CrewAI 완전 정복: 8시간 마스터 코스

생성일: 2025년 11월 11일

CrewAI 완전 정복

CrewAI 마스터 코스

강의 개요

Chapter 1. CrewAI 프레임워크 소개

학습 목표

- CrewAI의 핵심 개념을 이해한다
- 환경 설정과 설치를 완료한다
- 첫 번째 에이전트를 만들어본다

- CrewAI의 차별화된 특징을 파악한다

1.1. CrewAI란?

AI 에이전트 '팀'을 만드는 가장 쉬운 방법

CrewAI는 다중 AI 에이전트 시스템을 구축하기 위한 혁신적인 파이썬 프레임워크입니다. 각기 다른 전문성을 가진 AI 에이전트들이 '크루(Crew)'를 이루어 복잡한 임무를 협력하여 수행하도록 돕습니다.

CrewAI의 핵심 철학: 분업과 협업

하나의 만능 AI 대신, **전문화된 여러 에이전트가 팀을 이루어 협력**하는 것이 더 높은 품질의 결과를 만든다는 철학에 기반합니다. 이는 인간 사회의 전문가 팀과 같습니다.

- **분업의 힘:** 복잡한 작업을 작은 전문 영역으로 분할
- **협업의 시너지:** 각 에이전트의 전문성을 결합하여 더 나은 결과 창출

> CrewAI는
> **2024년부터 완전히 독립적인 프레임워크**
> 로 재구축되었습니다. 더 이상 LangChain에 의존하지 않으며, 처음부터 새롭게 설계된 아키텍처를 사용합니다.
>
> - ✅
>   **독립성:**
>   LangChain 없이도 완전히 작동
> - ✅
>   **호환성:**
>   LangChain 컴포넌트(LLMs, Tools)와 여전히 호환 가능
> - ✅
>   **성능:**
>   독자적인 최적화로 더 빠르고 안정적

1.2. CrewAI vs 다른 프레임워크

주요 프레임워크 비교 매트릭스

LangChain	포괄적 생태계, 풍부한 도구	범용 개발, 프로토타이핑	중간-높음	보통
LangGraph	순환 워크플로우, 상태 관리	복잡한 그래프 기반 작업	높음	높음
LlamaIndex	데이터 중심, RAG 특화	RAG 애플리케이션, 문서 처리	낮음-중간	높음
CrewAI	멀티 에이전트 팀, 직관적 API	협업 시스템, 복잡한 워크플로우	중간	높음
n8n	시각적 워크플로우, 노코드	비즈니스 자동화, 통합	낮음	보통
Agno	고성능 런타임, 통합 UI, FastAPI 기반	프로덕션 규모 시스템	낮음	매우 높음
AutoGen	대화형 에이전트, 유연한 상호작용	멀티 에이전트 대화, 연구/실험	중간-높음	높음

CrewAI의 차별화된 특징

멀티 에이전트 협업	✅ 핵심 기능	⚠️ 복잡한 구현 필요	❌ 제한적	✅ 대화 기반 협업
역할 기반 설계	✅ Role, Goal, Backstory	⚠️ 수동 구현	⚠️ 제한적	✅ 유연한 역할 설정
워크플로우 관리	✅ Sequential, Hierarchical	✅ LangGraph 필요	⚠️ 기본적	✅ 대화 기반 흐름
학습 곡선	✅ 중간	⚠️ 중간-높음	✅ 낮음-중간	⚠️ 중간-높음
RAG 지원	✅ 통합 지원	✅ 풍부한 기능	✅ 핵심 기능	⚠️ 기본적
커스텀 도구	✅ @tool 데코레이터	✅ 풍부한 도구	⚠️ 제한적	✅ 함수 호출 지원
프로덕션 준비	✅ 내장 모니터링	⚠️ 추가 설정 필요	⚠️ 기본적	⚠️ 기본적

CrewAI의 핵심 장점

- **직관적인 API:** Agent, Task, Crew, Process의 명확한 구조
- **자동화된 협업:** 에이전트 간 정보 전달과 조율 자동화
- **유연한 프로세스:** Sequential과 Hierarchical 프로세스 지원
- **풍부한 도구 생태계:** @tool 데코레이터로 쉬운 도구 개발
- **프로덕션 준비:** 모니터링, 로깅, 에러 핸들링 내장

프레임워크 선택 가이드

- **초보자/빠른 시작:** CrewAI
  → 직관적인 API와 명확한 구조로 빠르게 학습 가능. 멀티 에이전트 협업이 핵심 기능으로 제공됨
- **RAG 애플리케이션:** LlamaIndex 우선, CrewAI로 확장
  → LlamaIndex로 RAG 구축 후, 복잡한 에이전트 협업이 필요하면 CrewAI와 통합

- **복잡한 상태 관리/순환 워크플로우:** LangGraph 우선, CrewAI 고려
  → 그래프 기반 복잡한 상태 관리가 필요하면 LangGraph, 계층적 팀 구조가 필요하면 CrewAI
- **비즈니스 자동화/통합:** n8n으로 시작, 고급 AI 로직은 CrewAI
  → 노코드 워크플로우로 빠른 통합, AI 에이전트의 고급 의사결정이 필요하면 CrewAI 활용
- **고성능 프로덕션 시스템:** Agno (속도 중시) 또는 CrewAI (유지보수성 중시)
  → 최고 성능과 Private 배포가 필요하면 Agno, 개발 생산성과 확장성이 우선이면 CrewAI
- **연구/실험적 프로젝트:** AutoGen
  → 에이전트 간 대화 중심 상호작용과 유연한 실험이 필요한 경우

1.3. CrewAI 핵심 개념

CrewAI는 **Agent, Task, Crew, Process** 4가지 핵심 요소로 구성됩니다.

1. Agent (에이전트)

특정 역할(Role), 목표(Goal), 배경(Backstory)을 가진 행위자

```
Agent(
    role='연구 전문가',
    goal='주어진 주제에 대한 포괄적인 정보를 수집한다',
    backstory='당신은 정보과학 박사 학위를 가진 전문 연구자입니다.',
    llm=llm
)
```

에이전트 속성 상세 설명

role	에이전트의 직책/역할	"Senior Data Scientist", "Content Writer"
goal	달성하고자 하는 목표	"Create engaging blog posts", "Analyze market trends"
backstory	에이전트의 배경 및 전문성	경력, 전문 분야, 성격 등
verbose	작업 과정 상세 출력 여부	True/False
allow_delegation	다른 에이전트에게 작업 위임 허용 (기본값: False)	True/False
tools	사용 가능한 도구 목록	[SerperDevTool(), WebsiteSearchTool()]

2. Task (작업)

에이전트가 수행해야 할 구체적인 작업 단위 (Description, Expected Output)

```
Task(
    description='{topic}에 대한 포괄적인 연구를 수행하세요.',
    expected_output='핵심 사실, 통계, 트렌드를 포함한 상세한 연구 보고서.',
    agent=researcher
)
```

작업 속성 상세

description	작업에 대한 상세한 설명	매우 높음
expected_output	기대되는 결과물의 형태	높음
agent	작업을 수행할 에이전트	매우 높음
tools	작업에 필요한 도구들	중간
async_execution	비동기 실행 여부	낮음

3. Crew (크루)

에이전트와 임무를 모아 구성한 협업 팀

```
Crew(
    agents=[researcher, writer, editor],
    tasks=[research_task, writing_task, editing_task],
    process=Process.sequential
)
```

4. Process (프로세스)

팀이 임무를 처리하는 방식

- **Sequential (순차적):** 작업을 하나씩 순서대로 실행
- **Hierarchical (계층적):** 매니저 에이전트가 다른 에이전트들을 관리

1.4. 부록: CrewAI Enterprise

정의 및 개요

CrewAI Enterprise (CrewAI AMP - Agent Management Platform)는 오픈소스 CrewAI 프레임워크를 기반으로 구축된 **엔터프라이즈급 AI 에이전트 관리 플랫폼**입니다. 프로덕션 환경에서 AI 에이전트 워크플로우를 **배포(deploy), 모니터링(monitor), 확장(scale)**하는 데 필요한 모든 기능을 제공합니다.

시작하기

주요 기능 및 가치 제안

1️⃣ 보안 및 컴플라이언스

- **인증 및 규정:** SOC2, HIPAA, FedRAMP High 인증
- **접근 제어:** RBAC(역할 기반 접근 제어), 감사 로그, 암호화된 데이터 저장
- **엔터프라이즈 통합:** Auth0, Microsoft Entra ID 네이티브 지원

2️⃣ 배포 및 개발 도구

- **배포 방식:** GitHub 통합, Crew Studio(no-code), CLI 배포 지원
- **Crew Studio:** 노코드/로우코드 인터페이스로 에이전트 생성 및 관리
- **도구 저장소:** 크루 기능 확장을 위한 도구 라이브러리
- **API 통합:** REST API 엔드포인트 및 웹훅 스트리밍

3️⃣ 관찰성 및 모니터링

- **실시간 모니터링:** 에이전트 실행을 실시간으로 추적 및 디버깅
- **상세 트레이스:** 실행 트레이스, 비용/지연 분석, 성능 메트릭
- **실패 관리:** 재시도 정책, 실시간 테스팅, 알림
- **포괄적 가시성:** 크루 성능에 대한 상세 로깅 및 추적

4️⃣ 거버넌스 및 제어

- **정책 기반 제어:** 도구 사용 허가, 승인 워크플로우
- **프라이빗 도구 저장소:** 조직 전용 도구 관리
- **통합 제어 플레인:** 모든 에이전트와 워크플로우 중앙 관리

5️⃣ 확장성 및 성능

- **분산 실행:** 수평적 확장, 큐/스케줄링, 멀티 테넌시
- **성능 최적화:** 캐시/결과 재사용, 모델 라우팅, 쿼터/레이트리밋
- **관리 인프라:** 최소한의 마찰로 관리 인프라에 크루 배포

6️⃣ 프라이빗 배포 옵션

- **자체 인프라 배포:** 고객 데이터센터/VPC에 직접 배포 가능
- **환경 관리:** Dev/Stage/Prod 분리, 버전 관리, 롤백
- **데이터 프라이버시:** 완전한 데이터 주권 보장

7️⃣ 엔터프라이즈 통합

- **벡터DB:** Qdrant, Amazon Bedrock KB, MySQL, PostgreSQL, Weaviate
- **MCP 지원:** 양방향 MCP로 데스크톱/웹 앱/원격 에이전트 통합
- **이벤트 시스템:** 이벤트 버스, 웹훅 (인증 토큰 지원)
- **GitHub 연결:** GitHub 리포지토리 직접 연결하여 코드 배포

8️⃣ AI 고급 기능

- **신뢰성:** Hallucination detection 가드레일
- **최적화:** Query rewriting (검색 최적화)
- **파트너십:** NVIDIA NIM 마이크로서비스, NeMo Retriever/Guardrails

9️⃣ 전담 지원

- **24/7 기술 지원:** 전담 엔터프라이즈 지원팀
- **SLA 보장:** 가용성 및 응답 시간 보장

CrewAI OSS vs Enterprise 비교

대상 사용자	개발자, 소규모 팀, 스타트업	조직, 대규모 팀, 기업
사용 목적	프로토타입, PoC, 실험	프로덕션, 미션 크리티컬 시스템
라이선스	무료 (MIT/Apache)	상용 라이선스 (유료)
포커스	개발 생산성, 멀티 에이전트 API	운영/보안/스케일, 거버넌스
관찰성	기본 로깅	CrewAI AMP: 대시보드, 트레이스, 비용/지연 분석, 실시간 테스팅
보안	환경변수 기반 설정 권장	RBAC, Auth0/MS Entra ID, 비밀금고, 감사 로그, SOC2/HIPAA/FedRAMP
개발 도구	코드 기반 개발	Crew Studio (no-code/low-code), 도구 저장소, 고급 CLI
배포	앱/스크립트 수준 배포	GitHub 통합, Crew Studio, CLI / 관리 인프라 배포, 환경 분리, 버전 관리, 롤백
스케일	단일 프로세스/간단한 병렬	분산 실행, 큐/스케줄링, 멀티 테넌시
AI 기능	기본 LLM 연동	Hallucination detection, Query rewriting, NVIDIA NIM 통합
통합	커뮤니티 도구	양방향 MCP, 프라이빗 도구 저장소, 이벤트 버스/웹훅
지원	커뮤니티 지원 (Discord, GitHub)	전담 기술 지원 (24/7), SLA 보장

언제 Enterprise가 필요한가?

- **팀 협업이 중요할 때:** 조직 내 여러 팀이 에이전트를 공유/운영해야 할 때
- **보안·컴플라이언스가 필수일 때:** SOC2, HIPAA, FedRAMP 등의 규정 준수가 필요한 경우
- **프라이빗 배포가 필요할 때:** 데이터가 외부로 나가면 안 되는 경우 (자체 데이터센터/VPC에 배포)
- **대규모 실행이 필요할 때:** 수백~수천 개의 에이전트를 동시에 실행하고 관리해야 할 때
- **정확성이 중요할 때:** Hallucination detection 등 AI 신뢰성 보장이 필요한 프로덕션 환경
- **엔터프라이즈 통합이 필요할 때:** Auth0/Microsoft Entra ID, 기업용 벡터DB, 웹훅/이벤트 버스 연동이 필요한 경우
- **노코드 개발이 필요할 때:** Crew Studio를 통해 코딩 없이 에이전트 워크플로우를 구성하고 배포해야 할 때

Chapter 2. CrewAI로 AI Agent 만들기

학습 목표

- CrewAI의 기본 구성 요소(Agent, Task, Crew)를 이해한다.
- 환경 설정부터 에이전트 실행까지 전체 과정을 실습한다.

- 단일 에이전트와 다중 에이전트 크루를 구성하고 실행한다.
- 에이전트 간의 협업 방식(순차 프로세스)을 이해한다.

2.1. 설치 및 기본 설정

1. 가상환경 설정 (권장)

Conda를 사용하여 Python 3.11 기반의 가상환경을 생성하고 활성화합니다.

```
# Conda 가상환경 생성
conda create -n crewai-ssam python=3.11

# 가상환경 활성화
conda activate crewai-ssam
```

2. 라이브러리 설치

실습에 필요한 라이브러리들을 설치합니다. 프로젝트의 `requirements.txt` 파일을 사용하여 한 번에 설치할 수 있습니다.

```
# 전체 패키지 한 번에 설치 (권장)
pip install -r requirements.txt

# 또는 개별 설치
# 핵심 AI 프레임워크
pip install crewai>=0.177.0 crewai-tools>=0.0.1

# LangChain 생태계
pip install langchain>=0.1.0 langchain-community>=0.0.20 langchain-openai>=0.3.32

# LLM API 클라이언트
pip install openai>=1.106.1

# MCP (Model Context Protocol) - 2024년 신규 표준
pip install mcp>=0.9.0 mcpadapt anthropic>=0.18.0 httpx>=0.25.0

# 벡터 데이터베이스 및 임베딩
pip install chromadb>=1.0.20 sentence-transformers>=5.1.0

# 검색 및 웹 스크래핑 도구
pip install duckduckgo-search>=3.9.0 beautifulsoup4>=4.12.3

# 웹 프레임워크
pip install streamlit>=1.30.0

# 데이터 처리
pip install pandas>=2.2.3 numpy>=2.1.3

# 유틸리티 및 환경 관리
pip install requests>=2.32.3 python-dotenv>=1.1.0 pydantic>=2.10.3

# Jupyter 노트북 (선택)
pip install jupyter>=1.1.1 ipykernel>=6.29.5
```

3. 설치 확인

```
# 의존성 충돌 확인
pip check

# 주요 패키지 버전 확인
pip list | grep -E "(crewai|langchain|openai|streamlit)"
```

2.1.1. 환경 변수 설정

API 키 설정

프로젝트 루트에 `.env` 파일을 만들고 필요한 API 키를 입력합니다. 이 키는 에이전트가 LLM을 사용하는 데 필수적입니다.

```
# .env 파일 생성
touch .env

# .env 파일 내용 작성
# 필수 API 키
OPENAI_API_KEY=your_openai_key_here

# 선택적 API 키 (기능 확장용)
# SERPER_API_KEY=your_serper_key_here

# 기본 설정
DEFAULT_LLM_MODEL=gpt-3.5-turbo
DEFAULT_TEMPERATURE=0.7
EMBEDDING_MODEL=sentence-transformers/all-MiniLM-L6-v2
```

환경 변수 확인

```
# 환경 변수 로드 확인
python -c "import os; from dotenv import load_dotenv; load_dotenv(); print('OpenAI
```

2.2. 예제 파일 개요: 01_basics

이 챕터에서는 `01_basics` 폴더의 두 가지 핵심 예제를 다룹니다.

first_agent.py	단일 에이전트 생성과 실행	Agent, Task, Crew 기본 개념, kickoff 실행
simple_crew.py	다중 에이전트 협업 파이프라인	Process.sequential, 에이전트 간 자동 컨텍스트 전달

이 두 파일을 통해 CrewAI의 가장 기본적인 사용법부터 여러 에이전트가 협력하는 워크플로우까지 단계적으로 학습합니다.

2.3. 첫 번째 에이전트 만들기: first_agent.py

실습 목표: 창작 작가 에이전트

주어진 주제에 대해 창의적인 단편 소설을 작성하는 단일 AI 에이전트를 만들어봅니다. 이 과정을 통해 CrewAI의 3대 구성요소인 **Agent, Task, Crew**를 이해합니다.

1. 환경 설정 및 LLM 초기화

먼저 필요한 라이브러리를 import하고, 환경 변수를 로드한 다음 LLM을 설정합니다.

```
# 01_basics/first_agent.py

import warnings
import os
from crewai import Agent, Task, Crew
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

# Pydantic DeprecationWarning 무시
warnings.filterwarnings("ignore", category=DeprecationWarning, module="pydantic")

# 환경 변수 로드 (.env 파일에서 API 키 등을 불러옴)
load_dotenv()

# 기본 설정 상수
DEFAULT_MODEL = "gpt-3.5-turbo"
DEFAULT_TEMPERATURE = 0.7
DEFAULT_WORD_COUNT = 300
```

2. LLM 설정 함수

API 키를 확인하고 언어 모델을 설정하는 함수입니다. 에러 처리를 포함합니다.

```
def check_api_key():
    """OpenAI API 키가 올바르게 설정되었는지 확인합니다."""
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key or api_key == "your_openai_api_key_here":
        print("❌ OpenAI API 키가 설정되지 않았습니다!")
        print("💡 .env 파일에서 OPENAI_API_KEY를 실제 API 키로 설정해주세요.")
        return False
    return True

def setup_llm(model_name=None, temperature=None, verbose=True):
    """언어 모델(LLM)을 설정하고 초기화합니다."""
    if not check_api_key():
        return None

    model_name = model_name or os.getenv("DEFAULT_LLM_MODEL", DEFAULT_MODEL)
    temperature = temperature or float(os.getenv("DEFAULT_TEMPERATURE", str(DEFAUL

    try:
        llm = ChatOpenAI(
            model=model_name,  # 2024 권장: model 파라미터 사용
            temperature=temperature,
            api_key=os.getenv("OPENAI_API_KEY")
        )
        if verbose:
            print(f"✅ LLM 설정 완료: {model_name} (temperature: {temperature})")
        return llm
    except Exception as e:
        print(f"❌ LLM 설정 실패: {e}")
        return None
```

3. 에이전트 생성 함수

창작 작가 역할을 수행하는 에이전트를 생성합니다.

```
def create_writer_agent(llm):
    """창작 작가 역할을 수행하는 에이전트를 생성합니다."""
    return Agent(
        role='창작 작가',
        goal='매력적이고 창의적인 단편 소설을 작성한다',
        backstory="""당신은 10년 이상의 경험을 가진 재능 있는 창작 작가입니다.
        독자들을 사로잡는 매력적인 캐릭터와 예상치 못한 플롯 트위스트를 만드는
        독특한 능력을 가지고 있습니다. 당신의 글쓰기 스타일은 생생하고 감정적이며
        생각을 자극합니다. 모든 응답은 한국어로 작성해주세요.""",
        llm=llm,
        verbose=True
    )
```

2.3.1. 작업(Task) 정의

4. 작업(Task) 정의 함수

에이전트에게 부여할 구체적인 임무를 **Task**로 정의합니다. 무엇을 해야 하고(description), 어떤 결과물을 만들어야 하는지(expected_output) 명확하게 지시하는 것이 중요합니다.

```
def create_writing_task(agent, word_count=DEFAULT_WORD_COUNT):
    """단편 소설 작성 작업을 생성합니다."""
    return Task(
        description=f"""주제 '{{topic}}'에 대한 창의적인 단편 소설({word_count}단어)을 작성

        요구사항:
        - 깊이 있는 매력적인 캐릭터를 만들어주세요
        - 갈등과 해결이 있는 매력적인 플롯을 포함해주세요
        - 끝에 예상치 못한 반전을 추가해주세요
        - 생생한 묘사와 감정적 깊이를 사용해주세요
        - 일반 독자들이 읽기에 적합하게 만들어주세요
        - 모든 내용을 한국어로 작성해주세요""",

        expected_output=f"""완성된 단편 소설로 다음을 포함해야 합니다:
        - 독자를 사로잡는 매력적인 시작
        - 명확한 동기를 가진 잘 발달된 캐릭터들
        - 갈등과 해결이 있는 매력적인 플롯
        - 예상치 못한 반전 결말
        - 생생한 묘사와 감정적 깊이
        - 총 {word_count}단어
        - 적절한 이야기 구조 (시작, 중간, 끝)
        - 한국어로 작성된 완성된 소설""",

        agent=agent
    )
```

2.3.2. 크루(Crew) 구성 및 실행

5. 크루(Crew) 생성 함수

에이전트와 작업을 한 팀으로 묶어 **Crew**를 구성합니다.

```
def create_story_crew(agent, task, verbose=True):
    """스토리 생성을 위한 크루(Crew)를 생성합니다."""
    return Crew(
        agents=[agent],
        tasks=[task],
        verbose=verbose
    )
```

6. 전체 프로세스 실행 함수

위에서 만든 모든 함수를 조합하여 스토리 생성 전체 과정을 실행합니다.

```
def run_story_generation(model=None, temperature=None,
                         topic="a robot learning to paint",
                         word_count=DEFAULT_WORD_COUNT,
                         verbose=True):
    """스토리 생성 전체 프로세스를 실행합니다."""
    if verbose:
        print("🚀 CrewAI 첫 번째 에이전트 시작")
        print("=" * 50)

    # 1. LLM 설정
    llm = setup_llm(model, temperature, verbose)
    if not llm:
        return None

    # 2. 에이전트 생성
    if verbose:
        print("\n📝 에이전트 생성 중...")
    writer = create_writer_agent(llm)

    # 3. 작업 생성
    if verbose:
        print("\n📋 작업 생성 중...")
    writing_task = create_writing_task(writer, word_count)

    # 4. 크루 생성
    if verbose:
        print("\n👥 크루 생성 중...")
    story_crew = create_story_crew(writer, writing_task, verbose)

    # 5. 크루 실행
    if verbose:
        print(f"\n🎬 스토리 생성 시작...")
        print(f"📖 주제: {topic}")

    try:
        result = story_crew.kickoff(inputs={'topic': topic})

        if verbose:
            print(f"\n{'='*60}")
            print("📖 생성된 소설:")
            print('='*60)
            print(result)
            print(f"\n{'='*60}")
            print("✅ 스토리 생성 완료!")
```

```
        return str(result)

    except Exception as e:
        print(f"❌ 실행 중 오류 발생: {e}")
        return None

# 메인 실행
if __name__ == "__main__":
    run_story_generation()
```

2.4. 코드 분석: first_agent.py

코드 구조와 실행 흐름

`first_agent.py` 는 단순히 코드를 나열하는 것을 넘어, 각 기능을 함수로 분리하여 재사용성과 가독성을 높인 좋은 예시입니다. 이 구조를 이해하면 더 복잡한 Crew를 만들 때 큰 도움이 됩니다.

