 할지(검색·계산·즉답)를 코드가 아니라 LLM 이 매 순간 결정** — 개발자는 순서 대신 **도구(Tool) 목록** 만 제공한다.

Agent 를 이루는 다섯 가지

컴포넌트	역할	LangChain 구현
LLM	사고·결정의 두뇌	`ChatOpenAI`, `ChatAnthropic`, `ChatOllama`
Tools	LLM 이 호출할 수 있는 함수들	`Tool`, `@tool` 데코레이터
Prompt	시스템 페르소나 + ReAct 지시	`hub.pull("hwchase17/react")` 등
Memory	대화·중간 결과 보존	`MemorySaver` (LangGraph), `ChatMessageHistory`
Executor	루프 운영 — 결정·호출·관찰·반복	`create_agent` (1.x), `AgentExecutor` (0.x)

흐름 한 줄

```
LLM 이 prompt 따라 사고 → Tools 중 하나 호출 결정 → Executor 가 실행 → 결과 Memory 에 → LLM 다시 사고
```

LangChain 의 진화 (2024~2026)

- **0.x (deprecated)**: `initialize_agent`, `AgentExecutor` — 문자열 파싱, 커스텀 어려움
- **1.x (현재)**: `create_agent` + LangGraph — 표준화, 더 유연
- **LangGraph 만 사용**: 가장 세밀 제어 (Session 27)

> 💡 본 코스의 실습은 처음부터 `create_agent` (현행 권장) 로 통일한다. `initialize_agent`·`AgentExecutor` 는 옛 예제 독해를 위한 **참고용**으로만 짚고(다음 Part), 그 뒤 **LangGraph** (Session 27) 로 확장한다.

PART B

PART B
ReAct — Reason → Act → Observe

> Agent 작동의 표준 패턴 — 약 20분

"Reasoning + Acting in Language Models" (Yao et al. 2022)

핵심 아이디어

- LLM 이 **추론과 행동을 교차** 수행
- 각 단계에서 명시적으로 "Thought" 와 "Action" 을 출력
- Action 의 결과 ("Observation") 를 다음 사고에 반영

한 사이클 — 정확한 출력 형식

```
Question: 2024년 노벨 물리학상 수상자의 학력을 알려줘.

Thought: 먼저 2024년 노벨 물리학상 수상자를 알아야 한다. 위키피디아 검색.
Action: Wikipedia[2024 Nobel Prize Physics]
Observation: 2024년 수상자는 John Hopfield 와 Geoffrey Hinton.

Thought: 두 사람 중 Geoffrey Hinton 의 학력을 알아본다.
Action: Wikipedia[Geoffrey Hinton]
Observation: 케임브리지 대학 학사, 에든버러 대학 박사…

Thought: 정보 충분. 답변 작성.
Final Answer: Geoffrey Hinton 은 케임브리지 학사, 에든버러 박사…
```

ReAct 의 효과 — 4가지 이점

1. **투명성** — 모든 추론 과정이 보임 (디버그 가능)
2. **검증성** — 각 Action 결과가 다음 Thought 에 영향 → 잘못된 가정 자체 수정
3. **모듈성** — Tool 만 바꾸면 같은 패턴이 다른 도메인에서 동작
4. **확장성** — Thought-Action 쌍을 무한 반복 가능 (max_iterations 제한)

도구 목록 한 줄 — `create_agent`

```
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain_community.agent_toolkits.load_tools import load_tools
from langchain.agents import create_agent      # LangChain 1.x 현행 API

load_dotenv()

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

# 2개 도구 로드 — 위키피디아 + 수학 계산기
tools = load_tools(["wikipedia", "llm-math"], llm=llm)

# 도구 목록만 넘기면 ReAct 루프 전체가 자동 구성된다
agent = create_agent(llm, tools)

# 입력·출력은 messages 형식
result = agent.invoke({
```

`create_agent` 가 대신 처리하는 것

- "도구가 더 필요한가" 판단 → 도구 호출 → 결과 관찰 → 재호출 **루프 전체**
- 무한루프 방지·중단 조건 등 실행 제어 (내부 LangGraph 가 담당)

> 🎯 단 한 번의 `invoke` 로 위키피디아 검색 2회 + 계산기 1회가 자동 실행된다. 작성한 코드는 **도구 목록을 넘긴 한 줄** 뿐 — 내부 trace 는 다음 슬라이드에서 확인한다.

