세부 보고서

1. 프로젝트 소개

화면 맥락 인식 기반 사용자 작업 흐름 예측 지능형 에이전트

(Context-Aware Intelligent Agent for Workflow Monitoring and Prediction)

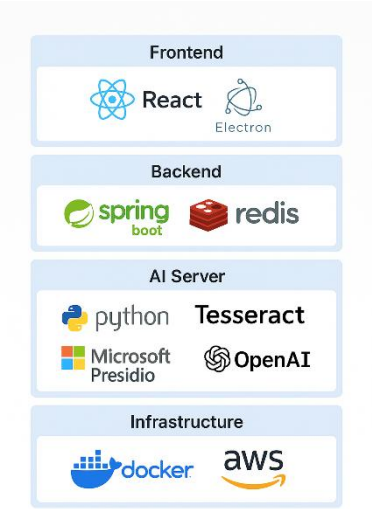
본 프로젝트는 사용자의 화면에서 발생하는 맥락적 변화를 능동적으로 감지하고, 이를 기반으로 예상 질문과 답변을 실시간으로 제안하는 AI 에이전트이다. 단순한 반응형 분석을 넘어, OCR 을 통한 텍스트 추출과 개인정보(PII) 마스킹, 이미지 설명 생성, 행동 및 질문 예측, 임베딩 기반 맥락 검색을 통합하여 사용자에게 지능적인 피드백을 제공한다. Redis 와 PostgreSQL 을 활용한 상태 관리, FastAPI 워커 기반의 분석 파이프라인, Spring SSE 를 통한 실시간 UI 반영 등 모듈화된 구조를 통해 확장성과 안정성을 확보하였다. 이를 통해 불필요한 데이터 전송·저장을 줄이고 개인정보 보호를 강화하며, 궁극적으로는 사용자 경험을 혁신하는 예측형 AI 도우미를 구현하였다.

2. 개발환경 (언어, Tool, 시스템 등)

본 프로젝트는 데스크톱 캡처·전송, 백엔드 전처리·캐시 관리, AI 분석 및 벡터 검색으로 이어지는 엔드투엔드 파이프라인을 구성한다. 프론트엔드는 데스크톱 환경에서 효율적인 화면 캡처와 가벼운 전처리를 담당하며, 백엔드는 이미지 중복 판별·캐싱 및 분석 대기열 관리, AI 서버는 OCR/PII 마스킹/설명 생성/행동·질문 예측을 수행한다. 분석 결과는 관계형 DB와 벡터 DB에 저장되어 이후의 맥락 검색과 선제 제안 기능을 지원한다.

- 프론트엔드: Electron, React
- 백엔드: Spring Boot(Java), Redis, PostgreSQL
- AI 서버: FastAPI(Python), Tesseract OCR, Microsoft Presidio(PII), OpenAI API(GPT-4.1-mini, GPT-5-mini, Embedding), FAISS(Vector DB)
- 배포/운영: Docker, (옵션) AWS S3

- 기타: SSE(Server-Sent Events), PostgreSQL LISTEN/NOTIFY



3. 개발배경 및 목적

3.1 배경

대형 언어모델(LLM) 기반 에이전트는 고도화되었지만, 여전히 "명시적 질의 → 반응" 구조에 머무는 한계가 존재한다. 사용자가 질문을 입력하거나 명령을 내리기 전까지 시스템이 능동적으로 다음 단계를 제시하기 어렵다는 점이 있다. 반면 실제 업무 맥락에서는 사용자가 "다음에 무엇을 해야 하는가"를 연쇄적으로 고민하는데, 예컨대 로그인 화면을 만들면, 자연스럽게 토큰 처리, 세션 만료, 에러 케이스, 보안 정책까지 이어서 준비해야한다. 그러나 기존 시스템은 이 맥락적 연쇄를 스스로 끌고 나가지 못한다

3.2 목적

본 프로젝트의 목적은 능동적 예측형 에이전트 구현으로, 화면의 변화를 주기적으로 관찰·요약·벡터화하여 사용자의 행동 히스토리와 현재 상황을 결합, 다음 행동을 예측하고 사용자가 묻기 전에 질문과 답변을 선제 제안한다. 또한 질의·응답·설명·행동 기록을 벡터 DB에 축적해 협업/리뷰/인수인계 등 지식 전승을 돕는다