1. 기능별 함수 분리

- `setup_llm()` : 언어 모델(LLM)을 설정하고 생성합니다.
- `create_writer_agent()` : 작가 에이전트를 생성합니다.
- `create_writing_task()` : 글쓰기 작업을 생성합니다.
- `create_story_crew()` : 에이전트와 작업을 묶어 크루를 생성합니다.

각 함수는 명확한 단일 책임을 가지므로, 코드를 이해하고 재사용하기 쉽습니다.

2. 전체 프로세스 실행

`run_story_generation()` 함수가 위에서 만든 함수들을 순서대로 호출하여 전체 스토리 생성 과정을 조율합니다. 마지막으로 `if __name__ == "__main__":` 블록 안에서 이 함수를 실행하여, 스크립트의 시작점을 명확히 합니다.

2.5. 다중 에이전트 협업: simple_crew.py

실습 목표: 콘텐츠 제작팀 구성

이번에는 **연구원, 작가, 편집자** 3명의 전문 에이전트로 구성된 팀을 만들어, 하나의 주제에 대한 고품질 기사를 작성하는 워크플로우를 자동화합니다.

연구 → 작성 → 편집으로 이어지는 순차적 파이프라인

이 실습을 통해 여러 에이전트가 어떻게 협력하고, 한 에이전트의 결과물이 다음 에이전트에게 어떻게 전달되는지 학습합니다.

1. 환경 설정 및 초기화

필요한 라이브러리를 import하고 환경을 설정합니다.

```
# 01_basics/simple_crew.py

import warnings
import os
from crewai import Agent, Task, Crew, Process
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

warnings.filterwarnings("ignore", category=DeprecationWarning, module="pydantic")
load_dotenv()

def check_api_key():
    """API 키 설정 확인"""
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key or api_key == "your_openai_api_key_here":
        print("❌ OpenAI API 키가 설정되지 않았습니다!")
        return False
    return True

def setup_llm():
    """LLM 설정 및 초기화"""
    model_name = os.getenv("DEFAULT_LLM_MODEL", "gpt-3.5-turbo")
    temperature = float(os.getenv("DEFAULT_TEMPERATURE", "0.1"))

    llm = ChatOpenAI(
        model=model_name,
        temperature=temperature,
        api_key=os.getenv("OPENAI_API_KEY")
    )
    print(f"✅ LLM 설정 완료: {model_name} (temperature: {temperature})")
    return llm
```

2. 전문 에이전트 생성 함수

각기 다른 역할, 목표, 배경을 가진 세 명의 에이전트를 생성하는 함수들입니다.

```
def create_research_agent(llm):
    """연구 전문가 에이전트 생성"""
    return Agent(
        role='연구 전문가',
        goal='주어진 주제에 대한 포괄적인 정보를 수집한다',
        backstory="""당신은 정보과학 박사 학위를 가진 전문 연구자입니다.
        다양한 소스에서 정보를 수집, 분석, 종합하는 광범위한 경험을 가지고 있습니다.
        핵심 인사이트를 식별하고 명확하고 체계적인 방식으로 제시하는 데 탁월합니다.
        모든 응답은 한국어로 작성해주세요.""",
        llm=llm,
        verbose=True
    )

def create_writer_agent(llm):
    """콘텐츠 작가 에이전트 생성"""
    return Agent(
        role='콘텐츠 작가',
        goal='연구 결과를 바탕으로 매력적이고 유익한 콘텐츠를 작성한다',
        backstory="""당신은 8년 이상의 경험을 가진 전문 콘텐츠 작가입니다.
        매력적인 기사, 블로그 포스트, 보고서를 만드는 경험이 풍부합니다.
        복잡한 정보를 독자 친화적이고 유익하며 매력적인 콘텐츠로 변환하는
        재능을 가지고 있습니다. 모든 응답은 한국어로 작성해주세요.""",
        llm=llm,
        verbose=True
    )

def create_editor_agent(llm):
    """콘텐츠 편집자 에이전트 생성"""
    return Agent(
        role='콘텐츠 편집자',
        goal='콘텐츠의 품질과 명확성을 검토하고 개선한다',
        backstory="""당신은 저널리즘과 출판 분야의 배경을 가진 숙련된 편집자입니다.
        세부사항에 대한 예리한 눈을 가지고 있으며 콘텐츠 구조, 문법, 전반적인
        가독성을 개선하는 데 탁월합니다. 모든 콘텐츠가 높은 편집 기준을
        충족하도록 보장합니다. 모든 응답은 한국어로 작성해주세요.""",
        llm=llm,
        verbose=True
    )
```

2.5.1. 분업화된 작업 정의

3. 작업 생성 함수

각 에이전트의 전문성에 맞는 작업을 생성하는 함수들입니다. 작업은 [연구 → 작성 → 편집] 순서로 연결됩니다.

```
def create_research_task(agent):
    """연구 작업 생성"""
    return Task(
        description="""'{topic}'에 대한 포괄적인 연구를 수행하세요.

        연구 요구사항:
        - 핵심 사실과 통계를 수집하세요
        - 주요 트렌드와 발전사항을 식별하세요
        - 관련 예시와 사례 연구를 찾으세요
        - 중요한 날짜와 이정표를 기록하세요
        - 핵심 이해관계자와 전문가를 식별하세요
        - 모든 내용을 한국어로 작성하세요""",

        expected_output="""포괄적인 연구 보고서로 다음을 포함해야 합니다:
        - 핵심 발견사항의 요약
        - 상세한 사실과 통계
        - 현재 트렌드와 발전사항
        - 관련 예시와 사례 연구
        - 중요한 날짜와 이정표
        - 핵심 이해관계자와 전문가
        - 출처와 참고문헌
        - 한국어로 작성된 완성된 보고서""",

        agent=agent
    )

def create_writing_task(agent):
    """글쓰기 작업 생성"""
    return Task(
        description="""연구 결과를 바탕으로 유익한 기사를 작성하세요.

        작성 요구사항:
        - 제공된 연구 데이터를 활용하세요
        - 매력적인 도입부를 만드세요
        - 명확한 제목으로 콘텐츠를 구조화하세요
        - 관련 예시와 통계를 포함하세요
        - 전문적이면서도 접근하기 쉬운 톤으로 작성하세요
        - 정확성과 신뢰성을 보장하세요
        - 모든 내용을 한국어로 작성하세요""",

        expected_output="""잘 구조화된 기사로 다음을 포함해야 합니다:
        - 매력적인 도입부
        - 명확한 섹션 제목
        - 잘 연구된 콘텐츠
```

```
        - 관련 예시와 통계
        - 전문적인 톤
        - 적절한 결론
        - 800-1000단어
        - 한국어로 작성된 완성된 기사""",

        agent=agent
    )

def create_editing_task(agent):
    """편집 작업 생성"""
    return Task(
        description="""기사의 품질과 명확성을 검토하고 편집하세요.

        편집 요구사항:
        - 문법과 맞춤법을 확인하세요
        - 문장 구조를 개선하세요
        - 가독성을 향상시키세요
        - 일관성을 보장하세요
        - 정확성을 검증하세요
        - 타겟 독자에 맞게 최적화하세요
        - 모든 내용을 한국어로 작성하세요""",

        expected_output="""다음을 포함한 완성된 기사:
        - 올바른 문법과 맞춤법
        - 개선된 문장 구조
        - 향상된 가독성
        - 일관된 스타일과 톤
        - 검증된 정확성
        - 타겟 독자에 최적화
        - 전문적인 포맷팅
        - 한국어로 작성된 최종 기사""",

        agent=agent
    )
```

2.5.2. 순차적 프로세스로 크루 구성

4. 크루 생성 함수

`process=Process.sequential` 설정을 통해 작업이 순서대로 실행되도록 합니다. 이것이 바로 협업의 핵심입니다.

```
def create_content_crew(agents, tasks):
    """콘텐츠 생성을 위한 크루 생성"""
    return Crew(
        agents=agents,
        tasks=tasks,
        process=Process.sequential,
        verbose=True
    )
```

5. 전체 프로세스 실행 함수

모든 구성 요소를 조합하여 콘텐츠 생성 전체 과정을 실행합니다.

```
def run_content_creation(topic="Artificial Intelligence in Healthcare"):
    """콘텐츠 생성 전체 프로세스 실행"""
    print("🚀 CrewAI 간단한 크루 시작")
    print("=" * 50)

    # API 키 확인
    if not check_api_key():
        return None

    # LLM 설정
    llm = setup_llm()
    if not llm:
        return None

    # 에이전트 생성
    print("\n👥 에이전트 생성 중...")
    research_agent = create_research_agent(llm)
    writer_agent = create_writer_agent(llm)
    editor_agent = create_editor_agent(llm)

    agents = [research_agent, writer_agent, editor_agent]
    print(f"   ✅ {len(agents)}개 에이전트 생성 완료")

    # 작업 생성
    print("\n📋 작업 생성 중...")
    research_task = create_research_task(research_agent)
    writing_task = create_writing_task(writer_agent)
    editing_task = create_editing_task(editor_agent)

    tasks = [research_task, writing_task, editing_task]
    print(f"   ✅ {len(tasks)}개 작업 생성 완료")

    # 크루 생성
    print("\n👥 크루 생성 중...")
    content_crew = create_content_crew(agents, tasks)
    print("   ✅ 크루 생성 완료")

    # 크루 실행
    print("\n🎬 콘텐츠 생성 시작...")
    print(f"📖 주제: {topic}")

    try:
        result = content_crew.kickoff(inputs={'topic': topic})
```

```
        print(f"\n{'='*60}")
        print("📖 생성된 콘텐츠:")
        print('='*60)
        print(result)
        print(f"\n{'='*60}")
        print("✅ 콘텐츠 생성 완료!")

        return str(result)

    except Exception as e:
        print(f"❌ 실행 중 오류 발생: {e}")
        return None

# 메인 실행
if __name__ == "__main__":
    run_content_creation()
```

실행 과정 (자동 컨텍스트 전달)

1. **1단계 (연구):** `research_agent` 가 `research_task` 를 수행하여 연구 보고서를 만듭니다.
2. **2단계 (작성):** `writer_agent` 가 **(자동으로)** 연구 보고서를 입력받아 `writing_task` 를 수행하고 기사 초안을 작성합니다.
3. **3단계 (편집):** `editor_agent` 가 **(자동으로)** 기사 초안을 입력받아 `editing_task` 를 수행하고 최종 결과물을 완성합니다.

2.6. Context 전달 메커니즘

에이전트 간의 자동 정보 흐름

순차적 프로세스(`Process.sequential`)에서 CrewAI의 가장 강력한 기능 중 하나는 **자동 컨텍스트 전달**입니다. 이전 작업의 결과가 다음 작업의 입력(Context)으로 자동으로 전달됩니다.

이는 마치 컨베이어 벨트처럼, 한 에이전트의 완성물이 다음 에이전트의 작업대로 자연스럽게 옮겨지는 것과 같습니다.

방식 1️⃣: 자동 전달 (Sequential Process)

`Process.sequential` 로 설정 시, tasks 리스트의 순서에 따라 이전 Task의 결과가 다음 Task의 context로 자동 주입됩니다. **simple_crew.py가 이 방식을 사용합니다.**

```
# simple_crew.py에서 사용하는 자동 전달 방식

# 1. 에이전트와 작업 생성
research_agent = create_research_agent(llm)
writer_agent = create_writer_agent(llm)
editor_agent = create_editor_agent(llm)

research_task = create_research_task(research_agent)
writing_task = create_writing_task(writer_agent)
editing_task = create_editing_task(editor_agent)

# 2. 크루 생성 (Process.sequential 설정)
content_crew = Crew(
    agents=[research_agent, writer_agent, editor_agent],
    tasks=[research_task, writing_task, editing_task],
    process=Process.sequential,  # 자동 전달 활성화
    verbose=True
)

# 3. 실행 시 자동 흐름:
# research_task 실행 → 결과물 (연구 보고서)
#   ↓ (자동 전달)
# writing_task 실행 → 결과물 (기사 초안)
#   ↓ (자동 전달)
# editing_task 실행 → 최종 결과물 (완성된 기사)
```

방식 2️⃣: 명시적 전달 (Context 파라미터)

Task 정의 시 `context` 파라미터를 사용하여 어떤 작업의 결과를 입력으로 받을지 직접 지정할 수 있습니다. 복잡한 의존성이 있을 때 유용합니다.

```
# 명시적 context 전달 방식 (복잡한 워크플로우용)

# 1. 작업 생성
research_task = Task(
    description="AI 기술 트렌드 연구",
    expected_output="연구 보고서",
    agent=research_agent
)

market_task = Task(
    description="AI 시장 분석",
    expected_output="시장 분석 보고서",
    agent=market_agent
)

# 2. 여러 작업의 결과를 명시적으로 받기
summary_task = Task(
    description="연구 보고서와 시장 분석을 종합하여 요약 작성",
    expected_output="종합 요약 보고서",
    agent=writer_agent,
    context=[research_task, market_task]  # 두 작업의 결과를 모두 입력으로 받음
)

# 3. 크루 생성
crew = Crew(
    agents=[research_agent, market_agent, writer_agent],
    tasks=[research_task, market_task, summary_task],
    verbose=True
)
```

2.7. 코드 분석: simple_crew.py

협업 파이프라인의 구조

`simple_crew.py` 는 `first_agent.py` 의 개념을 확장하여, 여러 에이전트가 어떻게 협력하여 더 복잡하고 품질 높은 결과물을 만드는지 보여줍니다. 이 코드의 핵심은 **분업과 협업**입니다.

1. 역할 분담 (에이전트 생성)

각기 다른 전문성을 가진 세 개의 에이전트를 별도의 함수(`create_research_agent` , `create_writer_agent` , `create_editor_agent`)로 생성합니다. 이를 통해 각 에이전트의 역할과 책임이 명확해집니다.

2. 작업 분할 (Task 생성)

전체 '기사 작성' 목표를 [연구 → 작성 → 편집]이라는 논리적인 세부 작업으로 나눕니다. 각 작업 또한 별도의 함수(`create_research_task` , `create_writing_task` , `create_editing_task`)로 정의하고, 가장 적합한 에이전트에게 할당합니다.

3. 워크플로우 자동화 (Crew 생성 및 실행)

`create_content_crew` 함수에서 에이전트와 작업을 결합하고, `process=Process.sequential` 설정을 통해 이들의 실행 순서를 정의합니다. 이 설정 하나만으로 연구 결과가 작가에게, 작가의 초안이 편집자에게 자동으로 전달되는 전체 워크플로우가 자동화됩니다.

2.8. Agent 설계 전략 (Agent Design Strategy)

효과적인 에이전트 설계를 위한 원칙

앞서 `first_agent.py` 와 `simple_crew.py` 를 통해 에이전트를 만드는 **방법(How)**을 배웠습니다. 이제 **왜(Why)** 그리고 **언제(When)** 특정 패턴을 사용해야 하는지, 효과적인 에이전트 설계를 위한 전략적 원칙을 학습합니다.

2.8.1. 에이전트 역할 정의 원칙

Role, Goal, Backstory 설계 가이드

1️⃣ Role: 구체적이고 명확하게

2️⃣ Goal: SMART 원칙 적용 측정 가능하게

Goal은 다음 기준을 충족해야 합니다:

- **S**pecific (구체적): "좋은 결과물" → "3-5개의 핵심 인사이트를 도출"
- **M**easurable (측정 가능): "연구 수행" → "500자 이내의 연구 요약 작성"
- **A**chievable (달성 가능): LLM이 현실적으로 수행 가능한 작업

- **R**elevant (관련성): 에이전트의 Role과 일치
- **T**ime-bound (시간 제한): 크루의 max_iter와 연계

3️⃣ Backstory: 효과적인 페르소나 디자인

Backstory는 단순한 배경 설명이 아니라, LLM의 **사고 방식과 행동 패턴**을 형성합니다.

```
# 효과적인 Backstory 예시
backstory="""당신은 10년 이상의 경력을 가진 데이터 분석 전문가입니다.
복잡한 데이터에서 패턴을 발견하고, 비즈니스 의사결정에 필요한
실행 가능한 인사이트를 도출하는 데 탁월합니다.
당신의 분석은 항상 구체적인 수치와 사례를 포함하며,
경영진도 쉽게 이해할 수 있도록 명확하게 전달됩니다.
모든 결과물은 한국어로 작성해주세요."""
```

2.8.2. 단일 책임 원칙 (Single Responsibility Principle)

한 에이전트 = 한 전문 영역

소프트웨어 공학의 **단일 책임 원칙(SRP)**은 에이전트 설계에도 적용됩니다. 각 에이전트는 하나의 명확한 책임만 가져야 합니다.

❌ 잘못된 설계: "만능 에이전트"

```
Agent(
    role='만능 AI 어시스턴트',
    goal='모든 종류의 작업을 완벽하게 수행한다',
    backstory='당신은 연구, 작성, 분석, 코딩 등 모든 것을 할 수 있는 AI입니다.'
)
```

문제점:

- 전문성 부족: 모든 것을 조금씩 알지만 어느 것도 깊이 있게 하지 못함
- 품질 저하: 각 작업에 최적화되지 않은 일반적인 결과물만 생성
- 디버깅 어려움: 문제 발생 시 어떤 부분이 잘못되었는지 파악 곤란

✅ 올바른 설계: 전문화된 에이전트 팀

```
# 01_basics/simple_crew.py 참조
researcher = Agent(
    role='연구 전문가',
    goal='주어진 주제에 대한 포괄적인 정보를 수집한다',
    backstory='정보과학 박사, 정보 수집 및 종합에 특화'
)

writer = Agent(
    role='콘텐츠 작가',
    goal='연구 결과를 바탕으로 매력적인 콘텐츠를 작성한다',
    backstory='8년 경력 전문 작가, 복잡한 정보를 쉽게 전달'
)

editor = Agent(
    role='콘텐츠 편집자',
    goal='콘텐츠의 품질과 명확성을 검토하고 개선한다',
    backstory='저널리즘 배경, 세부사항과 가독성에 특화'
)
```

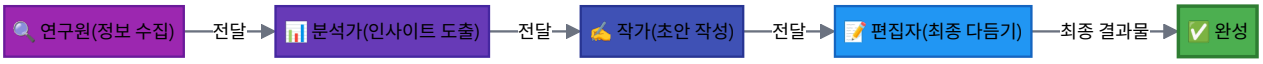
2.8.3. 에이전트 팀 구성 패턴

문제 유형에 맞는 협업 구조 선택

패턴 1: Pipeline (파이프라인) 🔄

특징: 순차적 작업 흐름, 각 단계의 출력이 다음 단계의 입력

적용 사례: 콘텐츠 생성, 데이터 처리, 보고서 작성

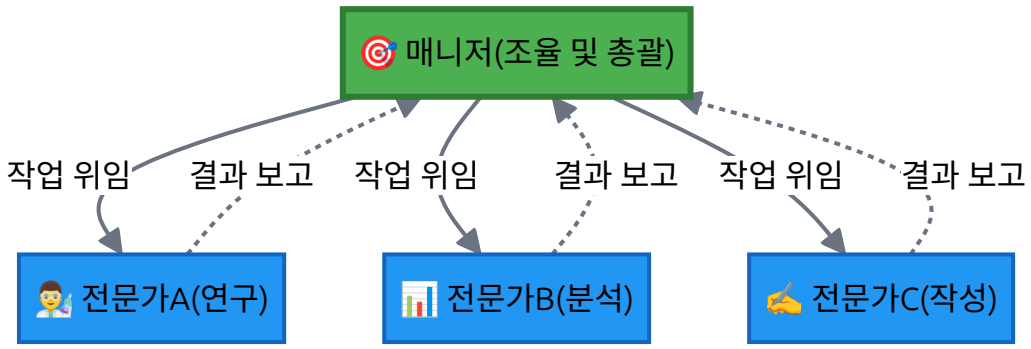


구현: `Process.sequential` 사용 (예: `simple_crew.py`)

패턴 2: Hub-and-Spoke (중앙 집중형) 🎯

특징: 중앙 조율자(매니저)가 전문가들에게 작업 위임

적용 사례: 프로젝트 관리, 복잡한 의사결정, 동적 워크플로우



구현: `Process.hierarchical` + 매니저 설정 (2가지 방식)

방식 1: manager_llm (간단, 권장)

CrewAI가 자동으로 매니저 에이전트를 생성합니다.

```
# Process.hierarchical - 자동 매니저 생성 방식
crew = Crew(
    agents=[researcher, analyst, writer],
    tasks=[research_task, analysis_task, writing_task],
    process=Process.hierarchical,
    manager_llm=ChatOpenAI(model="gpt-4"),  # 매니저용 LLM 지정
    verbose=True
)

# 매니저가 자동으로 생성되어 작업을 조율합니다
```

방식 2: manager_agent (세밀한 제어)

커스텀 매니저 에이전트를 직접 정의합니다.

