



Autonomous System

WEEK 04

함께하는 AI 에이전트 개발

4주차: 스스로 생각하고 판단하며 실행하는 AI



PROJECT GOAL

리서치 어시스턴트



DURATION

180 Mins (3h)



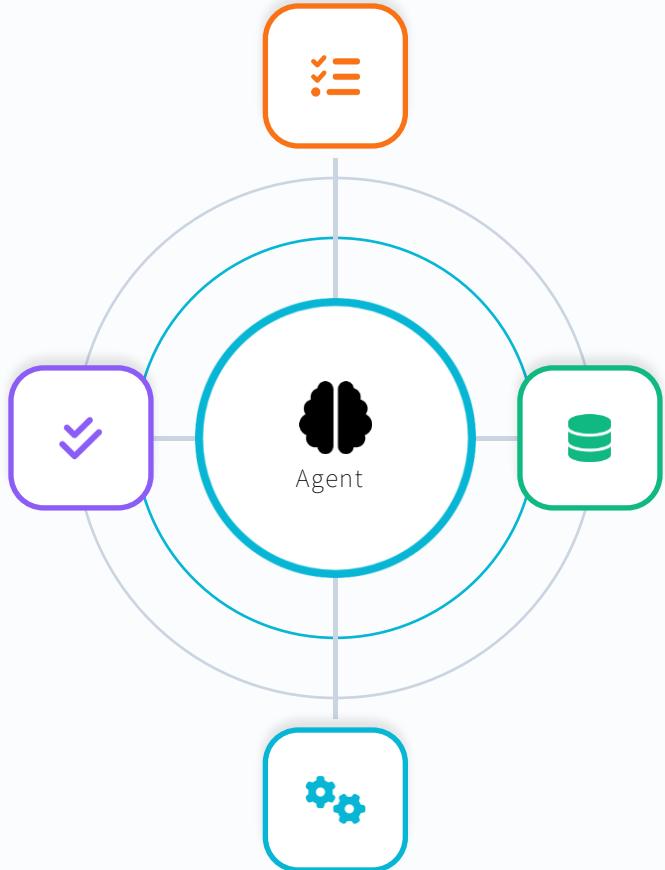
CORE STACK

Python, OpenAI API



INSTRUCTOR

Thomas Moon



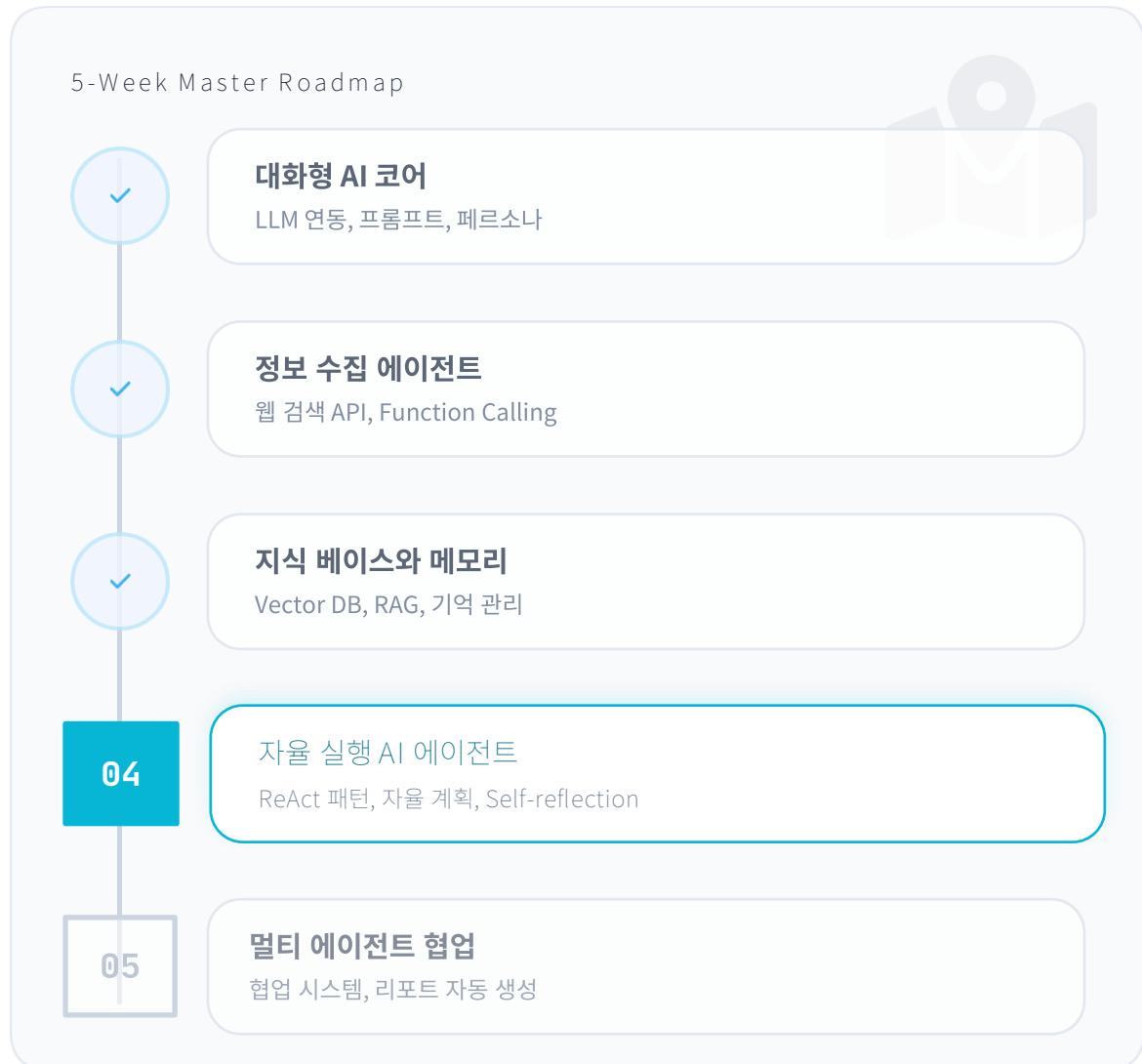
Today's Agenda & Roadmap

이번 시간의 학습 목표와 전체 과정을 조망합니다.

TODAY: WEEK 04

자율 실행 AI 에이전트 구축

- 지난 주 복습 및 현재 시스템 한계
- ReAct 패턴과 자율 실행 이론
- 실습 Part1: Task Decomposition 구현
복잡한 작업을 서브태스크로 분해
- 실습 Part2: 자율 실행 오케스트레이터 구현
- 마무리 및 다음 주 예고



01

SECTION

REVIEW

지난 주 복습 및 개선

3주차 성과 점검과 개선 포인트 확정

KEY TOPICS

✓ 정보 수집 에이전트 기능 점검

⌚ 중복 검색과 비용 효율성

🧠 Context Loss (문맥 손실) 문제

💾 RAG (검색 증강 생성) 도입

3주차 완성 핵심 코드 리뷰

현재까지 구축된 프로젝트 아키텍처

ConversationManager

Week 01

OpenAI API 연동 및 대화 히스토리/컨텍스트 관리. 시스템 페르소나 설정 및 토큰 관리 담당.

chat(user_input) save_conversation() load_history()

SearchAgent

Week 02

Tavily API를 활용한 웹 검색 실행 및 결과 파싱. 외부 정보 수집을 위한 도구(Tool) 역할.

search(query) search_with_memory() format_results()

MemoryManager

Week 03

ChromaDB 베이스 데이터베이스 관리. 문서 임베딩 생성, 저장 및 의미 기반 유사도 검색(RAG) 수행.

add_to_memory(text) search_memory(query) _get_embedding()

ai-research-assistant/

```

├── main.py
├── config/
│   ├── prompts.py
│   └── settings.py
└── src/
    ├── conversation_manager.py # Week 1
    ├── search_agent.py # Week 2
    ├── memory_manager.py # Week 3
    ├── tools/
    │   ├── tool_definitions.py
    │   └── web_search.py
    └── utils/
        └── embeddings.py
└── data/
    └── chroma_db/

```

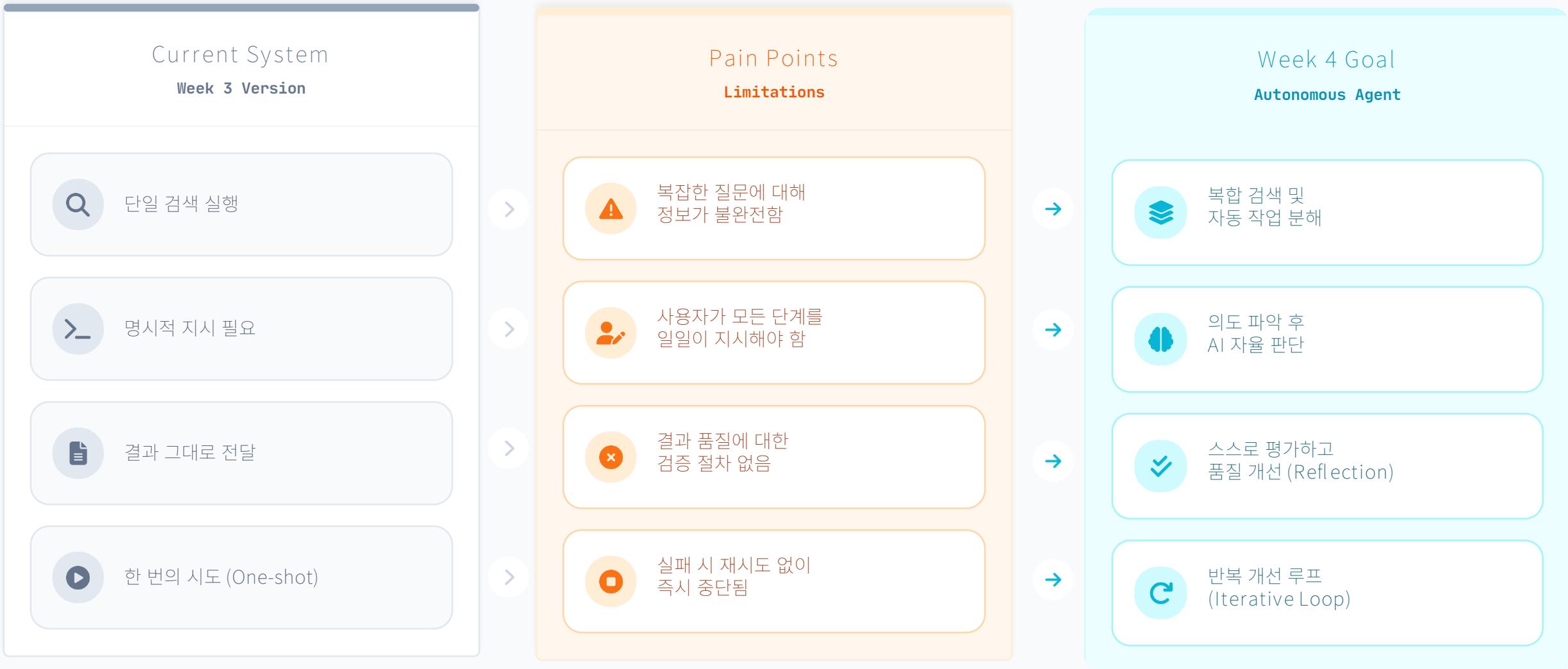
Troubleshooting & 개선

자주 발생하는 이슈 및 해결 방안

Q&A

현재 시스템의 한계점 분석

왜 자율 실행(Autonomous) 에이전트가 필요한가?



자율 실행 에이전트 목표

 복잡한 작업을 자동으로 서브태스크로 분해

 ReAct 패턴으로 추론-행동 순환 실행

 결과 품질을 스스로 평가하고 재시도

 전체 시스템 조율 및 무한 루프 방지

컴포넌트 상세 명세

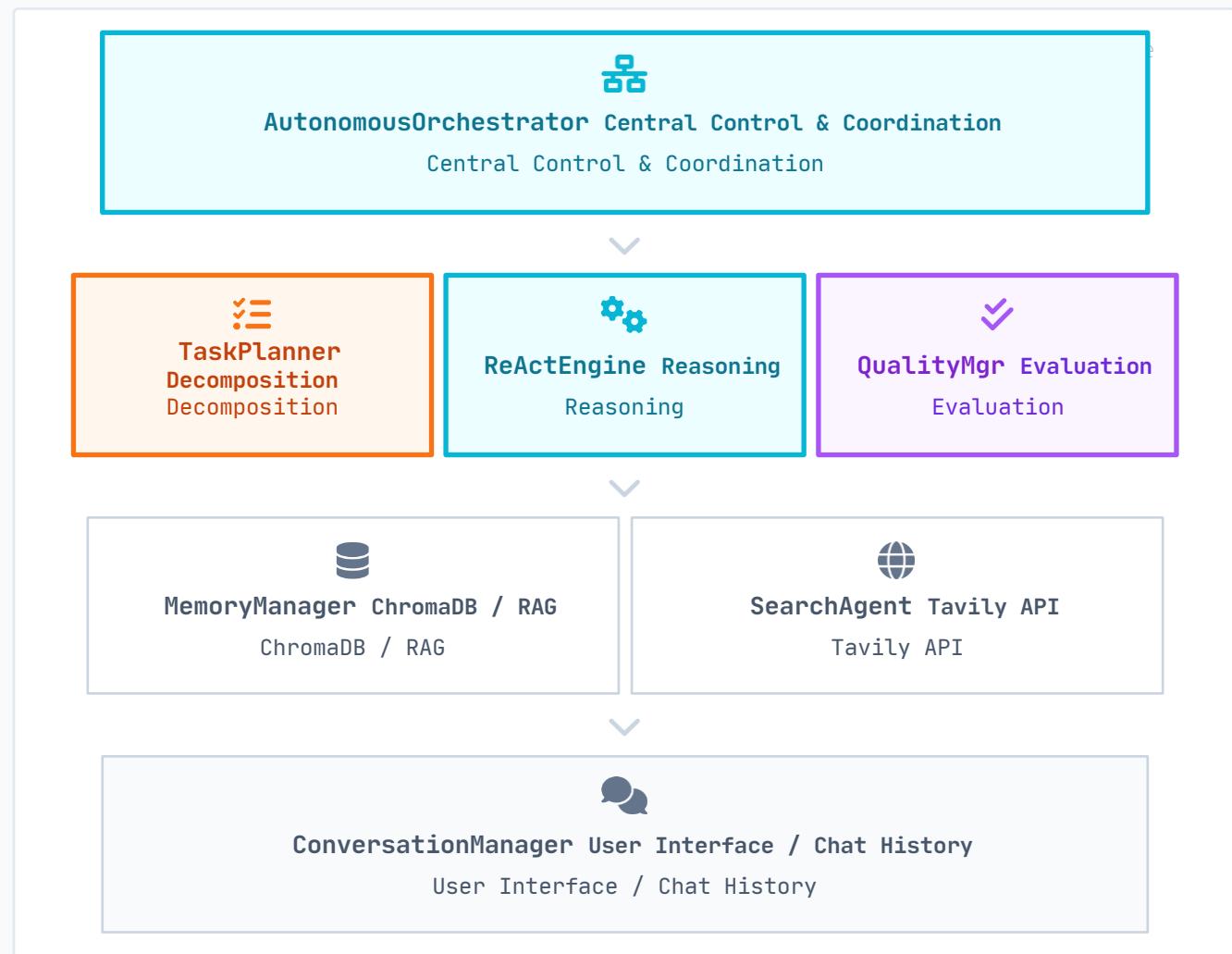
컴포넌트	파일명	역할
TaskPlanner	src/task_planner.py	복잡한 작업을 실행 가능한 서브태스크로 분해
ReActEngine	src/react_engine.py	Thought-Action-Observation 순환 실행
QualityManager	src/quality_manager.py	결과 품질 평가 및 재시도 여부 판단
LoopPrevention	src/loop_prevention.py	무한 루프 및 반복 행동 감지/방지
Autonomous Orchestrator	src/orchestrator.py	전체 시스템 통합 조율 및 실행 흐름 관리

4주차 완성 프로젝트 구조 & 아키텍처

자율 실행을 위한 오케스트레이터 및 엔진 추가

```

ai-research-assistant/
  main.py MODIFIED
  config/
    prompts.py MODIFIED
    settings.py
  src/
    conversation_manager.py
    search_agent.py
    memory_manager.py
    task_planner.py NEW
    react_engine.py NEW
    quality_manager.py NEW
    loop_prevention.py NEW
    orchestrator.py NEW
    tools/
  tests/
    test_task_planner.py NEW
    test_orchestrator.py NEW
  
```



02

SECTION

THEORY AI FUNDAMENTALS

ReAct 패턴과 자율 실행 이론

자율 에이전트의 핵심 두뇌, 추론과 행동의 순환 구조를 이해합니다.

