

# hanarang-rails 완전 가이드

---

4자매 AI가 달리는 결정론적 파이프라인 — 처음 보는 사람을 위한 전 구간 해설

나뭇하랑 / hanarang

2026-04-10

0. 이 문서는 누구를 위한 문서인가

1. 한 문단 요약

2. 배경 — 왜 다시 만들었는가

2.1 전임자 hanarang-harness 의 실패 모드

2.2 해결 전략 — 6 가지 설계 원칙

2.3 레일 메타포

3. 4 자매는 누구인가

4. 시스템 구성도

4.1 하이 레벨

4.2 물리 토폴로지

5. 데이터 모델 — MariaDB 스키마

5.1 테이블 요약

5.2 Pipeline 레코드의 생애

5.3 SubTask 트리

6. Orchestrator — XState FSM 엔진

6.1 파일 구조

6.2 상태 흐름

6.3 priorStages 체이닝

7. 역할 계층 — Manager/Principal/Lead/Junior

7.1 왜 계층이 있는가

7.2 역할 정의 ( [src/hierarchy/roles.ts](#) )

7.3 복잡도 스코어 ( [src/hierarchy/complexity.ts](#) )

8. Sister Agent — 레일 위를 달리는 열차

8.1 엔드포인트

8.2 실행 파이프라인 ( [src/spawn.ts](#) )

8.3 LLM 호출 — `openclaw infer model run`

- 8.4 코드 블록 추출 ( [src/code-extractor.ts](#) )
- 9. Git 자동 푸시 — Gitea 연동 ( [git-ops.ts](#) )
  - 9.1 흐름
  - 9.2 프리뷰 URL 추론
  - 9.3 왜 Gitea 인가
- 10. 대시보드 — hanarang-dashboard
  - 10.1 페이지
  - 10.2 SubTask 상세 드로어
  - 10.3 FileViewerModal
- 11. End-to-End 시나리오 — “todo 앱 만들어 줘”
  - 11.1 Step 0 — 트리거
  - 11.2 Step 1 — 오케스트레이터 진입
  - 11.3 Step 2 — Plan (하랑이)
  - 11.4 Step 3 — Implement (나랑이)
  - 11.5 Step 4 — Review (다랑이)
  - 11.6 Step 5 — Deploy (이랑이)
  - 11.7 Step 6 — 완료
- 12. HTTP API 명세
- 13. Sprint Contract — DoD 의 기계 검증
  - 13.1 왜 필요한가
  - 13.2 구조
  - 13.3 체크 타입 ( [src/contract/checks/](#) )
- 14. 보안 모델
  - 14.1 신뢰 경계
  - 14.2 Gitea 프록시 allowlist
  - 14.3 Zod 검증 경계
- 15. 실패/복원력 ( [src/resilience/](#) )
  - 15.1 재시도 정책
  - 15.2 타임아웃
  - 15.3 에스컬레이션
- 16. 설치/실행 가이드
  - 16.1 사전 요구
  - 16.2 rails 서버 구동

16.3 sister-agent 구동 (각 LXC)

16.4 대시보드 구동

16.5 스모크 테스트

17. 디렉토리 구조

18. 로드맵

19. 용어집

20. 참고 자료

부록 A. 용례 비교 — 구 하네스 vs rails

A.1 핸드오프

A.2 DoD

A.3 QA

부록 B. 자주 묻는 질문

부록 C. 라이선스 및 기여

# 0. 이 문서는 누구를 위한 문서인가

---

이 문서는 **hanarang-rails** 프로젝트를 처음 보는 사람이 한 번 읽고 다음 세 가지를 완전히 이해할 수 있게 하는 것이 목표다.

1. 이 시스템이 무엇이고, 왜 만들었으며, 어떤 문제를 해결하는지
2. 코드 한 줄부터 사용자 요청까지 어떤 경로로 흐르는지
3. 직접 클론해서 **E2E**로 돌려보려면 무엇이 필요한지

기존 AI 코딩 도구 (Claude Code, Cursor, Codex, OpenClaw) 를 써 본 경험이 있다면 이해가 빠르겠지만, 없어도 모든 용어는 문서 안에서 정의한다. LLM / 에이전트 / 파이프라인이라는 단어만 대충 알면 된다.

# 1. 한 문단 요약

---

hanarang-rails 는 4 개의 AI “자매” 에이전트가 하나의 요청을 받아서 기획 → 구현 → 리뷰 → 배포를 자동으로 끝내는 결정론적 파이프라인 오케스트레이터다. 기존 하네스는 “이 단계가 끝나면 다음 자매를 호출해 줘” 라고 LLM 에게 부탁하는 방식이었고, 그래서 자매가 중간에 길을 잃으면 사용자가 끼어들어 중재해야 했다. hanarang-rails 는 그 흐름을 XState 유한 상태 기계 (FSM) 와 Sprint Contract (DoD 의 기계 검증본) 로 물리적으로 강제한다. 자매는 “권고”를 받는 것이 아니라 **레일 위를 달리는 열차**처럼, 갈 수 있는 다음 상태가 코드로 고정되어 있다.

## 2. 배경 — 왜 다시 만들었는가

### 2.1 전임자 hanarang-harness 의 실패 모드

이전 프로젝트 [hanarang-harness](#) (Gitea 에 private archive 로 보존) 는 **권고 기반** 파이프라인이었다. 각 단계가 끝나면 LLM 이 “다음에 누구를 부르면 좋을지” 판단했고, 핸드오프는 Discord 멘션으로 전달되었다. 4 개월 운영하면서 다음 6 가지 고질 문제가 반복됐다.

| 코드 | 증상                                                                  | 원인                             |
|----|---------------------------------------------------------------------|--------------------------------|
| F1 | 하네스 skill 우회 — 자매가 혼자 worker 스폰해서 처리                                | skill 진입 강제 부재                 |
| F2 | DoD 자동 강제 실패 — <code>build</code> 통과만 보고 완료 판정                      | sprint contract / validator 부재 |
| F3 | QA 단계 누락 — 사용자가 수동으로 다량이 호출 필요                                      | 자동 라우팅 없음                      |
| F4 | 핸드오프 멘션 불안정 — 잘못된 자매 호출                                             | 분기가 LLM 판단에 의존                 |
| F5 | 중간 끊김 — <code>request-timed-out</code> 반복, <code>xhigh</code> 무한 대기 | 재시도/에스컬레이션 정책 없음               |
| F6 | 환경 검증 누락 — “Docker 없음” 으로 작업 skip 허용                                | 환경 전제 검사 없음                    |

본질은 단 한 줄로 요약된다: “자매가 하네스를 안 타고 본인이 처리한다.”