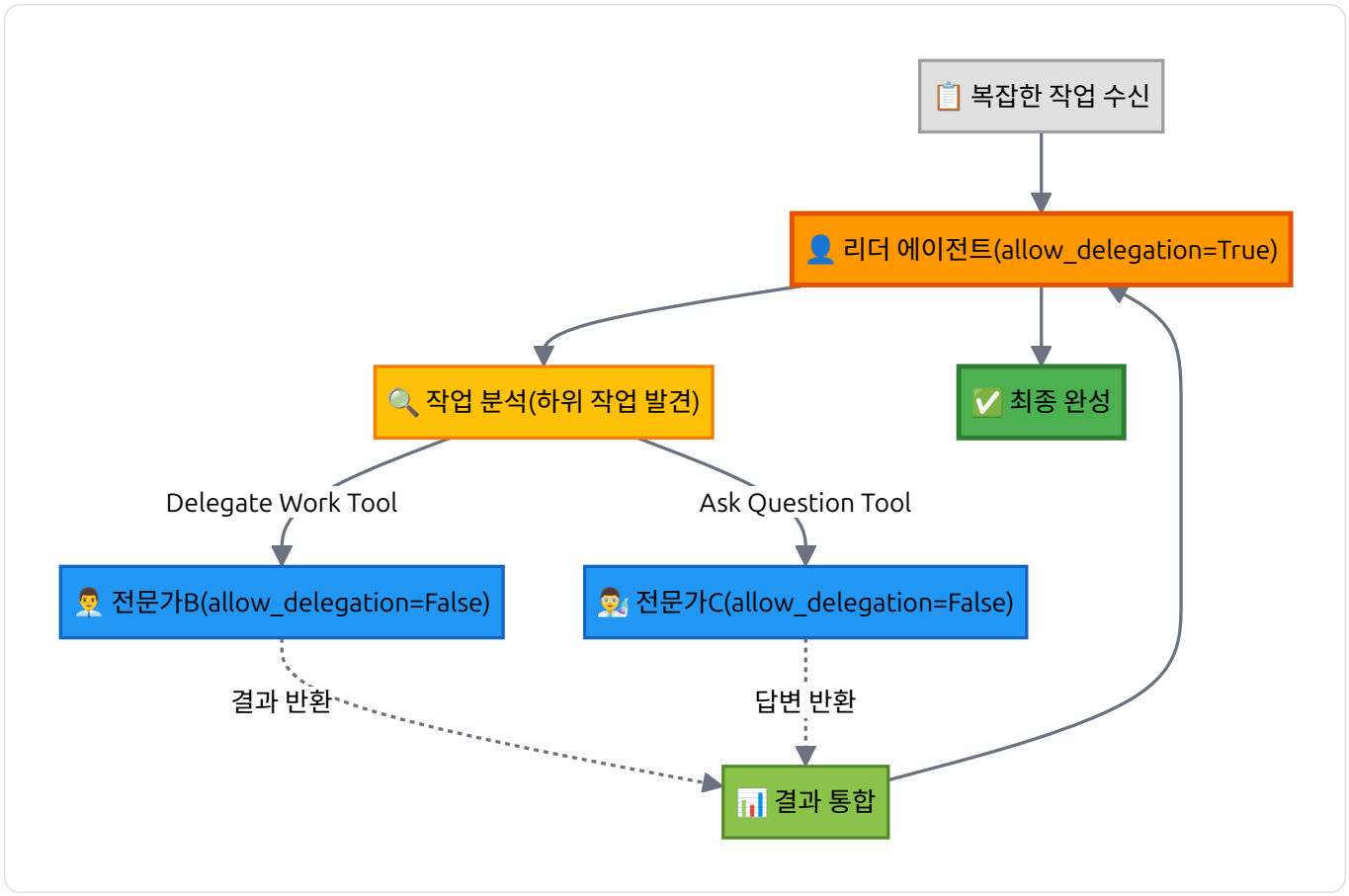
```
# Process.hierarchical - 커스텀 매니저 방식
# 1. 매니저 에이전트 생성
manager = Agent(
    role='프로젝트 매니저',
    goal='팀원들의 작업을 조율하고 최종 결과물을 완성한다',
    backstory="""당신은 10년 경력의 프로젝트 매니저입니다.
    팀원들의 강점을 파악하고 적절한 작업을 할당하며,
    전체 프로젝트가 원활하게 진행되도록 조율합니다.""",
    llm=llm,
    allow_delegation=True  # 작업 위임 권한 필수
)

# 2. 크루 생성
crew = Crew(
    agents=[researcher, analyst, writer],  # 전문가들
    tasks=[research_task, analysis_task, writing_task],
    process=Process.hierarchical,
    manager_agent=manager,  # 커스텀 매니저 지정
    verbose=True
)

# 매니저가 동적으로 작업 순서와 담당자를 결정합니다
```

실제 예제: `02_advanced/multi_agent_collaboration.py`

패턴 3: Delegation (위임 기반 협업) 🤝

특징: 에이전트가 자율적으로 판단하여 다른 에이전트에게 하위 작업을 위임하거나 질문

적용 사례: 복잡한 문제를 동적으로 분해, 유연한 작업 분배, 전문가 지식 활용



구현: `allow_delegation=True` 로 에이전트 설정 (Sequential 프로세스 사용)

```
# Delegation 패턴 예시
# 1. 리더 에이전트 (위임 권한 있음)
leader_agent = Agent(
    role='프로젝트 리더',
    goal='프로젝트 완료',
    backstory='필요시 팀원에게 작업을 위임하거나 질문할 수 있는 리더',
    allow_delegation=True,  # 🔑 위임 도구 자동 활성화
    llm=llm
)

# 2. 전문가 에이전트들 (위임 권한 없음 - 전문 분야에 집중)
specialist_a = Agent(
    role='데이터 분석 전문가',
    goal='데이터 분석 작업 수행',
    backstory='데이터 분석에 특화된 전문가',
    allow_delegation=False,  # 전문가는 자신의 작업에 집중
    llm=llm
)

specialist_b = Agent(
    role='보고서 작성 전문가',
    goal='보고서 작성',
    backstory='보고서 작성에 특화된 전문가',
    allow_delegation=False,
    llm=llm
)

# 3. Sequential 프로세스에서 위임 활성화
crew = Crew(
    agents=[leader_agent, specialist_a, specialist_b],
    tasks=[main_task],
    process=Process.sequential,  # Sequential 프로세스 사용
    verbose=True
)
```

2.8.4. Task 설계 Best Practices

명확한 작업 정의로 품질 향상

원칙 1: Description에 5W1H 명확화

Task의 `description`은 다음을 명확히 전달해야 합니다:

- **What** (무엇을): 정확히 어떤 작업을 수행해야 하는가?
- **Why** (왜): 이 작업의 목적과 중요성은 무엇인가?
- **Who** (누구를 위해): 결과물의 대상 독자는 누구인가?
- **When** (언제까지): 시간 제약이나 우선순위는?
- **Where** (어디서): 정보 출처나 참조 자료는?
- **How** (어떻게): 선호하는 접근 방식이나 방법론은?

```
# ✅ 5W1H가 명확한 Description
Task(
    description="""[What] {topic}에 대한 시장 조사를 수행하세요.
    [Why] 신규 사업 진출 가능성을 평가하기 위함입니다.
    [Who] 경영진과 투자자가 읽을 예정입니다.
    [When] 우선순위 높음, 핵심 정보만 간결하게 작성하세요.
    [Where] 최근 3개월 이내의 시장 보고서와 뉴스를 참조하세요.
    [How] 데이터 기반 분석을 통해 객관적으로 평가하세요.""",
    expected_output="...",
    agent=market_researcher
)
```

원칙 2: Expected Output에 구체적 기준 명시

모호한 기대 결과는 무한 반복과 낮은 품질의 주범입니다.

원칙 3: Context로 작업 간 의존성 명확화

Task의 `context` 파라미터로 이전 작업 결과를 명시적으로 전달합니다. 이는 여러 작업의 결과를 동시에 참조해야 할 때 특히 유용합니다.

```
# 실제 예제: 03_rag_system/rag_crew.py
# Agentic RAG 시스템에서 4단계 파이프라인 구성

# 1단계: 검색 결과 요약
retrieval_summary_task = Task(
    description="문서에서 검색된 정보를 요약하세요",
    expected_output="핵심 정보 요약",
    agent=retrieval_summarizer
)

# 2단계: 요약된 정보 분석 (1단계 결과 사용)
insight_analysis_task = Task(
    description="이전 단계에서 요약된 정보를 분석하세요",
    context=[retrieval_summary_task],  # retrieval_summary_task의 결과를 입력으로 받음
    agent=insight_analyzer
)

# 3단계: 답변 생성 (2단계 결과 사용)
response_generation_task = Task(
    description="분석 결과를 바탕으로 답변을 생성하세요",
    context=[insight_analysis_task],  # insight_analysis_task 결과 사용
    agent=response_generator
)

# 4단계: 사실 검증 (1단계 + 3단계 결과 사용)
fact_checking_task = Task(
    description="생성된 답변을 검증하고 최종 편집하세요",
    context=[response_generation_task, retrieval_summary_task],  # 다중 의존성
    agent=fact_checker
)

# 실행 흐름:
# retrieval_summary_task → insight_analysis_task → response_generation_task → fact_
#             ↓
#             └──────────────────────────────────────────────────────────────────
```

2.8.5. 실전 설계 체크리스트

에이전트 시스템 설계 전 점검 사항

- 전체 목표를 독립적인 하위 작업으로 분해했는가?
- 각 하위 작업은 하나의 명확한 목적을 가지는가?
- 작업 간 의존성과 순서가 명확한가?
- 각 하위 작업에 필요한 전문성을 식별했는가?
- 역할이 너무 넓거나 좁지 않은가? (적절한 범위)
- 역할 간 중복이나 공백은 없는가?
- Role은 구체적이고 전문화되어 있는가?
- Goal은 SMART 원칙을 충족하는가?
- Backstory가 기대하는 작업 스타일을 명확히 전달하는가?
- 필요한 도구(tools)가 할당되었는가?
- Description에 5W1H가 포함되어 있는가?
- Expected Output이 측정 가능한가? (형식, 분량, 품질 기준)
- Context로 작업 의존성을 명시했는가?
- 입력 변수({topic} 등)가 올바르게 정의되었는가?
- 작업 흐름에 적합한 프로세스를 선택했는가?
- 순차형(Sequential): 명확한 단계가 있는 작업
- 계층형(Hierarchical): 유동적이고 복잡한 작업
- max_iter로 무한 반복을 방지했는가?
- 결과물 검증 단계가 포함되어 있는가?
- 에러 처리와 재시도 로직이 있는가?
- 비용과 시간 제약을 고려했는가?

2.8.6. 설계 전략 핵심 요약

효과적인 에이전트 시스템을 위한 핵심 원칙

Chapter 3. 고급 기능과 도구

학습 목표

- 에이전트에게 자신만의 능력을 부여하는 **커스텀 도구**를 개발한다.
- 외부 API를 연동하여 에이전트가 실시간 정보에 접근하도록 한다.
- **순차(Sequential)**와 **계층(Hierarchical)** 프로세스의 차이를 이해하고 적절한 협업 방식을 선택한다.
- 복잡한 프로세스를 **워크플로우로 정의**하고 동적으로 실행하는 방법을 학습한다.
- Agent 개발 시 발생하는 **주요 이슈와 해결 방안**을 파악한다.

3.1. 예제 파일 개요: 02_advanced

이 챕터에서는 `02_advanced` 폴더의 예제들을 통해 CrewAI의 고급 기능을 마스터합니다.

custom_tools.py	내부 로직 기반 커스텀 도구 (7종)	@tool 데코레이터, 텍스트 분석, 데이터 변환, 리스트 처리, 자율적 도구 선택
serper_api_tool_calling_example.py	외부 API 연동 (웹 검색)	실시간 정보 접근, Multi-Tool Agent, OpenAI Tool Calling
mcp_server_integration.py	MCP 프로토콜 통합	Model Context Protocol, Tools/Resources/Prompts, 표준화된 LLM 통합
multi_agent_collaboration.py	협업 프로세스 비교	Sequential vs. Hierarchical 프로세스 심층 분석
workflow_orchestration.py	워크플로우 오케스트레이션	동적 Crew 구성, 재사용 가능한 워크플로우 정의
async_and_planning_example.py	비동기 실행 및 AI 기반 계획	kickoff_async(), kickoff_for_each(), planning=True, 성능 최적화
memory_system_example.py	메모리 시스템	Short-term/Long-term/Entity Memory, 대화 맥락 유지, 사용자 선호도 학습

3.2. 커스텀 도구 개발 (custom_tools.py)

에이전트에게 '능력'을 부여하기: @tool 데코레이터

도구(Tool)는 에이전트가 LLM의 한계를 넘어 다양한 작업을 수행하게 해주는 핵심 기능입니다. CrewAI에서는 `@tool` 데코레이터 하나로 평범한 파이썬 함수를 강력한 에이전트 도구로 만들 수 있습니다.

```
# 02_advanced/custom_tools.py
import warnings
import os
from crewai import Agent, Task, Crew, Process
from crewai.tools import tool
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv
import json
from datetime import datetime

# 중요: 함수의 독스트링(docstring)은 LLM이 도구의 기능을 이해하는 설명서 역할을 합니다.

@tool("TextAnalyzerTool")
def text_analyzer_tool(text: str) -> str:
    """텍스트를 분석하여 단어 수, 문자 수, 문장 수 등의 통계 정보를 제공합니다."""
    try:
        # 기본 통계
        char_count = len(text)
        char_no_spaces = len(text.replace(" ", "").replace("\n", ""))
        word_count = len(text.split())

        # 문장 수 계산
        sentence_endings = ['.', '!', '?', '。']
        sentence_count = sum(text.count(ending) for ending in sentence_endings)
        if sentence_count == 0:
            sentence_count = 1

        # 줄 수
        line_count = len(text.split('\n'))

        # 평균 단어 길이
        avg_word_length = char_no_spaces / word_count if word_count > 0 else 0

        result = f"""📊 텍스트 분석 결과:
• 총 문자 수: {char_count}자 (공백 포함)
• 순수 문자 수: {char_no_spaces}자 (공백 제외)
• 단어 수: {word_count}개
• 문장 수: {sentence_count}개
• 줄 수: {line_count}줄
• 평균 단어 길이: {avg_word_length:.1f}자"""

        return result
    except Exception as e:
        return f"텍스트 분석 오류: {e}"
```

```
@tool("CalculatorTool")
def calculator_tool(expression: str) -> str:
    """주어진 수학 표현식을 계산합니다. 기본 사칙연산과 괄호를 지원합니다.

    예: "2 + 3 * 4", "(100 - 20) / 2", "2024 - 1990"
    """
    try:
        # 보안을 위해 허용된 문자만 사용
        allowed_chars = set("0123456789+-*/.() ")
        if not all(c in allowed_chars for c in expression):
            return '⚠️ 오류: 허용되지 않는 문자가 포함되어 있습니다. 숫자와 +, -, *, /, (, )만

        result = eval(expression)
        return f"🔢 계산 결과: {expression} = {result}"
    except ZeroDivisionError:
        return "⚠️ 오류: 0으로 나눌 수 없습니다."
    except Exception as e:
        return f"⚠️ 계산 오류: {e}"
```

3.2.1. 다양한 커스텀 도구 제작

실습: custom_tools.py - 7가지 실용 도구

이 예제에서는 다음 7가지 도구를 구현하여 에이전트에게 다채로운 능력을 부여합니다:

1️⃣ 리스트 처리 도구

```
# 02_advanced/custom_tools.py

@tool("ListProcessorTool")
def list_processor_tool(operation: str, items: str) -> str:
    """리스트 데이터를 처리합니다. 정렬, 중복 제거, 통계 계산 등을 수행합니다.

    Args:
        operation: 'sort' (정렬), 'unique' (중복 제거), 'stats' (통계), 'reverse' (역순
        items: 쉼표로 구분된 항목들 (예: "사과,바나나,사과,딸기")
    """
    try:
        item_list = [item.strip() for item in items.split(',')]

        if operation.lower() == 'sort':
            sorted_list = sorted(item_list)
            result = ', '.join(sorted_list)
            return f"📋 정렬 결과 ({len(sorted_list)}개 항목):\n{result}"

        elif operation.lower() == 'unique':
            unique_list = list(dict.fromkeys(item_list))  # 순서 유지하며 중복 제거
            removed = len(item_list) - len(unique_list)
            result = ', '.join(unique_list)
            return f"🔍 중복 제거 결과 ({len(unique_list)}개 항목, {removed}개 중복 제거)

        elif operation.lower() == 'stats':
            from collections import Counter
            total = len(item_list)
            unique = len(set(item_list))
            counter = Counter(item_list)
            most_common = counter.most_common(3)

            stats = f"""📊 리스트 통계:
• 전체 항목 수: {total}개
• 고유 항목 수: {unique}개
• 중복 항목 수: {total - unique}개
• 가장 많이 등장한 항목:"""

            for item, count in most_common:
                stats += f"\n  - {item}: {count}회"

            return stats

        elif operation.lower() == 'reverse':
            reversed_list = list(reversed(item_list))
```

```
            result = ', '.join(reversed_list)
            return f"🔄 역순 정렬 결과 ({len(reversed_list)}개 항목):\n{result}"

        else:
            return "⚠️ 오류: 지원하지 않는 작업입니다. 'sort', 'unique', 'stats', 'revers

    except Exception as e:
        return f"⚠️ 리스트 처리 오류: {e}"
```

2️⃣ 데이터 변환 도구 (Data Formatter)

```
@tool("DataFormatterTool")
def data_formatter_tool(data_type: str, data: str) -> str:
    """데이터를 지정된 형식(JSON, CSV, Markdown)으로 포맷팅합니다.

    Args:
        data_type: 'json', 'csv', 'markdown' 중 하나
        data: 포맷팅할 데이터 (키:값 쌍을 쉼표로 구분, 예: "이름:홍길동,나이:30,직업:개발자")
    """
    try:
        # 데이터 파싱
        items = data.split(',')
        data_dict = {}
        for item in items:
            if ':' in item:
                key, value = item.split(':', 1)
                data_dict[key.strip()] = value.strip()

        if not data_dict:
            return "오류: 올바른 형식의 데이터가 아닙니다. 예: '이름:홍길동,나이:30'"

        # 형식에 따라 변환
        if data_type.lower() == 'json':
            formatted = json.dumps(data_dict, ensure_ascii=False, indent=2)
            return f"📄 JSON 형식:\\n```json\\n{formatted}\\n```"

        elif data_type.lower() == 'csv':
            headers = ','.join(data_dict.keys())
            values = ','.join(data_dict.values())
            formatted = f"{headers}\\n{values}"
            return f"📊 CSV 형식:\\n```csv\\n{formatted}\\n```"

        elif data_type.lower() == 'markdown':
            formatted = "| 항목 | 값 |\\n|------|------|\\n"
            for key, value in data_dict.items():
                formatted += f"| {key} | {value} |\\n"
            return f"📝 Markdown 표 형식:\\n```markdown\\n{formatted}```"

        else:
            return f"오류: 지원하지 않는 형식입니다. 'json', 'csv', 'markdown' 중 하나를 선

    except Exception as e:
        return f"데이터 포맷팅 오류: {e}"
```

3️⃣ 시스템/파일 도구 (Timestamp & File I/O)

```
@tool("TimestampTool")
def timestamp_tool(operation: str) -> str:
    """현재 시간과 날짜 정보를 다양한 형식으로 제공합니다.

    Args:
        operation: 'now' (현재 시각), 'date' (날짜만), 'time' (시간만), 'iso' (ISO 형식)
    """
    try:
        now = datetime.now()

        if operation.lower() == 'now':
            formatted = now.strftime("%Y년 %m월 %d일 %H시 %M분 %S초")
            return f"🕐 현재 시각: {formatted}"

        elif operation.lower() == 'date':
            formatted = now.strftime("%Y년 %m월 %d일 (%A)")
            weekday_kr = {
                'Monday': '월요일', 'Tuesday': '화요일', 'Wednesday': '수요일',
                'Thursday': '목요일', 'Friday': '금요일', 'Saturday': '토요일', 'Sunday
            }
            day_name = now.strftime("%A")
            formatted = formatted.replace(day_name, weekday_kr.get(day_name, day_na
            return f"📅 오늘 날짜: {formatted}"

        elif operation.lower() == 'time':
            formatted = now.strftime("%H시 %M분 %S초")
            period = "오전" if now.hour < 12 else "오후"
            hour_12 = now.hour if now.hour <= 12 else now.hour - 12
            hour_12 = 12 if hour_12 == 0 else hour_12
            return f"⏰ 현재 시각: {period} {hour_12}시 {now.minute}분 {now.second}초'

        elif operation.lower() == 'iso':
            formatted = now.isoformat()
            return f"🌐 ISO 8601 형식: {formatted}"

        else:
            return "⚠️ 오류: 지원하지 않는 형식입니다. 'now', 'date', 'time', 'iso' 중 하

    except Exception as e:
        return f"⚠️ 시간 정보 조회 오류: {e}"


@tool("FileWriteTool")
def file_write_tool(filename: str, content: str) -> str:
```

```
    """지정된 이름의 파일에 내용을 저장합니다. data/ 디렉토리에 저장됩니다."""
    try:
        file_path = os.path.join(DATA_PATH, filename)
        with open(file_path, 'w', encoding='utf-8') as f:
            f.write(content)

        # 파일 크기 계산
        file_size = os.path.getsize(file_path)
        return f"💾 파일 저장 완료: '{filename}' ({file_size} bytes, {DATA_PATH}/ 디렉
    except Exception as e:
        return f"⚠️ 파일 쓰기 오류: {e}"


@tool("FileReadTool")
def file_read_tool(filename: str) -> str:
    """data/ 디렉토리에서 지정된 이름의 파일을 읽어 내용을 반환합니다."""
    try:
        file_path = os.path.join(DATA_PATH, filename)
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()

        file_size = os.path.getsize(file_path)
        line_count = len(content.split('\n'))

        return f"""📂 '{filename}' 파일 읽기 성공 ({file_size} bytes, {line_count}줄):
{'='*50}
{content}
{'='*50}"""
    except FileNotFoundError:
        return f"⚠️ 오류: '{filename}' 파일을 찾을 수 없습니다. {DATA_PATH}/ 디렉토리를 확인
    except Exception as e:
        return f"⚠️ 파일 읽기 오류: {e}"
```

3.2.2. 자율적 도구 사용: 복합 문제 해결

실습: custom_tools.py

앞에서 배운 **7가지 도구**를 하나의 에이전트에 장착하여, 복잡한 멀티스텝 작업을 자율적으로 해결하는 실습입니다.

```
# 02_advanced/custom_tools.py

# 1. LLM 설정
llm = ChatOpenAI(
    model=os.getenv("DEFAULT_LLM_MODEL", "gpt-4"),
    temperature=0.1,
    api_key=os.getenv("OPENAI_API_KEY")
)

# 2. 다양한 도구를 장착한 에이전트 생성
tool_master_agent = Agent(
    role='다재다능한 데이터 분석 보조원',
    goal='주어진 모든 도구를 활용하여 복잡한 데이터 처리 작업을 수행한다',
    backstory="""당신은 텍스트 분석, 데이터 포맷팅, 계산, 리스트 처리, 파일 관리 등
    다양한 능력을 가진 전문 AI 어시스턴트입니다.
    각 도구의 특성을 정확히 이해하고 있으며, 주어진 작업에 가장 적합한 도구를
    자율적으로 선택하여 사용합니다.
    모든 최종 보고서는 한국어로 명확하게 작성해야 합니다.""",
    llm=llm,
    tools=[
        text_analyzer_tool,    # 1. 텍스트 통계 분석 (3.2)
        calculator_tool,       # 2. 수학 계산 (3.2)
        list_processor_tool,   # 3. 리스트 정렬/중복제거/통계 (3.2.1)
        data_formatter_tool,   # 4. JSON/CSV/Markdown 변환 (3.2.1)
        timestamp_tool,        # 5. 시간 정보 (3.2.1)
        file_write_tool,       # 6. 파일 저장 (3.2.1)
        file_read_tool         # 7. 파일 읽기 (3.2.1)
    ],
    verbose=True,
    allow_delegation=False
)

# 3. 샘플 데이터 정의
sample_text = """CrewAI는 멀티 에이전트 시스템을 쉽게 구축할 수 있는 프레임워크입니다.
각 에이전트는 고유한 역할과 목표를 가지며, 협업을 통해 복잡한 문제를 해결합니다."""

sample_data = "이름:CrewAI,유형:프레임워크,언어:Python,버전:0.40.0"

sample_fruits = "사과,바나나,오렌지,사과,포도,바나나,사과,딸기"

# 4. 여러 단계의 작업이 포함된 복합적인 Task 정의
complex_task = Task(
    description=f"""다음 작업들을 순서대로 수행하세요:
```

```
1. 텍스트 분석: 다음 텍스트를 분석하세요.
   "{sample_text}"

2. 데이터 포맷팅: 다음 데이터를 JSON 형식으로 변환하세요.
   "{sample_data}"

3. 리스트 처리: 다음 과일 리스트의 통계를 분석하세요.
   "{sample_fruits}"

4. 계산: (2024 - 2020) * 365 를 계산하세요.

5. 타임스탬프: 현재 날짜와 시각을 조회하세요.

6. 파일 저장: 위의 모든 결과를 종합하여 'analysis_report.txt' 파일에 저장하세요.

모든 결과를 포함한 종합 보고서를 작성하세요.""",

    expected_output="""다음 내용을 포함한 종합 보고서 (한국어):
• 텍스트 분석 결과 (문자 수, 단어 수 등)
• JSON 포맷팅된 데이터
• 과일 리스트 통계 (전체 개수, 중복 등)
• 계산 결과
• 현재 시각 정보
• 파일 저장 확인
• 최종 요약 (200자 이내)""",

    agent=tool_master_agent
)

# 5. 크루 생성 및 실행
task_crew = Crew(
    agents=[tool_master_agent],
    tasks=[complex_task],
    process=Process.sequential,
    verbose=True
)

result = task_crew.kickoff()
```

3.2.3. 도구 설계 5가지 원칙 (1/3)

LLM이 이해하고 사용할 수 있는 완벽한 도구 만들기

CrewAI 에이전트가 도구를 **정확하게 이해**하고, **적절한 상황에 선택**하며, **올바르게 사용**할 수 있도록 하는 5가지 핵심 원칙을 살펴봅니다.