`result["messages"]` 에 쌓인 실제 ReAct 흐름

`create_agent` 는 `verbose` 같은 플래그 대신 **전체 대화를 messages 리스트로 반환한다.** 이 리스트를 출력하면 ReAct 루프의 진행 과정이 한눈에 드러난다.

```
for m in result["messages"]:
    # 도구 호출 (AI 가 "이 도구를 부르자" 고 결정한 순간)
    if getattr(m, "tool_calls", None):
        for c in m.tool_calls:
            print(f"[도구 호출] {c['name']}({c['args']})")
    if m.content:
        prefix = {"human": "[사용자]", "ai": "[AI]", "tool": "[도구 결과]"}.get(m.type, m.type)
        print(f"{prefix} {m.content}")
```

```
[사용자] 한국의 인구가 일본 인구의 몇 퍼센트인지 알려줘
[도구 호출] wikipedia({'query': 'South Korea population'})
[도구 결과] 약 5,200만 명 (2024 추정)
[도구 호출] wikipedia({'query': 'Japan population'})
[도구 결과] 약 1억 2,300만 명
[도구 호출] Calculator({'expression': '5200 / 12300 * 100'})
[도구 결과] 42.27642276422764
[AI] 한국 인구는 일본 인구의 약 42.3% 입니다.
```

🎯 `result["messages"]` 출력은 학습·디버깅의 핵심 습관이다. 운영에서는 LangSmith 로 동일한 trace 를 수집한다.

인터넷 옛 예제에서 보게 될 이전 세대 API

2024년 이전 글·튜토리얼은 대부분 아래 방식으로 에이전트를 구성한다. **현재는 사용하지 않지만**, 옛 예제 독해를 위해 형태만 짚어 둔다.

```
# ⚠️ 이전 세대 (deprecated) — 참고용, 실습에서는 쓰지 않는다
from langchain.agents import initialize_agent, AgentType, AgentExecutor

agent = initialize_agent(
    tools, llm,
```

세대 간 변경점

항목	이전 (0.x, deprecated)	현행 (1.x)
생성 함수	`initialize_agent` / `AgentExecutor`	`create_agent`
동작 방식	`Action:` **문자열 파싱** (자주 깨짐)	Function Calling 기반 (안정)
입력 / 출력	`{"input": ...}` / `result["output"]`	`{"messages": [...]}` / `result["messages"]`
내부 엔진	자체 루프	LangGraph (커스텀·체크포인트 용이)

`initialize_agent` · `AgentType.ZERO_SHOT_REACT_DESCRIPTION` · `AgentExecutor` 는 모두 **deprecated**. 새 코드는 `create_agent` 로 통일한다 — 옛 예제는 `create_agent` 기준으로 치환해 읽는다.

> 🎯 ReAct **개념**(Thought·Action·Observation)은 그대로 유효하다. 변경된 것은 그 루프의 **실행 주체** — 옛 API 대신 `create_agent` 가 담당한다.

ReAct 가 실패하는 5가지 패턴

패턴	증상	해결
무한루프	같은 도구 반복 호출	`max_iterations` + early stopping
도구 선택 오류	잘못된 도구 호출	도구 description 다듬기 + few-shot 예시
파라미터 누락	`Action Input` 빈 칸	structured output / function calling
관찰 무시	Observation 보고도 같은 도구 또 호출	Memory 강화, 프롬프트 수정
포기 / hallucination	정보 부족인데 임의 답변	`Final Answer` 전 검증 단계 추가

더 발전된 패턴들 (Day 4 이후)

패턴	핵심 차이
ReAct (2022)	Thought-Action-Observation 직선 반복
Plan-and-Execute (2023)	먼저 N단계 계획 → 단계마다 ReAct
Self-Ask	LLM 이 자기에게 sub-question 던지기
ReWOO	사고와 도구 호출을 분리 (LLM 호출 ↓)
LLMCompiler	도구를 DAG 로 병렬 호출
LangGraph 기반	상태 머신 — 완전 커스텀 (Session 27)

> 🎯 본 코스에서는 **ReAct 기본기** 학습 → **LangGraph** 로 자유 형태 (Session 27).

