Day 7: Orchestration (3-Stage Pipeline) - 완전 가이드

> **GitHub Repository**: RAG-on-Local-CPU-minimal

> RAG 파이프라인 고도화 - 제어 가능한 구조

💡 한 줄 정의

> **"이 파이프라인은 '답변을 잘 만들기 위한 코드'가 아니라 'AI를 통제하기 위한 구조'다."**

🎯 학습 목표

Day 7의 핵심은 다음과 같습니다:

> **"RAG를 3단계로 분리하고, 각 단계를 추적 가능하게 만든다"**

이 Day를 마치면:

- ☑ Orchestration 패턴 이해
- ☑ 3-Stage Pipeline 구조
- ☑ Trace 객체로 추적
- ☑ 단계별 책임 분리
- ☑ 확장 가능한 구조 설계

핵심: Retrieve → Rerank → Generate

📂 프로젝트 구조

```
Rag_minimal_day7/
├─ main.py            # FastAPI 진입점 (API Boundary)
├─ pipeline.py        # 파이프라인 진입 (Entry Point)
├─ orchestration.py   # 제어 타워 ← 🧠 핵심
├─ models.py          # Trace 객체 ← 신규
├─ embedding.py       # 임베딩 생성
├─ vectorstore.py     # Vector 검색
├─ keyword_index.py   # Keyword 검색 ← 신규
├─ reranker.py        # Score Fusion
├─ ingest.py          # 자산 생성
├─ loader.py          # 문서 로딩
├─ llm.py             # LLM Stub
├─ config.py          # 설정
└─ data/
   └─ docs.txt        # 입력 문서
```

🤔 왜 이런 구조인가?

Day 6까지의 상태

단일 파이프라인:

```
def pipeline(query):
    candidates = vector_search(query)
    contexts = rerank(candidates)
    return contexts
```

문제점:

- 단계 구분 불명확
- 디버깅 어려움
- 중간 결과 추적 불가
- 확장 어려움

Day 7에서 추가되는 것

☑ Orchestration 패턴

```
def run_pipeline(query):
    # 1단계: Retrieve (추적 가능)
    retrieve_result = step_retrieve(query)

    # 2단계: Rerank (추적 가능)
    rerank_result = step_rerank(query, retrieve_result)

    # 3단계: Generate (추적 가능)
    generate_result = step_generate(query, rerank_result)

    # 전체 추적 객체
    return PipelineTrace(
        query=query,
        retrieve=retrieve_result,
        rerank=rerank_result,
        generate=generate_result
    )
```

개선점:

- ☑ 단계 분리 명확
- ☑ 디버깅 가능
- ☑ 관측 가능
- ☑ 정책/게이트웨이/에이전트 확장 가능

1. 왜 Orchestration이 필요한가?

Orchestration의 정의:

> 여러 독립적인 단계를 정해진 순서대로 실행하고, 각 단계의 결과를 추적하는 패턴

필요한 이유:

1) AI는 예측 불가능:

```
같은 질문에도:
- 검색 결과 다름
- 재정렬 순서 다름
- LLM 답변 다름

→ 각 단계를 추적해야 문제 파악 가능
```

2) 품질 관리:

```
잘못된 답변이 나왔을 때:
- Retrieve가 문제? → 후보 10개 확인
- Rerank가 문제? → 3개 선택 확인
- Generate가 문제? → 프롬프트 확인

→ Trace 객체로 즉시 파악
```

3) 실무 확장:

```
초기: Retrieve → Rerank → Generate
확장: Retrieve → 정책 체크 → Rerank → 승인 → Generate

→ Orchestration 구조면 단계만 추가
```

2. 왜 3단계로 나눴나?

STEP 1: Retrieve (검색)

- **목적:** Recall 확보 (놓치지 않기)
- **전략:** 넓게, 빠르게
- **결과:** 10개 후보

STEP 2: Rerank (재정렬)

- **목적:** Precision 확보 (정확하게)
- **전략:** 정밀하게, 느려도 됨
- **결과:** 3개 컨텍스트

STEP 3: Generate (생성)

- **목적:** 최종 답변
- **전략:** 컨텍스트 기반 생성

- **결과:** 1개 답변

왜 더 나누지 않나?

- 3단계가 표준 (실무 검증됨)
- 더 나누면 복잡도 증가
- 확장 시 중간 단계 추가 가능

3. 왜 Trace 객체가 필요한가?

Trace 객체란?