1️⃣ 명확한 인터페이스 (Clear Interface)

핵심: LLM이 도구를 정확하게 호출할 수 있도록 명확한 함수 시그니처와 타입 힌트를 제공해야 합니다.

```
# ✅ 좋은 예: 명확한 타입 힌트와 구체적인 파라미터명
@tool("ListProcessorTool")
def list_processor_tool(operation: str, items: str) -> str:
    """리스트 데이터를 처리합니다..."""

# ❌ 나쁜 예: 타입 힌트 없음, 모호한 파라미터명
@tool("ProcessTool")
def process(x, y):
    """뭔가 처리합니다..."""
```

2️⃣ 자세한 Docstring (Comprehensive Documentation)

핵심: Docstring은 LLM이 읽는 "사용 설명서"입니다. 목적, 파라미터, 예시를 반드시 포함하세요.

```
# ✅ 좋은 예: 상세한 docstring with Args
@tool("DataFormatterTool")
def data_formatter_tool(data_type: str, data: str) -> str:
    """데이터를 지정된 형식(JSON, CSV, Markdown)으로 포맷팅합니다.

    Args:
        data_type: 'json', 'csv', 'markdown' 중 하나
        data: 포맷팅할 데이터 (키:값 쌍을 쉼표로 구분, 예: "이름:홍길동,나이:30,직업:개발자")
    """

# ❌ 나쁜 예: 불충분한 docstring
@tool("FormatterTool")
def format_data(type, data):
    """데이터를 포맷팅합니다."""
```

3.2.3. 도구 설계 5가지 원칙 (2/3)

3️⃣ 철저한 에러 처리 (Robust Error Handling)

핵심: 모든 도구는 예상치 못한 입력이나 실행 오류에 대비해야 합니다. 에이전트가 오류를 이해하고 대응할 수 있도록 명확한 메시지를 반환하세요.

```
# ✅ 좋은 예: 구체적인 에러 처리
@tool("CalculatorTool")
def calculator_tool(expression: str) -> str:
    try:
        # 보안 검증
        allowed_chars = set("0123456789+-*/.() ")
        if not all(c in allowed_chars for c in expression):
            return '⚠️ 오류: 허용되지 않는 문자가 포함되어 있습니다.'

        result = eval(expression)
        return f"🔢 계산 결과: {expression} = {result}"

    except ZeroDivisionError:
        return "⚠️ 오류: 0으로 나눌 수 없습니다."
    except Exception as e:
        return f"⚠️ 계산 오류: {e}"

# ❌ 나쁜 예: 에러 처리 없음 (에이전트가 멈출 수 있음)
@tool("CalcTool")
def calc(expr: str) -> str:
    return f"결과: {eval(expr)}"  # 위험!
```

4️⃣ 단일 책임 원칙 (Single Responsibility Principle)

핵심: 각 도구는 하나의 명확한 목적만 가져야 합니다. 복잡한 작업은 여러 도구로 분할하세요.

```
# ✅ 좋은 예: 각 도구가 단일 책임 수행
@tool("TextAnalyzerTool")
def text_analyzer_tool(text: str) -> str:
    """텍스트 통계 분석만 수행"""
    # 단어 수, 문자 수, 문장 수 등만 계산

@tool("FileWriteTool")
def file_write_tool(filename: str, content: str) -> str:
    """파일 쓰기만 수행"""
    # 파일 저장 로직만 구현

# ❌ 나쁜 예: 하나의 도구가 너무 많은 일을 함
@tool("SuperTool")
def super_tool(action: str, *args) -> str:
    """텍스트 분석, 파일 쓰기, 계산, 검색 등 모든 것을 수행"""
    if action == "analyze": ...
    elif action == "write": ...
    # LLM이 언제 어떻게 사용할지 혼란스러워함
```

3.2.3. 도구 설계 5가지 원칙 (3/3)

5️⃣ 사용자 친화적 출력 (User-Friendly Output)

핵심: 도구의 출력은 사람이 읽기 쉽고, LLM이 해석하기 쉬운 형식이어야 합니다. 구조화된 텍스트와 시각적 요소를 활용하세요.

```
# ✅ 좋은 예: 이모지와 구조화된 출력
@tool("TextAnalyzerTool")
def text_analyzer_tool(text: str) -> str:
    result = f"""📊 텍스트 분석 결과:
• 총 문자 수: {char_count}자 (공백 포함)
• 순수 문자 수: {char_no_spaces}자 (공백 제외)
• 단어 수: {word_count}개
• 문장 수: {sentence_count}개
• 줄 수: {line_count}줄
• 평균 단어 길이: {avg_word_length:.1f}자"""
    return result

# ❌ 나쁜 예: 가독성 낮은 출력
@tool("AnalyzeTool")
def analyze(text: str) -> str:
    return f"{len(text)},{len(text.split())},{text.count('.')}"
    # 무엇이 무엇인지 알 수 없음
```

📚 5가지 원칙 종합 예시

완벽한 도구는 위 5가지 원칙을 모두 만족합니다:

```
@tool("ListProcessorTool")  # 원칙 1: 명확한 도구명
def list_processor_tool(operation: str, items: str) -> str:  # 원칙 1: 타입 힌트
    """리스트 데이터를 처리합니다. 정렬, 중복 제거, 통계 계산 등을 수행합니다.
    # 원칙 2: 명확한 설명

    Args:  # 원칙 2: 상세한 파라미터 문서
        operation: 'sort', 'unique', 'stats', 'reverse' 중 하나
        items: 쉼표로 구분된 항목들 (예: "사과,바나나,사과,딸기")
    """
    try:  # 원칙 3: 에러 처리
        item_list = [item.strip() for item in items.split(',')]

        if operation.lower() == 'unique':  # 원칙 4: 단일 기능 수행
            unique_list = list(dict.fromkeys(item_list))
            removed = len(item_list) - len(unique_list)
            result = ', '.join(unique_list)
            # 원칙 5: 사용자 친화적 출력
            return f"🔍 중복 제거 결과 ({len(unique_list)}개 항목, {removed}개 중복):\n{
        # ... 다른 operation 처리 ...

    except Exception as e:  # 원칙 3: 명확한 에러 메시지
        return f"⚠️ 리스트 처리 오류: {e}"
```

3.3. 외부 API 연동 (serper_api_tool_calling_example.py)

에이전트에게 실시간 웹 검색 능력 부여하기

지금까지는 내부 로직으로만 동작하는 도구를 배웠습니다. 이제 **외부 API**를 연동하여 에이전트가 실시간 정보에 접근하도록 만들어봅시다. `serper_api_tool_calling_example.py` 는 **Serper API**를 활용하여 웹, 뉴스, 이미지 검색 기능을 제공하는 예제입니다.

📦 구현할 3가지 검색 도구

SerperSearchTool	/search	일반 웹 검색 (지식 그래프 + 검색 결과)
SerperNewsTool	/news	최신 뉴스 검색 (출처 + 날짜 포함)
SerperImageTool	/images	이미지 검색 (제목 + 이미지 URL)

🔧 환경 설정

```
# 02_advanced/serper_api_tool_calling_example.py
import warnings
import os
import requests
from crewai import Agent, Task, Crew, Process
from crewai.tools import tool
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()

# API 키 확인 함수
def check_api_keys():
    """OpenAI와 Serper API 키가 설정되었는지 확인합니다."""
    if not os.getenv("OPENAI_API_KEY"):
        print("❌ OpenAI API 키가 설정되지 않았습니다!")
        return False
    if not os.getenv("SERPER_API_KEY"):
        print("❌ Serper API 키가 설정되지 않았습니다!")
        print("💡 https://serper.dev 에서 무료로 발급받을 수 있습니다.")
        return False
    print("✅ API 키 확인 완료")
    return True
```

3.3.1. Serper API 도구 구현

3가지 검색 도구의 상세 구현

1️⃣ 웹 검색 도구 (SerperSearchTool)

```
# 02_advanced/serper_api_tool_calling_example.py

@tool("SerperSearchTool")
def serper_search_tool(query: str) -> str:
    """Serper API를 사용한 고급 웹 검색 도구입니다.
    Google 검색 결과를 구조화된 형태로 반환하여 일반적인 질문에 대한 답변을 찾을 때 유용합니다.

    Args:
        query (str): 검색 질의어.

    Returns:
        str: 요약된 검색 결과 텍스트.
    """
    try:
        api_key = os.getenv("SERPER_API_KEY")
        url = "https://google.serper.dev/search"
        headers = {"X-API-KEY": api_key, "Content-Type": "application/json"}
        payload = {"q": query, "gl": "kr", "hl": "ko"}  # gl: 국가, hl: 언어

        response = requests.post(url, headers=headers, json=payload, timeout=15)
        response.raise_for_status()
        data = response.json()

        # 검색 결과를 요약하여 문자열로 반환
        summary = []
        if data.get("knowledgeGraph"):
            summary.append(f"- 지식 그래프: {data['knowledgeGraph'].get('description'
        if data.get("organic"):
            summary.append("\n- 주요 검색 결과:")
            for i, result in enumerate(data["organic"][:3], 1):
                summary.append(f"  {i}. {result['title']}: {result['snippet']}")

        return "\n".join(summary) if summary else f"'{query}'에 대한 검색 결과를 찾을 수

    except Exception as e:
        return f"Serper API 검색 중 오류 발생: {e}"
```

2️⃣ 뉴스 검색 도구 (SerperNewsTool)

```
@tool("SerperNewsTool")
def serper_news_tool(query: str) -> str:
    """Serper API를 사용한 뉴스 검색 도구입니다.
    특정 주제에 대한 최신 뉴스 기사를 찾을 때 사용합니다.

    Args:
        query (str): 검색 질의어.

    Returns:
        str: 상위 뉴스 요약 텍스트.
    """
    try:
        api_key = os.getenv("SERPER_API_KEY")
        url = "https://google.serper.dev/news"
        headers = {"X-API-KEY": api_key, "Content-Type": "application/json"}
        payload = {"q": query, "gl": "kr", "hl": "ko"}

        response = requests.post(url, headers=headers, json=payload, timeout=15)
        response.raise_for_status()
        data = response.json()

        summary = [f"'{query}' 관련 최신 뉴스:"]
        if data.get("news"):
            for i, news in enumerate(data["news"][:3], 1):
                summary.append(f"- {news['title']} ({news['source']}, {news['date']
        return "\n".join(summary)

    except Exception as e:
        return f"Serper API 뉴스 검색 중 오류 발생: {e}"
```

3️⃣ 이미지 검색 도구 (SerperImageTool)

```
@tool("SerperImageTool")
def serper_image_tool(query: str) -> str:
    """Serper API를 사용한 이미지 검색 도구입니다.
    특정 주제와 관련된 이미지를 찾고 이미지 URL 목록을 반환합니다.

    Args:
        query (str): 검색 질의어.

    Returns:
        str: 상위 이미지 링크 요약 텍스트.
    """
    try:
        api_key = os.getenv("SERPER_API_KEY")
        url = "https://google.serper.dev/images"
        headers = {"X-API-KEY": api_key, "Content-Type": "application/json"}
        payload = {"q": query, "gl": "kr", "hl": "ko"}

        response = requests.post(url, headers=headers, json=payload, timeout=15)
        response.raise_for_status()
        data = response.json()

        summary = [f"'{query}' 관련 이미지 링크:"]
        if data.get("images"):
            for i, image in enumerate(data["images"][:3], 1):
                summary.append(f"- {image['title']}: {image['imageUrl']}")
        return "\n".join(summary)

    except Exception as e:
        return f"Serper API 이미지 검색 중 오류 발생: {e}"
```

3.3.2. Multi-Tool Agent 구성 및 실행

3가지 검색 도구를 자율적으로 사용하는 에이전트

이제 3가지 Serper 도구를 하나의 에이전트에 장착하고, 복합적인 검색 Task를 수행하도록 설정합니다.

```
# 02_advanced/serper_api_tool_calling_example.py

def setup_serper_crew(topic: str):
    """Serper API 도구를 사용하는 Crew를 설정합니다."""

    # 1. OpenAI LLM 설정 (Tool Calling 지원 모델)
    openai_llm = ChatOpenAI(
        model="gpt-4",
        temperature=0.1,
        api_key=os.getenv("OPENAI_API_KEY")
    )
    print("✅ OpenAI LLM 설정 완료")

    # 2. Serper 검색 전문가 Agent 정의
    serper_agent = Agent(
        role="Serper 검색 전문가",
        goal="Serper API를 활용하여 사용자의 질문에 대해 포괄적이고 정확한 검색 결과를 제공한다",
        backstory="""당신은 Serper API를 활용한 고급 검색의 전문가입니다.
        웹 검색, 뉴스 검색, 이미지 검색 등 다양한 검색 도구를 활용하여
        사용자의 질문에 대한 다각도적이고 신뢰할 수 있는 정보를 제공합니다.
        모든 응답은 한국어로 작성해주세요.""",
        llm=openai_llm,
        tools=[serper_search_tool, serper_news_tool, serper_image_tool],  # 3개 도구
        verbose=True
    )
    print("✅ Serper 검색 전문가 Agent 생성 완료")

    # 3. Task 정의: 에이전트가 여러 도구를 자율적으로 선택하도록 유도
    serper_task = Task(
        description=f"""'{topic}'에 대한 종합적인 정보 수집을 위해 Serper API를 활용하세요.

        다음 작업을 수행하세요:
        1. 웹 검색으로 주제에 대한 일반 정보를 찾으세요
        2. 뉴스 검색으로 최신 동향을 파악하세요
        3. 관련 이미지를 찾아 보고서에 포함시키세요

        모든 결과를 종합하여 한국어로 보고서를 작성하세요.""",

        expected_output=f"""'{topic}'에 대한 종합 분석 보고서 (한국어):
        • 주제 개요 (웹 검색 결과)
        • 최신 뉴스 (3개 이상)
        • 관련 이미지 URL (3개 이상)
        • 최종 요약""",
```

```
        agent=serper_agent
    )
    print("✅ Serper Task 생성 완료")

    # 4. Crew 구성
    return Crew(
        agents=[serper_agent],
        tasks=[serper_task],
        process=Process.sequential,
        verbose=True
    )
```

📊 실행 흐름

```
def main():
    print("🚀 Serper API와 Multi-Tool Agent 연동 예제 시작")

    if not check_api_keys():
        return

    topic = "ChatGPT와 Claude 최신 기능 비교"
    serper_crew = setup_serper_crew(topic)

    print("\n🎬 Crew 실행 시작...")
    try:
        result = serper_crew.kickoff()
        print("\n--- 최종 결과 ---")
        print(result)
        print("\n✅ Crew 실행 완료!")
    except Exception as e:
        print(f"❌ Crew 실행 중 오류 발생: {e}")

if __name__ == "__main__":
    main()
```

3.4. MCP 프로토콜 통합 (mcp_server_integration.py)

에이전트 도구를 서버로 확장하기

> 3.3에서 배운 `@tool` 데코레이터는 간단하지만, 다음과 같은 한계가 있습니다:
>
> **❌ 문제 1: 재사용 불가**
>
> 도구를 다른 애플리케이션에서 사용하려면 코드를 복사해야 함
>
> **❌ 문제 2: 데이터 접근 제한**
>
> 함수 호출만 가능하고, 지속적인 데이터 소스에 접근하기 어려움
>
> **❌ 문제 3: 중앙 관리 어려움**
>
> 프롬프트 템플릿이나 설정을 여러 곳에 흩어져 관리

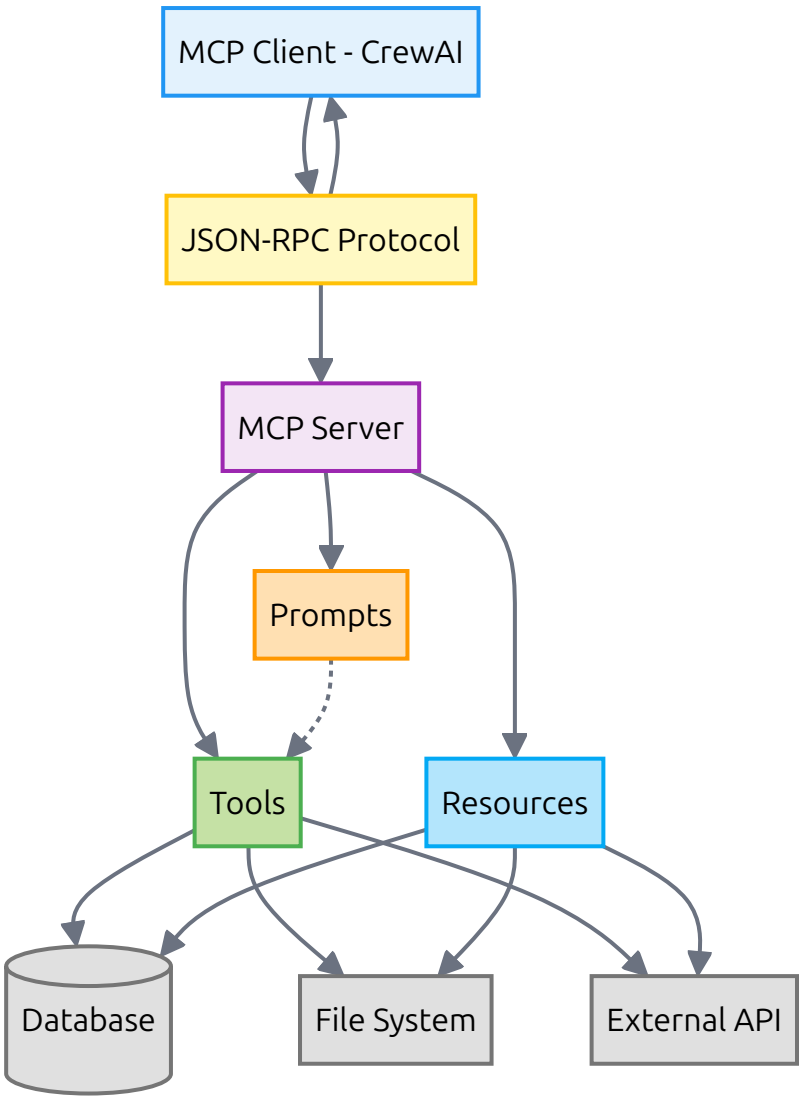
🌟 MCP가 적합한 사례

🎯 예제 프로젝트: 날씨 정보 MCP 서버

`mcp_server_integration.py` 는 날씨 정보를 제공하는 MCP 서버를 구축하는 완전한 예제입니다. 서울, 부산, 제주의 날씨 데이터를 제공하며, MCP의 3가지 핵심 기능을 모두 구현합니다.

📦 MCP 서버가 제공하는 3가지 핵심 기능

🔧 Tools (3개)	LLM이 호출하는 동적 함수= @tool 데코레이터와 유사	get_current_weather: 현재 날씨get_weather_forecast: 3일 예보 compare_cities: 도시 비교
📦 Resources (5개)	URI로 접근하는 정적/동적 데이터= 파일, DB 레코드처럼	weather://cities/all: 전체 데이터weather://cities/seoul: 서울 데이터 weather://stats/summary: 통계 요약
💬 Prompts (3개)	재사용 가능한 프롬프트 템플릿= 미리 만든 질문 양식	analyze_weather: 날씨 분석compare_weather: 도시 비교 weather_report: 전체 리포트



3.4.1. 더미 MCP 서버 직접 개발하기 🛠️

Tool, Resource, Prompt를 모두 포함한 완전한 MCP 서버

실제 API 없이 로컬에서 동작하는 완전한 MCP 서버를 직접 만들어보겠습니다. `dummy_weather_server.py` 는 서울, 부산, 제주의 날씨 데이터를 제공하며, **MCP의 3가지 핵심 기능(Tools, Resources, Prompts)을 모두 구현**합니다.

