

실패의 원인을 깊이 파고들어 해결하는 것을 좋아하는 개발자

각 프로젝트에서 마주한 문제, 해결을 위한 시도와 결과, 그에 따른 판단과 구현 과정을 담았습니다.

김재형

AI Application Developer · Backend Engineer

kkuldangi2@gmail.com · martinel2.github.io

RAG 근거 충족률을 66.25%에서 91.25%로, AI 문서 편집 품질을 82.5%에서 91.2%로 높이며, AI가 틀린 근거로 답하거나 사용자 문서를 함부로 바꾸지 않도록 평가·승인·복구 구조를 설계해 온 AI 애플리케이션 개발자입니다.

정확도와 비용, 응답 시간의 트레이드오프를 평가로 확인한 뒤 서비스에 맞는 구현을 선택합니다.

RAG 근거 선택: 근거 충족률 66.25% → 91.25%, 답이 없는 질문에 근거를 붙인 경우 20 → 0건

근거가 틀리면 답변 전체를 믿을 수 없는 서비스라, 근거를 고르는 단계를 판단 전용 모델 Jev로 교체했습니다. GPT로 고르면 비용·시간이 크고 검색 가중치 조정은 개선 폭이 작았기 때문입니다. 동일 후보·블라인드 교차 채점으로 비교했고, 후보 수와 병렬 수를 조정해 선택 시간 중앙값을 9.090초에서 1.137초로 줄였습니다.

에이전트 실행 환경: 사용자 정의 Skill과 승인 기반 실행, 편집 품질 82.5% → 91.2%(114건 평가)

사용자마다 다른 반복 작업을 Skill로 직접 만들게 하되, AI가 문서를 함부로 바꾸면 서비스를 믿고 맡길 수 없어 권한·프롬프트 주입 검사는 코드가, 지침과 수정안은 AI가, 게시와 반영은 사용자가 맡도록 나눴습니다. 코드 검사만으로는 요청 누락을 못 잡아 의미 평가 후 한 번만 다시 쓰게 했고, 사용자가 승인하기 전에는 문서를 바꾸지 않습니다.

데이터 파이프라인: PDF 변환 통과율 45.17% → 89.89%, 약품 변환 비용 예상 \$200 대비 \$11.18

논문의 표·수식이 깨지면 답변 근거가 되는 수치를 쓸 수 없어, 본문은 빠른 AnyDoc으로 추출하고 표·수식은 원본 영역을 따로 판독하도록 처리 경로를 나눴습니다. 약품 데이터는 같은 문장을 한 번만 변환하고, 실시간 응답이 필요 없는 작업이라 Batch API를 써 약 4만 4천 건을 예산 안에서 처리했습니다.

목차

시간이 없으시면 01-02-04를 먼저 보시면 됩니다. 표지의 세 가지가 각각 품질 평가, 에이전트 실행 환경, 데이터 파이프라인입니다.

FRUITION

01	RAG 근거 선택 - 충족률 66.25% → 91.25% 개선	p.6
02	에이전트 실행 환경 - 지침 작성·검증·승인 설계	p.11
03	작업 취소 - 저장한 변경의 역순 복구	p.16
04	PDF 변환 - 통과율 45.17% → 89.89% 개선	p.20
05	문서 병렬 분석 - 처리시간 73.8% 단축	p.24
06	에이전트 루프 - 60시간만에 MSA로 서비스 전환하기	p.26

PILLTIP

07	의약품 정제 - 예상 \$200 대비 \$11.18로 변환	p.33
08	복약 챗봇 - 건강 프로필의 외부 전송 제외	p.38

MORE

09	코드 밖에서도 이어지는 경험	p.42
	Rhwp · HWPX 저장 오류 수정 APPTIVE · 백엔드 멘토링 백준 945일 연속 문제 해결	
10	프로젝트 회고	p.43

Fruition

올린 문서를 LLM이 Wiki로 정리하고, RAG로 원문 근거를 찾아 답하거나 자연어로 문서를 편집하는 AI 워크스페이스입니다. 지금은 AWS EKS에 배포해 QA를 진행하고 있습니다.

시작 계기

팀원이 겪던 문서 관리의 불편함을 디자인 싱킹으로 구체화하며 시작했습니다. 문서를 저장만 해 두고 정작 필요할 때 찾지 못하는 불편을 없애는 게 목표였습니다.

GitHub · AI 분리 저장소 ↗

팀 구성과 내 역할

MY ROLE

AI

PDF 변환, Wiki 생성, RAG 검색·근거 답변, 문서 편집 Agent 등 AI 기능 전체

01 AI 기능 전체 리드

PDF문서 변환, Wiki 생성·정리, 검색·근거 답변, 문서 편집 Agent 등 AI 기능 전반의 설계·평가를 리드하고, 구현은 AI 코딩 도구로 진행했습니다.

02 MSA 설계 기여

초안은 기능별로 서비스를 나눴습니다. 그런데 그렇게 나누면 DB에 걸리는 부하는 그대로고, 서비스를 거치는 호출만 늘어 분산하는 의미가 없었습니다. 그래서 데이터 소유권을 기준으로 나누자고 제안했고, 팀원·멘토와 서비스 경계와 통신 방식을 논의해 최종 구조를 함께 완성했습니다.

백엔드 1명

Spring 기반 백엔드 전체

프론트엔드·DevOps 1명

프론트엔드 전체, MSA 기반 AWS 배포

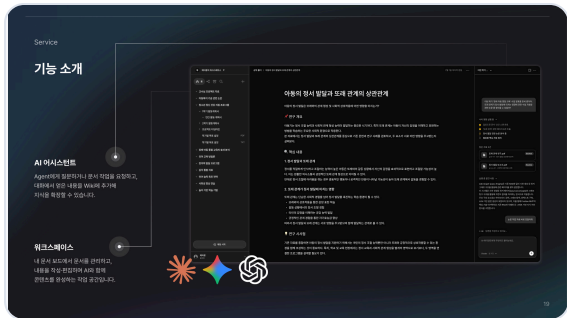
디자인

외주 디자이너

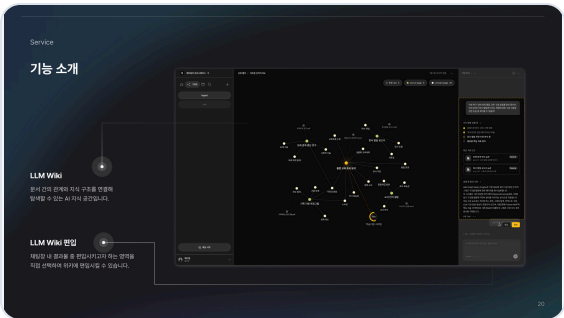
멘토 3명

Gen AI 품질·평가 / 백엔드 전반 / 마일스톤·MSA 설계·문서화

담당 기능과 서비스 화면



문서 편집·AI 어시스턴트
요청을 해석하고 수정안을 생성·검증하는 AI 기능을 리드했습니다. 편집기와 어시스턴트 UI는 팀 구현입니다.



Wiki 생성·검색
문서를 지식으로 정리하고 연결해 질문에 맞는 근거를 찾는 AI 기능을 리드했습니다. 그래프와 검색 UI는 팀 구현입니다.

Architecture

시스템 아키텍처

브라우저는 Spring Boot API를 통해 데이터와 AI Service을 사용합니다.

기술	구현
Next.js	문서 · Wiki · Chat · Agent UI
Spring Boot	인증 · 권한 · 워크스페이스 · 상태
FastAPI	Ingest · Query · Lint · 문서 편집 Agent
PostgreSQL	관계 · 버전 · 작업 상태
MinIO	원본 · Markdown · artifact

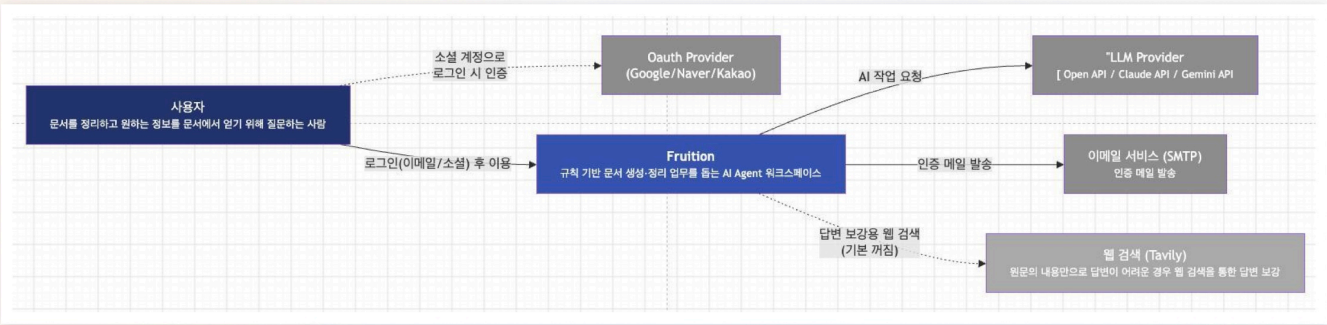
14

서비스 역할 구분

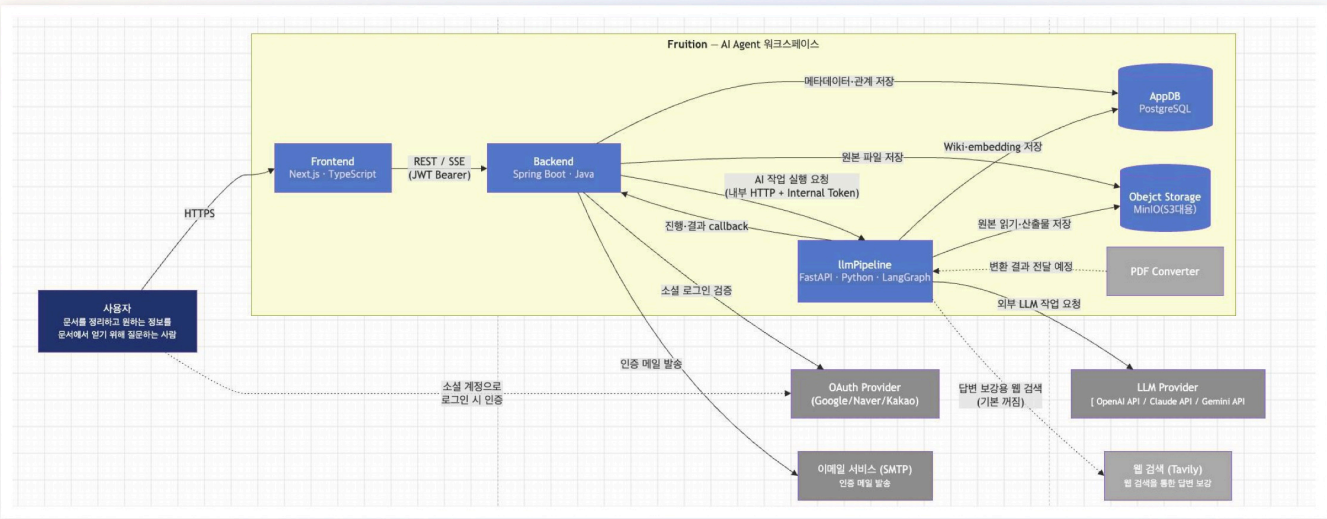
AI 기능 전체를 리드했고 프론트엔드·백엔드는 팀원이 담당했습니다. MSA는 제 초기 설계를 바탕으로 팀원·멘토와 논의해 완성했습니다.

시스템 아키텍처

| C1 전체 시스템 아키텍처 간략화



| C2 Fruition 시스템 일부 구체화



위는 전체 시스템, 아래는 Fruition 내부 구성입니다. 제 담당은 llmPipeline(FastAPI·Python·LangGraph)을 중심으로 한 AI 기능이며, 프론트엔드·백엔드는 팀원이 담당했습니다.

RAG 근거 선택 - 충족률 66.25% → 91.25% 개선과 Jev 최적화로 처리시간 87% 단축

Fruition은 사용자가 올린 문서를 근거로 답하는 서비스라, 근거가 틀리면 답변 전체를 믿을 수 없게 됩니다. 기존 검색이 놓치던 근거를 판단 전용 모델 Jev가 다시 고르도록 바꿔, 답변 근거 충족률을 66.25%에서 91.25%로 높였습니다. 늘어난 처리 시간은 후보 수와 병렬 처리를 조정해 줄였습니다.