- 미래 행동 예측: 과거/현재 맥락의 임베딩을 토대로 다음 단계 제시
- 선제적 질문 제안: 예상 질문과 그에 대한 답을 미리 제공

- 지식의 구조화: 설명/QA/행동 히스토리를 벡터로 축적하여 재활용

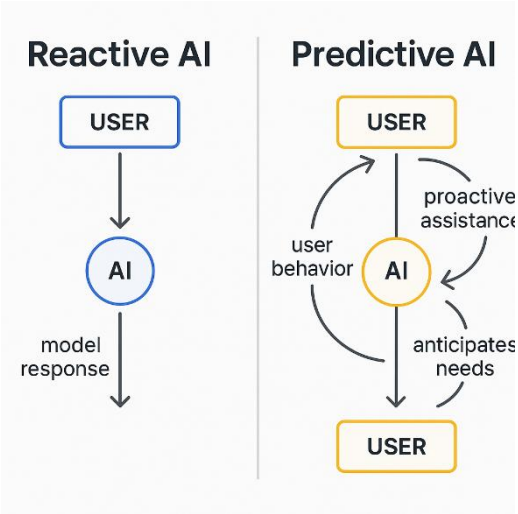


그림 1"반응형 AI vs 예측형 AI" 비교 다이어그램

4. 시스템 구성 및 아키텍처

4.1 전체 구조 개요

User PC(Electron Agent + React UI)

→ Back Server(Spring + Redis + PostgreSQL)

→ AI Server(FastAPI)

→ Vector DB(FAISS)로 이어지는 파이프라인

프론트는 캡처·1차 필터링·전송을, 백엔드는 2·3차 필터링과 캐시·대기열 관리를, AI 서버는 본 분석을 담당한다. 분석 결과는 DB에 기록되고, 벡터 DB로도 저장되어 이후 검색과 선제 예측에 활용된다.

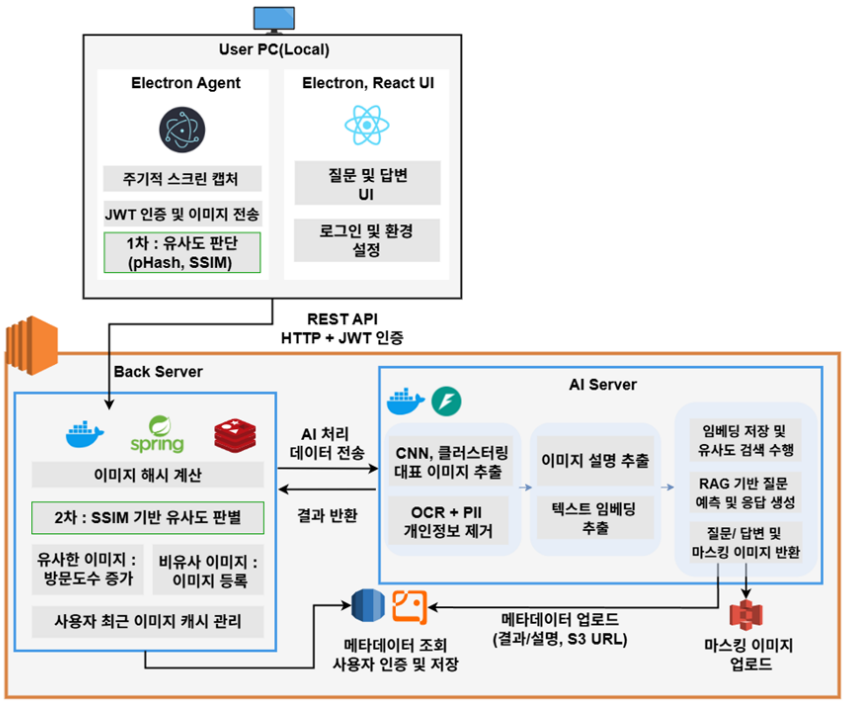


그림 2 전체 아키텍처 구조

4.2 주요 흐름(엔드투엔드)

① User PC (Electron Agent + React UI)

- 주기적 캡처: 약 1.5초 간격으로 전체 화면 캡처
- 1차 샘플링: SSIM 기준 0.9 이상이면 변화 미미로 간주, 전송 생략
- 전송 준비: Base64/바이트로 인코딩 후 Spring API로 업로드

② Back Server (Spring + Redis + PostgreSQL)

백엔드 서버(Spring)는 사용자 PC에서 전송된 이미지를 수신하여 중복 여부를 검증하고, 이후 AI 서버로 전달하기 위한 전처리 및 상태 관리 역할을 수행한다. 이 과정에서 Redis와 PostgreSQL이 함께 사용되며, 효율적이고 안전한 데이터 흐름을 보장한다.

먼저 2차 샘플링 단계에서는 dHash를 이용해 Redis에 보관된 최근 이미지들의 해시값

과 비교가 이루어진다. 해밍 거리(Hamming Distance)가 6 이하인 후보만을 선별함으로써, 의미 있는 차이가 없는 이미지들을 걸러낼 수 있다. 이어서 3차 샘플링 단계에서는 후보 이미지에 대해 SSIM(Structural Similarity Index)을 재검증한다. SSIM 값이 0.85 이상이면 기존 이미지 그룹에 속하는 것으로 판정하여 해당 그룹의 visitCnt와 lastVisitedAt 값을 갱신한다. 반대로 기준 미달 시에는 새로운 이미지로 판정된다.

