BRIE · standarda

AX 를
스스로 하게 만드는 체계

처음 하는 사람이 전문가 없이 업무 자동화 앱을 만들고, 올리고, 운영하기까지 — 부딪히는 7개의 벽과, 그 벽마다 standarda 가 세워 둔 장치.

Brie · 2026년 11월
모든 숫자와 화면은 실제 서버·저장소·작업 기록에서 가져왔다.

질문

AX 를 맡기면 무엇이 남나

업무 자동화(AX)를 외부 전문가에게 맡기면 빠르다. 그런데 만든 사람이 떠나는 날, 고객에게 무엇이 남는지는 따로 물어야 한다.

대신 만들어 주면

- 고칠 사람이 떠나면 멈춘다 — 작은 변경도 다시 외주
- 왜 그렇게 만들었는지는 만든 사람 머릿속에 있다
- 서버·계정·코드가 업체 쪽에 있으면 옮기기도 어렵다
- 회사 안에는 "AI 로 일하는 법"이 쌓이지 않는다

스스로 하게 되면

- 업무를 아는 사람이 직접 만들고 고친다
- 만든 이유와 겪은 일이 문서와 교훈으로 남는다
- 서버·계정·코드가 처음부터 우리 회사 명의다
- 다음 업무, 다음 사람으로 이어진다

그런데 직접 하려면 벽이 있다

요즘은 AI 에게 말로 시키면 앱이 나온다. 하지만 말 한마디 뒤에는 처음 하는 사람이 답할 수 없는 질문이 숨어 있다 — 무엇이 다 필요한지, AI 가 만든 것이 맞는지, 모르고 한 실수는 없는지, 서버에는 어떻게 올리는지. 보통은 이 질문마다 전문가를 찾는다. 그래서 결국 다시 "맡기는" 쪽으로 돌아간다.

주장

전문가의 머릿속을
시스템에 옮겨 담았다

Brie 는 AX 를 대신 해 주지 않는다.
고객이 스스로 하게 만든다.

처음 하는 사람이 혼자 해낼 수 있는 이유는 그가 전문가가 되어서가 아니다. 전문가가 하나씩 답하던 질문들의 답을 **출발점·부품·일하는 방식·검사기·서버 도구·교훈**으로 미리 만들어 두었기 때문이다. 사람은 질문마다 공부하는 대신 무엇을 만들지 정하고, 승인하고, 확인한다.

3분 24초
한 문장에서 실제 HTTPS 주소가 열리기까지 (벽 1)

436개
53개 프로젝트에서 모인 교훈 — 작업을 시작하면 관련된 것이 따라온다 (벽 6)

2박스
다른 회사의 개발·운영 서버에 같은 체계를 세운 실례 (벽 7)

이 자료는 처음 하는 사람이 부딪히는 벽을 하나씩 짚고, 각 벽에서 standarda 가 무엇을 대신 정해 두었는지를 실제 화면·명령·기록으로 보여 준다. 각 장 뒤의 "안 들여다보기" 쪽은 그 장치가 실제로 어떻게 생겼는지 궁금한 사람을 위한 것이다.

차례

처음 하는 사람이 부딪히는 7개의 벽

1	**빈 화면** "어디서부터?" — 질문의 답이 든 출발점을 세 번의 요청으로 연다	출발점(template)	6
2	**AI 부품** AI 호출·문서 읽기·메일·결과 검증을 검증된 공용 부품으로	공용 부품(core)	18
3	**AI 에게 어떻게 시키나** 코드를 읽지 못해도 무엇을 만들지는 내가 정한다	일하는 방식(/develop)	28
4	**모르는 실수** 모르고 한 실수는 저장하기 전에 막힌다	검사기(hook)	45
5	**서버·배포·운영** "배포해 줘" 한 마디, 실패하면 스스로 되돌아온다	서버 헬퍼(/release)	58
6	**혼자 배운 건 혼자만 안다** 한 사람이 겪은 일이 다음 사람의 출발점이 된다	교훈 고리(lesson-finder)	70
7	**우리 회사 인프라에서도?** 우리 회사의 서버·계정·도메인 위에 같은 체계가 선다	값 파일(site.env)	81
+	**처음부터 배포까지, 한 번에** 7개의 장치가 한 흐름으로 이어지는 모습과 지금까지의 기록		91

한 장 그림

standarda 는 여섯 층으로 되어 있다

각 층은 처음 하는 사람이 혼자 답하기 어려운 질문 묶음 하나를 맡는다. 어느 층도 사람에게 공부를 요구하지 않는다 — 정해 둔 답을 쓰게 한다.

출발점 standarda-template	새 프로젝트의 표준 골격. 계정 기능 한 세트, 완전한 견본 업무 앱, 디자인, AI 작업 규칙까지 파일 212개가 처음부터 들어 있다.	벽 1
공용 부품 standarda-core	AI 호출·캐시·비용 집계·문서 읽기·메일·결과 검증을 검증된 부품으로. 한 번 고치면 버전 번호만 올려 모든 프로젝트가 받는다.	벽 2
일하는 방식 규칙집·스킬·에이전트	개발 요청은 늘 같은 7단계(`/develop`)로. 사람은 쉬운 말 요약을 한 번 승인하고, 대조·점검은 AI 와 검증 에이전트가 한다.	벽 3
검사기 hook	저장·커밋하기 전에 도는 검사 11개. 모르고 한 위험한 실수는 이유와 함께 막히고, 고칠 방법이 안내된다.	벽 4
서버 도구 devtools·서버 헬퍼	개발 주소 열기, 운영 배포, DB 스냅샷과 자동 되돌리기, 여러 사람이 한 서버를 쓰는 구조, 값 파일 하나로 다른 회사에 세우기.	벽 5·7
지식 팀 wiki	교훈 436개와 색인, 변경 기록 114건. 다음 작업이 시작될 때 관련 교훈이 자동으로 따라오고, 두 번 겪은 일은 기본값으로 올라간다.	벽 6

01

벽 1 · 빈 화면

빈 화면에서 시작하지 않는다

업무 자동화 앱을 직접 만들겠다고 마음먹은 사람이 가장 먼저 만나는 벽은 기술이 아니라 **"어디서부터?"**다. standarda 에서 시작은 **한 문장**이고, 끝은 남에게 보여 줄 수 있는 **실제 주소**다.

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 첫 번째 장치.

벽 — 말로 시키면 되지 않나?

첫 화면을 띄우기도 전에 정해야 할 것이 너무 많다

요즘은 말로 시키면 된다. "앱 만들어 줘. 회원가입·로그인 만들어 줘. DB 만들어 줘. 서버 주소 준비해 줘." 맞다 — standarda 도 말로 시킨다. 문제는 **그 말 한마디 뒤에 숨은 질문들**이다.

"로그인 만들어 줘" → 그 뒤에 따라오는 질문들

무엇이 다 필요한가
회원가입·로그인·아이디 찾기·비밀번호 찾기가 한 세트로 다 있어야 한다는 것부터

누가 가입할 수 있나
아무나? 관리자 승인 후? 우리 회사 메일만? Google 로 가입하는 사람에게도 같은 규칙인가

가입할 때 무엇을 받나
이메일만? 이름·회사도? 약관 동의는? 아이디를 이메일로 쓸지 따로 만들지

비밀번호 규칙은
몇 자 이상, 어떤 기호까지? 화면에서 안내하는 규칙과 실제로 받아 주는 규칙이 같은가

잊어버리면
임시 비밀번호를 보낼까, 인증 후 직접 새로 정하게 할까. 아이디 찾기는 무엇을 찾아 주나

메일은 무엇으로 보내나
어떤 계정으로, 어떤 설정으로? 스팸함으로 가지는 않나. 설정이 빠지면 화면에 뭐가 뜨나

로그인하고 나면 어디로
공유받은 링크로 들어온 사람은 로그인 후 원래 보려던 화면으로 가야 한다

관리자는
첫 관리자 계정은 누가, 어떤 비밀번호로 만드나. 운영 서버에도 따로 있어야 하나

Google 로그인도 되나
로그인만 할지, Gmail·Drive 까지 쓸지. 둘은 별개의 동의다

실제 주소에서도 되나
내 PC 에서 되던 로그인이 https 실제 주소에 올리면 그대로 될까

AI 는 이 질문마다 **그럴듯한 답을 골라** 만들어 준다. 처음 해 보는 사람은 그 답이 맞는지, 무엇을 더 물어야 하는지 알 수 없다. 그래서 남는 길은 둘이다.

① 찾아보고 공부한다

질문마다 검색하고, 물어보고, 시행착오를 겪는다. 바이브 코딩이라도 결국 **본인의 시간과 노력**이 든다.

② 일단 넘어간다

모르는 질문은 고민조차 하지 않는다. 그러다 실제로 쓰는 날 **막히고 나서야** 그런 질문이 있었다는 걸 안다. 그때 고치는 데 더 오래 걸린다.

standarda 의 장치

그 질문들의 답이 든 출발점을 세 번의 요청으로 연다

standarda 의 출발점에는 앞 쪽 질문들의 답이 이미 정해져 들어 있다. 그래서 사람은 질문마다 공부하는 대신, 그 출발점을 여는 세 가지만 한다.