2026.09

Jev 1.13.0

BGE-M3

BM25

RAG

품질 평가

배경

기존 검색은 근거를 놓치거나 답이 없는 질문에도 근거를 붙여, 잘못된 답변으로 이어질 수 있었습니다. 정확한 정보를 찾아 전달하는 것이 이 서비스의 핵심이라 단계별 실행 결과를 분석했고, 정답 근거는 처음 후보에 모두 있었지만 검사 대상을 앞 8개 페이지로 제한하는 단계에서 90문항 중 76문항의 근거가 빠지고 있음을 확인했습니다. 후보에서 필요한 근거를 고르고, 없으면 없다고 판단하는 단계가 필요했습니다.

해결하면 얻는 것

근거가 틀리면 답변 전체를 믿을 수 없고, 신뢰를 잃은 사용자는 다시 쓰지 않습니다. 프로젝트의 핵심이 사용자에게 필요한 정보를 정확한 근거를 골라 제공한다는 점이기 때문에, 높은 정확도가 곧 사용자가 우리 제품을 신뢰하며, 계속 쓰고 다시 결제할 이유가 됩니다.

근거 충족률(Golden set을 통한 LLM 평가)

66.25% → 91.25%

Jev 근거 선택 시간

**중앙값
9.090초 → 1.137초**

답 없는 질문 20건에 대한 근거 반환

20건 → 0건

실험과 개선 과정

1 요청 분류 성능 비교와 RAG 적용 결정

문제 상황

질문을 문서 편집으로 분류하는 등 요청과 다른 작업으로 이어지는 문제가 있었습니다. 서비스의 신뢰성과 직결되는 문제라고 판단해, 작업 종류와 부가 조건을 더 정확하고 빠르게 고를 방법이 필요했습니다.

시도와 결과

9월 14일 출시된 선택형 판단 모델 Jev를, 사용자 요청을 질문·문서 편집·문서 생성·폴더 정리 등으로 나누는 라우팅에 적용했습니다. Jev의 벤치마크와 공식 Discord에 공유된 활용 사례를 보고, 우리 서비스에서도 라우팅 정확도를 높이고 비용과 시간을 줄일 수 있다고 판단했기 때문입니다.

모델이 판단한 98개 요청에서 작업 종류와 부가 조건을 모두 맞힌 수는 기존 JSON 방식 77개, 같은 선택지를 받은 GPT-5-nano 63개, Jev 81개였습니다. 판단시간 중앙값은 8.967초에서 0.658초로 줄었고, 같은 100문항의 API 비용은 GPT 방식 약 \$0.1027, Jev 약 \$0.0408이었습니다.

판단

정확도·속도·비용 모두에서 가능성을 확인해, 기존 검색이 풀지 못한 답변 근거 선택에도 적용하기로 했습니다.

Jev를 고른 이유는 두 가지입니다. 첫번째는 GPT에게 근거를 고르게 하면 같은 선택지를 줘도 비용이 약 2.5배였고 판단 시간도 길었습니다. 질문마다 수백 개 후보를 검사하는 단계에는 부담이 컸습니다. 검색 점수를 조정해 순위를 다시 매기는 쪽은, 하이브리드 가중치와 RRF를 여러 값으로 실험해도 점수가 비슷한 수준에 머물렀습니다. 두번째로는 임베딩이 한국어 표현의 차이를 충분히 구분하지 못하는 경우가 있어, 분야마다 다른 질문에 하나의 가중치를 맞추는 것으로는 개선 폭이 작다고 봤습니다. 그래서 순위를 더 다듬기보다, 모아 온 후보에서 필요한 근거를 직접 판단하게 했습니다.

실험과 개선 과정 (계속)

2 전체 근거 검사로 평가 질문 90건의 근거 충족

문제 상황

질문 원문으로 검색하고 의미 점수 비중을 높여, 질문 110개에서 정답 자료가 첫 번째에 오는 경우를 50개에서 57개로 늘렸습니다. 다만 편향이 있는 실험이었고, 새로운 질문에서도 필요한 근거를 안정적으로 고르는 문제가 남아 있었습니다. 범용적인 질문에서 안정적인 답변을 내보내주는 것이 프로젝트의 핵심이어서 해당 부분의 품질을 올릴 필요가 있었습니다.

시도와 결과

Jev가 검색이 놓친 근거도 찾을 수 있는지 확인하려고, 테스트 위키의 근거 2,020개를 거르지 않고 모두 Jev로 검사했습니다. 근거 충족률을 26.7%에서 100%로 높였지만, 처리 시간 중앙값이 1.511초에서 5.338초로 늘었고 질문마다 모든 근거를 검사해야 했습니다.

차이가 너무 커서 기존 검색의 탈락 단계를 추적했고, 정답 페이지가 앞 8개 페이지 제한에서 90문항 중 76문항이나 빠지고 있었음을 확인했습니다.

판단

기존 검색은 고르기도 전에 정답을 버리고 있어, 이 차이를 같은 후보에서의 선택 능력 차이로만 볼 수는 없었습니다. 또 질문마다 모든 근거를 검사하면 문서가 늘수록 호출 비용도 함께 늘어, 사용자가 비용을 선결제하는 서비스에 그대로 쓸 수 없었습니다. 그래서 검색으로 후보를 먼저 줄이고 Jev가 근거를 고르는 구조로 바꾸기로 했습니다.

3 동일 500개 후보 비교에서 근거 충족률 87.50% 달성

문제 상황

전체 근거 검사는 비용이 커서 서비스에 그대로 쓸 수 없었고, 평가 질문도 특정 문서에 몰려 있었으며 질문에 문서 제목까지 들어 있었습니다. 기존 검색은 일부 페이지만, Jev는 전체 근거를 보고 판단했기 때문에, 정확도 차이가 판단력 때문인지, 볼 수 있는 범위 때문인지를 판정하기가 어려웠습니다. 후보를 줄이면서 비교 조건도 맞춰야 했습니다.

시도와 결과

논문 9편으로 질문 100개(답이 있는 질문 80개, 없는 질문 20개)를 새로 만들어 실행 전에 고정했습니다. 두 방식에 같은 후보 500개를 주고 반환 개수를 8개로 맞췄으며, 정답은 입력에서 빼고 GPT와 Gemini가 방식 이름을 모른 채 채점했습니다.

그 결과, 근거 충족률을 66.25%에서 87.50%로 높였지만, 후보를 순차 처리해 처리 시간 중앙값은 기존 0.431초에서 9.090초로 늘었습니다.

판단

같은 조건에서도 Jev가 근거를 더 잘 골라, 검색은 후보 수집, Jev는 최종 근거 판단을 맡는 구조를 유지했습니다. 다만 공개 논문과 직접 만든 질문의 결과라 일반적인 정확도로 단정하지 않았고, 순차 처리로 늘어난 대기 시간을 줄이는 것을 다음 과제로 정했습니다.

실험과 개선 과정 (계속)

4 300개 후보와 병렬 수 비교로 대기 시간 단축

문제 상황

근거 선택이 끝나야 답변이 시작되므로 처리 시간은 곧 사용자의 대기 시간입니다. 500개 순차 처리 중앙값 9.090초는 기존 0.431초보다 너무 길어 사용이 불가능하다 판단되었습니다. 하지만, 높은 정확도를 포기하긴 아까워 처리시간을 줄이고자 했습니다.

시도와 결과

후보 갯수가 비용과 처리속도에 영향을 주므로, 후보를 상위 500개에서 300개로 줄이고 최대 4개 묶음을 병렬 처리했습니다. 시간을 2.251초로 줄였고, 필요한 근거를 모두 고른 질문도 70개에서 73개로 늘렸습니다.

병렬 수는 근거 없이 정한 값이라, 같은 100문항에서 병렬 수만 4·6·8·10으로 바꿔 다시 측정했습니다. 선택 시간 중앙값은 2.066·1.165·0.945·0.886초였고, 선택이 달라진 42문항을 블라인드로 채점한 근거 충족은 37·38·36·37개로 차이가 없었습니다.

판단

8병렬 이후로는 시간 이득이 0.06초로 작았고, 운영에서는 동시 요청이 작업자 수 × 병렬 수로 늘어나 8병렬을 골랐습니다. 후보 추리기 포함 1.137초로 기존 0.431초보다는 여전히 느리고, 후보를 더 줄이는 안은 필요한 근거가 250위 밖에 있는 사례가 있어 접었습니다.

처리 시간이 늘어나는 것은 감수하고, 정확한 근거 전달이 이 서비스의 핵심이라고 판단해 Jev를 채택했습니다.

5 후보 생성 방식 비교로 운영 설정의 한계 확인

문제 상황

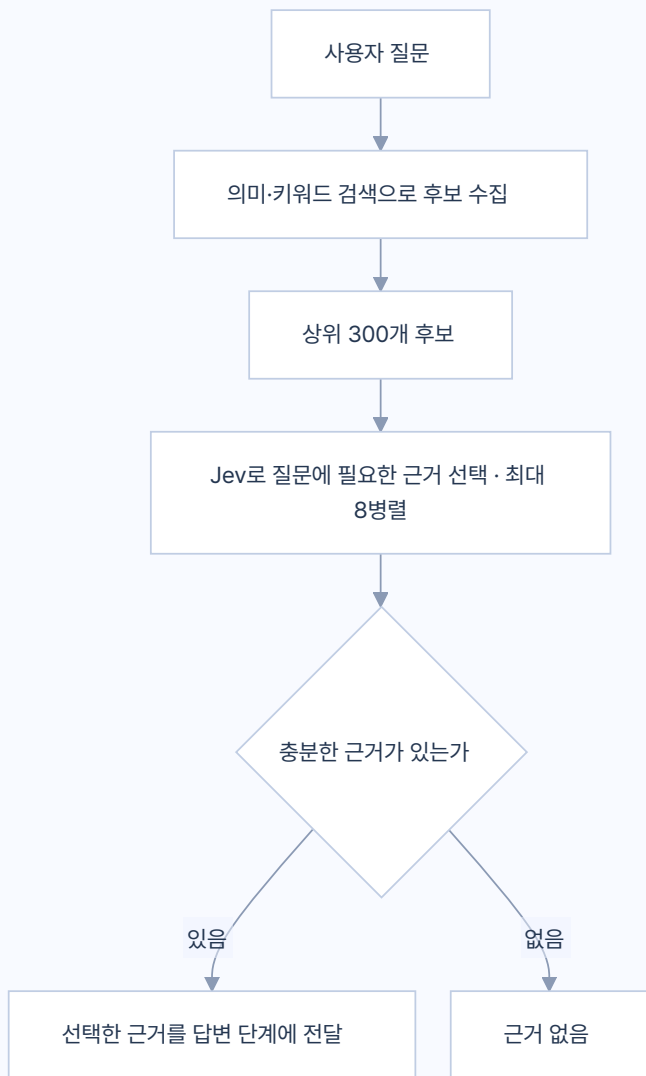
Jev가 잘 골라도 필요한 근거가 후보에 없으면 답변에 쓸 수 없습니다. 현재 후보 생성 방식이 정답 근거를 어디까지 남기는지 확인해야 했습니다.

시도와 결과

운영 중인 혼합 방식(임베딩 0.75 + 정규화 BM25 0.25)과 BGE-M3 단독, RRF, 교대 합집합을 저장된 점수로 다시 비교했습니다. 후보 100개 안에 정답 인용을 모두 남긴 문항은 BGE-M3 단독 74/80, 운영 혼합 63/80이었고, 후보를 300개 이상으로 늘리면 차이가 79/80과 78/80으로 좁아졌습니다.

판단

운영 중인 혼합 가중치가 BGE-M3 단독보다 정답 근거를 후순위로 매깁니다. 후보 300개인 현재 구성에서는 차이가 작아 운영 설정은 그대로 두고, 후보를 더 줄일 때 먼저 해결할 조건으로 남겼습니다.