저장 전략은 Redis와 PostgreSQL 간의 역할 분담을 통해 보안성과 성능을 동시에 확보하였다.

- Redis에는 원본 이미지, 썸네일, 최근 해시 리스트가 TTL(Time To Live) 1시간으로 보관되며, 만료 시 자동으로 삭제된다.
- PostgreSQL에는 screenshot_image 테이블에 메타데이터만 저장되며, 초기 분석 상태는 PENDING으로 지정된다.

이러한 구조를 통해 데이터베이스에는 원본 이미지가 남지 않고, Redis에서도 일정 시간이 지나면 자동 삭제되어 개인정보 보호를 강화할 수 있다.

Table "public.screenshot_image"

Column	Type	Collation	Nullable	Default
id	bigint		not null	generated by default as identity
analysis_result	text			
analysis_status	character varying(255)		not null	
captured_at	timestamp(6) without time zone			
image_hash	bigint			
last_visited_at	timestamp(6) without time zone			
visit_cnt	integer		not null	
user_id	bigint			

그림 4 screenshot_image 데이터 테이블 구조

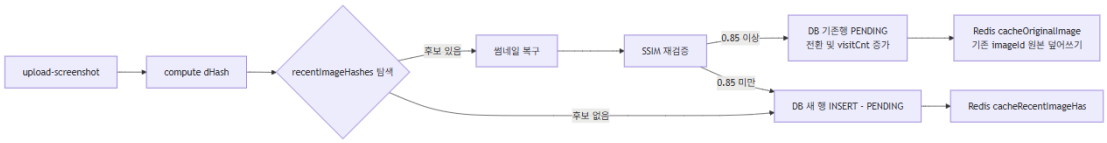
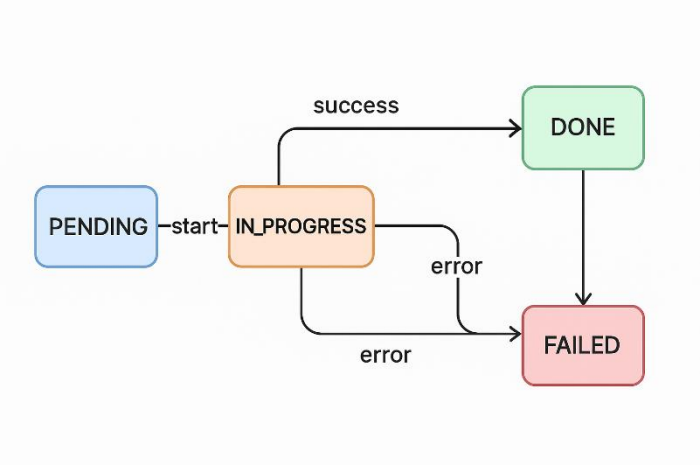


그림 3 백엔드 캐시 관리 — dHash→해밍거리→SSIM 기준→ 분기 도식

③ AI Server (FastAPI + Worker Thread)

AI 서버는 이미지 분석을 담당하는 핵심 계층으로, FastAPI 기반의 워커(Worker) 스레드가 동작한다. FastAPI 서버가 실행되면 on_startup() 단계에서 워커가 기동하며, 이후 지속적으로 PostgreSQL을 모니터링한다.

워커는 주기적으로 claim_batch()를 호출하여 analysis_status = PENDING 상태의 레코드를 가져오며, 가져온 즉시 IN_PROGRESS로 전환하여 중복 처리를 방지한다. 이후 Redis에서 user:{uid}:img:{img_id} 키를 사용해 원본 이미지를 조회한다. 만약 TTL 만료로 인해 원본이 존재하지 않는 경우에는 해당 레코드가 FAILED(original_expired_or_missing) 상태로 기록된다.