1 말 한마디로 프로젝트가 생긴다

40초

"create-project 로 'settle helper' 만들어 줘"

1. 이름에서 폴더·데이터베이스 이름을 정하고, 같은 이름이 이미 있으면 멈춘다
2. 서버에서 비어 있는 자리(포트)를 하나 잡는다
3. 표준 골격으로 프로젝트를 만든다 — 계정 기능, 견본 업무 앱, 디자인, AI 작업 규칙까지 파일 212개
4. GitHub 에 비공개 저장소를 만들어 연결하고, AI 가 이 프로젝트를 기억할 공간을 이어 둔다

2 명령 한 줄로 실행 준비가 끝난다

1분 26초

`./setup.sh` — 사람이 넣는 값은 DB 비밀번호 하나

1. 프로젝트 전용 파이썬 환경을 만들고 필요한 프로그램을 설치한다 (시간 대부분)
2. 설정 파일을 만든다 — 암호 키는 자동으로 만들고, DB 비밀번호는 실제로 접속해 본 뒤 저장한다
3. 데이터베이스와 표를 만든다
4. 관리자 계정을 만든다 — 아이디는 `settle_helper_admin` 처럼 프로젝트 이름을 따서, 비밀번호는 무작위로 만들어 한 번만 보여 준다
5. 첫 버전을 저장하고 GitHub 에 올린다

3 한 번 켜면 실제 주소가 열린다

1분 18초

"/dev"

1. 주소 `settle-helper-dev.brie.co.kr` 을 등록한다
2. HTTPS 인증서에 이 주소를 추가하고, 웹서버가 이 프로젝트로 연결되게 한다
3. 이 주소로 들어오는 요청을 받도록 프로젝트 설정을 고친다
4. 서버를 켜고, 바깥에서 주소가 실제로 열리는지 확인한다
5. 접속 주소를 프로젝트 문서에 적어 둔다 — 다음 사람과 다음 AI 가 찾을 수 있게

3분 24초
한 문장 → HTTPS 주소 (실측)

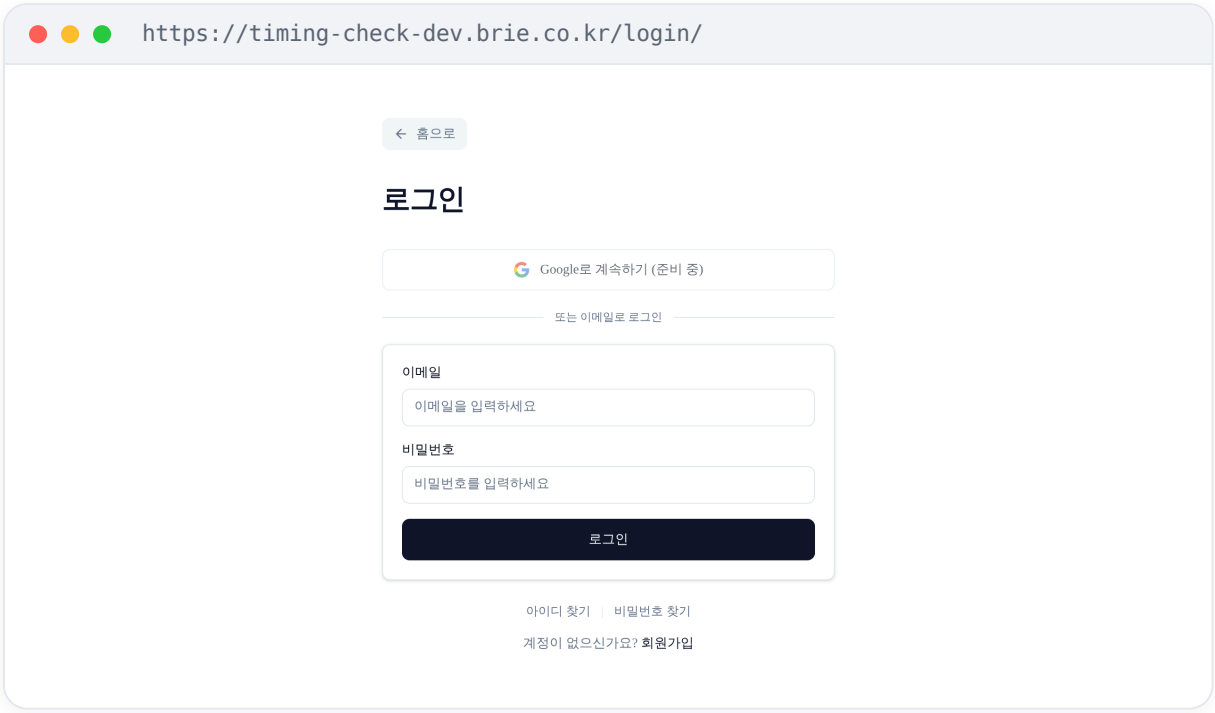
11 → 3
사람이 거치는 단계 (손으로 하면 11 단계)

1개
사람이 입력하는 값 (DB 비밀번호)

실제로 열린 화면

업무 기능이 하나도 없어도 이미 쓸 수 있는 앱이다

실측 때 만든 프로젝트를 바로 열어 본 화면이다. 업무 기능은 아직 하나도 만들지 않았지만, 가입·로그인부터 견본 업무 화면까지 이미 동작한다. 사람이 할 일은 여기에 "우리 업무"를 얹는 것뿐이다.



실제 화면 생성 직후의 로그인 화면 — 회원가입·아이디/비밀번호 찾기까지 들어 있다 (Google 로그인은 키를 넣으면 켜진다)



실제 화면 견본 업무 앱 — 검색·목록·새로 만들기. 내 데이터만 보인다 (목록의 세 건은 캡처용으로 넣은 예시 데이터)

01+

출발점 안 들여다보기

3분 만에 생긴 것은 무엇으로 되어 있나

한 문장으로 만든 프로젝트는 **standarda-template** 이라는 표준 골격에서 나온다. 이 골격이 어떤 기술로 되어 있고, 계정·데이터·화면·파일·공통 기능을 어떤 약속으로 정해 두었는지 여섯 쪽에 걸쳐 본다.

1. **스택** — 화면부터 데이터베이스까지 무엇으로 만드나
2. **회원가입·로그인** — 계정 흐름이 어떻게 정해져 있나
3. **데이터·API·관리 화면** — 업무 데이터를 다루는 약속
4. **화면과 JS** — 화면 한 장을 만드는 약속
5. **정적 파일** — 코드와 파일을 어디에 두고 어떻게 내보내나
6. **utils** — 자주 하는 일을 함수 하나로

출발점 안 ① 스택

널리 쓰는 기술만으로, 한 사람이 화면부터 DB 까지 다룬다

화면·서버·데이터베이스를 따로 다른 기술로 나누지 않는다. 서버 하나(Django)가 화면도 그리고 API 도 내보내므로, 업무를 아는 한 사람이 AI 와 함께 처음부터 끝까지 고칠 수 있다.

층	무엇으로	왜 이렇게
화면	Django 템플릿 + HTML·CSS·JavaScript 프레임워크 없음 · 디자인 시스템 `base.css`	빌드 도구 없이 파일을 고치면 바로 반영된다. 배울 것이 적다
서버	Python 3.11 · Django 4.2 · Django REST Framework 3.14	화면과 API 를 한 서버에서. 계정·관리 화면이 기본으로 딸려 온다
데이터베이스	PostgreSQL	프로젝트마다 자기 DB 를 따로 가진다
AI	Claude · GPT · Gemini, LangChain · LangGraph 공용 부품 `standarda-core` 를 통해 호출	모델을 바꿔도 코드를 거의 고치지 않는다 (벽 2 에서 자세히)
설정	개발은 `.env`, 운영은 `.env.production` — 서버가 스스로 골라 읽는다	같은 코드로 개발·운영을 오간다
운영	gunicorn + 웹서버	배포 명령 하나로 올린다

함께 들어 있는 것

- 엑셀·이미지 처리(openpyxl, pandas, Pillow)
- 브라우저 자동화(Playwright) — 화면 캡처 매뉴얼·자동 테스트
- 기본 시간대 서울, 날짜는 시간대를 지켜 저장

버전을 고정하는 방식

- 처음엔 넓은 범위로 설치하고, 개발 서버에서 실제 설치된 버전을 고정해 저장한다
- 공용 부품 standarda-core 는 버전 번호(태그)로 고정한다 — 올릴 때만 바뀐다

출발점 안 ② 회원가입·로그인

계정 흐름 다섯 가지가 처음부터 한 세트로 동작한다

벽 1 의 "로그인 만들어 줘" 뒤에 숨은 질문들에 대해, 출발점은 아래처럼 답을 정해 두었다. 아이디는 이메일이다.

회원가입 이메일 입력 → 6자리 인증번호 메일 → 3분 안에 입력 → 비밀번호·회사명(선택) → 로그인 화면으로

로그인 이메일·비밀번호 → 공유받은 링크로 왔다면 원래 보려던 화면으로, 아니면 첫 화면으로

아이디 찾기 이메일 인증번호 확인 → 가입한 아이디 표시

비밀번호 찾기 이메일 인증번호 확인 → 새 비밀번호를 직접 입력 → 로그인 화면으로

로그아웃 버튼 하나 → 로그인 화면으로

함께 정해진 것	출발점의 답
비밀번호 규칙	8자 이상, 영문·숫자·키보드 기호 각 1자 이상. 화면이 안내하는 규칙과 서버가 검사하는 규칙이 같고, 둘이 같은지 자동 테스트가 확인한다
로그인 없이 보이는 화면	첫 화면·로그인·가입·아이디 찾기·비밀번호 찾기 5개. 나머지는 로그인해야 열린다
프로필	계정이 어떤 방법으로 생기든(가입·관리자 도구·명령어) 함께 자동으로 만들어진다
관리자 계정	프로젝트를 만들 때 `이름_admin` 으로 자동 생성, 비밀번호는 무작위로 한 번만 보여 준다
메일 발송	Google 메일(SMTP)로 보낸다. 설정이 빠지면 "메일 설정이 아직 안 됐다"고 화면에 원인을 알려 준다
Google 계정	Google 로그인(첫 로그인 시 자동 가입)과, 사용자별 Gmail·Drive 사용 동의를 서버에 갖춰 두었다. 화면 버튼은 Google 키를 넣은 뒤 켠다

출발점 안 ③ 데이터·API·관리 화면

모든 업무 데이터가 견본 하나와 같은 모양으로 만들어진다

출발점에는 목록·상세·등록·수정·삭제를 모두 갖춘 **견본 업무 앱**이 들어 있다. 새 업무 앱은 이 견본을 통째로 복사해 시작하므로, 데이터 구조·API·관리 화면이 프로젝트마다 같은 약속을 따른다.