◎ Key Learning Points

- ✓ 자율 에이전트의 정의와 특성
- ✓ Thought-Action-Observation 순환
- ✓ ReAct 패턴 (Reasoning + Acting)
- ✓ 자율 실행의 도전과제와 전략

Keywords:

● React ● Autonomous Agent ● Reasoning & Acting

자율 에이전트(Autonomous Agent)란?

정의와 4가지 핵심 특성

What is Autonomous Agent?

목표를 부여받으면 **스스로 계획하고**, 판단하며 실행하고,
그 결과를 평가하여 **스스로 개선하는** AI 시스템입니다.

Core Definition

단순한 지시 이행을 넘어, "어떻게(How)" 목표를 달성할지
스스로 결정하는 주체성을 가집니다.

전통적 프로그램과의 차이:

모든 경로가 코딩된 것이 아니라, 상황에 따라 경로를 동적으로 생성합니다.



전통적 챗봇 VS 자율 에이전트

단순 응답을 넘어선 자율적 문제 해결 능력

Criteria 비교 기준	Traditional 기존 챗봇	Autonomous 자율 에이전트
 실행 단위	 1회성 응답 Single Turn	 다단계 작업 Multi-step Tasks
 의사결정	 사용자 주도 User Directed	 AI 자율 판단 AI Autonomous
 오류 처리	 실패 시 종료 Fail & Stop	 자동 재시도 Auto-Retry
 품질 관리	 없음 None	 자기 평가 Self-Reflection

ReAct란? - Synergizing Reasoning & Acting in Language Models



(1) Hotspot QA

질문: Apple Remote 외에 원래 Apple Remote와 상호작용하도록 설계된 프로그램을 제어할 수 있는 다른 장치는 무엇인가요?

(1a) 기본
답변: iPod ❌

(1b) CoT

생각: 단계별로 생각해 봅시다. Apple Remote는 원래 Apple TV와 상호작용하도록 설계되었습니다. Apple TV는 iPhone, iPad, iPod Touch로 제어할 수 있습니다. 따라서 답변은 iPhone, iPad, iPod Touch입니다.
답변: iPhone, iPad, iPod Touch ❌

(1c) Act-Only

행동 1: 검색[Apple Remote]
관찰 1: Apple Remote는 원격 제어 장치입니다... 원래 Front Row 미디어 센터 프로그램을 제어하도록 설계되었습니다.

행동 2: 검색[Front Row]
관찰 2: [Front Row]를 찾을 수 없습니다. 유사한 것: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (소프트웨어)', ...]

행동 3: 검색[Front Row (소프트웨어)]
관찰 3: Front Row는 중단된 미디어 센터 소프트웨어입니다...

행동 4: 완료[예] ❌

(1d) ReAct (Reason + Act)

생각 1: Apple Remote를 검색하여 원래 상호작용하도록 설계된 프로그램을 찾아야 합니다.
행동 1: 검색[Apple Remote]
관찰 1: Apple Remote는 2005년 10월에 Apple이 소개한 원격 제어 장치입니다... 원래 Front Row 미디어 센터 프로그램을 제어하도록 설계되었습니다.

생각 2: Apple Remote는 원래 Front Row 미디어 센터 프로그램을 제어하도록 설계되었습니다. [Front Row]를 검색하여 다른 어떤 장치가 이를 제어할 수 있는지 찾아야 합니다.
행동 2: 검색[Front Row]
관찰 2: [Front Row]를 찾을 수 없습니다. 유사한 것: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (소프트웨어)', ...]

생각 3: Front Row가 발견되지 않았습니다. [Front Row (소프트웨어)]를 검색해야 합니다.
행동 3: 검색[Front Row (소프트웨어)]
관찰 3: Front Row는 중단된 미디어 센터 소프트웨어입니다...

생각 4: Front Row (소프트웨어)는 Apple Remote 또는 키보드 기능 키로 제어됩니다. 따라서 답변은 키보드 기능입니다.

AI Agent 기법을 처음 소개한 논문 →

ReAct: Synergizing Reasoning and Acting in Language Models

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, Yuan Cao

[Submitted on 6 Oct 2022 (v1), last revised 10 Mar 2023 (this version, v3)]

<https://arxiv.org/abs/2210.03629>

Published as a conference paper at ICLR 2023

REACT: SYNERGIZING REASONING AND ACTING IN LANGUAGE MODELS

Shunyu Yao^{*,1}, Jeffrey Zhao², Dian Yu², Nan Du², Izhak Shafran², Karthik Narasimhan¹, Yuan Cao²

¹Department of Computer Science, Princeton University

²Google Research, Brain team

¹{shunuy, karthikn}@princeton.edu

²{jeffreyzhao, dianyu, dunan, izhak, yuancao}@google.com

ABSTRACT

While large language models (LLMs) have demonstrated impressive performance across tasks in language understanding and interactive decision making, their abilities for reasoning (e.g. chain-of-thought prompting) and acting (e.g. action plan generation) have primarily been studied as separate topics. In this paper, we explore the use of LLMs to generate both reasoning traces and task-specific actions in an interleaved manner, allowing for greater synergy between the two: reasoning traces help the model induce, track, and update action plans as well as handle exceptions, while actions allow it to interface with and gather additional information from external sources such as knowledge bases or environments. We apply our approach, named ReAct, to a diverse set of language and decision making tasks and demonstrate its effectiveness over state-of-the-art baselines in addition to improved human interpretability and trustworthiness. Concretely, on question answering (HotpotQA) and fact verification (Fever), ReAct overcomes prevalent issues of hallucination and error propagation in chain-of-thought reasoning by interacting with a simple Wikipedia API, and generating human-like task-solving trajectories that are more interpretable than baselines without reasoning traces. Furthermore, on two interactive decision making benchmarks (ALFWorld and WebShop), ReAct outperforms imitation and reinforcement learning methods by an absolute success rate of 34% and 10% respectively, while being prompted with only one or two in-context examples.

ReAct란? - Synergizing Reasoning & Acting in Language Models

모델 구성	HotpotQA (정답률 EM)	FEVER (정확도 Acc)
Standard	28.7%	57.1%
CoT	29.4%	56.3%
Act-only	25.7%	58.9%
ReAct	27.4%	60.9% 
CoT-SC→ReAct	34.2% 	64.6% 
ReAct→CoT-SC	35.1% 	62.0%
Fully Supervised	67.5%	89.5%

※ 참고

CoT(Chain of Thought Prompting): 답을 바로 출력하는 대신, 중간 추론 과정을 단계별로 자연어로 작성하도록 유도하는 프롬프트 기법

SC(Self-Consistency Decoding): 동일한 질문에 대해 여러 번 CoT 응답을 생성하고, 그 중 다수결로 정답을 선택하는 방식

- ◆ CoT-SC→ReAct 또는 ReAct→CoT-SC가 가장 성능 좋음

- ◆ ReAct 단독도 CoT, Act-only보다 강력

항목	HotpotQA (EM)	FEVER (Accuracy)	선택
과제 유형	질의응답 (텍스트 생성)	사실 검증 (문장 분류)	
출력 형식	자연어 응답 (예: "1,800 to 7,000 ft")	분류 라벨 (SUPPORTS / REFUTES / NEI)	
평가 기준	모델 응답이 정답과 완전히 일치하는가	모델이 정답 라벨을 맞췄는가	
부분 정답 인정 여부	 없음 (부분 정답도 0점)	 해당 없음 (정답이면 1점)	
응답 길이/복잡도	자유로운 자연어 문장	3지 선다형 분류 문제	

C.O.T(Chain Thought Prompting) / S.C(Self-Consistency Decoding)

C.O.T

목적

- 모델에게 "생각하는 흐름"을 가르쳐 복잡한 문제(수학, 논리, 멀티홀 질문 등)를 더 정확히 풀게 하려는 기법

예시

● 일반 프롬프트:

plaintext

복사 편집

Q: 앤리스가 사과를 3개 갖고 있고 밥에게 2개를 주면 몇 개 남나요?
A: 1개

● CoT 프롬프트:

plaintext

복사 편집

Q: 앤리스가 사과를 3개 갖고 있고 밥에게 2개를 주면 몇 개 남나요?
A: 먼저 앤리스는 사과 3개를 가지고 있습니다.
그 중 2개를 밥에게 줍니다.
그러면 $3 - 2 = 1$ 이므로, 1개가 남습니다.
정답: 1개

S.C

정의

Self-Consistency (SC)는

동일한 질문에 대해 여러 번 CoT 응답을 생성하고, 그 중 다수결로 정답을 선택하는 방식입니다.

과정

- CoT 프롬프트로 여러 번 응답 생성 (예: 10~20개)
- 각 응답의 최종 정답을 추출
- 가장 많이 나온 정답을 최종 정답으로 채택

CoT + SC 조합 예시

plaintext

복사 편집

Q: $23 + 47$ 는 얼마인가요?
→ CoT 프롬프트로 10개 생성

답1: $23 + 47 = 70$
 답2: $23 + 47 = 70$
 답3: $23 + 47 = 71$ ✗
 ...
 답10: $23 + 47 = 70$

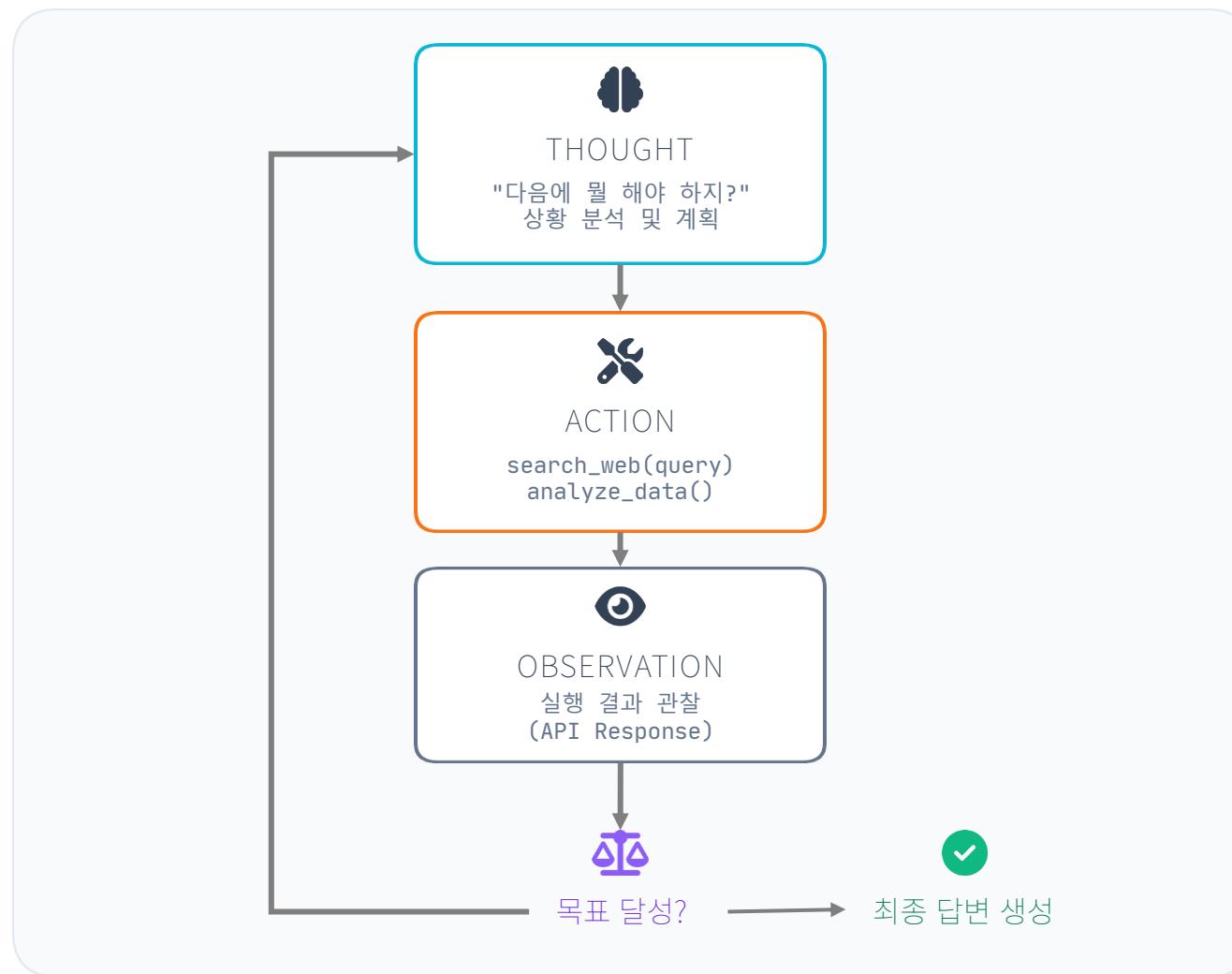
→ 정답 후보 중 '70'이 가장 많이 등장 → 최종 정답: 70

관련 논문: <https://arxiv.labs.arxiv.org/html/2201.11903>

관련 논문: <https://arxiv.org/abs/2203.11171>

ReAct 순환 구조 다이어그램

Reasoning + Acting = Autonomous



Reasoning (추론)

현재 상태와 목표를 비교하여 논리적인 다음 행동을 결정하는 단계입니다. LLM의 Chain of Thought 능력을 활용합니다.

Acting (행동)

외부 세계와 상호작용합니다. 검색, 계산, 데이터베이스 조회 등 도구를 실제로 실행합니다.

Observation (관찰)

행동의 결과를 그대로 받아들입니다. 이 정보는 프롬프트에 추가되어 다음 추론의 근거가 됩니다.

Loop or Finish

정보가 충분하면 종료(Answer), 부족하거나 오류가 있으면 다시 Thought 단계로 돌아가 재시도합니다.

