CHOI SUNGJANG

2025

PORTFOLIO

PROJECT : RulebaseAgent — 규정 기반 Tool-Using Agent

T. 010-8967-9399 | Github. https://github.com/GrowingChoi | Email. palga65@gmail.com | Blog. blog.naver.com/growingdev95

01. 프로젝트 개요

프로젝트 명 : RulebaseAgent - 규정 기반 Tool-Using Agent(개인프로젝트)

기간 : 2025.11

사용기술 : Python · FastAPI · OpenAI GPT-5-mini · JSON Planner · Multi-step Reasoning · Chunking · Metadata Search

1-1. 한문장 요약

- 규정 분석 업무를 위해 Planner → Executor → Tool Pipeline을 직접 설계하여
  LLM이 스스로 작업 단계를 결정하고 실행하는 Human-like Tool-Using Agent 시스템을 구축한 개인 프로젝트

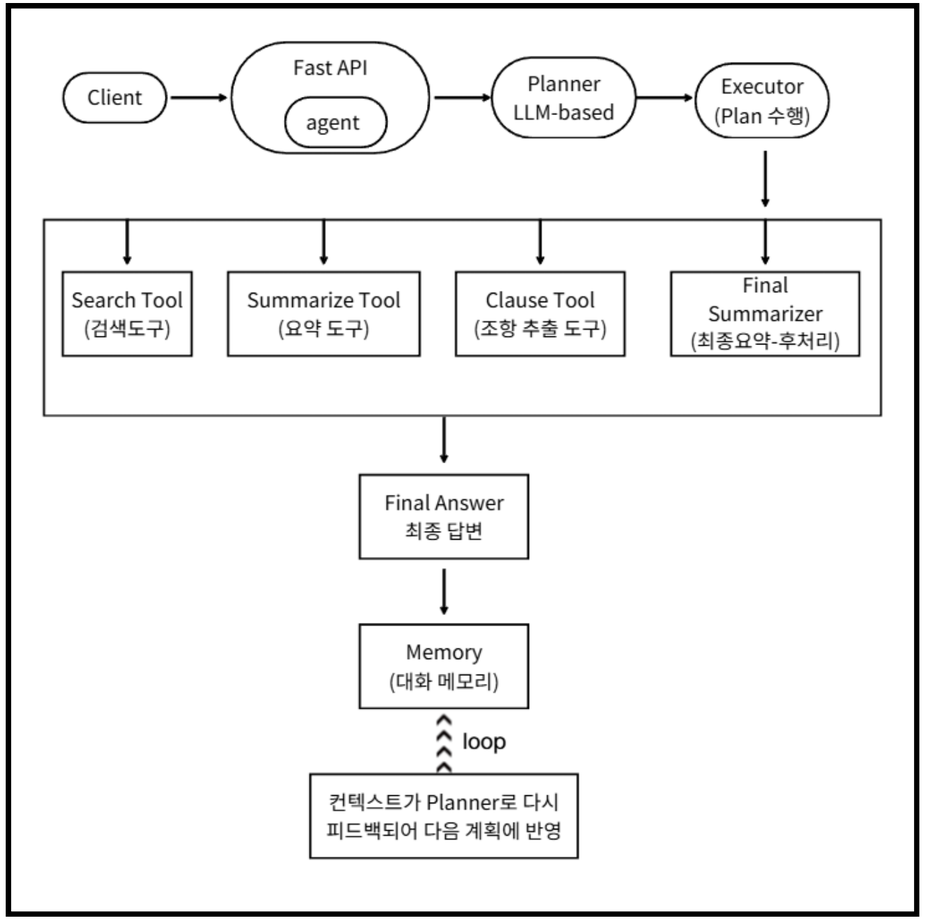
1-2. 프로젝트를 시작한 이유

- 기존 RAG 방식은 검색 흐름이 고정되어 Reasoning 확장성이 부족하다고 느낌
- LLM이 스스로 무엇을 해야 하는지 Plan을 만들고 Tool을 조합하는 구조를 직접 구현하고 싶었음
- 규정 분석·조항추출·요약등 "도메인 특화 작업"을 Agentic 방식으로 처리하는 작은 실험을 해보고자 함
- 외부 프레임워크 없이 순수 Python·FastAPI 기반으로 Agentic AI의 핵심 구조를 재현하는 것을 목표로 함