### 2.2 해결 전략 — 6 가지 설계 원칙

`.claude/rules/principles.md` 에 명시된 하드 룰이다. 이 원칙은 타협하지 않는다.

1. **강제 > 권고.** 모든 파이프라인 전이는 코드로 강제한다. LLM 판단에 맡기지 않는다.
2. **결정론적 FSM.** 자매 간 핸드오프는 XState 상태 전이다. 멘션은 사용자 알림 전용이다.
3. **Sprint Contract = 불변 계약.** 모든 스프린트는 시작 전에 `sprint-contract.json` 을 생성하고, DoD 를 Zod 스키마로 표현한 validator 가 pass / fail 을 판정한다. `build` 통과 = 완료는 금지다.
4. **Skill 강제 진입.** OpenClaw 자매가 하네스 skill 을 우회하면 post-hook 이 감지해 작업을 revert 한다.
5. **QA 체크리스트 의무.** 다량은 스프린트 타입별 체크리스트를 전부 체크해야 PASS 를 낼 수 있다.
6. **환경 검증 선행.** 실기동 검증 환경이 없으면 스프린트를 시작하지 않는다. “Docker 없음 → skip” 같은 escape hatch 는 contract 에서 사전 차단한다.

### 2.3 레일 메타포

왜 이름이 “rails” 인가?

- **레일 (rail) = XState FSM:** 갈 수 있는 경로를 물리적으로 제한

- **신호등 = Sprint Contract**: 다음 역으로 갈 수 있는 조건
- **역 (station) = 자매 작업 단계**: Plan / Implement / Review / Deploy
- **차단봉 = Skill 강제 진입 hook**
- **긴급 정차 버튼 = 에스컬레이션 policy**
- **중앙 통제소 = MariaDB orchestrator state**

사용자는 출발 버튼만 누르고, 긴급 상황에서만 호출된다.

## 3. 4 자매는 누구인가

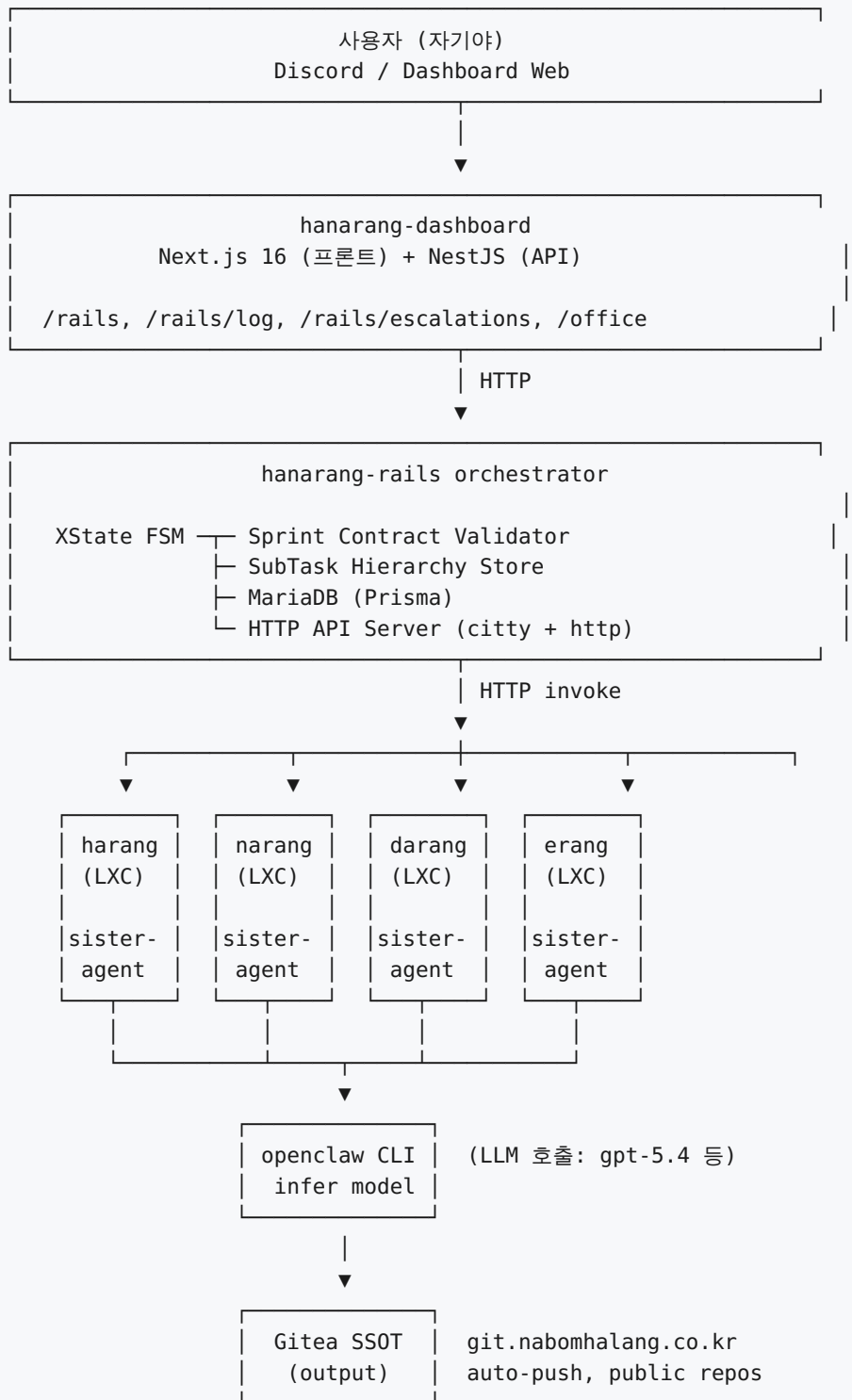
4 자매는 4 개의 서로 다른 LLM 에이전트다. 각자 성격/말투/역할이 다르고, OpenClaw 런타임 위에서 독립된 LXC 컨테이너에 돌아간다.

| 자매 | 영문     | 역할                       | 단계        | 주 모델          |
|----|--------|--------------------------|-----------|---------------|
| 하랑 | harang | Planner — 요구사항 해석, 계획 작성 | plan      | gpt-5.4       |
| 나랑 | narang | Implementer — 코드/문서 생성   | implement | gpt-5.4       |
| 다랑 | darang | Reviewer — QA, 체크리스트 검증  | review    | gpt-codex-5.3 |
| 이랑 | erang  | Deployer — 배포 검증, 인프라    | deploy    | glm-5-turbo   |

각 자매는 내부적으로 **manager** → **principal** → **lead** → **junior** 4 단계 계층을 가진다. 사용자가 “X 를 만들어 줘” 라고 하면, 각 자매의 manager 가 태스크를 받고 복잡도에 따라 하위 junior / lead 에게 분배한다. 복잡한 태스크일수록 더 깊게 파고들어가 병렬 처리된다 (자세한 내용은 § 7).