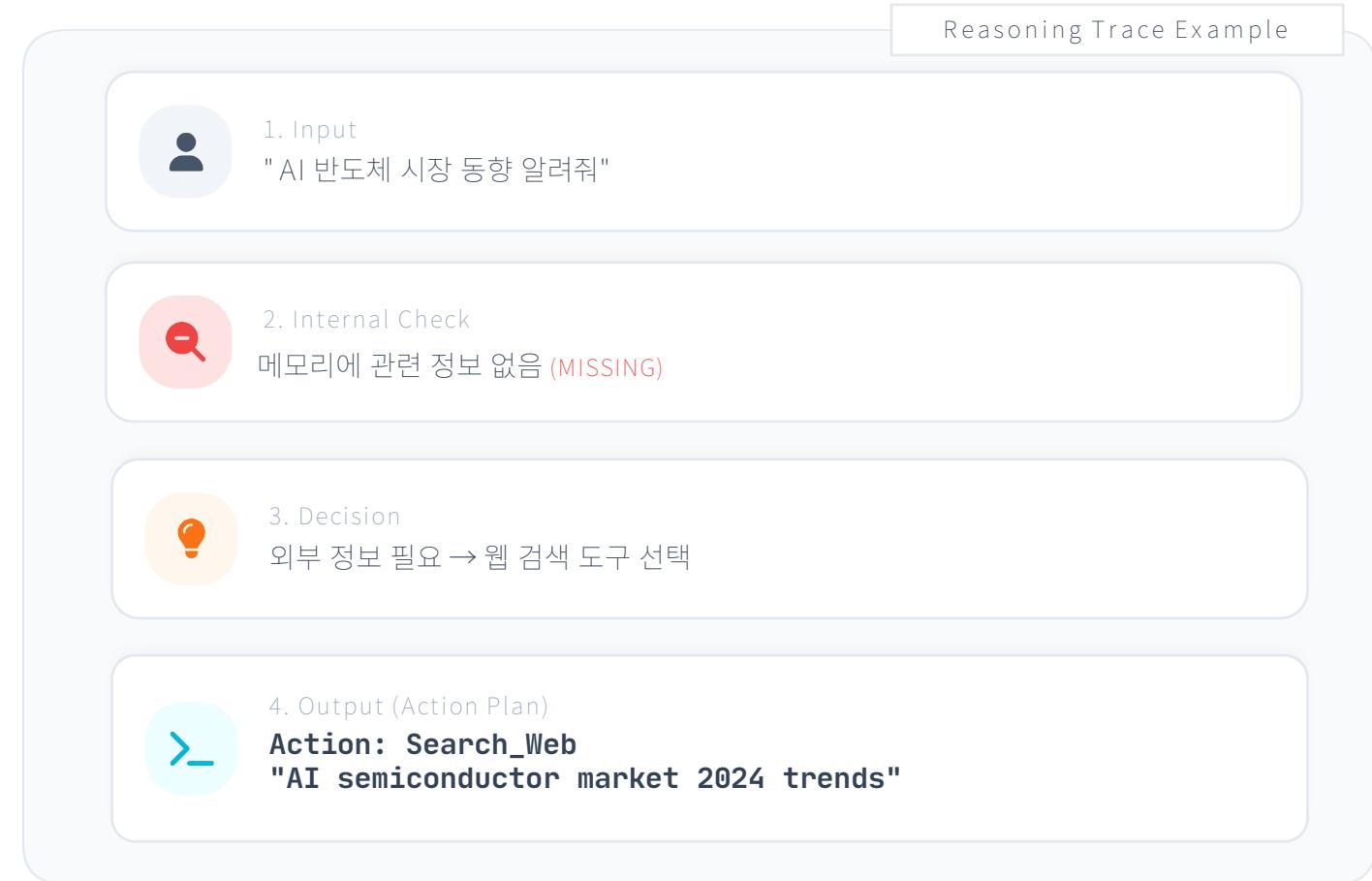
Thought - 추론 단계

상황 분석 및 행동 결정 프로세스

Role of 'Thought'

LLM이 현재 상황을 분석하고, 목표 달성을 위해 **다음에 무엇을 해야 할지** 논리적으로 결정하는 단계입니다.

- ✓ 사용자 의도 및 맥락 파악
- ✓ 가용 정보(Memory) 확인
- ✓ 최적의 도구(Tool) 선택



Action - 실행 단계

도구 실행 및 외부 상호작용

Execution Phase

추론(Thought) 단계를 통해 결정된 구체적인 행동을 수행합니다. 에이전트는 이 단계를 통해 외부 세계와 상호작용합니다.

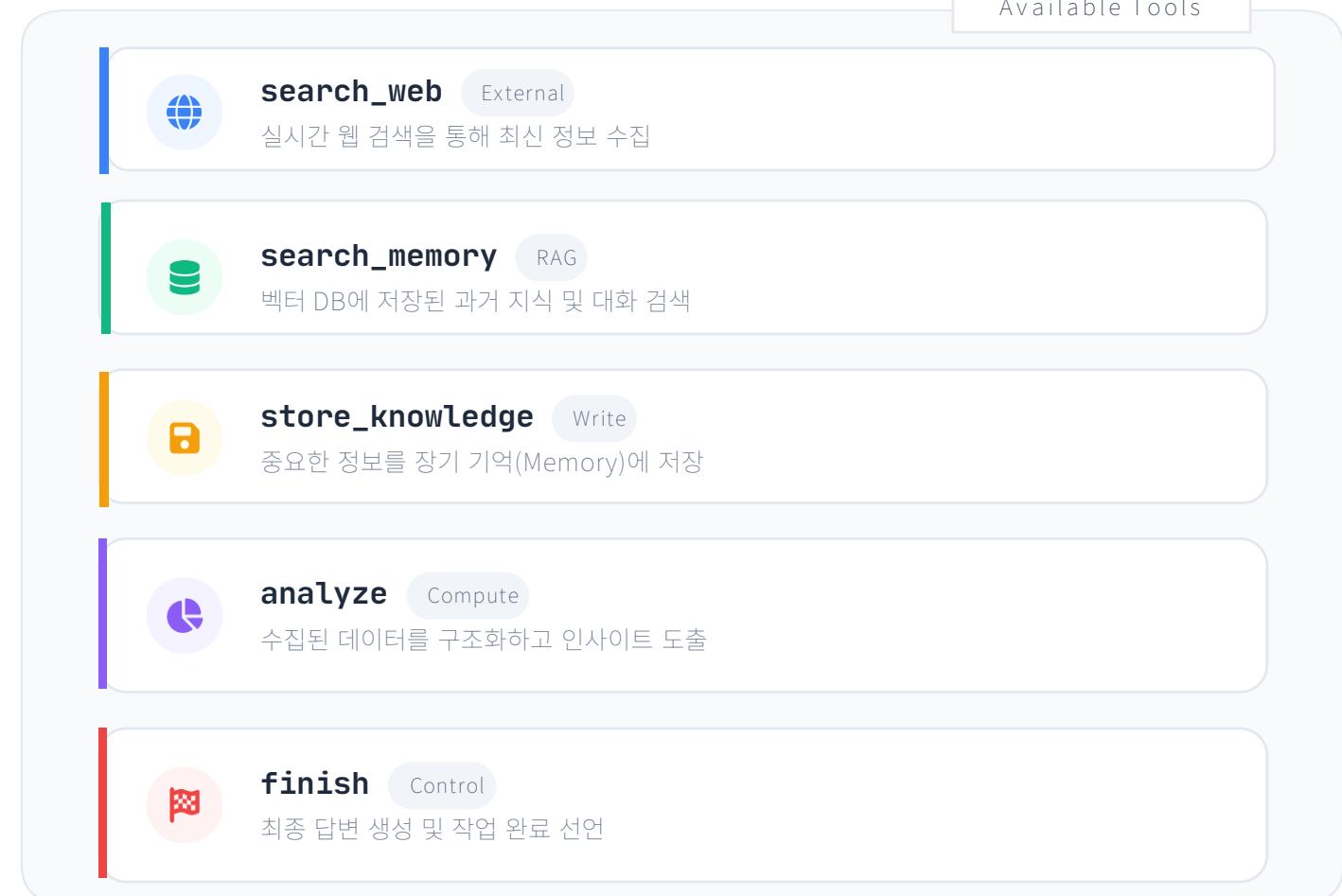
</> Action Format

LLM은 다음과 같은 엄격한 JSON 형식으로 도구 실행을 요청합니다.

JSON Output

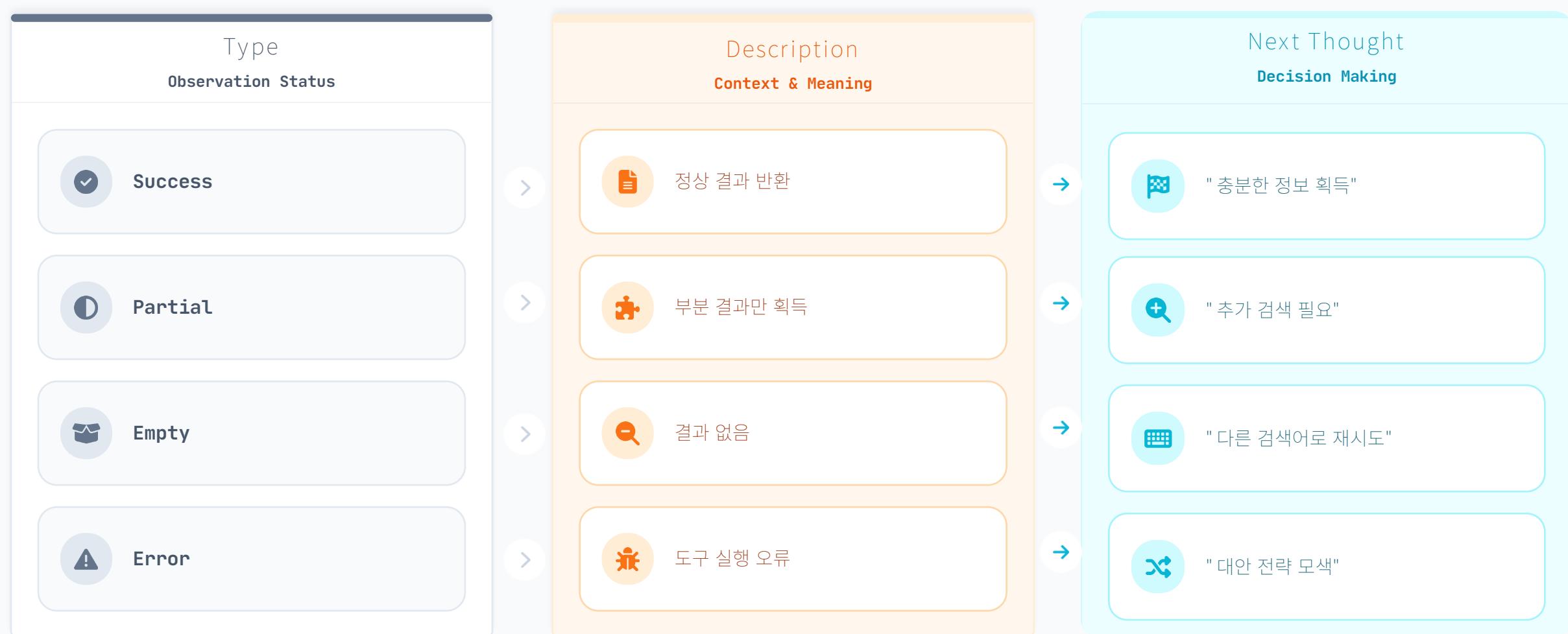
```
{  
  "action": "search_web",  
  "action_input": "AI semiconductor 2024"  
}
```

Available Tools



Observation – 관찰 단계

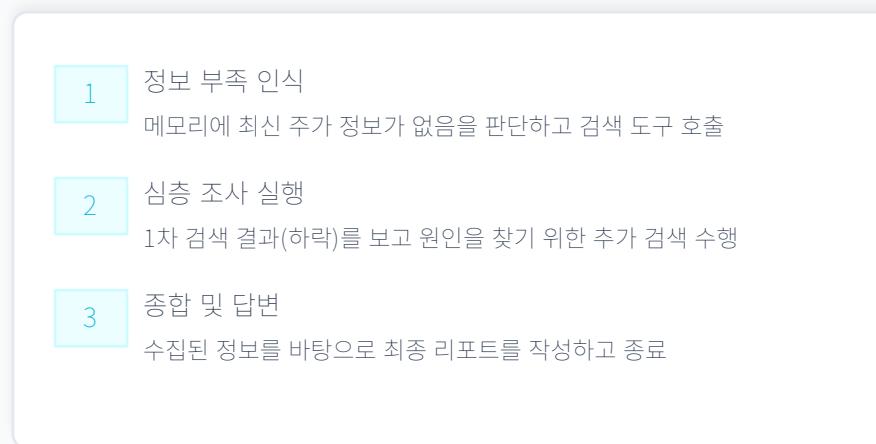
행동(Action)의 결과를 분석하여 다음 전략 수립



ReAct 실제 예시 (전체 흐름)

ReAct Loop Process

에이전트가 생각(Thought)하고 행동(Action)하며 관찰(Observation)하는 과정을 반복하여 답을 찾아냅니다.



Request: "테슬라 주가 동향과 원인 분석해줘"

```

react_agent_logs.log

# Iteration 1: 주가 정보 검색
Thought: 사용자가 테슬라 주가 동향과 원인을 묻고 있다. 2024년 최신 주가 정보가 필요하다.
Action: {
  "action": "search_web",
  "action_input": "Tesla stock price trend 2024"
}
Observation: 테슬라 주가는 연초 대비 약 15% 하락했다. 주요 원인으로는 전기차 수요 둔화와 판매량 감소가 꼽힌다...

# Iteration 2: 하락 원인 심층 조사
Thought: 주가 하락은 확인했다. 판매량 감소의 구체적인 이유와 경쟁 상황을 더 알아봐야 한다.
Action: {
  "action": "search_web",
  "action_input": "Tesla sales decline reasons 2024 competition"
}
Observation: 중국 BYD가 테슬라를 제치고 전기차 판매 1위를 차지했다. 가격 경쟁 심화가 주요인이다...

# Iteration 3: 최종 답변 생성
Thought: 주가 하락 폭과 주요 원인(BYD 경쟁, 수요 둔화)을 모두 파악했다. 이제 답변할 수 있다.
Action: {
  "action": "finish",
  "answer": "2024년 테슬라 주가는 약 15% 하락했습니다. 주요 원인은 1) BYD와의 경쟁 심화로 인한 판매량 감소, 2) 전반적인 전기차 수요 둔화입니다..."
}
  
```

자율 에이전트의 도전과제와 해결 전략

자율 실행 시스템의 안정성과 품질을 확보하는 방법

Infinite Loop 무한 루프 방지



Problem

- 동일한 생각이나 행동을 계속 반복하는 현상

Solution

- 최대 반복 횟수 제한 및 순환 패턴 감지

Implementation

```
</> LoopPrevention
```

Hallucination 환각 현상 제어



Problem

- 사실이 아닌 정보를 사실처럼 생성

Solution

- 결과 품질 자가 평가 및 근거(Source) 요구

Implementation

```
</> QualityManager
```

Cost / Efficiency 비용 및 효율성



Problem

- 과도한 API 호출로 인한 비용 및 속도 저하

Solution

- 조기 종료(Early Exit) 및 스텝 제한 설정

Implementation

```
</> should_stop()
```

03

SECTION

HANDS-ON

IMPLEMENTATION

HANDS-ON PART 1

Task Decomposition 구현

◎ Key Learning Points

- ✓ 복잡한 작업의 구조적 분해
- ✓ 서브태스크 의존성(Dependency) 관리
- ✓ TaskPlanner 클래스 구현
- ✓ LLM 기반 작업 계획 프롬프트

Keywords:

● Task Planning

● Subtask

● Dependency

● Priority

왜 작업 분해가 필요한가?