1-3. 목표(Goal)

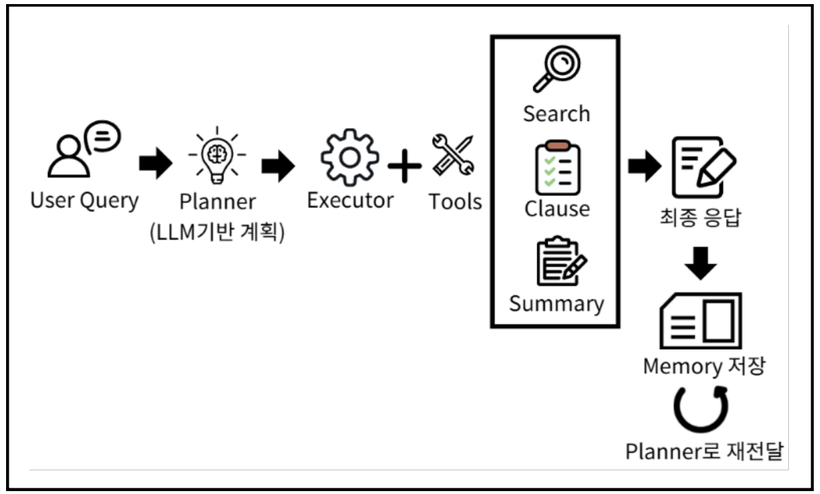
- 기존 RAG 방식의 고정된 검색 흐름을 보완
- LLM이 직접 Plan을 생성·수정하며단계별로 행동을 선택하는 Agentic 구조를 구현
- 규정 검색 → 조항 추출 → 요약 과정을 자동으로 연결하는
  Human-like Multi-step Reasoning 시스템을 개발하는 것

아키텍쳐 흐름도



02. 개발 과정

2-1. 전체 개발 흐름



- "무엇을 할지"를 LLM Planner가 판단
- Plan은 JSON 형태의 구조화된 행동 지시서 역할
- Executor는 Plan을 해석하여 Tool을 실행
- Tool은 각각 독립된 기능(Search/Clause/Summary)으로 구성
- 결과는 다시 Planner로 돌아가 Multi-step Reasoning을 이어감
- 사람이 문제를 해결하는 흐름에 가까운 Human-like Agent 구조 구현

2-2. 개발 상세

- Planner - JSON 형태의 단일 스텝 계획 생성

Planner는 사용자 질문과 Memory의 최근 대화를 기반으로 LLM에게 다음 실행 단계를 JSON Plan으로 생성하도록 요청하는 역할을 수행했고, 이를 통해 Agent가 매번 어떤 Tool을 실행해야 할지 판단할 수 있도록 구성했습니다.

```
def plan(self, user_query: str, memory_context: str) -> Dict:
    """
    LLM에게 JSON 형태의 플랜을 생성하도록 요청.
    {
      "tool": "search" | "summarize" | "extract_clause" | "final_answer",
      "tool_input": { ... },
      "reason": "왜 이 액션을 선택했는지",
      "is_final": bool
    }
    """
    tools_str = ", ".join(self.tool_names)
    context_part = f"최근 대화 컨텍스트:\n{memory_context}\n\n" if memory_context else ""
```

```
    response = client.chat.completions.create(
        model=MODEL_NAME,
        messages=[
            {
                "role": "system",
                "content": "너는 Tool-Using Agent의 플래너로서, 항상 유효한 JSON만 생성해야 한다."
            },
            {"role": "user", "content": prompt},
        ],
    )
    raw = response.choices[0].message.content.strip()
```

이후 JSON 파싱 및 기본값 설정을 수행했고 파싱이 실패할 경우에는 기본적으로 search Tool을 수행하도록 설정했습니다. 그리고 final_answer 선택 시 answer가 비어 있을 경우 기본 메시지를 자동 삽입하여, Agent가 오류 없이 다음 단계를 이어갈 수 있도록 설계했습니다