데이터(모델) — 모든 표가 가진 기본 칸

칸	뜻
`slug`	주소에 쓰는 고유 이름. 저장할 때 자동으로 만들어진다
`name`	이름
`owner`	만든 사람. 화면·API 는 "내가 볼 수 있는 것"만 꺼내는 한 가지 방법(`visible_to`)으로만 조회한다
`active`	삭제하면 지우지 않고 숨긴다 — 관리 화면에서 되살릴 수 있다
`timestamp` · `updated`	만든 시각 · 마지막으로 고친 시각 (이름을 모든 앱에서 통일)

API — 주소 다섯 개로 끝난다

요청	하는 일
`GET /api/앱/?q=…&page=…`	목록 — 검색, 20개씩 페이지
`POST /api/앱/`	등록 — 만든 사람은 로그인한 사람으로 자동 지정
`GET · PATCH · DELETE /api/앱/이름/`	상세 · 일부 수정 · 숨김 삭제

응답은 늘 같은 모양 — 성공 `{"success": true, "data": …}` , 입력 오류 `{"success": false, "errors": …}` . 화면 쪽 JS 는 이 모양 하나만 알면 된다.

화면 주소도 같은 규칙

- `/앱/` 목록 · `/앱/new/` 등록
- `/앱/이름/` 상세 · `/앱/이름/edit/` 수정
- 등록과 수정은 같은 화면을 쓴다

관리 화면(Django admin)

- 표마다 목록 칸·검색·"숨김 여부" 필터
- 날짜는 서울 시각 `YYYY.MM.DD`
- 개발자 없이 데이터를 찾아보고 고칠 수 있다

출발점 안 ④ 화면과 JS

화면 한 장 = 템플릿 하나 + JS 하나, 부품은 이미 만들어져 있다

화면마다 파일 구성이 같고, 버튼·입력창·표·모달 같은 부품은 디자인 시스템에 이미 들어 있다. 그래서 새 화면은 부품을 조립하고 그 화면의 동작만 쓰면 된다.

`templates/앱/화면.html`	화면 틀
`static/js/앱/화면.js`	그 화면의 동작 — 화면 하나에 JS 하나
`static/css/앱/화면.css`	그 화면에만 필요한 모양 (필요할 때만)

디자인 시스템 `base.css` 에 들어 있는 부품

색 체계(변수) · 뒤로 가기 · 페이지 머리말 · 버튼 4종 + 처리 중 표시 · 입력 폼·필터 · 진행률 · 결과 요약·목록 · 상태 배지 5종 · 표 · 빈 화면 안내 · 모달 · 정보 영역 · 파일 링크 · 애니메이션 · 모바일 대응

색은 직접 값 대신 변수로 쓴다 — 회사 색으로 바꿀 때 한 곳만 고치면 된다.

모든 화면이 함께 쓰는 `api.js`

함수	해 주는 일
`apiFetch()`	서버 호출 한 줄. 요청 형식·인증 토큰을 알아서 붙이고, 로그인이 풀렸으면 로그인 화면으로 보내고, 서버의 오류 문구를 꺼내 준다
`escapeHtml()`	사용자가 입력한 글자를 화면에 그대로 안전하게 보여 준다
`formatDateTime()`	날짜를 `2026.11.11 13:50` 형식으로 통일
`goToLogin()`	지금 보던 화면을 기억한 채 로그인 화면으로

화면 동작은 늘 같은 순서로 쓴다 — 입력 확인 → 버튼을 "처리 중"으로 → 서버 호출 → 결과 그리기 → 버튼 되돌리기.

출발점 안 ⑤ 정적 파일

코드와 파일을 나눠 두고, 개발과 운영에서 내보내는 길만 다르다

CSS·JS·이미지 같은 정적 파일과 사용자가 올린 파일은 코드 폴더 밖, 프로젝트 폴더 옆에 따로 모인다. 코드를 새로 받아도 올린 파일이 섞이거나 사라지지 않는다.

```
~/settle_helper/                ← 프로젝트 (파이썬 환경 포함)
    └── src/                    ← 코드 (GitHub 에 올라가는 곳)
          ├── static/           ← 내가 만든 CSS · JS · 이미지
          └── templates/
~/settle_helper_static_cdn/     ← 코드 밖, 프로젝트 옆
    ├── static/                 ← 운영용으로 모은 정적 파일
    └── media/                  ← 사용자가 올린 파일
```

	개발 서버	운영 서버
정적 파일	개발 서버가 `src/static/` 을 직접 내보낸다. 고치면 새로고침만으로 반영	배포할 때 한곳(`_static_cdn/static`)으로 모으고, 웹서버가 직접 내보낸다
파일 이름	그대로	내용이 바뀌면 이름에 표시가 붙어, 사용자 브라우저가 옛 파일을 붙잡고 있지 않는다
올린 파일	개발 서버가 내보낸다(로그인 필요)	웹서버가 `_static_cdn/media` 에서 직접 내보낸다
모으는 작업	필요 없다	배포 명령이 매번 알아서 한다

폴더는 프로젝트를 만들 때(`setup.sh`) 자동으로 생긴다. 출발점에 들어 있는 업로드 칸은 프로필 이미지 하나이며, 날짜별 폴더에 겹치지 않는 이름으로 저장된다.

출발점 안 ⑥ utils

프로젝트마다 반복되는 일은 함수 하나로 끝낸다

`utils/` 에는 거의 모든 프로젝트가 똑같이 필요로 하는 일을 미리 만들어 둔 함수가 모여 있다. 화면과 API 를 만들 때는 이 함수들을 가져다 쓰고, 같은 코드를 다시 쓰지 않는다.

함수(파일)	하는 일	그래서 덜어 주는 것
`login_required`	로그인해야 열리는 화면을 기본으로 만든다. 열어 둘 화면만 목록에 적는다	화면마다 "로그인 확인" 코드를 넣지 않는다
`api_auth`	로그인이 풀린 API 요청에 "로그인 필요"(401)로 답한다	화면 JS 가 이 답을 보고 로그인 화면으로 보낸다 (`apiFetch`)
`validator_utils`	비밀번호 규칙, 빈 값·중복·존재 여부, 3분 시간 제한, 두 값 일치 등 입력 확인	가입·찾기 화면의 확인 규칙을 한 곳에서 관리
`accounts_utils.send_email`	Google 메일로 HTML 메일 발송. 텍스트판 함께 첨부, 한글 발신자 이름 처리	인증번호·알림 메일을 한 줄로 보낸다
`generator_utils`	겹치지 않는 무작위 이름(slug) 생성	모든 데이터의 주소용 이름을 자동으로
`error_data` 외	오류 번호 목록, 권한 확인, 응답 형식 등 계정 기능이 쓰는 보조 함수	계정 기능 안에서 쓰인다

새 업무 앱은 견본 방식을 따른다

업무 앱(견본 앱)은 Django·DRF 가 기본으로 주는 기능만으로 만들어져 있고, 위 함수 중 로그인 관련 함수와 이름(slug) 생성 함수만 가져다 쓴다. 계정 기능 전용 보조 함수는 계정 앱 안에서만 쓰인다 — 새 앱을 만들 때 따라야 할 기준은 하나, 견본 앱이다.

부록 — 실측 기록

3분 24초는 이렇게 쟀다

항목	내용
일시·장소	2026-11-11, Brie 개발 서버(brie-box). 관리자 계정으로 실행
방법	3쪽의 세 단계를 사람이 하듯 그대로 실행하고 단계마다 시작·끝 시각을 기록했다. 1·3단계는 Claude Code 에 말로 요청했고, 2단계는 `./setup.sh` 에 DB 비밀번호를 넣어 실행했다
끝난 기준	바깥에서 주소 `https://timing-check-dev.brie.co.kr/` 가 정상 응답(HTTP 200)한 시각
포함하지 않은 것	처음 쓰는 사람이 한 번만 하는 준비 — 서버 접속, GitHub·Claude 로그인 (교육 첫 시간에 함께 한다)
뒷정리	측정에 쓴 프로젝트·저장소·데이터베이스·주소는 측정 직후 모두 지웠다

02

벽 2 · AI 부품

AI 기능은
직접 만들지 않는다

출발점을 열었다면 다음은 "우리 업무"다. 문서를 읽고, 엑셀을 정리하고, 메일을 보내고, 결과를 시트에 적는 일이다. 이 일들에 필요한 AI 기능을 매번 새로 만들면 처음 하는 사람은 그 자리에서 멈춘다. standarda 에서는 검증된 **부품**을 꺼내 쓴다.

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 두 번째 장치.