🔧 1. Tools (3개) - 동적 함수 실행

get_current_weather	특정 도시의 현재 날씨 정보 조회	city: "seoul", "busan", "jeju"
get_weather_forecast	3일 날씨 예보 조회	city: "seoul", "busan", "jeju"
compare_cities	여러 도시의 날씨 비교	cities: 쉼표로 구분 (예: "seoul,busan,jeju")

📦 2. Resources (5개) - URI 기반 데이터 접근

weather://cities/all	전체 도시 날씨 데이터	JSON
weather://cities/seoul	서울 날씨 상세 정보	JSON
weather://stats/summary	전체 도시 통계 요약	텍스트

💬 3. Prompts (3개) - 프롬프트 템플릿

analyze_weather	특정 도시 날씨 분석 및 여행 추천	city
compare_weather	여러 도시 날씨 비교 및 순위	cities
weather_report	전체 도시 날씨 리포트	없음

1️⃣ 서버 클래스 초기화

```
class DummyWeatherServer:
    """더미 날씨 정보를 제공하는 MCP 서버"""

    def __init__(self):
        # MCP 서버 객체 생성
        self.server = Server("dummy-weather-server")
        # 핸들러 등록
        self._setup_handlers()
```

2️⃣ 도구 목록 핸들러

```
def _setup_handlers(self):
        """MCP 서버 핸들러 설정"""

        @self.server.list_tools()
        async def list_tools() -> list[Tool]:
            """사용 가능한 도구 목록 반환"""
            return [
                Tool(
                    name="get_current_weather",
                    description="특정 도시의 현재 날씨 정보를 조회합니다. 사용 가능한 도시: se
                    inputSchema={
                        "type": "object",
                        "properties": {
                            "city": {
                                "type": "string",
                                "description": "조회할 도시 (seoul, busan, jeju)",
                                "enum": ["seoul", "busan", "jeju"]
                            }
                        },
                        "required": ["city"]
                    }
                ),
                Tool(name="get_weather_forecast", ...),
                Tool(name="compare_cities", ...)
            ]
```

3️⃣ Resources 핸들러 (URI 기반 데이터 접근)

```
@self.server.list_resources()
        async def list_resources() -> list[Resource]:
            """사용 가능한 리소스 목록 반환"""
            return [
                Resource(
                    uri="weather://cities/all",
                    name="전체 도시 날씨 데이터",
                    description="서울, 부산, 제주의 전체 날씨 정보 (JSON 형식)",
                    mimeType="application/json"
                ),
                Resource(uri="weather://cities/seoul", ...),
                Resource(uri="weather://stats/summary", ...)
            ]

        @self.server.read_resource()
        async def read_resource(uri: str) -> str:
            """리소스 읽기"""
            uri_str = str(uri)  # AnyUrl 객체를 문자열로 변환

            if uri_str == "weather://cities/all":
                return json.dumps(WEATHER_DATA, ensure_ascii=False, indent=2)
            elif uri_str.startswith("weather://cities/"):
                city = uri_str.split("/")[-1]
                return json.dumps(WEATHER_DATA[city], ensure_ascii=False)
            # ...
```

4️⃣ Prompts 핸들러 (프롬프트 템플릿)

```
@self.server.list_prompts()
        async def list_prompts() -> list[Prompt]:
            """사용 가능한 프롬프트 목록 반환"""
            return [
                Prompt(
                    name="analyze_weather",
                    description="특정 도시의 날씨를 분석하고 여행 추천을 제공합니다",
                    arguments=[
                        {"name": "city", "description": "분석할 도시", "required": Tr
                    ]
                ),
                Prompt(name="compare_weather", ...),
                Prompt(name="weather_report", ...)
            ]

        @self.server.get_prompt()
        async def get_prompt(name: str, arguments: dict) -> GetPromptResult:
            """프롬프트 템플릿 반환"""
            if name == "analyze_weather":
                city = arguments.get("city", "seoul").lower()
                data = WEATHER_DATA[city]
                prompt_text = f"다음 날씨 정보를 바탕으로 {data['city']} 여행에 대한 분석..

                return GetPromptResult(
                    messages=[
                        PromptMessage(
                            role="user",
                            content=TextContent(type="text", text=prompt_text)
                        )
                    ]
                )
```

3.4.2. 더미 MCP 서버 구현 - 도구 실행

call_tool 핸들러와 Stdio 실행

3️⃣ 도구 실행 핸들러

```
@self.server.call_tool()
        async def call_tool(name: str, arguments: dict) -> list[TextContent]:
            """도구 실행"""

            if name == "get_current_weather":
                city = arguments.get("city", "seoul").lower()

                if city not in WEATHER_DATA:
                    return [TextContent(
                        type="text",
                        text=f"오류: '{city}'는 지원하지 않는 도시입니다."
                    )]

                data = WEATHER_DATA[city]
                result = f"""
📍 {data['city']} 현재 날씨

🌡️  온도: {data['temperature']}°C
☁️  날씨: {data['condition']}
💧 습도: {data['humidity']}%
💨 풍속: {data['wind_speed']}m/s

🕐 조회 시각: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}
"""
                return [TextContent(type="text", text=result.strip())]

            elif name == "get_weather_forecast":
                # 3일 예보 로직...
            elif name == "compare_cities":
                # 도시 비교 로직...
            else:
                return [TextContent(type="text", text=f"오류: 알 수 없는 도구 '{name}'
```

4️⃣ Stdio Transport로 서버 실행

```
async def run(self):
        """서버 실행 - stdio transport 사용"""
        async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
            await self.server.run(
                read_stream,
                write_stream,
                self.server.create_initialization_options()
            )

async def main():
    """메인 함수"""
    import sys
    # ⚠️ 중요: stdout은 JSON-RPC 전용, 로그는 stderr로!
    print("🌤️  더미 날씨 MCP 서버 시작...", file=sys.stderr, flush=True)
    print("📡 Stdio 모드로 실행 중", file=sys.stderr, flush=True)
    print("", file=sys.stderr, flush=True)
    print("✅ 제공 기능:", file=sys.stderr, flush=True)
    print("   🔧 Tools: 3개 (날씨 조회, 예보, 비교)", file=sys.stderr, flush=True)
    print("   📦 Resources: 5개 (도시별 데이터, 통계)", file=sys.stderr, flush=True)
    print("   💬 Prompts: 3개 (분석, 비교, 리포트)", file=sys.stderr, flush=True)

    server = DummyWeatherServer()
    await server.run()

if __name__ == "__main__":
    asyncio.run(main())
```

⚠️ 핵심 주의사항:

MCP Stdio 프로토콜에서는 `stdout` 이 JSON-RPC 메시지 전용입니다. 모든 로그는 반드시 `file=sys.stderr` 로 출력해야 합니다!

3.4.3. MCP 통합 방법: MCPServerAdapter

Context Manager 패턴으로 MCP 서버 연결하기

> MCP 서버를 직접 구현하고 수동으로 래핑하는 대신, `MCPServerAdapter` 를 사용하면
>
> **10-20줄의 코드**
>
> 로 650+ MCP 서버를 즉시 사용할 수 있습니다!

📦 필수 설치

```
pip install 'crewai-tools[mcp]' mcp crewai langchain-openai
```

🔑 핵심 구조 (5단계)

```
# 1️⃣ Import 및 환경 설정
from crewai_tools import MCPServerAdapter
from mcp import StdioServerParameters
from crewai import Agent, Task, Crew, Process
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()  # .env에서 API 키 로드

# 2️⃣ 서버 파라미터 설정
server_params = StdioServerParameters(
    command="python3",
    args=["dummy_weather_server.py"],
    env=os.environ
)

# 3️⃣ Context Manager 연결
with MCPServerAdapter(server_params, connect_timeout=30) as mcp_tools:
    print(f"✅ {len(mcp_tools)}개 도구")

    # 4️⃣ Agent 생성
    weather_analyst = Agent(
        role="날씨 분석 전문가",
        goal="MCP 도구로 날씨 분석",
        tools=mcp_tools,  # ← MCP 도구 할당
        llm=ChatOpenAI(model="gpt-4o-mini")
    )

    # 5️⃣ Task 및 Crew 실행 (Async)
    task = Task(
        description="서울, 부산, 제주 세 도시의 현재 날씨를 조회하고 비교 분석하세요.",
        expected_output="세 도시의 날씨 비교 분석",
        agent=weather_analyst
    )

    crew = Crew(
        agents=[weather_analyst],
        tasks=[task],
        process=Process.sequential
    )

    result = await crew.kickoff_async()  # ⚠️ Async 필수!
```

> ⚠️ **핵심 주의사항**
>
> **1. Context Manager 필수**
>
> with MCPServerAdapter(...) as mcp_tools:
>
> ← 자동 연결 관리
>
> **2. Async 실행**
>
> await crew.kickoff_async()
>
> ← MCP는 비동기 필수
>
> **3. connect_timeout 설정**
>
> 느린 서버는 60초 이상 권장 (기본값: 30초)

3.4.4. MCPServerAdapter 실전 예제

Tool, Resource, Prompt 세 가지 기능 모두 시연

> `mcp_server_integration.py` 는 MCP의 3가지 핵심 기능을 모두 시연합니다:
>
> - **예제 1 (Tools):**
>
> MCPServerAdapter로 CrewAI Agent에 통합
> - **예제 2 (Resources):**
>
> MCP Client로 URI 기반 데이터 읽기
> - **예제 3 (Prompts):**
>
> 프롬프트 템플릿 조회 및 실행

📘 예제 1: Tools - CrewAI Agent 통합

MCPServerAdapter를 사용하여 MCP Tools를 CrewAI Agent에 통합합니다. Agent가 MCP 도구를 사용하여 Task를 수행합니다.

1️⃣ 서버 파일 경로 확인

```
from pathlib import Path

# 더미 서버 파일 경로 확인
server_path = Path(__file__).parent / "dummy_weather_server.py"
if not server_path.exists():
    print(f"⚠️  더미 서버 파일을 찾을 수 없습니다: {server_path}")
    return

print(f"✅ 더미 MCP 서버 발견: {server_path.name}")
```

2️⃣ Stdio 서버 파라미터 설정

```
from mcp import StdioServerParameters

server_params = StdioServerParameters(
    command="python3",              # Python 인터프리터
    args=[str(server_path)],        # 서버 스크립트 경로
    env=os.environ                  # 환경 변수 전달
)
```

3️⃣ Context Manager로 연결 및 Agent 생성

```
with MCPServerAdapter(server_params, connect_timeout=30) as mcp_tools:
    print(f"✅ {len(mcp_tools)}개의 도구 로드 완료\n")

    # 도구 목록 출력
    for i, tool in enumerate(mcp_tools, 1):
        tool_name = tool.name if hasattr(tool, 'name') else str(tool)
        print(f"   {i}. {tool_name}")

    # Agent 생성
    weather_analyst = Agent(
        role="날씨 분석 전문가",
        goal="MCP 도구를 사용하여 한국 주요 도시의 날씨를 분석합니다",
        backstory="10년 경력의 기상 데이터 분석 전문가",
        tools=mcp_tools,  # ← MCP 도구 할당
        llm=ChatOpenAI(model="gpt-4o-mini"),
        verbose=True
    )
```

4️⃣ Task 및 Crew 실행

```
# Task 생성
    task1 = Task(
        description="서울, 부산, 제주 세 도시의 현재 날씨를 조회하고 비교 분석하세요.",
        expected_output="세 도시의 날씨 비교 분석 (온도, 날씨 상태, 특징)",
        agent=weather_analyst
    )

    # Crew 생성 및 실행
    crew = Crew(
        agents=[weather_analyst],
        tasks=[task1],
        process=Process.sequential,
        verbose=True
    )

    result = await crew.kickoff_async()  # ⚠️ 비동기 실행 필수!

    print("\n✅ 작업 완료!")
    print(f"📊 결과:\n{result}")
```

📘 예제 2: Resources - URI 기반 데이터 읽기

MCP Client를 사용하여 Resources에 직접 접근합니다.

```
from mcp.client.stdio import stdio_client, StdioServerParameters as ClientParams
from mcp import ClientSession

# Stdio 클라이언트로 서버 연결
async with stdio_client(ClientParams(
    command="python3",
    args=[str(server_path)],
    env=os.environ
)) as (read, write):
    async with ClientSession(read, write) as session:
        # 초기화
        await session.initialize()
```

```
# 리소스 목록 조회
        resources = await session.list_resources()

        for i, resource in enumerate(resources.resources, 1):
            print(f"   {i}. {resource.name}")
            print(f"      URI: {resource.uri}")
            print(f"      설명: {resource.description}")

        # 리소스 읽기 예제
        # 1. 전체 도시 데이터
        result = await session.read_resource(uri="weather://cities/all")
        print(result.contents[0].text)  # JSON 데이터

        # 2. 특정 도시 데이터
        result = await session.read_resource(uri="weather://cities/seoul")
        print(result.contents[0].text)

        # 3. 통계 요약
        result = await session.read_resource(uri="weather://stats/summary")
        print(result.contents[0].text)
```

💬 예제 3: Prompts - 프롬프트 템플릿 실행

MCP Client를 사용하여 Prompts를 조회하고 실행합니다.

```
async with ClientSession(read, write) as session:
    await session.initialize()

    # 프롬프트 목록 조회
    prompts = await session.list_prompts()

    for i, prompt in enumerate(prompts.prompts, 1):
        print(f"   {i}. {prompt.name}")
        print(f"      설명: {prompt.description}")
        if prompt.arguments:
            args_str = ", ".join([f"{arg.name}{'*' if arg.required else ''}"
                                  for arg in prompt.arguments])
            print(f"      파라미터: {args_str}")
```

```
# 1. 날씨 분석 프롬프트
    result = await session.get_prompt(
        name="analyze_weather",
        arguments={"city": "seoul"}
    )
    message = result.messages[0]
    print(f"Role: {message.role}")
    print(f"Content:\n{message.content.text}")

    # 2. 날씨 비교 프롬프트
    result = await session.get_prompt(
        name="compare_weather",
        arguments={"cities": "seoul,busan,jeju"}
    )
    message = result.messages[0]
    print(f"Content:\n{message.content.text}")
```

3.4.5. OpenAI Tool Calling vs MCP: 완전 비교

언제 무엇을 사용해야 할까?

3.3에서 배운 **OpenAI Tool Calling**(@tool 데코레이터)와 3.4의 **MCP**는 모두 에이전트에 도구를 제공하는 방식이지만, 접근 방법과 사용 시나리오가 크게 다릅니다.

📊 핵심 비교

통신 방식	직접 함수 호출	서버-클라이언트 (JSON-RPC)
상태 관리	Stateless (매 요청마다 독립)	Stateful (세션 유지)
기능 범위	Tools (도구만)	Tools + Resources + Prompts
구현 복잡도	⭐ 간단 (@tool 데코레이터)	⭐⭐⭐ 복잡 (서버 구축 필요)
데이터 공유	함수 파라미터로 전달	Resources로 대용량 데이터 공유
표준화	OpenAI 전용	여러 LLM 제공자 호환
애플리케이션 간 공유	불가능 (각 앱마다 구현 필요)	가능 (서버 한 번 구축)

🎯 언제 어떤 방식을 선택할까?

🔧 코드 구조 비교

🚀 실전 시나리오별 선택 가이드

📈 성능 비교

초기 구축 시간	⚡ 5-10분	⏱️ 1-2시간
실행 오버헤드	⚡ 거의 없음	⏱️ JSON-RPC 통신
확장성 (10+ 도구)	⚠️ 관리 어려움	✅ 구조화 용이
여러 앱 공유	❌ 불가능	✅ 가능

3.4.6. Tool 할당 방식 비교

Agent Tools vs Task Tools vs MCP Tools

CrewAI에서 도구를 에이전트에게 제공하는 방식은 크게 3가지가 있습니다. 각 방식의 특징과 사용 시나리오를 이해하는 것이 중요합니다.

1️⃣ 세 가지 Tool 할당 방식

Agent Tools	Agent(tools=[...])	에이전트의 모든 작업에서 사용 가능	에이전트의 전반적인 능력 (검색, 계산 등)
Task Tools	Task(tools=[...])	해당 Task 수행 시에만 사용 가능	특정 작업에만 필요한 특수 도구
MCP Tools	MCP 서버에서 동적 로드	런타임에 동적으로 할당	외부 시스템 통합, 서버 기반 도구

2️⃣ 코드 예시 비교

3️⃣ Tool 우선순위와 병합 규칙

> 🔍 **Task 실행 시 사용 가능한 도구**

4️⃣ 실전 선택 가이드

3.5. 고급 협업 프로세스 (multi_agent_collaboration.py)

순차(Sequential) vs 계층(Hierarchical) 프로세스 완전 정복

CrewAI는 두 가지 핵심 협업 프로세스를 제공합니다. 워크플로우의 복잡성에 따라 적절한 방식을 선택해야 합니다.

Sequential	정의된 순서대로 작업 실행	예측 가능, 간단, 디버깅 용이	유연성 부족, 동적 대응 불가	조립 라인 같은 선형적 워크플로우
Hierarchical	매니저 에이전트가 작업 조율/위임	유연함, 동적 작업 결정, 복잡한 문제 해결	더 많은 LLM 호출(비용/시간 ↑), 복잡함	인간 팀처럼 유동적인 프로젝트

3.5.1. 클래스 기반 시스템 설계

전문가 팀을 체계적으로 관리하기

프로젝트가 복잡해지면, 관련된 에이전트, 작업, 크루를 하나의 클래스(`MultiAgentCollaborationSystem`)로 묶어 관리하는 것이 효과적입니다. 이 설계 패턴은 코드의 구조를 명확하게 하고 재사용성을 높입니다.

1️⃣ 클래스 구조: Dictionary로 관리하기

```
class MultiAgentCollaborationSystem:
    def __init__(self, llm_model: str = "gpt-4"):
        """다중 에이전트 협업 시스템을 초기화합니다"""
        self.llm_model = llm_model
        self.llm = None

        # 📦 Dictionary로 에이전트/작업/크루를 관리 (키-값 구조)
        self.agents = {}   # {'project_manager': Agent(...), 'researcher': Agent(.
        self.tasks = {}    # {'research': Task(...), 'analysis': Task(...), ...}
        self.crews = {}    # {'sequential_collaboration': Crew(...), ...}

        # 초기화 메서드를 순서대로 실행
        self._setup_llm()           # LLM 설정
        self._create_agents()       # 6개 전문 에이전트 생성 (모두 사용)
        self._create_tasks()        # 각 에이전트의 작업 정의
        self._create_crews()        # Sequential/Hierarchical 크루 구성
```

2️⃣ 6개 전문 에이전트 생성하기 (모두 사용)

```
def _create_agents(self):
    """프로젝트 수행에 필요한 6명의 전문 에이전트를 생성합니다"""

    # 매니저 에이전트 (계층형 프로세스에서 팀 조율) ✅ Hierarchical
    self.agents['project_manager'] = Agent(
        role='프로젝트 매니저',
        goal='복잡한 다중 에이전트 프로젝트를 조율하고 관리한다',
        backstory='여러 전문 팀이 관련된 복잡한 프로젝트를 조율하는 전문가입니다.',
        llm=self.llm,
        allow_delegation=True  # 필수: 매니저는 작업을 위임할 수 있어야 함
    )

    # 작업자 에이전트 5명
    self.agents['researcher'] = Agent(role='연구 전문가', ...)  # ✅ Both
    self.agents['analyst'] = Agent(role='데이터 분석 전문가', ...)  # ✅ Both
    self.agents['content_creator'] = Agent(role='콘텐츠 생성 전문가', ...)  # ✅ Both
    self.agents['quality_assurance'] = Agent(role='품질 보증 전문가', ...)  # ✅ Hiera
    self.agents['technical_expert'] = Agent(role='기술 전문가', ...)  # ✅ Hierarchic
```

3️⃣ 협업 실행 메서드

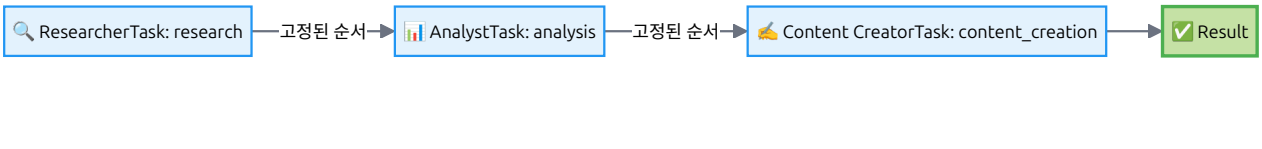
```
def execute_sequential_collaboration(self, topic: str) -> Dict[str, Any]:
    """순차적 협업을 실행합니다"""
    print(f"\n🔄 순차적 협업 실행: {topic}")
    result = self.crews['sequential_collaboration'].kickoff(inputs={'topic': topic}
    return {'collaboration_type': 'Sequential', 'topic': topic, 'result': result}

def execute_hierarchical_collaboration(self, topic: str) -> Dict[str, Any]:
    """계층적 협업을 실행합니다"""
    print(f"\n🏗️ 계층적 협업 실행: {topic}")
    result = self.crews['hierarchical_collaboration'].kickoff(inputs={'topic': topi
    return {'collaboration_type': 'Hierarchical', 'topic': topic, 'result': result}

# 💡 사용 예시
system = MultiAgentCollaborationSystem()
system.execute_sequential_collaboration("2024년 국내 원격근무 시장 동향 분석")
```

3.5.2. 순차 vs 계층 프로세스 비교

실제 코드로 배우는 두 가지 협업 방식

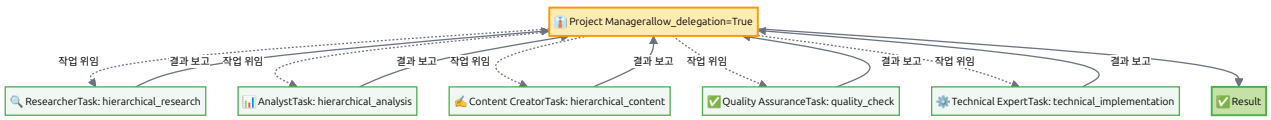
동일한 에이전트와 작업을 사용하지만, `process` 와 `manager_agent` 설정만 다른 두 개의 크루를 만들어 차이점을 비교해봅니다.

1️⃣ 순차적(Sequential) 프로세스 - 3명의 전문가




```
# 작업이 정의된 순서(연구 → 분석 → 콘텐츠 생성)대로 엄격하게 실행
self.crews['sequential_collaboration'] = Crew(
    agents=[
        self.agents['researcher'],      # 1. 연구 전문가
        self.agents['analyst'],         # 2. 데이터 분석 전문가
        self.agents['content_creator']  # 3. 콘텐츠 생성 전문가
    ],
    tasks=[self.tasks['research'], self.tasks['analysis'], self.tasks['content_crea
    process=Process.sequential,
    verbose=True,
    max_iter=10  # ⚠️ 중요: 최대 반복 횟수 제한 (무한 루프 방지)
)
```

2️⃣ 계층적(Hierarchical) 프로세스 - 5명의 전문가 + 매니저



커스텀 매니저를 직접 정의하여 더 많은 제어권을 가집니다. 이 예제에서는 **QA 전문가와 기술 전문가가 추가**됩니다.

```
# 작업자 에이전트 리스트 (매니저 제외, 5명으로 확장)
worker_agents = [
    self.agents['researcher'],          # 1. 연구 전문가
    self.agents['analyst'],             # 2. 데이터 분석 전문가
    self.agents['content_creator'],     # 3. 콘텐츠 생성 전문가
    self.agents['quality_assurance'],   # 4. 품질 보증 전문가
    self.agents['technical_expert']     # 5. 기술 전문가
]

# 계층적 프로세스 전용 작업 (매니저 작업은 제외, 5개 작업)
hierarchical_tasks = [
    self.tasks['hierarchical_research'],
    self.tasks['hierarchical_analysis'],
    self.tasks['hierarchical_content'],
    self.tasks['quality_check'],
    self.tasks['technical_implementation']
]

self.crews['hierarchical_collaboration'] = Crew(
    agents=worker_agents,                           # 5명의 작업자
    tasks=hierarchical_tasks,                       # 5개의 작업
    process=Process.hierarchical,
    manager_agent=self.agents['project_manager'],  # 커스텀 매니저 지정
    verbose=True,
    max_iter=10
)
```

CrewAI가 자동으로 매니저를 생성합니다. 대부분의 경우 이 방식으로 충분합니다.

```
# 주석 해제하고 manager_agent 주석 처리하면 사용 가능
hierarchical_crew = Crew(
    agents=worker_agents,  # 작업자만 포함
    tasks=hierarchical_tasks,
    process=Process.hierarchical,
    manager_llm="gpt-4"  # 매니저 역할을 수행할 LLM 지정 (자동 생성)
)
```

3.6. 워크플로우 오케스트레이션 (workflow_orchestration.py)

복잡한 프로세스의 재사용 및 관리

프로젝트가 복잡해지면 여러 에이전트와 작업으로 구성된 파이프라인을 '워크플로우' 단위로 정의하고 재사용하는 패턴이 유용합니다. `WorkflowOrchestrator` 클래스는 이 패턴을 구현한 예제입니다.

