최종 프로젝트 · 아키텍처 가이드

# API 서버는 주문만 받고, 멀티에이전트는 워커가 돌린다

프런트엔드 · FastAPI · Celery 워커 · 외부 API · 오토힐링 · Docker
— 전체 구조, 예시 폴더 구조, 빌드·실행 구성

AI 애플리케이션 개발 과정 · 최종 프로젝트 공통 자료

오늘의 목표

# 이 자료를 다 보고 나면 할 수 있어야 하는 것

멀티에이전트 + LLM + 외부 API가 섞인 서비스를 **"요청 받는 곳"**과 **"일하는 곳"**으로 나눠 설계한다.

### ① 왜 나누는지 설명한다

LLM 호출을 HTTP 요청 안에서 기다리면 무엇이 깨지는지 말할 수 있다.

### ② 전체 그림을 그린다

프런트 → API → 큐 → 워커 → LLM/외부 API, 그리고 결과가 돌아오는 길을 그릴 수 있다.

### ③ 폴더를 나눈다

`backend/`, `agents/`, `shared/`, `docker/`가 각각 무엇을 담고 어떤 이미지가 되는지 안다.

### ④ 오토힐링 위치를 안다

재시도·폴백·자동 재시작을 **어느 계층**에 넣어야 하는지 구분한다.

문제 상황

# LLM을 API 요청 안에서 그냥 호출하면?

멀티에이전트 한 번 = LLM 호출 여러 번 + 외부 API 여러 번 → **수십 초 ~ 수 분**.

```python
# ❌ 이렇게 만들면 안 됩니다
@router.post("/analyze")
async def analyze(body: Req):
    result = await graph.ainvoke(body.dict())
    #  ↑ Supervisor → 진단 → 복구 → 검증
    #    LLM 6번 + 외부 API 4번 … 90초
    return result
```

사용자는 90초 동안 로딩 스피너만 봅니다. 그 사이에 무슨 일이 생길까요?

- **타임아웃** — 로드밸런서·프록시(보통 30~60초)가 먼저 연결을 끊음 → 504
- **새로고침 = 작업 증발** — 브라우저를 닫으면 결과를 받을 길이 없음
- **서버 재시작 = 작업 유실** — 배포하는 순간 진행 중이던 작업이 사라짐
- **재시도 불가** — 외부 API가 잠깐 죽었을 때 "나중에 다시"가 안 됨
- **서버가 막힘** — 무거운 작업이 API 서버 자원을 잡아먹어 로그인·조회까지 느려짐