벽 — 업무 한 줄 뒤에 숨은 질문

"이 PDF 읽어서 정리해 줘" 한 줄에 정해야 할 것이 줄줄이 따라온다

업무 자동화의 대부분은 문서를 읽고, AI 에게 묻고, 결과를 어딘가에 남기는 일이다. 말로 시키면 AI 가 코드를 써 준다. 하지만 그 코드가 실제 업무에서 매일 돌아가려면 아래 질문들에 먼저 답해야 한다.

"이 PDF 읽어서 정리해 줘" → 그 뒤에 따라오는 질문들

어느 AI 를 쓰나
Claude? GPT? Gemini? 키는 어디에 두고, 나중에 바꾸려면 코드를 얼마나 고쳐야 하나

글자로 읽나, 그림으로 보나
글자로 읽으면 빠르고 싸지만 표·도장·손글씨를 놓친다. 그림으로 보면 정확하지만 비싸다

엑셀·한글·워드는
파일 형식마다 읽는 방법이 다르다. 한글(HWP)은 글자부터 깨지기 쉽다

AI 답을 프로그램이 읽을 수 있나
표 형식(JSON)으로 달라고 해도 앞뒤에 설명이 붙어 온다. 그대로 쓰면 프로그램이 멈춘다

비용은 얼마나 드나
같은 지시문을 매번 보내는데 아낄 방법은? 이번 달 얼마 썼는지는 어떻게 아나

결과는 어디에 남기나
구글 시트에 쓰고, 드라이브에 올리고, 메일로 알리려면 각 서비스 연결부터

녹음 파일은
회의 녹음이 길면 한 번에 보낼 수 없다. 나눠 보내고 다시 이어 붙여야 한다

모델이 새로 나오면
새 버전으로 바꾸는 순간 잘 되던 호출이 실패하지는 않나

벽 1 과 같다. AI 는 질문마다 그럴듯한 답을 골라 만들어 준다. 처음 하는 사람은 그 답이 맞는지 알 수 없고, 질문마다 찾아보고 공부하거나, 업무에 쓰는 날 막히고 나서야 안다.

standarda 의 장치

답이 정해진 부품 52개를, AI 가 알아서 꺼내 쓴다

앞 쪽 질문들의 답은 공용 부품 묶음 **standarda-core** 에 함수로 들어 있다. 그리고 출발점의 AI 작업 규칙이 "AI 기능은 core 부품으로만 만든다"고 정해 두었다. 그래서 사람은 부품 이름을 몰라도 된다.

① 사람이 말한다
"거래처 PDF 를 읽어서 금액을 시트에 정리해 줘"

→

② AI 가 규칙을 읽는다
"PDF 는 core 의 읽기 부품으로, 시트는 core 의 시트 부품으로"

→

③ 부품을 조립한다
PDF 읽기 → AI 에 묻기 → 답 정리 → 시트에 쓰기

부품 묶음 — 공개 함수 52개 + 구글 서비스 연결 43개

AI 연결
Claude·GPT·Gemini·OpenRouter. 모델은 설정 한 줄로 바꾼다

문서 읽기
PDF·엑셀은 글자 또는 그림으로, 한글·워드·RTF, 구글 시트

AI 답 다루기
답에서 글자만 꺼내기, 표 형식(JSON)으로 바로 쓰기

업무 도구
Gmail·구글 시트·드라이브·문서, 사용자별 구글 계정 연결

메일·음성·발표
알림 메일 꾸미기, 회의 녹음 받아쓰기, 슬라이드 PDF 만들기

비용·검증
반복 지시문 캐시, 사용량·비용 집계, 결과 숫자 대조

앞 쪽 질문	부품이 정해 둔 답
어느 AI 를 쓰나	함수 하나로 연결. 모델 이름은 설정 파일에 — 코드는 그대로
글자로 읽나, 그림으로 보나	두 방법이 모두 부품으로 있다. 문서 성격에 맞춰 고르기만 한다
AI 답을 프로그램이 읽을 수 있나	앞뒤 설명을 걷어내고 표 형식만 꺼내는 함수 — 가장 많이 쓰이는 부품
비용은 얼마나 드나	반복 지시문은 자동으로 캐시, 구간별 사용량·비용을 모아 준다
모델이 새로 나오면	새 모델의 차이는 부품이 흡수한다 (다음 쪽)

이 장치가 "스스로 하게" 만드는 이유

한 곳에서 고치면,
모든 프로젝트가 함께 고쳐진다

AI 는 몇 달마다 새 모델이 나오고, 그때마다 예전엔 되던 것이 갑자기 안 된다. 프로젝트마다 AI 코드를 따로 갖고 있으면 그 원인을 각자 찾아야 한다. standarda 에서는 부품 한 곳을 고치고, 각 프로젝트는 부품 버전만 올린다.

실제로 겪은 일	원인	부품이 정한 답
새 Claude 모델·SDK 로 바꾸자 **AI 호출이 전부 실패**	새 버전이 예전 설정값 하나를 더 이상 받지 않았다	부품이 모델을 보고 그 값을 빼고 보낸다 — 한 번 고쳐 전 프로젝트 반영
같은 지시문을 반복해서 보내는데 **캐시 적중률 3% 미만** — 비용이 그대로 나감	캐시 표시를 붙이는 위치가 틀렸다	AI 도구 호출 뼈대가 캐시 표시를 자동으로 붙인다
Gemini 새 버전에서 **답을 글자로 못 읽음**	답이 글자가 아니라 여러 조각의 목록으로 왔다	어느 모델의 답이든 글자 하나로 맞춰 주는 함수
알림 메일의 **줄바꿈이 사라짐** — 두 번 반복	메일 본문을 만들 때마다 각자 다르게 처리	알림 메일 HTML 을 만드는 부품 하나로 통일
회의 녹음 **받아쓰기 실패**	파일 형식·특수 문자 처리	녹음을 표준 형식으로 바꾸고, 나눠서 동시에 받아쓴 뒤 정리하는 부품

잘 만든 것은 부품으로 올라온다

한 프로젝트에서 검증된 기능은 core 로 올라와 모두의 부품이 된다. 슬라이드 PDF 만들기, 회의 녹음 받아쓰기, 결과 검증 함수가 그렇게 올라왔다. core 는 2026년 2월 이후 버전 30개를 냈고, 이 서버의 프로젝트 13개가 core 를 버전으로 고정해 쓰고 있다.

처음 하는 사람은 AI 의 변화를 쫓지 않아도 된다

새 모델에 맞추는 일, 비용을 아끼는 일, 파일 형식마다 읽는 법을 찾는 일은 부품이 맡는다. 사람은 업무 흐름에만 집중한다.

02+

부품 안 들여다보기

standarda-core 에는 무엇이 들어 있나

공용 부품 묶음 **standarda-core** 는 모든 프로젝트가 설치해 쓰는 파이썬 패키지다. 부품 21개 파일에 공개 함수 52개, 구글 서비스 연결 43개가 들어 있다. 다섯 쪽에 걸쳐 묶음별로 본다.

1. **AI 연결** — 모델 4곳을 같은 방법으로
2. **문서 읽기와 AI 답 다루기** — PDF·엑셀·한글·시트, 그리고 답 정리
3. **업무 도구·메일·음성** — 구글 서비스, 알림 메일, 받아쓰기
4. **비용·에이전트·검증** — 아끼고, 돌리고, 확인하는 부품
5. **버전과 약속** — 고쳐도 쓰는 쪽이 깨지지 않게

부품 안 ① AI 연결

모델 네 곳을 같은 방법으로 부르고, 바꿀 때는 설정만 고친다

어느 회사의 AI 든 함수 하나로 연결한다. 코드에는 "AI 를 쓴다"만 적고, 어떤 모델을 쓸지는 설정 파일이 정한다.

함수	연결하는 곳	기본 모델 (설정으로 변경)
`get_anthropic_llm()`	Claude	`claude-sonnet-5-5`
`get_openai_llm()`	GPT	`gpt-6-luna`
`get_google_llm()`	Gemini	`gemini-3.8-flash`
`get_openrouter_llm()`	OpenRouter — 여러 회사 모델을 한 곳에서	쓸 때 지정
`get_openai_embedding()`	문장을 숫자로 바꾸기(검색용)	`text-embedding-3-small`

모델을 정하는 순서

1. 코드에 적은 값
특정 단계만 다른 모델을 쓸 때

→

2. 설정 파일(.env)
보통은 여기서 정한다 — 바꿔도 코드는 그대로

→

3. 부품의 기본값
아무것도 안 정하면

모델이 바뀔 때마다 생기는 차이를 부품이 흡수한다

차이	부품이 하는 일
새 Claude 모델은 예전 설정값(temperature)을 받지 않는다	모델·SDK 버전을 보고 그 값을 빼고 보낸다
생각하는 모델은 답이 길다	답 길이 상한을 넉넉히 기본으로 둔다(설정으로 변경)
Gemini 새 버전은 답이 여러 조각으로 온다	답을 글자 하나로 맞추는 `response_text()`
키가 없으면	프로그램을 멈추지 않고, 경고를 남기고 "연결 없음"을 돌려준다

모든 연결에 응답 대기 시간·재시도 횟수를 정할 수 있고, 호출 기록은 추적 도구(LangSmith)에 자동으로 남는다 — 어느 단계에서 무엇을 묻고 무엇을 받았는지 나중에 볼 수 있다.

부품 안 ② 문서 읽기와 AI 답 다루기

문서는 성격에 맞는 길로 읽고, AI 답은 바로 쓸 수 있는 모양으로 받는다

PDF 와 엑셀에는 읽는 길이 두 가지씩 있다. 문서가 깔끔한 글자 위주면 글자로, 표·도장·스캔본이면 그림으로 읽는다. 어느 쪽이든 함수 하나다.