근거 총족 건수 53/80 → 73/80, 총 처리시간 1.137초

근거 총족률(Golden set을 통한 LLM 평가)

66.25% → 91.25%

기존 선택기 53/80 → Jev 개선안 73/80

Jev 근거 선택 시간

중앙값 9.090초 → 1.137초

500후보 순차 → 300후보 8병렬

답 없는 질문 20건에 대한 근거 반환

20건 → 0건

기존 선택기 → Jev 개선안

자체 제작 평가셋에서 답이 있는 80개 질문의 근거 총족률을 66.25%에서 91.25%로 높였고, 답이 없는 질문 20개에 근거를 잘못 붙인 경우는 20건에서 0건으로 줄였습니다. 처리 시간은 기존 선택기보다 길지만, 정확한 근거 전달이 이 서비스의 핵심이라고 판단해 Jev를 채택했습니다. 같은 뜻의 표현을 묶는 판단에도 시험해, 정답 수는 기존 71개·Jev 72개로 유지하면서 판단시간 중앙값을 8.162초에서 0.254초로 줄일 수 있음을 확인했습니다.

결과의 범위 · 트레이드오프

검색은 의미·키워드 검색과 Wiki 그래프 탐색으로 원문 근거를 모으는 고정 파이프라인으로 두고, 답변과 근거·관련 페이지·탐색 경로를 나눠 보여 주도록 설계했습니다. 검색 전략을 스스로 세우는 Query Agent는 비용과 할루시네이션 위험 때문에 보류하고, 한 질문이 여러 번의 검색 계획과 재계획을 요구할 때 도입하기로 기준을 정했습니다. 그 전까지는 고정 파이프라인이 더 적은 호출로 같은 근거를 모으고 실패 지점도 추적하기 쉽습니다. 시간 단축은 Jev 설정을 바꾸기 전후의 비교이며, 개선 후에도 기존 선택기보다 느립니다. 근거 선택은 답변 전에 끝나야 해서 1.137초는 사용자 대기로 남고, 이 구간을 줄이는 일은 과제로 남겼습니다. 검색은 후보 수집, Jev는 최종 근거 판단을 맡도록 단계를 나눠 선택 방식만 바꿔 끼울 수 있게 했습니다.

Jev 공식 소개 · TypeSafe AI ↗

CASE 02

Fruition / AI Agent · 사용자 정의 Skill · HITL · Prompt Injection

사용자 정의 Agent Skill과 안전한 실행 설계

사용자마다 반복하는 문서 작업이 다르고, AI가 바꾼 문서는 매번 믿을 수 있어야 했습니다. 반복 작업을 Agent Skill로 직접 정의해 다시 쓰게 하고, 검사·게시·변경 승인을 한 흐름으로 설계해 AI 코딩 도구로 구현했습니다.

2026.08 - 2026.09

LangGraph

Agent Skill

Prompt Injection 대응

LLM Evaluation

Human-in-the-loop

컨텍스트 엔지니어링

대화 요약 메모리

LangSmith

LLM Observability

배경

회의록 정리처럼 사용자마다 반복하는 작업이 달라, 개발자가 전부 미리 만들 수는 없었습니다. 또 AI가 만든 편집안이 요청 일부를 빠뜨리거나 문장을 어색하게 바꾼 채 사용자에게 넘어갔고, 코드의 형식·범위 검사만으로는 걸러내지 못했습니다. 항상 같은 결과가 필요한 작업을 운에 맡기게 되면 사용자는 불편을 겪고, API 비용을 선결제하는 결제 구조에서는 재결제를 망설이게 만드는 이유가 될 수 있었습니다.

해결하면 얻는 것

AI 편집 결과가 매번 달라 다시 요청하면 선결제한 비용이 재작업에 쓰이고, 이를 낭비라고 생각이 든 사용자들은 재결제를 망설이게 됩니다. 결과가 안정적이고 승인 전에는 문서가 바뀌지 않으면 반복 작업을 믿고 맡겨 계속 쓰게 됩니다.

기존 Agent

94 / 114

코드 수정 · 평가자 제외

98 / 114

평가자 포함 최종 구현

104 / 114

판단과 구현 과정

1 사용자 정의 작업 지침의 작성·게시 기능 구현

문제 상황

회의록 정리처럼 사용자마다 반복하는 작업이 달라, 개발자가 모두 미리 만들어 둘 수는 없었습니다. 하지만 사용자가 반복된 작업을 매번 같은 요청 프로세스를 반복하도록 만드는 것은 사용자로서 불편하다고 생각했습니다.

판단

기능을 하나씩 늘리는 대신, 사용자가 자연어로 작업 지침을 정의하고 다시 쓰게 하기로 했습니다. AI가 만든 지침은 사용자가 검토하기 전까지 제안으로만 다룹니다.

구현

자연어 요청을 다시 쓸 수 있는 Markdown Skill로 정리했습니다. 사용자가 초안을 검토·수정해 게시할 때 Skill과 버전을 저장합니다. 게시한 Skill은 덮어쓰지 않고 불변 버전으로 덧붙이고, 실행할 버전을 가리키는 포인터를 따로 뒀습니다. 어떤 실행이 어느 버전을 썼는지 추적할 수 있고, 새 버전에 문제가 생겨도 이전 정의가 남습니다.

확인한 결과

작성 단계에서는 아무것도 저장하지 않는지, 게시하면 새 버전이 생기고 실행 버전이 그쪽을 가리키는지, 다른 워크스페이스나 /명령으로도 게시한 Skill을 불러오는지 단위 테스트로 확인했습니다. 기능을 마쳤을 때 Skill 모듈 74개와 파이프라인 전체 546개 테스트를 통과했습니다.

2 작업 지침의 권한·승인 우회 방지 검증

문제 상황

지침과 참고 문서에 개인정보나 승인을 건너뛰라는 지시가 섞이면 문서가 승인 없이 바뀌거나 정보가 썰 수 있었습니다. 표현도 제각각이고, 초안은 게시 전에 다시 고쳐질 수 있었습니다.

판단

명확한 위험과 권한 검사는 코드가 말고, 표현을 바꾼 우회 의도는 LLM으로 찾았습니다. 검토 뒤에도 내용이 바뀔 수 있으므로 이전 검사 결과만으로 게시를 허용하지 않았습니다.

구현

입력과 참고 문서에 규칙 검사를 적용하고, LLM이 지정한 위험 문구가 원문에 정말 있는지 서버에서 대조했습니다. 참고 문서는 권한과 원문을 검사한 뒤 필요한 구조만 모델에 넘겼습니다. 최종 게시 때는 고친 내용으로 다시 검사합니다. 위험 문구는 자동으로 지우지 않고, 사용자가 재생성을 고른 경우에만 뺀 뒤 결과를 다시 검사합니다.

확인한 결과

승인 우회·권한 상승 등 6가지 유형의 위험 문구 25개와 개인정보·인증 정보를 규칙으로 검사합니다. '승인 없이 관련 문서를 이동한다' 같은 지시가 모델 호출 전에 막히는지, LLM이 짚은 문구의 위치가 원문과 같은지, 최종 게시 때 넣은 이메일이 저장 없이 막히는지를 테스트로 확인했습니다. 막히면 원문과 문제 위치를 보여 줘, 사용자가 직접 고치거나 재생성을 고를 수 있습니다.

판단과 구현 과정 (계속)

3 변경 미리보기와 사용자 승인 후 반영

문제 상황

Skill을 게시했더라도, 할루시네이션으로 인해 사용자가 원치 않은 방식으로 생성, 편집, 정리되는 경우 이를 다시 복구하기 위해 롤백을 하러가야하는 경우가 생길때마다 사용자가 불편함을 겪을 수 있습니다. 따라서 실행할 때 바뀔 문서와 내용은 따로 확인받아야 했습니다.

판단

지침 게시와 실제 문서 변경은 승인을 따로 받기로 했습니다. 작업 분류는 라우터가, 실행은 실행기가 맡되, 무엇이 바뀌는지는 사용자가 확인합니다.

구현

편집은 수정 전후 차이를, 폴더 정리는 만들 폴더와 옮길 문서를 계획으로 보여 줬습니다. 출력 형식과 수정 범위는 코드로 검사하고, 승인한 변경만 실행합니다.

확인한 결과

문서 6개로 계획·승인·실행을 해 보니, 승인 전에는 아무것도 바뀌지 않았고 승인 뒤에 주제별 폴더 3개가 생기고 문서가 옮겨졌습니다.

4 대화 맥락과 편집 입력의 범위 결정

문제 상황

대화가 길거나 편집할 문서가 크면 모델 입력이 늘어, 사용자가 부담하는 비용과 대기 시간이 커집니다. 그래서 무엇을 남기고 무엇을 자를지 정해야 했습니다.

판단

대화는 최근 메시지 6개(질문과 답 약 세 번)만 원문으로 두고, 그 이전은 누적 요약으로 압축했습니다. 편집은 문서 전체 대신 대상 줄 범위와 주변 문맥만 보내고, 표나 코드 블록 경계에 걸치는 범위는 거절했습니다.

구현

최근 메시지가 6개가 되면 기존 요약과 최근 메시지를 함께 넣어 누적 요약을 갱신합니다. 요약 호출이 실패하면 요약을 바꾸지 않아 대화가 끊기지 않습니다. 라우터에는 요약과 최근 메시지에 이전 턴의 라우팅 결정까지 넘겨, '그거'처럼 가리키는 말의 대상을 이전 턴에서 찾게 했습니다. 편집 입력은 대상 줄 범위에 위아래 문맥 줄을 붙여 자르고, 구조 경계를 걸치는 범위는 예외로 막았습니다.

확인한 결과

1,000줄 문서에서 모델에 보낸 입력은 원문의 5.58%, 5,000줄에서는 1.12%였고, 입력을 만드는 시간의 p95는 각각 7.7ms, 34.4ms였습니다. 표·코드 블록·수식에 일부만 걸치는 범위가 거절되는지도 테스트했고, 실제 HTTP 요청에서 코드 블록 일부를 고른 요청은 422로 막혔습니다.

판단과 구현 과정 (계속)

5 LLM 실행 추적과 민감정보 경계 설정

문제 상황

평가 루프가 평범한 Python 흐름이라 의미 추출·평가·보정·재시도를 단계별로 볼 수 없었습니다. 그렇다고 추적을 켜면 도구 조회 원문과 개인정보가 외부 추적 도구로 나가 보안에 취약해지는 문제가 있었습니다.

판단

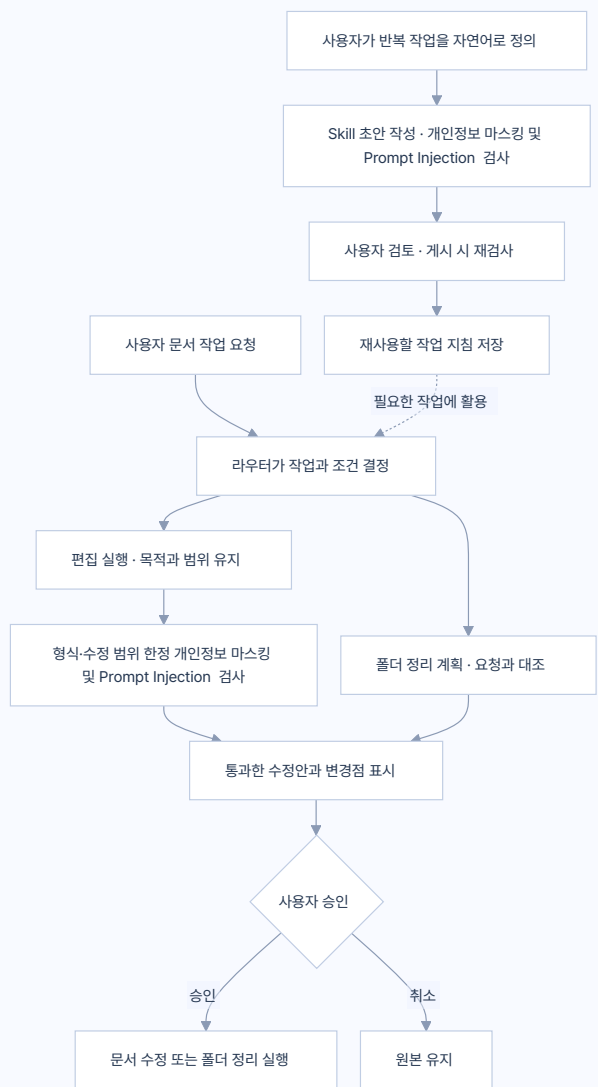
생성·평가·재시도를 단계별로 보려고 품질 루프는 LangGraph 노드 단위로 추적하고, 사용자 문서를 다루는 Agent 그래프는 추적을 끄고 원문을 상태에 남기지 않았습니다.