딕셔너리 기반 클래스 구조

재사용할 수 있는 에이전트, 작업, 워크플로우를 딕셔너리로 관리하고, 실행 이력을 리스트로 추적합니다.

```
# 02_advanced/workflow_orchestration.py
class WorkflowOrchestrator:
    def __init__(self, llm_model: str = "gpt-4"):
        self.llm_model = llm_model
        self.llm = None
        self.agents = {}              # 5개 에이전트 저장소
        self.tasks = {}               # 5개 작업 저장소
        self.workflows = {}           # 2개 워크플로우 레시피 저장소
        self.execution_history = []   # 실행 이력 추적

        self._setup_llm()
        self._create_agents()
        self._create_tasks()
        self._define_workflows()

    def _create_agents(self):
        """5명의 전문 에이전트 생성"""
        self.agents['workflow_manager'] = Agent(
            role='워크플로우 관리자',
            goal='데이터 처리 워크플로우의 전반적인 계획과 조율을 담당합니다',
            backstory='...',
            llm=self.llm
        )
        self.agents['data_collector'] = Agent(
            role='데이터 수집 전문가',
            goal='다양한 소스에서 데이터를 수집하고 검증합니다',
            backstory='...',
            llm=self.llm
        )
        self.agents['data_processor'] = Agent(
            role='데이터 처리 전문가',
            goal='수집된 데이터를 정제하고 분석 가능한 형태로 변환합니다',
            backstory='...',
            llm=self.llm
        )
        self.agents['report_generator'] = Agent(
            role='보고서 생성 전문가',
            goal='분석 결과를 명확하고 설득력 있는 보고서로 작성합니다',
            backstory='...',
            llm=self.llm
        )
        self.agents['quality_assurance'] = Agent(
            role='품질 보증 전문가',
            goal='최종 결과물의 품질을 검증하고 개선 사항을 제안합니다',
```

```
            backstory='...',
            llm=self.llm
        )

    def _create_tasks(self):
        """5개의 작업 생성 (각 에이전트에 대응)"""
        self.tasks['workflow_planning'] = Task(
            description='...',
            expected_output='...',
            agent=self.agents['workflow_manager']
        )
        self.tasks['data_collection'] = Task(
            description='...',
            expected_output='...',
            agent=self.agents['data_collector']
        )
        # ... 나머지 3개 작업 ...
```

3.6.1. 워크플로우 '레시피' 정의

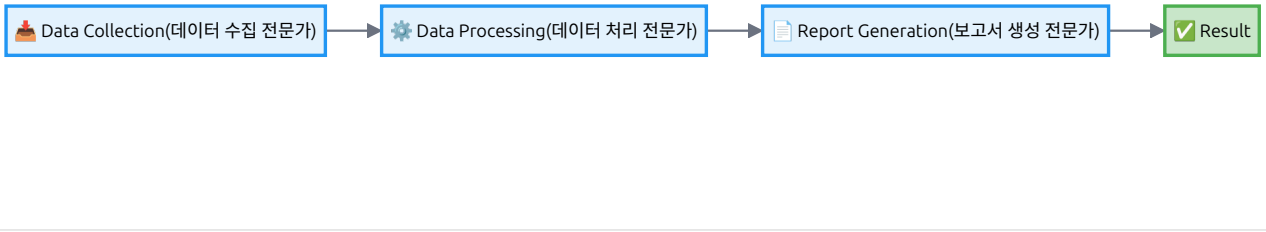
실습: workflow_orchestration.py

미리 만들어 둔 에이전트와 작업을 조합하여, 다양한 워크플로우 '레시피'를 딕셔너리 형태로 정의합니다.

```
# 02_advanced/workflow_orchestration.py
def _define_workflows(self):
    # 기본 워크플로우: 수집 → 처리 → 보고
    self.workflows['basic_workflow'] = {
        'name': '기본 데이터 처리 워크플로우',
        'description': '데이터 수집, 처리, 보고서 생성을 위한 간단한 순차 워크플로우',
        'steps': ['data_collection', 'data_processing', 'report_generation'],
        'process': Process.sequential
    }

    # 고급 워크플로우: 계획 → 수집 → 처리 → 보고 → 품질 보증
    self.workflows['advanced_workflow'] = {
        'name': '고급 데이터 분석 워크플로우',
        'description': '계획, 실행, 품질 보증을 포함한 포괄적인 워크플로우',
        'steps': ['workflow_planning', 'data_collection', 'data_processing',
                  'report_generation', 'quality_assurance'],
        'process': Process.sequential
    }
```

📊 워크플로우 시각화

두 가지 워크플로우를 다이어그램으로 비교해보겠습니다:



> 🔍 **워크플로우 비교**

💡 **핵심:** 이 방식은 코드 변경 없이 새로운 워크플로우를 쉽게 추가하거나 수정할 수 있게 해줍니다. 필요한 단계(steps)만 선택하여 다양한 조합의 워크플로우를 만들 수 있습니다.

3.6.2. 동적 워크플로우 실행

실습: workflow_orchestration.py

`execute_workflow` 메소드는 워크플로우 이름을 받아, 해당 레시피에 따라 필요한 에이전트와 작업만으로 **실시간으로 새로운 Crew를 생성**하여 실행합니다. 실행 시간을 측정하고 이력을 자동으로 추적합니다.

```
# 02_advanced/workflow_orchestration.py
def execute_workflow(self, workflow_name: str, inputs: Dict[str, Any]) -> Dict[str,
    """워크플로우를 실행하고 실행 이력을 추적합니다."""
    workflow = self.workflows[workflow_name]

    # 1. 워크플로우 정의에 따라 필요한 에이전트와 작업 선택
    workflow_agents = [self.tasks[step].agent for step in workflow['steps']]
    workflow_tasks = [self.tasks[step] for step in workflow['steps']]

    # 2. 동적으로 Crew 생성
    crew = Crew(
        agents=workflow_agents,
        tasks=workflow_tasks,
        process=workflow['process'],
        verbose=True
    )

    # 3. 실행 시간 측정 시작
    start_time = time.time()

    # 4. 생성된 Crew 실행
    try:
        result = crew.kickoff(inputs=inputs)
        status = 'completed'
    except Exception as e:
        result = str(e)
        status = 'failed'

    # 5. 실행 시간 계산
    duration = time.time() - start_time

    # 6. 실행 이력 추적 (자동으로 저장)
    self.execution_history.append({
        'workflow_name': workflow_name,
        'start_time': start_time,
        'duration': duration,
        'status': status,
        'inputs': inputs
    })

    # 7. 메타데이터를 포함한 결과 반환
    return {
        'workflow_name': workflow_name,
        'status': status,
```

```
        'duration': duration,
        'result': result,
        'execution_count': len(self.execution_history)
    }

# --- 실행 예시 ---
orchestrator = WorkflowOrchestrator()

# 기본 워크플로우 실행 (3단계)
basic_result = orchestrator.execute_workflow(
    'basic_workflow',
    inputs={'topic': '2024년 AI 트렌드'}
)
print(f"✅ {basic_result['workflow_name']} 완료 ({basic_result['duration']:.2f}초)")

# 고급 워크플로우 실행 (5단계)
advanced_result = orchestrator.execute_workflow(
    'advanced_workflow',
    inputs={'topic': '생성형 AI 시장 분석'}
)
print(f"✅ {advanced_result['workflow_name']} 완료 ({advanced_result['duration']:.2

# 실행 이력 확인
print(f"\\n📊 총 {len(orchestrator.execution_history)}개 워크플로우 실행됨")
for history in orchestrator.execution_history:
    print(f"  - {history['workflow_name']}: {history['status']} ({history['duratio
```

- **동적 Crew 생성**

  : 워크플로우마다 필요한 에이전트만 선택하여 효율적인 리소스 사용

- **자동 시간 측정**

  : 각 워크플로우의 실행 시간을 자동으로 측정하여 성능 분석 가능

- **실행 이력 추적**

  : 모든 워크플로우 실행 기록을 저장하여 디버깅과 최적화에 활용

- **에러 처리**

  : try-except로 실패한 워크플로우도 추적하여 안정성 향상

- **메타데이터 반환**

  : 결과뿐 아니라 실행 정보도 함께 반환하여 투명성 제공

이 패턴은 복잡한 에이전트 시스템을 매우 유연하고 확장 가능하게 만들어주는 강력한 설계 방식입니다. 프로덕션 환경에서 여러 워크플로우를 관리할 때 특히 유용합니다.

3.6.3. Crew 구성: Agent-Task 매핑과 Process 기준

Crew 구성 시 이해해야 할 핵심 규칙

Crew를 구성할 때 Agent와 Task의 관계, 그리고 Process가 무엇을 기준으로 실행되는지 이해하는 것이 매우 중요합니다.

1️⃣ Agent와 Task의 매핑 관계

> 📌 **기본 원칙**
>
> - **Agent 수 ≠ Task 수:**
>
> Agent와 Task의 개수는 서로 다를 수 있습니다
> - **1:N 관계:**
>
> 한 Agent가 여러 Task를 수행할 수 있습니다
> - **Task.agent 필수 (Sequential):**
>
> Sequential Process에서는 모든 Task에 `agent` 속성이 필수입니다
> - **Task.agent 선택 (Hierarchical):**
>
> Hierarchical Process에서는 Task에 agent 미지정 시 Manager가 동적으로 할당합니다

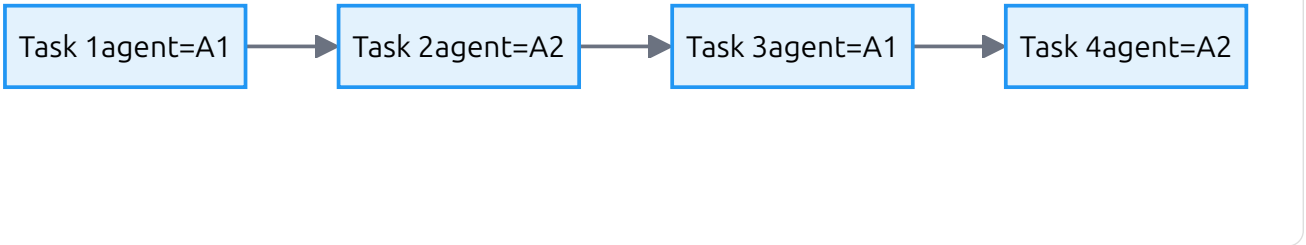
2️⃣ 매핑 패턴 예시

1:1 매핑	3명	3개 (각각 다른 agent)	✅ 각 Agent가 하나씩 담당
1:N 매핑	2명	5개 (일부 Task가 같은 agent)	✅ 한 Agent가 여러 Task 수행
Agent 미사용	5명 정의	3개 (3명의 agent만 사용)	✅ 나머지 2명은 대기 (문제없음)
Agent 미지정 (Sequential)	3명	Task에 agent 미지정	❌ 에러 발생
Agent 미지정 (Hierarchical)	Manager + Workers	Task에 agent 미지정	✅ Manager가 동적 할당

3️⃣ 코드 예시: 다양한 매핑 패턴

4️⃣ Process 실행 기준: Task 기준 vs Agent 기준

5️⃣ 실행 흐름 비교 다이어그램



3.7. 비동기 실행과 AI 기반 계획 (async_and_planning_example.py)

CrewAI는 2024년 대규모 업데이트를 통해 프로덕션 환경에 필수적인 고급 기능들을 추가했습니다.

클래스 구조 및 구성 요소

```
# 1. 표준 크루 (비동기 실행 및 배치 처리용)
self.crews['standard_crew'] = Crew(
    agents=[market_researcher, trend_analyst, report_writer],
    tasks=[market_research, trend_analysis, report_writing],
    process=Process.sequential,
    max_iter=10
)

# 2. AI 계획 기반 크루 (자동 작업 최적화)
self.crews['planning_crew'] = Crew(
    agents=[market_researcher, trend_analyst, competitor_analyst, report_writer],
    tasks=[market_research, trend_analysis, competitor_analysis, report_writing],
    planning=True,  # AI가 작업 순서 자동 최적화
    process=Process.sequential,
    max_iter=10
)
```

3.7.1. 비동기 단일/병렬 실행

실습: async_and_planning_example.py

```
# 02_advanced/async_and_planning_example.py
async def execute_async_single(self, topic: str):
    """단일 크루를 비동기로 실행"""
    start_time = time.time()

    result = await self.crews['standard_crew'].kickoff_async(
        inputs={'topic': topic}
    )

    duration = time.time() - start_time
    return {'topic': topic, 'duration': duration, 'result': result}

# 실행
asyncio.run(system.execute_async_single("2024년 생성형 AI 시장"))
```

```
async def execute_async_multiple(self, topics: List[str]):
    """여러 크루를 동시에 비동기 실행"""

    # asyncio.gather로 병렬 실행
    tasks = [
        self.crews['standard_crew'].kickoff_async(inputs={'topic': topic})
        for topic in topics
    ]

    results = await asyncio.gather(*tasks, return_exceptions=True)
    return results

# 실행 예시
topics = ["클라우드 컴퓨팅", "사이버 보안", "블록체인"]
results = await system.execute_async_multiple(topics)

# ✅ 병렬 실행으로 3배 시간 절약!
# 순차 실행: 30분 → 비동기 병렬: 12분
```

- **병렬 처리**

  : 여러 크루가 동시에 실행되어 대기 시간 최소화

- **에러 처리**

  : return_exceptions=True로 일부 실패 시에도 나머지 계속 실행

- **리소스 효율**

  : I/O 대기 시간을 활용하여 CPU 유휴 시간 감소

3.7.2. 배치 처리

실습: async_and_planning_example.py

동일한 워크플로우를 여러 입력에 대해 순차적으로 반복 실행합니다.

```
# 02_advanced/async_and_planning_example.py
def execute_batch_sync(self, topics: List[str]):
    """여러 입력에 대해 순차적으로 배치 처리"""

    inputs_list = [{'topic': topic} for topic in topics]
    results = self.crews['standard_crew'].kickoff_for_each(inputs=inputs_list)

    return results

# 실행 예시
batch_topics = ["IoT", "5G", "메타버스"]
results = system.execute_batch_sync(batch_topics)
# 각 주제를 순서대로 처리 (안정적, 예측 가능)
```

```
async def execute_batch_async(self, topics: List[str]):
    """여러 입력에 대해 비동기로 배치 처리"""

    inputs_list = [{'topic': topic} for topic in topics]
    results = await self.crews['standard_crew'].kickoff_for_each_async(
        inputs=inputs_list
    )

    return results

# 실행 예시
results = await system.execute_batch_async(batch_topics)
# 비동기 처리로 시간 단축 효과 극대화!
```

3.7.3. AI 기반 자동 작업 계획

실습: async_and_planning_example.py

`planning=True` 를 설정하면 LLM이 작업을 분석하고 최적의 실행 순서를 자동으로 결정합니다.

```
# 02_advanced/async_and_planning_example.py
def execute_with_planning(self, topic: str):
    """AI 기반 계획 기능을 사용하여 크루 실행"""

    # AI가 작업 순서를 자동으로 최적화
    result = self.crews['planning_crew'].kickoff(inputs={'topic': topic})

    return result

# 실행 예시
result = system.execute_with_planning("메타버스 기술 동향")

# 🧠 LLM이 자동으로:
# 1. 4개 작업(시장조사, 트렌드, 경쟁사, 보고서)을 분석
# 2. 최적의 실행 순서 결정
# 3. 필요 시 작업 재정렬 또는 건너뛰기
```

1. **작업 분석**

   : LLM이 모든 Task의 description과 goal을 검토

2. **의존성 파악**

   : 작업 간 선후 관계 자동 인식

3. **순서 최적화**

   : 가장 효율적인 실행 순서 결정

4. **동적 조정**

   : 실행 중 상황에 따라 계획 수정 가능

3.8. 메모리 시스템 (memory_system_example.py)

CrewAI는 메모리 시스템을 통해 에이전트가 과거 실행 내역을 기억하고 학습할 수 있게 합니다. 에이전트의 능력을 크게 향상시키는 정교한 메모리 아키텍처를 제공합니다.

메모리 시스템 개요

💡 이 강의에서는 Basic Memory System을 다룹니다

> **CrewAI의 Basic Memory System은 4가지 컴포넌트로 구성됩니다:**

```
from crewai import Crew, Agent, Task, Process

# 메모리가 활성화된 크루
crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,  # 🔑 Short-term, Long-term, Entity memory 자동 활성화
    verbose=True
)
```

memory=True의 작동 원리 (How It Works)

`memory=True` 설정 시, CrewAI는 내부적으로 다음과 같은 초기화를 자동으로 수행합니다:

메모리 저장 위치

> **📂 플랫폼별 기본 저장 경로**
>
> **💡 Storage Location:**
>
> Platform-specific location via `appdirs` package
>
> **🔧 Custom Storage Directory:**
>
> Set `CREWAI_STORAGE_DIR` environment variable

3.8.1. 연속된 대화에서 메모리 효과

메모리가 활성화된 크루는 이전 대화를 기억하고 맥락을 유지합니다.

실행 흐름

```
# 첫 번째 대화
result1 = demo.execute_with_memory("생성형 AI에 대해 알려줘")

# 두 번째 대화 - 이전 대화 참조
result2 = demo.execute_with_memory("방금 말한 것 중에서 가장 중요한 게 뭐야?")
# 💡 에이전트가 "생성형 AI"에 대한 이전 답변을 기억하고 참조

# 세 번째 대화 - 관련 주제로 확장
result3 = demo.execute_with_memory("그럼 이 기술을 비즈니스에 어떻게 활용할 수 있을까?")
# 💡 대화의 맥락을 유지하며 답변
```

핵심 메서드: execute_with_memory()

```
def execute_with_memory(self, query: str) -> Dict[str, Any]:
    """메모리 시스템이 활성화된 크루로 실행합니다."""
    self.execution_count += 1
    print(f"\n🧠 메모리 활성화 크루 실행 (#{self.execution_count})")
    print(f"쿼리: {query}")
    start_time = time.time()

    try:
        result = self.crew_with_memory.kickoff(inputs={'query': query})
        duration = time.time() - start_time

        return {
            'execution_number': self.execution_count,
            'query': query,
            'duration': duration,
            'result': result,
            'memory_enabled': True,
            'status': 'completed'
        }
    except Exception as e:
        # 에러 처리...
        return {...}
```

3.8.2. 메모리 유무 비교

동일한 질문에 대해 메모리가 있는 크루와 없는 크루의 차이를 비교합니다.

비교 실행 코드

```
comparison_query = "머신러닝의 주요 알고리즘 3가지를 설명해줘"

# 1️⃣ 메모리 활성화 크루
with_memory_result = demo.execute_with_memory(comparison_query)

# 2️⃣ 메모리 비활성화 크루
without_memory_result = demo.execute_without_memory(comparison_query)
```

결과 비교

항목	메모리 활성화 ✅	메모리 비활성화 ❌
이전 대화 참조	가능	불가능
실행 횟수 추적	누적됨 (1번, 2번, ...)	추적 안 됨
사용자 선호도	학습 및 반영	매번 초기화
성능	약간 느림 (메모리 검색)	빠름
디스크 사용	ChromaDB 저장 공간 필요	저장 안 함
적합한 용도	챗봇, 개인 비서, 연구 도우미	일회성 작업, 빠른 테스트

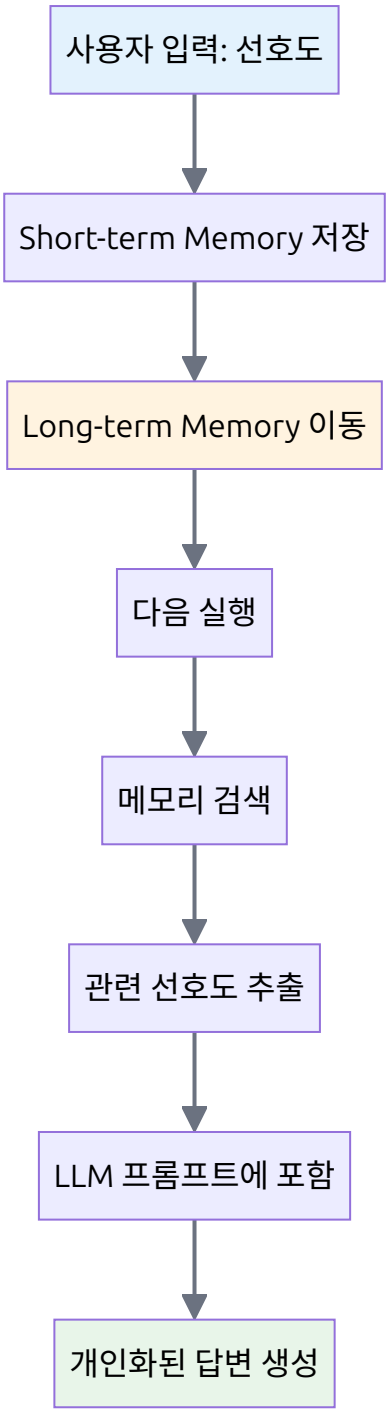
3.8.3. 사용자 선호도 학습 및 개인화

메모리 시스템을 통해 에이전트가 사용자의 선호도를 학습하고 답변에 반영합니다.

