CHOI SUNGJANG

2026

PORTFOLIO

Backend-based AI Platform Engineer Portfolio

[Portfolio Roadmap]

01. Intro - AI 서비스를 바라보는 엔지니어링 기준
02. Project - RulebaseAgent (Agent 구조 설계)
03. Project - NAVI (문서 기반 AI 서비스 설계)

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

01. Intro

- AI 서비스를 바라보는 엔지니어링 기준

  저는 AI를 하나의 기능이라기보다 기존 서비스 위에서 안정적으로 운영되어야 하는 시스템으로 바라봅니다.
  공공·금융 도메인에서 백엔드 시스템을 개발하며, 서비스의 안정성은 알고리즘 자체보다 구조와 흐름 설계에서 결정되는 경우가 많다는 점을 여러 번 경험했습니다.

  AI 기능을 붙인 서비스들을 만들어보면서, 응답이 항상 같은 방식으로 나오지 않는다는 점을 자연스럽게 체감하게 됐습니다. 그래서 결과를 하나하나 통제하는 방식보다 입력부터 응답까지의 흐름이 흔들리지 않도록 감싸는 구조를 어떻게 만들지에 더 많은 시간을 쓰게 됐습니다. 그 이유는 모델이 어떤 파이프라인 안에서 동작하느냐가 서비스 품질을 좌우한다고 느꼈기 때문입니다.

  이 포트폴리오에는 이러한 관점이 실제 구현으로 어떻게 이어졌는지를 담았습니다.
  개인 프로젝트에서는 Tool-Using Agent 구조를 설계하며 AI 파이프라인의 확장성을 고민했고, 팀 프로젝트에서는 문서 기반 RAG 시스템을 구축하며 운영의 안정성을 고민하며 설계했습니다.

02. Project : RulebaseAgent

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

사용기술 : Python · FastAPI · OpenAI GPT-5-mini · Docker

1-1. 한 줄 요약

복잡한 규정 질의에 대해, LLM이 스스로 판단하고 필요한 도구를 선택해 실행하는 Agent 파이프라인

1-2. 문제 정의

규정·정책 질의는 단순 검색만으로는 해결되지 않는 경우가 많았습니다.
질문에 따라 검색 → 판단 → 요약 단계가 달라졌고, 단일 RAG 구조로는 이러한 흐름을 유연하게 처리하기 어려웠습니다. 특히 기능이 늘어날수록 조건 분기와 예외 처리 코드가 복잡해지며, 구조 자체가 빠르게 흔들리는 문제가 발생했습니다.

1-3. 접근 방식

문제를 "정답률"이 아닌 "구조의 안정성" 관점에서 다시 보았습니다.

- LLM은 판단에 집중
- 실행은 구조적으로 분리
- 기능이 늘어나도 기존 흐름에 영향이 최소화되는 구조 필요

이를 위해 Planner–Executor 구조의 Agent 파이프라인을 설계했습니다.