한 장으로 이해하기

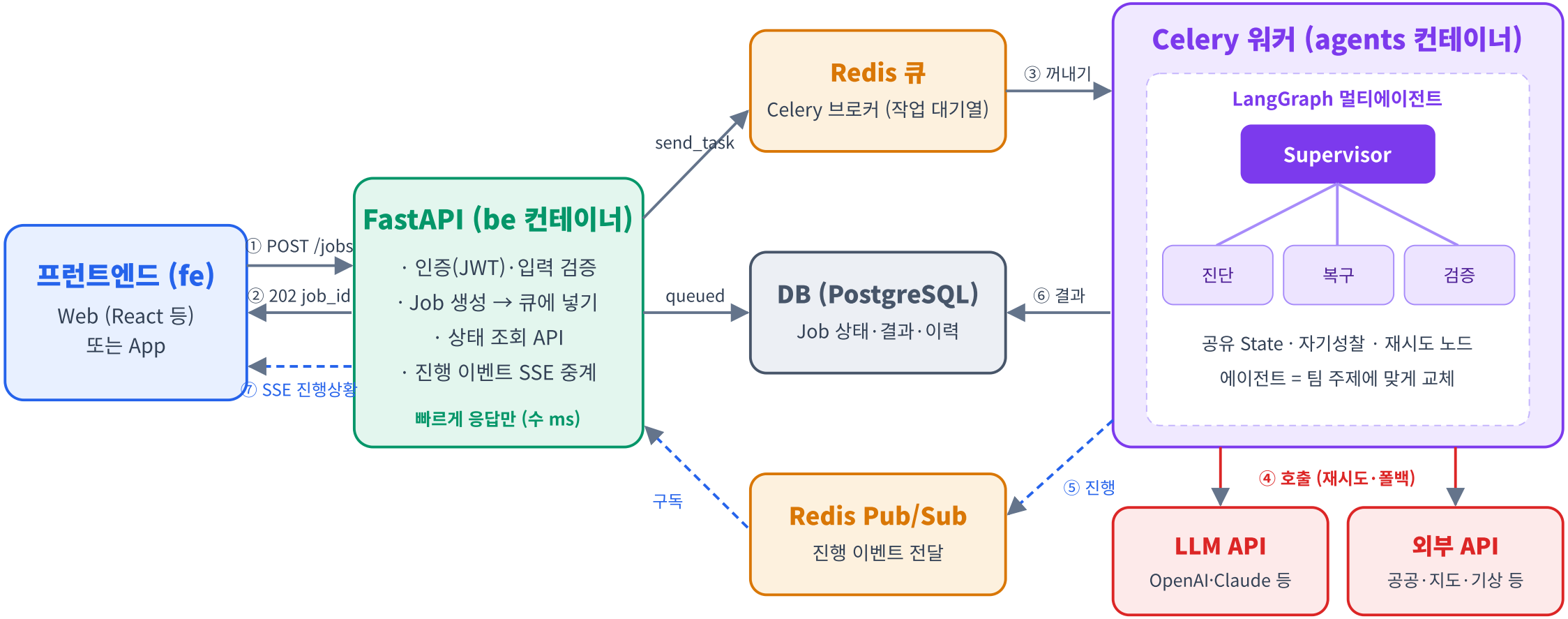
# 식당으로 비유하면

홀 직원이 직접 요리까지 하면 손님 응대가 멈춥니다. 그래서 **주문 접수**와 **조리**를 나눕니다.

| 식당 | 우리 시스템 | 하는 일 |
|---|---|---|
| 손님 | 프런트엔드 | 웹/앱 화면. 주문하고, 진행 상황을 봄 |
| 홀 직원 (카운터) | FastAPI API 서버 | 주문 확인(인증·입력 검증) 후 주문표를 걸고 **바로** 진동벨을 건넴 |
| 주문표 | Job (DB 한 행) | 무엇을 요청했고 지금 어떤 상태인지 기록 |
| 주문 레일 | 큐 (Redis) | 주방이 처리할 순서대로 주문표가 대기하는 곳 |
| 주방 / 셰프 팀 | 워커 / 멀티에이전트 | 주문표를 하나 꺼내 Supervisor·전문 에이전트들이 협업해 처리 |
| 식자재 납품업체 | LLM · 외부 API | 가끔 늦거나 품절 → 재시도하거나 다른 업체로 (= 오토힐링) |
| 진동벨 | job_id + SSE | "지금 굽는 중", "완성!"을 손님에게 알려줌 |

전체 구조

# 최종 프로젝트 표준 아키텍처



구성요소

# 누가 무엇을 책임지나

| 구성요소 | 책임 (해야 할 일) | 하면 안 되는 일 | 기술 예시 |
|---|---|---|---|
| 프런트엔드 | 요청 보내기, job_id로 진행 상황 구독, 결과 표시, 에이전트 사고 과정 가시화 | LLM API 키 보관·직접 호출 | React/Next.js, Streamlit, Flutter |
| API 서버 | 인증·권한, 입력 검증, Job 생성, 큐 투입, 상태 조회, SSE 중계 | LLM·에이전트를 요청 안에서 실행 | FastAPI, Pydantic, SQLAlchemy |
| 큐 / 브로커 | 작업을 안전하게 보관하고 워커에게 순서대로 전달 | 결과를 영구 저장 (그건 DB) | Redis (Celery 브로커) |
| 워커 | Job 꺼내서 LangGraph 실행, 진행 이벤트 발행, 결과 저장, 실패 시 재시도 | HTTP 요청을 직접 받음 | **Celery** (+ Flower 모니터링) |
| DB | Job 상태·결과·감사 로그의 "단일 진실 공급원" | — | PostgreSQL, Supabase |
| LLM·외부 API | 워커 안의 **통합 계층(integrations)**에서만 호출, 타임아웃·재시도·폴백 적용 | 에이전트 코드 곳곳에서 날것으로 호출 | LiteLLM, httpx, tenacity |

흐름

# 요청 한 건의 일생 (9단계)

핵심: API는 ②에서 **즉시** 응답하고 빠진다. 무거운 일은 ④~⑧, 워커가 한다.

FE ① 사용자가 "분석 요청" 클릭 → `POST /api/v1/jobs`

API ② 검증 후 DB에 Job 저장 `status=queued`

API ③ Celery로 `send_task("agents.run_agent_job", [job_id])` → **202 + job_id** 응답

FE ④ `GET /jobs/{id}/events` (SSE) 구독 시작

Worker ⑤ Celery 워커가 태스크 수신 → `status=running`

Worker ⑥ LangGraph 실행. 노드가 끝날 때마다 `{"type":"step","node":"diagnosis"}` 발행

Ext ⑦ 외부 API 실패 → 재시도 / 폴백 (오토힐링)