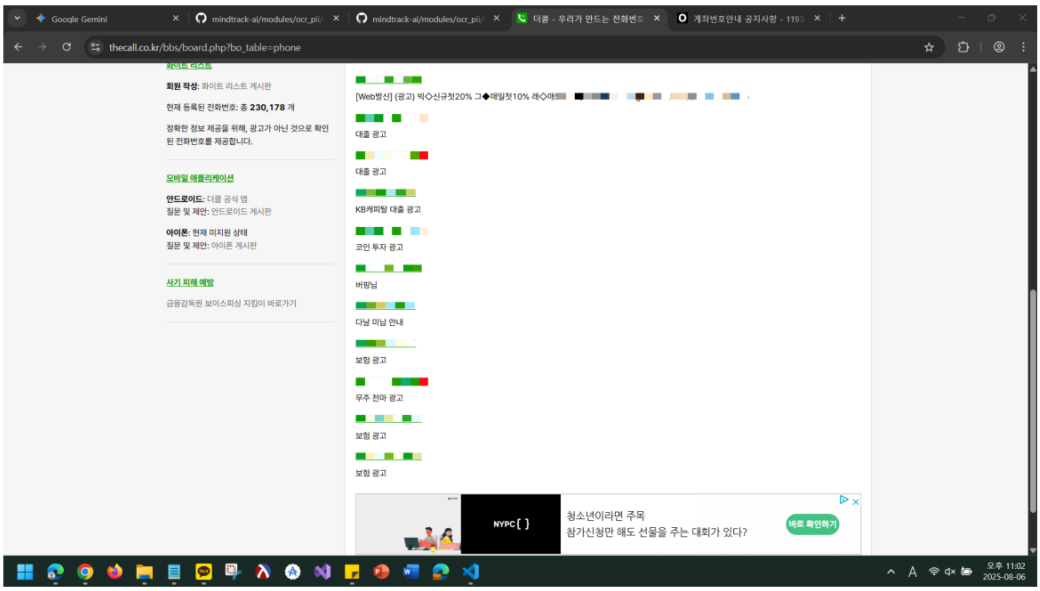


성공적으로 원본을 불러왔다면 아래와 같은 분석 파이프라인이 수행된다.

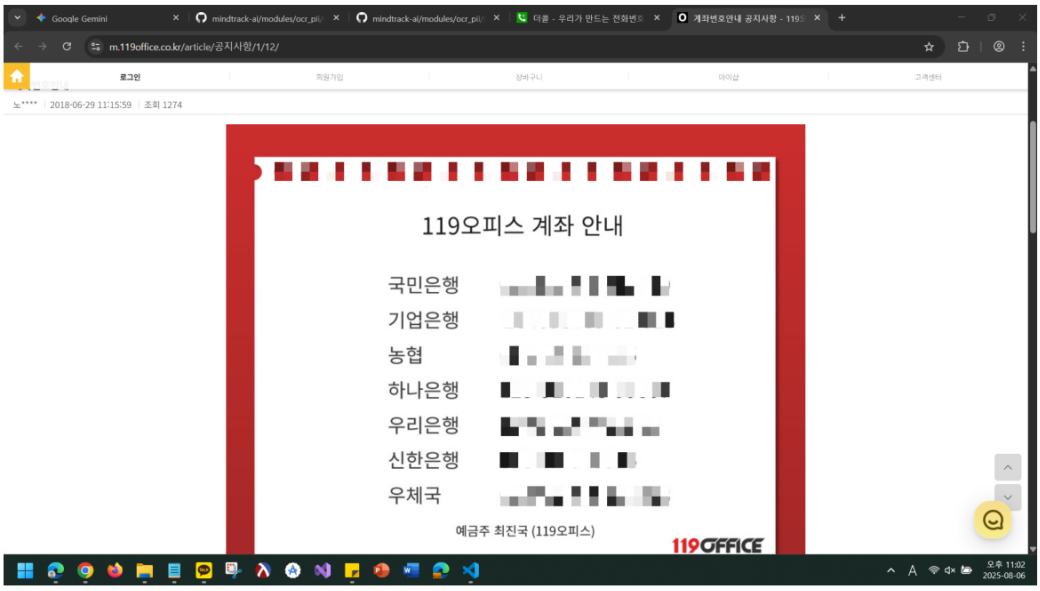
1. 대표 이미지 선정 (ImageClusterSelector)
    - 업로드된 모든 이미지에서 임베딩을 추출하고 KMeans 기반 클러스터링을 수행한다.
    - 가장 큰 클러스터에서 메도이드(medoid) 이미지를 대표 이미지로 선택한다.
2. OCR + PII 처리

- extract_text_data 함수가 pytesseract 를 이용해 화면 속 텍스트를 추출하며, 각 단어의 좌표 정보(x, y, width, height)를 함께 반환한다.
- 예시:
    - Box 1: (503, 293, 919, 35)
    - Box 2: (922, 504, 313, 37)
    - Box 3: (930, 571, 314, 29) ...
- Presidio Analyzer 를 통해 주민등록번호, 이메일 등 PII 를 탐지한다.
- 탐지된 좌표 영역은 OpenCV 의 blur_area 함수로 모자이크 처리한다.

Blurred PII Result



Blurred PII Result



3. 이미지 설명 생성 (ImageDescription)

- OpenAI API(gpt-4.1-mini)에 대표 이미지와 프롬프트를 전달한다.
- 장면에 대한 JSON 기반 상세 설명을 자동 생성한다.

4. 텍스트 임베딩 생성 (EmbeddingGenerator)

- OpenAI **Embeddings API(text-embedding-3-small)**로 설명 및 QA 텍스트를 벡터화한다.
- 생성된 벡터는 FAISS 벡터 DB 에 저장된다.

5. 컨텍스트 분석

- PostgreSQL 에서 최근 N 개(recent_k) 기록을 불러온다.
- 현재 설명과의 유사도를 계산하고, 상위 M 개(search_top_k)를 검색하여 맥락 기반 비교를 수행한다.