문서	글자로 읽기 — 빠르고 싸다	그림으로 보기 — 표·스캔본도 정확하게
PDF	`send_pdf_to_text_llm()` 글자를 뽑아 AI 에 보낸다	`send_pdf_to_vision()` 쪽마다 이미지로 바꿔 AI 가 본다 (기본 15쪽까지)
엑셀	`send_excel_to_text_llm()` 셀 값을 글자로 뽑아 보낸다	`send_excel_to_vision()` PDF 로 바꾼 뒤 그림으로 본다 — 서식·병합 셀이 많을 때
구글 시트	`send_sheet_to_llm()` — 시트 내용을 그대로 AI 에 보낸다	
기타 문서	`extract_text_from_document()` — 텍스트·마크다운·워드·PDF·**한글(HWP·HWPX)**·RTF·ODT 에서 글자를 뽑는다. 한글 인코딩이 깨지지 않게 여러 방식을 차례로 시도한다	

글자 인식(OCR) 프로그램은 따로 두지 않는다 — 스캔본도 AI 가 그림으로 읽는다.

AI 답 다루기 — 가장 많이 쓰이는 부품

함수	하는 일
`parse_json_from_llm()`	AI 답에서 앞뒤 설명을 걷어내고 표 형식(JSON)만 꺼낸다. 프로젝트 코드에서 가장 많이 불리는 부품
`response_text()`	어느 모델의 답이든 글자 하나로 맞춘다

```
pdf  = "거래처_계약서.pdf"
answer = send_pdf_to_vision(pdf, "계약 금액과 기간을 JSON 으로", get_anthropic_llm())
data   = parse_json_from_llm(answer)      # → {"금액": ..., "기간": ...}
```

사람이 이 코드를 쓰지 않는다 — "이 계약서에서 금액과 기간을 뽑아 줘"라고 하면 AI 가 이 부품들로 조립한다.

부품 안 ③ 업무 도구·메일·음성

회사가 이미 쓰는 도구에 결과를 바로 남긴다

자동화의 결과는 새 화면이 아니라 늘 쓰던 곳에 남을 때 쓰인다. 구글 업무 도구 네 가지를 부품으로 연결해 두었다.

도구	기능 수	할 수 있는 일
구글 시트	21	값·수식 읽고 쓰기, 한꺼번에 고치기, 시트 복제·삭제·복사, 행 넣고 빼기, 셀·머리글 찾기, PDF 로 내보내기
구글 드라이브	11	폴더·파일 목록, 폴더 만들기, 올리기·내려받기, 복사·이동, HTML 을 구글 문서로 바꿔 올리기
Gmail	8	첨부 메일·알림·답장 보내기, 받은편지함·보낸 메일 조회, 첨부 내려받기, 답장 찾기
구글 문서	3	문서 내용 읽기 — 탭이 여러 개인 문서도

연결 방식은 두 가지 — 회사 계정 하나로 쓰거나, 사용자마다 자기 구글 계정으로 동의하고 쓴다(내 메일·내 드라이브를 대상으로 하는 업무).

알림 메일 — `render_email_html()`

- 머리글·본문·버튼이 갖춰진 HTML 메일을 함수 하나로
- 줄바꿈·특수 문자가 깨지지 않게 처리
- 메일 프로그램마다 모양이 같도록 꾸밈을 메일 안에 넣는다

회의 녹음 받아쓰기 — `transcribe_audio()`

- 녹음을 표준 형식으로 바꾸고
- 길면 잘게 나눠 동시에 받아쓴 뒤 순서대로 이어 붙인다
- 받아쓴 글에 섞인 쓸 수 없는 문자를 걸러 낸다

발표 자료 — `create_presentation()`

- 슬라이드 내용만 주면 16:9 HTML 과 PDF 를 만든다
- 보고서·결과 공유 자료를 자동으로 만들 때

필요할 때만 설치

- 발표·한글·음성·모델 최적화 부품은 쓰는 프로젝트만 추가 설치한다
- 기본 설치는 가볍게 유지된다

부품 안 ④ 비용·에이전트·검증

아끼고, 돌리고, 확인하는 일도 부품으로 해 둔다

AI 를 매일 업무에 쓰기 시작하면 세 가지가 남는다. 비용이 얼마나 드는지, 여러 단계를 스스로 처리하는 에이전트를 어떻게 돌리는지, 결과를 어떻게 믿을지.

비용

부품	하는 일
지시문 캐시	매번 같은 긴 지시문은 한 번만 제값을 내고 다음부터는 싸게 보낸다. 캐시 표시를 붙이는 위치를 부품이 정해 둔다
`track_usage()`	"이 작업 한 번에 얼마"를 잰다 — 그 구간의 모든 AI 호출 사용량을 모아 비용으로 바꾼다(모델별 가격표 내장)
단계별 모델 고르기	여러 단계로 된 작업에서 단계마다 어떤 모델 조합이 같은 정확도에 가장 싼지 실험한다(AgentOpt)

에이전트 — `build_tool_loop_graph()`

AI 가 도구(검색·조회·계산 등)를 골라 쓰고, 결과를 보고, 다시 판단하는 반복 구조의 뼈대다. 도구를 순서대로 실행하고, 대화가 길어져도 캐시를 이어 붙이고, 반복 횟수 상한과 마지막 답 꺼내기까지 정해져 있다. 프로젝트는 도구만 만들어 끼운다.

결과 검증 — 3단계

① 출력 점검
처리한 건수, 빈 값, 튀는 값

→

② 기록과 대조
시스템에 남은 값과 하나씩 비교 — 숫자는 형식이 달라도 같게 본다

→

③ 원본 확인
일부를 골라 원본 문서와 직접 비교

AI 가 "다 처리했다"고 말하는 것과 실제로 맞게 처리된 것은 다르다. 검증 부품은 그 차이를 숫자로 보여 준다 — 사람은 다른 것만 보면 된다.

부품 안 ⑤ 버전과 약속

부품을 고쳐도 쓰는 쪽은 한 줄도 바꾸지 않는다

여러 사람이 만든 여러 프로젝트가 같은 부품을 쓴다. 그래서 부품은 "쓰는 쪽 코드는 그대로, 버전만 올리면 좋아진다"는 약속을 지킨다.

약속	지키는 방법
버전으로 고정해서 쓴다	각 프로젝트는 core 를 버전 번호로 고정해 설치한다. 올리기 전까지는 아무것도 바뀌지 않는다 — 오래된 프로젝트도 그대로 돈다
있던 것은 지우지 않는다	함수와 입력값은 지우거나 이름을 바꾸지 않고, 새 옵션만 더한다. 꼭 합쳐야 하면 옛 이름을 남겨 둔다
테스트가 지킨다	자주 쓰는 함수 9개의 입력값 이름을 테스트로 묶어 두었다. 이름이 사라지면 자동 검사가 실패한다. 파이썬 두 버전·깨끗한 설치에서 매번 확인
바뀐 것은 기록한다	버전마다 무엇이 바뀌었는지, 쓰는 쪽이 할 일이 있는지, 어느 프로젝트에서 확인했는지를 팀 위키에 남긴다

약속을 문서로만 두었을 때 생긴 일

2026년 8월, 함수 두 개를 하나로 합치면서 옛 입력 방식 하나가 빠졌다. 그 방식으로 부르던 엑셀 그림 읽기가 모두 실패했고, 위치로만 부르던 프로젝트들은 멀쩡해서 몇 버전 동안 드러나지 않았다. 그날 바로 옛 방식을 되살리고, 자동 검사를 들였다 — **하루에 버전 세 개**. 그 뒤로 약속은 사람이 아니라 테스트가 지킨다.

버전 흐름 — 2026년 2월 ~ 11월, 버전 30개

시기	무엇이 들어왔나
2 ~ 4월	한 프로젝트의 공통 기능을 떼어 내 부품 묶음을 만들었다 — 문서 읽기, AI 연결
4 ~ 5월	구글 드라이브·Gmail·문서 부품 확장, 문서 글자 뽑기
6 ~ 7월	캐시·비용 집계, 에이전트 뼈대, OpenRouter
7 ~ 8월	프로젝트에서 검증된 기능을 부품으로 — 결과 검증, 발표 자료, 한글, 사용자별 구글 연결, 받아쓰기, 알림 메일
8월	약속을 테스트로 — 자동 검사 도입
9 ~ 11월	새 모델 세대(Claude 5 등)에 맞춤

03

벽 3 · AI 에게 어떻게 시키나

코드를 읽지 못해도 무엇을 만들지는 내가 정한다

출발점과 부품이 있어도, 결국 일은 AI 에게 말로 시킨다. 처음 하는 사람이 여기서 막히는 이유는 AI 가 못해서가 아니라, **AI 가 만든 것이 맞는지 판단할 방법이 없어서**다. standarda 는 AI 가 일하는 방식 자체를 정해 둔다.

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 세 번째 장치.

벽 — "기능 만들어 줘" 뒤에 숨은 질문

AI 에게 일을 맡기는 순간,
맡긴 사람이 답해야 할 것이 생긴다

AI 는 시키면 만든다. 문제는 그다음이다. 결과를 판단하고, 이어서 고치고, 남에게 넘기는 일은 여전히 사람의 몫인데, 코드를 읽지 못하면 그 판단을 할 근거가 없다.