Worker ⑧ 결과 DB 저장 `status=succeeded` + `done` 이벤트

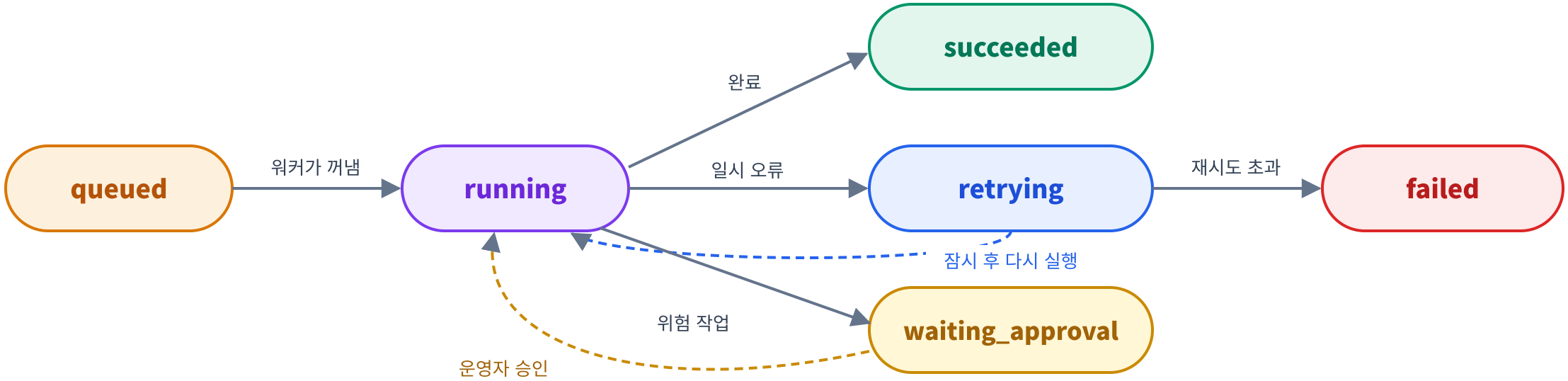
FE ⑨ done 수신 → `GET /jobs/{id}`로 최종 결과 표시

새로고침해도 괜찮습니다. job_id만 있으면 `GET /jobs/{id}`로 언제든 현재 상태·결과를 다시 볼 수 있어요.

JOB 상태

# Job은 "상태 머신"으로 관리한다

DB의 `jobs.status` 한 컬럼이 화면·재시도·오토힐링·감사 로그의 기준이 됩니다.




※ **waiting_approval**: 서비스 재시작처럼 위험한 복구는 사람 승인 후 진행 (가드레일·Human-in-the-loop). 팀 주제에 필요 없으면 생략 가능.

실시간 표시

# 워커의 진행 상황을 화면에 보여주는 3가지 방법

| 방식 | 동작 | 장점 | 단점 |
|---|---|---|---|
| **폴링** | 프런트가 2초마다 `GET /jobs/{id}` | 가장 쉬움. 일단 이것부터 | 지연·불필요한 요청 |
| **SSE** ⭐ 권장 | 서버 → 브라우저 한 방향 스트림. 이벤트가 생길 때마다 push | HTTP 그대로, 구현 간단, 자동 재연결 | 단방향 (요청은 별도 POST) |
| **WebSocket** | 양방향 연결 유지 | 채팅처럼 주고받기가 잦을 때 | 연결 관리·인증·배포 설정이 더 복잡 |

**추천 순서:** ① 폴링으로 흐름을 먼저 완성 → ② SSE로 교체해 "에이전트 사고 과정 가시화" 구현

**LLM 토큰 스트리밍도 같은 길**로 흘려보내면 됩니다: 워커가 `{"type":"token","text":"..."}` 이벤트를 발행 → SSE로 전달.

선택 기준

# 무엇을 워커로 보내고, 왜 Celery인가

### 워커로 보낼 작업

- 멀티에이전트 실행 (LLM 여러 번)
- 외부 API 여러 개를 엮는 작업
- 재시도·승인 대기가 필요한 작업
- 주기 작업 (모니터링·이상 감지) → Celery Beat

### API에서 바로 해도 되는 것

- DB 조회·저장, 로그인
- 수 초 안에 끝나는 **단일** LLM 응답 스트리밍

| Celery 용어 | 뜻 (식당 비유) |
|---|---|
| **Broker** | 태스크 메시지가 대기하는 곳 = 주문 레일. 우리는 **Redis** |
| **Task** | 워커가 실행할 함수 = 레시피. `@app.task`로 등록 |
| **Worker** | 태스크를 꺼내 실행하는 프로세스 = 주방. 컨테이너 여러 개로 확장 |
| **Queue** | 태스크를 나누는 줄. 예: `agents`, `monitor` |
| **Beat** | 정해진 시각에 태스크를 넣는 스케줄러 = 알람 |
| **Flower** | 워커·태스크 상태를 보는 웹 대시보드 (시연·모니터링용) |

왜 Celery? 파이썬 백그라운드 작업의 사실상 표준 → 자료·예제가 가장 많고, 재시도·시간 제한·스케줄·모니터링이 모두 갖춰져 있음.
(가벼운 대안: arq, Taskiq / 데모 전용: FastAPI BackgroundTasks)

예시 폴더 구조 ①

# 저장소 최상위: 소스 폴더 = 이미지 단위

```
team-project/
├── frontend/      # → {팀}-fe 이미지 (웹/앱)
├── backend/       # → {팀}-be 이미지 (FastAPI)
├── agents/        # → {팀}-agents 이미지
│                  #   (Celery 워커 + LangGraph)
├── shared/        # be·agents 공용 코드
├── tests/
├── docs/          # 산출물(설계서·결과서)
├── docker/        # 빌드·실행 스크립트 (뒤에서)
├── .env           # 실제 값 — 커밋 금지
├── .env.example   # 키 이름만
└── .gitignore
```

## be와 agents를 왜 나누나?

**be**는 가볍고 빨라야 하고(요청 응답), **agents**는 무겁고 오래 걸립니다(LangGraph·LLM SDK). 이미지를 나누면 **따로 빌드·배포·확장**할 수 있고, 에이전트를 고쳐도 API 서버는 그대로 둘 수 있습니다.