## 4. 시스템 구성도

### 4.1 하이 레벨



## 4.2 물리 토폴로지

| 역할                 | 호스트          | IP                    | 내용                                             |
|--------------------|--------------|-----------------------|------------------------------------------------|
| 사용자                | 개인 PC        | —                     | Discord 클라이언트, 대시보드 웹 브라우저                     |
| Proxmox hypervisor | 192.168.1.31 | —                     | VM / LXC 전체 호스트                                |
| Dev VM (SSOT)      | VM 200       | 10.10.10.169          | hanarang-rails + hanarang-dashboard 실제 구동, PM2 |
| 하랑이 LXC            | LXC          | 내부망                   | sister-agent daemon + OpenClaw 런타임             |
| 나랑이 LXC            | LXC          | 내부망                   | //                                             |
| 다랑이 LXC            | LXC          | 내부망                   | //                                             |
| 이랑이 LXC            | LXC          | 내부망                   | //                                             |
| Gitea              | Docker       | git.nabomhalang.co.kr | SSOT 저장소 (public + private), SSH 2222          |
| MariaDB            | Dev VM       | 10.10.10.169:3306     | hanarang_rails DB                              |

하나의 파이프라인 요청은 최대 10 개 이상의 서브 프로세스로 확장될 수 있다 (4 자매 × manager/principal/lead/junior 계층). 병렬 실행은 Promise.all 기반이고, 동시 실행 한도는 자매별로 설정 가능하다 (기본 8, 나랑이는 6).

## 5. 데이터 모델 — MariaDB 스키마

`prisma/schema.prisma` 에 정의되어 있다. 파이프라인 한 번의 실행이 각 테이블에 남기는 흔적을 따라가면 시스템 전체가 보인다.

### 5.1 테이블 요약

| 테이블                            | 설명                                               | 키    |
|--------------------------------|--------------------------------------------------|------|
| <code>pipelines</code>         | 하나의 파이프라인 실행 (= 사용자 요청 1 회)                      | ULID |
| <code>state_transitions</code> | FSM 상태 전이 로그 (감사 용)                              | auto |
| <code>sub_tasks</code>         | 자매/역할별 서브 태스크 트리                                 | ULID |
| <code>sub_task_events</code>   | 서브 태스크 수명 이벤트 (spawned/started/completed/failed) | auto |
| <code>contracts</code>         | Sprint Contract 스냅샷 (DoD + validator 정의)         | ULID |
| <code>escalations</code>       | 사용자 개입이 필요해진 예외 상황                               | ULID |
| <code>actor_spawns</code>      | 자매 프로세스 스폰 로그 (레거시)                              | auto |

### 5.2 Pipeline 레코드의 생애

```

idle -(START)→ running -(ALL_STAGES_DONE)→ completed
                        |
                        ├──(TIMEOUT 3회)→ escalated
                        └──(FATAL_ERROR)→ failed
  
```

`currentState` 는 XState 의 현재 노드, `contextJson` 은 FSM 의 context (전 단계 결과물 포함) 을 serialize 한 것이다. 매 전이마다 `StateTransition` row 가 한 줄씩 추가되므로, 나중에 `GET /api/transitions?pipelineId=...` 로 전체 이력을 재생할 수 있다.

### 5.3 SubTask 트리

각 파이프라인은 여러 개의 `sub_tasks` 를 만든다. 예를 들어 “todo 앱 만들어 줘” 라는 요청 하나가 다음 트리를 만들 수 있다.

```

harang-manager (role=manager, stage=plan)
└─ harang-principal (plan 의 세부 항목 3 개를 쪼갬)
    │ harang-lead-1
    └─ harang-lead-2
narang-manager (role=manager, stage=implement)
└─ narang-lead-frontend
    │ │ narang-junior-html
    │ │ narang-junior-css
    │ └─ narang-junior-js
    └─ narang-lead-backend
        └─ narang-junior-api
darang-manager (role=manager, stage=review)
erang-manager (role=manager, stage=deploy)

```

`parentId` 체인으로 트리를 재구성할 수 있고, 대시보드의 “서브태스크 상세 드로어”가 이 트리를 직접 렌더링한다.

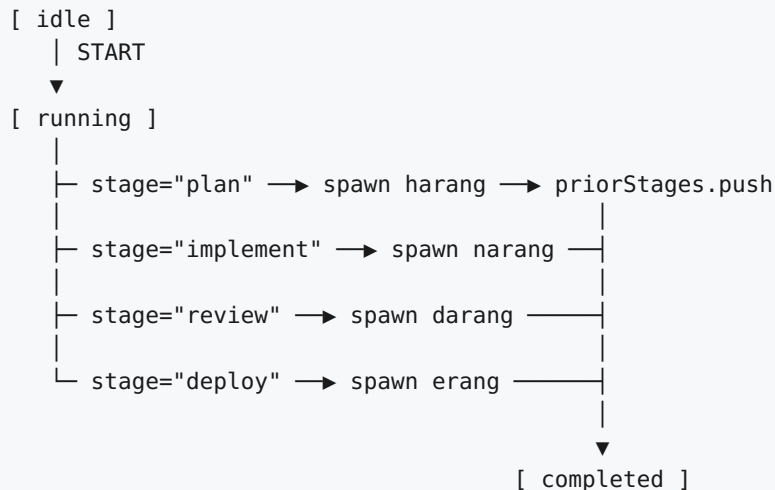
## 6. Orchestrator — XState FSM 엔진

`src/orchestrator/` 는 파이프라인의 심장이다.