[목표]
- 기능 추가 시 기존 판단 로직 수정 없는 구조

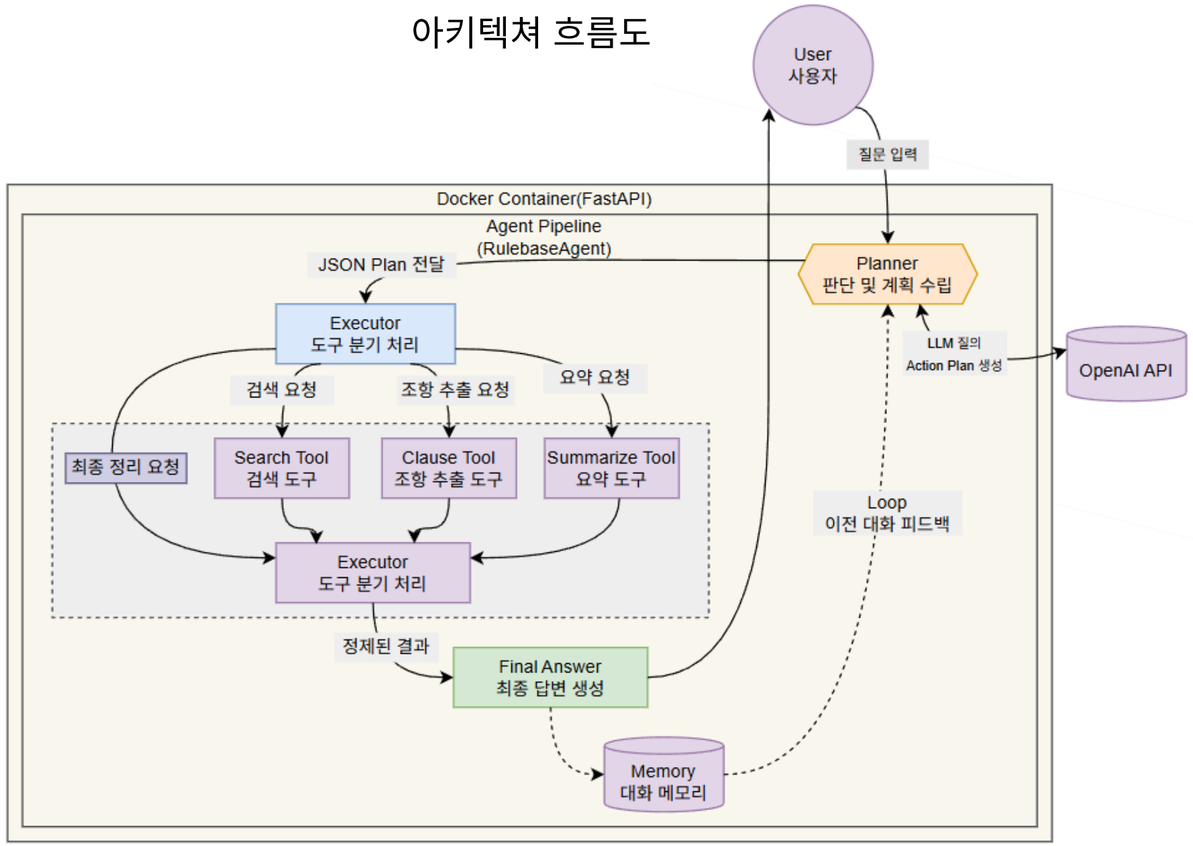
[핵심 아이디어]
- LLM은 판단, 실행은 시스템 책임으로 분리

[의사결정]
- 판단/실행 단계 분리
- Tool 단위 모듈화
- Action Plan 명시적 정의

[고려사항]
초기 설계 비용 증가
→ 확장 시 복잡도 급증 방지

02. Project : RulebaseAgent

아키텍쳐 흐름도



핵심 설계 포인트

1. 판단과 실행의 분리

   LLM은 사용자의 의도를 해석해 어떤 작업이 필요한지를 판단하는 역할만 맡고,
   실제 실행은 Executor가 담당하도록 구조를 분리했습니다.
   이를 통해 판단 로직과 실행 로직이 서로 얽히지 않도록 했습니다.

2. Tool 단위 확장성

   Search, Clause, Summary 기능을 각각 독립적인 Tool로 구성해서 필요한 기능만 선택적으로 실행할 수 있도록 했습니다.
   새로운 기능이 추가되더라도 기존 로직을 크게 수정하지 않도록 설계했습니다.

3. 서비스 관점 구현

   실험용 코드로 끝나지 않도록, FastAPI 기반의 REST API 형태로 기능을 구성했습니다. Docker 컨테이너 환경에서 바로 실행할 수 있게 정리해서 외부 서비스와 연동 가능한 구조를 염두에 두고 구현했습니다

이 프로젝트를 통해 얻은 관점

AI 서비스를 구현하면서, 모델 성능이나 개별 기능만큼이나 파이프라인 설계가 서비스 안정성에 큰 영향을 준다는 점을 체감했습니다.
NAVI와 RulebaseAgent 두 프로젝트를 진행하며, 구조가 정리되어 있을수록 문제를 파악하고 수정하는 과정이 훨씬 수월하다는 경험을 했습니다.