## shared/에는 무엇을?

양쪽이 똑같이 알아야 하는 것만: 설정(Settings), DB 연결, `jobs` 테이블 모델과 상태 상수. 빌드할 때 두 이미지에 모두 복사됩니다.

**.env는 커밋 금지.** LLM·외부 API 키는 be·agents 환경변수로만. 프런트 코드에 키가 들어가면 누구나 볼 수 있습니다.

예시 폴더 구조 ②

# backend / agents / shared 상세

```
backend/                    # {팀}-be
├── app/
│   ├── main.py             # 앱 생성·라우터 등록
│   ├── core/
│   │   ├── security.py     # JWT 검증·권한
│   │   └── celery_client.py # 태스크 "보내기" 전용
│   ├── api/v1/
│   │   ├── jobs.py         # POST/GET /jobs, SSE
│   │   ├── approvals.py    # 위험 작업 승인
│   │   └── health.py       # /health
│   ├── schemas/job.py      # Pydantic 요청·응답
│   └── services/job_service.py
└── requirements.txt        # fastapi, celery, redis

shared/                     # 두 이미지 공용
├── config.py               # Settings(환경변수)
├── db.py                   # 엔진·세션
└── models.py               # jobs 테이블·상태 상수
```

```
agents/                     # {팀}-agents
├── worker/
│   ├── celery_app.py       # Celery 앱·설정
│   ├── tasks.py            # run_agent_job 태스크
│   ├── job_store.py        # jobs 상태 변경
│   └── events.py           # 진행 이벤트 발행
├── graph/                  # LangGraph 멀티에이전트
│   ├── build.py            # build_graph()
│   ├── state.py            # 공유 State
│   ├── supervisor.py
│   ├── nodes/              # 팀 주제에 맞게
│   │   └── diagnosis.py, recovery.py …
│   └── prompts/
├── integrations/           # 바깥 세상과의 경계
│   ├── base.py             # 타임아웃·재시도 공통
│   ├── llm.py              # LLM 게이트웨이
│   └── public_api.py
├── healing/                # 폴백·정책·주기 점검
└── requirements.txt        # celery, langgraph, httpx
```

가장 중요한 규칙

# be는 agents 코드를 import하지 않는다




### ✅ 이름으로 보내기

be는 `send_task("agents.run_agent_job", ...)`처럼 **문자열 이름**으로 보냅니다. 그래서 be 이미지에는 LangGraph·LLM 라이브러리가 없어도 됩니다.

### ✅ 인자에는 ID만

큰 입력(문서·대화 내용)은 DB에 저장하고 태스크 인자로는 `job_id`만. 재시도·감사 추적이 쉬워지고 브로커가 가벼워집니다.

### ❌ agents → backend

에이전트 노드에서 FastAPI·Request를 import하면 안 됩니다. 에이전트는 HTTP를 몰라야 단독으로 테스트할 수 있습니다.

예시 코드 ① · API 서버 (BE)

# Job 만들고 Celery로 보낸 뒤 바로 202

backend/app/api/v1/jobs.py

```python
from fastapi import APIRouter, Depends, status
from app.core.celery_client import celery_client
from app.core.security import current_user
from app.schemas.job import JobCreate, JobAccepted, JobRead
from app.services import job_service
from shared.db import get_db

router = APIRouter(prefix="/jobs", tags=["jobs"])

@router.post("", status_code=status.HTTP_202_ACCEPTED,
             response_model=JobAccepted)
async def create_job(body: JobCreate, db=Depends(get_db),
                     user=Depends(current_user)):
    job = await job_service.create(db, user.id, body)  # ① DB: queued
    celery_client.send_task("agents.run_agent_job",      # ② 이름으로 전송
                            args=[job.id], queue="agents")
    return JobAccepted(job_id=job.id, status="queued")  # ③ 즉시 응답

@router.get("/{job_id}", response_model=JobRead)
async def get_job(job_id: str, db=Depends(get_db)):
    return await job_service.get(db, job_id)  # 상태·결과 조회
```

backend/app/core/celery_client.py

```python
from celery import Celery
from shared.config import settings

# 보내기 전용 클라이언트
# (태스크 함수는 여기 없음!)
celery_client = Celery(
    "be", broker=settings.redis_url)
```

### 포인트

· `send_task`는 브로커에 메시지만 넣고 즉시 돌아옵니다 (수 ms).
· 태스크 이름 `"agents.run_agent_job"`은 워커 쪽 `@app.task(name=...)`와 **정확히 같아야** 합니다.
· 결과는 Celery가 아니라 **DB의 jobs 테이블**에서 조회합니다.