"견본 앱에 결재 상태 칸 추가해 줘" → 그 뒤에 따라오는 질문들

내가 원한 게 맞나
AI 는 "다 했다"고 말한다. 코드를 읽지 못하는 사람은 무엇으로 맞다고 판단하나

어디서 내가 확인하나
매 단계 물어보면 일이 끝나지 않고, 안 물어보면 엉뚱한 것이 나온다

다른 기능은 멀쩡한가
이번에 고친 것이 어제까지 되던 화면을 망가뜨리지 않았나

동시에 여러 개를 시키면
두 기능을 함께 맡기면 서로의 작업을 덮어쓰지 않나

내일의 AI 는 기억하나
오늘 왜 이렇게 만들었는지, 다음 대화의 AI 는 모른다

오류가 나면
AI 가 원인을 확인하지 않고 그럴듯한 추측으로 고치지는 않나

무엇을 남겨야 하나
끝난 뒤 동료가 이어받으려면 어떤 기록이 있어야 하나

매번 같은 품질인가
사람마다, 날마다 AI 에게 시키는 방식이 달라 결과도 들쭉날쭉하다

이 질문들은 코드 지식이 아니라 **일하는 방식**의 문제다. 숙련된 개발자는 경험으로 답하지만, 처음 하는 사람에게는 답할 근거가 없다.

standarda 의 장치

개발 요청은 늘 같은 순서로 돌고, 사람은 쉬운 말 요약을 한 번 승인한다

"만들어 줘", "고쳐 줘" 같은 개발 요청은 따로 말하지 않아도 `/develop` 이라는 정해진 흐름으로 처리된다. 흐름 안에서 사람이 하는 일은 한 번의 승인뿐이고, 나머지 확인은 AI 와 검증 에이전트가 한다.

1 **격리된 작업 공간을 연다**

요청마다 정본과 분리된 작업 공간(worktree)을 따로 만든다. 여러 기능을 동시에 시켜도 서로 덮어쓰지 않는다.

2 **관련 맥락을 한꺼번에 읽는다**

그 앱의 기존 스펙·코드·테스트·할 일 목록을 읽는다. 큰 작업이면 팀이 쌓아 둔 교훈 중 관련된 것을 따로 찾아 온다.

3 **사람이 한 번 승인한다** 사람

"무엇이 바뀌나"를 사용자 관점의 쉬운 말 5~7줄로, 바꿀 파일 목록과 함께 한 메시지(15줄 안팎)로 보여 준다. 질문을 여러 번 나눠 묻지 않는다.

> "○○ 화면에서 결재 상태를 대기·승인·반려 중에서 고를 수 있다"

4 **스펙을 두 층으로 남긴다**

승인한 내용을 `docs/specs/앱.md` 에 쓴다 — 위층은 비개발자용 요약, 아래층은 코드와 하나씩 대조할 수 있는 검증용 명세.

5 **정해진 규칙대로 만든다**

AI 작업 규칙의 체크리스트를 따른다 — AI 기능은 core 부품으로, 화면은 정해진 JS 방식으로, 오류는 숨기지 않게.

6 **스펙과 코드를 대조한다**

AI 가 정해진 점검표(`spec_check.md`)대로 스펙의 명제마다 코드를 확인한다. **하나라도 어긋나면 합치기가 막히고**, 고친 항목만 다시 확인한다. 화면 주소가 실제로 열리는지도 본다.

7 **점검하고 정본에 합친다**

AI 가 합치기 전 점검표(`merge_check.md`)로 미커밋 변경·필수 문서·다른 작업과 겹치는 파일·관련 테스트를 확인한다. 통과하면 정본에 합치고, 작업 공간을 정리하고, 할 일 목록에서 완료 처리한다.

오타·문구·설정값처럼 자명한 변경은 이 흐름 없이 바로 고친다. 회원가입·로그인 요청은 이미 있는 기능이므로 먼저 `/signup` 으로 확인한다.

이 장치가 "스스로 하게" 만드는 이유

판단은 사람이 쉬운 말로 하고, 확인은 시스템이 한다

앞 쪽의 질문들은 숙련자라면 경험으로 답한다. standarda 는 그 경험을 흐름과 규칙으로 바꿔, 처음 하는 사람도 같은 답을 갖고 일하게 한다.

질문	standarda 의 답
내가 원한 게 맞나	코드가 아니라 **쉬운 말 요약**을 보고 승인한다. 맞게 만들어졌는지는 그 요약에서 나온 검증 명세를 기준으로 AI 가 점검표대로 하나씩 확인한다
어디서 내가 확인하나	시작 전 **한 번**. 판단이 필요한 지점을 한곳에 모았다
다른 기능은 멀쩡한가	합치기 전 점검이 관련 테스트를 돌리고, 다른 작업과 겹친 파일을 알려 준다
동시에 여러 개를 시키면	요청마다 격리된 작업 공간. 끝나면 정본 하나로 합친다
내일의 AI 는 기억하나	스펙 파일과 프로젝트 메모리가 남는다. 다음 대화의 AI 는 그것부터 읽는다
오류가 나면	규칙이 "추측하지 말고 로그부터"를 정해 두었고, 로그를 먼저 읽는 에이전트가 있다
무엇을 남겨야 하나	스펙은 반드시, 버그 수정이면 원인·해결·교훈을 담은 이슈 문서 — 빠지면 합치기 전에 걸린다
매번 같은 품질인가	누가 시키든 같은 흐름·같은 규칙. 사람마다 다른 "시키는 법"이 하나로 정리된다

이 흐름은 실제로 쓰면서 다듬어졌다

- **스펙을 두 층으로** — 처음 스펙은 개발자용 명세뿐이었다. 비개발자가 승인할 수 있도록 맨 위에 쉬운 말 요약 층을 더했다.
- **"모든 개발 요청에 기본"** — 요청할 때만 쓰던 흐름을, 따로 말하지 않아도 적용되도록 바꿨다. 쓰는 사람이 흐름을 기억할 필요가 없다.
- **확인은 한 번으로** — 교육 리허설에서 단계마다 묻고 답하느라 시간이 걸렸다. 질문을 한 메시지로 모으고, 검증도 한 번 돌린 결과를 합치기 단계가 그대로 쓰도록 줄였다.
- **검증은 메인 AI 가 직접** — 2026-11-13 교육 리허설에서 검증을 맡긴 하위 에이전트가 멈춰 7분을 기다렸다. 그날 바로 스펙 대조와 합치기 전 점검을 메인 AI 가 따르는 점검표로 바꾸고, 남은 하위 에이전트에는 "5분 안에 보고가 없으면 멈추고 메인이 직접" 규칙을 걸었다.

처음 하는 사람에게 필요한 것은 코딩 실력이 아니라 판단할 자리다

standarda 는 그 자리를 "쉬운 말 요약을 승인하는 한 번"으로 만들어 두었다. 다음 쪽부터는 이 흐름을 받치는 AI 작업 규칙, 스킬 17개, 에이전트 12개를 하나씩 본다.

03+

하네스 안 들여다보기

AI 가 일하는 방식은 무엇으로 정해져 있나

출발점에는 코드와 함께 **AI 가 일하는 방식**이 들어 있다. 이것을 하네스(harness, 마구)라고 부른다 — 힘센 말이 엉뚱한 데로 달리지 않게 채우는 장비처럼, AI 가 정해진 방향으로 일하게 한다.

구성	쉬운 말로	규모
AI 작업 규칙 `CLAUDE.md`	AI 가 매 대화를 시작할 때 자동으로 읽는 근무 규칙집	536줄, 16개 섹션
스킬	AI 에게 주는 업무 매뉴얼. "배포해 줘"처럼 말로 부르면 그 순서를 따른다	17개
에이전트	AI 가 일을 나눠 맡기는 전문 보조원. 대부분 읽기만 하고 결과를 보고한다	12개
작업 공간	같은 코드의 작업용 사본. 여러 작업이 동시에 돌아도 섞이지 않는다	요청마다 하나

AI 가 규칙을 어기려 할 때 기계가 멈추는 장치(hook)는 벽 4 에서 따로 다룬다.

하네스 안 ① AI 작업 규칙

AI 는 매번 같은 규칙집을 읽고 같은 방식으로 일한다

`CLAUDE.md` 는 AI 가 대화를 시작할 때마다 자동으로 읽는 문서다. 사람마다 AI 에게 다르게 설명하던 "이 프로젝트에서는 이렇게 한다"가 여기 한곳에 적혀 있어, 누가 시키든 AI 가 같은 기준으로 일한다.

