AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

PR 자동 리뷰어 에이전트
90분 안에 붙이는 라인별 자동 리뷰

Session 10 · 라인별 PR 자동 리뷰어 에이전트

Python 300줄 · GitHub REST · Anthropic Messages API · GitHub Actions · 최소권한

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

PART 1

왜 PR 자동 리뷰어인가

> 세션 목표 · 리뷰 병목 · AI 페어 리뷰의 강점·한계 · 사람과의 분업 설계.

출처: GitHub API | Anthropic API | PR 자동화

세션 목표 — *열린 PR 에 AI 가 먼저 붙는다*

이 세션이 끝나면 참여자 리포에 세 가지가 붙어 있다.

🐍 reviewer.py

파이썬 300줄 남짓. `github_api` · `diff` · `llm` · `prompts` · `__main__` 5개 모듈.

⚙️ ai-review.yml

`.github/workflows/` 아래. PR 이 열리면 자동으로 워크플로가 돈다.

💬 인라인 코멘트

열린 PR 에 라인별 리뷰가 실제로 달린다. 사람이 열기 전에 AI 가 먼저 훑는다.

📌 **강조점 — 완제품이 아니다.** 완제품 카테고리는 두 갈래 — **① 완제품 GitHub Action (예: `openai/codex-action`)**, **② SaaS 리뷰 봇 (예: CodeRabbit · Ellipsis · Greptile)**. 이 세션은 둘 다 붙이지 않는다. **자기 조직 규약에 맞춘 리뷰어를 자기 손으로 만든다** 는 것이 이 세션의 정체성. 그래야 리뷰 문구도, 무시할 파일 패턴도, false positive 억제 규칙도 팀 컨텍스트에 맞춘다.

출처: GitHub API | Anthropic API | PR 자동화

리뷰가 병목이 되는 세 지점

개발 팀에서 코드 리뷰가 병목이 되는 지점은 대체로 세 곳이다.

⏳ 시니어 시간이 유한

팀에 시니어가 두 명이면 하루에 소화 가능한 PR 수가 정해져 있다. 그 이상 열리면 큐가 쌓인다.

⚖️ PR 크기 편중

100줄 이하 PR 은 5분 만에 리뷰가 붙는다. 800줄 넘는 PR 은 아무도 손대지 않는다. 이 편중이 릴리스 리듬을 무너뜨린다.

🔁 반복 지적의 피로

'함수 이름 스네이크로' · 'except: 로 다 잡지 마세요' · 'print 대신 logger' — 같은 지적을 같은 시니어가 매주 다른 PR 에.

📌 AI 자동 리뷰어는 이 셋 중 셋째 문제에 정확히 맞물린다. **반복 지적을 위임한다.** 시니어는 아키텍처 · 트레이드오프 · 팀 컨벤션 위배 같은 판단이 필요한 지점만 본다.

출처: GitHub API | Anthropic API | PR 자동화

AI 리뷰어 — *강점과 한계 5축*

축	강점	한계
속도	300줄 diff 를 30초 안에 훑음	대용량 PR 은 분할·재조립 필요
일관성	같은 룰을 지치지 않고 매번 적용	룰이 애매하면 매번 다른 지적
커버리지	테스트·yaml·마이그레이션도 훑음	리포 전체 맥락 · 파일 밖 결정 이유 모름
비용	Haiku 급으로 PR 하나에 수백 원	대형 리포 다PR 이면 월 수십만원 가능
판단	mechanical (스타일·명명·미사용) 강함	architectural (모듈 분리·트레이드오프) 약함

📌 **핵심 관찰.** AI 는 **mechanical review**, 사람 시니어는 **architectural review**. 이 분업이 세팅되면 팀의 리뷰 큐가 반으로 준다. 세팅이 안 되면 사람이 AI 지적을 다시 리뷰해야 해서 오히려 늘어난다.

출처: GitHub API | Anthropic API | PR 자동화

분업 — *AI 는 라인, 사람은 PR*

분업이 코드로 굳어지려면 규약이 리포 안에 명시돼야 한다. 이 세션에서는 두 원칙만 못박는다.

🤖 AI 담당

- **라인 단위 인라인 코멘트**
- 스타일 · 명명 · 미사용 변수
- print 대신 logger · bare except
- 하드코딩 시크릿 흔적

👩‍💻 사람 담당

- **PR 상단 총평** (approve · request changes)
- 아키텍처 · 트레이드오프
- 팀 컨벤션 위배 판단
- **승인 권한은 오직 사람**

⚠️ **AI 가 approve 하지 않는다.** 승인 권한은 사람만. GitHub 브랜치 보호 규칙 (`Require approvals`) 으로 강제. 이 두 원칙이 세팅돼야 AI 가 지적을 놓쳐도, 잘못 짚어도, 최종 안전망이 사람에게 남는다.

PART 2

아키텍처 — *5단계 파이프라인*

> 파이프라인 5단계 · GitHub Actions 트리거 · diff 파싱 · 청크 분할 · 인라인 코멘트 API · Anthropic 프롬프트 설계.

출처: GitHub API | Anthropic API | PR 자동화

리뷰어 5단계 — *함수 구성과 1:1 대응*

단계	입력	출력	담당
1. 트리거	`pull_request` 이벤트	워크플로 시작	GitHub Actions
2. diff 수집	PR 번호	파일별 patch	`GET /repos/{o}/{r}/pulls/{n}/files`
3. 필터·분할	파일별 patch	리뷰할 청크 목록	리뷰어 파이썬 코드
4. LLM 호출	청크	코멘트 후보 (JSON)	Anthropic Messages API
5. 인라인 코멘트	코멘트 후보	PR 리뷰 게시	`POST /repos/{o}/{r}/pulls/{n}/reviews`

📌 이 다섯 단계 사이에 **실패해도 되는 단계 (필터·분할) 와 실패하면 안 되는 단계 (인라인 코멘트)** 가 있다. 실패해도 되는 곳은 조용히 스킵. 실패하면 안 되는 곳은 재시도 + 예외 로그.

출처: GitHub API | Anthropic API | PR 자동화

리뷰어 5단계 — 함수 구성과 1:1 대응

출처: GitHub API | Anthropic API | PR 자동화

GitHub Actions 트리거 — *무엇을 잡을까*

`pull_request` 이벤트 안에도 세부 액션 타입이 여러 개다. 다 잡으면 워크플로가 과도하게 돈다.