복잡한 작업 처리의 핵심: Divide and Conquer

User Request

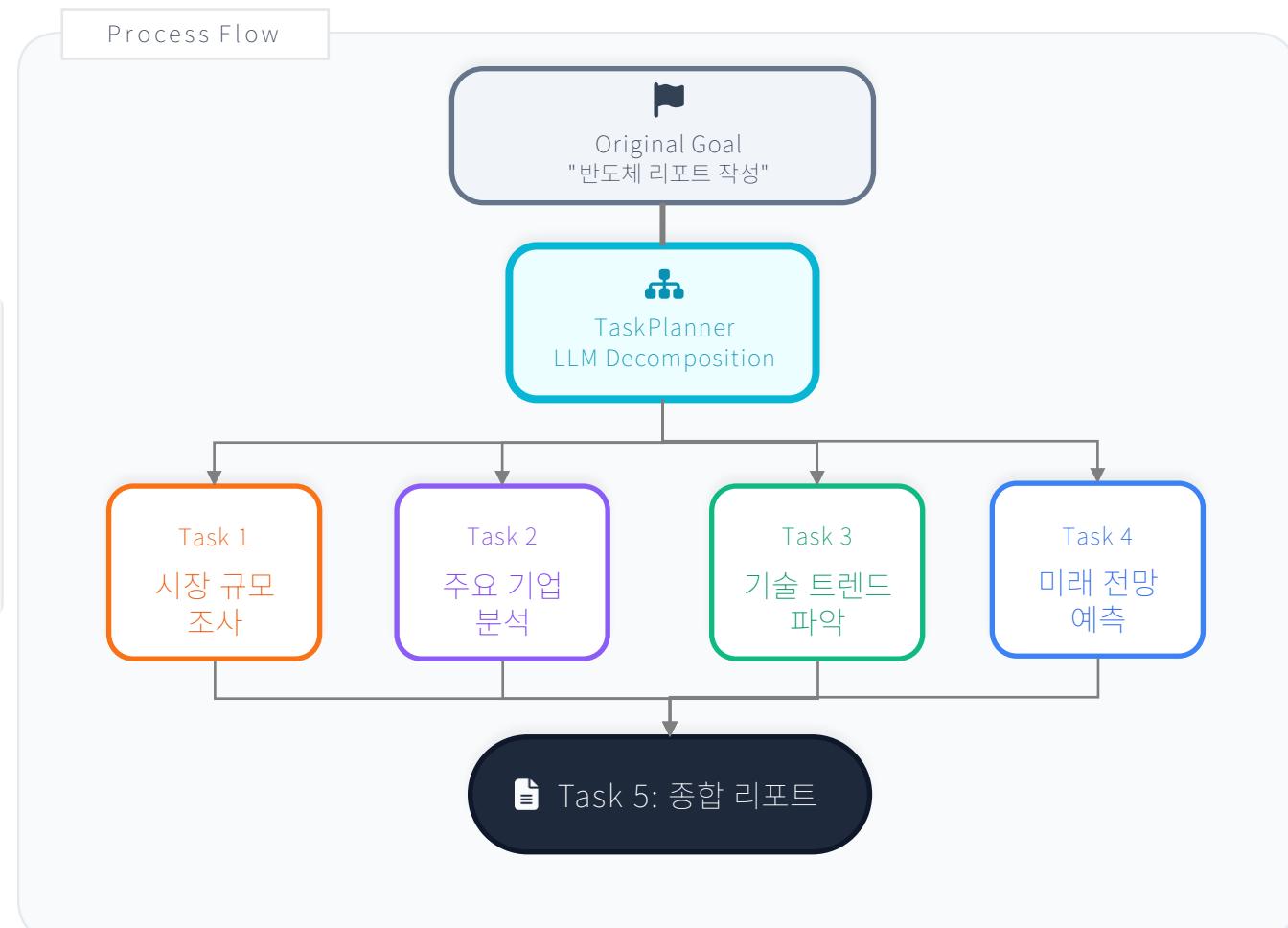
"AI 반도체 시장 종합 리포트 작성해줘"
단일 프롬프트로 처리하기엔 너무 광범위하고 복잡한 요청

⚠ 단일 처리 시 문제점

- ✖ 광범위한 검색: 너무 많은 정보를 한 번에 검색하여 정확도 하락
- ✖ 중요 내용 누락: 특정 세부 사항(기업, 전망 등)을 놓칠 가능성
- ✖ 일관성 부족: 논리적 구조 없이 나열식 답변 생성

→ 해결책: Task Decomposition

복잡한 목표를 실행 가능한 작은 단위(Subtask)로 쪼개어 순차적 또는 병렬적으로 처리합니다.



Part 1 작업 목표

☰ 구현 체크리스트

이번 섹션에서 구현할 핵심 기능 목록입니다.

- TaskStatus Enum (태스크 상태 관리) ENUM

- Subtask Dataclass (데이터 구조) DATA

- TaskPlanner 클래스 (LLM 작업 분해) CLASS

- 의존성 관리 (그래프 처리) GRAPH

- 우선순위 로직 (실행 순서 결정) LOGIC

- TASK_DECOMPOSE_PROMPT PROMPT

```
src/task_planner.py
```

```
class TaskPlanner:  
    def __init__(self, client: OpenAI):  
        self.client = client  
        self.tasks: List[Subtask] = []  
  
    def decompose(self, goal: str) -> List[Subtask]:  
        """LLM으로 작업 분해"""  
        pass  
  
    def get_next_task(self) -> Optional[Subtask]:  
        """실행 가능한 다음 작업"""  
        pass  
  
    def update_status(self, task_id, status, result=None):  
        """상태 업데이트"""  
        pass  
  
    def is_complete(self) -> bool:  
        """완료 여부"""  
        pass
```

작업 분해 – 구조 설계

TASK_DECOMPOSE_PROMPT = """당신은 작업 분해 전문가입니다.

규칙

1. 각 서브태스크는 독립 실행 가능
2. 명확한 완료 조건
3. 의존성 명시
4. 3-7개 서브태스크

목표

{goal}

JSON 출력

```
{"subtasks": [  
    {"id": "task_1", "description": "...", "priority": 1, "dependencies": []}  
]}
```

LLM 프롬프트 규칙

작업 분해 시 LLM에게 강제할 핵심 규칙입니다.



독립 실행 가능 (Standalone)

각 서브태스크는 이전 작업의 결과만 있으면 독립적으로 수행 가능해야 함



명확한 완료 조건

AI가 작업의 성공/실패 여부를 객관적으로 판단할 수 있어야 함



의존성 명시 (Dependency)

실행 순서를 결정하기 위해 선행되어야 할 작업 ID를 명시



적정 개수 유지

복잡도 관리를 위해 3~7개의 서브태스크로 분할 권장

Subtask 데이터 구조

❖ 컴포넌트 상세 명세

```
class TaskStatus(Enum):
    PENDING = "pending"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"
    SKIPPED = "skipped"
```

❖ Subtask Dataclass (데이터 구조)

```
@dataclass
class Subtask:
    id: str
    description: str
    priority: int
    dependencies: List[str] =
        field(default_factory=list)
    status: TaskStatus = TaskStatus.PENDING
    result: Optional[str] = None
    attempts: int = 0
    max_attempts: int = 3
```

우선 순위 결정 로직



src/task_planner.py

```
def get_next_task(self) -> Optional[Subtask]:
    executable = [
        task for task in self.tasks
        if task.status == TaskStatus.PENDING
        and self._dependencies_met(task)
        and task.attempts < task.max_attempts
    ]
    if not executable:
        return None
    return sorted(executable, key=lambda t: t.priority)[0]
```



src/task_planner.py

```
def _dependencies_met(self, task: Subtask) -> bool:
    for dep_id in task.dependencies:
        dep_task = self._get_task_by_id(dep_id)
        if not dep_task or dep_task.status != TaskStatus.COMPLETED:
            return False
    return True
```

조건 1: `task.status == TaskStatus.PENDING`

의미: 아직 실행되지 않은 태스크만 선택

제외: IN_PROGRESS, COMPLETED, FAILED, SKIPPED

조건 2: `self._dependencies_met(task)`

의미: 이 태스크가 의존하는 모든 태스크가 완료되었는지 확인

예시: task_4가 [task_2, task_3]에 의존하면, 둘 다 COMPLETED여야 함

조건 3: `task.attempts < task.max_attempts`

의미: 재시도 횟수가 최대치를 넘지 않았는지 확인

예시: max_attempts=3이면, 3번 실패한 태스크는 제외

의존하는 태스크 ID 순회

ID로 태스크 객체 찾기

하나라도 미완료면 False

모두 완료면 True

우선 순위 결정 로직 (동작 예시)

1 상황 설정 (Scenario)

```
# 태스크 목록 상태
task_1:COMPLETED, prio=1
task_2:PENDING, prio=2, dep=[1]
task_3:PENDING, prio=2, dep=[1]
task_4:PENDING, prio=3, dep=[2,3]
task_5:PENDING, prio=1, att=3(MAX)
```

3 정렬 및 선택 (Sorting)

Executable Tasks

task_2
Prio=2task_3
Prio=2

↓ Sort by Priority (Ascending)

Selected
task_2우선순위 동일(2)
리스트 순서 유지
First Item 반환

2 필터링 과정 (Filtering)

태스크	1. PENDING	2. 의존성	3. 시도횟수	결과
task_1	Completed	-	-	제외
task_2	Pass	task_1 완료	0 < 3	✓ 포함
task_3	Pass	task_1 완료	0 < 3	✓ 포함
task_4	Pass	2,3 미완료	Pass	제외
task_5	Pass	Pass	3 >= 3	제외

Logic Summary

1. 필터링 (AND 조건)

PENDING 상태 AND 의존성 완료 AND 시도횟수 잔여

2. 정렬 (Sorting)

priority 속성 오름차순 (낮은 숫자가 높은 우선순위)

3. 선택 (Selection)

정렬된 리스트의 첫 번째([0]) 요소 반환

실습 Part1 구현 프롬프트 - 1

현재 프로젝트 구조를 확인하고 4주차 작업을 위한 준비를 해주세요.

[확인 사항]

1. src/ 폴더에 기존 파일들이 있는지 확인

- conversation_manager.py (Week 1)
- search_agent.py (Week 2)
- memory_manager.py (Week 3)

2. config/prompts.py 파일 구조 확인

3. 필요시 tests/ 폴더 생성

[출력]

- 현재 프로젝트 구조

- 4주차에 추가될 파일 목록 확인

실습 Part1 구현 프롬프트 - 2

src/task_planner.py 파일을 새로 생성하고
TaskStatus Enum과 Subtask dataclass를 구현해주세요.

[파일 위치]
src/task_planner.py

[구현 내용]

1. Import 문

```
from enum import Enum
from dataclasses import dataclass, field
from typing import List, Optional, Dict, Any
from openai import OpenAI
import json
import logging

logger = logging.getLogger(__name__)
```

2. TaskStatus Enum

- PENDING: 대기 중
- IN_PROGRESS: 실행 중
- COMPLETED: 완료
- FAILED: 실패
- SKIPPED: 건너뜀

3. Subtask Dataclass

```
@dataclass
class Subtask:
    id: str          # 태스크 고유 ID (예: "task_1")
    description: str # 태스크 설명
    priority: int   # 우선순위 (1이 가장 높음)
    dependencies: List[str] = field(default_factory=list) # 의존 태스크 ID 목록
    status: TaskStatus = TaskStatus.PENDING
    result: Optional[str] = None    # 실행 결과
    attempts: int = 0              # 시도 횟수
    max_attempts: int = 3         # 최대 시도 횟수
```

[테스트 코드 (주석)]

```
# from src.task_planner import TaskStatus, Subtask
#
# # TaskStatus 테스트
# print(TaskStatus.PENDING.value) # "pending"
#
# # Subtask 테스트
# task = Subtask(
#     id="task_1",
#     description="시장 규모 조사",
#     priority=1,
#     dependencies=[]
# )
# print(f"Task: {task.id}, Status: {task.status.value}")
```

실습 Part1 구현 프롬프트 - 3

src/task_planner.py에 TaskPlanner 클래스의 기본 구조를 추가해주세요.

[추가 위치]
Subtask 클래스 아래에 추가

[구현 내용]

```
class TaskPlanner:
    """LLM을 활용한 작업 분해 플래너"""

    def __init__(self, client: OpenAI):
        """
        Args:
            client: OpenAI 클라이언트
        """
        self.client = client
        self.tasks: List[Subtask] = []
        self.original_goal: str = ""
        logger.info("TaskPlanner initialized")

    def decompose(self, goal: str) -> List[Subtask]:
        """목표를 서브태스크로 분해"""
        pass

    def get_next_task(self) -> Optional[Subtask]:
        """실행 가능한 다음 태스크 반환"""
        pass

    def update_status(self, task_id: str, status: TaskStatus, result: Optional[str] = None) -> bool:
        """태스크 상태 업데이트"""
        pass
```

```
def is_complete(self) -> bool:
    """모든 태스크 완료 여부 확인"""
    pass

def _dependencies_met(self, task: Subtask) -> bool:
    """의존성 충족 여부 확인"""
    pass

def _get_task_by_id(self, task_id: str) -> Optional[Subtask]:
    """ID로 태스크 찾기"""
    pass

def visualize(self) -> None:
    """현재 태스크 상태 시각화"""
    pass
```

[확인 사항]
- 클래스 기본 구조만 생성 (메서드 내용은 pass)
- docstring 포함
- 타입 힌트 포함

실습 Part1 구현 프롬프트 - 4

config/prompts.py 파일에 작업 분해를 위한 프롬프트 템플릿을 추가해주세요.

[추가 위치]

config/prompts.py 파일 하단에 추가

[추가할 내용]

```
# ===== Week 4: Task Decomposition =====
```

```
TASK_DECOMPOSE_PROMPT = """당신은 복잡한 작업을 분해하는 전문가입니다.
```

역할

주어진 목표를 독립적으로 실행 가능한 서브태스크로 분해합니다.

규칙

1. 각 서브태스크는 독립적으로 실행 가능해야 합니다
2. 각 태스크는 명확한 완료 조건이 있어야 합니다
3. 태스크 간 의존성을 명시해야 합니다
4. 3-7개의 서브태스크로 분해합니다
5. 마지막 태스크는 항상 "결과 종합 및 정리"여야 합니다

목표
{goal}

출력 형식 (JSON)
반드시 다음 형식의 유효한 JSON만 출력하세요.

```
{
  "subtasks": [
    {
      "id": "task_1",
      "description": "첫 번째 작업 설명",
      "priority": 1,
      "dependencies": []
    },
    {
      "id": "task_2",
      "description": "두 번째 작업 설명",
      "priority": 2,
      "dependencies": ["task_1"]
    }
  ]
}
```

실습 Part1 구현 프롬프트 - 5

src/task_planner.py의 TaskPlanner 클래스에서 decompose() 메서드를 구현해주세요.

[Import 추가]
파일 상단에 추가:
from config.prompts import TASK_DECOMPOSE_PROMPT

[구현 내용]

```
def decompose(self, goal: str) -> List[Subtask]:
    """목표를 서브태스크로 분해"""
    self.original_goal = goal
    self.tasks = []

    try:
        response = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[
                {"role": "system", "content": "작업 분해 전문가입니다. JSON만 출력합니다."},
                {"role": "user", "content": TASK_DECOMPOSE_PROMPT.format(goal=goal)}
            ],
            temperature=0.3,
            response_format={"type": "json_object"}
        )
    
```

```
        content = response.choices[0].message.content
        data = json.loads(content)

        for task_data in data.get("subtasks", []):
            subtask = Subtask(
                id=task_data["id"],
                description=task_data["description"],
                priority=task_data["priority"],
                dependencies=task_data.get("dependencies", [])
            )
            self.tasks.append(subtask)

        logger.info(f"Decomposed '{goal}' into {len(self.tasks)} subtasks")
        return self.tasks

    except json.JSONDecodeError as e:
        logger.error(f"Failed to parse LLM response: {e}")
        fallback_task = Subtask(id="task_1", description=goal, priority=1)
        self.tasks = [fallback_task]
        return self.tasks
```

실습 Part1 구현 프롬프트 - 6

src/task_planner.py의 TaskPlanner 클래스에서
헬퍼 메서드들을 구현해주세요.

[구현할 메서드]

```
def _get_task_by_id(self, task_id: str) -> Optional[Subtask]:  
    """ID로 태스크 찾기"""  
    for task in self.tasks:  
        if task.id == task_id:  
            return task  
    return None  
  
def _dependencies_met(self, task: Subtask) -> bool:  
    """태스크의 의존성이 모두 충족되었는지 확인"""  
    for dep_id in task.dependencies:  
        dep_task = self._get_task_by_id(dep_id)  
        if not dep_task:  
            logger.warning(f"Dependency {dep_id} not found for {task.id}")  
            return False  
        if dep_task.status != TaskStatus.COMPLETED:  
            return False  
    return True
```

실습 Part1 구현 프롬프트 - 7

src/task_planner.py의 TaskPlanner 클래스에서
get_next_task()와 update_status() 메서드를 구현해주세요.