선호도 학습 과정

```
# 1단계: 사용자 선호도 표현
result4 = demo.execute_with_memory(
    "나는 실무에 바로 적용 가능한 구체적인 예제를 선호해"
)
# 💾 선호도가 장기 메모리에 저장됨

# 2단계: 새로운 질문 - 저장된 선호도가 반영됨
result5 = demo.execute_with_memory("딥러닝을 설명해줘")
# 💡 에이전트가 "실무 예제"를 포함하여 답변할 가능성 ↑
```

개인화의 작동 원리


사용자 입력: 선호도
Short-term Memory 저장
Long-term Memory 이동
다음 실행
메모리 검색
관련 선호도 추출
LLM 프롬프트에 포함
개인화된 답변 생성

실전 활용 사례

활용 분야	메모리 활용 방식	효과
챗봇/개인 비서	대화 히스토리 추적, 사용자 이름/선호도 기억	자연스러운 대화, 높은 만족도
연구 도우미	이전 연구 결과 재활용, 중복 조사 방지	시간 절약, 비용 절감
고객 서비스	고객별 상호작용 히스토리 관리	맞춤형 서비스, 문제 빠른 해결
교육 시스템	학습자별 진도, 취약점 기록	개인화 학습 경로 제공
프로젝트 관리	프로젝트 히스토리 추적, 의사결정 근거 보존	일관성 유지, 지식 손실 방지

3.9. 실전 트러블슈팅 및 모범 사례

CrewAI 에이전트 개발의 핵심 노하우

CrewAI로 에이전트를 개발할 때 마주칠 수 있는 문제들과 그 해결책을 실전 코드와 함께 상세히 알아봅니다. 프로덕션 레벨의 안정적이고 예측 가능한 에이전트 시스템을 구축하기 위한 구체적인 가이드입니다.

> **1. 무한 루프 (Infinite Loops)**
>
> 에이전트가 작업을 완료하지 못하고 계속 반복하는 문제
>
> **2. 부정확한 도구 사용**
>
> 에이전트가 도구를 잘못 선택하거나 오류를 발생시키는 문제
>
> **3. 높은 비용 및 느린 속도**
>
> API 비용이 급증하고 실행 시간이 오래 걸리는 문제
>
> **4. 품질 저하 및 환각**
>
> 에이전트가 부정확하거나 사실이 아닌 내용을 생성하는 문제

3.9.1. 이슈 1: 무한 루프 (Infinite Loops)

현상: 에이전트가 작업을 완료하지 못하고 같은 생각이나 도구 사용을 계속 반복합니다. 특히 계층적 구조에서 에이전트들이 서로에게 작업을 계속 위임하며 발생합니다.

> **시나리오:**
>
> 3개 에이전트(연구원, 분석가, 작성자)로 구성된 계층적 Crew가 "화성 탐사 계획" 작성 중 무한 루프 발생
>
> **증상:**
>
> 매니저가 작업자에게 위임 → 작업자가 "정보 부족"으로 매니저에게 재위임 → 매니저가 다시 작업자에게 위임 (반복)

원인 분석 (4가지 주요 원인)

해결 방안 (다층적 접근법)

핵심: 무한 루프를 방지하려면 **여러 전략을 동시에 적용**해야 합니다. 하나만으로는 불충분합니다. 아래 4가지 전략을 모두 구현하는 것을 권장합니다.

프로세스 유형과 작업 개수에 따라 적절한 최대 반복 횟수를 설정합니다.

```
# 순차형: 3개 작업, 작업 흐름이 명확하므로 여유있게
sequential_crew = Crew(
    agents=[researcher, analyst, writer],
    tasks=[research_task, analysis_task, writing_task],
    process=Process.sequential,
    max_iter=10  # 3개 작업, 예측 가능한 흐름
)

# 계층형: 5개 작업, 작업 증가를 고려하여 설정
hierarchical_crew = Crew(
    agents=worker_agents,  # 5명: researcher, analyst, writer, QA, technical
    tasks=hierarchical_tasks,  # 5개 작업
    process=Process.hierarchical,
    manager_agent=project_manager,
    max_iter=10  # 작업이 늘어났으므로 Sequential과 같은 값 사용
)

# 💡 팁: 작업 개수가 증가하면 max_iter도 비례하여 증가
#         - 작업 3개 → max_iter=10
#         - 작업 5개 → max_iter=10 (작업당 2회 반복 여유)
#         - 작업 8개 이상 → max_iter=15+ 고려
```

측정 가능한 완료 조건을 명시하여 에이전트가 "언제 끝낼지" 정확히 알게 합니다.

```
# ❌ 나쁜 예시: 모호하고 측정 불가능
bad_task = Task(
    description="주제에 대해 연구하세요",
    expected_output="포괄적인 연구 보고서"  # 언제 "포괄적"인가?
)

# ✅ 좋은 예시: 구체적이고 측정 가능
good_task = Task(
    description="{topic}에 대한 핵심 정보와 최신 동향을 조사하세요",
    expected_output="""3-5개의 핵심 발견사항을 포함한 연구 요약 (500자 이내, 한국어)"""
)

# ✅ 더 좋은 예시: 형식 템플릿 제공
best_task = Task(
    description="{topic}에 대한 SWOT 분석을 수행하세요",
    expected_output="""다음 형식으로 작성:
강점(Strength): 1. ___ 2. ___
약점(Weakness): 1. ___ 2. ___
기회(Opportunity): 1. ___ 2. ___
위협(Threat): 1. ___ 2. ___
각 항목 30자 이내, 총 300자, 한국어"""
)
```

매니저는 조율만, 작업자는 실행만 담당하도록 명확히 분리합니다.

```
# 매니저 에이전트
manager = Agent(
    role="프로젝트 매니저",
    goal="팀을 조율하고 관리한다",
    allow_delegation=True  # 매니저는 위임 가능
)

# 작업자 에이전트들 (매니저 제외)
worker_agents = [
    researcher,      # allow_delegation=False
    analyst,         # allow_delegation=False
    content_creator  # allow_delegation=False
]

# 계층적 프로세스 전용 작업
hierarchical_tasks = [
    research_task,
    analysis_task,
    content_task
    # 매니저 전용 작업은 포함하지 않음!
]

# Crew 생성 시 매니저를 별도로 지정
hierarchical_crew = Crew(
    agents=worker_agents,          # 매니저 포함하지 않음
    tasks=hierarchical_tasks,
    process=Process.hierarchical,
    manager_agent=manager,         # 조율만 담당
    max_iter=8
)
```

```
worker_agent = Agent(
    role="연구 전문가",
    goal="연구를 수행한다",
    allow_delegation=False  # 다른 에이전트에게 위임 금지
)
```

> **5️⃣ 상세한 로깅 활성화**
>
> 에이전트의 사고 과정을 추적하여 무한 루프의 원인을 진단합니다.
>
> **6️⃣ 타임아웃 설정 (LLM 레벨)**
>
> 개별 LLM 호출에 대한 타임아웃을 설정하여 응답 없음 상황을 방지합니다.
>
> **7️⃣ 체크포인트 기반 재개 (실험적)**
>
> 장시간 실행되는 Crew의 경우, 중간 상태를 저장하여 재시작 시 이어서 실행합니다.

> - **max_iter을 너무 높게 설정 (예: 100):**
>
>   비용 폭발 위험. 대부분의 작업은 10회 이내 완료되어야 합니다.
> - **expected_output을 너무 짧게 (예: "결과"):**
>
>   에이전트가 여전히 완료 조건을 모릅니다.
> - **모든 에이전트에 allow_delegation=True:**
>
>   순환 위임이 발생할 수 있습니다.
> - **계층적 프로세스에 매니저를 agents 리스트에 포함:**
>
>   매니저가 자기 자신에게 작업을 위임하는 문제 발생.

3.9.2. 이슈 2: 부정확한 도구 사용

현상: 에이전트가 엉뚱한 도구를 사용하거나, 필요한 인자를 잘못된 형식으로 전달하여 오류가 발생합니다.

> **시나리오:**
>
> "Samsung Electronics" 정보를 조회해야 하는데, 에이전트가 "삼성전자"로 검색하여 결과를 찾지 못함
>
> **원인:**
>
> 도구의 docstring에 "회사 이름은 영문 공식 명칭으로 입력" 명시 누락

원인 분석 (3가지 주요 원인)

해결 방안: 완벽한 도구 정의

1. Pydantic 기반 파라미터 검증

타입 검증과 에러 메시지를 자동화합니다.

2. 도구 사용 로깅 및 모니터링

에이전트의 도구 사용 패턴을 추적하여 문제를 빠르게 발견합니다.

3. 도구 실패 시 재시도 메커니즘

일시적 오류에 대한 자동 재시도를 구현합니다.

- **너무 많은 도구 제공:**

  에이전트가 혼란스러워합니다. 5-7개 이하로 제한하세요.

- **유사한 이름의 도구:**

  "search_db"와 "db_search"처럼 헷갈리는 이름은 피하세요.

- **복잡한 반환값:**

  중첩된 딕셔너리나 객체 대신 간단한 문자열 또는 JSON을 반환하세요.

- **에러를 숨기기:**

  try-except로 모든 에러를 잡아 빈 문자열을 반환하면 디버깅이 불가능합니다.

3.9.3. 이슈 3: 높은 비용 및 느린 속도

현상: 간단한 작업을 처리하는 데도 LLM API 비용이 많이 청구되고 실행 시간이 너무 오래 걸립니다.

시나리오:

3개 에이전트로 구성된 Crew가 간단한 보고서 작성에 GPT-4 호출 50회, 비용 $5 발생

개선 후:

모델 차등 할당 + 캐싱으로 호출 20회, 비용 $0.80으로 감소 (84% 절감)

원인 분석 (5가지 주요 원인)

해결 방안: 비용 최적화 전략

작업 복잡도에 따라 적절한 모델을 선택합니다.

```
from langchain_openai import ChatOpenAI

# 비용 대비 성능 매트릭스
# GPT-4 Turbo: $10/1M tokens (input), 복잡한 추론
# GPT-4o mini: $0.15/1M tokens (input), 일반적인 작업
# GPT-3.5 Turbo: $0.50/1M tokens (input), 단순 작업

# 🔥 비용 최적화 예시
data_collector_agent = Agent(
    role="데이터 수집가",
    goal="웹에서 정보를 수집합니다",
    llm=ChatOpenAI(model="gpt-4o-mini"),  # 단순 수집 작업
    tools=[search_tool]
)

analyst_agent = Agent(
    role="데이터 분석가",
    goal="수집된 데이터를 분석하고 인사이트 도출",
    llm=ChatOpenAI(model="gpt-4-turbo"),  # 복잡한 분석 필요
    tools=[analysis_tool]
)

writer_agent = Agent(
    role="보고서 작성자",
    goal="분석 결과를 요약하여 보고서 작성",
    llm=ChatOpenAI(model="gpt-4o-mini"),  # 형식화된 작업
    tools=[]
)

# 비용 절감: 3개 모두 GPT-4 사용 시 $10/M vs 혼합 사용 시 $2/M
```

```
# Agent 레벨 캐싱
caching_agent = Agent(
    role="연구원",
    goal="정보 조사",
    cache=True  # 동일한 도구 호출 결과를 캐싱
)

# LLM 레벨 캐싱 (LangChain)
from langchain.globals import set_llm_cache
from langchain.cache import SQLiteCache

set_llm_cache(SQLiteCache(database_path=".langchain.db"))
# 동일한 프롬프트에 대해 캐시된 응답 반환

# 실전 팁: 캐시는 개발/테스트 단계에서 특히 유용
# 프로덕션에서는 캐시 TTL 설정 필요
```

```
import asyncio

# 독립적인 여러 Crew를 동시 실행
async def run_parallel_crews():
    results = await asyncio.gather(
        market_research_crew.kickoff_async(inputs={"topic": "AI"}),
        competitor_analysis_crew.kickoff_async(inputs={"topic": "AI"}),
        trend_analysis_crew.kickoff_async(inputs={"topic": "AI"})
    )
    return results

# 순차 실행: 30분 → 비동기 실행: 12분
```

4️⃣ 프롬프트 최적화

5️⃣ max_iter 적절히 설정

6️⃣ 비용 모니터링 시스템 구축

- ☑ 각 에이전트에 작업 복잡도에 맞는 모델 할당했는가?
- ☑ 캐싱을 활성화했는가? (개발/테스트 환경)
- ☑ 독립적 작업을 비동기로 실행하는가?
- ☑ max_iter이 10 이하로 설정되어 있는가?
- ☑ 프롬프트가 간결하고 명확한가? (불필요한 장황한 설명 제거)
- ☑ 비용 모니터링 시스템을 구축했는가?

3.9.4. 이슈 4: 품질 저하 및 환각(Hallucination)

현상: 에이전트가 지시와 다른 엉뚱한 결과물을 만들거나, 사실이 아닌 내용을 지어냅니다.

시나리오:

"AI 연구원" 에이전트가 존재하지 않는 논문과 통계를 인용하여 보고서 작성

원인:

Agent의 role과 goal이 너무 일반적이고, expected_output에 "출처 명시" 요구사항 누락

원인 분석 (4가지 주요 원인)

해결 방안: 구체성(Specificity) 극대화

1. 구조화된 출력으로 환각 방지

Pydantic 모델을 사용하여 출력 형식을 강제합니다.

2. 다단계 검증 워크플로우

생성 → 검토 → 수정의 다단계 프로세스를 구축합니다.

3. 제약사항(Guardrails) 설정

에이전트가 지켜야 할 명확한 규칙을 backstory에 명시합니다.

- ☑ Agent의 role, goal, backstory가 구체적이고 측정 가능한가?
- ☑ Task의 expected_output이 형식과 분량을 명시하는가?
- ☑ 출처 명시, 팩트 체크 등의 제약사항이 포함되어 있는가?
- ☑ Pydantic 모델로 출력 형식을 강제하는가?
- ☑ 다단계 검증 프로세스(생성 → 검토 → 수정)가 구현되어 있는가?
- ☑ Human-in-the-loop으로 중요한 결정을 검증하는가?

3.10. 실무 활용 팁: 인간 개입 및 구조화된 출력

에이전트의 자율성을 제어하고, 결과물의 형식을 표준화하는 두 가지 핵심적인 실무 기능을 알아봅니다.

1. 인간의 개입 (Human-in-the-loop)

에이전트가 중요한 결정을 내리거나, 도구를 사용하기 전에 사람의 승인을 받도록 설정할 수 있습니다. 이는 안정성과 예측 가능성을 크게 높여줍니다.

```
# Task 실행 전 사람의 확인을 요구하도록 설정
human_approval_task = Task(
    description="민감한 데이터베이스를 업데이트하는 중요한 작업입니다. 계속 진행하시겠습니까?",
    expected_output="사용자의 승인 또는 거절 확인.",
    agent=db_admin_agent,
    human_input=True  # 이 부분이 핵심입니다!
)

# kickoff() 실행 시, 이 Task 차례에서 실행이 멈추고 터미널에 사용자 입력을 요구합니다.
```

2. 구조화된 출력 (Structured Output)

Task의 최종 결과물을 일반 텍스트가 아닌, 지정된 Pydantic 모델이나 JSON 형식으로 강제할 수 있습니다. 이를 통해 결과물을 프로그램에서 안정적으로 파싱하고 사용할 수 있습니다.

```
from pydantic import BaseModel, Field

# 1. 원하는 출력 구조를 Pydantic 모델로 정의
class BlogPost(BaseModel):
    title: str = Field(description="블로그 포스트의 제목")
    introduction: str = Field(description="독자의 흥미를 끄는 간결한 서론")
    body_content: str = Field(description="포스트의 본문 내용")
    conclusion: str = Field(description="주요 내용을 요약하는 결론")

# 2. Task에 Pydantic 모델을 지정
structured_output_task = Task(
    description="AI의 미래에 대한 블로그 포스트를 작성하세요.",
    expected_output="지정된 BlogPost Pydantic 모델 형식의 블로그 포스트.",
    agent=writer_agent,
    output_pydantic=BlogPost  # 이 부분이 핵심입니다!
)

# crew.kickoff()의 결과는 BlogPost 객체가 됩니다.
# result.title, result.introduction 등으로 쉽게 접근할 수 있습니다.
```

Chapter 4. RAG 시스템 구현: 기초부터 Agentic RAG까지

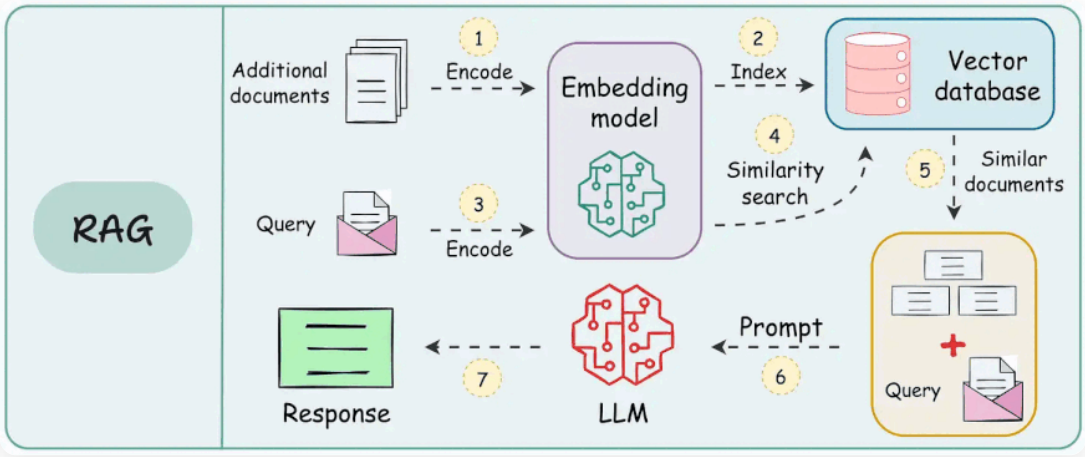
학습 목표

- RAG의 개념과 단순 RAG vs Agentic RAG의 차이점을 이해한다.
- RAG 시스템의 파일 구조와 각 모듈의 역할을 파악한다.
- 색인(Indexing), 검색(Retrieval), 생성(Generation) 전 과정을 학습한다.
- 여러 전문 에이전트가 협력하는 Agentic RAG 파이프라인을 구축한다.
- 컨텍스트 기반 작업 연결로 답변의 신뢰도를 극대화하는 방법을 익힌다.

4.1. RAG란 무엇인가?

Retrieval-Augmented Generation (검색 증강 생성)

RAG는 LLM이 답변을 생성할 때, **외부 지식 베이스(Vector DB 등)에서 관련 정보를 실시간으로 검색(Retrieval)**하여, 그 정보를 바탕으로 답변의 정확성과 신뢰도를 높이는(Augmented Generation) 기술입니다.

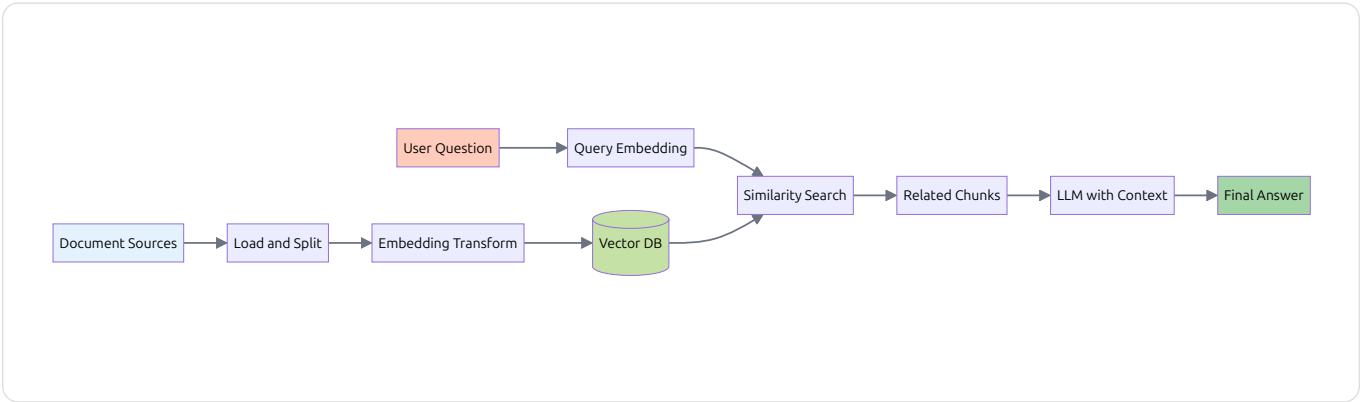


RAG의 필요성

- **최신 정보 반영:** LLM의 학습 데이터 시점 이후의 최신 정보를 답변에 활용.
- **환각(Hallucination) 억제:** 사실에 근거한 답변 생성으로 신뢰도 향상.
- **지식 확장:** 내부 문서, DB 등 특정 도메인의 비공개 지식을 LLM에 주입.

4.2. RAG 시스템 아키텍처

RAG 시스템은 크게 '색인(Indexing)'과 '검색 및 생성(Retrieval & Generation)' 두 단계로 나뉩니다.



1. 색인 (Indexing): 지식 베이스 구축 단계 (오프라인)

1. **Load:** PDF, TXT, DB 등 다양한 소스에서 데이터 로드.
2. **Split:** 로드된 데이터를 의미 있는 작은 단위(Chunk)로 분할.

3. **Embed:** 각 데이터 청크를 벡터(Vector)로 변환.
4. **Store:** 변환된 벡터를 ChromaDB와 같은 벡터 저장소에 저장.

2. 검색 및 생성 (Retrieval & Generation): 사용자 쿼리 처리 단계 (온라인)

1. **Retrieve:** 사용자 질문을 벡터로 변환하여, 벡터 저장소에서 가장 유사한 데이터 청크 검색.
2. **Generate:** 검색된 데이터 청크를 원본 질문과 함께 LLM에 전달하여, 풍부한 컨텍스트를 바탕으로 최종 답변 생성.

4.3. 단순 RAG vs Agentic RAG

Agentic RAG는 단순 RAG를 넘어, 여러 전문 에이전트가 협력하여 검색된 정보의 품질을 높이고 최종 답변을 생성하는 고도화된 방식입니다.