# 워커: 태스크를 받아 LangGraph를 돌리고 진행 상황을 알린다

agents/worker/tasks.py

```python
from agents.worker.celery_app import app
from agents.worker import job_store as js
from agents.worker.events import publish
from agents.graph.build import build_graph
from agents.integrations.base import TransientError
graph = build_graph()                    # 워커 시작 시 한 번만 조립
@app.task(name="agents.run_agent_job", bind=True, max_retries=3)
def run_agent_job(self, job_id: str):
    if not js.claim(job_id):             # queued→running 선점 실패면
        return                           # 이미 처리 중 → 중복 실행 방지
    try:
        state = {}
        for upd in graph.stream(js.get_input(job_id), stream_mode="updates"):
            node, data = next(iter(upd.items()))
            publish(job_id, {"type": "step", "node": node})
            state.update(data or {})
        js.succeed(job_id, state)
        publish(job_id, {"type": "done"})
    except TransientError as e:          # 잠깐의 장애 → 다시
        if self.request.retries < self.max_retries:
            js.mark_retrying(job_id)
            raise self.retry(exc=e, countdown=10 * (self.request.retries + 1))
        js.fail(job_id, e); raise        # 재시도 소진
    except Exception as e:               # 복구 불가 → 실패 기록
        js.fail(job_id, e); raise        # status=failed + failed 이벤트
```

agents/worker/celery_app.py

```python
from celery import Celery
from shared.config import settings

app = Celery("agents", broker=settings.redis_url,
             include=["agents.worker.tasks"])
app.conf.update(
    task_default_queue="agents",
    task_acks_late=True,      # 끝난 뒤 완료 처리
    task_reject_on_worker_lost=True, # 워커가
                     # 죽으면 다른 워커가 다시 받음
    worker_prefetch_multiplier=1, # 하나씩
    task_soft_time_limit=540,  # 9분: 예외로 경고
    task_time_limit=600,       # 10분: 강제 종료
    task_ignore_result=True,   # 결과는 DB에
)
```

agents/worker/events.py

```python
import json, redis
r = redis.Redis.from_url(settings.redis_url)

def publish(job_id, ev):
    r.publish(f"job:{job_id}",
              json.dumps(ev, ensure_ascii=False))
```

CELERY 실전 메모

# 중복 실행 막기와 워커 실행 방법

agents/worker/job_store.py — 상태 "선점(claim)"

```python
from sqlalchemy import update
from shared.db import SessionLocal
from shared.models import Job

def claim(job_id: str) -> bool:
    """queued/retrying 인 Job만 running으로.
    UPDATE 한 번이라 두 워커가 동시에 와도
    한 명만 성공(rowcount == 1)."""
    with SessionLocal() as s, s.begin():
        res = s.execute(
            update(Job)
            .where(Job.id == job_id,
                   Job.status.in_(["queued", "retrying"]))
            .values(status="running"))
    return res.rowcount == 1
```

acks_late를 켜면 같은 태스크가 두 번 올 수 있습니다 → 이 선점 로직이 "복구가 두 번 실행"을 막아 줍니다.

## Celery 태스크는 동기 함수

워커 쪽 코드(graph.stream, httpx.Client, 동기 DB 세션)는 **동기로 작성**하는 것이 가장 단순합니다. async 코드를 꼭 써야 하면 태스크 안에서 asyncio.run(...)으로 감쌉니다.

## 로컬 개발 실행 (터미널 3개)

```
docker run -p 6379:6379 redis:7-alpine
uvicorn app.main:app --reload            # backend/
celery -A agents.worker.celery_app worker \
       -Q agents -l INFO                 # 루트에서
```

Windows에서는 워커에 --pool=solo를 붙이세요.

**Flower** (localhost:5555)로 워커 수·대기 태스크·실패·재시도를 실시간으로 보여주면 오토힐링 시연에 좋습니다.

예시 코드 ③ · 진행 상황 스트리밍

# SSE: 워커 이벤트를 브라우저까지

backend/app/api/v1/jobs.py (이어서) — 워커가 Redis로 발행한 이벤트를 중계

```python
import json
from fastapi.responses import StreamingResponse

@router.get("/{job_id}/events")
async def job_events(job_id: str, request: Request):
    async def stream():
        ps = request.app.state.redis.pubsub()
        await ps.subscribe(f"job:{job_id}")
        try:
            async for msg in ps.listen():
                if msg["type"] != "message":
                    continue
                yield f"data: {msg['data']}\n\n"  # SSE 형식
                if json.loads(msg["data"])["type"] in ("done", "failed"):
                    break
        finally:
            await ps.unsubscribe()
            await ps.aclose()
    return StreamingResponse(stream(),
                             media_type="text/event-stream")
```

frontend/src/hooks/useJob.ts

```javascript
const res = await fetch("/api/v1/jobs", {
  method: "POST", headers, body });
const { job_id } = await res.json();

const es = new EventSource(
  `/api/v1/jobs/${job_id}/events`);
es.onmessage = (e) => {
  const ev = JSON.parse(e.data);
  if (ev.type === "step") showStep(ev.node);
  if (ev.type === "done" || ev.type === "failed") {
    es.close();
    loadResult(job_id); // GET /jobs/{id}
  }
};
```