✅ 잡을 세 타입

- `opened` — PR 이 처음 열렸을 때. 초기 리뷰.
- `synchronize` — PR 브랜치에 커밋이 추가됐을 때. 재리뷰.
- `ready_for_review` — Draft 였다가 Ready 로 승격됐을 때.

❌ 안 잡을 것

- `edited` · `labeled` · `assigned` — 코드 변경 없음
- **Draft PR** — `if: draft == false` 로 스킵
- 문서·이미지·LICENSE — `paths` 필터로 제외

📌 참여자가 실험 중인 Draft PR 에 AI 가 계속 코멘트를 달면 노이즈다. 워크플로 조건절에서 아예 걸러낸다. `paths` 필터는 워크플로 파일 안에서 확장자별로 미리 잡는다.

출처: GitHub API | Anthropic API | PR 자동화

diff 파싱 — *GitHub REST 응답 모양*

GitHub REST API 는 PR 의 파일별 변경을 이렇게 반환한다.

```
[
  {
    "filename": "backend/app/main.py",
    "status": "modified",
    "additions": 12,
    "deletions": 3,
    "patch": "@@ -10,3 +10,12 @@ def handler():\n ..."
  }
]
```

📄 patch 필드

unified diff 문자열. `@@ -a,b +c,d @@` 헝크 헤더로 시작.

- `+` 로 시작하는 줄 — 추가된 줄
- `-` 로 시작하는 줄 — 삭제된 줄
- 나머지 — 컨텍스트

🚫 무시해야 할 세 부류

- `status == 'removed'` — 파일 통째로 삭제
- `patch is None` — 바이너리 (이미지·PDF)
- `changes > MAX_PATCH_LINES (=3000)` — 자동 생성물 (lockfile · migration snapshot)

출처: GitHub API | Anthropic API | PR 자동화

청크 분할 — *파일 단위 우선, 크면 헝크*

파일 하나의 patch 가 500줄이라면 그대로 LLM 에 넣기에는 너무 크다. 청크로 자른다.

✂️ 자르는 두 기준

- **파일 단위 우선** — 200줄 안이면 통째로 한 청크
- **헝크 단위 분할** — 크면 `@@` 헝크 하나가 곧 한 청크

🧮 컨텍스트 창 계산

- Sonnet 5 컨텍스트 200K 지만 다 쓰면 비용 폭발
- **patch 3000 라인 초과 시 파일 통째 스킵** (`MAX_PATCH_LINES = 3000`, Part 3 코드)
- 토큰 기반 재분할은 이 세션 범위 밖 — 라인 상한 하나로 컷

📎 청크마다 붙일 3가지

patch 만 던지면 LLM 이 파일 나머지를 알 방법이 없다. 최소 3가지는 함께.

- **파일 경로** (`backend/app/services/rss.py`)
- **파일 언어** (`python`)
- **PR 제목·본문** (전체 PR 의 의도)

📌 컨텍스트를 이 세 조각으로 좁혀야 프롬프트가 안정된다. 파일 전체를 붙이면 오히려 노이즈. 필요한 만큼만.

출처: GitHub API | Anthropic API | PR 자동화

인라인 코멘트 — *두 API 중 두 번째만 사용*

GitHub REST API 는 PR 코멘트를 게시하는 두 가지 경로를 제공한다. 이 세션에서 쓰는 것은 두 번째.

① Issues Comments (버림)

`POST /repos/{o}/{r}/issues/{n}/comments`

- PR 하단에 붙는 총평
- 라인 위치 못 지정
- 이 세션에서는 안 씀

② Reviews (사용)

`POST /repos/{o}/{r}/pulls/{n}/reviews`

- 리뷰 하나에 라인별 코멘트 여러 개 묶음
- PR 상단에 리뷰 카드 하나 뜸
- 각 라인 코멘트가 접혀 있음

📌 하나의 리뷰 요청에 여러 라인 코멘트를 묶어 한 번에 보낸다. 개별 코멘트 API 를 여러 번 부르는 것보다 UX 가 훨씬 낫다.

출처: GitHub API | Anthropic API | PR 자동화

리뷰 게시 요청 — *핵심 필드 3개*

```
{
  "commit_id": "abc123...",
  "event": "COMMENT",
  "body": "AI Reviewer 총평",
  "comments": [
    {
      "path": "backend/app/main.py",
      "line": 42,
      "side": "RIGHT",
      "body": "print 대신 logger.info 를 권장합니다."
    }
  ]
}
```

🎯 `event: 'COMMENT'`

approve · request changes 는 **절대 사용 안 함**. 사람 리뷰어의 권한.

📍 `path` · `line` · `side`

코멘트가 붙을 위치. `side: 'RIGHT'` 는 diff 오른쪽 (추가된 줄) 을 가리킨다.

🔗 `commit_id`

리뷰가 어느 커밋에 붙는지. PR 의 최신 head SHA. 커밋이 밀리면 리뷰가 stale 표시.

📌 세 필드 조합이 라인별 인라인 코멘트를 GitHub 이 정확히 표시하게 만드는 열쇠. 하나만 어긋나도 API 가 `422 Unprocessable Entity` 를 뱉는다.

출처: GitHub API | Anthropic API | PR 자동화

LLM 프롬프트 설계 — *세 축을 지킨다*

LLM 호출부는 프롬프트가 반이다. 시스템 프롬프트 카드 하나가 이 리뷰어의 정체성이 된다.

🎭 역할 (Role)

'너는 시니어 파이썬 리뷰어' · '너는 우리 조직 스타일 가이드를 아는 리뷰어'.

정체성이 흐리면 응답도 흐릿하다.

📐 명시성 (Explicit)

지적할 카테고리를 열거.

'명명 · 미사용 변수 · except 남용 · print 대신 logger · 마법 상수' 처럼.

📦 출력 스키마 (JSON)

CO-STAR 의 R (Response). **JSON 배열 강제**, 자유 서술 금지.

파이썬이 파싱해 인라인 코멘트로 옮길 수 있어야 한다.

※ CO-STAR — Context / Objective / Style / Tone / Audience / Response.

출력 스키마 예시.

```
[
  {
    "line": 42,
    "severity": "warning",
    "category": "logging",
    "message": "print 대신 logger.info 를 권장합니다."
  }
]
```

⚠️ **자유 텍스트 응답은 파싱 실패 확률이 높다.** JSON 스키마를 프롬프트 안에서 못박고, 코드에서도 방어 파싱 (앞뒤 대괄호 잘라내기) 을 둔다.