[구현 내용]

```
def get_next_task(self) -> Optional[Subtask]:
    """실행 가능한 다음 태스크 반환"""
    executable = [
        task for task in self.tasks
        if task.status == TaskStatus.PENDING
        and self._dependencies_met(task)
        and task.attempts < task.max_attempts
    ]

    if not executable:
        # 재시도 가능한 실패 태스크 확인
        retryable = [
            task for task in self.tasks
            if task.status == TaskStatus.FAILED
            and task.attempts < task.max_attempts
            and self._dependencies_met(task)
        ]
        if retryable:
            task = sorted(retryable, key=lambda t: t.priority)[0]
            task.status = TaskStatus.PENDING
            return task
        return None

    next_task = sorted(executable, key=lambda t: t.priority)[0]
    next_task.status = TaskStatus.IN_PROGRESS
    next_task.attempts += 1
```

```
logger.info(f"Next task: {next_task.id} (attempt {next_task.attempts})")
return next_task

def update_status(self, task_id: str, status: TaskStatus, result: Optional[str] = None) -> bool:
    """태스크 상태 업데이트"""
    task = self._get_task_by_id(task_id)
    if not task:
        logger.error(f"Task not found: {task_id}")
        return False

    old_status = task.status
    task.status = status

    if result is not None:
        task.result = result

    logger.info(f"Task {task_id}: {old_status.value} → {status.value}")
    return True
```

실습 Part1 구현 프롬프트 - 8

src/task_planner.py의 TaskPlanner 클래스에서
is_complete()와 visualize() 메서드를 구현해주세요.

[구현 내용]

```
def is_complete(self) -> bool:
    """모든 태스크 완료 여부 확인"""
    if not self.tasks:
        return True

    for task in self.tasks:
        if task.status not in [TaskStatus.COMPLETED,
TaskStatus.SKIPPED]:
            return False
    return True

def visualize(self) -> None:
    """현재 태스크 상태를 시각적으로 출력"""
    status_icons = {
        TaskStatus.PENDING: "hourglass",
        TaskStatus.IN_PROGRESS: "refresh",
        TaskStatus.COMPLETED: "checkmark",
        TaskStatus.FAILED: "cross",
        TaskStatus.SKIPPED: "skip"
    }

    print("\n" + "*60)
    print(f"📋 Task Plan: {self.original_goal[:50]}...")
    print("*60)
```

```
for task in self.tasks:
    icon = status_icons.get(task.status, "?")
    deps_str = f" (depends: {' '.join(task.dependencies)})" if task.dependencies else ""
    print(f"\n{icon} [{task.id}] {task.description}")
    print(f"  Priority: {task.priority} | Status: {task.status.value}{deps_str}")

    if task.result:
        result_preview = task.result[:100] + "..." if len(task.result) > 100 else task.result
        print(f"  Result: {result_preview}")

    completed = sum(1 for t in self.tasks if t.status == TaskStatus.COMPLETED)
    total = len(self.tasks)
    progress = (completed / total * 100) if total > 0 else 0
    print("\n" + "-"*60)
    print(f"📊 Progress: {completed}/{total} ({progress:.1f}%)")
    print("*60 + "\n")

def get_summary(self) -> Dict[str, Any]:
    """현재 상태 요약 반환"""
    return {
        "goal": self.original_goal,
        "total_tasks": len(self.tasks),
        "completed": sum(1 for t in self.tasks if t.status == TaskStatus.COMPLETED),
        "failed": sum(1 for t in self.tasks if t.status == TaskStatus.FAILED),
        "pending": sum(1 for t in self.tasks if t.status == TaskStatus.PENDING),
        "in_progress": sum(1 for t in self.tasks if t.status == TaskStatus.IN_PROGRESS),
        "is_complete": self.is_complete()
    }
```

실습 Part1 구현 프롬프트 - 9

tests/test_task_planner.py 파일을 생성하고 TaskPlanner의 단위 테스트를 작성해주세요.

[파일 위치]
tests/test_task_planner.py

[구현 내용]

```
"""TaskPlanner 단위 테스트"""
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))))

from openai import OpenAI
from src.task_planner import TaskPlanner, TaskStatus, Subtask

def test_subtask_creation():
    """Subtask 생성 테스트"""
    print("\n" + "*50)
    print("TEST 1: Subtask Creation")
    print("*50)

    task = Subtask(id="task_1", description="테스트 태스크",
priority=1)

    assert task.id == "task_1"
    assert task.status == TaskStatus.PENDING
    assert task.attempts == 0
    print("✓ 기본 Subtask 생성 성공")
```

```
task2 = Subtask(id="task_2", description="의존성 태스크", priority=2,
dependencies=["task_1"])
assert "task_1" in task2.dependencies
print("✓ 의존성 Subtask 생성 성공")

return True

def test_task_decomposition():
    """작업 분해 테스트"""
    print("\n" + "*50)
    print("TEST 2: Task Decomposition")
    print("*50)

    client = OpenAI()
    planner = TaskPlanner(client)

    tasks = planner.decompose("전기차 시장 동향 분석")

    assert len(tasks) >= 3
    assert len(tasks) <= 7
    print(f"✓ {len(tasks)}개 태스크로 분해됨")

    for task in tasks:
        assert task.id.startswith("task_")
        assert len(task.description) > 0
        print(f" - {task.id}: {task.description[:40]}...")

    return True
..... (전체 내용은 별첨 참조)
```

Part 1 단위 테스트

💡 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> python tests/test_task_planner.py

⚙️ 단위 테스트 항목

1. Subtask Creation Test

Subtask 기본 생성, 의존성(dependencies) 포함 생성

2. Decomposition test

Decompose("전기차 시장 동향 분석") → 3~7개 태스크, id/description 검증

3. Task Execution Flow Test

Decompose → get_next_task → update_status(COMPLETED) 반복 후 is_complete() 및 visualize() 확인

4. Retry_mechanism Test

FAILED 태스크 재시도 3회 후 max_attempts 초과 시 get_next_task()가 None 반환하는지 확인

Part 1 단위 테스트 (성공 시 화면)

TEST 1: Subtask Creation

- 기본 Subtask 생성 성공
- 의존성 Subtask 생성 성공

TEST 2: Task Decomposition

- 6개 태스크로 분해됨
 - task_1: 전기차 시장에 대한 기본 데이터 수집...
 - task_2: 전기차 제조업체 및 모델 분석...
 - task_3: 전기차 판매 및 성장률 데이터 분석...
 - task_4: 전기차 관련 정책 및 규제 조사...
 - task_5: 소비자 선호도 및 시장 트렌드 조사...
 - task_6: 결과 종합 및 정리...

TEST 3: Dependency Management

- 의존성 없는 태스크 우선 선택
- 미충족 의존성 정상 감지
- 충족 의존성 정상 감지

...

... (중략)

- 전체 실행 흐름 테스트 성공

TEST 5: Retry Mechanism

- 1차 시도
- 2차 시도
- 3차 시도
- max_attempts 초과 후 재시도 차단

테스트 결과 요약

- Subtask Creation: PASS
- Task Decomposition: PASS
- Dependency Management: PASS
- Execution Flow: PASS
- Retry Mechanism: PASS

총 결과: 5/5 통과

04

SECTION

HANDS-ON

IMPLEMENTATION

HANDS-ON PART 2

자율 실행 오케스트레이터 구현

◎ Key Implementation Goals

- ✓ ReActEngine (추론-행동 순환)
- ✓ LoopPrevention (무한 루프 방지)
- ✓ QualityManager (품질 평가 및 재시도)
- ✓ AutonomousOrchestrator (전체 통합)

Keywords:

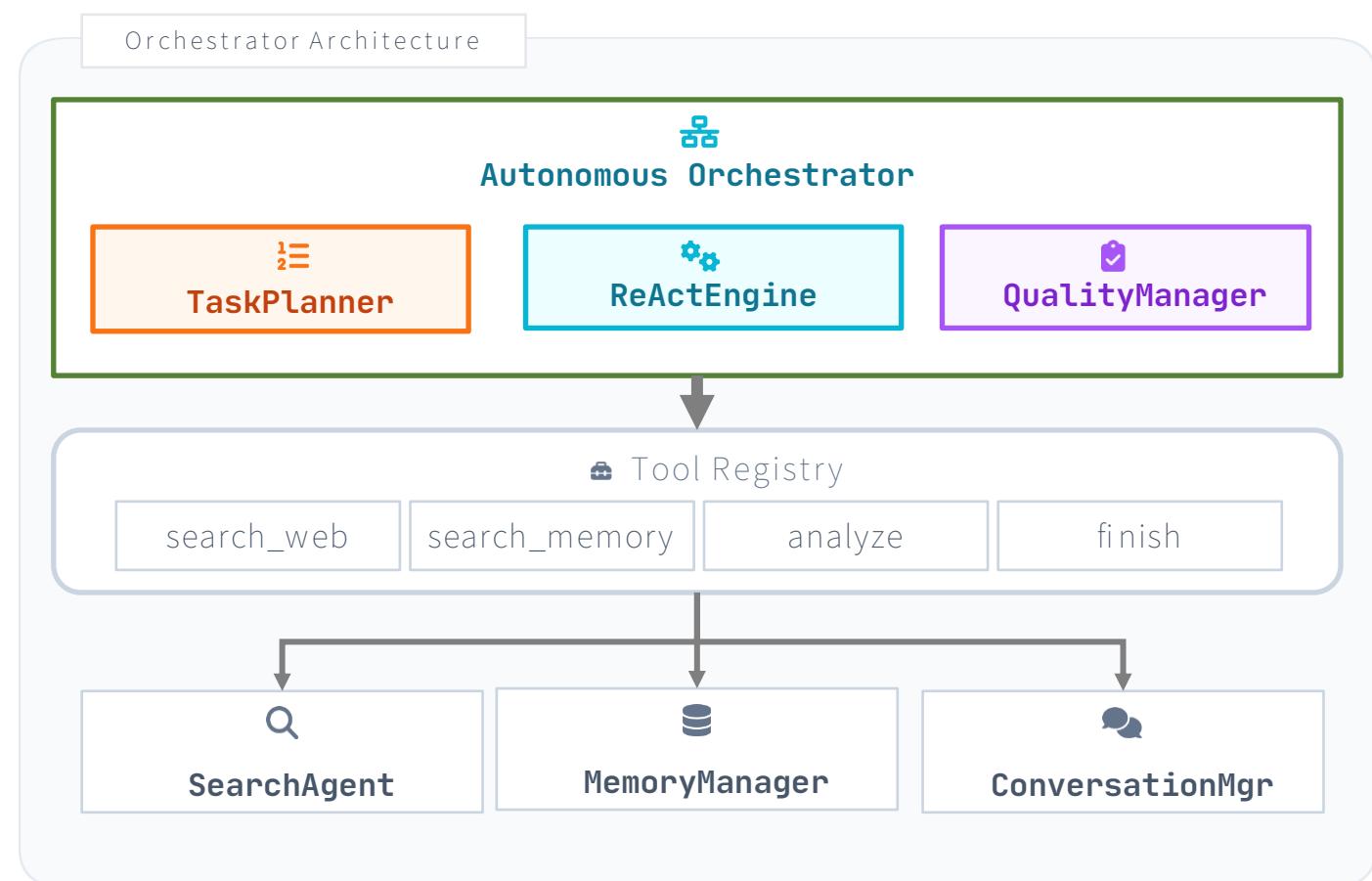
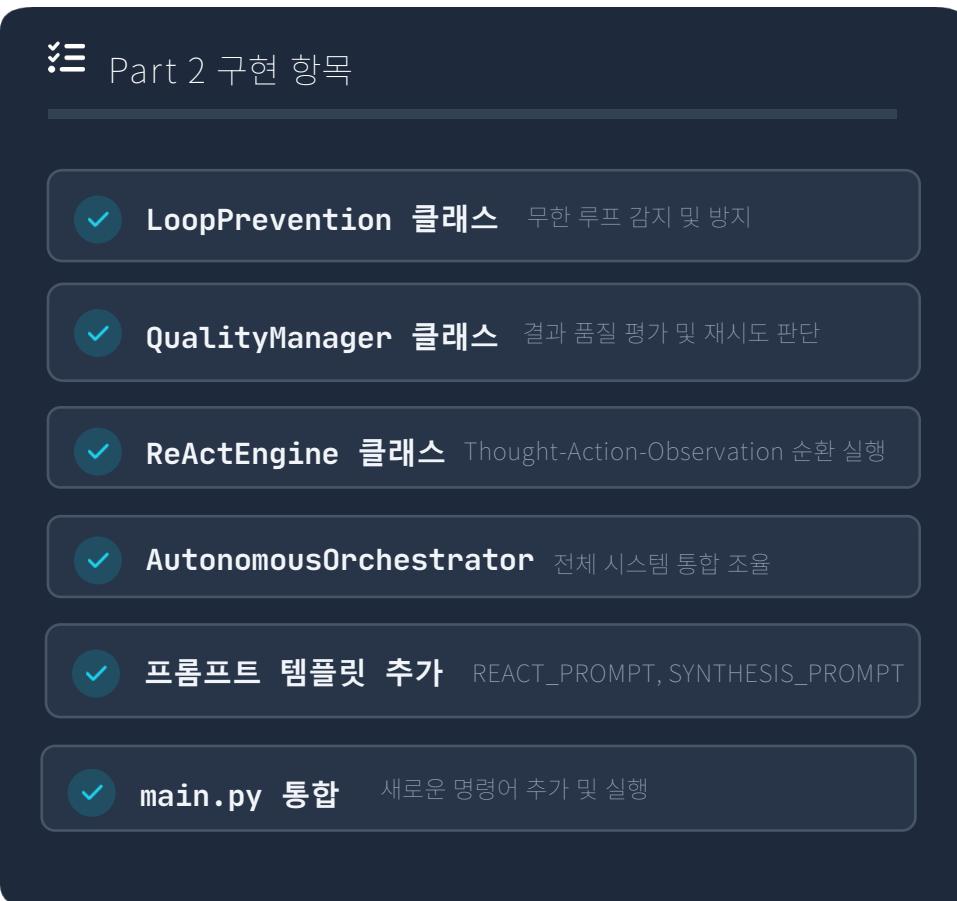
● ReActEngine