구현

평가 루프를 LangGraph 그래프로 다시 짜서 응답 생성·평가·재시도를 각각의 노드로 드러냈습니다. 추적 설정이 켜져 있어도 키가 없으면 실행은 그대로 하고 추적만 건너뛰어, 경고와 불필요한 네트워크 재시도를 없앴습니다. Agent 그래프는 추적을 끄고 도구 조회 원문을 상태 저장에서 뺐고, 보관 기간이 끝나면 저장 상태를 실행 기록보다 먼저 지웁니다. 직접 남기는 프롬프트 로그는 기록 전에 숫자 개인정보를 마스킹합니다.

확인한 결과

요청을 실제로 실행해 그래프 노드와 모델 호출 구간이 추적에 잡히는지 확인했습니다. 시간을 재는 실험은 추적과 프롬프트 로그를 모두 끄고 돌려, 계측 부담이 측정값에 섞이지 않게 했습니다.



사용자 정의 작업과 승인 흐름 구현 · 편집 품질 82.5% → 91.2% (114건 평가)

기존 Agent

94 / 114

82.5%

코드 수정 · 평가자 제외

98 / 114

86.0%

평가자 포함 최종 구현

104 / 114

91.2%

사용자가 반복 작업을 직접 정의해 Skill로 게시하고 다시 쓸 수 있게 했습니다. 게시할 때는 최종 내용을 다시 검사하고, 실제 문서 변경은 미리보기와 사용자 승인을 거칩니다. 문서 6개로 폴더 정리를 계획, 승인, 실행해 정상 작동함을 확인했습니다. 편집 품질은 19종 요청에서 나온 114개의 초안으로 비교했습니다. 편집 목적을 편집 단계까지 전달하고 재시도 입력을 고쳐 통과를 94개에서 98개로, 요청 충족과 의미 보존을 따로 평가하고 한 번만 다시 쓰게 해 104개까지 높였습니다. 평가에서 정상인 경우까지 거부하는 문제는 생성과 평가 프롬프트를 분리해 해결했고, 다시 실패하면 수정안을 반환하지 않도록 구현했습니다. AI는 작업 지침과 수정안을 쓰고, 코드는 권한과 실행 범위를 검사하고, 사용자는 게시와 최종 반응을 결정하도록 책임을 나눴습니다.

결과의 범위 · 트레이드오프

편집 품질 평가의 환산 평균 시간은 11.11초에서 18.92초로 늘었습니다. 지연을 감수하되 재작성 횟수는 제한했습니다.

구현 회고 - 사용자가 직접 만드는 Agent Skill [↗](#) 후속 회고 - 신뢰도를 위한 LLM 평가의 양면성 [↗](#)

CASE 03

Fruition / AI Agent · 보상 복구 · 감사 이력

AI 작업 취소와 이미 저장한 변경의 복구

AI가 문서를 바꾸는 서비스에서 취소가 확실하지 않으면 사용자는 작업을 맡길 수 없습니다. 실행만 멈추면 이미 저장한 문서·폴더·Wiki·Skill 변경이 남아, 커밋한 변경까지 역순으로 되돌리는 경로를 만들었습니다. 다른 작업의 정상 기여는 지우지 않도록 대상 기여만 빼고 복원했습니다.

2026.08 — 2026.09

보상 복구

먹등기

append-only 이력

PostgreSQL 실행 잠금

감사 이력

배경

여러 저장소에 걸친 AI 작업은 긴 트랜잭션 하나로 묶을 수 없었습니다. 그 상태에서 사용자가 취소하면 무엇을 어디까지 되돌릴지 정해야 했습니다.

해결하면 얻는 것

취소해도 AI가 남긴 변경이 남으면 사용자 데이터가 망가져 사용자 이탈과 문의로 이어집니다. 확실히 되돌릴 수 있어야 큰 작업까지 AI에 안심하고 맡겨, 크레딧 사용량을 늘릴 수 있습니다.

복구 대상 저장소

3곳

복구 실행

**모델 호출
없음**

취소 확정 조건

**전체 복구
확인**

판단과 구현 과정

1 실행 중단만으로는 부족하다는 판단

문제 상황

AI 작업은 모델 호출 사이사이에 문서·폴더·Wiki·Skill을 여러 번 저장합니다. 그래서 실행만 멈추면 취소한 작업의 일부가 그대로 남았습니다. 하지만 core DB, AI DB, 객체 저장소에 걸친 실행을 긴 트랜잭션 하나로 묶을 수도 없었습니다.

판단

이미 커밋한 변경까지 되돌리기로 했습니다. 모델에게 반대 작업을 시키면 원래 값과 승인 범위, 멱등성을 보장할 수 없었습니다. 또한 서비스 전체를 트랜잭션 하나로 잡으면 모델·HTTP·객체 저장소의 긴 실행과 장애를 버틸 수 없어 두 방법 모두 기각했습니다.

구현

AI 쪽은 실행 기록으로 역작업을 정하고, 업무 데이터의 원자적 복구는 backend가 맡도록 나눴습니다. Agent 도구는 변경 전 snapshot과 원래 승인에 묶인 역작업 인자·멱등키를 저장하고, 일반 작업은 각 DB의 trigger로 변경 전후 값을 같은 트랜잭션에 기록했습니다. 객체 저장소는 쓰기 전에 원본과 예정 결과를 남겼습니다.

확인한 결과

실제 worker로 돌리는 테스트에서 복구 중 모델을 만들면 실패하게 해 두고, 모델과 게이트웨이를 한 번도 부르지 않고 취소가 끝나는 걸 확인했습니다. 폴더 2개·문서 6개를 정리하는 작업을 9개 지점 각각에서 취소해도 원래 상태로 돌아왔고, 응답이 유실된 뒤 새 worker가 같은 멱등키로 복구를 이어 가는 것도 확인했습니다.

2 취소 이후의 쓰기를 막고 자식부터 역순으로 복구

문제 상황

취소를 받아도 실행 중인 handler가 계속 쓰면, 복구한 값이 다시 덮였습니다. 부모 작업부터 되돌리면 자식이 남긴 변경이 그대로 남았습니다. 복구가 제대로 되지 않아 사용자가 원하는 버전으로 복구하지 못하는 상황이 발생할 수 있었습니다.

판단

취소 상태를 먼저 저장해 이후 쓰기를 막고, 실행 중인 handler가 끝나면 자식부터 역순으로 되돌렸습니다.

구현

Agent 턴과 복구가 같은 PostgreSQL 실행 잠금을 쓰게 했습니다. worker 연결이 끊기면 잠금도 풀리니, 남은 실행 중 기록이 복구를 막지 않습니다. 본문은 새 revision으로 복구해, 늦게 도착한 이벤트가 원문을 다시 덮지 못하게 했습니다.

확인한 결과

취소 뒤 쓰기 거부, 실행 중인 handler를 기다린 뒤의 복구, 자식 먼저 복구를 테스트로 확인했습니다. 프로세스를 강제로 종료해도 복구는 같은 잠금을 기다렸다가 끝났고, 늦게 온 이전 revision 이벤트는 복구한 본문을 덮지 못했습니다. 다른 사용자가 바꾼 내용이 있으면 덮지 않고 재시도 상태로 남겼다가, 충돌이 풀린 뒤 다시 취소하면 정상 완료됐습니다.

판단과 구현 과정 (계속)

3 다른 작업의 기여를 지우지 않는 복구

문제 상황

Wiki의 개념 페이지는 여러 문서의 기여를 합친 결과입니다. 특정 작업을 과거 snapshot으로 뺏으면, 이후 들어온 다른 문서의 정상 기여까지 사라졌습니다.

판단

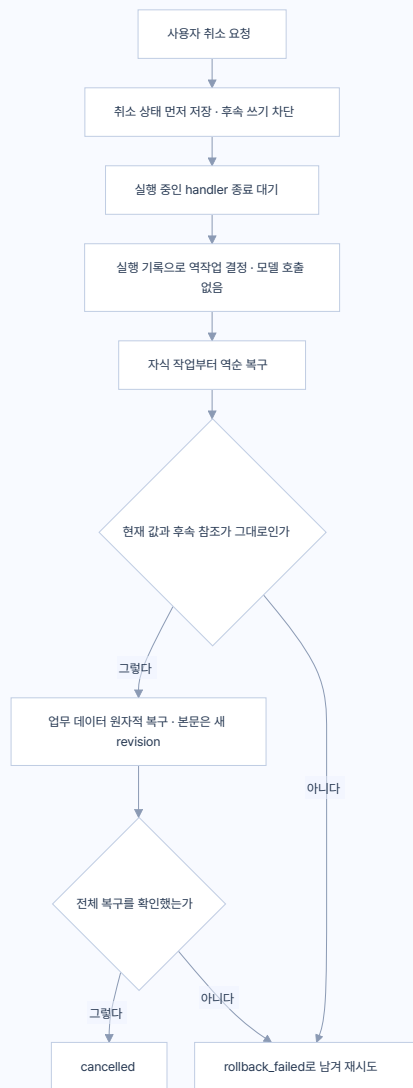
완성된 결과에서 해당 부분만 지우는 방법은 어느 문장이 어느 문서에서 왔는지 안정적으로 판단하기 어려웠습니다. 과거 snapshot으로 통째로 되돌리는 방안도 중간에 새 문서가 기여를 진행한 경우, 정보가 유실될 수 있어 기각했습니다. 대신 이력은 덧붙이기만 하고, 복구할 때 대상 기여만 빼고 다시 조립하기로 결정했습니다.

구현

작업, 페이지 revision, 문서별 기여를 덧붙이기만 하는 이력으로 남기고, 복구 때는 대상 기여를 뺀 상태로 복원하거나 다시 조립했습니다. 페이지와 이력은 지우지 않고 삭제 표시만 남겨 감사 이력을 보존했습니다.

확인한 결과

복구 직전에 잠금을 잡고 기여 목록(작업·revision·활성 여부)을 다시 읽어, 미리보기 때와 다르면 복구를 거부했습니다. 고른 기여만으로 페이지를 다시 만들고, 한 페이지라도 실패하면 현재 상태를 하나도 바꾸지 않는 것을 테스트로 확인했습니다. 복구도 새 작업과 revision으로 남겨 무엇을 되돌렸는지 기록했습니다.



취소한 작업이 남긴 변경을 되돌리고, 다른 작업의 기여는 유지

복구 대상 저장소

3곳

core DB · AI DB · 객체 저장소

복구 실행

모델 호출 없음

실행 기록의 역작업만 사용

취소 확정 조건

전체 복구 확인

미확인은 재시도 상태로 보존

사용자에게 보이는 내용, 이름, 부모, 순서는 되돌리고 버전·감사·역등 기록은 남겼습니다. 복구는 모델 설정이나 호출 없이 실행 기록만으로 하고, 현재 값과 후속 참조를 확인해 다른 작업의 변경은 덮지 않습니다. 전체 복구가 확인돼야 취소가 확정되고, 충돌이나 불분명한 결과는 재시도 상태로 남깁니다. AI가 문서를 바꾸는 기능이라면 되돌릴 범위와 남길 기록부터 정해야 한다는 걸 구현하며 배웠습니다. 실제 Gemini·Kafka·PostgreSQL·Redis 환경에서 직접 돌려 보니 승인 전에는 아무것도 바뀌지 않았고, 마지막 문서 이동 중에 취소하자 역작업 8개로 폴더 2개와 문서 6개가 원래대로 돌아왔습니다.