PART C

PART C
Tool Calling — Function Calling 의 진화

모델의 함수 호출 메커니즘 — 약 25분

2023년 6월 — LLM 함수 호출의 표준화

Before (Function Calling 이전)

- LLM 응답은 항상 텍스트
- 텍스트를 직접 파싱해 함수 인자 추출 — 불안정 (정규식, json 파싱 실패)
- ReAct 의 `Action: ...` 형식도 그래서 종종 깨짐

After (Function Calling)

- 모델이 학습 때부터 **JSON 함수 호출 형식** 으로 응답하도록 fine-tuned
- 응답 구조에 `tool_calls` 필드 — 안정적인 구조화 출력
- 잘못된 함수명·파라미터 호출 거의 사라짐

OpenAI API — 직접 사용

```
from openai import OpenAI

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "지정 도시의 현재 날씨 조회",
        "parameters": {
            "type": "object",
            "properties": {
                "city": {"type": "string", "description": "도시명 한국어"},
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["city"],
        },
    },
}]

response = OpenAI().chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "서울 날씨 어때?"}],
    tools=tools,
    tool_choice="auto",         # auto / required / {"type":"function", "function":{"name":...}}
)
print(response.choices[0].message.tool_calls)
# [ChatCompletionMessageToolCall(id='call_abc', function=Function(name='get_weather',
#   arguments='{"city": "서울", "unit": "celsius"}'), type='function')]
```

4단계 중 1~2 — Define · Call

```
from openai import OpenAI
import json

client = OpenAI()

# ═══ 1. 도구 정의 ===
def get_weather(city: str, unit: str = "celsius") → dict:
    # 실제로는 API 호출. 여기서는 mock.
    return {"city": city, "temperature": 20, "condition": "맑음", "unit": unit}
```

⬇️ 코드가 길어 다음 장에 이어집니다.

4단계 중 1~2 — Define · Call (이어서)

```
tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "지정 도시의 현재 날씨",
        "parameters": {
            "type": "object",
            "properties": {
                "city": {"type": "string"},
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["city"],
        },
    },
}]
available = {"get_weather": get_weather}

# ═══ 2. 1차 호출 — 모델이 도구 호출 결정 ===
messages = [{"role": "user", "content": "서울과 도쿄 날씨 비교해줘"}]
resp = client.chat.completions.create(
    model="gpt-4o-mini", messages=messages, tools=tools, tool_choice="auto"
)
msg = resp.choices[0].message
messages.append(msg.model_dump())   # 모델 응답을 history 에
```

💡 `tool_choice` : `"auto"` (LLM 결정) / `"required"` (반드시 호출) / `{"type":"function","function":{"name":"X"}}` (특정 도구 강제)

4단계 중 3~4 — Execute · Reply

```
# === 3. 도구 실행 ===
if msg.tool_calls:
    for call in msg.tool_calls:
        fn_name = call.function.name
        fn_args = json.loads(call.function.arguments)
        result = available[fn_name](**fn_args)
        messages.append({
            "role": "tool",
            "tool_call_id": call.id,
            "name": fn_name,
            "content": json.dumps(result, ensure_ascii=False),
        })

# ═══ 4. 2차 호출 — 결과로 답변 생성 ===
final = client.chat.completions.create(
    model="gpt-4o-mini", messages=messages, tools=tools
)
print(final.choices[0].message.content)
# → "서울은 맑고 20°C, 도쿄는 흐리고 18°C 입니다."
```