### 6.1 파일 구조

| 파일                      | 역할                                                                                                       |
|-------------------------|----------------------------------------------------------------------------------------------------------|
| <code>machine.ts</code> | XState <code>setup({types}).createMachine(...)</code> 로 FSM 정의                                           |
| <code>runner.ts</code>  | 파이프라인 실행 루프 (actor 생성, 이벤트 dispatch, 단계 간 체이닝)                                                           |
| <code>persist.ts</code> | <code>getPersistedSnapshot()</code> 으로 FSM 상태를 DB 에 왕복 저장                                                |
| <code>context.ts</code> | FSM context 타입 (pipelineId, stage 결과, priorStages...)                                                    |
| <code>events.ts</code>  | <code>START</code> , <code>STAGE_DONE</code> , <code>TIMEOUT</code> , <code>FATAL_ERROR</code> 등 이벤트 스키마 |

### 6.2 상태 흐름



각 stage 는 순차적으로 실행되지만, **stage 내부** 에서는 계층 구조 (manager → principal → lead → junior) 가 `Promise.all` 로 병렬 실행된다. 그래서 한 stage 안에 10 개 이상의 junior 가 동시에 코드를 쓰는 일이 자주 생긴다.

## 6.3 priorStages 체이닝

가장 중요한 구조적 결정. `plan` 의 결과물 텍스트가 `implement` 의 프롬프트에 통째로 들어간다. `implement` 가 만든 파일 목록이 `review` 의 입력이 되고, `review` 의 verdict 가 `deploy` 의 컨텍스트가 된다. 자매는 다음 자매의 결과물을 모른 채 일하지 않는다.

구현:

```
// src/orchestrator/runner.ts
const priorStages: PriorStageOutput[] = [];
for (const stage of ["plan", "implement", "review", "deploy"]) {
  const result = await invokeSister(stage, { priorStages });
  priorStages.push({ stage, text: extractStageText(result) });
}
```

`extractStageText` 는 결과물 JSON 에서 `summary`, `repoUrl`, `rawUrlBase`, `producedFiles`, `filesCount` 를 뽑아 자연어 요약으로 합친다.

## 7. 역할 계층 — Manager/Principal/Lead/Junior

### 7.1 왜 계층이 있는가

LLM 한 개에게 “todo 앱 풀스택으로 만들어 줘” 라고 던지면 컨텍스트 한계에 부딪힌다. 사람 팀과 똑같이, 부장은 방향을 결정하고 신입은 코드를 친다. 이걸 구조적으로 강제하면 LLM 의 약점 (컨텍스트 파편화, 집중력 분산) 을 회피할 수 있다.

### 7.2 역할 정의 ( `src/hierarchy/roles.ts` )

| Role      | 한국어 | 주 모델          | 하위 스폰 가능                | 최대 스폰 |
|-----------|-----|---------------|-------------------------|-------|
| manager   | 부장  | gpt-5.4       | principal, lead, junior | 4     |
| principal | 수석  | gpt-5.4       | lead, junior            | 3     |
| lead      | 선임  | gpt-codex-5.3 | junior                  | 4     |
| junior    | 신입  | glm-5-turbo   | (없음)                    | 0     |

manager 는 직접 코드를 짜지 않는다. 대신 하위 직원에게 쪼개서 던진다. junior 는 리프 노드이며 실제 파일 생성을 책임진다.

### 7.3 복잡도 스코어 ( `src/hierarchy/complexity.ts` )

태스크가 들어오면 먼저 complexity 점수를 계산한다.

```
score = (길이_점수 × 0.3)
        + (키워드_점수 × 0.5)
        + (범위_점수 × 0.2)
```

키워드 “풀스택”, “데이터베이스”, “인증”, “배포”, “아키텍처” 등은 가산점. 최종 score (0-100) 는 tier 로 매핑된다.

| Tier     | 점수     | 권장 분해                           |
|----------|--------|---------------------------------|
| trivial  | 0-20   | junior 한 명                      |
| simple   | 21-40  | lead 한 명 또는 junior 2            |
| moderate | 41-60  | principal 1, lead 1, junior 2-3 |
| complex  | 61-80  | principal 1, lead 2, junior 4   |
| massive  | 81-100 | principal 2, lead 3, junior 6+  |

이 분해는 `planner.ts` 의 `DecompositionPlan` 으로 표현되고, `spawn.ts` 의 재귀 트리 워커가 그걸 받아 실제 LLM 호출 그래프를 만든다.

## 8. Sister Agent — 레일 위를 달리는 열차

`sister-agent/` 는 각 LXC 에 독립적으로 배포되는 daemon 이다. 네 자매 모두 동일한 코드 베이스를 쓰지만, 환경변수 `AGENT_NAME` (harang / narang / darang / erang) 로 정체성을 구분한다.

### 8.1 엔드포인트

```
POST /invoke
Content-Type: application/json

{
  "pipelineId": "01HXXXX...",
  "stage": "implement",
  "task": { "title": "...", "description": "...", "workdir": "" },
  "priorStages": [ { "stage": "plan", "text": "..." } ],
  "timeoutMs": 600000,
  "railsApiUrl": "http://10.10.10.169:18800"
}
```

리턴은 `HandoffMessage` 디스크리미네이티드 유니온이다.

```
{ "stage": "implement", "verdict": "IMPL_DONE",
  "payload": { "branch": "main", "commits": [...], "workdir": "...",
    "selfTestReport": { "producedFiles": [...], "repoUrl": "..." } }}
```

### 8.2 실행 파이프라인 ( `src/spawn.ts` )

```
runSpawnNode(ctx, node)
├─ prompts.build(role, stage, task, priorStages) // 한국어 역할 프롬프트
├─ llm.infer(prompt, model) // openclaw CLI 호출
├─ maybeExtractFiles(llmText, role, stage) // 코드 블록 파싱
│   ├─ ``lang:path 패턴 감지
│   ├─ 파일 경로 sanitize
│   └─ ctx.producedFiles.push(`${stage}/files/${path}`)
├─ for child of node.children: // 하위 직원 재귀
│   await runSpawnNode(ctx, child) // Promise.all
└─ buildSuccessResult(node, producedFiles)
```

## 8.3 LLM 호출 — `openclaw infer model run`

각 자매는 로컬에서 `openclaw infer model run --model gpt-5.4 --json` 서버프로세스를 실행한다. stdout 은 Zod 로 검증된 후 쓴다. LLM 응답의 결정론성은 아래 세 가지로 관리한다.