02. Project : RulebaseAgent

1) Planner

파일 경로 : app/agent/planner.py

```python
class Planner:
    def __init__(self, tool_names: List[str]):
        """
        [Planner 초기화]
        사용할 수 있는 Tool의 이름 목록(tool_names)을 받아 관리합니다.
        """
        self.tool_names = tool_names

    def plan(self, user_query: str, memory_context: str) -> Dict[str, Any]:
        """
        [Prompt Engineering & Parsing]
        LLM에게 JSON 포맷으로 계획을 수립하도록 요청하고, 응답을 파싱합니다.
        """
        tools_str = ", ".join(self.tool_names)

        # System Prompt: JSON 형식을 강제하고 Tool 사용법을 주입
        prompt = f"""
        당신은 Tool-Using Agent의 플래너입니다.
        다음에 실행할 '단 한 번의' 액션을 JSON으로만 반환하세요.
        [사용 가능한 Tool 목록]: {tools_str}
        ...
        """

        # OpenAI API 호출
        response = client.chat.completions.create(
            model=MODEL_NAME,
            messages=[{"role": "user", "content": prompt}]
        )

        # JSON 파싱 및 예외 처리 (Fallback Mechanism)
        try:
            return json.loads(response.choices[0].message.content.strip())
        except json.JSONDecodeError:
            # 파싱 실패 시 '기본 검색'으로 안전하게 유도
            return {"tool": "search", "tool_input": {"query": user_query}}
```

도구 목록을 받아 LLM에게 JSON 출력을 강제하고, 파싱 실패 시 예외 처리를 수행하는 핵심 로직입니다. (프롬프트 일부는 너무 길어서 중략했습니다.)

2) Executor

파일 경로 : app/agent/executor.py

```python
class Executor:
    def __init__(self, tool_registry: Dict[str, Tool]):
        # [Tool Registry] 문자열 Key로 도구 객체 관리
        self.tool_registry = tool_registry

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

        # 1. 최종 답변 처리 (Final Answer)
        if tool_name == "final_answer":
            return {
                "output": tool_input.get("answer"),
                "is_final": True
            }

        # 2. 도구 조회 및 실행 (Routing)
        tool = self.tool_registry.get(tool_name)
        if tool:
            try:
                # 실제 도구 실행 (Run Tool)
                result = tool.run(user_query=user_query, tool_input=tool_input)
            except Exception as e:
                result = {"error": f"실행 중 오류 발생: {e}"}

        return {"tool": tool_name, "output": result, "is_final": False}
```

Planner의 결과를 받아 실제 도구 객체를 찾아 실행하거나, 최종 답변을 반환합니다.

02. Project : RulebaseAgent

3) BaseTool

파일 경로 : app/agent/tools/base.py

```python
from abc import ABC, abstractmethod

class Tool(ABC):
    """
    [Interface Definition]
    모든 Tool이 따라야 하는 공통 인터페이스 (Abstract Base Class)
    """
    name: str
    description: str = ""

    @abstractmethod
    def run(self, *, user_query: str, tool_input: Dict[str, Any]) -> Any:
        """
        실제 실행 로직.
        Planner가 넘겨준 tool_input을 받아 구체적인 기능을 수행합니다.
        """
        raise NotImplementedError
```

Python 표준 abc 모듈을 활용하여 도구의 공통 규격을 정의했습니다. 모든 도구가 run 메서드를 필수로 구현하도록 강제함으로써, 기능이 확장되더라도 시스템의 일관성이 유지되도록 설계했습니다.