결과의 범위 · 트레이드오프

설계와 단위 검증 중심의 사례이고, 운영 트래픽에서의 복구 성공률은 재지 않았습니디. Ingest 때부터 Source별 기여를 보존해야 해서 저장 비용이 늘어납니다. 남은 기여 조합의 snapshot이 없으면 Page를 다시 조립해야 하고, snapshot이 없는 구버전 작업 때문에 배포 전에 실행과 큐를 비워야 했습니다.

CASE 04

Fruition / PDF 문서 변환 · 데이터 파이프라인

PDF → Markdown 변환 - 내부 평가 통과율 45.17% → 89.89% 개선

Fruition에 올라오는 논문은 대부분 PDF이고, 표의 수치와 수식이 깨지면 잘못된 근거로 답하게 됩니다. AI로 고치는 과정의 누락을 추적하고, 원본 배치와 내용 유형에 맞춰 처리 경로를 나눴습니다.

2026.07 — 2026.09

PDF 변환 오픈 소스 : AnyDoc, Docling

LLM API

sLLM (Qwen2.5 7B)

Python

LLM-as-a-Judge

배경

논문 PDF를 답변 근거로 쓰려면 표·수식·그림까지 살린 Markdown이 필요했습니다.

해결하면 얻는 것

PDF 복원률이 떨어져 핵심 수치가 누락되면 문서가 Wiki에 온전히 편입되지 못해 낮은 품질의 서비스를 받게 되고, 이는 사용자 이탈로 이어질 수 있습니다. 복원 품질이 곧 Wiki와 답변 품질의 바탕입니다.

기존 Docling

45.17%

개선 파이프라인

89.89%

실험과 개선 과정

1 손상 영역 선택 복원과 탐지 누락 확인

문제 상황

오픈소스를 통한 변환으로는 표의 행·열 관계와 수식이 깨져, PDF 논문의 핵심 수치를 Wiki에 편집하기 어렵게 추출했습니다. 손상된 부분들을 복원할 방법이 필요했습니다.

시도와 결과

Docling으로 구조를 뽑고 손상된 표·수식만 로컬 Qwen sLLM으로 복원했습니다. 표는 비전 모델, 수식은 텍스트 모델로 나눠 두 모델을 모두 쓸 때보다 복원율을 높이고 sLLM 처리 시간을 약 42% 줄였습니다.

sLLM이 고친 96개 중 85개는 통과했지만, 손대지 않은 쪽에 오류·누락이 107개 남아 있었습니다. OCR 대조를 더하면 통과율을 80.00%에서 97.53%로 높일 수 있었지만, 멀쩡한 내용 50개까지 고칠 대상으로 골랐고 처리 시간의 합은 약 12분 35초였습니다.

판단

고칠 곳을 고르는 단계부터 오류를 놓쳤고, 이를 메우는 검사에도 비용이 들었습니다. 그래서 원본 페이지 전체를 비전 sLLM이 읽게 하는 방식을 다음으로 검토했습니다.

2 전체 페이지 sLLM 변환의 표·수식 오류와 비용 확인

문제 상황

고칠 곳을 ai가 판단하는 방식은 오류를 놓쳤고, OCR 대조를 더하면 정상적인 내용까지 고르면서 검사 시간과 비용이 늘었습니다. 손상된 부분만을 적절히 골라 처리하도록 하는 방법을 모색해야 했습니다.

시도와 결과

논문 PDF 25개(415쪽)를 비전 sLLM으로 통째로 처리했습니다. 내용 보존 통과율은 91.92%였지만 표·수식의 값과 구조가 모두 맞은 비율은 58.84%에 그쳤고, 모든 페이지를 처리하는 부담도 컸습니다.

판단

모든 내용을 같은 방식으로 모델에 맡긴 게 문제라고 판단했습니다. 본문은 필요한 곳에만 AI를 쓰고, 표·수식은 각자의 구조를 살릴 수 있는 입력을 줘야겠다고 생각했습니다. 그래서 본문·표·수식의 경로를 나누고, 본문은 빠르게 뽑으면서 필요한 곳만 AI로 고칠 방법을 모색했습니다.

실험과 개선 과정 (계속)

3 본문 추출 비교, AnyDoc 적용과 상용 API 전환

문제 상황

페이지 전체를 sLLM으로 돌려도 표·수식 오류가 남았고, 본문까지 전부 모델에 보내는 부담이 컸습니다. 이 부담을 줄여 처리 시간과 정확도를 향상시키고자, 주로 손상되는 표·수식은 따로 복원하고, 본문은 빠르게 뽑을 방법이 필요했습니다.

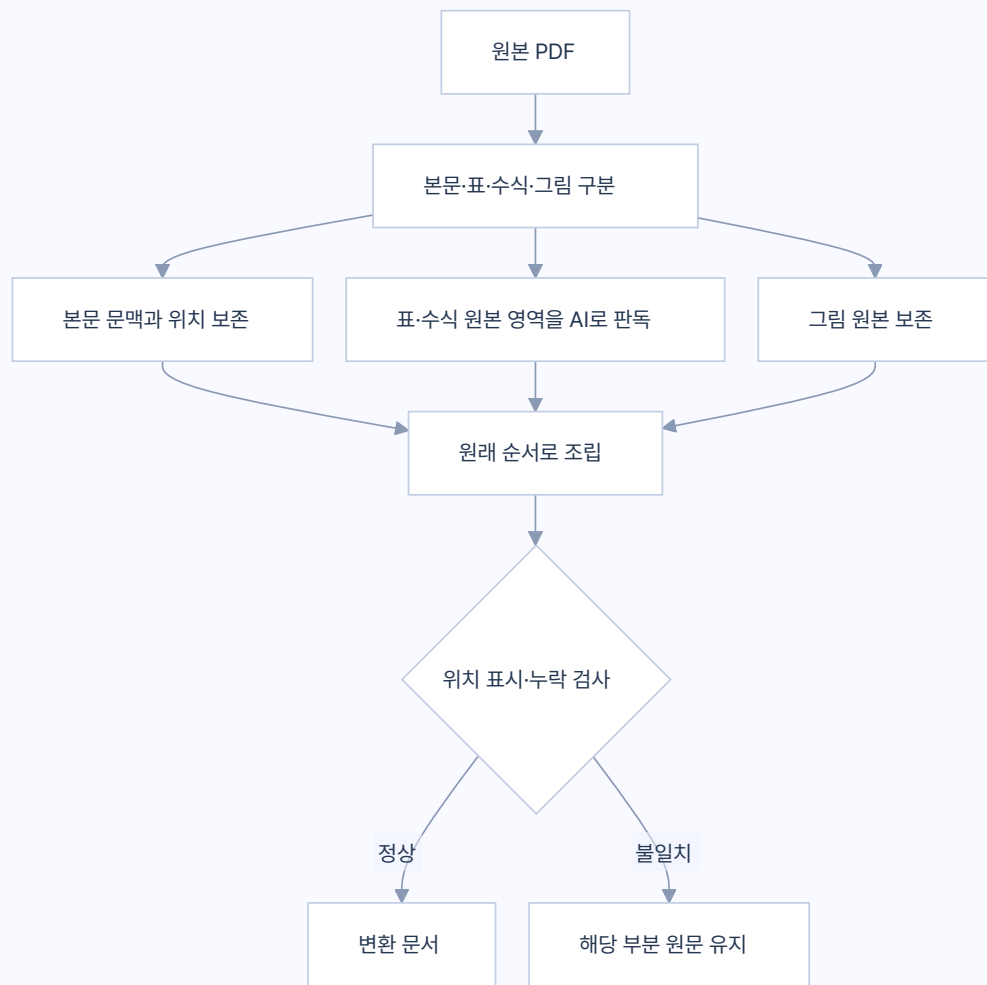
시도와 결과

Threads에서 알게 된 PDF 변환 도구 AnyDoc을, GitHub의 Docling 대비 평가 결과를 보고 기존 실험의 PDF 4개(30쪽)로 비교했습니다.

AnyDoc의 텍스트 추출은 1.68초로, OCR과 표 분석까지 포함한 Docling 기록 134.19초보다 약 80배 짧았습니다. 다만 한 문서는 PDF 안에 글자 코드를 실제 문자로 잇는 정보가 없어 글자가 깨졌습니다.

판단

처리 범위가 달라 전체 변환이 80배 빨라졌다고 볼 수는 없지만, 본문 추출이 빠르다는 점 때문에 AnyDoc을 본문에만 쓰기로 했습니다. 표·수식 복원은 유지하고, 글자가 깨지거나 읽는 순서가 섞인 페이지는 AI로 보완했으며, 그림은 원본을 살렸습니다. 또 초기 사용자가 없는 단계에서는 GPU 서버로 sLLM을 운영하는 비용이 더 커, 복원 모델을 상용 LLM API로 바꿨습니다.



30페이지 원문 대조에서 실사용 기준 통과율 45.17% → 89.89%

기존 Docling

45.17%

총 30페이지

개선 파이프라인

89.89%

총 30페이지

같은 논문의 본문·표·수식 등을 445개 평가 단위로 나눠 원문과 대조했고, 기존 Docling과 개선한 파이프라인은 이름을 숨기고 평가했습니다. 실사용 기준 통과율을 201개(45.17%)에서 400개(89.89%)로, 값·기호까지 똑같아야 하는 엄격 기준을 193개(43.37%)에서 393개(88.31%)로 높였습니다.

결과의 범위 · 트레이드오프

PDF 4개(30쪽)에 대한 내부 LLM 평가입니다. 본문, 제목, 그림은 경미한 차이를 허용했고, 표와 수식은 값, 기호, 구조가 일치해야 통과시켰습니다.

[PDF → Markdown에 SLLM 사용해보기 ↗](#) [PDF 변환 오픈소스 AnyDoc을 사용해보자 ↗](#)

CASE 05

Fruition / Kafka 비동기 병렬 처리 · AI 파이프라인

문서 병렬 분석 - 처리시간 282.11초 → 73.89초 단축

Wiki 생성이 늦으면 사용자는 올린 문서를 바로 쓸 수 없습니다. 문서를 하나씩 처리하던 구조를 병렬로 바꿔 분석 시간을 줄이고, 문서 분석과 벡터 생성의 확장 기준을 따로 세웠습니다.

2026.09

AI 파이프라인

Kafka

PostgreSQL

동시성·메모리

배경

Wiki 생성이 늦으면 사용자는 올린 문서를 바로 쓸 수 없는데, 뒤 문서가 앞 문서의 외부 AI 응답을 기다리고 있었습니다. 작업 분배에는 메시지를 다시 읽을 수 있고 같은 키 안에서 순서를 보장하는 Kafka를 써, 오래 걸리는 AI 작업을 비동기로 돌리면서 ID 기준으로 중복 처리를 막았습니다. Worker를 1개에서 4개로 늘려 분석 시간을 282.11초에서 73.89초로 줄였습니다. Worker 안의 동시 AI 요청을 2개에서 4개로 늘리자 95.92초로 오히려 느려졌고, 같은 GPU에서 벡터 생성을 2개 돌리자 각각 약 9.5초에서 19~21초로 늘었습니다. 두 단계는 병렬화 효과가 달라 확장 기준을 나눴습니다.

해결하면 얻는 것

Wiki 생성이 늦으면 올린 문서를 바로 쓰지 못해, 첫 사용에서 가치를 느끼기 전에 이탈할 수 있습니다. 대기를 줄이면 업로드 직후부터 검색과 질문에 문서를 쓸 수 있습니다.