1. **엄격한 프롬프트 템플릿** — 역할/단계별 한국어 템플릿이 `prompts.ts` 에 고정
2. **구조화 응답 요구** — “이 형식 밖으로 나가면 재시도” 지시를 프롬프트 끝에 삽입
3. **코드 블록 규약** — ````lang:path/to/file.ext` 형태로 내놓으라고 명시, 파서가 이걸 기대

## 8.4 코드 블록 추출 ( `src/code-extractor.ts` )

LLM 응답에서 파일을 꺼내는 로직이다. 기대 포맷:

```
```html:frontend/index.html
<!doctype html>
...

```css:frontend/style.css
body { ... }
```
```

파서는:

1. 정규식으로 ````` 블록 탐지
2. 언어 뒤의 `:path` 힌트 추출
3. path sanitize: `..`, 절대 경로, 백슬래시 금지
4. path 가 없으면 언어별 기본 파일명 ( `snippet.html` 등)
5. `{ path, lang, content }` 리스트 반환

이 결과는 `maybeExtractFiles` 가 받아서 실제 파일로 쓰고 `ctx.producedFiles` 에 상대 경로를 기록한다.

## 9. Git 자동 푸시 — Gitea 연동 (`git-ops.ts`)

파이프라인이 만든 파일은 즉시 Gitea 에 올라가 실행 가능한 URL 로 바뀐다.

### 9.1 흐름

1. implement stage 가 끝나면 `commitAndPush(pipelineId, workdir)` 호출
2. 리포 이름은 `rails-${pipelineId.slice(-10).toLowerCase()}` (예: `rails-abcd012345`)
3. Gitea API 로 `hanarang` org 에 public repo 자동 생성 `POST /api/v1/orgs/hanarang/repos`
4. 로컬 `git init` → 커밋 → `git push https://user:TOKEN@git.nabomhalang.co.kr/...`
5. 리턴:

```
{ ok:true, repoUrl:"https://git.nabomhalang.co.kr/hanarang/rails-abcd012345",
  rawUrlBase:"https://git.nabomhalang.co.kr/hanarang/rails-abcd012345/raw/branch/main",
  commit:"a1b2c3d", filesCount:7 }
```

### 9.2 프리뷰 URL 추론

`derivePreviewUrl` 이 `producedFiles` 에서 `.html` 파일을 찾아 `${rawUrlBase}/${html파일경로}` 로 즉시 열 수 있는 공개 URL 을 계산한다. 결과는 `selfTestReport.deployUrl` 에 들어가고, 대시보드의 “url” 배치가 달린 `FileRow` 로 사용자에게 보여진다. 클릭하면 브라우저에서 바로 열린다.

### 9.3 왜 Gitea 인가

- `git.nabomhalang.co.kr` 은 우리 내부 SSOT 서버다 (Docker 로 Dev VM 에서 돌고 있음)
- `gh` CLI 는 GitHub 전용이라 쓸 수 없고, 대신 `tea` CLI 또는 REST API 로 접근한다
- 토큰: `.env` 의 `GITEA_TOKEN` 에 저장, 코드에서는 URL 에 `user:TOKEN@` 형태로만 사용

## 10. 대시보드 — hanarang-dashboard

`hanarang-dashboard` 는 별도 repo 이며, rails 가 돌고 있는 모든 것을 시각화한다. Next.js 16 (Turbopack) + NestJS API + Socket.IO 실시간 이벤트로 만들어졌다.

### 10.1 페이지

| 경로                              | 설명                            |
|---------------------------------|-------------------------------|
| <code>/rails</code>             | 활성 파이프라인 리스트 + SubTask 트리 시각화 |
| <code>/rails/log</code>         | 상태 전이 감사 로그 (SIEM 스타일)        |
| <code>/rails/escalations</code> | 에스컬레이션 큐                      |
| <code>/office</code>            | 4 자매 대화 스트림 (사용자가 구경하는 용)     |
| <code>/sisters/[name]</code>    | 자매 개별 프로필 + 통계                |

### 10.2 SubTask 상세 드로어

`/rails` 에서 노드를 클릭하면 우측 드로어가 열린다. 이 드로어에 들어가는 정보:

- **헤더**: 자매 아바타, role 배지, title, breadcrumb (부모 체인)
- **상태/모델 그리드**: state, agent, model, duration, complexity, ID
- **설명**: 태스크 description
- **산출물 (Artifacts)**:
  - `.md` 로그 파일 → 클릭 시 모달로 내용 표시 (`FileViewerModal`)
  - 추출된 코드 파일 → 클릭 시 Gitea 프록시로 페치해서 표시
  - Deploy URL → 브라우저 외부 링크
- **LLM 응답**: `react-markdown` 으로 렌더링 (front matter 는 분리)
- **하위 노드 리스트**: children 요약
- **이벤트 로그**: SubTaskEvent 전체

### 10.3 FileViewerModal

가장 최근에 추가된 기능. 대시보드에서 파일 내용을 보고 싶을 때 쓴다.

```
FILE    implement/files/index.html
      [복사] [닫기]
```

```
<!doctype html>
<html>
...
```

두 가지 소스 타입을 받는다:

1. `{ type: 'llm', text }` — 이미 메모리에 있는 LLM 결과물 (로그 .md 용)
2. `{ type: 'url', url }` — Gitea raw URL, 백엔드 `/api/rails/file-content` 프록시로 페치

프록시는 Gitea 호스트만 allowlist 한다 ( [git.nabomhalang.co.kr](https://git.nabomhalang.co.kr) ). 외부 URL 은 거부.

.md 파일은 `react-markdown` 으로, 아닌 것은 `<pre>` 로 표시. Front matter ( `--- ... ---` ) 는 상단 메타 박스로 분리한다.

## 11. End-to-End 시나리오 — “todo 앱 만들어 줘”

처음 보는 사람이 가장 궁금해할 “한 번의 실행” 을 코드 흐름으로 따라가자.

### 11.1 Step 0 — 트리거

사용자가 Discord 에 다음과 같이 쓴다.

```
/rails start project:todo-app requirements:"간단한 todo 웹앱 하나 만들어 줘"
```

Discord 봇은 이걸 HTTP 요청으로 바꿔 Dev 서버 대시보드 백엔드로 보낸다.

```
POST http://dev-vm/api/rails/pipelines/start
{
  "project": "todo-app",
  "requirements": "간단한 todo 웹앱 하나 만들어 줘"
}
```

### 11.2 Step 1 — 오케스트레이터 진입

대시보드 백엔드 ( [RailsService](#) ) 가 rails orchestrator 에 포워딩.