섹션	무엇이 적혀 있나
팀 위키 참조	작업 전에 팀 위키부터 읽는다. 서버 사용법·포트 고르는 법
명령·구조·환경 설정	자주 쓰는 명령, 공통 앱 5개(견본 앱 포함), core 부품 목록, 설정 값 목록, 파일 저장 위치 규약
구글 연동·화면 규칙	시트·메일·드라이브·문서를 쓰는 법, 화면은 프레임워크 없이·공통 JS 로
코드 규칙 (128줄, 가장 큼)	반복된 실수를 규칙으로 만든 목록 — 아래
디버깅 규칙	추측하지 말고 증거부터 — 5단계
개발 흐름	개발 요청은 `/develop` 이 기본, 회원가입은 `/signup` 먼저
Git 규칙	정본은 하나, 작업은 사본에서, 끝나면 정본에 합치고 정리까지
문서 규칙	문서 종류별 위치, 버그 기록 양식과 "교훈을 어디까지 퍼뜨릴지" 분류
스킬·에이전트 규칙	만드는 위치, 이름표(description) 쓰는 법, 말로도 부를 수 있게, 되돌릴 수 없는 일은 실행 전 확인

코드 규칙에 들어 있는 것 — 한 번씩 겪은 일이 규칙이 됐다

- **AI 기능은 core 부품으로만** — 모델 연결은 `get_*_llm()`, AI 답은 `response_text()`·`parse_json_from_llm()`, PDF·엑셀은 글자/그림 중 골라서, 받아쓰기는 `transcribe_audio()`
- **날짜는 서울 시각으로** — 저장·표시를 정해진 함수로. 아니면 9시간이 어긋난다
- **오류를 조용히 넘기지 않는다** — 문제가 생기면 대충 다른 값으로 때우지 말고, 기록을 남기고 드러낸다
- **시트 범위는 그때그때 계산** — 행이 늘어도 빠지는 데이터가 없게
- **메일은 정해진 방식으로** — 주소·본문 만드는 함수를 하나로. 본문이 한 줄로 뭉개지는 일이 반복됐다
- **글자 맞추기로 분기하지 않는다** — "이 단어가 있으면 A" 대신 글 전체를 AI 에게 맡긴다
- **새 업무 앱은 견본을 복사해서** — 목록·상세·등록·수정·삭제를 한 번에

하네스 안 ① AI 작업 규칙 (이어서)

오류는 증거로 고치고, 교훈은 다음 프로젝트로 넘긴다

디버깅 규칙 — 5단계

1 **로그부터 본다**
"안 돼요"라는 말을 들으면 추측하기 전에 서버 기록을 읽는 에이전트(`debug-server-log`)에게 먼저 맡긴다.

2 **응답 내용을 본다**
"오류 400" 같은 숫자만 보고 단정하지 않는다. 서버가 돌려준 문장을 읽는다.

3 **같은 조건으로 재현했는지 확인한다**
내 쪽에서는 되는데 사용자 쪽에서만 안 되면, 조건이 다른 것이다(로그인 상태·브라우저 등).

4 **DB 를 직접 본다**
"저장됐을 것이다"가 아니라 실제로 들어 있는 값을 확인한다.

5 **다음을 위해 기록이 남게 한다**
오류가 나면 원인 추적 정보가 로그에 남도록 코드를 쓴다.

핵심 문장
디버깅의 품질은 AI 가 더 똑똑하게 추측하는 데서가 아니라, **AI 가 닿을 수 있는 진짜 증거의 양**에서 나온다.

Git 규칙 — 정본은 하나

- 작업은 늘 최신 정본에서 갈라진 **사본**에서 한다. 끝나면 정본에 합치고, 원격 저장소에 올리고, 사본을 지운다 — 사람이 따로 시키지 않아도.
- 정본은 앞으로만 나아간다. 되돌릴 때는 "되돌리는 기록"을 새로 쌓는다.

문서 규칙 — 교훈이 도는 고리

버그를 고치면 원인·해결·교훈을 담은 이슈 문서를 남기고, 그 교훈을 **어디까지 퍼뜨릴지** 네 층 중 하나로 고른다.

층	교훈이 가는 곳
1 코드 규칙	이 규칙집의 코드 규칙으로 — 앞으로 모든 AI 가 따른다
2 공통 · 3 분야별	팀 위키의 교훈 모음으로 — 다른 프로젝트에서 `lesson-finder` 가 찾아 준다
4 이 프로젝트만	이슈 문서에만 남긴다

하네스 안 ② 스킬 — 개발·운영 (1/2)

서버 켜기부터 회원가입까지, 자주 하는 일은 매뉴얼이 있다

스킬은 AI 에게 주는 업무 매뉴얼이다. 슬래시(`/dev`)로 불러도 되고 "서버 켜 줘"처럼 말로 해도 된다. 개발 흐름 `/develop` 은 앞에서 보았으므로, 나머지를 본다.

`/dev` 개발 서버를 켜고, 처음이면 외부 주소까지

언제 쓰나	"서버 띄워 줘", "dev 서버 켜 줘". 회원가입·메일·화면 점검 스킬이 서버가 꺼져 있으면 이걸 안내한다
하는 일	1. 포트와 외부 주소가 준비됐는지 **따로따로** 확인한다 2. 포트가 없으면 빈 포트를 골라 규칙집에 적는다 3. 외부 주소가 없으면 사람에게 묻고 도메인·인증서·웹서버 연결을 한 번에 처리한다 4. 그 주소로 들어오는 요청을 받도록 설정을 고친다 5. 서버를 켜고 주소와 기록 위치를 알려 준다
사람이 할 일	외부에 주소를 열지 정한다(Y/n)
결과물	실행 중인 서버, `https://이름-dev.brie.co.kr`, 규칙집의 접속 정보
왜 있나	포트만 미리 채워진 프로젝트에서 주소 설정이 통째로 건너뛰어진 일이 있었다 — 그래서 두 상태를 따로 본다

`/release` 가장 긴 매뉴얼 정본의 코드를 운영 서버에 올린다

언제 쓰나	"배포해 줘", "운영에 올려 줘". 정본에 합친 것만으로는 운영에 반영되지 않는다
하는 일	1. 처음이면 운영 서버에 프로젝트 자리를 만든다(DB·서비스·주소·인증서·설정 파일) 2. 이번에 올라갈 변경 목록을 보여 준다 3. 배포 명령 한 줄로 코드 받기·설치·DB 변경·정적 파일·재시작·상태 확인을 한다 4. 처음이면 운영 관리자 계정을 만들어 한 번만 보여 준다
사람이 할 일	상태 확인이 실패했을 때 어떻게 할지 정한다
결과물	운영 반영, 종료 보고(처음이면 운영 관리자 계정)
왜 있나	개발 서버와 운영 서버를 나눠 운영 데이터를 지킨다. 배포 중 되돌리기 장치는 **벽 5** 에서 자세히

`/signup` 회원가입은 새로 만들지 않고, 필요한 것만 고친다

언제 쓰나	"회원가입 만들어 줘", "가입 승인제로 해 줘", "회원가입이 안 돼". `/develop` 보다 먼저 쓴다
하는 일	1. 이미 있는 가입 기능을 진단한다(읽기만) 2. 정책 질문 5개를 쉬운 말로 한 번에 묻는다 — 누가 가입하나·로그인 방법·가입 항목·약관·회원 종류 3. 기본과 다른 것만 `/develop` 으로 고친다(관리자 승인제는 참고 구현 포함) 4. 메일 설정(`/email-setup`) → 브라우저로 직접 가입해 보기 → 운영 반영·고객 안내 문구
사람이 할 일	가입 정책에 답하고, 직접 가입·로그인·비밀번호 찾기를 해 본다
결과물	단계별 결과표, 정책 결정표, 고객 안내 문구
왜 있나	실제 고객 프로젝트에서 "회원가입 개발해 줘"를 받고 이미 있는 기능을 다시 만들 뻔했다(2026-11-06)

하네스 안 ② 스킬 — 개발·운영 (2/2)

메일과 운영 절차도 처음 하는 사람이 끝까지 갈 수 있게

`/email-setup` 가입·찾기 인증 메일이 실제로 도착하게 한다

언제 쓰나	"메일 설정해 줘", "인증 메일이 안 가". `/signup` 이 중간에 부른다
하는 일	1. 지금 코드·설정 상태를 확인한다 2. 옛 발송 코드가 있으면 바꿀 내용을 보여 주고 확인받아 교체한다 3. 보내는 도메인의 메일 설정(DNS)을 점검하고, 관리자에게 보낼 요청 문구를 만들어 준다 4. 구글 앱 비밀번호 발급을 안내한다 5. 테스트 메일을 보내 받은편지함·스팸함 도착까지 확인한다 6. 운영 서버 설정에도 반영한다
사람이 할 일	보낼 계정을 정하고 앱 비밀번호를 직접 설정 파일에 넣는다. 메일이 실제로 왔는지 본다
결과물	단계별 결과표, DNS 요청 문구, 남은 일 목록
왜 있나	"보냈다"와 "도착했다"는 다르다 — 발송 성공 메시지만 보고 끝냈다가 스팸함에 쌓인 일

`/runbook` 처음 하는 사람도 복사·붙여넣기로 끝까지 가는 절차서

언제 쓰나	"서버 셋업 절차 문서", "배포 절차 정리해 줘"
하는 일	1. 명령마다 어디서 실행하는지·전제 조건·채울 값·완료 확인을 붙인다 2. 원격에서 도는 명령, 되돌릴 수 없는 단계는 따로 표시한다 3. 경로·전제 → 단계 → 완료 기준 → 문제 해결 순서로 쓰고 스스로 검산한다
사람이 할 일	주제와 저장 위치를 정한다
결과물	절차서 문서
왜 있나	"이 명령을 어디서 실행하지?"라는 질문이 한 번이라도 생기면 실패한 절차서다

스킬이 서로를 부른다

회원가입 하나만 해도 `/signup` → `/develop` → `/email-setup` → `/dev` → `/release` 가 차례로 이어진다. 사람은 "회원가입 만들어 줘" 한 마디와 그 사이의 결정 몇 가지만 한다.

하네스 안 ③ 스킬 — 고객 산출물 (1/2)

만든 것을 고객에게 넘길 자료도 스킬이 만든다

업무 시스템은 만들고 끝나지 않는다. 쓰는 사람에게 사용법을, 확인하는 사람에게 테스트 방법을 넘겨야 한다. 이 자료들을 실제 코드와 화면을 기준으로 만든다.