4) 대화 컨텍스트 메모리

파일 경로 : app/agent/memory.py

```python
from typing import List, Dict

class ConversationMemory:
    """
    [Context Management]
    Planner가 이전 대화의 맥락(Context)을 파악할 수 있도록
    대화 히스토리를 저장하고 관리하는 메모리 모듈
    """
    def __init__(self, max_turns: int = 5):
        # 무한히 길어지는 것을 방지하기 위한 Sliding Window 적용
        self.max_turns = max_turns
        self._history: List[Dict[str, str]] = []

    def add_turn(self, user: str, agent: str) -> None:
        """사용자 질문과 에이전트 답변을 한 쌍으로 저장"""
        self._history.append({"user": user, "agent": agent})
        if len(self._history) > self.max_turns:
            self._history = self._history[-self.max_turns :]

    def get_context_str(self) -> str:
        """
        [Prompt Injection]
        LLM 프롬프트에 주입하기 최적화된 문자열 포맷으로 변환
        """
        if not self._history:
            return ""

        lines = []
        for turn in self._history:
            lines.append(f"User: {turn['user']}")
            lines.append(f"Agent: {turn['agent']}")

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

LLM은 이전 대화를 기억하지 못합니다. 그래서 이를 보완하기 위해 최근 대화 N개(최대값=5)만 핵심적으로 기억하는 '슬라이딩 윈도우' 방식을 적용하여 불필요한 토큰 비용도 절약하고 Planner가 이전 대화의 맥락을 놓치지 않고 대화를 이어갈 수 있도록 설계했습니다.

03. Project : NAVI

프로젝트 명 : NAVI (사내 규정 문서 기반 RAG 시스템)

사용기술
Python · Django · Django REST Framework · OpenAI API (GPT-4o-mini) · Sentence-Transformers (KoE5) · Qdrant (Vector DB) · Docker Compose · Nginx · AWS S3

1-1. 한 줄 요약

비정형 규정 문서를 구조화해 검색 실패율을 낮추고, 질의에 대응하는 사내 규정 검색 RAG 시스템

1-2. 문제 정의

사내 규정 문서는 대부분 PDF 형태로 관리되고 있었고, 문서 수가 늘어날수록 수작업 검색과 오류가 빠르게 증가했습니다.