핵심 — 모델은 직접 함수를 실행하지 않는다

- 모델은 "**어떤 함수를 어떤 인자로 부를지**" 만 JSON 으로 알려줌 (2단계)
- 실제 실행은 **우리 코드** 가 담당 (3단계) — 그래서 보안·권한 통제 가능
- 실행 결과를 `role: "tool"` 로 다시 넣어줘야 모델이 최종 답을 만든다 (4단계)

앞의 50줄을 LangChain 으로 압축

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

# ═══ 1. 도구 — @tool 데코레이터 ===
@tool
def get_weather(city: str, unit: str = "celsius") → dict:
    """지정 도시의 현재 날씨 조회.

    Args:
        city: 도시명 (한국어)
        unit: 온도 단위 (celsius | fahrenheit)
    """
    return {"city": city, "temperature": 20, "condition": "맑음", "unit": unit}

# Docstring + 타입힌트 → 자동으로 OpenAI tool schema 생성
print(get_weather.args_schema.model_json_schema())

# ═══ 2. Pydantic 으로 더 엄격하게 ===
class WeatherInput(BaseModel):
    city: str = Field(description="도시명 한국어")
    unit: str = Field(default="celsius", description="celsius 또는 fahrenheit")

@tool("get_weather", args_schema=WeatherInput, return_direct=False)
def get_weather_v2(city: str, unit: str = "celsius") → str:
```

💡 `@tool` 만 붙이면 앞의 OpenAI 50줄짜리 tool schema 가 자동 생성된다 — Agent 구성은 다음 슬라이드에서 다룬다.

`create_agent` 한 줄로 Agent 구성

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent

# ═══ 3. Agent 구성 — create_agent (LangChain 1.x) ═══
llm = ChatOpenAI(model="gpt-4o-mini")
agent = create_agent(llm, [get_weather, search_web, calculator])

# ═══ 4. 사용 ===
result = agent.invoke({"messages": [{"role": "user", "content": "서울 날씨?"}]})
print(result["messages"][-1].content)
```

`@tool` 이 자동 추출하는 것

함수 요소	→ 변환 결과
함수명	tool name
docstring	description
타입 힌트	parameter schema
Optional / default	required 여부

> 🎯 **LangChain 의 가치**: 50줄 OpenAI 직접 호출이 5줄로. 그러나 **내부는 동일한 Function Calling**.

N개 도구 중 하나를 선택하는 메커니즘

LLM 관점의 처리 순서

- 모든 도구의 **description** 이 system prompt 또는 별도 tools 파라미터에 주입된다
- 사용자 질의가 입력된다
- 모델이 학습된 분포에 따라 "질의에 가장 부합하는 도구" 를 출력한다

Description 이 중요한 이유

```
# ❌ 나쁜 description
@tool
def f(x: str) → str:
    """함수입니다."""
    ...

# ✅ 좋은 description
@tool
def search_internal_wiki(query: str) → str:
    """**회사 내부 위키** 에서 키워드로 검색.
    제품 매뉴얼·API 문서·정책 가이드 가 들어있음.
    외부 정보 (Wikipedia, 뉴스) 는 다른 도구 사용."""
    ...
```

도구가 많아질 때 — 선택 전략

도구 5개 이상이면

- **description 명확** — "언제 이걸 쓰면 안 되는가" 도 포함
- **카테고리화** — 검색·계산·작성·실행 등 prefix
- **Few-shot** — system prompt 에 (질의 → 선택된 도구) 예시 2~3개
- **Routing layer** — Day 4 (Routing 패턴) 에서 다룰 LLM 라우터

도구가 매우 많을 때 (>20)

- 전체 도구 목록을 모두 노출하면 context 낭비 + 정확도 저하
- **RAG 기반 도구 선택**: 도구 description 을 벡터DB 에 색인 → 질의로 top-5 도구만 LLM 에 노출

PART D

PART D
미니 실습 — 직접 도구 만들기

약 25분

실습 과제 (25분) — 개요

목표

자기 분야에 맞는 **커스텀 도구 3개** 를 만들어 Agent 에 연결.