단순 RAG (Simple RAG)

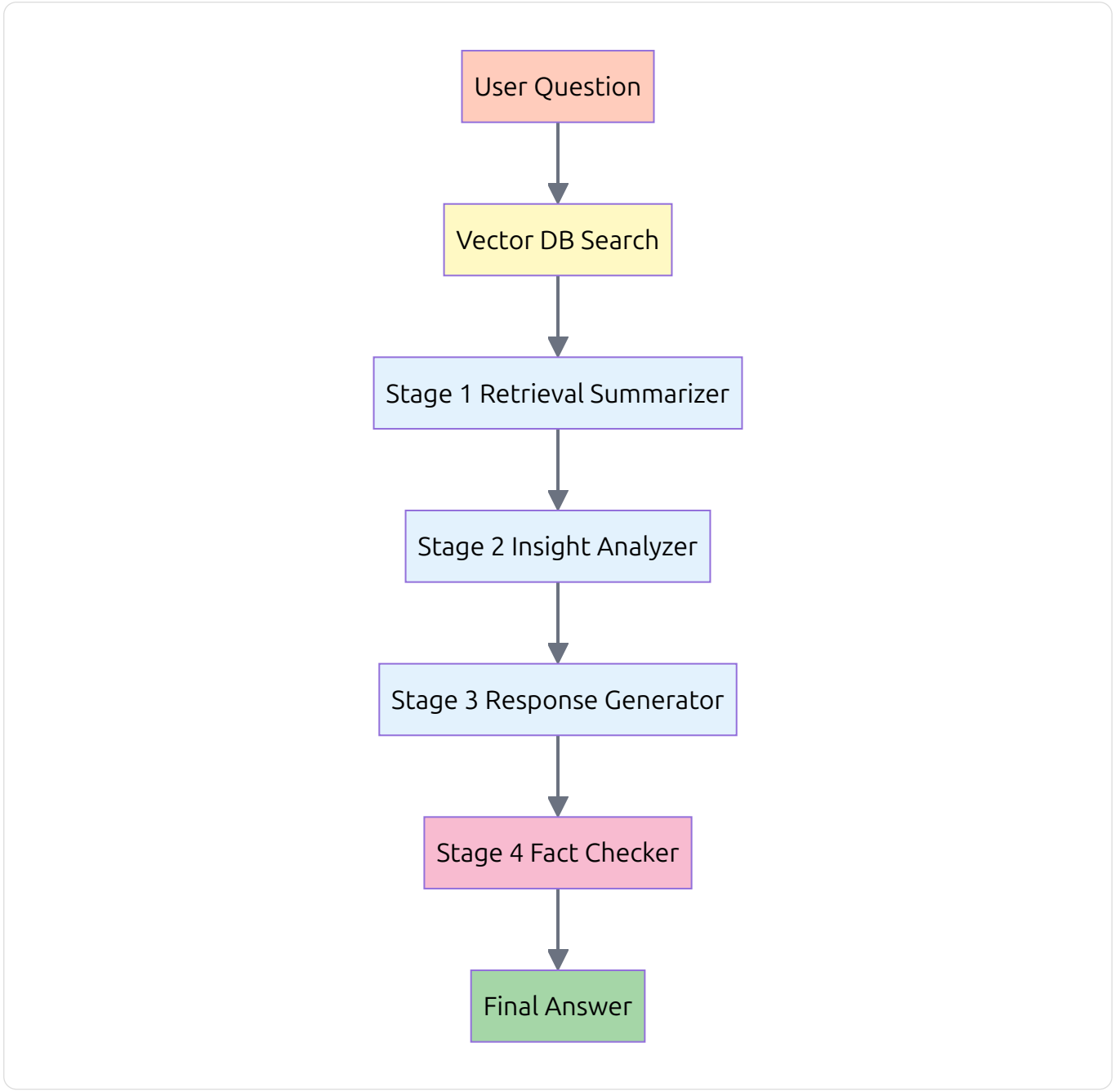
Retrieve → Generate

검색된 정보를 추가 컨텍스트로 활용하여 LLM이 바로 답변을 생성하는 간단한 2단계 프로세스입니다.

Agentic RAG (rag_crew.py에서 구현)

Retrieve → Summarize → Analyze → Generate → Fact-Check & Refine

검색된 정보를 바탕으로 여러 에이전트가 요약, 심층 분석, 답변 생성, 사실 검증 등 분업화된 작업을 순차적으로 수행하여 답변의 깊이와 신뢰도를 극대화합니다.



4.4. RAG 시스템 파일 구조 및 역할

03_rag_system/ 폴더 구성

RAG 시스템은 4개의 모듈로 구성되며, 각 모듈은 명확한 역할을 담당합니다.

4.5. 색인(Indexing) - document_processor.py

다양한 문서 처리하기

`document_processor.py` 는 PDF, TXT, JSON, CSV 등 다양한 형식의 문서를 로드하고, 텍스트를 추출하여 벡터 DB에 저장할 준비를 합니다.

```
# 03_rag_system/document_processor.py

class DocumentProcessor:
    def process_file(self, file_path: str) -> Dict[str, Any]:
        """파일 확장자에 따라 적절한 처리 메소드를 호출합니다."""
        file_path = Path(file_path)

        if file_path.suffix.lower() == '.txt':
            return self._process_txt_file(file_path)
        elif file_path.suffix.lower() == '.json':
            return self._process_json_file(file_path)
        elif file_path.suffix.lower() == '.csv':
            return self._process_csv_file(file_path)
        elif file_path.suffix.lower() == '.pdf':
            return self._process_pdf_file(file_path)
        # ...

    def _process_pdf_file(self, file_path: Path) -> Dict[str, Any]:
        """PDF 파일을 읽고 처리합니다."""
        loader = PyPDFLoader(str(file_path))
        pages = loader.load_and_split()
        text = " ".join([page.page_content for page in pages])
        # ...
```

4.6. 벡터 저장소 및 검색: vector_store.py

재사용 가능한 DB 관리 클래스 및 검색 기능

`AdvancedVectorStore` 클래스는 ChromaDB와의 모든 상호작용을 캡슐화하여, 다른 스크립트에서 DB 관련 코드를 간결하게 유지하도록 돕습니다. CRUD, 검색, 배치 처리 등 실전적인 기능을 제공합니다.

1️⃣ 핵심 메서드

```
# 03_rag_system/vector_store.py

class AdvancedVectorStore:
    def __init__(self, persist_directory: str = "./chroma_db"):
        # 클라이언트 및 임베딩 모델 설정
        self.client = chromadb.PersistentClient(path=persist_directory)
        self.embedding_model = SentenceTransformer("sentence-transformers/all-MiniL

    def create_collection(self, collection_name: str, ...):
        # 컬렉션 생성 또는 로드

    def add_documents(self, collection_name: str, documents: List[str], ...):
        # 문서 임베딩 및 DB 추가

    def search_documents(self, collection_name: str, query: str, n_results: int = 5
        """쿼리와 유사한 문서를 검색합니다."""
        # Step 1: 쿼리를 벡터로 임베딩
        query_embedding = self.embedding_model.encode([query])

        # Step 2: 벡터 DB에서 유사도 검색
        results = collection.query(
            query_embeddings=query_embedding.tolist(),
            n_results=n_results
        )

        # Step 3: 결과 포맷팅 (id, document, metadata, distance)
        return self._format_search_results(results)
```

2️⃣ 임베딩 모델

`sentence-transformers/all-MiniLM-L6-v2` 모델을 사용하여 텍스트를 벡터로 변환합니다. 빠르고 효율적이며, 한국어도 어느 정도 지원합니다.

3️⃣ 유사도 측정

ChromaDB는 코사인 유사도를 기반으로 문서를 검색합니다. 반환되는 `distance` 값이 **작을수록** 쿼리와 문서가 유사합니다.

4.7. RAG 공통 유틸리티: rag_utils.py

중복 제거와 일관성 확보

RAG 시스템의 모든 파일에서 공통으로 사용되는 기능들을 제공하는 유틸리티 모듈입니다. 중복 코드를 제거하고 일관된 인터페이스를 제공합니다.

📦 주요 구성 요소

🔄 사용 패턴

```
# 03_rag_system의 모든 파일에서 동일하게 사용

# document_processor.py
from rag_utils import get_llm, RAGConfig, CHROMA_PERSIST_DIRECTORY

llm = get_llm()  # 일관된 LLM 설정
vector_store = AdvancedVectorStore(persist_directory=CHROMA_PERSIST_DIRECTORY)

# rag_crew.py
from rag_utils import get_llm, UNIFIED_COLLECTION_NAME

llm = get_llm("gpt-4o-mini")  # 필요 시 커스텀 모델
collection = UNIFIED_COLLECTION_NAME  # 통일된 컬렉션 이름

# vector_store.py
from rag_utils import CHROMA_PERSIST_DIRECTORY

self.client = chromadb.PersistentClient(path=CHROMA_PERSIST_DIRECTORY)
```

4.8. Agentic RAG 크루 구성

4개의 전문 에이전트 정의

🔍 검색 결과 요약 전문가

역할:

방대한 문서에서 핵심 정보를 빠르게 파악

목표:

원본의 본질을 유지하면서 정보를 압축하고 구조화

📊 정보 심층 분석가

역할:

표면적 정보를 넘어 이면의 의미를 파헤침

목표:

숨겨진 인사이트, 패턴, 연관성 및 불일치점 발견

✍️ 지식 기반 답변 생성가

역할:

복잡한 분석을 이해하기 쉬운 설명으로 변환

목표:

사용자 질문에 대해 명확하고 포괄적인 답변 생성

✅ 사실 검증 및 최종 편집자

역할:

진실을 파헤치는 탐정과 같은 꼼꼼함

목표:

원본 문서 근거 확인 및 신뢰도 100% 확보

4.9. Agentic 워크플로우: 명시적 컨텍스트 전달

context를 이용한 작업 연결

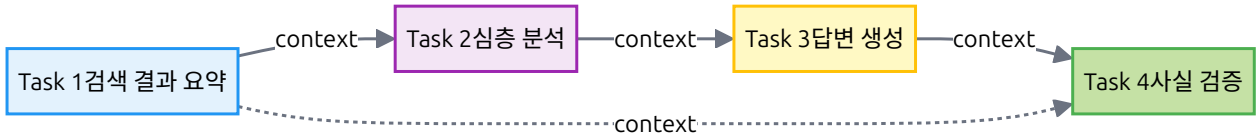
Agentic RAG의 핵심은 이전 단계의 결과물을 다음 단계의 에이전트가 명확히 입력받아 작업을 수행하는 것입니다. CrewAI에서는 Task의 `context` 속성을 사용하여 이를 구현합니다.

1️⃣ Context 파라미터를 이용한 작업 체인

```
# 03_rag_system/rag_crew.py - _create_tasks() 메서드

# 작업 1: 검색 결과 요약 (context 의존성 없음)
retrieval_summary_task = Task(
    description='''사용자 쿼리 '{query}'와 관련된 다음 문서들을 검토하고, 각 문서의 핵심 내용을

    문서 내용:
    {documents}

    요약은 후속 분석 단계에서 사용할 수 있도록 명확하고 구조화된 형식이어야 합니다.''',
    expected_output='각 문서의 핵심 내용을 담은 구조화된 요약 보고서.',
    agent=retrieval_summarizer
)

# 작업 2: 심층 분석 (작업 1의 결과를 context로 받음)
insight_analysis_task = Task(
    description='''앞선 단계에서 요약된 문서 내용을 바탕으로, 사용자 쿼리 '{query}'와 관련하여 심

    요구사항:
    - 주요 주제, 개념, 인물, 사건 등을 식별하세요.
    - 정보들 사이의 숨겨진 연관성이나 패턴을 찾아내세요.
    - 정보가 서로 충돌하거나 모순되는 지점을 명확히 지적하세요.
    - 정보가 부족한 부분(Information Gaps)을 식별하세요.''',
    expected_output='상세한 분석 보고서 (주요 주제, 연관성, 모순점, 정보 공백 포함)',
    agent=insight_analyzer,
    context=[retrieval_summary_task]  # ← Task 1의 결과를 입력으로 받음
)

# 작업 3: 답변 생성 (작업 2의 결과를 context로 받음)
response_generation_task = Task(
    description='''분석 보고서를 바탕으로, 사용자의 초기 쿼리 '{query}'에 대한 포괄적이고 이해하

    답변은 분석된 핵심 인사이트를 모두 포함해야 하며, 논리적이고 구조적으로 작성되어야 합니다.''',
    expected_output='사용자의 질문에 대한 잘 구조화된 답변 초안 (서론, 본론, 결론)',
    agent=response_generator,
    context=[insight_analysis_task]  # ← Task 2의 결과를 입력으로 받음
)

# 작업 4: 사실 검증 (작업 3과 작업 1의 결과를 모두 context로 받음)
fact_checking_task = Task(
    description='''작성된 답변 초안을 원본 문서 요약본과 비교하여 사실 관계를 철저히 검증하고, 최종

    검증 절차:
    1. 답변 초안의 모든 주장이 원본 문서 요약에 의해 뒷받침되는지 확인하세요.
```

```
    2. 'Hallucination'(환각)이나 잘못된 정보가 있다면 즉시 수정하세요.
    3. 문맥을 더 명확하게 하거나 표현을 다듬어 가독성을 높이세요.
    4. 최종적으로, 수정이 완료된 완벽한 답변을 제출하세요.''',
    expected_output='오류가 모두 수정되고 사실 관계가 검증된, 최종적이고 완성도 높은 답변.',
    agent=fact_checker,
    context=[response_generation_task, retrieval_summary_task]  # ← Task 3과 Task 1
)
```

2️⃣ 컨텍스트 흐름 다이어그램



4.10. 검색 통합 및 전체 실행 흐름

RAG 시스템의 통합된 워크플로우

`rag_crew.py` 의 `process_query` 메서드는 검색 결과를 가져와 에이전트 파이프라인에 전달합니다.

1️⃣ process_query 메서드: 검색과 에이전트 통합

```
# 03_rag_system/rag_crew.py - process_query() 메서드 (lines 193-237)

class RAGCrew:
    def process_query(self, query: str, collection_name: str, n_results: int = 5)
        """
        사용자의 쿼리를 받아 Agentic RAG 프로세스를 실행합니다.

        Args:
            query: 사용자 질문 (예: "CrewAI에서 계층적 프로세스는 어떻게 작동하나요?")
            collection_name: 검색할 벡터 DB 컬렉션 이름 (예: "crewai_docs")
            n_results: 검색할 문서 개수 (기본값: 5)

        Returns:
            Dict containing query, retrieval_results, and final_answer
        """
        print_wrapped(f"🚀 Agentic RAG 크루가 쿼리 처리를 시작합니다: \"{query}\"", borde

        # 1단계: 벡터 저장소에서 관련 문서 검색 (Retrieval)
        print(f"\n📚 1단계: '{collection_name}' 컬렉션에서 관련 문서 검색 중...")
        vector_store = AdvancedVectorStore()
        retrieval_results = vector_store.search_documents(
            collection_name=collection_name,
            query=query,
            n_results=n_results
        )

        if not retrieval_results:
            print("❌ 검색된 관련 문서가 없습니다. RAG 프로세스를 종료합니다.")
            return {'error': '관련 문서를 찾을 수 없습니다.'}

        # 검색 결과를 LLM이 이해하기 쉬운 형태로 포맷팅
        documents_str = self._format_retrieval_results(retrieval_results)
        print(f"✅ {len(retrieval_results)}개의 관련 문서를 찾았습니다.")

        # 2단계: 포맷팅된 검색 결과를 입력으로 하여 RAG 크루 실행
        print("\n🤖 2단계: Agentic Crew 실행 (요약 → 분석 → 생성 → 검증)")
        inputs = {
            'query': query,
            'documents': documents_str  # 검색된 문서를 첫 번째 Task에 전달
        }

        try:
            crew_result = self.crew.kickoff(inputs=inputs)
```

```
            result = {
                'query': query,
                'retrieval_results': retrieval_results,
                'final_answer': crew_result,
            }

            print_wrapped("✅ Agentic RAG 크루 쿼리 처리 완료!")
            return result

        except Exception as e:
            print(f"❌ RAG 크루 실행 중 오류 발생: {e}")
            return {'error': str(e)}
```

2️⃣ _format_retrieval_results 헬퍼 메서드

```
# 03_rag_system/rag_crew.py - _format_retrieval_results() 메서드 (lines 239-253)

def _format_retrieval_results(self, results: List[Dict[str, Any]]) -> str:
    """검색 결과를 LLM의 컨텍스트로 사용하기 위해 문자열로 포맷팅합니다."""
    formatted_results = []
    for i, result in enumerate(results):
        doc = result.get('document', '')
        metadata = result.get('metadata', {})
        distance = result.get('distance', float('nan'))

        formatted_results.append(
            f"--- 문서 {i+1} (관련도: {1-distance:.4f}) ---\n"
            f"내용: {doc}\n"
            f"출처: {metadata.get('file_name', 'N/A')}\n"
        )

    return "\n".join(formatted_results)
```

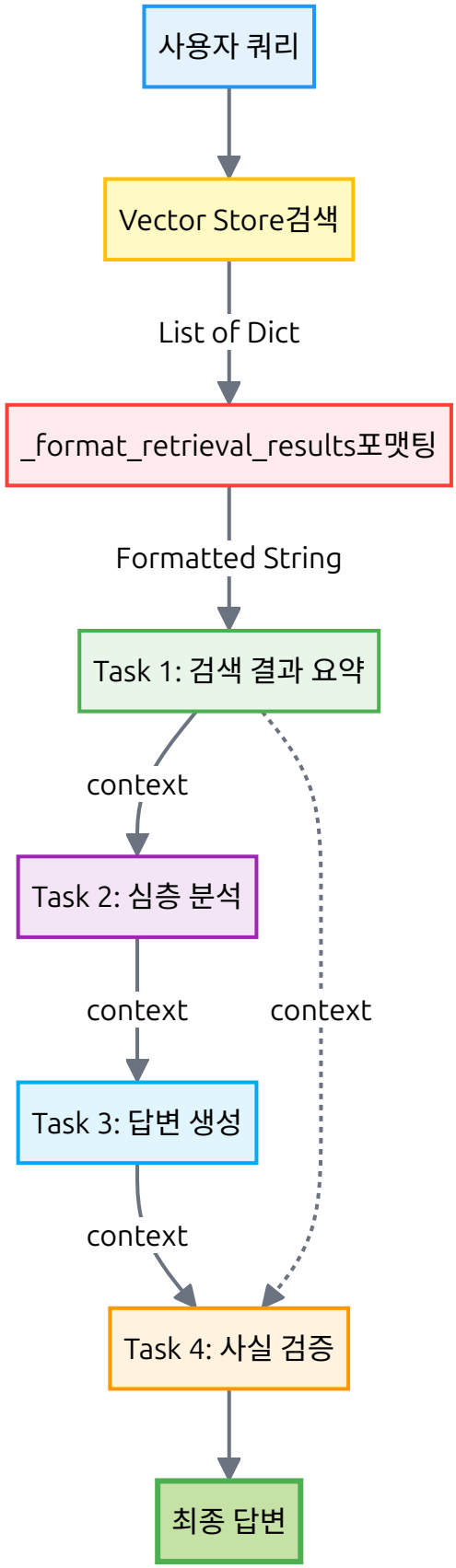
3️⃣ 포맷팅 출력 예시

4️⃣ 전체 실행 흐름: 단계별 데이터 변환

```
# 사용자 질문 → 검색 → 포맷팅 → 에이전트 파이프라인 → 최종 답변

# 사용자 입력
user_query = "CrewAI에서 계층적 프로세스는 어떻게 작동하나요?"

# 🔹 Step 1: 벡터 DB 검색
retrieval_results = vector_store.search_documents(
    collection_name="crewai_docs",
    query=user_query,
    n_results=5
)
# ↓ 출력: [{'document': '...', 'metadata': {...}, 'distance': 0.12}, ...]

# 🔹 Step 2: 검색 결과를 LLM 친화적 형식으로 포맷팅
documents_str = _format_retrieval_results(retrieval_results)
# ↓ 출력: "--- 문서 1 (관련도: 0.8800) ---\n내용: ...\n출처: ..."

# 🔹 Step 3: Crew에 입력 전달
inputs = {
    'query': user_query,           # Task description의 {query} 플레이스홀더에 주입
    'documents': documents_str     # Task description의 {documents} 플레이스홀더에 주입
}

# 🔹 Step 4: Agentic RAG 파이프라인 실행
crew_result = rag_crew.kickoff(inputs=inputs)

# 실행 순서:
# ① Task 1 (retrieval_summarizer):
#    - Input: {query}와 {documents}를 포함한 Task description
#    - Output: "각 문서의 핵심 내용을 요약한 보고서"
#
# ② Task 2 (insight_analyzer):
#    - Input: Task 1의 출력 (context parameter로 자동 주입)
#    - Output: "주요 주제, 연관성, 모순점, 정보 공백 분석 보고서"
#
# ③ Task 3 (response_generator):
#    - Input: Task 2의 출력 (context parameter로 자동 주입)
#    - Output: "사용자 질문에 대한 답변 초안"
#
# ④ Task 4 (fact_checker):
#    - Input: Task 3의 출력 + Task 1의 출력 (context parameter로 자동 주입)
#    - Output: "사실 검증 및 수정된 최종 답변"

# 🔹 Step 5: 최종 결과 반환
```

```
final_answer = crew_result  # Task 4의 최종 출력
```

5️⃣ 데이터 흐름 다이어그램



4.11. 학습 포인트: RAG 시스템 마스터하기

`rag_crew.py` 를 실행하여 Agentic RAG 시스템의 성능을 확인합니다.

```
# RAG 시스템 실행 예제

# 1. 문서 처리 및 색인
python 03_rag_system/document_processor.py "data/crewai_complete_guide.pdf"

# 2. Agentic RAG 실행
python 03_rag_system/rag_crew.py "CrewAI에서 계층적 프로세스는 어떻게 작동하나요?"
```

Chapter 5. 핵심 요약 및 전망

지금까지 무엇을 배웠고, 앞으로 무엇을 할 수 있을까요?

🚀 CrewAI 핵심 역량 요약

- **역할 기반 에이전트 설계:** 명확한 역할(Role), 목표(Goal), 배경(Backstory)을 부여하여 전문가 에이전트를 만드는 방법.
- **체계적인 작업 분업:** 복잡한 문제를 작은 작업(Task)으로 나누고, 순차(Sequential) 또는 계층(Hierarchical) 프로세스를 통해 협업시키는 방법.
- **무한한 능력 확장:** 커스텀 도구(Tools)를 연동하여 에이전트가 외부 세계와 상호작용하고, 실시간 정보에 접근하게 하는 방법.
- **고급 시스템 구축:** RAG, 워크플로우 오케스트레이션 등 실무적인 에이전트 시스템을 구축하는 설계 패턴.

🔭 미래 전망: 자율 에이전트의 시대

CrewAI와 같은 프레임워크는 단순한 챗봇을 넘어, 스스로 문제를 해결하는 **자율적인 AI 워크포스**를 구축하는 핵심 기술입니다. 앞으로 AI 에이전트는 다음과 같은 영역에서 더욱 중요해질 것입니다.

- **복잡한 분석 및 리서치 자동화**
- **소프트웨어 개발 및 테스트 프로세스 보조**
- **개인화된 비서 및 고객 지원 시스템**

- **콘텐츠 생성 및 마케팅 캠페인 자동화**

이제 여러분의 아이디어를 CrewAI를 통해 현실로 만들 차례입니다!

5.1. 마무리

CrewAI를 마스터하여 AI 에이전트의 무한한 가능성을 탐험해보세요!

추가 학습 리소스

- **CrewAI 공식 문서:** https://docs.crewai.com/
- **CrewAI GitHub:** https://github.com/joaomdmoura/crewAI
- **CrewAI 커뮤니티:** CrewAI Discord 서버에서 활발한 커뮤니티 참여
- **CrewAI 공식 블로그:** 최신 업데이트와 사용 사례
- **LLM 에이전트 모니터링 (LangSmith):** https://www.langchain.com/langsmith
- **컨테이너화 (Docker):** https://www.docker.com/
- **하버드 비즈니스 리뷰 - AI 에이전트:** AI가 비즈니스를 어떻게 혁신하는가