> 각 단계의 결과를 저장하는 데이터 클래스

구조:

```
@dataclass
class PipelineTrace:
    query: str                  # 질문
    retrieve: RetrieveResult    # 1단계 결과
    rerank: RerankResult        # 2단계 결과
    generate: GenerateResult    # 3단계 결과
```

활용:

1) 디버깅:

```
trace = run_pipeline("RAG란?")

print("후보:", trace.retrieve.candidates)
# → 10개 확인

print("선택:", trace.rerank.contexts)
# → 3개 확인

print("답변:", trace.generate.answer)
# → 최종 답변 확인
```

2) 품질 검증:

```
# 후보에 정답이 있었나?
if "RAG" in " ".join(trace.retrieve.candidates):
    print("✅ Retrieve 성공")
else:
    print("❌ Retrieve 실패")

# 재정렬이 올바른가?
if "RAG" in trace.rerank.contexts[0]:
```

```
    print("☑ Rerank 성공")
else:
    print("❌ Rerank 실패")
```

3) A/B 테스트:

```
# A안: Vector만
trace_a = run_pipeline_vector_only(query)

# B안: Vector + Keyword
trace_b = run_pipeline_hybrid(query)

# 비교
compare(trace_a, trace_b)
```

4. 왜 책임을 분리했나?

파일별 책임:

main.py - API Boundary

```
@app.post("/chat")
def chat_api(req):
    return pipeline(req.query)
```

- 역할: HTTP 요청만 처리
- 로직: ❌ 없음

pipeline.py - Entry Point

```
def pipeline(query):
    return run_pipeline(query)
```

- 역할: 진입점만
- 로직: ❌ 없음

orchestration.py - Control Tower

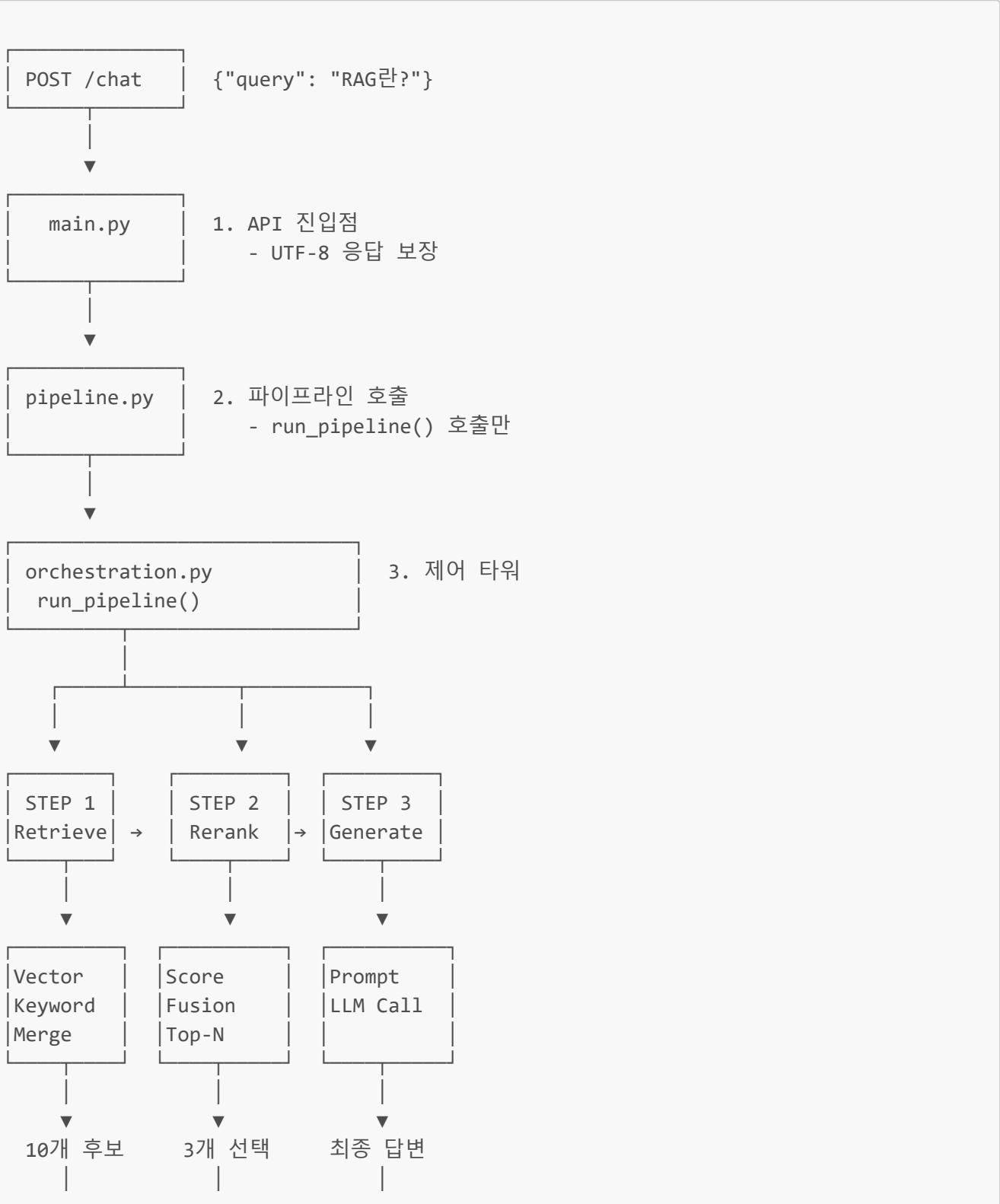
```
def run_pipeline(query):
    r = step_retrieve(query)
    rr = step_rerank(query, r)
    g = step_generate(query, rr)
    return PipelineTrace(...)
```