예시 도메인별 도구

도메인	도구 후보
웹 개발자	`check_url`, `lookup_domain`, `validate_json`
데이터 분석	`query_postgres`, `summarize_csv`, `plot_chart`
학생	`search_arxiv`, `wiki_lookup`, `convert_units`
마케팅	`tweet_search`, `sentiment_analyze`, `competitor_check`

진행 단계 (총 25분)

1. **도구 정의** (10분) — `@tool` 로 도구 3개 작성
2. **Agent 구성** (5분) — `create_agent` 한 줄
3. **질문 3가지** (10분) — 각 질의가 어떤 도구를 부르는지 trace 관찰

> 🎯 자기 도메인에서 실제로 쓸 법한 도구를 떠올려 본다 — 다음 슬라이드에 1단계 코드 예시.

1단계 — 도구 정의 (10분)

```
from langchain_core.tools import tool
import requests, json

@tool
def check_url_status(url: str) → dict:
    """URL 의 HTTP 상태 코드와 응답 시간을 확인."""
    import time, requests
    t = time.time()
    try:
        r = requests.head(url, timeout=5, allow_redirects=True)
        return {"url": url, "status": r.status_code,
                "elapsed_ms": int((time.time()-t)*1000)}
    except Exception as e:
        return {"url": url, "error": str(e)}

@tool
def get_kospi_close(date: str) → dict:
    """주어진 날짜의 KOSPI 종가 (YYYY-MM-DD)."""
    # ... 평소 쓰는 데이터 소스 결합
    ...

@tool
def my_rag_search(question: str) → str:
    """색인해 둔 내 문서에서 관련 내용을 검색."""
```

💡 도구마다 docstring 한 줄이 곧 LLM 의 선택 기준 — 명확하게 쓸수록 도구 선택이 정확해진다.

2~3단계 — Agent 구성 · 질문 · 보고

2단계 — Agent 구성 (5분)

```
from langchain.agents import create_agent
agent = create_agent(llm, [check_url_status, get_kospi_close, my_rag_search])
```

3단계 — 도구를 부르는 3가지 질문 (10분)

- "https://example.com 살아있어?"
- "2024-12-30 KOSPI 종가는?"
- "내 문서에서 RAG 평가 메트릭 5개 찾아줘"

각 질의가 어떤 도구를 호출하는지 trace 관찰.

결과 보고

- LLM 이 도구를 잘못 선택한 사례 1건 → description 수정으로 해결
- Day 2 RAG 와 Agent 결합으로 확보한 효용 정리

핵심 7가지

1. ✅ Agent = **LLM + Tool Loop + Decision**
2. ✅ Agent 5 컴포넌트 — LLM · Tools · Prompt · Memory · Executor
3. ✅ **ReAct 패턴** — Thought · Action · Observation 반복
4. ✅ **Function Calling** — JSON 구조화 응답으로 안정성 ↑
5. ✅ LangChain `@tool` 데코레이터 — docstring + 타입힌트 자동 추출
6. ✅ 실습은 처음부터 `create_agent` (1.x) — `AgentExecutor` (0.x) 는 옛 예제 참고용
7. ✅ ReAct 실패 5가지 패턴 + 해결책

이번 과정의 목표 — RAG 는 "항상 검색하는" 고정 파이프라인, 에이전트는 "무엇을 할지 LLM 이 판단하는" 동적 루프이다. 다음 회차부터 **앞서 만든 RAG 검색을 "도구" 로 전환해**, 검색 여부를 스스로 판단하는 에이전트로 확장한다.

실습 과제 — 직접 해보기

필수

- [ ] 도구 3개 만들어 Agent 한 사이클 동작 확인
- [ ] ReAct trace 1건 캡처 — verbose 로그 또는 LangSmith 화면

옵션

- [ ] **OpenAI Function Calling** 직접 호출 (SDK 만, LangChain 없이)
- [ ] 인터넷의 옛 `initialize_agent` 예제 하나를 찾아 `create_agent` 로 옮겨 보기
- [ ] RAG retriever 를 **도구** 로 변환 → Agentic RAG 패턴 직접

📤 제출

- 도구 3개 코드 + trace 1건 — gist/Slack