02. 개발 과정

- Executor

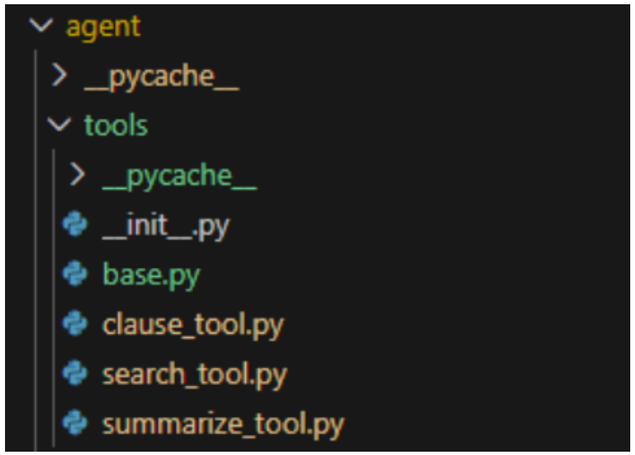
```python
def execute(self, plan: Dict[str, Any], user_query: str) -> Dict[str, Any]:
    tool = plan.get("tool")
    tool_input = plan.get("tool_input") or {}
    result: Any = None

    if tool == "search":
        query = tool_input.get("query", user_query)
        result = self.search_tool.run(query=query)
    elif tool == "summarize":
        docs: List[str] = tool_input.get("texts", [])
        if not docs:
            result = "summarize 할 텍스트가 없습니다."
        else:
            result = self.summarize_tool.run(texts=docs, user_query=user_query)
    elif tool == "extract_clause":
        docs: List[str] = tool_input.get("texts", [])
        if not docs:
            result = "추출할 규정 텍스트가 없습니다."
        else:
            result = self.clause_tool.run(texts=docs, user_query=user_query)
    elif tool == "final_answer":
        # final_answer는 executor에서 별도 처리 없이 plan 자체에 들어있다고 가정할 수도 있음.
        result = tool_input.get("answer", "별도의 최종 답변이 제공되지 않았습니다.")
    else:
        result = f"알 수 없는 tool: {tool}"
```

Executor는 Planner가 생성한 Plan에서 tool 값을 읽고 해당 값에 따라 if-elif 체인으로 Tool을 실행하는 방식으로 구현되어 있었습니다.

이 구조는 초기 구현 단계에서는 충분히 단순하고 직관적이어서 빠르게 개발을 진행할 수 있었으며, 실제로 Agent 흐름을 검증하는 데 무리가 없었습니다.

- Tools - Search / Clause / Summary



각 tool을 분리한 파일 구조

```python
def run(self, texts: List[str], user_query: str) -> str:
    joined = "\n\n".join(texts)
    prompt = f"""
            아래는 규정 문서의 일부 내용입니다. 사용자의 질문에 답변할 수 있도록
            관련된 부분을 중심으로 간단하고 명확하게 요약해 주세요.

            [사용자 질문]
            {user_query}

            [관련 규정 내용]
            {joined}
        """
    response = client.chat.completions.create(
        model=MODEL_NAME,
        messages=[
            {
                "role": "system",
                "content": "너는 규정과 정책을 잘 설명해주는 한국어 규정 전문가야."
            },
            {"role": "user", "content": prompt},
        ],
    )
    return response.choices[0].message.content.strip()
```

summarize_tool의 run()

```python
def run(self, texts: List[str], user_query: str) -> str:
    joined = "\n\n".join(texts)
    prompt = f"""
            아래 규정 내용에서, 사용자의 질문과 직접적으로 연관된 조항(조건, 허용 범위, 예외)을
            조항 단위로 정리해 주세요.

            [사용자 질문]
            {user_query}

            [관련 규정 내용]
            {joined}

            출력 형식 예시:
            - 관련 조항 요약 1
            - 관련 조항 요약 2
            - ...

            꼭 한국어로 간결하게 정리해 주세요.
        """
```

clause_tool의 run()

```python
def run(self, query: str, top_k: int = 3) -> List[Dict]:
    """
    매우 단순한 키워드 매칭 기반 검색.
    """
    query_lower = query.lower()
    scored = []
    for rule in self.rules:
        text = (rule["title"] + " " + rule["content"]).lower()
        score = sum(query_lower.count(token) for token in query_lower.split())
        if score > 0:
            scored.append((score, rule))
    scored.sort(key=lambda x: x[0], reverse=True)
    results = [item[1] for item in scored[:top_k]]
    return results
```

search_tool의 run()

Tool은 독립된 실행 단위를 구성하고 있었며 각 Tool은 Planner의 Plan에 따라 호출되도록 설계되었습니다.