6. 행동 및 질문 예측 (ActionPredictor)

- OpenAI API(gpt-4.1-mini)를 이용해 사용자의 다음 행동과 예상 질문을 예측한다.
- _service.answer_question()을 통해 Top3 질문에 대한 자동 답변을 생성한다.

7. 최종 응답 구성

- 대표 이미지 경로, 설명 텍스트, 예측 행동 리스트, 예측 질문 리스트를 반환한다.
- 저장 구조:
    - screenshot_image: 메타데이터 + 상태 전이 (PENDING → IN_PROGRESS → DONE/FAILED)
    - suggestions 및 suggestion_items: 예측 질문 Top3 + 자동 생성된 답변
    - FAISS: 설명/QA/문맥 임베딩

완료 시 analysis_status 는 DONE 으로 업데이트되며, 실패 시 mark_failed()를 통해 FAILED 상태 및 원인 로그가 기록된다 오류 발생 시에는 mark_failed 함수가 호출되어 상태가 FAILED 로

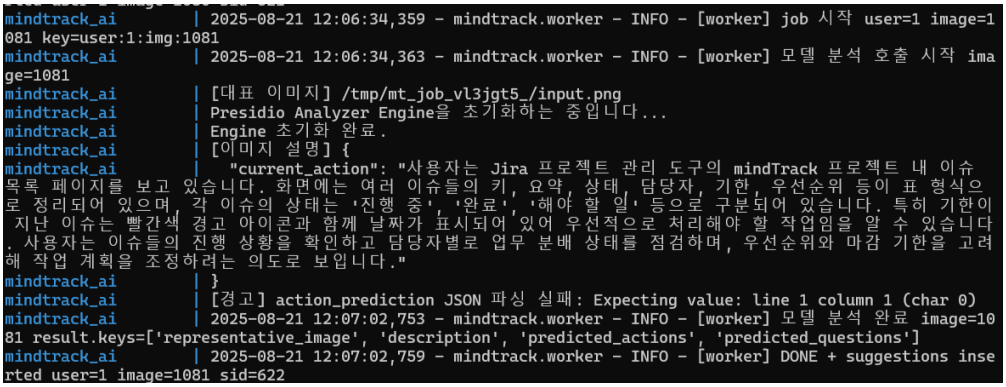


그림 5 AI 분석 과정 로그

업데이트되며, 원인 메시지가 기록된다.

④ 실시간 반영(React UI)

- LISTEN/NOTIFY + SSE: PostgreSQL의 테이블 변경을 Spring이 수신하여 SSE로 프론트에 전송
- UI 업데이트: 새 suggestion_items가 생성되면 화면에 자동 반영

5. 프로젝트 주요기능 및 구조도

5.1 기능 및 특징

- 중복 억제형 캡처 파이프라인: 1차(클라이언트 SSIM 0.9), 2차(dHash/해밍 ≤6), 3차(SSIM 0.85)로 네트워크·스토리지 낭비 최소화
- 개인정보 보호 설계: 원본은 RDB에 저장하지 않고 Redis TTL(1시간) 로만 보관, OCR 후 PII 마스킹
- 설명/예측/QA 통합: GPT로 설명 생성 → 행동/질문 예측 → Top3 자동답변 생성 및 저장
- 지식의 벡터화: 설명/QA/맥락을 FAISS에 저장, 맥락 기반 검색·선제 제안에 재활용
- 실시간 피드백 루프: LISTEN/NOTIFY + SSE 로 프론트 자동 반영

5.2 개발 과정(주요 단계와 방법)

1. 아이디어/요건 정의: 반응형의 한계를 예측형으로 전환하는 사용자 가치 정의
2. 아키텍처 설계: 캡처→전처리→대기열→분석→저장→반영의 흐름 분리/결합
3. 백엔드/AI 모듈 동시 개발: Spring(전처리·캐시·대기열), FastAPI(워커·분석 파이프라인)
4. 품질 튜닝: SSIM 임계값(0.9/0.85), dHash 해밍 거리(≤6) 탐색·테스트
5. 보안/프라이버시: PII 마스킹, 원본의 TTL 캐시화, 키 패턴(user:{uid}:img:{img_id}) 정합화
6. 실시간 UX: LISTEN/NOTIFY + SSE로 사용자 피드백 지연 최소화
7. Docker 통합: 캡처→분석→UI 반영까지 자동 시연