```
POST http://127.0.0.1:18800/pipelines/start
```

rails 는:

1. ULID 를 발급해 [pipelines](#) 테이블에 새 row 를 만든다 ( `currentState='idle'` )
2. XState actor 를 생성해 [START](#) 이벤트 dispatch → [running](#) 상태로 전이
3. [state\\_transitions](#) 에 `idle → running` 한 줄 기록
4. 4 단계 루프를 시작한다

### 11.3 Step 2 — Plan (하랑이)

rails 는 harang LXC 의 [/invoke](#) 로 POST:

```
{ stage:"plan", task:{ title:"todo-app", description:"..."}, priorStages:[] }
```

harang sister-agent 는:

1. [harang-manager](#) SubTask row 생성, [sub\\_task\\_events](#) 에 `spawned` , `started` 이벤트
2. complexity 계산 → 점수 35 → [simple](#) tier → principal 1 + junior 1 로 분해

3. 각 하위 노드를 `Promise.all` 로 LLM 호출
4. junior 가 반환한 계획을 manager 가 취합, ```md:plan.md` 코드 블록으로 감싼 응답을 만들
5. `maybeExtractFiles` 로 `plan/files/plan.md` 로 저장, `producedFiles` 에 기록
6. `{stage:"plan", verdict:"PLAN_READY", payload:{ planDir, sprintId, selfTestReport:{producedFiles} }}` 리턴

rails 는 결과를 `priorStages[0]` 에 푸시한다.

## 11.4 Step 3 — Implement (나랑이)

```
POST /invoke
{ stage:"implement",
  task:{...},
  priorStages:[ { stage:"plan", text:"<plan.md 요약>" } ] }
```

narang sister-agent 는:

1. complexity 60 → `moderate` → principal 1 + lead 2 (frontend/backend) + junior 4
2. 병렬로 LLM 호출, junior 들이 각각 HTML / CSS / JS / server.js 를 생성
3. 모든 산출물을 `implement/files/...` 로 저장
4. `git-ops.commitAndPush(pipelineId, workdir)` 호출
  - Gitea 에 `rails-abcd012345` repo 생성
  - `git push` 성공
  - `repoUrl`, `rawUrlBase`, `commit` 리턴
5. `derivePreviewUrl(producedFiles, rawUrlBase)` 로 `https://.../implement/files/frontend/index.html` 계산
6. `{stage:"implement", verdict:"IMPL_DONE", payload:{ ..., selfTestReport:{ producedFiles, repoUrl, rawUrlBase, deployUrl } }}` 리턴

## 11.5 Step 4 — Review (다랑이)

```
POST /invoke
{ stage:"review",
  priorStages:[
    { stage:"plan", text:"..." },
    { stage:"implement", text:"repoUrl=...\nfilesCount=7\n..." }
  ] }
```

darang 은 `rawUrlBase` 로 Gitea 파일을 직접 페치해서 읽고, QA 체크리스트를 돌린다. 결과는 `verdict:"APPROVE" | "REQUEST_CHANGES" | "ABORT"`.

REQUEST\_CHANGES 가 나오면 rails 는 implement 로 되돌려 재시도 (최대 3 회). 3 회 실패 시 `escalated` 상태로 전이하고 사용자에게 알림.

## 11.6 Step 5 — Deploy (이랑이)

erang 은 deploy URL 이 실제로 열리는지 verify, 필요하면 추가 설정 파일을 쓴다. 최종적으로 `{stage:"deploy", verdict:"DEPLOY_DONE", payload:{ deployArtifactPath, verificationResults }}`.

## 11.7 Step 6 — 완료

rails 는 `running` → `completed` 로 전이, 대시보드 Socket.IO 로 실시간 브로드캐스트. 사용자 Discord 에는 최종 deploy URL 이 포스트된다.

```

✓ todo-app 완료
  repo: https://git.nabomhalang.co.kr/hanarang/rails-abcd012345
  deploy: https://git.nabomhalang.co.kr/hanarang/rails-abcd012345/raw/branch/main/
  implement/files/frontend/index.html
  duration: 4m 12s
  sub-tasks: 11 (완료 11, 실패 0)
```

## 12. HTTP API 명세

rails orchestrator 가 노출하는 엔드포인트. 대시보드 백엔드와 sister-agent 가 소비한다.

| 메서드  | 경로                                                              | 설명                                     |
|------|-----------------------------------------------------------------|----------------------------------------|
| GET  | <code>/health</code>                                            | 헬스체크                                   |
| GET  | <code>/pipelines?limit=N</code>                                 | 파이프라인 리스트                              |
| GET  | <code>/pipelines/:id</code>                                     | 파이프라인 상세 (state, context, transitions) |
| POST | <code>/pipelines/start</code>                                   | 새 파이프라인 실행                             |
| POST | <code>/pipelines/:id/abort</code>                               | 파이프라인 강제 종료                            |
| GET  | <code>/api/pipelines/:id/sub-tasks</code>                       | SubTask 트리                             |
| GET  | <code>/api/sub-tasks/:id</code>                                 | 서브태스크 상세 (parents/children/events)     |
| GET  | <code>/api/transitions?pipelineId=...&amp;limit=100</code>      | 상태 전이 이력                               |
| GET  | <code>/api/escalations?pipelineId=...&amp;resolved=false</code> | 에스컬레이션 큐                               |

대시보드 쪽 ( `backend/src/rails/` ) 은 이것들을 래핑해서 `/api/rails/*` 로 재노출하고, 인증/인가를 한 겹 더 얹는다.

## 13. Sprint Contract — DoD 의 기계 검증

---

### 13.1 왜 필요한가

F2 실패 모드 (“build 통과 = 완료”) 를 막기 위해서. 스프린트가 시작되기 전에 “이 스프린트는 무엇으로 끝난 것으로 보는가” 를 기계가 읽을 수 있는 형태로 고정한다.

### 13.2 구조

`.claude/state/contracts/<task-id>.sprint-contract.json`