03. 문제점 발견 및 구조 개선

3-1. 최초 구현에서 발견한 문제점

- 초기 Agent 구조는 정상적으로 동작했지만, 확장성과 일관성 측면에서 여러 한계를 확인했습니다.

```python
if tool == "search":
    query = tool_input.get("query", user_query)
    result = self.search_tool.run(query=query)
elif tool == "summarize":
    docs: List[str] = tool_input.get("texts", [])
    if not docs:
        result = "summarize 할 텍스트가 없습니다."
    else:
        result = self.summarize_tool.run(texts=docs, user_query=user_query)
elif tool == "extract_clause":
    docs: List[str] = tool_input.get("texts", [])
    if not docs:
        result = "추출할 규정 텍스트가 없습니다."
    else:
        result = self.clause_tool.run(texts=docs, user_query=user_query)
elif tool == "final_answer":
    # final_answer는 executor에서 별도 처리 없이 plan 자체에 들어있다고 가정할 수도 있음.
    result = tool_input.get("answer", "별도의 최종 답변이 제공되지 않았습니다.")
else:
    result = f"알 수 없는 tool: {tool}"
```

- if-elif의 확장성 한계
  - Tool이 늘어날수록 executor.py가 비대해짐
  - 새로운 Tool 추가 시 Executor 내부 수정이 필요
- 메시지 & 입력 검증 불일치
  - "텍스트 없음" 에러 메시지가 Tool별로 다름
  - 사용자 응답 일관성 떨어짐

```python
def run(self, texts: List[str], user_query: str) -> str:
    joined = "\n\n".join(texts)
    prompt = f"""
                아래 규정 내용에서, 사용자의 질문과 직접적으로 연관된
                조항 단위로 정리해 주세요.

                [사용자 질문]
                {user_query}

                [관련 규정 내용]
                {joined}
def run(self, query: str, top_k: int = 3) -> List[Dict]:
    """
    매우 단순한 키워드 매칭 기반 검색.
    """
    query_lower = query.lower()
    scored = []
    for rule in self.rules:
        text = (rule["title"] + " " + rule["content"]).lower()
        score = sum(query_lower.count(token) for token in query_lower.split())
        if score > 0:
            scored.append((score, rule))
    scored.sort(key=lambda x: x[0], reverse=True)
    results = [item[1] for item in scored[:top_k]]
    return results
```

- Tool 인터페이스 불일치
  - 어떤 Tool은 query, 어떤 Tool은 texts,
    어떤 Tool은 user_query 필요
  - Executor가 각 Tool의 규칙을 모두 알아야 하므로
    책임이 과도함

```python
def __init__(self, search_tool: SearchTool, summarize_tool: SummarizeTool, clause_tool: ClauseTool):
    self.search_tool = search_tool
    self.summarize_tool = summarize_tool
    self.clause_tool = clause_tool
```

- Tool 추가시 유지보수 문제
  - Tool 추가할 때마다 Executor를 수정해야 함
  - 구조적으로 확장에 닫혀 있고 수정에 열려있는 형태

```python
elif tool == "final_answer":
    # final_answer는 executor에서 별도 처리 없이 plan 자체에 들어있다고 가정할 수도 있음.
    result = tool_input.get("answer", "별도의 최종 답변이 제공되지 않았습니다.")
else:
    result = f"알 수 없는 tool: {tool}"
```

- 예외처리 미흡
  - Tool 실행 중 오류가 나면 500 에러 리스크
  - 서비스 안정성 낮아짐

04. 구조 개선

4-1. 확장성과 안정성을 갖춘 Agent 구조로 재설계

- 초기 구조에서 발견된 문제(확장성 부족, 입력 검증 불일치, Executor 책임 과중 등)를 해결하기 위해 아래와 같은 구조 개선을 단계적으로 진행했습니다.