PART 3

Python 구현 — *5개 모듈*

> 프로젝트 구성 · `github_api.py` · `diff.py` · `llm.py` · `prompts.py` · `__main__.py` · pytest 단위 테스트.

출처: GitHub API | Anthropic API | PR 자동화

프로젝트 트리 — *demos/pr_reviewer/*

```
pr_reviewer/
├─ reviewer/
│   ├─ __init__.py
│   ├─ __main__.py          # python -m reviewer 진입점
│   ├─ github_api.py        # PR 조회·리뷰 게시
│   ├─ diff.py              # patch 파싱·청크 분할
│   ├─ llm.py               # Anthropic 호출
│   └─ prompts.py           # SYSTEM_PROMPT 등 상수
├─ tests/
│   ├─ test_diff.py
│   └─ fixtures/
│       └─ sample_patch.diff
├─ pyproject.toml
├─ .github/
│   └─ workflows/
│       └─ ai-review.yml
└─ README.md
```

📌 이 트리가 이 세션의 목적지. 100% 는 아니어도 90% 는 여기 도달해야 한다. 강사는 사전에 뼈대 리포를 준비 — 초기 상태는 빈 셸 + `pyproject.toml`.

출처: GitHub API | Anthropic API | PR 자동화

`github_api.py` ① — *`GitHubClient`* 스캐폴드

GitHub API 호출은 `httpx` 로 직접 부른다. `PyGithub` 도 좋지만 리뷰 API 는 파라미터가 특수해서 직접 부르는 편이 명확하다.

```
import os
from typing import Any

import httpx

GITHUB_API = "https://api.github.com"


class GitHubClient:
    def __init__(self, token: str, repo: str) → None:
        self.token = token
        self.repo = repo  # "owner/name" 형태
        self.headers = {
            "Authorization": f"Bearer {token}",
            "Accept": "application/vnd.github+json",
            "X-GitHub-Api-Version": "2022-11-28",
        }

    async def get_pr(self, pr_number: int) → dict[str, Any]:
        url = f"{GITHUB_API}/repos/{self.repo}/pulls/{pr_number}"
        async with httpx.AsyncClient() as c:
            r = await c.get(url, headers=self.headers, timeout=30)
```

📌 `X-GitHub-Api-Version` **헤더.** GitHub 이 REST v3 를 안정화하면서 이 헤더로 버전 고정을 권장. `Bearer` 토큰 — GitHub Actions 에서는 `${{ secrets.GITHUB_TOKEN }}` 이 자동 주입.

출처: GitHub API | Anthropic API | PR 자동화

github_api.py ② — list_pr_files 페이지네이션

```
async def list_pr_files(self, pr_number: int) → list[dict[str, Any]]:
    url = f"{GITHUB_API}/repos/{self.repo}/pulls/{pr_number}/files"
    files: list[dict[str, Any]] = []
    async with httpx.AsyncClient() as c:
        page = 1
        while True:
            r = await c.get(
                url,
                params={"per_page": 100, "page": page},
                headers=self.headers,
                timeout=30,
            )
            r.raise_for_status()
            batch = r.json()
            if not batch:
                break
            files.extend(batch)
            if len(batch) < 100:
                break
            page += 1
    return files
```

⚠️ **페이지네이션 필수.** PR 파일이 100개 넘어가는 것이 드물지 않다 (마이그레이션 · 리팩토링). `per_page=100` 에 페이지 루프 없이 첫 100개만 보고 끝내면 뒤쪽 파일이 조용히 리뷰 대상에서 빠진다.

출처: GitHub API | Anthropic API | PR 자동화

github_api.py ③ — post_review 리뷰 게시

```
    async def post_review(
        self,
        pr_number: int,
        commit_sha: str,
        body: str,
        comments: list[dict[str, Any]],
    ) → dict[str, Any]:
        url = f"{GITHUB_API}/repos/{self.repo}/pulls/{pr_number}/reviews"
        payload = {
            "commit_id": commit_sha,
            "event": "COMMENT",
            "body": body,
            "comments": comments,
        }
        async with httpx.AsyncClient() as c:
            r = await c.post(url, json=payload, headers=self.headers, timeout=60)
            r.raise_for_status()
            return r.json()
```

📌 **한 번의 POST 로 리뷰 카드 + 라인별 코멘트 여러 개.** 이 API 하나가 이 세션의 최종 산출 지점. 앞서 다이어그램에서 강조한 '실패하면 안 되는 단계 5' 가 여기.

출처: GitHub API | Anthropic API | PR 자동화

`diff.py` ① — 상수와 `should_review`

patch 파싱은 정규식 한두 개면 충분하다. 라이브러리 안 쓰고 직접 짜는 편이 이해가 쉽다.

```
import re
from dataclasses import dataclass, field

ALLOWED_EXT = {".py", ".ts", ".tsx", ".js", ".go", ".java", ".kt", ".rb", ".rs"}
MAX_PATCH_LINES = 3000
HUNK_HEADER = re.compile(r"^@@ -(\d+)(?:,\d+)? \+(\d+)(?:,\d+)? @@")

@dataclass
class Chunk:
    path: str
    language: str
    patch: str
    line_map: dict[int, int] = field(default_factory=dict)


def should_review(file_entry: dict) → bool:
    if file_entry.get("status") == "removed":
        return False
    if file_entry.get("patch") is None:
        return False
    if file_entry.get("changes", 0) > MAX_PATCH_LINES:
        return False
    path = file_entry["filename"]
```

📌 **허용 확장자.** 조직 스택에 맞춰 편집. 파이썬만 쓰는 팀이면 `.py` 하나만. 프론트 팀이면 `.ts` · `.tsx` 만. 넓게 잡을수록 API 비용이 는다.

diff.py ② — build_line_map (안전핀)

```
def build_line_map(patch: str) → dict[int, int]:
    """patch 안 상대 라인 -> 대상 파일의 실제 라인 번호."""
    line_map: dict[int, int] = {}
    file_line = 0
    for idx, raw in enumerate(patch.splitlines(), start=1):
        m = HUNK_HEADER.match(raw)
        if m:
            file_line = int(m.group(2))
            continue
        if raw.startswith("+") and not raw.startswith("+++"):
            line_map[idx] = file_line
            file_line += 1
        elif raw.startswith("-") and not raw.startswith("---"):
            continue
        else:
            # 공통 컨텍스트 라인
            file_line += 1
    return line_map
```

⚠️ **이 함수가 리뷰어의 안전핀.** LLM 은 'patch 몇 번째 줄에 문제가 있다' 는 식으로 답한다. 그 상대 라인을 파일의 실제 라인 번호로 옮겨야 인라인 코멘트가 정확한 위치에 붙는다. 이 매핑이 없으면 코멘트가 엉뚱한 줄에 뜬다.