● Self-Reflection

● Quality Evaluation

실습 Part2 작업 목표

자율 실행 오케스트레이터 및 시스템 통합



ReAct Engine 클래스

자율 실행 순환 루프 구현

Data Structure

에이전트의 사고와 행동 기록을 관리하는
ReActStep 데이터 구조체

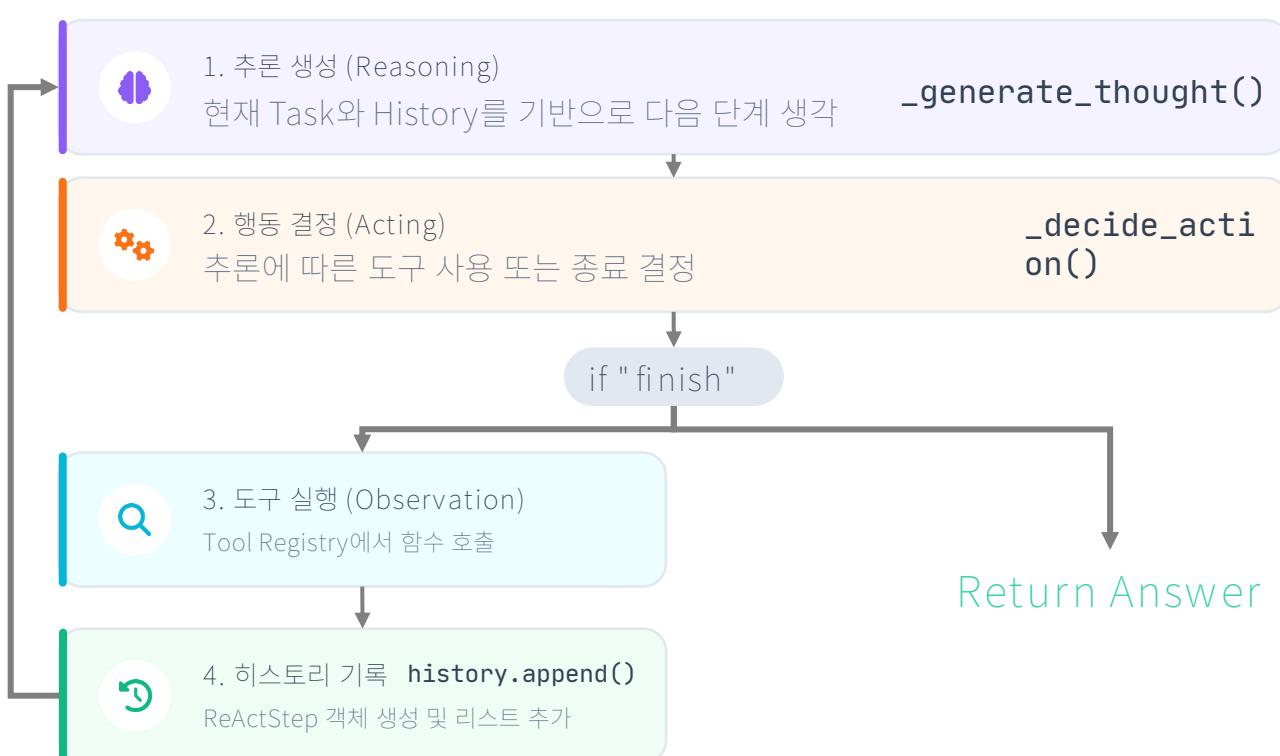
iteration 반복 횟수 (int)
현재 몇 번째 루프인지 추적하여 무한 루프 방지

thought 추론 내용 (str)
LLM이 생성한 현재 상황 분석 및 계획

action 행동 및 입력 (Dict)
실행할 도구 이름과 파라미터 {name, input}

observation 관찰 결과 (str)
도구 실행 후 반환된 실제 결과값

run() Execution Flow



⚠ Max Iterations 초과 시:

`_force_finish()`

ReAct 프롬프트 템플릿

에이전트의 사고 방식과 행동 양식을 정의

핵심 구성 요소

LLM이 ReAct 패턴을 정확히 수행하도록 강제하는 시스템 프롬프트

-  도구(Tools) 정의
사용 가능한 도구 목록과 각 도구의 용도를 명확히 기술하여 Hallucination 방지
-  출력 형식 (JSON)
Thought-Action 구조와 JSON 포맷을 엄격히 지정하여 파싱 에러 최소화
-  동적 컨텍스트
현재 작업(Task)과 실행 히스토리(History)를 런타임에 주입

```
config/prompts.py
```

```
REACT_SYSTEM_PROMPT = """당신은 리서치 어시스턴트입니다.

## 도구
- search_web: 웹 검색
- search_memory: 메모리 검색
- store_knowledge: 지식 저장
- analyze: 정보 분석
- finish: 작업 완료

## 출력 형식
Thought: [현재 상황 분석]
Action: {"action": "도구명", "action_input": "입력값"}

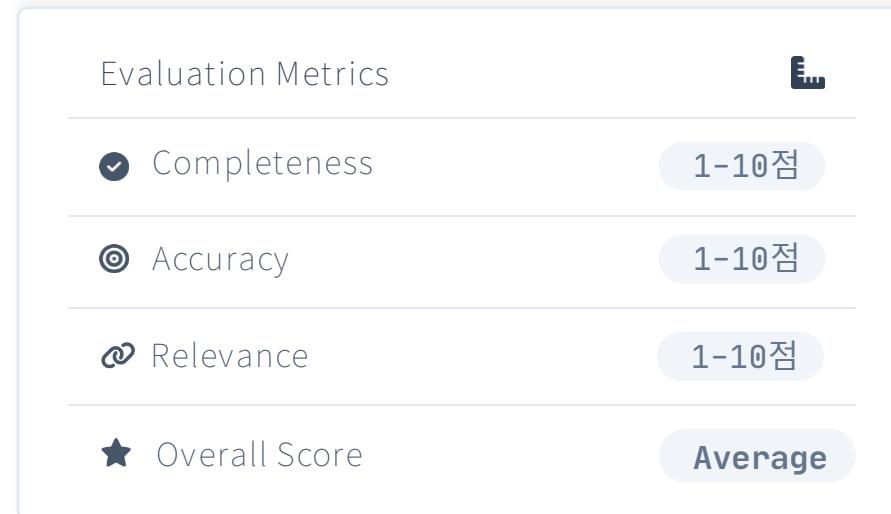
완료 시:
Thought: [완료 이유]
Action: {"action": "finish", "answer": "최종 답변"}

## 현재 작업: {task}
## 히스토리: {history}
"""
```

Self-Reflection 메커니즘

QualityManager

LLM을 심판(Judge)으로 활용하여 결과물을 정량적으로 평가하고 피드백을 생성합니다.



Retry Condition

만약 overall < 7 이면,
피드백을 포함하여 작업을 재시도합니다.

에이전트 스스로 결과 품질을 평가하고 재시도를 결정

```

src/quality_manager.py

class QualityManager:
    def __init__(self, client: OpenAI):
        self.client = client

    def evaluate(self, task: str, result: str) -> Dict:
        prompt = f"""작업: {task}
결과: {result}
평가 (1-10점):
- completeness
- accuracy
- relevance
JSON: {{'completeness': N, 'accuracy': N, 'relevance': N,
"overall": 평균, "feedback": "개선점"}}"""
        response = self.client.chat.completions.create(...)
        return json.loads(response.choices[0].message.content)

    def should_retry(self, evaluation: Dict) -> bool:
        return evaluation.get("overall", 0) < 7

```

무한 루프 방지 전략

에이전트의 오작동 및 리소스 낭비 방지

LoopPrevention 클래스

에이전트가 동일한 생각이나 행동을 반복하며 갇히는 상황 (Infinite Loop)을 실시간으로 감지하고 차단



Thought 해시 중복 감지

LLM이 생성한 '생각(Thought)' 문자열을 해싱하여 이전에 동일한 사고 과정이 있었는지 확인합니다.



Action 반복 감지

동일한 도구와 입력값(Action Input)을 3회 이상 연속으로 호출하는 경우를 감지합니다.



트립와이어(Tripwire)

루프가 감지되면 즉시 실행을 중단(Force Finish)하거나 에러를 반환하여 비용을 절감합니다.

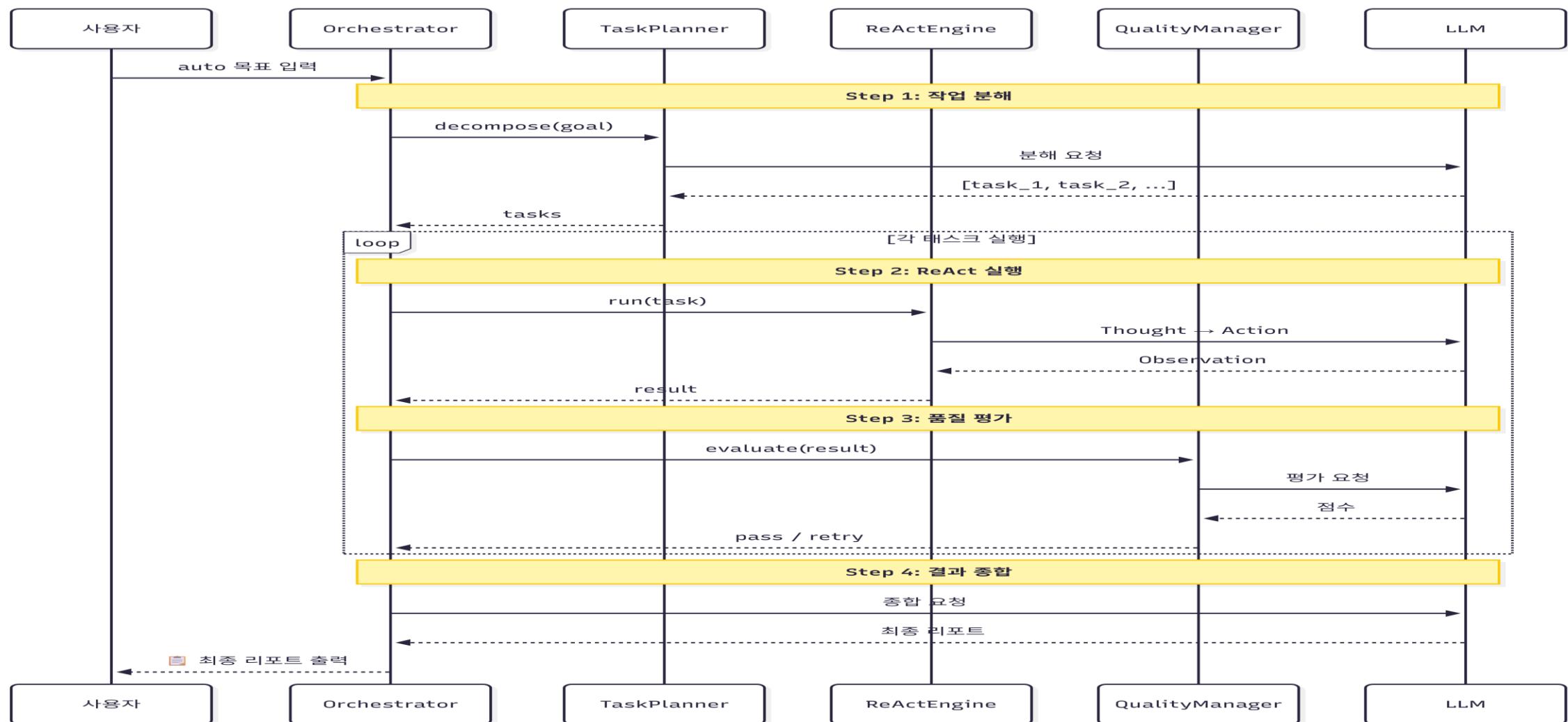


src/loop_prevention.py

```
class LoopPrevention:
    def __init__(self):
        self.action_history: List[str] = []
        self.thought_hashes: set = set()

    def check_loop(self, thought: str, action: Dict) -> bool:
        # 동일 생각 반복
        thought_hash = hash(thought[:100])
        if thought_hash in self.thought_hashes:
            return True
        self.thought_hashes.add(thought_hash)
        # 동일 액션 3회 연속
        action_str = f'{action["action"]}:{action.get("action_input", "")[:50]}'
        self.action_history.append(action_str)
        if len(self.action_history) >= 3:
            if len(set(self.action_history[-3:])) == 1:
                return True
            return False
        return False
```

실행 흐름 시퀀스 다이어그램



실습 Part2 구현 프롬프트 - 1

src/loop_prevention.py 파일을 새로 생성하고
무한 루프 방지를 위한 LoopPrevention 클래스를 구현해주세요.

[파일 위치]

src/loop_prevention.py

[구현 내용]

....

무한 루프 방지 모듈

ReAct 실행 중 무한 루프를 감지하고 방지합니다.

....

```
from typing import List, Dict, Any, Set
from dataclasses import dataclass, field
import logging
```

```
logger = logging.getLogger(__name__)
```

```
@dataclass
class LoopPrevention:
```

무한 루프 감지 및 방지

감지 기준:

1. 동일한 Thought 반복
2. 동일한 Action 3회 연속
3. 최대 반복 횟수 초과

```
max_iterations: int = 10
max_same_action: int = 3
```

```
action_history: List[str] = field(default_factory=list)
thought_hashes: Set[int] = field(default_factory=set)
iteration_count: int = 0

def reset(self) -> None:
    """상태 초기화"""
    self.action_history = []
    self.thought_hashes = set()
    self.iteration_count = 0

def check_loop(self, thought: str, action: Dict[str, Any]) ->
    Dict[str, Any]:
    """루프 여부 확인"""
    self.iteration_count += 1

    result = {"is_loop": False, "reason": None, "should_stop": False}

    # 1. 최대 반복 횟수 확인
    if self.iteration_count > self.max_iterations:
        result["is_loop"] = True
        result["reason"] = f"최대 반복 횟수({self.max_iterations}) 초과"
        result["should_stop"] = True
        return result

    # 2. 동일 Thought 반복 확인
    thought_hash = hash(thought[:100])
    if thought_hash in self.thought_hashes:
        result["is_loop"] = True
        result["reason"] = "동일한 Thought 반복"
    else:
        self.thought_hashes.add(thought_hash)
```

```
# 3. 동일 Action 연속 확인
action_str = f"{action.get('action', '')}:{str(action.get('action_input', ''))[:50]}"
self.action_history.append(action_str)

if len(self.action_history) >= self.max_same_action:
    recent = self.action_history[-self.max_same_action:]
    if len(set(recent)) == 1:
        result["is_loop"] = True
        result["reason"] = f"동일 Action {self.max_same_action}회 연속"
        result["should_stop"] = True

return result

def get_stats(self) -> Dict[str, Any]:
    """현재 통계 반환"""
    return {
        "iteration_count": self.iteration_count,
        "unique_thoughts": len(self.thought_hashes),
        "total_actions": len(self.action_history),
        "max_iterations": self.max_iterations
    }
```

[테스트 코드 (주석)]

```
# from src.loop_prevention import LoopPrevention
# lp = LoopPrevention(max_iterations=5,
# max_same_action=3)
# result = lp.check_loop("첫 번째 생각", {"action": "search_web", "action_input": "AI"})
# print(f"Loop: {result['is_loop']}") # False
```

실습 Part2 구현 프롬프트 - 2

src/quality_manager.py 파일을 새로 생성하고
결과 품질 평가를 위한 QualityManager 클래스를 구현해주세요.

[파일 위치]

src/quality_manager.py

[구현 내용]

```
"""
품질 관리 모듈
ReAct 실행 결과의 품질을 평가하고 재시도 여부를 판단합니다.

from typing import Dict, Any, Optional
from openai import OpenAI
import json
import logging

logger = logging.getLogger(__name__)

class QualityManager:
    """결과 품질 평가 및 재시도 판단"""

    def __init__(self, client: OpenAI, min_quality_score: float = 7.0,
                 max_retries: int = 3):
        self.client = client
        self.min_quality_score = min_quality_score
        self.max_retries = max_retries
        self.evaluation_history: list = []
        logger.info(f"QualityManager initialized (min_score: {min_quality_score})")
```

```
def evaluate(self, task: str, result: str) -> Dict[str, Any]:
    """결과 품질 평가"""
    prompt = f"""당신은 AI 결과물 품질 평가 전문가입니다.

    ## 평가할 작업
    {task}

    ## 평가할 결과
    {result}

    ## 평가 기준 (각 1-10점)
    1. completeness (완전성): 요청한 내용을 모두 포함하는가
    2. accuracy (정확성): 정보가 정확한가
    3. relevance (관련성): 요청과 관련된 내용인가

    ## 출력 형식 (JSON)
    {"completeness": 점수, "accuracy": 점수, "relevance": 점수, "overall": 평균점수, "feedback": "개선
    방향"}"""

    JSON만 출력하세요."""

    try:
        response = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[
                {"role": "system", "content": "품질 평가 전문가입니다. JSON만 출력합니다."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.3,
            response_format={"type": "json_object"}
        )
        ..... (전체 내용은 별첨 참조)
```

실습 Part2 구현 프롬프트 - 3

config/prompts.py 파일에 ReAct 실행을 위한 프롬프트 템플릿을 추가해주세요.