**주의 2가지**

· Pub/Sub은 구독 전 이벤트를 저장하지 않음 → 화면 진입 시 GET `/jobs/{id}`로 현재 상태를 먼저 그리기

· EventSource는 Authorization 헤더를 못 붙임 → 쿠키 인증 또는 fetch 기반 SSE 라이브러리 사용

오토힐링

# 오토힐링은 4개 계층에 나눠서 넣는다

"오토힐링 = 에이전트가 알아서 고친다"가 전부가 아닙니다. **아래 계층일수록 단순·확실**하게.

| 계층 | 무엇이 고장나나 | 자동 조치 | 코드 위치 |
|---|---|---|---|
| ① 호출 | 외부 API 타임아웃·5xx·429, LLM 과부하 | 타임아웃 → 지수 백오프 재시도 → **폴백**(백업 API·캐시·다른 모델) | `agents/integrations/` `agents/healing/` |
| ② 에이전트 | LLM 출력 형식 오류, 근거 부족, 잘못된 판단 | 스키마 검증 실패 → 재생성, 자기성찰 노드에서 재진단(최대 N회) → 사람 이관 | `agents/graph/` |
| ③ 작업 | Job 처리 중 예외, 워커 강제 종료 | Celery `self.retry()`, `acks_late`로 워커가 죽어도 재전달, 상태 선점(claim)으로 중복 방지, 최종 실패는 failed | `agents/worker/tasks.py` `celery_app.py` |
| ④ 인프라 | 프로세스 다운, 메모리 초과 | 헬스체크 실패 → 컨테이너 자동 재시작, 워커 여러 개 띄우기 | `docker/compose/compose.yml` `agents/healing/monitor.py` |

평가 포인트: 복구를 **시연**하려면 장애를 일부러 주입하고(예: 외부 API URL을 끊기) 감지 시각·복구 시각을 **로그로 남겨야** 합니다.

예시 코드 ④ · 호출 계층 오토힐링

# 외부 API 래퍼: 타임아웃 → 재시도 → 폴백

agents/integrations/base.py · public_api.py

```python
import httpx
from tenacity import (retry, stop_after_attempt,
    wait_exponential, retry_if_exception_type)

class TransientError(Exception): ...  # "잠깐 기다리면 될 오류"

@retry(stop=stop_after_attempt(3),
       wait=wait_exponential(min=1, max=8),       # 1→2→4초
       retry=retry_if_exception_type(TransientError), reraise=True)
def get_json(client, url, params):
    try:
        r = client.get(url, params=params, timeout=5.0)
    except (httpx.TimeoutException, httpx.NetworkError) as e:
        raise TransientError(str(e)) from e
    if r.status_code == 429 or r.status_code >= 500:
        raise TransientError(f"HTTP {r.status_code}")
    r.raise_for_status()           # 4xx = 내 잘못 → 재시도 X
    return r.json()

def get_crowd(area: str) -> dict:
    with httpx.Client() as c:
        try:
            return get_json(c, PRIMARY_URL, {"area": area})
        except TransientError:
            if cached := cache.get(area):            # 폴백 ① 캐시
                return {**cached, "stale": True}
            return get_json(c, BACKUP_URL, {"area": area})  # ② 백업
```

### 오류를 두 종류로 나누기

**일시적** (타임아웃·5xx·429) → 재시도할 가치 있음
**영구적** (400·401·404) → 재시도해도 똑같음, 즉시 실패

### 재시도는 두 겹

① 여기(tenacity): 몇 초 안의 짧은 재시도
② 그래도 실패 → `TransientError`가 태스크까지 올라가 Celery `self.retry()`로 **수십 초 뒤 전체 재실행**

### 폴백 결과는 표시하기

`"stale": true`처럼 남겨 에이전트·화면이 "최신 아님"을 알게 합니다. LLM도 `integrations/llm.py`(LiteLLM 등)에 모아 모델 폴백을 한 곳에서.

# docker/ 폴더: 빌드와 실행을 한곳에

```
docker/
├── build.sh              # 모든 이미지 빌드 + VERSION 갱신
├── start.sh              # 컴포즈 시작
├── stop.sh               # 컴포즈 종료
├── restart.sh            # 재시작 (stop → start)
├── compose/
│   ├── compose.yml       # 서비스 정의
│   └── .env              # 서비스 환경설정
│                         #  (루트 .env 심볼릭 링크 가능)
└── images/               # 우리 소스로 빌드하는 이미지
    ├── {팀}-fe/          # 프런트엔드
    │   ├── Dockerfile
    │   └── VERSION       # 예: 261119-1650
    ├── {팀}-be/          # 백엔드 API 서버
    │   ├── Dockerfile
    │   └── VERSION
    ├── {팀}-agents/      # Celery 워커 + 에이전트
    │   ├── Dockerfile
    │   └── VERSION
    └── ...               # 필요 시 추가
```

### 폴더 이름 = 이미지 이름

`images/bike-be/` → 이미지 `bike-be:261119-1650`. 팀 프리픽스로 다른 팀·프로젝트 이미지와 섞이지 않습니다.