Worker 1개

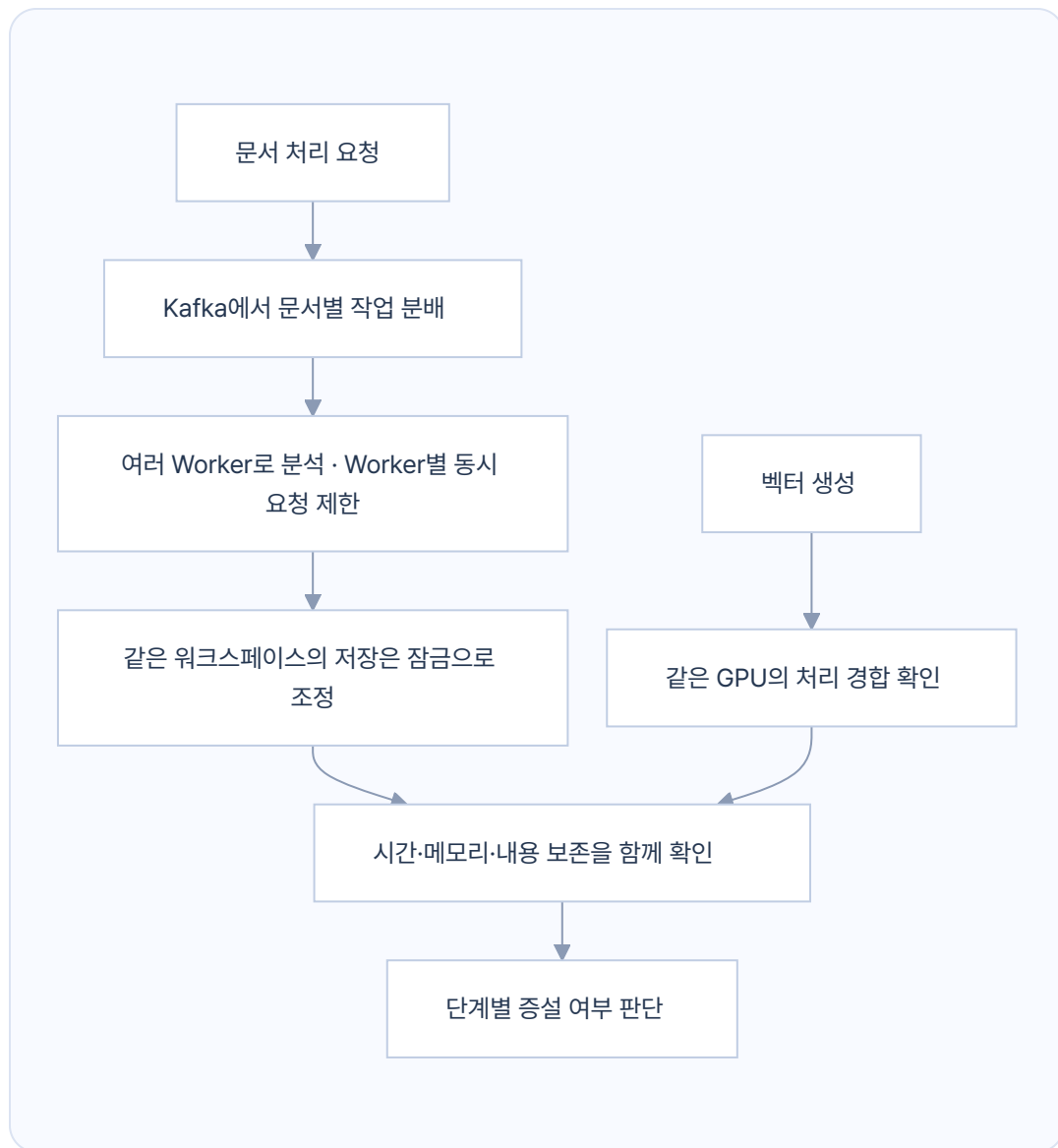
282.11초

Worker 4개

73.89초

프로세스 메모리 합

약 4배



문서 4개 분석 시간 73.8% 단축 · 문서 100개 큐 재현 300건 누락·실패 0건

Worker 1개

282.11초

4문서(8번의 호출)

Worker 4개

73.89초

최대 API 동시 요청 8개

프로세스 메모리 합

약 4배

351.7 → 1,401.0MiB

Worker 4개, 각 2개 동시 요청 조건에서 문서 4개의 총 처리 시간을 약 73.8% 줄였습니다. 대신 프로세스 메모리 합이 351.7MiB에서 1,401.0MiB로 약 4배 늘었습니다. 미리 정해 둔 사실 16개 중 완전히 보존된 수도 Worker 1·2·4개에서 10·14·11개로 달라, 조건별 1회의 작은 표본이지만 품질이 같다고 보지는 않았습니다. 동시 요청을 더 늘리는 제안과, 같은 GPU에서 벡터 생성을 병렬화하는 제안은 오히려 느려져 제외했습니다. 외부 응답 대기과 장치 경합은 서로 다른 병목이라, 확장 방식도 단계마다 달라야 했습니다. 저장된 AI 응답으로 문서 100개의 큐 처리를 3가지 조건에서 재현해, 총 300건 모두 누락·실패 없이 처리됨을 확인했습니다. 메모리 증가는 MSA로 EKS에서 AI 기능을 분산 배치해 감당할 수 있다고 보고, 문서 병렬 처리에는 최대 Worker 4개를 쓰기로 했습니다.

결과의 범위 · 트레이드오프

같은 문서로 조건별 1회 분석한 결과입니다. 저장, 후처리까지 포함한 E2E 속도는 아닙니다.

CASE 06

Fruition / 코딩 에이전트 운용 · MSA 전환

코딩 에이전트 루프 - MSA 이식 PR 16개를 60시간 만에 병합

MSA 전환을 일주일 안에 마쳐야 일정대로 배포하고 피드백을 받을 수 있었습니다. 이식은 코딩 에이전트에 맡기고, 저는 작업 계획과 검토 기준을 정하고 병합을 판단했습니다. 조정 에이전트가 순서와 우선순위를 관리하는 루프입니다.

2026.08

코딩 에이전트

Claude Code

MSA

코드 리뷰

배경

MSA 전환을 일주일 안에 마쳐야 일정대로 배포하고 새 구조에 대한 피드백도 받을 수 있었습니다. 기능이 이미 다 만들어져 있어 옮기며 고칠 곳이 많았고, 작업도 여러 갈래라 사람이 직접 하기엔 시간이 부족했습니다. 그래서 제가 쉬거나 자는 동안에도 작업이 이어지도록 코딩 에이전트 루프를 만들기로 했습니다. 다만 한 세션에서 오래 작업하면 맥락이 요약되면서 앞서 정한 계획을 놓치는 문제가 있었습니다.

해결하면 얻는 것

전환이 늦어지면 배포와 피드백 일정이 함께 밀리고, 계획을 벗어난 코드를 고치는 재작업 비용도 늘어납니다. 같은 인원으로 일정을 지키는 것이 목표였습니다.

병합 PR

16개

병합 전 필수 수정

13건

브랜치당 에이전트

2개

루프 설계와 판단 과정

1 역할을 나눈 에이전트 루프 구성

문제 상황

한 세션에 모든 작업을 맡기면 맥락이 요약되며 계획이 흐려졌습니다. 그래서 계획을 지키지 않은 코드나 행위가 발생해, 누락된 커밋이 생기거나, 계획과 다른 코드를 생성하는 경우가 있었습니다. 이를 수정하는데 또 비용이 들기 때문에, 이러한 낭비를 줄이고자 했습니다.

시도와 결과

작업 계획과 기준을 문서로 고정하고, 구현·검토·통합 리뷰로 역할을 나눴습니다. 브랜치마다 작업 공간을 따로 두고 에이전트 2개를 병렬로 돌렸고, 작업 간 의존성과 우선순위는 조정 에이전트가 판단하게 했습니다.

판단

세션이 바뀌어도 같은 문서를 기준으로 이어갈 수 있어, 긴 작업을 여러 에이전트에 나눠도 방향이 흩어지지 않았습니다.

2 PR 진행 기간: 8/9 → 8/13

문제 상황

작업들이 얹혀 있어, 한꺼번에 열면 충돌과 재작업이 생길 수 있었습니다.

시도와 결과

8/9에 #156-#159를 만들어 8/10에 병합하고, 8/11에 #167-#169-#170 등을 병렬로 진행한 뒤, 8/13 #175에서 통합 결함을 고쳤습니다. 5일 동안 PR 16개를 병합했습니다.

판단

선행 작업이 병합된 뒤에 다음 묶음을 열어, 병렬로 진행하면서도 충돌을 줄였습니다.

루프 설계와 판단 과정 (계속)

3 병합 전 리뷰와 병합 판단

문제 상황

에이전트가 만든 변경을 그대로 병합하면 결함까지 함께 배포될 수 있었습니다.

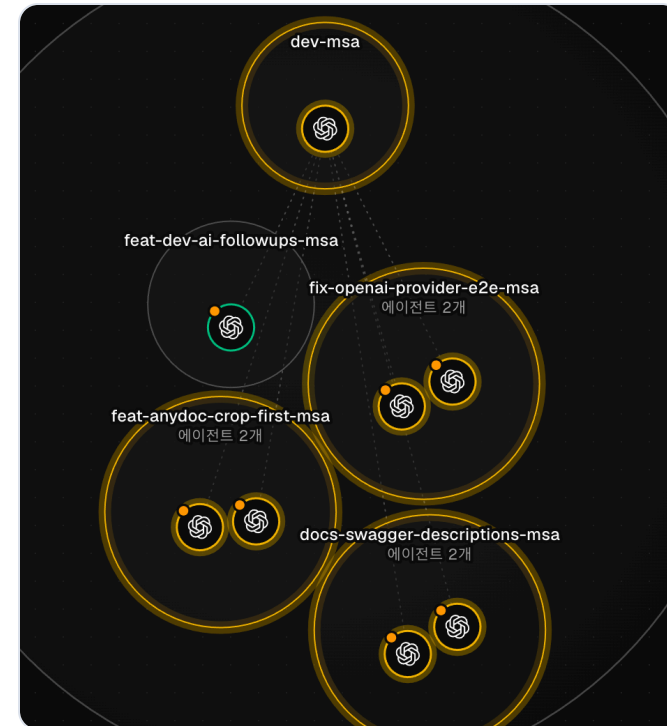
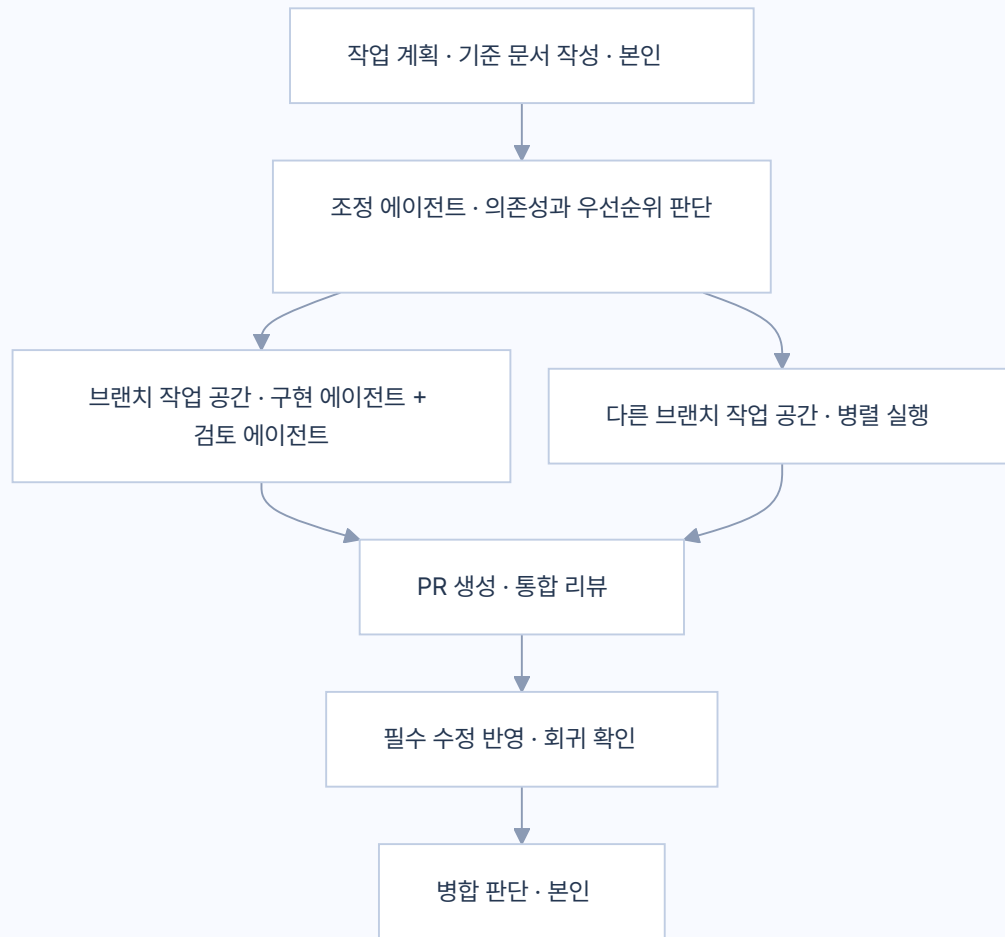
시도와 결과

리뷰에서 나온 필수 수정 13건을 반영하고 회귀를 확인한 뒤에만 병합했습니다. 문서를 복구한 뒤 이전 본문이 반환되던 결함도 병합 전에 고쳤습니다.