```
{
  "taskId": "11.2",
  "sprintId": "SPRINT-003",
  "version": "v1",
  "checks": [
    { "id": "files-exist", "type": "file-exists", "paths": ["src/contract/
      generator.ts"] },
    { "id": "tests-pass", "type": "command-success", "cmd": "pnpm test src/contract" },
    { "id": "schema-valid", "type": "artifact-schema", "path": "out/contract.json",
      "schema": "ContractSchema" }
  ],
  "nonGoals": ["UI 변경"],
  "reviewerProfile": "static",
  "riskFlags": ["security-sensitive"]
}
```

### 13.3 체크 타입 ( [src/contract/checks/](#) )

| 타입                             | 의미                             |
|--------------------------------|--------------------------------|
| <code>file-exists</code>       | 경로 존재 여부                       |
| <code>command-success</code>   | 셸 명령 exit code 0               |
| <code>http-status</code>       | URL 응답 2xx                     |
| <code>regex-in-file</code>     | 파일 내용이 정규식 매칭                  |
| <code>artifact-schema</code>   | JSON 산출물이 Zod 스키마 통과           |
| <code>db-query</code>          | DB 쿼리가 기대 행 수 리턴               |
| <code>process-listening</code> | 포트 LISTEN 확인                   |
| <code>manual</code>            | 수동 체크박스 (escape hatch, 최소화 권장) |

하나라도 FAIL 이 나오면 스프린트는 [cc:완료](#) 가 될 수 없다.

# 14. 보안 모델

## 14.1 신뢰 경계

| 경계                   | 정책                                |
|----------------------|-----------------------------------|
| 사용자 → 대시보드           | 세션 쿠키 인증 (NestJS)                 |
| 대시보드 → rails         | 내부망 전용 HTTP, 토큰 없음 (향후 추가 예정)     |
| rails → sister-agent | 내부망 HTTP, AGENT_NAME 환경변수로 정체성 고정 |
| sister-agent → LLM   | OpenClaw 런타임이 API 키 관리            |
| rails → Gitea        | .env 의 GITEA_TOKEN , URL 에만 주입    |

## 14.2 Gitea 프록시 allowlist

대시보드 백엔드 /api/rails/file-content 는 URL.host === 'git.nabomhalang.co.kr' 만 허용. 외부 URL 은 404 를 돌려준다. 이유: 악의적 링크로 백엔드에서 임의 HTTP 요청을 트리거하는 SSRF 공격 방지.

## 14.3 Zod 검증 경계

모든 외부 입력 (HTTP body, subprocess stdout, 파일 로드) 은 Zod 스키마를 통과한 뒤에만 내부 타입으로 들어온다. 경계 밖에서는 any 금지.

## 15. 실패/복원력 ( `src/resilience/` )

---

### 15.1 재시도 정책

Exponential backoff — `1s, 2s, 4s, 8s`, 최대 `30s`. 기본 3 회. 매 재시도는 `sub_task_events` 에 `retry` 이벤트로 기록된다.

### 15.2 타임아웃

자매 `/invoke` 응답 기본 600 초 (LLM 이 오래 걸릴 수 있어서). 초기에는 30 초로 잡았다가 `request-timed-out` 재현 → 600 초로 변경.

### 15.3 에스컬레이션

N 회 실패 시 `escalations` 테이블에 row 추가, Discord 에 사용자 멘션. 상태는 `escalated` 로 전이하고 파이프라인은 정지한다. 사용자가 `rails resume <id>` 를 호출하면 `escalated` → `running` 으로 복구.

## 16. 설치/실행 가이드

### 16.1 사전 요구

- Node 22 + pnpm
- MariaDB 10.11+
- Gitea 인스턴스 (또는 환경변수 `GITEA_TOKEN` + `GITEA_API_URL` 재설정)
- OpenClaw 런타임 (각 자매 LXC)

### 16.2 rails 서버 구동

```
git clone https://git.nabomhalang.co.kr/hanarang/hanarang-rails.git
cd hanarang-rails
pnpm install
cp .env.example .env
# DATABASE_URL, GITEA_TOKEN 등 채우기
pnpm prisma migrate deploy
pnpm build
pnpm rails serve          # 18800 포트
```

### 16.3 sister-agent 구동 (각 LXC)

```
cd sister-agent
pnpm install
pnpm build
AGENT_NAME=harang RAILS_API_URL=http://dev-vm:18800 \
node dist/server.js
```

### 16.4 대시보드 구동

별도 repo `hanarang-dashboard` 참조. `pnpm build && pm2 start ecosystem.config.js`.

### 16.5 스모크 테스트

```
pnpm rails run hello-world --mock -r "Try a pipeline"
pnpm rails status
```

`--mock` 모드는 실제 LLM 호출 없이 결정론 파이프라인만 확인한다.

## 17. 디렉토리 구조