- 역할: 전체 흐름 제어
- 로직: ☑ 여기만!

왜 이렇게?

- 변경 지점 명확
- 테스트 용이
- 확장 안전

🔄 실행 흐름도



```
        └──────────┬──────────┘
                   ▼
        ┌──────────────────┐
        │PipelineTrace     │  4. 전체 추적
        │  - retrieve      │
        │  - rerank        │
        │  - generate      │
        └──────────────────┘
                   │
                   ▼
              JSON 응답
```

⚙ 설치 및 실행

1. 패키지 설치

```
cd Rag_minimal_day7
pip install fastapi uvicorn sentence-transformers faiss-cpu rank-bm25
```

2. 서버 실행

```
uvicorn main:app --reload
```

초기 로딩:

```
INFO: Loading assets...
INFO: Vector store built
INFO: Keyword index built
INFO: Reranker initialized
INFO: Server ready
```

3. API 테스트

curl (PowerShell):

```
curl -X POST "http://127.0.0.1:8000/chat" `
  -H "Content-Type: application/json" `
  -d '{"query":"RAG란 무엇인가?"}'
```

예상 응답:

```
{
  "query": "RAG란 무엇인가?",
  "retrieve": {
    "candidates": ["...", "...", "..."]
  },
  "rerank": {
    "contexts": ["...", "...", "..."]
  },
  "generate": {
    "answer": "RAG는..."
  }
}
```

🧠 핵심 개념 Deep Dive

1. Orchestration 패턴 상세

제어 흐름:

```python
def run_pipeline(query: str) -> PipelineTrace:
    # 1단계: Retrieve
    retrieve_result = step_retrieve(query)

    # 2단계: Rerank (1단계 결과 사용)
    rerank_result = step_rerank(query, retrieve_result)

    # 3단계: Generate (2단계 결과 사용)
    generate_result = step_generate(query, rerank_result)

    # 전체 추적
    return PipelineTrace(
        query=query,
        retrieve=retrieve_result,
        rerank=rerank_result,
        generate=generate_result
    )
```

핵심 원칙:

- 각 단계는 독립적
- 결과는 Trace 객체로 반환
- 다음 단계로 명시적 전달

2. 각 단계 상세

STEP 1: Retrieve (검색)

```
def step_retrieve(query: str) -> RetrieveResult:
    store, keyword, _ = get_assets()

    # Query 임베딩
    qvec = embed([query])[0]

    # 1-1. Vector Search
    vector_candidates = store.search(qvec, VECTOR_TOP_K)

    # 1-2. Keyword Search
    keyword_candidates = keyword.search(query, KEYWORD_TOP_K)

    # 1-3. 후보 병합 (중복 제거)
    merged = list(dict.fromkeys(
        vector_candidates + keyword_candidates
    ))

    print("CANDIDATES:", merged)

    return RetrieveResult(candidates=merged)
```

핵심:

- Hybrid Search 수행
- Recall 중시 (놓치지 않기)
- 중복 제거

STEP 2: Rerank (재정렬)

```
def step_rerank(query: str, r: RetrieveResult) -> RerankResult:
    store, keyword, reranker = get_assets()

    # 2-1. Vector 유사도 계산
    vector_scores = {}
    qvec = embed([query])[0]
    for c in r.candidates:
        vector_scores[c] = qvec @ embed([c])[0]

    # 2-2. Keyword 매칭 점수
    keyword_scores = {}
    for c in r.candidates:
        keyword_scores[c] = sum(
            1 for t in query.split() if t in c
        )

    # 2-3. Score Fusion
    contexts = reranker.rerank(
        query,
        r.candidates,
        vector_scores,
        keyword_scores,
```

```
        RERANK_TOP_N
    )

    print("RERANKED:", contexts)

    return RerankResult(contexts=contexts)
```