판단

코드를 직접 쓰지 않는 대신, 통과 기준을 정하고 그 기준으로 병합을 판단하는 것을 제 역할로 삼았습니다.

개선한 처리 흐름



계획과 기준은 제가 정하고, 구현과 검토는 역할을 나눈 에이전트가 브랜치별로 병렬 수행한 구조입니다.

5일 동안 PR 16개 병합 · 병합 전 필수 수정 13건 반영

8월 9일부터 13일까지 5일 동안 12시간씩 루프를 돌려 PR 16개를 병합했습니다. 병합 전 리뷰에서 나온 필수 수정 13건을 반영하고 회귀를 확인했습니다. 그중에는 문서를 복구한 뒤 이전 본문이 반환되던 결함도 있었습니다. 구현은 코딩 에이전트가 맡았고, 저는 루프 구성, 작업 계획과 기준 문서 작성, 결과 검토와 병합 판단을 맡았습니다.

결과의 범위 · 트레이드오프

코드를 직접 쓰지 않고 기준 문서와 리뷰 결과로 병합을 판단했습니다. 그래서 리뷰 기준에 없는 결함은 놓칠 수 있고, 기준 문서를 계속 보완하는 것이 과제로 남았습니다.

병합 PR

16개

2026.08.09 – 08.13

병합 전 필수 수정

13건

리뷰 지적 반영

브랜치당 에이전트

2개

구현 · 검토

Pilltip

의약품 정보를 쉬운 말로 안내하고, 사용자·가족의 복약 정보와 일정 관리를 돕는 개인 맞춤 AI 복약 서비스입니다.

시작 계기

부산대학교 졸업과제로 시작했습니다. 약품 정보가 여러 곳에 흩어져 있어, 지금 건강 상태에서 이 약을 먹어도 되는지 바로 확인하기 어려운 문제를 풀고자 했습니다.

[GitHub 저장소](#)

팀 구성과 내 역할

MY ROLE

백엔드·AI

의약품·건강기능식품 DB, 검색·자동완성, 복약 알림, 복용 위험·주의 정보 표출, 데이터 정제, AI 챗봇

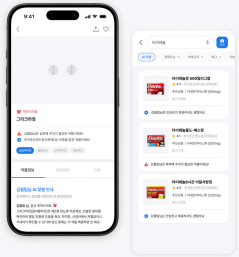
- | | |
|---|---|
| 01 백엔드·AI 기능 개발 | 02 의약품·건강기능식품 데이터 모델링, 검색·자동완성과 복용 위험 정보 표출 |
| 03 복약 알림과 기록 | 04 친구, 가족 기능 |
| 05 데이터 정제 파이프라인 구축 - 예상 \$200 대비 실제 \$11.18로 변환 | 06 RAG 기반 AI 챗봇 |
| 07 전시 피드백을 반영한 건강기능식품 지원 확대 | |

Android 1명	Kotlin 기반 Android 앱 전반
백엔드·보안 1명	사용자 DB, 로그인, 문진표, AES-GCM 암호화
디자인	디자이너

담당 기능과 서비스 화면

Solution

편리한 사용성
약품 검색 기능



DUR 기반 위험 태그 제공

사용자가 약물을 검색하면, 해당 약물의 성분과 사용자의 건강 정보를 활용하여 DUR(Drug Utilization Review) 평가를 통해 위험도를 평가하고 위험도 표시하여, 일반 사용자도 복용 위험을 직관적으로 파악할 수 있도록.

진료받은 병원과 복용처까지 표시하여

입원기간은 확인하고 복용처까지 확인하여

DUR 감지 약물

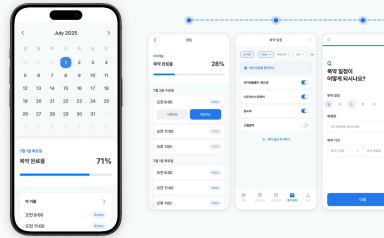
DUR 비감지 약물

의약품 검색·DUR 표출

의약품 DB, 검색·자동완성, 복약 위험정보 표출을 담당했습니다. 데이터 정제 파이프라인으로 약물 설명을 쉬운 말로 변환했습니다. 사진은 검색·주의 태그 화면입니다.

Solution

놓치지 않을 거예요
스마트 복약 알림

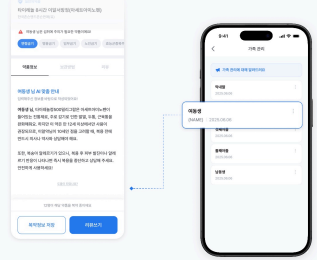


복약 알림·로그

FCM을 활용한 알림과 복약 기록 기능을 담당했습니다.

Solution

내가 직접 챙기는
우리 가족 건강 관리

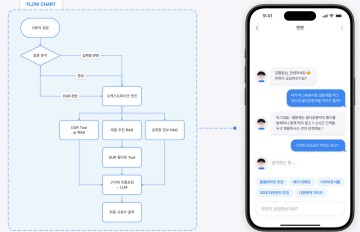


가족 프로필 전환

관리할 가족을 바꾸어 해당 프로필의 복약 정보를 확인하는 기능을 담당했습니다. 딥링크 기반 친구 초대도 구현했으며, 이 화면은 가족 전환 기능을 보여줍니다.

Solution

나만의 약사님
필립 AI 챗봇



AI 챗봇

질문에 따라 RAG 검색과 DUR 확인을 연결하는 오케스트레이션을 담당했습니다. 발표 자료의 예시 대화이며 의약 판단 정확도의 평가 결과는 아닙니다.

CASE 07

Pilltip / 데이터 파이프라인 · GPT Batch API · PySpark

의약품 정제 파이프라인 - 예상 \$200 대비 실제 \$11.18로 변환

Pilltip은 사용자가 지금 이 약을 먹어도 되는지 이해해야 하는 서비스라, 어려운 의약품 정보를 쉬운 말로 바꾸는 일이 핵심이었습니다. 같은 문장은 한 번만 변환해 실제 API 지출 \$11.18로 모든 약품을 처리했고, 2026년에는 PySpark로 재현해 효과를 측정했습니다.

2025.07 · 재현 2026.10

데이터 파이프라인

GPT Batch API

PySpark

MinHashLSH

배경

모든 약품의 설명을 쉬운 말로 제공해야 했지만, 약 4만 4천 건(약 1GB)을 그대로 변환하면 API 비용이 약 \$200으로 예상됐습니다. 약품 수를 줄이지 않고 예산 안에서 처리해야 했습니다.

해결하면 얻는 것

변환 비용이 예산을 넘으면 설명을 제공할 의약품을 줄여야 해 서비스 범위가 좁아집니다. 건당 변환 비용을 낮추면 같은 예산으로 모든 의약품을 다룰 수 있습니다.

원문 처리 예상 비용

약 \$200

실제 API 지출

\$11.18

완전 일치 중복 제거

-94.1%

문제 분석과 구현 과정

1 완전 일치 문장 중복 제거

문제 상황

모든 약품의 설명을 제공하려면 약 4만 4천 건, 1GB 규모의 원문을 변환해야 했고, 그대로 하면 API 비용이 약 \$200으로 예상됐습니다.

시도와 결과

약품 원문 100개를 직접 살펴 약 70%가 중복되는 내용임을 확인했고, 이 분석과 작업 계획으로 전처리 기간 7일을 확보했습니다. 원문을 문장 블록으로 나눠 완전히 같은 문장은 하나로 합치고, 각 약품과 문장 블록의 연결은 남겼습니다.

판단

약품 수를 줄이는 대신 같은 문장을 한 번만 변환해 재사용하기로 했습니다.

2 GPT Batch API 일괄 변환과 약품별 재연결

문제 상황

중복을 걷어낸 뒤에도 남은 변환 비용을 더 낮춰야 했습니다.

시도와 결과

사용 중에 바로 응답해야 하는 작업이 아니라 일괄 처리의 대기 시간을 받아들이고, 고유 문장을 GPT Batch API로 변환했습니다. 남겨 둔 약품 ID와 문장 블록 ID의 연결로 각 약품에 결과를 다시 붙여, 예상했던 약 \$200 대비 실제 API 지출 \$11.18로 변환을 마쳤습니다.

판단

중복 제거와 Batch 변환으로 약품 범위를 줄이지 않고 기능을 만들었습니다.

문제 분석과 구현 과정 (계속)

3 2026 PySpark 재현 - 어미 통일과 유사 문장 통합의 효과 측정

문제 상황

2025년 정제 코드 일부와 원본 파일이 남아 있지 않아, 완전 일치 이후 단계의 효과를 숫자로 설명할 수 없었습니다.

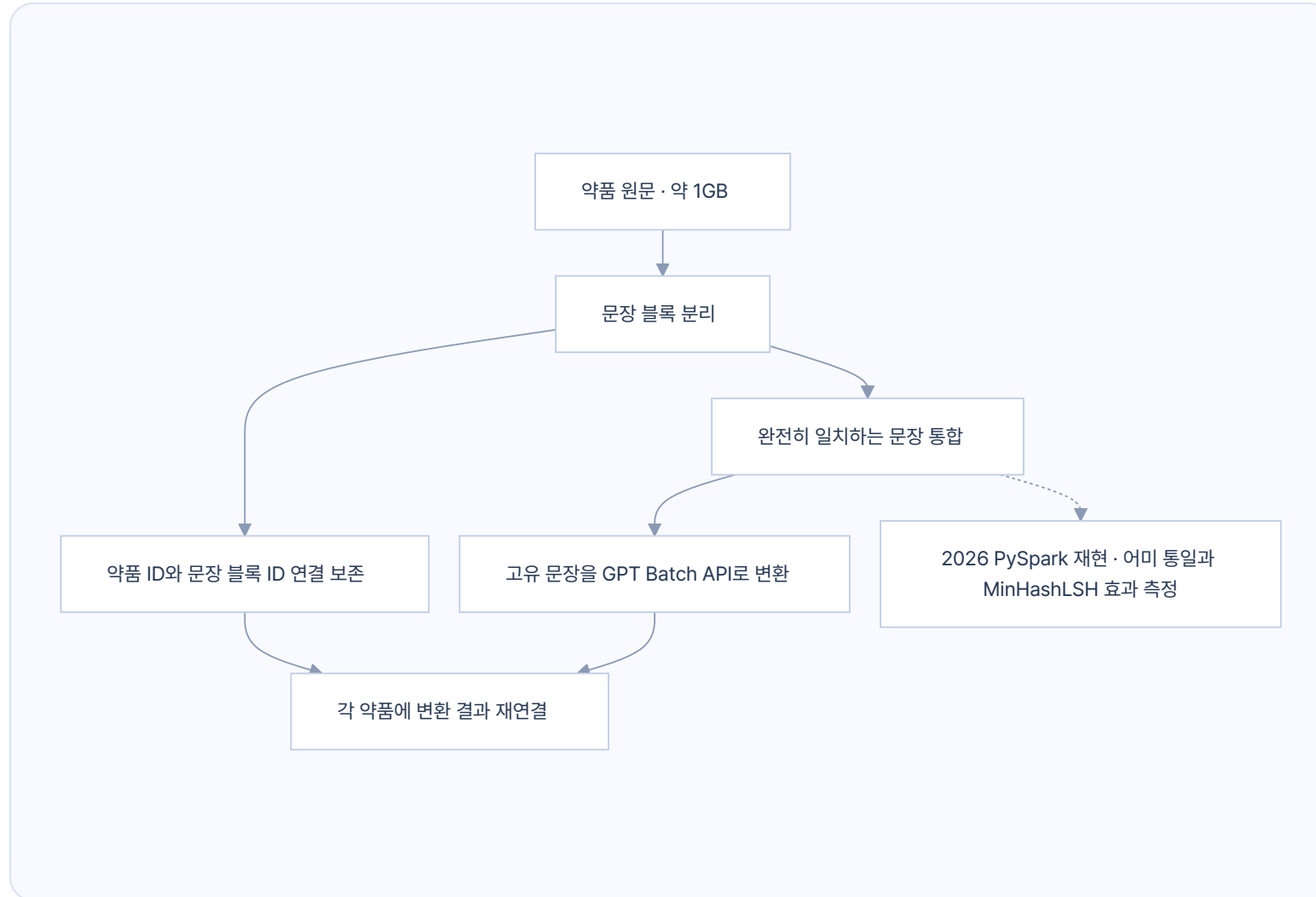
시도와 결과

식약처 공개 API로 42,714개 품목을 다시 수집해 324만여 개 블록을 여러 코어로 나눠 처리하고 데이터가 커져도 클러스터로 확장할 수 있도록 PySpark로 정규화와 완전 일치 중복 제거를 재현했습니다. 이어 어미 통일 사전과 MinHashLSH(글자 3-gram)를 적용하고, 임계값 0.1·0.2·0.3에서 줄어든 양과 합쳐진 표본을 비교했습니다.