[추가 위치]
config/prompts.py 파일의 TASK_DECOMPOSE_PROMPT 아래에 추가

[추가할 내용]

```
# ===== Week 4: ReAct Engine =====
```

REACT_SYSTEM_PROMPT = """당신은 자율적으로 사고하고 행동하는 리서치 어시스턴트입니다.

사용 가능한 도구

- search_web: 웹에서 정보 검색
- search_memory: 저장된 메모리에서 검색
- store_knowledge: 중요한 정보를 메모리에 저장
- analyze: 수집된 정보 분석 및 통합
- finish: 작업 완료 및 최종 답변 제출

응답 형식

반드시 다음 형식으로 응답하세요:

Thought: [현재 상황 분석 및 다음 행동 결정 이유]
Action: {"action": "도구명", "action_input": "입력값"}

작업 완료 시:

Thought: [완료 이유]
Action: {"action": "finish", "answer": "최종 답변"}

규칙

1. 매번 Thought로 시작하여 현재 상황을 분석하세요
2. Action은 반드시 유효한 JSON 형식이어야 합니다

3. 충분한 정보가 모이면 finish로 완료하세요
4. 같은 검색을 반복하지 마세요

```
## 현재 작업  
{task}
```

```
## 지금까지의 진행 상황  
{history}  
"""
```

REACT_USER_PROMPT = """위 작업을 수행하세요.
Thought로 시작하여 현재 상황을 분석하고, 적절한 Action을 선택하세요."""

```
# ===== Week 4: Result Synthesis =====
```

SYNTHESIS_PROMPT = """당신은 리서치 결과를 종합하는 전문가입니다.

원래 목표
{goal}

수집된 결과들
{results}

요청
위 결과들을 종합하여 하나의 완성된 리포트를 작성해주세요.

리포트 형식

1. 개요 (1-2문장)
2. 핵심 발견 (3-5개 bullet points)
3. 상세 분석 (필요시)
4. 결론 및 시사점

명확하고 구조화된 형식으로 작성하세요."""

실습 Part2 구현 프롬프트 - 4

src/react_engine.py 파일을 새로 생성하고
ReActEngine 클래스를 구현해주세요.

[파일 위치]
src/react_engine.py

[구현 내용]

```
"""
ReAct 실행 엔진
Thought-Action-Observation 순환을 통해 작업을 자율적으로
수행합니다.
"""


```

```
from typing import Dict, Any, List, Optional, Callable
```

```
from dataclasses import dataclass
```

```
from openai import OpenAI
```

```
import json
```

```
import re
```

```
import logging
```

```
from config.prompts import REACT_SYSTEM_PROMPT,
```

```
REACT_USER_PROMPT
```

```
from src.loop_prevention import LoopPrevention
```

```
logger = logging.getLogger(__name__)
```

```
@dataclass
class ReActStep:
    """ReAct 단일 스텝 기록"""
    iteration: int
    thought: str
    action: Dict[str, Any]
    observation: str

class ReActEngine:
    """ReAct 패턴 기반 자율 실행 엔진"""

    def __init__(self, client: OpenAI, tools: Dict[str, Callable],
max_iterations: int = 10):
        self.client = client
        self.tools = tools
        self.max_iterations = max_iterations
        self.history: List[ReActStep] = []
        self.loop_prevention =
LoopPrevention(max_iterations=max_iterations)
        logger.info(f"ReActEngine initialized with tools:
{list(tools.keys())}")

    def run(self, task: str) -> str:
        """작업 실행"""
        self.reset()
        logger.info(f"Starting ReAct for task: {task[:50]}...")

        for i in range(self.max_iterations):
            logger.info(f"==== Iteration {i+1}/{self.max_iterations}
```

..... (전체 내용은 별첨 참조)

실습 Part2 구현 프롬프트 - 5

src/orchestrator.py 파일을 새로 생성하고 AutonomousOrchestrator 클래스를 구현해주세요.

[파일 위치]
src/orchestrator.py

[구현 내용]

```
"""
자율 실행 오케스트레이터
TaskPlanner, ReActEngine, QualityManager를 조율하여 복잡한
작업을 수행합니다.
"""

from typing import Dict, Any, List, Optional
from openai import OpenAI
import logging
```

```
from src.task_planner import TaskPlanner, TaskStatus
from src.react_engine import ReActEngine
from src.quality_manager import QualityManager
from src.search_agent import SearchAgent
from src.memory_manager import MemoryManager
from config.prompts import SYNTHESIS_PROMPT
```

```
logger = logging.getLogger(__name__)
```

```
class AutonomousOrchestrator:
    """자율 실행 오케스트레이터"""

    def __init__(self, client: OpenAI, memory_manager: Optional[MemoryManager] = None, search_agent: Optional[SearchAgent] = None):
        self.client = client

        self.memory_manager = memory_manager or
MemoryManager(
            collection_name="orchestrator_memory",
            persist_directory="data/chroma_db"
        )
        self.search_agent = search_agent or
SearchAgent(memory_manager=self.memory_manager)

        self.task_planner = TaskPlanner(client)
        self.quality_manager = QualityManager(client)

        self.tools = self._create_tool_registry()
        self.react_engine = ReActEngine(client, self.tools)

        self.execution_log: List[Dict] = []
        logger.info("AutonomousOrchestrator initialized")

    def _create_tool_registry(self) -> Dict[str, callable]:
        """도구 레지스트리 생성"""

        def search_web(query: str) -> str:
            result = self.search_agent.search(query)
            return self.search_agent.format_for_llm(result)
..... (전체 내용은 별첨 참조)
```

실습 Part2 구현 프롬프트 - 6

main.py를 수정하여 자율 실행 모드를 추가해주세요.

[수정 사항]

1. Import 추가

```
from src.orchestrator import AutonomousOrchestrator
```

2. Orchestrator 초기화 (main 함수 내)

```
print("Initializing Autonomous Orchestrator...")
```

```
orchestrator = AutonomousOrchestrator(
```

```
    client=OpenAI(),
```

```
    memory_manager=memory_manager,
```

```
    search_agent=search_agent
```

```
)
```

```
print("✓ Autonomous Orchestrator Ready")
```

3. 새 명령어 추가 (while 루프 내)

```
elif user_input.lower().startswith('auto '):
```

```
    goal = user_input[5:].strip()
```

```
if not goal:
```

```
    print("사용법: auto <목표>")
```

```
    print("예시: auto AI 반도체 시장 동향 분석")
```

```
    continue
```

```
print(f"\n🚀 자율 실행 모드 시작")
```

```
print(f"목표: {goal}")
```

```
print("-" * 50)
```

```
try:
    result = orchestrator.execute(goal, verbose=True)
    print("\n" + "=" * 50)
    print("📋 최종 리포트")
    print("=" * 50)
    print(result)
except Exception as e:
    print(f"🔴 자율 실행 오류: {e}")
    continue

elif user_input.lower() in ['auto-stats', '자율통계']:
    stats = orchestrator.get_stats()
    print("\n📊 자율 실행 통계")
    print(f"총 실행 횟수: {stats['total_executions']}")
    if stats['quality_stats']:
        qs = stats['quality_stats']
        print(f"평균 품질 점수: {qs.get('average_score', 0):.1f}/10")
        print(f"품질 통과율: {qs.get('pass_rate', 0)*100:.1f}%")
    continue
```

4. 도움말 업데이트

```
print(" auto <목표> - NEW 자율 실행 모드")
```

```
print(" auto-stats - NEW 자율 실행 통계")
```

실습 Part2 구현 프롬프트 - 7

tests/test_orchestrator.py 파일을 생성하고
Part 2의 통합 테스트를 작성해주세요.

[파일 위치]

tests/test_orchestrator.py

[구현 내용]

```
"""AutonomousOrchestrator 통합 테스트"""
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(_file_))))

from openai import OpenAI
from src.loop_prevention import LoopPrevention
from src.quality_manager import QualityManager
from src.react_engine import ReActEngine
from src.orchestrator import AutonomousOrchestrator

def test_loop_prevention():
    """LoopPrevention 테스트"""
    print("\n" + "*50)
    print("TEST 1: Loop Prevention")
    print("*50)

    lp = LoopPrevention(max_iterations=5, max_same_action=3)
```

```
result = lp.check_loop("생각1", {"action": "search_web", "action_input": "AI"})
assert not result["is_loop"]
print("✅ 정상 액션 통과")

lp.check_loop("생각2", {"action": "analyze", "action_input": "data"})

for i in range(3):
    result = lp.check_loop(f"생각{i+3}", {"action": "search_web", "action_input": "same"})

    assert result["is_loop"]
    print("✅ 동일 액션 연속 감지")

lp.reset()
for i in range(6):
    result = lp.check_loop(f"생각{i}", {"action": f"action_{i}", "action_input": str(i)})

    assert result["should_stop"]
    print("✅ 최대 반복 감지")

return True
```

```
def test_quality_manager():
    """QualityManager 테스트"""
    print("\n" + "*50)
    print("TEST 2: Quality Manager")
    print("*50)
```

..... (전체 내용은 별첨 참조)

Part 2 단위 테스트

① 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> python tests/test_orchestrator.py

⚙️ 단위 테스트 항목

1. Loop Prevention Test

정상 액션 통과, 동일 액션 3회 연속 시 루프 감지, reset 후 6회 반복 시 강제 정지

2. Quality_Manager Test

좋은 결과"/"부족한 결과" 평가, overall \geq 6 검증

3. React_Engine Test

Mock search_web/analyze 도구로 "AI 시장 규모 조사" 실행, 결과.history 존재 확인

4. Orchestrator Test

AutonomousOrchestrator(client): "Python의 장단점을 3가지씩 정리해주세요" 실행(verbose=False), 결과 길이 > 50, get_stats() 확인

4주차 전체 통합 테스트 개요

시스템 동작 검증 및 시나리오 개요

◎ 통합 테스트 개요

4주차에서 구현한 자율 실행 AI 에이전트가 사용자 관점에서 정상적으로 작동하는지 검증

검증 항목	설명
자율 실행 기능	auto 명령어로 목표를 입력하면 스스로 계획하고 실행하는가
작업 분해	복잡한 목표를 적절한 서브태스크로 분해하는가
품질 관리	결과 품질이 낮으면 자동으로 재시도하는가
기초 기능 호환	1-3주차 기능(대화, 검색, 메모리)이 정상 작동하는가

▣ 테스트 시작 전 준비사항

- OpenAI API 키 설정 (.env)
- Tavily API 키 설정 (.env)
- ChromaDB 정상 작동

☰ 테스트 시나리오 (총 7개)

#	시나리오	테스트 내용 (Command)	핵심 확인 포인트
1	간단한 자율 실행	auto Python 장점 3가지 정리	작업 분해 → 순차 실행 → 리포트 생성
2	복잡한 리서치	auto 전기차 시장 동향 분석	5개 이상 태스크, 의존성 처리, 웹 검색 활용
3	품질 재시도	auto 양자컴퓨터 원리 분석	품질 7.0 미만 시 재시도 발생 여부
4	통계 확인	auto-stats	실행 횟수, 평균 품질, 통과율 표시
5	에러 처리	auto (빈 입력)	사용법 안내 메시지 표시
6	기존 기능 호환	일반 대화, 검색, 메모리 기능	1-3주차 기능 정상 작동
7	의존성 처리	auto 삼성전자 실적 분석	의존성 순서대로 실행 (선행 작업 완료 대기)

통합 테스트 실행

💡 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> python main.py

또는 스크립트 사용:

Windows: run.bat

macOS/Linux: ./run.sh

시나리오 1: 간단한 자율 실행 테스트

◎ 목적

자율 실행 모드가 정상적으로 작동하는지 기본 확인

> 테스트 입력

COMMAND

You: auto Python 프로그래밍 언어의 장점 3가지 정리

✓ 통과 기준 (Pass Criteria)

작업 분해: 3-7개의 서브태스크로 분해됨

순차 실행: task_1 → task_2 → task_3 순서로 실행

품질 점수: 각 태스크마다 품질 점수 표시 (7.0 이상)

최종 리포트: 요청한 내용(장점 3가지)이 포함됨

Output Console - Expected Result

You: auto Python 프로그래밍 언어의 장점 3가지 정리

🚀 자율 실행 모드 시작

목표: Python 프로그래밍 언어의 장점 3가지 정리

🎯 목표: Python 프로그래밍 언어의 장점 3가지 정리

📋 Step 1: 작업 분해 중...

2026-02-03 15:05:14,596 - httpx - INFO - HTTP Request: POST

https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"

2026-02-03 15:05:14,608 - src.task_planner - INFO - Decomposed 'Python 프로그래밍 언어의 장점 3가지 정리' into 6 subtasks

📋 Task Plan: Python 프로그래밍 언어의 장점 3가지 정리...

⌚ [task_1] Python의 간결한 문법을 조사하여 장점으로 정리한다.

Priority: 1 | Status: pending

… (중략)

📊 Progress: 6/6 (100.0%)

📋 최종 리포트

Python 프로그래밍 언어의 장점 리포트

시나리오 2: 복잡한 리서치 작업 테스트

◎ 목적

웹 검색과 메모리 활용이 포함된 복잡한 작업 수행 확인

▶ 테스트 입력

COMMAND

You: auto 2024년 전기차 시장 동향과 주요 기업 분석

✓ 통과 기준 (Pass Criteria)

작업 분해: 5개 이상의 서브테스크 생성

의존성 처리: 의존성 있는 task는 앞에 task 실행 후, 수행

웹 검색 활용: 최신 시장 데이터 포함

구조화된 리포트: 개요, 핵심 발견, 결론 포함

Output Console - Expected Result

You: auto 2024년 전기차 시장 동향과 주요 기업 분석

🚀 자율 실행 모드 시작

목표: 2024년 전기차 시장 동향과 주요 기업 분석

🎯 목표: 2024년 전기차 시장 동향과 주요 기업 분석

📋 Step 1: 작업 분해 중...

2026-02-03 15:17:31,099 - httpx - INFO - HTTP Request: POST

https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"

2026-02-03 15:17:31,112 - src.task_planner - INFO - Decomposed '2024년 전기차 시장 동향과 주요 기업 분석' into 6 subtasks

📋 Task Plan: 2024년 전기차 시장 동향과 주요 기업 분석...

⌚ [task_1] 2024년 전기차 시장 동향에 대한 자료 수집

Priority: 1 | Status: pending

(중략)

📊 Progress: 6/6 (100.0%)

📋 최종 리포트

2024년 전기차 시장 동향 및 주요 기업 분석 리포트

시나리오 3: 품질 재시도 테스트

◎ 목적

품질이 낮을 때 자동으로 재시도하는지 확인

> 테스트 입력

COMMAND

You: **auto 지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시**

✓ 통과 기준 (Pass Criteria)

재시도 발생: 품질 7.0 미만 시 "재시도 중..." 메시지 표시

최대 3회: 최대 3회까지만 재시도

품질 개선: 재시도 후 품질 점수가 상승하는 경향

Output Console - Expected Result

You: auto 지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시

🚀 자율 실행 모드 시작

목표: 지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시

=====

📋 Step 1: 작업 분해 중...

2026-02-03 15:34:00,366 - httpx - INFO - HTTP Request: POST

https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"

2026-02-03 15:34:00,375 - src.task_planner - INFO - Decomposed '지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시' into 5 subtasks

=====

📋 Task Plan: 지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시...

=====

⌚ [task_1] 지난주 테슬라 주가 데이터를 수집한다.

Priority: 1 | Status: pending
(중략)

2026-02-03 15:35:36,910 - src.quality_manager - INFO - Quality: 6.0/10 (FAIL)

품질 점수: 6.0/10

재시도 중... (2/3)

2026-02-03 15:35:36,911 - src.react_engine - INFO - Starting ReAct for task: 이전 결과가 품질 기준을 충족하지 못했습니다.