1) BaseTool 인터페이스 도입

- 여러 Tool이 서로 다른 방식으로 실행되던 문제를 해결하기 위해공통 실행 구조(run 메서드)를 갖는 BaseTool을 도입했습니다.

```python
class Tool(ABC):
    """
    모든 Tool이 따라야 하는 공통 인터페이스.

    - name: planner가 선택할 때 사용할 문자열 이름
    - description: 프롬프트/문서화용 설명
    - run(): 실제 실행 로직 (user_query + tool_input 기반)
    """

    name: str
    description: str = ""

    @abstractmethod
    def run(self, *, user_query: str, tool_input: Dict[str, Any]) -> Any:
        """
        Tool 실행 메서드.
        - param user_query: 원본 사용자 질문
        - param tool_input: planner가 넘겨준 세부 입력 파라미터
        - return: Tool 실행 결과
        """
        raise NotImplementedError
```

- 개선 후 효과
  - Tool 간 입력·오류처리 로직이 일관화됨
  - Executor가 Tool 내부 구조를 몰라도 실행 가능해짐

2) Tool Registry 패턴 적용 (if-elif 제거)

- 기존 Executor의 조건문 기반 실행 구조를 제거하고, 문자열 key로 Tool을 관리하는 Registry 패턴을 적용했습니다.

```python
def build_tool_registry(*, data_path: str) -> Dict[str, Tool]:
    """
    프로젝트에서 사용할 Tool 인스턴스를 생성하고
    `{tool_name: tool_instance}` 형태의 레지스트리를 만들어 반환.

    새 Tool을 추가하고 싶으면 여기에서만 인스턴스를 추가해도 됨.
    """
    search_tool = SearchTool(data_path=data_path)
    summarize_tool = SummarizeTool()
    clause_tool = ClauseTool()

    tools = [search_tool, summarize_tool, clause_tool]
    return {tool.name: tool for tool in tools}


__all__ = [
    "Tool",
    "SearchTool",
    "SummarizeTool",
    "ClauseTool",
    "build_tool_registry",
]
```

- 개선 후 효과
  - Executor 수정 없이 Tool 추가 가능
  - 구조가 단순해지고 확장성 확보

3) Executor 역할 단순화

- Executor가 입력 검증·오류처리까지 담당하던 구조를 개선하여 "Plan 해석 → Tool 실행"만 수행하도록 역할을 단순화했습니다.

```python
def execute(self, plan: Dict[str, Any], user_query: str) -> Dict[str, Any]:
    tool_name: str = plan.get("tool", "")
    tool_input: Dict[str, Any] = plan.get("tool_input", {}) or {}

    if tool_name == "final_answer":
        result = tool_input.get("answer", "별도의 최종 답변이 제공되지 않았습니다.")
        is_final = True

    else:
        tool = self.tool_registry.get(tool_name)
        if tool is None:
            result = f"알 수 없는 tool: {tool_name}"
            is_final = True  # 더 진행해봐야 의미 없으니 종료
        else:
            try:
                result = tool.run(user_query=user_query, tool_input=tool_input)
            except Exception as e:
                result = {"error": f"'{tool_name}' 실행 중 오류 발생: {e}"}
    return {
        "tool": tool_name,
        "tool_input": tool_input,
        "output": result,
        "reason": plan.get("reason", ""),
        "is_final": is_final,
    }
```

- 개선 후 효과
  - 책임 분리가 명확해져 유지보수성이 크게 향상됨
  - Agent 전체 흐름을 안정적으로 제어할 수 있음

05. Agent 개발을 통해 얻은 핵심 인사이트

설계–구현–개선을 통해 얻은 기술적 성장

- Planner–Executor–Tool 간 역할을 분리하며 Agent 시스템 설계에서 구조적 책임 분리의 중요성을 명확하게 이해할 수 있었습니다.
- 예외 상황에서도 Reasoning 흐름을 유지하기 위한 안정적인 설계 방식의 필요성을 실전 구현을 통해 체감했습니다.
- if-elif 기반 구조를 개선하면서 확장성과 일관성을 갖춘 Agent 환경을 구축하는 설계 패턴을 익힐 수 있었습니다.
- Tool 인터페이스를 통일하고 Registry 구조를 적용하며 시스템 안정성과 유지보수성을 강화하는 경험을 할 수 있었습니다.
- 이번 경험은 단순 기능 구현을 넘어, LLM의 '행동'을 설계하는 사고 방식을 갖추는 계기가 되었습니다.

본 프로젝트는 기존 RAG → Agent 구조로 진화시키며
설계·구현·리팩토링 전 과정을 스스로 진행한
Agentic 시스템 구축 경험입니다.

END

CHOI SUNGJANG

2025

PORTFOLIO

T. 010-8967-9399 | Github. https://github.com/GrowingChoi | Email. palga65@gmail.com | Blog. blog.naver.com/growingdev95