판단

0.3이면 18.78%가 줄었지만 다른 질환(Lapp·Total 유당분해효소 결핍증)의 문장이 합쳐졌고, 0.2에서도 주의 문장 일부가 사라졌습니다. 잘못 합치면 틀린 주의사항을 전하게 되므로, 숫자와 조건이 같을 때만 합치는 안전장치와 함께 0.1(추가 6.69% 감소)을 골랐습니다.

개선한 처리 흐름



같은 문장은 한 번만 변환하고 약품별 연결을 보존해 결과를 다시 붙였습니다. 2026년에는 같은 과정을 PySpark로 재현해 측정했습니다.

실제 API 지출 \$11.18 · PySpark 재현에서 텍스트 94.1% 감소

전체 원문을 처리할 때 예상한 약 \$200 대비 실제 API 지출 \$11.18로 변환을 마쳤습니다. 설명할 약품의 범위는 줄이지 않고, 고유 문장의 변환 결과를 재사용해 예산 안에서 기능을 만들었습니다. 2025년 정제 코드 일부가 남아 있지 않아, 2026년에 식약처 공개 API로 42,714개 품목(순수 텍스트 960MB)을 다시 수집해 같은 정제 과정을 PySpark로 재현했습니다. 완전 일치 중복 제거로 텍스트를 744MB에서 44.1MB로 94.1% 줄였고, 어미 통일 사전으로 0.78%, MinHashLSH 유사 문장 통합으로 6.69%를 더 줄였습니다.

결과의 범위 · 트레이드오프

약 \$200은 사전 추산이고 \$11.18은 실제 지출입니다. 2026년 수치는 다시 수집한 데이터로 측정해 2025년 처리 데이터와 같지 않습니다. 재현 코드는 AI 코딩 도구로 작성했고, 실험 설계와 임계값 판단을 맡았습니다.

원문 처리 예상 비용

약 \$200

추산치

실제 API 지출

\$11.18

2025 변환 작업

완전 일치 중복 제거

-94.1%

2026 PySpark 재현 · 744MB → 44.1MB

CASE 08

Pilltip / AI Agent · 개인정보 유출 방지

복약 챗봇 - 건강 프로필을 외부 AI 입력에서 분리

Pilltip은 건강 상태에 맞춰 복약 정보를 안내해야 하지만, 그 건강 정보는 가장 민감한 개인정보입니다. 그래서 증상에 맞는 의약품 탐색은 모델에 맡기고, 건강 프로필을 쓰는 판단은 코드로 직접 관리하는 복약 챗봇을 만들었습니다.

2025.09

Spring AI

Weaviate

RAG

AI Orchestration

배경

사용자는 일상적인 말투로 증상을 설명하지만, 이는 약품과 건강기능식품 효능에 쓰인 단어와 달라 단어 검색으로는 맞는 제품을 찾기 어려웠습니다. 또한, 개인화에는 임신 여부·복용약·기저질환을 통한 연산이 필요했는데, 이를 외부 LLM에 보내면 민감정보가 나가는 범위를 통제하기 어려웠습니다.

해결하면 얻는 것

개인 건강 정보를 외부 AI로 보내면 유출 위험과 규제 부담을 지고, 보내지 않으면 개인 맞춤이라는 핵심 기능을 잃습니다. 판단을 내부에서 하면 두 가지를 함께 지킬 수 있습니다.

설계와 구현 과정

1 증상과 약물 효능을 연결하는 의미 검색 구현

문제 상황

사용자는 자신의 말투로 증상을 설명합니다. 그래서 제품에 적힌 사전적인 단어와 겹치지 않는 경우가 많았고, 이로 인해 알맞은 제품을 찾아오지 못하는 문제가 있었습니다. 따라서 관련 정보를 이어 줄 방법이 필요했습니다.

판단

처음에는 증상 동의어 사전을 만들기로 하고 코드까지 작성했습니다. 하지만 다뤄야 할 증상이 너무 많았고, 증상을 여러 개 말하면 사전 항목들이 서로 겹쳤습니다. 그중 그 사람에게 맞는 약을 골라야 하는데, 약물 설명이 모호하면 거의 모든 증상에 걸렸습니다. 그래서 자연어 증상과 약물 효능을 잇는 일은 의미 검색에 맡기고, 질문에 필요한 정보를 주도록 처리 경로를 나눴습니다.

구현

벡터 DB는 무료이면서 하이브리드 검색이 되는 후보 중 Milvus를 먼저 시험했지만 메모리를 많이 차지해, 비용적으로 이득이 없다고 판단했습니다. 그래서 상대적으로 경량인 Weaviate에 약물 효능을 저장해 증상과 뜻이 가까운 정보를 찾게 했습니다. 백엔드가 Spring이라 Python 컨테이너를 따로 띄우는 부하를 피하려고, 오케스트레이션도 Spring AI로 같은 서버에서 처리했습니다. 질문에 따라 약물 탐색·복약 위험·섭취량 정보를 처리하는 경로를 나눴습니다.

확인한 결과

사용자 시나리오 기반 E2E 테스트에서, 자연어로 증상을 말하면 질문에 맞는 처리 경로(약물 탐색·복약 위험·섭취량)로 이어지는 흐름을 확인했습니다. 증상별 정확도를 수치로 평가한 기록은 증상의 종류가 너무 많아 남기지 못했습니다.

설계와 구현 과정 (계속)

2 건강 프로필의 내부 처리와 외부 AI 입력 분리

문제 상황

개인화에는 임신 여부·복용약·기저질환을 활용한 연산이 필요했지만, 이 프로필을 외부 LLM에 넘기면 민감정보가 새어 나갈 수 있었습니다. 이는 보안적으로 치명적인 문제라고 판단해, 내부에서 이를 처리할 방법을 고안해야 했습니다.

판단

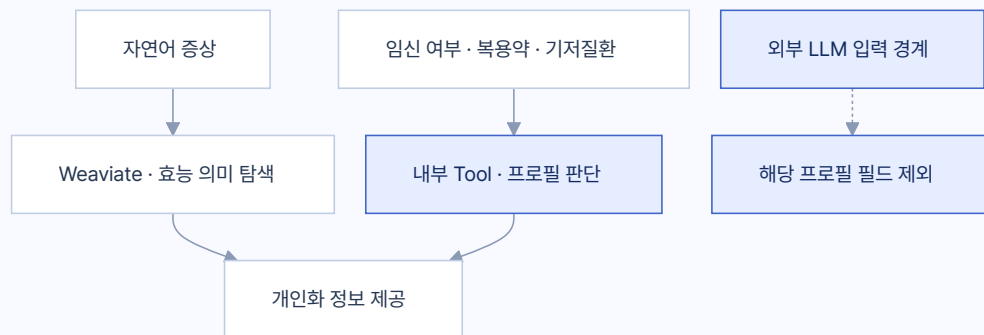
처음에는 건강 프로필을 외부 AI에 보내되 민감한 내용만 걸러내는 방식을 검토했습니다. 하지만 걸러야 할 표현과 경우가 너무 많아 코드로 모두 막기는 어렵다고 판단했습니다. 내부 서버에 AI 모델을 작동시키는 방법도 있었지만, 온프레미스 환경이어서 서버에 부하가 갈 수 있었습니다. 그래서 증상 표현을 이해하는 일과 건강 프로필의 조건을 판단하는 일을 나누고, 프로필을 쓰는 판단은 입력과 규칙을 직접 관리할 수 있는 내부 코드에 두었습니다.

구현

임신 여부·복용약·기저질환 필드는 외부 LLM 입력에서 뺐습니다. 내부 코드의 프로필 판단을 증상 검색 결과와 이어, 개인화 정보를 주도록 구현했습니다.

확인한 결과

임신 여부·복용약·기저질환은 외부 LLM 호출 입력에 넣지 않고, 내부 Tool이 이 정보로 금기·중복·상호작용을 판단해 해당 약을 걸러 내도록 처리 경로를 나눴습니다. 약 상세 화면의 AI 맞춤 안내에도 이 세 가지 정보는 보내지 않았습니다.



증상 의미 검색과 건강 프로필의 내부 판단 분리

자연어 증상으로 약품 정보를 찾으면서도, 임신 여부·복용약·기저질환 필드는 외부 LLM 입력에서 뺐습니다. 민감정보를 다루는 서비스에서 모델에 맡길 판단과 코드가 책임질 판단의 경계를 먼저 정하고 구현했습니다.

결과의 범위 · 트레이드오프

프로필 조건이 바뀌면 내부 코드도 고쳐야 하는 부담이 있습니다. 그래도 민감한 건강 정보가 외부 모델로 나가는 범위를 줄이는 게 먼저라, 이 판단은 AI에 맡기지 않고 코드에서 처리했습니다. 특정 프로필 항목을 외부 입력에서 뺐 것이며, 모든 경로의 유출 차단을 측정한 결과는 아닙니다. 사용자가 늘면 AI 처리를 별도 서비스로 분리하는 것이 맞다고 봅니다.

코드 밖에서도 이어지는 경험

Rhwp · HWPX 저장 오류 수정

Claude Code에 작업 계획과 기여 규칙을 주고 이슈 분석부터 PR까지 자동화해, textFlow 보존 오류 수정(PR #1213) 등 PR 7건이 병합되고 Contributor로 등재됐습니다.

[GitHub 저장소](#) ↗

APPTIVE · 백엔드 멘토링

멘티 경험을 교육 개선으로 연결했습니다. 멘티 12명을 대상으로 6회의 멘토링과 코드 리뷰를 진행했습니다.

백준 945일 연속 문제 해결

하루 한 문제를 목표로 solved.ac 기준 최장 945일 연속 문제를 해결했습니다. 누적 1,659문제, solved.ac Platinum IV.

[풀이 저장소](#) ↗

프로젝트 회고

ARCHITECTURE / 데이터 소유권

MSA 전환과 피드백

기능별 분리에서 데이터 소유권으로 관점을 바꿨습니다. 비동기 처리의 효과와 늘어난 운영 부담을 함께 기록했습니다.

기획 / 사용자 문제 정의

디자인 싱킹 회고

기능부터 고민하던 기획을 돌아보고, 팀원들이 겪은 문서 탐색의 불편함을 모아 함께 해결할 문제를 정의한 과정입니다.

[블로그 글 더보기 ↗](#)

김재형

AI Application Developer · Backend Engineer

kkuldangi2@gmail.com

martinel2.github.io

github.com/Martinel2

RAG 근거 충족률을 66.25%에서 91.25%로, AI 문서 편집 품질을 82.5%에서 91.2%로 높이며, AI가 틀린 근거로 답하거나 사용자 문서를 함부로 바꾸지 않도록 평가·승인·복구 구조를 설계해 온 AI 애플리케이션 개발자입니다.

정확도와 비용, 응답 시간의 트레이드오프를 평가로 확인한 뒤 서비스에 맞는 구현을 선택합니다.