### images/에 없는 것

Redis·PostgreSQL·Flower처럼 **공식 이미지를 그대로 쓰는 서비스**는 빌드하지 않고 compose.yml에서 바로 가져옵니다.

### VERSION = 빌드 시각

YYMMDD-hhmm 형식으로 build.sh가 자동 기록. 문제가 생기면 이전 시각 태그로 바로 되돌릴 수 있습니다.

**.env 링크:** `ln -s ../../.env docker/compose/.env`
compose는 `compose.yml`과 같은 폴더의 `.env`를 자동으로 읽습니다.

# build.sh와 Dockerfile

docker/build.sh

```
#!/usr/bin/env bash
# 사용법: ./docker/build.sh            (전체)
#        ./docker/build.sh bike-agents (일부만)
set -euo pipefail
cd "$(dirname "$0")/.."            # 프로젝트 루트로 이동

VERSION=$(date +%y%m%d-%H%M)       # 예: 261119-1650
TARGETS=("$@")
if [ ${#TARGETS[@]} -eq 0 ]; then
  TARGETS=($(ls docker/images))    # 인자 없으면 전부
fi

for name in "${TARGETS[@]}"; do
  dir="docker/images/$name"
  echo "▶ $name:$VERSION"
  docker build -f "$dir/Dockerfile" \
    -t "$name:$VERSION" -t "$name:latest" .   # 컨텍스트=루트
  echo "$VERSION" > "$dir/VERSION"  # 성공했을 때만 기록
done
```

빌드 컨텍스트가 **프로젝트 루트**라서 Dockerfile이 `shared/`, `backend/`, `agents/`를 모두 COPY할 수 있습니다.

docker/images/{팀}-be/Dockerfile

```
FROM python:3.12-slim
WORKDIR /srv
COPY backend/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY shared/ ./shared/
COPY backend/app/ ./app/
ENV PYTHONPATH=/srv
CMD ["uvicorn", "app.main:app", \
     "--host", "0.0.0.0", "--port", "8000"]
```

docker/images/{팀}-agents/Dockerfile

```
FROM python:3.12-slim
WORKDIR /srv
COPY agents/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY shared/ ./shared/
COPY agents/ ./agents/
ENV PYTHONPATH=/srv
CMD ["celery", "-A", "agents.worker.celery_app", \
     "worker", "-Q", "agents", "-l", "INFO"]
```

requirements를 먼저 COPY → 소스만 바뀌면 pip 단계는 캐시 재사용 (빌드 빨라짐).

# compose.yml: 서비스 조립

docker/compose/compose.yml

```yaml
services:
  fe:
    image: ${TEAM}-fe:${FE_VERSION:-latest}
    ports: ["80:80"]
  be:
    image: ${TEAM}-be:${BE_VERSION:-latest}
    env_file: .env
    ports: ["8000:8000"]
    depends_on: [redis, db]
    restart: unless-stopped
  agents:
    image: ${TEAM}-agents:${AGENTS_VERSION:-latest}
    env_file: .env
    depends_on: [redis, db]
    restart: unless-stopped      # 죽으면 자동 재시작
    deploy: { replicas: 2 }      # 워커 2개
  agents-beat:                   # 주기 작업 (같은 이미지!)
    image: ${TEAM}-agents:${AGENTS_VERSION:-latest}
    env_file: .env
    command: celery -A agents.worker.celery_app beat -l INFO
  flower:                        # 모니터링 대시보드
    image: mher/flower
    command: celery --broker=redis://redis:6379/0 flower
    ports: ["5555:5555"]
  redis: { image: redis:7-alpine, restart: unless-stopped }
  db:
    image: postgres:16
    env_file: .env
    volumes: [pgdata:/var/lib/postgresql/data]
volumes: { pgdata: {} }
```

docker/compose/.env (예시)

```
COMPOSE_PROJECT_NAME=bike
TEAM=bike
# 롤백할 때만 지정 (비우면 latest)
# AGENTS_VERSION=261118-0930
REDIS_URL=redis://redis:6379/0
POSTGRES_PASSWORD=change-me
DATABASE_URL=postgresql://postgres:change-me@db/app
OPENAI_API_KEY=...
```

- **버전 고정·롤백**: AGENTS_VERSION만 바꾸고 start → 그 시각 이미지로
- **같은 이미지, 다른 command**: agents와 agents-beat
- **주의**: 단독 Docker의 `restart`는 **프로세스가 죽었을 때만** 재시작합니다. 응답 없음(unhealthy) 감지·재시작은 `healing/monitor`가 맡도록 설계하세요.