기존 검색 방식은

- 질의 의도와 무관한 문서가 함께 검색
- 문서 유형 차이로 인해 정확한 응답을 만들기 어려움

단순 Vector Search만으로는 실무 환경에서 요구하는 정확도와 안정성을 확보하기 어렵다고 판단했습니다.

1-3. 접근 방식

문제를 모델 성능이 아닌 데이터 흐름과 검색 구조 관점에서 재정의했습니다.

- 문서를 먼저 구조화
- 질의 의도에 맞는 문서만 선별 검색
- 검색·재정렬·응답 단계를 명확히 분리

이를 통해 운영 환경에서도 흔들리지 않는 RAG 파이프라인을 목표로 설계했습니다.

[목표]

- 문서 수와 질의 패턴이 늘어나도 검색 품질과 응답 흐름이 쉽게 흔들리지 않는 구조

[핵심 아이디어]

- 검색 정확도를 모델에 맡기기보다, 검색 이전과 이후 흐름을 구조로 먼저 통제

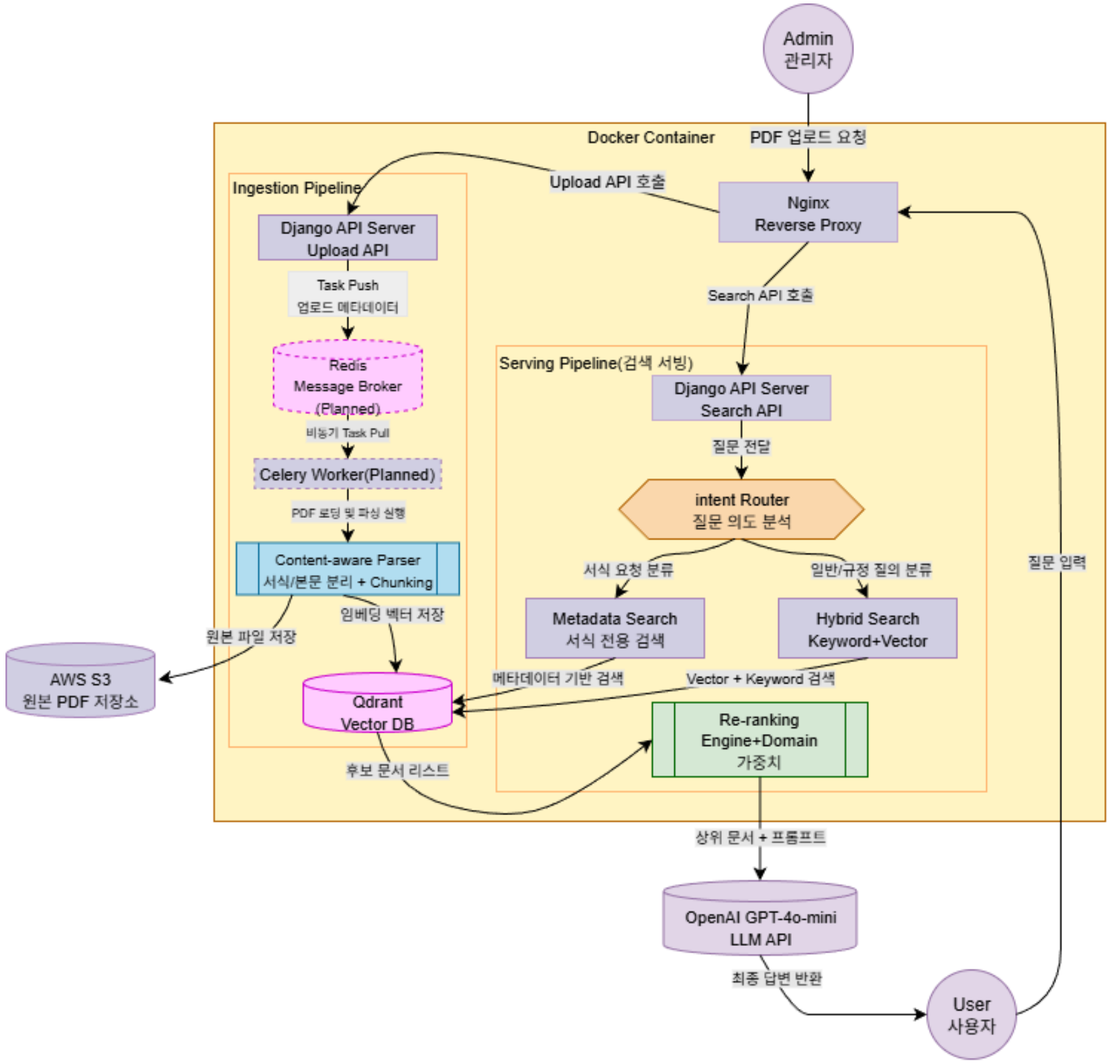
[의사결정]

- 검색 이전 단계에서 문서 전처리 파이프라인 분리
- 질의 의도에 맞지 않는 문서는 메타데이터 기준으로 사전 제외
- 검색 이후 단계도 하나로 묶지 않고 분리

[고려사항]
전처리 및 구조 설계에 초기 작업 비용 발생
→ 운영 중 수정 범위와 영향도 감소

03. Project : NAVI

아키텍쳐 흐름도



핵심 설계 포인트

1. 전처리 파이프라인의 분리

   처음에는 PDF에서 텍스트만 추출해 사용했는데, 이 방식으로는 문서의 서식이나 문단 구조가 깨지는 문제가 있었습니다.

   그래서 검색 전에 문서 구조를 먼저 정리하는 게 필요하다고 판단했고, 서식과 본문을 함께 인식하는 전처리 단계를 별도로 분리해 구성했습니다.

2. 메타데이터 기반 필터링

   Vector Search만 사용했을 때, 질문 의도와 상관없는 문서가 함께 검색되는 경우가 자주 발생했습니다.

   이를 보완하기 위해 문서 유형과 도메인 정보를 메타데이터로 관리했고, 검색 단계 이전에 한 번 더 걸러주는 방식으로 검색 범위를 줄였습니다.

3. 검색·재정렬·응답 단계 분리

   검색부터 응답 생성까지 하나의 흐름으로 묶여 있으면, 어느 지점에서 문제가 생겼는지 파악하기가 어려웠습니다.

   그래서 검색, 재정렬, 응답 생성 단계를 나누어 구성했고, 특정 단계만 수정하거나 조정할 수 있도록 구조를 정리했습니다.

4. 운영을 고려한 백엔드 구조

   실험용 코드에 그치지 않도록, 처음부터 서비스 형태를 염두에 두고 구조를 잡았습니다.

   Django REST Framework 기반으로 API를 구성했고, 로컬과 배포 환경의 차이를 줄이기 위해 Docker Compose 환경에서 실행 가능하도록 구성했습니다.

5. 구조를 먼저 고정하고, 세부를 조정하는 접근

   파라미터를 계속 조정하는 방식보다는, 전체 데이터 흐름을 먼저 정리하는 게 더 중요하다고 느꼈습니다.

   그래서 파이프라인 구조를 먼저 확정한 뒤, 필요한 부분만 세부적으로 조정하는 방식으로 접근했습니다.

03. Project : NAVI

1) 질의 처리 및 하이브리드 검색 흐름

파일 경로 : backend/chatbot/services/rag_search.py