`diff.py` ③ — *`language_of`* · *`to_chunks`*

```
def language_of(path: str) → str:
    return {
        ".py": "python",
        ".ts": "typescript",
        ".tsx": "typescript",
        ".js": "javascript",
        ".go": "go",
        ".java": "java",
        ".kt": "kotlin",
        ".rb": "ruby",
        ".rs": "rust",
    }.get("." + path.rsplit(".", 1)[-1], "text")

def to_chunks(files: list[dict]) → list[Chunk]:
    chunks: list[Chunk] = []
    for f in files:
        if not should_review(f):
            continue
        patch = f["patch"]
        chunks.append(
            Chunk(
                path=f["filename"],
                language=language_of(f["filename"]),
                patch=patch,
                line_map=build_line_map(patch),
            )
```

📌 `language_of` 는 LLM 에게 파일 언어를 알려주기 위한 것. Python diff 인지 Go diff 인지에 따라 지적 카테고리가 달라진다. 확장자별 매핑 한 dict 로 처리.

출처: GitHub API | Anthropic API | PR 자동화

prompts.py ① — *SYSTEM_PROMPT (프롬프트 카드)*

```
SYSTEM_PROMPT = """너는 시니어 코드 리뷰어다.
주어진 diff 를 읽고, 개선이 필요한 라인만 골라 JSON 배열로 반환한다.

리뷰 원칙.
1. 추가된 줄 (patch 에서 + 로 시작) 만 지적한다. 삭제·컨텍스트 라인은 무시.
2. 다음 카테고리에 해당하는 것만 남긴다.
   - naming (변수·함수·클래스 명명이 컨벤션과 어긋남)
   - error_handling (except 남용, silent failure, 광범위 catch)
   - logging (print 대신 logger 권장, 로그 레벨 오용)
   - security (하드코딩 시크릿, SQL 인젝션 여지, 안전하지 않은 역직렬화)
   - performance (명백한 O(n^2) 반복, 불필요 재계산)
   - readability (5줄 이상 중첩, 매직 상수, 이름과 실제 동작 불일치)
3. 스타일 취향은 지적하지 않는다 (탭 vs 스페이스, 함수 순서, 컴마 위치).
4. 확실하지 않으면 지적하지 않는다. False positive 가 미검출보다 나쁘다.

출력 스키마.
반드시 JSON 배열 하나만 반환. 배열 밖 텍스트 금지.
각 원소는 다음 필드.

{
  "line": <patch 안 상대 라인 번호 (1부터)>,
  "severity": "info" | "warning" | "critical",
  "category": "naming" | "error_handling" | "logging" | "security" | "performance" | "readability",
  "message": "한국어로 2문장 이내. 어떻게 고칠지 짧게 제안 포함."
}

지적할 것이 없으면 빈 배열 [] 반환."""
```

프롬프트 — *세 장치의 효과*

📏 범위 명시

'추가된 줄만' 을 원칙 1번으로 못박음.

안 그러면 삭제 줄까지 지적함. 이 한 줄이 노이즈의 반을 잡는다.

📋 카테고리 화이트리스트

6개만 열거. '이거 아니면 지적 금지' 프레임.

카테고리 밖 지적을 자연스럽게 억제.

🤫 불확실하면 침묵

'확실하지 않으면 지적하지 않는다' 명시.

이 한 줄이 false positive 를 **눈에 띄게 줄인다** — 강사 경험치로는 30~40%, 팀·모델·리포마다 편차 큼.

USER_TEMPLATE (청크마다 채워 보냄).

```
USER_TEMPLATE = """파일: {path}
언어: {language}
PR 제목: {pr_title}
PR 본문(발췌): {pr_body}

--- diff ---
{patch}
--- end ---

위 diff 를 리뷰해줘."""
```

📌 시스템 프롬프트는 정체성 · 원칙 · 스키마. 유저 메시지는 이번 청크의 컨텍스트. 이 분리가 매 청크마다 프롬프트 재구성 부담을 줄인다.

출처: GitHub API | Anthropic API | PR 자동화

`llm.py` ① — `ReviewComment` 데이터클래스 · 상수

```
import asyncio
import json
import logging
from dataclasses import dataclass

from anthropic import APIError, AsyncAnthropic

from .diff import Chunk
from .prompts import SYSTEM_PROMPT, USER_TEMPLATE

logger = logging.getLogger(__name__)

MODEL = "claude-sonnet-5"
MAX_TOKENS = 2048


@dataclass
class ReviewComment:
    path: str
    line: int
    severity: str
    category: str
    message: str
```

📌 `ReviewComment` 데이터클래스가 파이썬 안의 표준 형태. LLM JSON → `ReviewComment` → API payload 로 이어지는 사슬. `MODEL` 상수는 `claude-sonnet-5` 로 고정 — 팀 정책 바뀌면 이 한 줄만 갈아끼운다.

출처: GitHub API | Anthropic API | PR 자동화

llm.py ② — Reviewer 클래스 · 프롬프트 조립

```
class Reviewer:
    def __init__(self, api_key: str) → None:
        self.client = AsyncAnthropic(api_key=api_key)

    async def review_chunk(
        self, chunk: Chunk, pr_title: str, pr_body: str
    ) → list[ReviewComment]:
        user_msg = USER_TEMPLATE.format(
            path=chunk.path,
            language=chunk.language,
            pr_title=pr_title,
            pr_body=(pr_body or "")[:800],
            patch=chunk.patch,
        )
        # 다음 슬라이드에서 재시도·파싱
```

📌 `pr_body[:800]` — 너무 긴 PR 본문은 토큰 낭비. 800자 컷은 실전에서 요약 없이 그대로 넣어도 리뷰 품질이 유지되는 상한선.