# start · stop · restart 스크립트와 일상 작업 흐름

docker/start.sh

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")/compose"
docker compose up -d "$@"
docker compose ps
```

docker/stop.sh

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")/compose"
docker compose down
# DB 볼륨은 유지됨
# (down -v 는 DB 삭제!)
```

docker/restart.sh

```
#!/usr/bin/env bash
set -euo pipefail
DIR="$(cd "$(dirname "$0")" && pwd)"
"$DIR/stop.sh"
"$DIR/start.sh"
```

| 상황 | 명령 |
|---|---|
| 처음 띄우기 | `./docker/build.sh && ./docker/start.sh` |
| 에이전트 코드만 고침 | `./docker/build.sh bike-agents && ./docker/start.sh` (바뀐 서비스만 새로 생성) |
| 깨끗하게 다시 시작 | `./docker/restart.sh` |
| 워커 로그 보기 / 워커 늘리기 | `cd docker/compose && docker compose logs -f agents` / `docker compose up -d --scale agents=4` |
| 장애 주입 (오토힐링 시연) | `docker compose exec agents sh -c "kill 1"` → 워커 자동 재시작·태스크 재전달 확인 (Flower로 시연) |

처음 한 번 실행 권한 부여: `chmod +x docker/*.sh`

우리 팀에 대입하기

# 같은 구조, 다른 내용물

구조는 그대로 두고 **Job · 에이전트 · 외부 API · 힐링 포인트**만 팀 주제에 맞게 채웁니다. (아래는 예시)

| | 묶음배달 오토힐링 | 축제 다국어 안심 관광 비서 | 자전거 안전 상담·코스 추천 |
|---|---|---|---|
| **Job 1건** | "이 배달 묶음에 이상 있음 → 진단·재배정" | "외국인 관광객 질문 1건 → 안전 답변 생성" | "출발지·조건 → 코스 추천 + 안전 상담" |
| **에이전트** | 감지 → 진단 → 재배정(복구) → 검증 | 언어·의도 파악 → 혼잡도 조회 → 경로·대피 안내 → 응답 검증 | 상담 → 코스 탐색 → 위험구간 결합 → 검증 |
| **외부 API** | 지도·경로, 기상 | 공공 혼잡도·관광 정보, 번역(LLM) | 자전거도로·사고다발 공공데이터, 기상 |
| **힐링 포인트** | 지연·이탈 감지 시 재배정 / 경로 API 실패 시 거리 추정 폴백 | 공공 API 1차 → 2차 → 캐시(stale 표시) 페일오버 | 기상 API 장애 시 기본 안전 안내 + 재시도 / LLM 모델 폴백 |
| **주기 작업** | N분마다 진행 중 배달 점검 | N분마다 혼잡도 수집·캐시 갱신 | 매일 공공데이터 갱신 |

구현 순서

# 처음부터 LLM 붙이지 말고, 뼈대부터

### 1 가짜 워커

Celery 태스크 안에서 `time.sleep(3)` 세 번 + 이벤트 발행만. **POST → 202 → 폴링 → done**이 돌면 성공.

### 2 상태·DB

jobs 테이블과 상태 머신 (queued/running/succeeded/failed) 완성. 새로고침해도 결과가 보여야 함.

### 3 SSE 화면

폴링을 SSE로 교체. 노드 이름이 화면에 하나씩 찍히는 "사고 과정 타임라인".

### 4 에이전트 연결

sleep 자리에 `graph.stream()`. 에이전트는 노트북/단위 테스트로 먼저 검증한 뒤 붙이기.

### 5 외부 API·힐링

`agents/integrations/` 래퍼에 재시도·폴백. 장애 주입 스위치(환경변수)를 만들어 둔다.

### 6 측정·배포

`docker/build.sh` → `start.sh`로 배포. 장애 주입 → 감지·복구 시간 로그 → Auto-Healing 결과서.

1~3단계는 **LLM 비용 0원**으로 끝납니다. 구조가 먼저 돌아가야 에이전트 문제와 인프라 문제를 구분해서 디버깅할 수 있어요.

점검

# 흔한 실수 체크리스트

☐ API 엔드포인트 안에서 `graph.ainvoke()`를 기다리고 있다

☐ `BackgroundTasks`로 멀티에이전트를 돌린다 (재시작 시 유실)

☐ Celery 태스크 인자로 대화 전문·파일 내용을 통째로 넘긴다 → ID만

☐ Job 상태를 DB에 안 남겨서 새로고침하면 결과가 사라진다

☐ `docker compose down -v`로 DB 볼륨까지 지운다

☐ 모든 오류를 무조건 재시도한다 (400·401도)

☐ 재시도 횟수·타임아웃 상한이 없다 → 무한 루프·비용 폭탄

☐ LLM·외부 API 키가 프런트 코드나 GitHub에 있다

☐ API 서버의 `async def` 안에서 `requests`·`time.sleep` 같은 동기 호출을 한다

☐ 같은 요청이 두 번 들어와 복구가 두 번 실행된다 → 멱등 키

☐ 이미지를 `latest`로만 관리해 이전 버전으로 되돌릴 수 없다

한 줄 요약: **API는 주문만 받고(202), 워커가 요리하고(LangGraph), 진동벨로 알리고(SSE), 납품이 늦으면 다른 길로 간다(오토힐링).**