```python
def hybrid_search(self, query: str, domain_list: List[str] = None,
                  file_types: List[str] = None, min_recency: int = None,
                  top_k: int = None) -> List[Dict[str, Any]]:
    """
    [하이브리드 검색 구현]
    사용자 질의에 대해 도메인/문서타입/최신성 필터와 벡터 검색을 결합합니다.
    """
    # 1. 메타데이터 필터(Metadata Filter) 동적 생성
    filters = []
    if domain_list:
        filters.append({"key": "domain_primary", "match": {"any": domain_list}})

    if min_recency:
        filters.append({"key": "recency_score", "range": {"gte": min_recency}})

    query_filter = {"must": filters} if filters else None

    # 2. Vector Search 실행 (Qdrant)
    results = self.search(query, flt=query_filter, top_k=top_k)

    # 3. 비즈니스 로직 기반 재순위화 (Reranking)
    if results:
        results = self._rerank_results(results, query, domain_list)

    return results
```

단순 벡터 검색만으로는 어려운 정확한 필터링을 위해 '의미 기반 검색'과 '조건 필터링'을 결합한 부분입니다.

2) 비지니스 로직 기반 재순위화

파일 경로 : backend/chatbot/services/rag_search.py

```python
def _rerank_results(self, results: List[Dict[str, Any]], query: str,
                    domain_list: List[str] = None) -> List[Dict[str, Any]]:
    """
    [Custom Reranking Logic]
    Vector 유사도 점수 + 도메인 가중치 + 최신성 점수를 합산하여 재정렬합니다.
    """
    for result in results:
        # 가중치 1: 사용자 부서/도메인과 문서 도메인 일치 시 가산점
        domain_score = 0
        if domain_list and result.get('domain_primary') in domain_list:
            domain_score = 2.0  # 도메인 일치 시 높은 가중치 부여

        # 가중치 2: 문서 최신성(Recency) 점수 반영 (최신일수록 높은 점수)
        recency_score = result.get('recency_score', 1)

        # 최종 점수 = 벡터 유사도 + 도메인 점수 + 최신성 점수
        result['final_score'] = result['score'] + domain_score + (recency_score * 0.1)