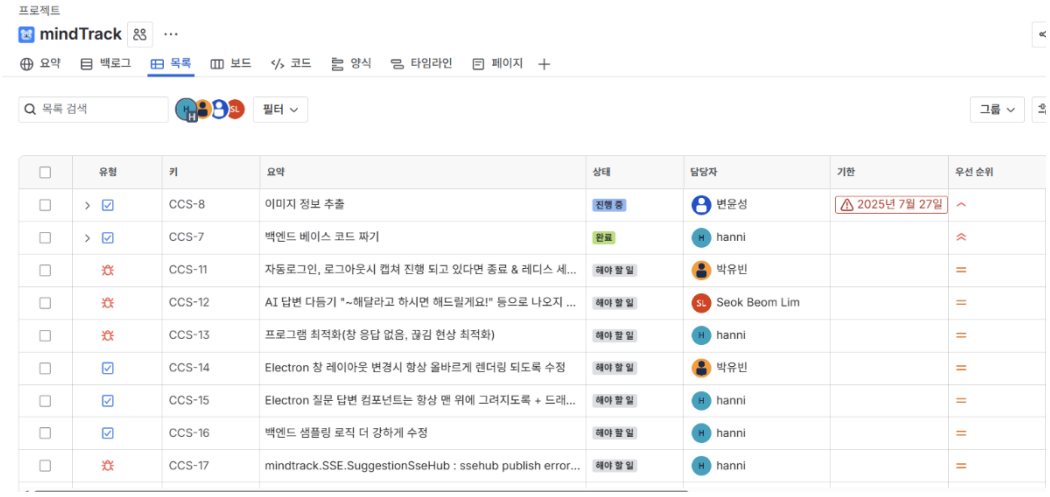


그림 6 개발과정 업무 생성 및 할당 — Jira

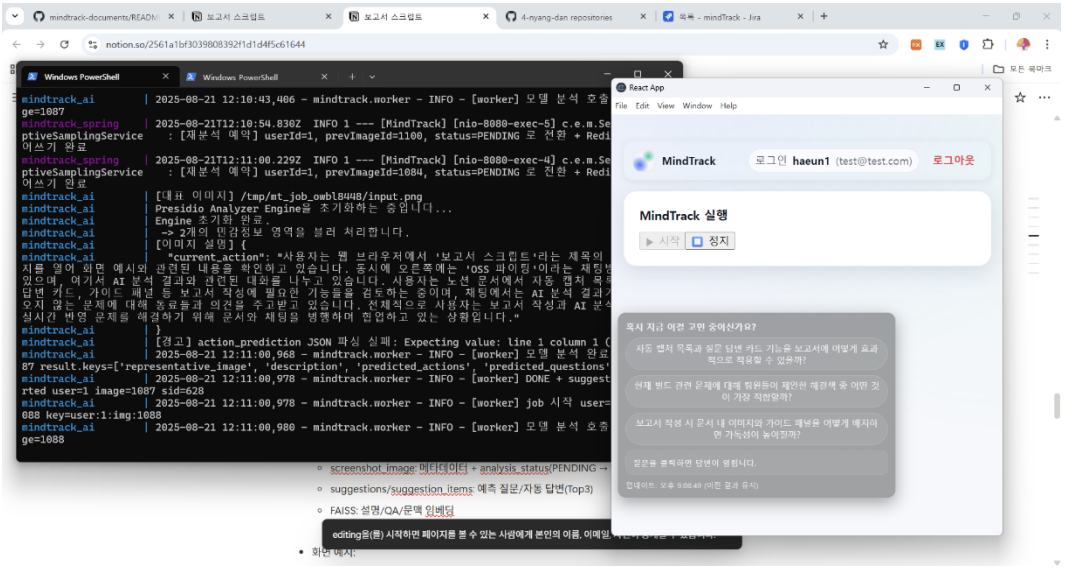


그림 7 도커 + 로컬 환경 테스트

5.3 결과물 및 화면 예시

본 프로젝트의 결과물은 분석 데이터의 저장 구조와 사용자 화면(UI) 반영 두 축으로 구분된다.

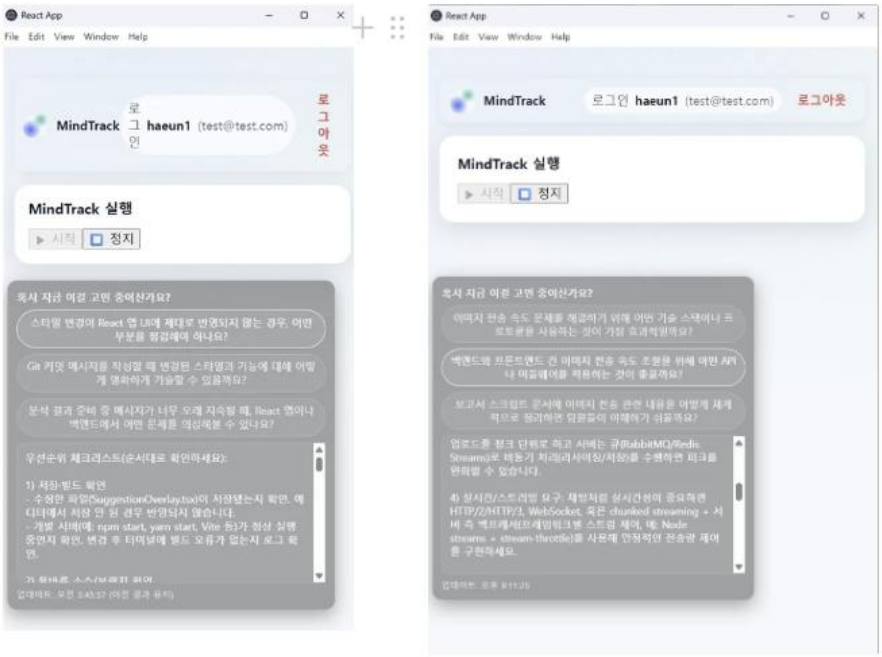
1. 결과 저장 구조
    - screenshot_image 테이블
        - 캡처된 이미지의 메타데이터(사용자 ID, 캡처 시각, 파일 경로 등)와 함께 분석 상태(PENDING → IN_PROGRESS → DONE/FAILED)가 기록된다.
    - suggestions 및 suggestion_items 테이블
        - 분석 과정에서 예측된 질문과 그에 대한 자동 생성 답변이 Top3 형태로