출처: GitHub API | Anthropic API | PR 자동화

`llm.py` ③ — *3회 재시도 + 방어 파싱*

```
        for attempt in range(3):
            try:
                resp = await self.client.messages.create(
                    model=MODEL,
                    max_tokens=MAX_TOKENS,
                    system=SYSTEM_PROMPT,
                    messages=[{"role": "user", "content": user_msg}],
                )
                text = resp.content[0].text.strip()
                items = json.loads(_extract_json_array(text))
                return _to_comments(chunk, items)
            except (APIError, json.JSONDecodeError) as exc:
                logger.warning(
                    "chunk %s attempt %d failed: %s", chunk.path, attempt + 1, exc
                )
                await asyncio.sleep(2**attempt)

        logger.error("chunk %s: giving up after 3 attempts", chunk.path)
        return []


def _extract_json_array(text: str) → str:
    """LLM 이 앞뒤에 코드펜스를 붙여도 JSON 배열만 뽑아냄."""
    start = text.find("[")
    end = text.rfind("]")
    if start == -1 or end == -1:
```

⚠️ `_extract_json_array`. LLM 이 시스템 프롬프트에 'JSON 만' 이라고 못박아도 가끔 코드펜스를 붙인다. 앞뒤 대괄호로 잘라내는 방어 코드가 실전에서 반드시 필요.

출처: GitHub API | Anthropic API | PR 자동화

`llm.py` ④ — `_to_comments` line_map 필터

```
def _to_comments(chunk: Chunk, items: list[dict]) → list[ReviewComment]:
    out: list[ReviewComment] = []
    for it in items:
        rel_line = int(it.get("line", 0))
        real_line = chunk.line_map.get(rel_line)
        if real_line is None:
            # LLM 이 삭제 라인이나 컨텍스트 라인 지적 시 무시
            continue
        out.append(
            ReviewComment(
                path=chunk.path,
                line=real_line,
                severity=str(it.get("severity", "info")),
                category=str(it.get("category", "readability")),
                message=str(it.get("message", "")).strip(),
            )
        )
    return out
```

📌 **line_map 필터가 인라인 코멘트 실패의 90% 를 잡는다.** LLM 이 삭제 라인이나 컨텍스트 라인을 지적하면 `line_map.get(rel_line)` 이 None. 조용히 스킵. 이 필터 없이 GitHub 리뷰 API 를 부르면 `422` 로 리뷰 전체가 실패.

출처: GitHub API | Anthropic API | PR 자동화

`__main__.py` ① — *환경변수 · 조립*

지금까지 만든 세 조각 (github_api · diff · llm) 을 한 진입점에서 조립.

```
import asyncio, logging, os, sys

from .diff import to_chunks
from .github_api import GitHubClient
from .llm import ReviewComment, Reviewer

logger = logging.getLogger("reviewer")


async def main() → int:
    token = os.environ["GITHUB_TOKEN"]
    api_key = os.environ["ANTHROPIC_API_KEY"]
    repo = os.environ["GITHUB_REPOSITORY"]  # "owner/name"
    pr_number = int(os.environ["PR_NUMBER"])

    gh = GitHubClient(token=token, repo=repo)
    reviewer = Reviewer(api_key=api_key)

    pr = await gh.get_pr(pr_number)
    files = await gh.list_pr_files(pr_number)
    chunks = to_chunks(files)
```

📌 환경변수 4개 — `GITHUB_TOKEN` · `ANTHROPIC_API_KEY` · `GITHUB_REPOSITORY` · `PR_NUMBER`. `GITHUB_REPOSITORY` 는 Actions 가 자동 주입 (`owner/name` 형태).

출처: GitHub API | Anthropic API | PR 자동화

`__main__.py` ② — `Semaphore` + `gather` 병렬 리뷰

```
    # 청크별 병렬 리뷰 (동시성 4)
    sem = asyncio.Semaphore(4)

    async def run(chunk):
        async with sem:
            return await reviewer.review_chunk(
                chunk, pr_title=pr["title"], pr_body=pr.get("body") or ""
            )

    results = await asyncio.gather(*(run(c) for c in chunks))
    comments: list[ReviewComment] = [c for group in results for c in group]

    logger.info("total comments: %d", len(comments))

    if not comments:
        logger.info("nothing to review")
        return 0
```

📌 **동시성 4 의 이유.** 청크가 20개면 순차 호출 시 60초 넘게 걸린다. 병렬 4개면 15초. Anthropic API 티어별 rate limit 을 감안한 안전 수준.

출처: GitHub API | Anthropic API | PR 자동화

`__main__.py` ③ — `post_review` 게시 · 엔트리

```
    api_comments = [
        {"path": c.path, "line": c.line, "side": "RIGHT",
         "body": f"**[{c.category}] {c.severity}** {c.message}"}
        for c in comments
    ]
    body = (
        f"AI Reviewer 가 diff 를 훑고 {len(comments)}개의 라인 코멘트를 남겼습니다. "
        "판단이 필요한 최종 승인은 사람 리뷰어에게 맡깁니다."
    )
    await gh.post_review(
        pr_number=pr_number,
        commit_sha=pr["head"]["sha"],
        body=body,
        comments=api_comments,
    )
    return 0


if __name__ == "__main__":
    sys.exit(asyncio.run(main()))
```

📌 `side: RIGHT` 은 새 코드 (`+` 라인) 에 코멘트를 붙이라는 뜻. `commit_sha` 는 반드시 `pr["head"]["sha"]` — 그 이후 커밋이 밀리면 리뷰가 stale 처리된다.

출처: GitHub API | Anthropic API | PR 자동화

pytest ① — `line_map` 매핑 검증

LLM 호출부는 통합 테스트로 미룬다. `diff` **파서는 단위 테스트가 반드시 필요.** `line_map` 이 어긋나면 인라인 코멘트가 엉뚱한 줄에 붙기 때문.

```
from reviewer.diff import build_line_map, should_review

SAMPLE = """@@ -10,3 +10,6 @@ def handler(req):
     return req
+
+def new_helper(x):
+    print(x)
+    return x + 1"""


def test_build_line_map_maps_added_lines():
    m = build_line_map(SAMPLE)
    # 상대 라인 3 (첫 + 라인) -> 파일 라인 11
    assert m[3] == 11
    # 상대 라인 5 (print 줄) -> 파일 라인 13
    assert m[5] == 13
```

📌 hunk 헤더 `@@ -10,3 +10,6 @@` 를 시작점으로 `+` 라인만 세어 `line_map` 을 만든다. 상대 라인 → 실제 파일 라인 매핑이 이 테스트 하나에서 검증된다.