(중략)

🌟 작업 완료!

=====

📋 Task Plan: 지난주 테슬라 주가 변동을 시간대별로 분석하고 정확한 수치 제시...

=====

시나리오 4: 통계 확인 테스트

◎ 목적

자율 실행 통계가 정상적으로 기록되는지 확인

> 테스트 입력

COMMAND

1. 먼저 몇 가지 자율 실행 수행

You: auto Python vs JavaScript 비교
 You: auto 클라우드 컴퓨팅의 장단점

- # 2. 통계 확인

You: auto-stats

✓ 통과 기준 (Pass Criteria)

총 실행 횟수: 수행한 auto 명령 횟수와 일치

평균 품질 점수: 7.0 이상

통과율: 대부분 70% 이상

Output Console - Expected Result

You: auto 클라우드 컴퓨팅의 장단점

🚀 자율 실행 모드 시작
 목표: 클라우드 컴퓨팅의 장단점

📋 Task Plan: 클라우드 컴퓨팅의 장단점...

⌚ [task_1] 클라우드 컴퓨팅의 정의와 기본 개념 조사
 Priority: 1 | Status: pending

⌚ [task_2] 클라우드 컴퓨팅의 장점 목록 작성
 Priority: 2 | Status: pending (depends: task_1)

(중략)

결론 및 시사점

클라우드 컴퓨팅은 비용 절감, 효율성 향상, 확장성 등 많은 장점을 제공하지만, 보안 위험, 다운타임, 공급업체 종속과 같은 단점도 존재합니다. 기업은 클라우드 서비스의 이점을 극대화하고 단점을 최소화하기 위해 신중하게 공급업체를 선택하고, 보안 정책을 강화하며, 지속적인 비용 관리 전략을 수립해야 합니다. 클라우드 컴퓨팅은 현대 비즈니스 환경에서 필수적인 도구로 자리 잡고 있으며, 이를 통해 기업은 경쟁력을 유지하고 성장할 수 있는 기회를 얻을 수 있습니다.

You: auto-stats

📊 자율 실행 통계

총 실행 횟수: 5
 평균 품질 점수: 8.7/10
 품질 통과율: 96.6%

시나리오 5: 에러 처리 테스트

◎ 목적

잘못된 입력이나 예외 상황에서 적절히 처리되는지 확인

> 테스트 입력

COMMAND

1. 비어 있는 입력

You: **auto**

✓ 통과 기준 (Pass Criteria)

입력이 비었으므로 사용법 제시

> 테스트 입력

COMMAND

2. 매우 짧은 목표

You: **auto 안녕**

✓ 통과 기준 (Pass Criteria)

오류 없이 처리 (간단한 태스크로 분해)

● ● ● Output Console - Expected Result

You: auto
사용법: auto <목표>
예시: auto AI 반도체 시장 동향 분석

● ● ● Output Console - Expected Result

You: auto 안녕

🚀 자율 실행 모드 시작

목표: 안녕

(중략)

결론 및 시사점

인사말과 친구의 반응을 이해하는 과정은 단순한 소통을 넘어 서로의 정서를 이해하고 관계를 강화하는 데 중요한 역할을 합니다. 친구 간의 대화에서 정서적 지능과 적극적 경청의 필요성을 강조하며, 앞으로의 대화에서 이러한 요소들을 고려하는 것이 관계를 더욱 깊고 의미 있게 만들 수 있음을 알립니다. 향후 연구는 다양한 상황에서의 친구 간의 대화 패턴과 정서적 반응을 분석하여, 보다 효과적인 소통 방법을 제시할 필요가 있습니다.

시나리오 6: 기존 기능 호환성 테스트

◎ 목적

4주차 기능 추가 후에도 기존 1-3주차 기능이 정상 작동하는지 확인

▶ 테스트 입력

COMMAND

You: 인공지능이란 무엇인가요?

✓ 통과 기준 (Pass Criteria)

일반 대화이므로 정상 응답

▶ 테스트 입력

COMMAND

You: 오늘 주요 뉴스를 검색해줘

✓ 통과 기준 (Pass Criteria)

웹을 검색해서 결과 표시

● ● ● Output Console - Expected Result

You: 인공지능이란 무엇인가요?

처리 중...

2026-02-03 16:27:01,818 - src.conversation_manager - INFO - 대화 상태 변경: idle -> responding
 2026-02-03 16:27:01,818 - src.conversation_manager - INFO - 일반 응답 모드 - 간결한 응답 프롬프트 추가
 2026-02-03 16:27:01,818 - src.conversation_manager - INFO - 사용자 메시지 추가 (총 1개, 상태: responding): 인공지능이란 무엇인가요?...

2026-02-03 16:27:03,991 - httpx - INFO - HTTP Request: POST
<https://api.openai.com/v1/chat/completions> "HTTP/1.1 200 OK"
 (중략)

AI: 인공지능(AI)은 기계나 소프트웨어가 인간의 지능을 모방하여 학습, 문제 해결, 의사 결정 등을 수행할 수 있도록 하는 기술입니다. AI는 데이터를 분석하고 패턴을 인식하여 스스로 개선하는 능력을 가지고 있으며, 여러 분야에서 활용되고 있습니다.

어떤 특정 측면이나 분야에 대해 더 알고 싶으신가요?

● ● ● Output Console - Expected Result

You: 오늘 주요 뉴스를 검색해줘

처리 중...

2026-02-03 16:29:25,926 - src.tools.web_search - INFO - Tavily 검색 시작 - 쿼리: '오늘 주요 뉴스', depth: basic, max_results: 5

2026-02-03 16:29:28,048 - src.tools.web_search - INFO - Tavily 검색 완료 - 결과: 5개, 소요 시간: 2.12초

2026-02-03 16:29:28,048 - src.conversation_manager - INFO - 검색 완료: 5개 결과, 2.12초

2026-02-03 16:29:31,542 - httpx - INFO - HTTP Request: POST
 [상세 내용]

- 주요 정치 뉴스: 민주당 내에서의 합당 반대 의견이 격화되고 있으며, 당권파와 반대파 간의 갈등이 심화되고 있습니다.

- 경제 동향: 코스피가 하루 만에 5,200으로 복귀하였고, 물가상승률이 지난 5개월 내 최저치를 기록했습니다.

시나리오 6: 기존 기능 호환성 테스트

◎ 목적

4주차 기능 추가 후에도 기존 1-3주차 기능이 정상 작동하는지 확인

▶ 테스트 입력

COMMAND

You: **memory**

✓ 통과 기준 (Pass Criteria)

메모리 통계 정상 표시

Output Console - Expected Result

You: memory
2026-02-03 16:32:11,396 - src.memory_manager - INFO - 전체 문서 조회: 25개
2026-02-03 16:32:11,396 - src.memory_manager - INFO - 상세 통계 조회: 25개 문서

▣ 메모리 시스템 대시보드

컬렉션: research_assistant_memory
총 문서 수: 25개
평균 텍스트 길이: 167자
임베딩 차원: 1536

📁 소스별 분포:

- conversation: 12개
- autonomous_agent: 13개

📅 기간별 분포:

- 최근 24시간: 18개
- 최근 7일: 25개
- 최근 30일: 25개
- 그 이전: 0개

💾 캐시 정보:

- 캐시 크기: 10개
- 적중률: 50.0%

2026-02-03 16:32:11,396 - src.memory_manager - INFO - 대시보드 출력 완료

시나리오 7: 의존성 처리 테스트

◎ 목적

태스크 간 의존성이 올바르게 처리되는지 확인

> 테스트 입력

COMMAND

You: **auto 삼성전자 2024년 실적 분석 및 투자 의견 작성**

✓ 통과 기준 (Pass Criteria)

의존성 표시: (depends: task_1) 형태로 표시

실행 순서: task_1 완료 후 task_2, task_3 실행

복수 의존성: task_2, task_3 모두 완료 후 task_4 실행

Output Console - Expected Result

You: auto 삼성전자 2024년 실적 분석 및 투자 의견 작성

🚀 자율 실행 모드 시작

목표: 삼성전자 2024년 실적 분석 및 투자 의견 작성

📋 Step 1: 작업 분해 중...

2026-02-03 16:34:26,346 - httpx - INFO - HTTP Request: POST

https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"

2026-02-03 16:34:26,346 - src.task_planner - INFO - Decomposed '삼성전자 2024년 실적 분석 및 투자 의견 작성' into 6 subtasks

📋 Task Plan: 삼성전자 2024년 실적 분석 및 투자 의견 작성...

⌚ [task_1] 삼성전자의 2024년 실적 관련 자료 수집
Priority: 1 | Status: pending

⌚ [task_2] 2024년 실적 예측 및 주요 지표 분석
Priority: 2 | Status: pending (depends: task_1)

⌚ [task_3] 경쟁사 분석 및 시장 동향 조사
Priority: 3 | Status: pending (depends: task_1)

⌚ [task_4] 투자 의견 작성에 필요한 재무 비율 계산
Priority: 4 | Status: pending (depends: task_2, task_3)

⌚ [task_5] 투자 의견 초안 작성
Priority: 5 | Status: pending (depends: task_4)

(후략)

05

SECTION

SUMMARY REVIEW PREVIEW

마무리 및 다음 주 예고

이번 주 학습 내용을 정리하고 다음 단계인 멀티 에이전트 협업을 준비합니다.

◎ Session Wrap-up

- ✓ 4주차 완성 기능 요약 (TaskPlanner)
- ✓ 5주차 예고: 멀티 에이전트 협업
- ✓ 핵심 학습 내용 리뷰 (ReAct 패턴)
- ✓ 최종 과제 안내 및 Q&A

Keywords:

● Wrap-up

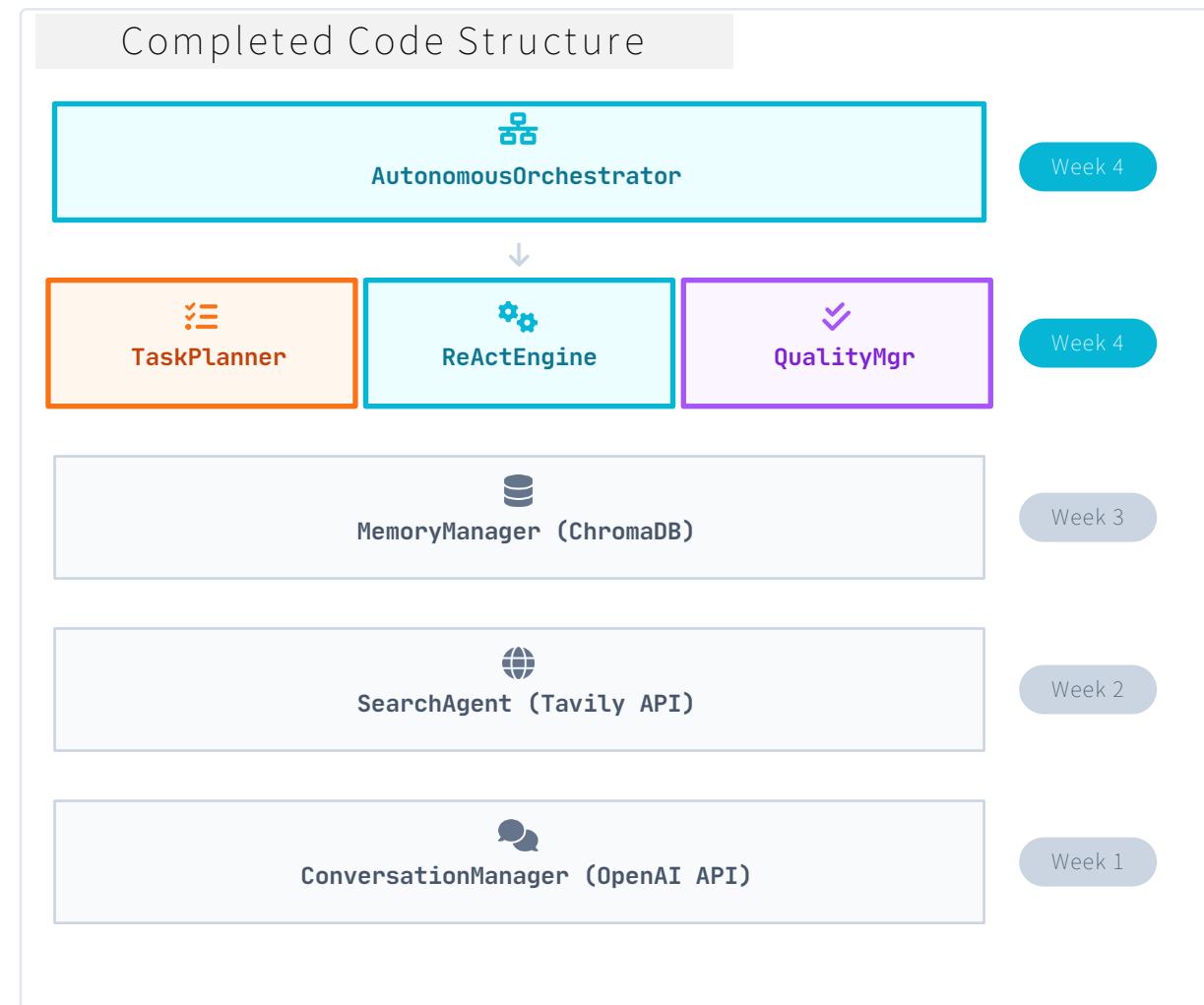
● Future Preview

● Multi-Agent

4주차 완성 기능

이번 주 구현한 핵심 기능 및 시스템 완성도

File	Key Classes/Functions
src/task_planner.py	TaskStatus, Subtask, TaskPlanner
src/loop_prevention.py	LoopPrevention
src/quality_manager.py	QualityManager
src/react_engine.py	ReActStep, ReActEngine
src/orchestrator.py	AutonomousOrchestrator
config/prompts.py	TASK_DECOMPOSE_PROMPT, REACT_SYSTEM_PROMPT, SYNTHESIS_PROMPT
tests/test_task_planner.py	5개 테스트 함수
tests/test_orchestrator.py	4개 테스트 함수



4주차 핵심 학습 내용 정리

개념 (Concept)	핵심 포인트 (Key Point)
🤖 자율 에이전트	Goal → Plan → Act → Reflect
♾️ ReAct 패턴	Thought → Action → Observation
➊ Task Decomposition	복잡한 목표를 작은 단위(Subtask)로 분해
💡 Self-Reflection	결과 품질 자체 검증 및 자동 재시도

🔑 개념 상세

ReAct Pattern
Thought → Action → Observation → (반복) → Answer

Task Decomposition
복잡한 목표 → [Subtask 1] → [Subtask 2] → ... → [최종 결과]

Quality Management
결과 생성 → 품질 평가 (1-10점) → 7점 미만? → 재시도

Loop Prevention
최대 반복: 10회
동일 Action 연속: 3회 초과 시 중단

5주차 예고: 멀티 에이전트 협업 시스템

핵심 개념 KEY CONCEPTS

- 멀티 에이전트 아키텍처
전문화된 에이전트들이 협업하여 복잡한 작업 수행
- 역할 분담 전략
각 에이전트에 특화된 역할과 책임 부여 (R/R)
- 에이전트 간 통신
구조화된 프로토콜로 정보와 결과 교환
- 협업 패턴
순차, 병렬, 반복 실행 방식 조합 최적화



주요 아키텍처 SYSTEM DESIGN

- 에이전트 구성
Researcher 웹 검색 및 정보 수집 담당
Analyzer 클러스터링, 트렌드 분석
Writer 구조화된 리포트 작성
Critic 완성도, 정확성, 출처 검증
- 통합 시스템
Coordinator 협업 관리 및 개선 루프
Output Markdown/HTML 리포트 변환