    # 최종 점수 기준 내림차순 정렬
    results.sort(key=lambda x: x.get('final_score', 0), reverse=True)
    return results
```

비슷한 문장을 추출하는 것이 아닌 사용자 질의에 얼마나 부합한 문서인지를 판별하는 단계입니다. 이를 통해서 유사도는 높지만 옛날버전의 문서가 상단에 노출되는 문제를 보정했습니다.

03. Project : NAVI

3) 정규식 기반 서식 추출 파이프라인

파일 경로 : backend/pdf_form_extractor_v6.py

```python
class PDFFormExtractorV6:
    def __init__(self, input_dir, output_dir):
        # 1. [Standard Patterns] 표준 서식 식별을 위한 정규식
        self.form_start_patterns = [
            r'^\s*\[별지\s*제\d+호\s*서식\]',  # 표준: [별지 제1호 서식]
            r'^\s*\[별표\s*\d*\]',             # 표준: [별표 1]
            r'^\s*\[부록\s*\d*\]',             # 표준: [부록]
        ]

        # 2. [Edge Cases] 정규식으로 식별 불가능한 예외 케이스 수동 관리
        # (OCR 인식 오류, 비표준 줄바꿈, 특수 기호 누락 등)
        self.manual_exceptions = [
            "별지 제10호의2 서식",  # '의2' 같은 파생 서식 패턴
            "부  록 (제5조 관련)",  # 중간 공백이 불규칙한 경우
            "첨부 서식 1."          # 대괄호([])가 누락된 경우
        ]

    def is_form_page(self, text):
        """
        [Hybrid Detection]
        정규식 패턴 매칭 실패 시, 예외 리스트를 2차로 확인하여 데이터 유실 방지
        """
        if not text: return False
        first_line = text.split('\n')[0].strip()

        # Step 1: 정규식 기반 표준 패턴 검사
        for pattern in self.form_start_patterns:
            if re.search(pattern, first_line, re.IGNORECASE):
                return True

        # Step 2: 수동 예외 케이스 검사 (Fallback)
        # 정규식 패턴을 벗어난 비정형 서식 제목 처리
        if any(ex in first_line for ex in self.manual_exceptions):
            return True

        return False
```

정규식 기반의 패턴 매칭을 기본으로 하되, OCR 오류나 비표준 서식 등 정규식으로 놓칠 수 있는 예외 케이스를 수동 리스트로 관리하여 서식 추출 누락을 최소화했습니다.

4) 질의 의도 분석

파일 경로 : backend/chatbot/services/pipeline.py

```python
def _determine_search_strategy(query: str, keywords: List[str], estimated_domains: List[str]):
    """
    [Intent Router]
    사용자 질의와 키워드를 분석하여 최적의 검색 전략(Strategy)을 결정합니다.
    """
    # Strategy 1: 서식/양식 관련 질의 -> 서식 전용 검색 모드
    if _is_form_related_query(query, keywords):
        return {
            'type': 'form_specific',
            'priority': 'forms_first'
        }

    # Strategy 2: 특정 도메인이 명확한 경우 -> 도메인 필터링 검색
    if estimated_domains and len(estimated_domains) == 1:
        return {
            'type': 'domain_specific',
            'domain': estimated_domains[0]
        }

    # Strategy 3: 일반 질의 -> 하이브리드 검색 (Vector + Keyword)
    return {
        'type': 'hybrid',
        'domain_list': estimated_domains
    }
```

사용자 질의 패턴을 분석하여 최적의 검색 전략을 수행함으로써, 불필요한 탐색을 줄이고 검색 정확도를 높이는 관문 역할을 합니다.

구조를 먼저 고민하며,
AI 서비스를 구성하는 파이프라인을 설계하고 구현해 본
프로젝트입니다.

END

CHOI SUNGJANG
2026
PORTFOLIO

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