저장된다.

- suggestions는 상위 질문 세트를, suggestion_items는 개별 질문·답변을 저장하는 하위 구조로 설계되어 있다.

- FAISS 벡터DB
  - 이미지 설명, 자동 생성된 QA, 최근 문맥 임베딩이 저장된다.
  - 이후 유사도 검색과 컨텍스트 기반 제안에 활용된다.

2. 화면 예시 (React UI)
   - 상태 표시
     - 분석이 진행 중일 때는 "분석 중" 상태가, 오류가 발생하면 "실패" 상태가 화면에 출력된다.
   - 질문·답변 카드
     - 분석이 완료되면 Top3 질문이 카드 형태로 제시된다.
     - 사용자가 질문을 선택하면, 자동 생성된 답변이 카드 하단에 표시된다.



5.4 실행 및 테스트 방법(요약 가이드)

- 환경 준비: Docker로 Redis/PostgreSQL/FastAPI/Spring/Electron 구성, (옵션) S3 크리덴셜
- 기동 순서: DB/Redis → Spring → FastAPI(워커 자동 기동) → Electron/React
- 기능 테스트:

1. 화면 변화를 만들어 캡처 유도(1.5초 간격) → 전송 여부(SSIM 0.9) 확인

2. 중복 이미지 업로드 후 dHash/해밍≤6 후보 선별과 SSIM 0.85 재검증 확인

3. PostgreSQL screenshot_image의 상태 전이(PENDING→IN_PROGRESS→DONE/FAILED 확인

4. Redis 키(user:{uid}:img:{img_id})로 원본 존재 여부 및 TTL 만료 시 FAILED 기록 확인

5. suggestions/suggestion_items 생성 및 Top3 자동 답변 삽입 확인

6. LISTEN/NOTIFY 트리거 후 SSE 이벤트가 프론트에 반영되는지 확인

부록: 설계 상의 핵심 결정과 이유*

본 프로젝트의 설계 과정에서는 단순한 기능 구현을 넘어, 비용 절감·프라이버시 보호·안정적인 상태 관리·실시간 사용자 경험 보장이라는 4가지 핵심 목표를 반영하였다. 아래는 이러한 설계 상의 주요 결정과 그 이유를 상세히 정리한 것이다.

1. 3단계 샘플링으로 전송/저장 비용 절감

   이미지 전송과 저장 비용을 최소화하기 위해 다단계 샘플링 전략을 채택하였다.

   1차 필터링은 클라이언트 단에서 SSIM ≥ 0.9 기준을 적용하여, 거의 동일한 스크린샷은 서버 전송 전에 걸러낸다.

   2차 필터링은 백 서버(Spring)에서 Redis에 저장된 최근 이미지 해시(dHash)와 비교하며, 해밍 거리 ≤ 6인 후보만 추린다.

   마지막 3차 필터링에서는 SSIM ≥ 0.85 조건을 재검증하여, 기존 그룹에 속하는 경우 DB의 visitCnt와 lastVisitedAt 값을 갱신하고, 기준을 충족하지 못하면 새로운 이미지로 판정한다.

   이 구조를 통해 중복 이미지 저장을 최소화하고, 네트워크 전송 및 스토리지 점유를 크게 줄일 수 있었다.

2. 프라이버시 우선 정책

   프로젝트는 민감한 사용자의 화면 이미지를 다루기 때문에 프라이버시 보호를 최우선 가치로 삼았다.

- Redis: 원본 이미지, 썸네일, 해시 리스트를 TTL 1시간으로만 보관하며, 만료 시 자동 삭제된다.
- PostgreSQL: 영구 저장되는 것은 이미지 원본이 아닌 메타데이터에 한정된다. screenshot_image 테이블에는 파일 경로나 썸네일 대신 캡처 시점, 해시, 분석 상태 등의 관리 정보만 기록된다.