출처: GitHub API | Anthropic API | PR 자동화

pytest ② — `should_review` 필터 세 케이스

```
def test_should_review_skips_removed():
    assert should_review({"status": "removed", "patch": "x", "filename": "a.py"}) is False


def test_should_review_skips_binary():
    assert should_review({"status": "added", "patch": None, "filename": "a.py"}) is False


def test_should_review_skips_markdown():
    assert (
        should_review(
            {"status": "modified", "patch": "x", "changes": 5, "filename": "README.md"}
        )
        is False
    )
```

📌 이 네 테스트가 통과하면 나머지 파이프라인은 LLM 이 정상 응답을 내는 한 큰 문제가 없다. **파서 하나가 이 시스템의 안전핀.**

PART 4

GitHub Actions 배포 — *workflow yaml · 최소권한*

> workflow yaml 전체 · GITHUB_TOKEN 권한 · 시크릿 관리 · 큰 PR 분할 · 첫 실배포 · 트러블슈팅.

출처: GitHub API | Anthropic API | PR 자동화

`.github/workflows/ai-review.yml` ① — *트리거 · 권한 · 동시성*

```yaml
name: AI PR Reviewer

on:
  pull_request:
    types: [opened, synchronize, ready_for_review]
    paths:
      - "**/*.py"
      - "**/*.ts"
      - "**/*.tsx"
      - "**/*.js"
      - "**/*.go"

permissions:
  contents: read
  pull-requests: write

concurrency:
  group: ai-review-${{ github.event.pull_request.number }}
  cancel-in-progress: true
```

📌 상단 블록에서 트리거 · 최소권한 · 동시성 정책까지 정하고 넘어간다. 이후 `jobs` 블록은 실제 실행 스텝만 담아 가독성 확보.

출처: GitHub API | Anthropic API | PR 자동화

.github/workflows/ai-review.yml ② — *jobs · steps*

```yaml
jobs:
  review:
    if: github.event.pull_request.draft == false
    runs-on: ubuntu-latest
    timeout-minutes: 5
    steps:
      - name: Checkout
        uses: actions/checkout@v5

      - name: Setup Python
        uses: actions/setup-python@v6
        with:
          python-version: "3.12"
          cache: pip

      - name: Install reviewer
        run: pip install -e ./tools/pr_reviewer

      - name: Run AI review
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
          PR_NUMBER: ${{ github.event.pull_request.number }}
        run: python -m reviewer
```

출처: GitHub API | Anthropic API | PR 자동화

workflow yaml — *여섯 핵심 세팅*

세팅	값	왜
`on.pull_request.types`	opened · synchronize · ready_for_review	코드 변경 있는 세 타입만
`on.pull_request.paths`	`**/*.py` · `**/*.ts` 등	README 만 바꾼 PR 은 워크플로 자체가 안 뜸
`permissions`	`contents: read` · `pull-requests: write`	최소권한, 그 이상 주지 않음
`concurrency`	PR 번호 그룹 · cancel-in-progress	연달아 커밋 시 이전 실행 취소
`if: draft == false`	Draft PR 스킵	실험 중 노이즈 방지
`timeout-minutes: 5`	상한 5분	LLM · API 지연 시에도 매달리지 않음

📌 이 여섯이 워크플로의 '안전 프레임'. 하나만 빠져도 실전에서 노이즈 · 낭비 · 권한 사고 중 하나가 난다. 참여자가 자기 리포에 옮길 때 이 여섯을 체크리스트로 삼는다.

출처: GitHub API | Anthropic API | PR 자동화

`GITHUB_TOKEN` — *왜 최소권한*

GitHub Actions 워크플로에는 기본으로 `GITHUB_TOKEN` 이 주입된다. 이 토큰의 스코프는 워크플로 파일의 `permissions` 블록으로 정한다.

✅ 이 리뷰어에 필요한 최소권한

- `contents: read` — 리포 체크아웃, 코드 읽기
- `pull-requests: write` — 리뷰 게시, 코멘트 · 라벨 조작

❌ 주지 말아야 할 것

- `contents: write` — 코드 밀어넣을 이유 없음
- `actions: write` — 다른 워크플로 트리거 불필요
- `id-token: write` — OIDC, 이 세션 범위 밖

⚠️ **포크 PR 의 함정.** 외부 기여자가 포크에서 여는 PR 은 GitHub 이 기본적으로 `GITHUB_TOKEN` 을 read-only 로 강등한다. 보안 조치다. 포크 PR 에도 동작시키려면 `pull_request_target` 이벤트로 바꿔야 하는데, 그러면 포크의 악의적 코드가 리포 시크릿에 접근할 위험이 생긴다. **이 세션 범위에서는 내부 브랜치 PR 만.**

출처: GitHub API | Anthropic API | PR 자동화

시크릿 관리 — *리포 시크릿 · `.env`*

시크릿은 두 개다.

🔑 두 시크릿

- `GITHUB_TOKEN` — Actions 자동 주입, 등록 불필요
- `ANTHROPIC_API_KEY` — 참여자 직접 등록
- **경로**: Settings → Secrets and variables → Actions → New repository secret

📜 세 원칙

- **리포 시크릿에 등록** — PAT 를 코드·이슈에 붙여넣지 않는다
- **Organization 시크릿 승격 가능** — 여러 리포에 붙일 때
- **로컬 테스트는 `.env`** — `.gitignore` 필수

`.gitignore` 라인.

```
.env
.env.local
*.pem
```

⚠️ 참여자 대부분이 시크릿 이름 오타에서 걸린다. `ANTHROPIC_API_KEY` — 대문자 언더스코어 그대로. 오타 시 `KeyError` 로 워크플로 실패.

출처: GitHub API | Anthropic API | PR 자동화

큰 PR — *분할 · 병렬 · 안전판*

PR 하나가 많은 파일로 커지는 경우 (마이그레이션 · 대규모 리팩토링) 를 어떻게 다룰까.