핵심:

- 두 가지 점수 계산
- Score Fusion (가중치 결합)
- Precision 중시 (정확하게)

STEP 3: Generate (생성)

```
def step_generate(query: str, rr: RerankResult) -> GenerateResult:
    # 3-1. Context Join
    context = "\n".join(rr.contexts)

    # 3-2. Prompt 구성
    prompt = f"""
문서:
{context}

질문:
{query}
"""

    # 3-3. LLM 호출
    answer = generate(prompt)

    return GenerateResult(answer=answer)
```

핵심:

- 컨텍스트 조합
- 프롬프트 템플릿
- LLM 호출 (Day 7은 Stub)

3. Trace 객체 구조

```
# models.py

@dataclass
class RetrieveResult:
    candidates: List[str]  # Vector + Keyword 병합

@dataclass
class RerankResult:
```

```
    contexts: List[str]    # Top-N 선택

@dataclass
class GenerateResult:
    answer: str            # 최종 답변

@dataclass
class PipelineTrace:
    query: str
    retrieve: RetrieveResult
    rerank: RerankResult
    generate: GenerateResult
```

특징:

- `@dataclass`: 자동으로 `__init__`, `__repr__` 생성
- 불변성: 각 단계 결과 보존
- 타입 힌트: IDE 지원

📖 코드 Deep Dive

config.py

```
# Retrieval 설정
MODEL_NAME = "all-MiniLM-L6-v2"
EMBEDDING_DIM = 384

# Retrieve 단계
VECTOR_TOP_K = 8      # Vector Search
KEYWORD_TOP_K = 8     # Keyword Search

# Rerank 단계
RERANK_TOP_N = 3      # 최종 선택

DATA_PATH = "data/docs.txt"
```

keyword_index.py (신규 파일)

```
from typing import List

class KeywordIndex:
    def __init__(self, documents: List[str]):
        """
        Keyword 기반 간이 검색 인덱스

        Args:
            documents: 전체 문서 리스트
        """
```

```
        self.documents = documents

    def search(self, query: str, top_k: int) -> List[str]:
        """
        Keyword 매칭 기반 검색

        Args:
            query: 질문
            top_k: 반환할 개수

        Returns:
            candidates: 매칭된 문서들
        """
        query_terms = set(query.lower().split())

        # 각 문서의 매칭 점수 계산
        scored = []
        for doc in self.documents:
            doc_terms = set(doc.lower().split())

            # 공통 단어 개수
            overlap = len(query_terms & doc_terms)

            if overlap > 0:
                scored.append((doc, overlap))

        # 점수 기준 정렬
        scored.sort(key=lambda x: x[1], reverse=True)

        # Top-K 반환
        return [doc for doc, score in scored[:top_k]]
```

핵심:

- 간단한 Keyword 검색
- 공통 단어 개수로 스코어링
- 교육용 최소 구현

reranker.py (Score Fusion)

```
from typing import List, Dict

class Reranker:
    def __init__(self, alpha: float = 0.7, beta: float = 0.3):
        """
        Score Fusion Reranker

        Args:
            alpha: Vector 점수 가중치
            beta: Keyword 점수 가중치
        """
```

```
        self.alpha = alpha
        self.beta = beta

    def rerank(
        self,
        query: str,
        candidates: List[str],
        vector_scores: Dict[str, float],
        keyword_scores: Dict[str, float],
        top_n: int
    ) -> List[str]:
        """
        Score Fusion 기반 재정렬

        Args:
            query: 질문
            candidates: 후보 문서들
            vector_scores: Vector 유사도 점수
            keyword_scores: Keyword 매칭 점수
            top_n: 반환할 개수

        Returns:
            contexts: 재정렬된 Top-N
        """
        # 정규화 (0~1)
        max_vector = max(vector_scores.values()) if vector_scores else 1
        max_keyword = max(keyword_scores.values()) if keyword_scores else 1

        # Score Fusion
        fused = []
        for c in candidates:
            v_score = vector_scores.get(c, 0) / max_vector
            k_score = keyword_scores.get(c, 0) / max_keyword

            # α * vector + β * keyword
            final_score = self.alpha * v_score + self.beta * k_score

            fused.append((c, final_score))

        # 정렬
        fused.sort(key=lambda x: x[1], reverse=True)

        # Top-N 반환
        return [doc for doc, score in fused[:top_n]]
```

핵심:

- 두 점수 정규화
- 가중치 결합 (α, β)
- Top-N 선택

💡 학습 효율을 높이는 팁

1. 터미널 로그 확인

orchestration.py의 print 출력:

```
CANDIDATES: ['Doc A', 'Doc B', 'Doc C', ..., 'Doc J']
RERANKED: ['Doc B', 'Doc A', 'Doc E']
```

분석:

- Retrieve: 10개 수집
- Rerank: 순서 변경
- B가 1등으로 상승

2. Trace 객체 확인

Python 콘솔:

```
from pipeline import pipeline

trace = pipeline("RAG란?")

print("질문:", trace.query)
print("후보 개수:", len(trace.retrieve.candidates))
print("선택 개수:", len(trace.rerank.contexts))
print("답변:", trace.generate.answer)
```

3. 단계별 실행 시간 측정

orchestration.py 수정:

```
import time

def run_pipeline(query):
    start = time.time()

    r = step_retrieve(query)
    t1 = time.time() - start

    rr = step_rerank(query, r)
    t2 = time.time() - start - t1

    g = step_generate(query, rr)
    t3 = time.time() - start - t1 - t2

    print(f"Retrieve: {t1:.2f}s")
    print(f"Rerank: {t2:.2f}s")
```

```
    print(f"Generate: {t3:.2f}s")

    return PipelineTrace(...)
```

🔧 직접 해보기: 실습

실습 1: TOP_K, TOP_N 조정

config.py:

```
# 실험 1: 넓게
VECTOR_TOP_K = 15
KEYWORD_TOP_K = 15
RERANK_TOP_N = 5

# 실험 2: 좁게
VECTOR_TOP_K = 5
KEYWORD_TOP_K = 5
RERANK_TOP_N = 2
```

비교:

- 검색 품질
- 응답 시간

실습 2: Score Fusion 가중치 변경

reranker.py:

```
# 실험 1: Vector 중시
alpha = 0.9
beta = 0.1

# 실험 2: Keyword 중시
alpha = 0.3
beta = 0.7

# 실험 3: 균등
alpha = 0.5
beta = 0.5
```

비교:

- 재정렬 결과
- 최종 답변 품질

실습 3: 중간 단계 추가

orchestration.py:

```
def step_filter(r: RetrieveResult) -> RetrieveResult:
    """정책 체크 단계 (예시)"""
    filtered = [
        c for c in r.candidates
        if len(c) > 50  # 너무 짧은 문서 제거
    ]
    return RetrieveResult(candidates=filtered)

def run_pipeline(query):
    r = step_retrieve(query)
    r = step_filter(r)  # ← 신규 단계
    rr = step_rerank(query, r)
    g = step_generate(query, rr)
    return PipelineTrace(...)
```

☑ Day 7 완료 기준

다음을 설명할 수 있으면 통과입니다:

5가지 필수 질문

1. **Orchestration이란 무엇인가?** ☑ 여러 단계를 순서대로 실행하고 추적하는 패턴
2. **왜 3단계로 나눴나?** ☑ Retrieve (넓게) → Rerank (정밀) → Generate (최종)
3. **Trace 객체의 역할은?** ☑ 각 단계 결과 저장, 디버깅/품질 검증/A/B 테스트
4. **orchestration.py의 책임은?** ☑ 전체 흐름 제어, 단계 순서 정의
5. **확장은 어떻게?** ☑ 단계 추가만 하면 됨 (정책, 게이트웨이 등)

➡ SOON 다음 단계 (Day 8)

Day 8에서는:

LLM Gateway 패턴:

```
# 외부 LLM API 연결
gateway = LLMGateway(
    provider="openai",  # or "anthropic"
    api_key=os.getenv("API_KEY")
)

# 로컬 LLM 연결
gateway = LLMGateway(
    provider="local",
    model_path="./models/tinyllama.gguf"
```

```
)

# Gateway 사용
response = gateway.generate(prompt)
```

추가 기능:

- OpenAI/Anthropic API 실제 연결
- API Key 관리
- 로컬 LLM 통합
- 모델 교체 용이

💬 학습 후 자가 점검

모두 답할 수 있다면 Day 7 완료! 🎉

1. **Orchestration 패턴이란?** ☑ 단계 분리 + 순서 제어 + 결과 추적
2. **3단계의 이름과 역할은?** ☑ Retrieve(검색), Rerank(재정렬), Generate(생성)
3. **Trace 객체 구조는?** ☑ query, retrieve, rerank, generate
4. **왜 책임을 분리했나?** ☑ 변경 지점 명확, 테스트 용이, 확장 안전
5. **실무 확장은 어떻게?** ☑ 정책 체크, 승인 단계, 게이트웨이 등 추가

📜 라이선스

⭐ 이 프로젝트가 도움이 되었다면 Star를 눌러주세요!