이로써 원본 이미지가 장기 보관되지 않아, 데이터 유출 및 개인정보 노출 가능성을 최소화하였다.

3. 상태머신 기반 워커 처리

AI 서버(FastAPI)는 상태머신 기반 처리 로직을 사용하여 안정성과 예측 가능성을 확보하였다.

- 초기 상태는 PENDING: 아직 분석되지 않은 대기 상태
- 워커가 할당 즉시 IN_PROGRESS로 전환: 중복 처리 방지
- 분석 성공 시 DONE, 오류 발생 시 FAILED로 마무리

또한, Redis TTL 만료로 원본 이미지를 찾을 수 없는 경우에는 FAILED(original_expired_or_missing)로 기록해 오류 원인 추적이 가능하다.

이와 같은 명시적인 상태 전이(state transition)는 다수의 워커가 동시에 작업하더라도 안전한 병렬 처리를 보장한다.

4. 표준화된 키 스킴 도입

Redis와 DB 간의 데이터 추적성을 일관되게 유지하기 위해,

user:{uid}:img:{img_id} 형태의 표준 키 스킴을 설계하였다.

이 키 구조는 사용자별/이미지별로 명확한 구분이 가능하고,

Redis 캐시와 PostgreSQL 메타데이터를 쉽게 매칭할 수 있게 해주며,

향후 로그 분석이나 오류 추적 시에도 강력한 기준점으로 작용한다.

5. 실시간 사용자 경험 (UX) 강화

사용자는 이미지 분석이 완료되자마자 결과를 확인할 수 있어야 한다. 이를 위해 PostgreSQL

의 LISTEN/NOTIFY 기능과 Spring의 SSE(Server-Sent Events)를 결합하였다.

- AI 서버가 분석을 완료하면 PostgreSQL에서 NOTIFY 이벤트가 발생한다.
- Spring 백엔드는 해당 이벤트를 감지하여, SSE 채널을 통해 즉시 사용자 프론트(UI)로 결과를 push한다.

이 구조 덕분에 사용자는 별도의 새로고침 없이도 분석 완료 알림과 결과를 실시간으로 받아볼 수 있다.

6.기대효과

1. 코드 오류 지적, 환경 변수 문제 해결, 실행 중 에러 분석 등 실제 개발 과정에서 자주 발생하는 상황에 대해 더 정밀하고 맥락적인 답변 제공
2. 구현된 핵심 기능
    - 사용자의 미래 행동 예측
    - 앞으로 발생할 수 있는 질문 선제 제시
    - 과거·현재 맥락 반영을 통한 향상된 답변 제공
3. 토큰 사용 최적화, 지연 시간 단축, MCP·에이전트 연동 등을 통해 지속적 성능 개선 및 확장 가능

7. 활용분야

1. 개발자 지원 도구: 코드 오류 탐지, 환경 변수 문제 해결, 실행 에러 분석 등 실제 개발 환경에서 활용 가능
2. 맞춤형 AI 비서: 사용자의 맥락을 이해하고 미래 질문을 선제적으로 제시 → 개인화된 업무 지원
3. 스마트 산업 영역: 스마트 팩토리·스마트 시티와 같이 복잡한 시스템 데이터를 입력값으로 활용, 맥락적 메모리 기반 예측 제공
4. 차세대 AI 서비스: 단순 질의응답을 넘어 능동적으로 사용자의 흐름을 예측하고 지원하는 새로운 AI 모델로 확장

8. 한계점 및 개선사항

1. 세션 관리 한계: 현재는 세션 종료 처리가 일부 상황에서 지연되거나 중복 처리 가능성이 있어, 엔드포인트 레벨의 이벤트 감지를 통해 개선 필요
2. 데이터 TTL 한계: TTL 만료 전 데이터 손실이나 만료 후 재요청 시 처리 지연 발생 가능 → 다단계 캐싱 전략과 백업 로직 필요
3. 확장성 한계: 현재 구조는 단일 Redis/DB 의존성이 크기 때문에, 대규모 트래픽 환경에서는 Redis Cluster, DB 샤딩 등의 확장 전략 필요
4. 분석 정확도 개선: dHash/SSIM 기반의 중복 판정은 완벽하지 않아, 추가적인 ML 기반 유사도 판별 모델로 고도화 가능
5. • 토큰 사용량 문제
   A. 화면 기반 맥락 분석 과정에서 불필요하게 많은 토큰이 사용됨
   B. 토큰 최소화 전략 및 요약 기반 컨텍스트 관리 기법 필요
6. • 지연 시간 문제
   A. 분석 및 응답 생성 과정에서 일정 수준의 딜레이 발생
   B. 파이프라인 최적화, 캐싱 강화, 비동기 처리 개선을 통한 속도 향상 필요