📊 남은 두 문제

- **총 청크 수가 너무 많을 때** — 100개 청크면 병렬 4개로도 25 라운드
- **한 청크가 6000 토큰 넘을 때** — 헝크 단위로 재분할, 그래도 초과면 스킵

🛡️ 상한 안전판

```
# reviewer/diff.py 상단에 함께 정의
MAX_CHUNKS_PER_PR = 30

# __main__.py 에서 청크 목록 조립 직후
if len(chunks) > MAX_CHUNKS_PER_PR:
    skipped = [c.path for c in chunks[MAX_CHUNKS_PER_PR:]]
    chunks = chunks[:MAX_CHUNKS_PER_PR]
    body_prefix = (
        f"[알림] PR 규모가 커서 상위 {MAX_CHUNKS_PER_PR}개 "
        f"파일만 리뷰했습니다."
    )
```

📌 `MAX_CHUNKS_PER_PR = 30` 상한을 두고, 초과 시 상단 코멘트에 '일부 파일은 리뷰 대상에서 제외됐습니다' 를 남긴다. 이 안전판이 있어야 큰 마이그레이션 PR 에서도 워크플로가 5분 안에 끝난다.

출처: GitHub API | Anthropic API | PR 자동화

첫 실배포 — *순서와 확인 포인트*

🚀 참여자 순서 5단계

1. Fork 뜬 샌드박스 리포에 `tools/pr_reviewer/` 복사
2. `.github/workflows/ai-review.yml` 을 루트에 배치
3. Settings → Secrets 에 `ANTHROPIC_API_KEY` 등록
4. 아무 파일 (스멜 심어둔 것) 을 살짝 편집해 PR 을 연다
5. Actions 탭에서 워크플로 확인, 30~60초 후 PR 새로고침

✅ 확인 포인트 3개

- **Actions 로그** — `INFO: PR #N: X files, Y chunks` · `INFO: total comments: Z`
- **PR 페이지** — 상단에 AI Reviewer 리뷰 카드, 각 라인에 인라인 코멘트
- **코멘트 위치** — 지적이 실제 문제 라인에 붙었는지 (`line_map` 검증)

첫 실행 실패 시 자주 나오는 원인.

에러	원인
`HTTPError 403`	`permissions.pull-requests: write` 누락
`HTTPError 422 Unprocessable Entity`	`line` 이 diff 안의 라인이 아님 (line_map 실패)
`KeyError: 'ANTHROPIC_API_KEY'`	시크릿 이름 오타
`HTTPError 401 authentication_error`	Anthropic 키 오타·만료

PART 5

실전 · 확장 — *스멜 시연 · 튜닝 · 로드맵*

> 5종 스멜 실시연 · false positive 억제 3 튜닝 · 확장 아이디어 4개 · 도입 로드맵.

출처: GitHub API | Anthropic API | PR 자동화

5종 스멜 — *샌드박스 리포에 심어둔 것*

강사는 사전에 준비한 `vibecoding-review-sandbox` 리포를 열어 보인다. 파이썬 파일 하나에 다섯 종 코드 스멜이 심어져 있다.

```
import os

DB_PASSWORD = "hunter2"                        # ① security

def process(items):
    for i in range(len(items)):                # ② readability (파이썬 관용구 위배)
        for j in range(len(items)):            # ③ performance (O(n^2))
            if items[i] == items[j]:
                print(items[i])                 # ④ logging (print 대신 logger)

    try:
        do_something()
    except:                                    # ⑤ error_handling (bare except)
        pass
```

📌 **기대 결과.** ① · ③ · ④ · ⑤ 는 잡을 확률이 매우 높다. ② 는 카테고리 분류가 모델에 따라 갈린다. 참여자가 처음 놀라는 지점은 잡는 개수보다 **코멘트 문구의 자연스러움** — 시스템 프롬프트에 '한국어로' 를 못박은 효과.

출처: GitHub API | Anthropic API | PR 자동화

5종 스멜 — *예상 지적 vs 실측 카드*

🎯 예상 지적 (프롬프트 기준)

- ① **security** — `DB_PASSWORD = 'hunter2'` 하드코딩
- ② **readability** — `range(len(items))` 관용구 위배
- ③ **performance** — 이중 for 루프 O(n^2)
- ④ **logging** — `print(items[i])` → `logger.info`
- ⑤ **error_handling** — bare `except:` → 명시적 예외

📊 실측 (예시 실행)

- ① · ④ · ⑤ — **잡음** (critical · warning · warning)
- ③ — 잡음 (warning, category=performance)
- ② — 카테고리를 readability 대신 style 로 분류하는 경우 있음

총평. 잡는 개수는 4~5개, 5회 중 4회는 5개 다 잡음.

📌 강사는 실시간으로 몇 개 잡혔는지 참여자와 함께 센다. 잡히지 않은 스멜이 있으면 시스템 프롬프트를 즉석에서 편집해 다시 돌린다 — **프롬프트 튜닝의 실시간 감각.**

출처: GitHub API | Anthropic API | PR 자동화

false positive 억제 — *3 실전 튜닝*

첫 배포 후 며칠 돌려 보면 false positive 가 반드시 나온다. 실전에서 이걸 줄이는 세 가지.

① 프롬프트 원칙 추가

'확실하지 않으면 지적 금지' 옆에 **'이미 라인 안에서 처리된 사항은 다시 지적하지 않는다'** 추가.

예: 다음 줄에 로그가 있으면 print 지적을 낮춘다.

② 심각도 필터

초기에는 `severity == 'critical'` 만 게시.

팀이 익숙해지면 `warning` 도 포함. `info` 는 로그로만.

③ 화이트리스트 파일 skip

`generated/` · `vendor/` · `node_modules/` 는 workflow `paths` 에서 배제.

paths 로 못 잡는 경우 파이썬에서 두 번째 필터.

심각도별 게시 정책 예시.

```
SEVERITY_ALLOW = {"warning", "critical"}

api_comments = [
    {"path": c.path, "line": c.line, "side": "RIGHT",
     "body": f"**[{c.category}] {c.severity}** {c.message}"}
    for c in comments
    if c.severity in SEVERITY_ALLOW
]
```

📌 이 세 튜닝이 세팅되면 사흘 안에 false positive 가 **절반 이하로 줄어든다.** 튜닝은 프롬프트 · 필터 · paths 세 층에서 동시에.

출처: GitHub API | Anthropic API | PR 자동화

확장 아이디어 4 — *하나만 골라 자기 리포에*

이 리뷰어는 시작점이다. 자기 리포로 가져가 확장할 네 방향.