```

hanarang-rails/
├── src/
│   ├── orchestrator/      FSM 엔진
│   ├── hierarchy/         계층/복잡도/플래너
│   ├── handoff/           자매 간 메시지 스키마 + 트랜스포트
│   ├── contract/          Sprint Contract + validator
│   ├── qa/                QA 체크리스트 runtime
│   ├── enforcement/       Skill 강제 진입 / bypass 감지
│   ├── resilience/        재시도/타임아웃/에스컬레이션
│   ├── server/http.ts     HTTP API 서버
│   ├── cli/               citty 기반 rails CLI
│   ├── config/            env + config loader
│   └── bridge/            Discord 브릿지 (v0.2 예정)
├── sister-agent/
│   └── src/
│       ├── server.ts       /invoke HTTP 서버
│       ├── spawn.ts        재귀 트리 실행기
│       ├── hierarchy.ts     역할 트리 builder
│       ├── complexity.ts    스코어 계산
│       ├── planner.ts       복잡도 → 분해 계획
│       ├── roles.ts         역할 정의
│       ├── prompts.ts       한국어 프롬프트 템플릿
│       ├── llm.ts           openclaw CLI wrapper
│       ├── code-extractor.ts 코드 블록 파서
│       ├── git-ops.ts       Gitea API + git push
│       └── rails-client.ts   rails 에 이벤트 report
├── prisma/schema.prisma    DB 스키마
├── .plans/
│   ├── OVERVIEW.md
│   ├── failure-audit.md
│   ├── design/             설계 문서 9 종
│   ├── sprints/            스프린트 000-007 명세
│   └── migration/
├── docs/
│   ├── GUIDE.md            ★ 이 문서
│   ├── migration-guide.md
│   ├── operations.md
│   └── discord-setup.md
├── hooks/                  OpenClaw pre/post-tool hooks
├── qa-templates/          QA 체크리스트 6 종
├── install.sh              설치 자동화
└── rails.config.example.yaml
  
```

## 18. 로드맵

| 버전     | 상태 | 내용                                           |
|--------|----|----------------------------------------------|
| v0.1.0 | 완료 | Sprint 000-007, FSM/contract/QA/migration 코어 |
| v0.1.1 | 완료 | 실 LLM 통합, 계층 실행, 파일 추출, Gitea auto-push      |
| v0.1.2 | 완료 | 대시보드 아티팩트 뷰, MD 파일 뷰어 모달                     |
| v0.2   | 진행 | Discord 브릿지 정식화, GatewayHttpTransport 분리     |
| v0.3   | 계획 | Skill 강제 진입 실측, OpenClaw hook 프로덕션 적용        |
| v0.4   | 계획 | 멀티 테넌시 (여러 사용자 동시 실행)                        |
| v1.0   | 계획 | 외부 공개 + 문서화 완성                               |

## 19. 용어집

| 용어              | 정의                                                             |
|-----------------|----------------------------------------------------------------|
| 자매 (Sister)     | 4 개의 LLM 에이전트 중 하나 (harang/narang/darang/erang)                |
| 자기야             | 사용자 (나봄하랑) 에 대한 4 자매의 호칭                                       |
| OpenClaw        | 하나랑 생태계에서 사용하는 AI 런타임. Claude Code 기반이지만 별개 브랜드                |
| Rails           | 이 프로젝트. 결정론적 파이프라인 오케스트레이터                                     |
| Harness         | rails 의 전임자 <a href="#">hanarang-harness</a> . 권고 기반이라 실패가 잦았음 |
| FSM             | Finite State Machine. XState 로 구현                              |
| Sprint Contract | 스프린트 시작 전에 쓰는 DoD 기계 검증 스펙                                     |
| DoD             | Definition of Done. 완료 조건                                      |
| SubTask         | 자매/역할별로 쪼개진 서브 태스크                                             |
| Stage           | 파이프라인의 주 단계 (plan/implement/review/deploy)                     |
| Role            | 자매 내부의 직급 (manager/principal/lead/junior)                      |
| Escalation      | 자동 복구 실패 시 사용자에게 넘기는 예외 상황                                     |
| priorStages     | 이전 단계 결과물 텍스트의 누적 배열                                           |
| producedFiles   | 자매가 이번 실행에서 만든 파일의 상대 경로 리스트                                   |
| SSOT            | Single Source of Truth. 여기서는 Dev VM 위의 Gitea + MariaDB         |
| LXC             | 리눅스 컨테이너. Proxmox 에서 각 자매를 격리 실행                               |

## 20. 참고 자료

---

- 원본 실패 감사: `.plans/failure-audit.md`
- 설계 문서: `.plans/design/state-machine.md`, `sprint-contract.md`, `hierarchy.md`, `deployment.md`, `handoff.md`, `retry-policy.md`, `qa-template.md`, `transports.md`, `triggers.md`
- 스프린트 명세: `.plans/sprints/SPRINT-000` ~ `SPRINT-007`
- 마이그레이션 가이드: `docs/migration-guide.md`
- 운영 가이드: `docs/operations.md`
- Discord 셋업: `docs/discord-setup.md`
- 전임자 repo: `hanarang/openclaw-harness` (private archive)
- 대시보드 repo: `hanarang/hanarang-dashboard`

## 부록 A. 용례 비교 — 구 하네스 vs rails

### A.1 핸드오프

#### 구 하네스

```
harang → "이제 다람이가 구현해 주세요" (Discord 멘션)
narang → 잠시 뒤 멘션을 본다 (혹은 못 봄)
  → 본인 판단으로 스폰, 직접 처리
  → skill 을 안 탐 (F1)
```

#### rails

```
stage="plan" → FSM context.priorStages.push({ stage:"plan", text:... })
XState transition(STAGE_DONE) → guard 검사 → next state="implement"
runner 가 자동으로 POST /invoke (stage=implement) → narang 실행
narang 은 선택권이 없다. 호출된 대로만 실행
```

### A.2 DoD

구 하네스: `npm run build` 가 exit 0 → 완료 처리.

rails: Sprint Contract 의 `checks[]` 가 전부 pass 해야 `cc:완료`. `artifact-schema` 체크는 산출물 JSON 을 Zod 로 한 번 더 검증한다.

### A.3 QA

구 하네스: 사용자가 “다람야 이거 QA 해 줘” 라고 멘션. 다람이가 답장 없음 → 사용자가 중재.

rails: FSM 이 자동으로 `review` stage 로 전이. 다람이는 반드시 호출되고, QA 템플릿의 체크리스트를 전부 채워야 `APPROVE` 를 낼 수 있다.

## 부록 B. 자주 묻는 질문

---

**Q. 왜 Claude Code 가 아니라 OpenClaw 인가?** A. OpenClaw 는 하나랑이 내부에서 쓰는 커스텀 런타임이다. Claude Code 를 포크한 것이 아니라 별개의 구현이다. 4 자매는 OpenClaw 위에 올라가 있고, 이 rails 레포 자체는 Claude Code 세션에서 개발한다.

**Q. 왜 SQLite 가 아니라 MariaDB 를 쓰나?** A. 초기 설계에서는 SQLite 를 썼지만, Dev VM 에 MariaDB 가 이미 있고 대시보드가 같은 DB 를 공유하는 게 간단해서 MariaDB 로 옮겼다. Prisma 로 추상화되어 있어 다시 바꾸는 것도 어렵지 않다.

**Q. 병렬 실행은 어디까지 가능한가?** A. stage 는 순차 (plan → implement → ...), stage 내부의 junior 스폰은 병렬. 기본 `default:8` 동시 실행, 나랑이는 빌드 자원 때문에 6 으로 제한. `concurrencyLimits.overrides` 로 자매별 조정 가능.

**Q. LLM 이 헛소리를 하면?** A. 세 겹의 방어가 있다. (1) 프롬프트 템플릿이 구조화 응답을 강제. (2) Zod 가 응답을 검증, 실패 시 재시도. (3) Sprint Contract 가 최종 산출물을 정적 검증.

**Q. 사용자 개입 없이 며칠 단위 장기 태스크가 가능한가?** A. 현재 v0.1.x 는 한 번의 파이프라인 = 한 번의 기획 → 배포 사이클이다. 더 긴 수명의 프로젝트는 여러 파이프라인을 엮는 방식으로 다룬다. v0.4 멀티 테넌시에서 검토 예정.

**Q. 테스트는 어떻게?** A. Vitest 105 테스트가 현재 통과. FSM, contract, QA, migration 핵심 경로를 커버한다. E2E 는 `--mock` 모드로 돌릴 수 있다.

## 부록 C. 라이선스 및 기여

---

- 라이선스: MIT ( [LICENSE](#) )
- 저작권: 나뭇하랑 / hanarang
- 기여: PR 환영. [.plans/](#) 문서 규약을 따를 것.
- 문의: Discord 또는 Gitea issue

하나랑의 4 자매가 사용자 중재 없이 달릴 수 있는 레일을 깎다 — 그것이 이 프로젝트의 처음이자 끝의 목표다.