#	방향	요지
1	**테스트 자동 생성**	PR 에서 추가된 함수를 파악해 pytest 케이스 초안을 별도 코멘트로 제안
2	**릴리스 노트 생성**	`push tag` 이벤트에 붙여 최근 PR 목록을 요약해 릴리스 노트로
3	**회귀 감지**	변경된 함수와 관련 테스트 파일을 함께 읽고, 테스트 커버 안 되는 변경을 표시
4	**규약 위배 알림**	`CLAUDE.md` 나 팀 스타일 가이드를 함께 넣고, 위배 라인만 코멘트

⚠️ **네 개를 다 붙이지 말 것.** 하나만 골라 자기 리포에 실전 배치. 실패도 성공도 다 데이터가 된다. 여럿을 한꺼번에 붙이면 어느 방향이 노이즈의 원인인지 분간이 안 된다.

출처: GitHub API | Anthropic API | PR 자동화

도입 로드맵 — *조용히 시작 · 확신 후 확산*

실전 팀에 이 리뷰어를 붙이는 3단계 로드맵.

📅 1주차 — 조용히 시작

자기 사이드 리포 하나에서 리뷰어만 켠다.

- 코멘트 개수 기록
- false positive 개수 기록
- 팀에 아직 공유 X

📅 2~3주차 — 튜닝

시스템 프롬프트를 팀 컨벤션에 맞춰 편집.

- 심각도 필터로 노이즈 관리
- paths · 화이트리스트 조정
- 프롬프트 카드 커밋으로 이력 관리

📅 4주차 — 팀 공유

팀 리포 하나에 붙여 데모.

- 시니어와 함께 코멘트 품질 검토
- 팀 의견을 `CLAUDE.md` 규약에 반영
- 다른 리포로 확산

⚠️ **첫날부터 팀 메인 리포에 붙이지 않는다.** 팀원이 노이즈에 한 번 실망하면 이후 이 리뷰어를 신뢰하지 않는다. 조용히 시작해 확신이 서면 확산.

PART 6

정리 — *한 문장 · 사이클 5축*

> 세션 소감 한 문장 · 마무리 · 사이클 5축 회수.

출처: GitHub API | Anthropic API | PR 자동화

세션 소감 — *한 문장으로 남긴다*

이 세션이 끝났다. 참여자 리포에 300줄 남짓의 파이썬 · 40줄 남짓의 workflow yaml 이 붙었고, 열린 PR 하나에 AI 가 라인별 코멘트를 실제로 남겼다.

🎯 오늘 실습의 뼈대 한 문장

"AI 는 라인, 사람은 PR. 리뷰어 코드는 팀 규약이 코드로 응고된 형태."

🤖 AI 는 라인

mechanical review — 스타일 · 명명 · 미사용 · print · bare except.

👩‍💻 사람은 PR

architectural review — 모듈 분리 · 트레이드오프 · 승인.

📇 코드로 응고

프롬프트 카드가 리포에 박히면 팀 규약이 매 PR 에 자동 적용.

출처: GitHub API | Anthropic API | PR 자동화

마무리 — *자기 리포로 가져갈 것*

🚀 다음 한 걸음

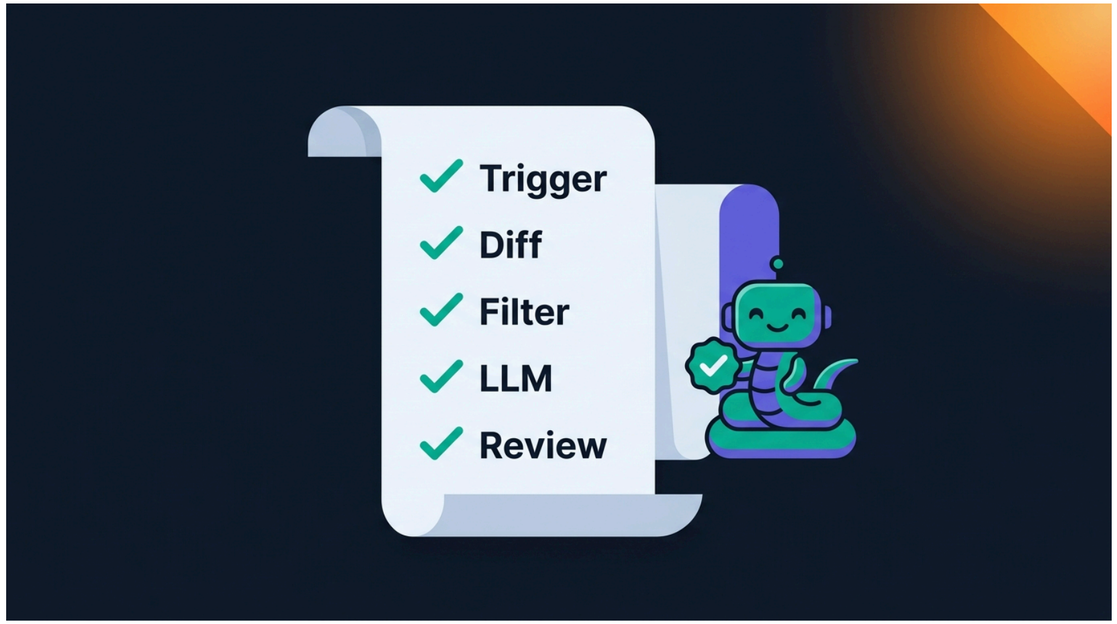
- 확장 아이디어 4 중 **하나만** 선택
- 사이드 리포로 1주 조용히 돌리기
- 팀 규약을 프롬프트 카드에 응고
- 사람 승인 원칙을 브랜치 보호로 강제

📎 다른 스택에도 그대로

- 5단계 파이프라인은 언어 독립
- 파이썬 5모듈 구조 → TS·Go 로 그대로 이식
- workflow yaml 6세팅은 그대로 재사용
- 프롬프트 카드만 조직 컨벤션에 맞춰 편집

출처: GitHub API | Anthropic API | PR 자동화

리뷰어 사이클 5축 — *그림 한 장으로 회수*



✅ 사이클 5축 회수

1. **Trigger** — `pull_request` 세 타입 + Draft 스킵
2. **Diff** — `list_pr_files` 페이지네이션 · `line_map` 매핑
3. **Filter** — `should_review` 로 삭제·바이너리·과대 스킵
4. **LLM** — SYSTEM_PROMPT 카드 · JSON 스키마 · 3회 재시도
5. **Review** — `event: COMMENT` 로 라인별 게시, 승인은 사람

📌 그림의 다섯 축이 이 세션의 뼈대. 코드는 잊어도 다섯 축만 남으면 다음 리뷰어를 자기 손으로 다시 짤 수 있다.