`/client-usage-guide` 고객이 "어떻게 쓰는지" 이해하는 사용 가이드

언제 쓰나	"사용 가이드 만들어 줘"
하는 일	1. 기능 설명이 코드보다 낡았으면 먼저 최신으로 고친다 2. 고객이 하나의 기능으로 느끼는 단위로 묶는다 3. 묶음별 가이드를 쓴다 4. 실제 화면 캡처를 `ui-tester` 에게 맡긴다 — 기존 데이터만 쓰고 저장·전송은 누르지 않는다 5. Word 파일로도 만든다
사람이 할 일	대상 기능을 고르고 로그인 정보를 준다
결과물	가이드 md + Word, 화면 캡처, 목차·전체 개요 문서
왜 있나	기능 문서는 개발 단위로 잘게 쪼개져 있어 고객이 읽기 어렵다. 고객 배포본은 Word 인 경우가 많다

`/client-test-guide` 고객이 직접 따라 하며 합격을 체크하는 테스트 가이드

언제 쓰나	"고객 테스트 가이드 만들어 줘"
하는 일	1. 기능 문서와 코드의 마지막 수정 시각을 비교해, 코드가 새로우면 문서를 다시 쓴다 2. 기능마다 접속 방법 → 준비물 → 단계표(조작·입력·예상 결과) → 결과 체크 → 문제 생기면 3. 두 개 이상이면 목차를 만든다
사람이 할 일	대상 기능을 고른다
결과물	기능별 테스트 가이드, 목차
왜 있나	"세무 담당자도 이해할 수준"이 기준

`/manual` 실제 화면을 찍어 누를 곳을 표시한 슬라이드 매뉴얼

언제 쓰나	"사용자 매뉴얼 PDF 만들어 줘"
하는 일	1. 개발 서버를 켜고 화면을 자동으로 찍는다 2. 누를 곳에 주황 박스를 그려 슬라이드로 만든다 3. PDF 를 쪽마다 눈으로 검수하고 필요한 쪽만 다시 만든다
사람이 할 일	섹션별로 어느 화면을 찍을지 정한다(예시 1개가 들어 있다)
결과물	16:9 HTML·PDF 매뉴얼, 캡처 이미지
왜 있나	고객 데이터 대신 가짜 데이터로 찍는다

하네스 안 ③ 스킬 — 고객 산출물 (2/2)

문서·보고서·화면 점검까지, 형식이 정해진 일은 형식대로

/pdf-doc 화면과 무관한 고객 전달용 A4 문서

언제 쓰나	"안내서 PDF", "가입 가이드 PDF" — 이 소개 자료도 이 스킬로 만들었다
하는 일	1. 독자·목적·색을 정한다 2. A4 한 장 = 한 섹션으로 HTML 을 쓴다 3. 화면 그림은 직접 그리고 "참고용 예시 화면"이라고 표시한다 4. PDF 로 바꾸고 쪽마다 이미지로 넘침을 확인한다
결과물	PDF 와 원본 HTML
왜 있나	실제 캡처인 척하는 그림은 만들지 않는다

/create-docx 표·색 박스·이미지가 든 Word 문서

언제 쓰나	"Word 파일로 만들어 줘". 사용 가이드와 소개 문서 만들기가 이 부품을 쓴다
하는 일	1. 제목·문단·목록·표·색 박스·이미지를 받아 Word 로 조판한다 2. 이미지가 없으면 멈추지 않고 자리를 표시해 둔다
결과물	Word 파일

/feature-doc 만든 기능을 사실대로 설명하는 문서

언제 쓰나	기능이 완성됐을 때, 설명이 코드와 어긋났을 때
하는 일	1. 코드를 읽고(추측 금지) 무엇이 되는가·어떻게 동작하는가·무엇을 조심해야 하는가를 쓴다
결과물	`docs/features/기능.md`
왜 있나	"강력한", "손쉽게" 같은 꾸밈말은 쓰지 않는다

/screen-audit 모든 화면이 모바일·태블릿·PC 에서 깨지지 않았는지

언제 쓰나	"모바일에서 화면 다 깨졌는지 확인해 줘"
하는 일	1. 화면 폭 세 가지로 모든 페이지를 찍고, 숫자로 깨짐 후보를 고른다 2. 사진 묶음을 `screen-reviewer` 여러 명에게 동시에 나눠 눈으로 판정하게 한다 3. 깨졌다고 나온 화면은 다시 직접 본다 4. 심각한 순서의 목록을 만든다 — 고쳐 달라면 `/develop` 으로
사람이 할 일	점검할 주소와 테스트 계정을 적는다
결과물	화면별 판정 목록과 대표 이미지
왜 있나	숫자만으로는 놓치는 게 많아 눈 판정을 같이 쓴다

하네스 안 ④ 스킬 — 기록과 템플릿 순환

프로젝트에서 좋아진 것은 템플릿으로, 템플릿의 개선은 프로젝트로

한 프로젝트에서 찾은 더 나은 방법이 그 프로젝트에만 남으면 다음 프로젝트는 같은 시행착오를 다시 겪는다. 두 스킬이 출발점(템플릿)과 프로젝트 사이를 양쪽으로 잇는다.

`/sync-to-template` 프로젝트에서 좋아진 공통 자산을 출발점으로 올린다

언제 쓰나	"템플릿에 반영해 줘"
하는 일	1. 규칙집·스킬·공통 앱 등 공통 자산을 출발점과 비교해 차이를 표로 보여 준다 2. 항목마다 반영할지 묻는다 3. 프로젝트 고유값(포트·이름 등)을 자리표시로 바꿔 출발점에 넣는다 4. 변경 이력과 "교육 포인트"를 팀 위키에 남긴다
사람이 할 일	항목별로 반영 여부를 정한다. 관리자가 최종 반영한다
결과물	출발점 변경, 팀 위키 변경 이력
왜 있나	이후 새로 만드는 모든 프로젝트가 그 개선을 갖고 시작한다

`/sync-from-template` 이미 만든 프로젝트에 출발점의 최신 개선을 골라 가져온다

언제 쓰나	"최신 템플릿 반영해 줘"
하는 일	1. 출발점과 프로젝트를 비교한다 2. 네 등급으로 나눈다 — 일하는 방식·도구(적극 반영) / 환경 파일(판단) / 규칙집 공통 부분(줄 단위로 합치기) / 데이터 구조(절대 덮지 않음) 3. 어디까지 가져올지 묻고 반영한다
사람이 할 일	어느 등급까지 가져올지 정한다
결과물	프로젝트 커밋
왜 있나	출발점 수정은 새 프로젝트에만 적용된다 — 기존 프로젝트는 이 스킬로 따라잡는다

`/structure-doc` 데이터가 어떻게 흐르고 왜 이렇게 설계했는지

언제 쓰나	필요할 때 직접
하는 일	1. 코드의 연결 관계를 파악해 개요·구조도·구성요소·데이터 모델·설계 결정을 쓴다
결과물	`docs/structures/주제.md`

`/update-skill-agent-list` 스킬·에이전트 목록표를 다시 만든다

언제 쓰나	스킬·에이전트를 바꾼 뒤 — 빠뜨리면 커밋할 때 알림이 뜬다
하는 일	1. 스킬·에이전트 폴더를 훑어 목록표를 새로 만든다
결과물	`docs/skill_agent_list/` 목록표

하네스 안 ⑤ 에이전트 — 검증

"다 했다"는 말을 다른 눈이 확인한다

에이전트는 메인 AI 가 일을 나눠 맡기는 전문 보조원이다. 대부분 읽기만 하고 결과를 보고하며, 일부는 더 가볍고 빠른 모델로 돈다. 검증 에이전트는 메인 AI 가 만든 것을 다른 눈으로 본다.

스펙 대조·합치기 전 점검은 메인 AI 가 직접 한다 (2026-11-13~)

원래 `spec-verifier`·`merge-master` 두 에이전트가 맡던 일이다. 리허설에서 검증 에이전트가 멈춰 메인이 7분을 기다린 뒤, 두 에이전트를 메인 AI 가 따르는 점검표 (`spec_check.md`·`merge_check.md`)로 바꿨다. 점검 항목은 같다 — 스펙의 명제마다 코드 대조(입출력·흐름·업무 규칙·주소·계산), 합치기 전 미커밋 변경·필수 문서·다른 작업과 겹친 파일·관련 테스트.

`output-verifier` 배치 작업의 결과값 자체가 맞는지 3단계로 대조

언제 쓰나	"이번 실행 결과 검증해 줘"
하는 일	1. 결과 전체의 상태 분포와 의심 패턴을 본다 2. 시트·DB 에 실제 기록된 값을 다시 읽어 계산값과 맞춘다(목표: 불일치 0) 3. 의심 항목과 정상 몇 건을 원본 문서에서 다시 읽어 확인한다
결과물	층별 보고와 이상 항목 목록
왜 있나	"코드가 스펙대로인가"(스펙 대조)와 "이번 실행 결과가 맞나"는 다른 질문이다

`screen-reviewer` 가벼운 모델 · 읽기 전용 화면 캡처를 열어 보고 깨졌는지 판정

언제 쓰나	`/screen-audit` 이 여러 명을 동시에
하는 일	1. 캡처를 한 장씩 열어 정상/깨짐을 판정한다 2. 증상(글자 세로 쌓임·잘림·넘침·겹침 등)과 심각도를 붙인다
결과물	심각한 순서의 판정표

하네스 안 ⑤ 에이전트 — 조사·진단

과거의 교훈을 찾고, 오류의 증거를 모은다

`lesson-finder` 가장 가벼운 모델 팀 교훈 중 이번 작업에 관련된 것만 골라 준다

언제 쓰나	`/develop` 이 큰 작업일 때 자동
하는 일	1. 팀 위키의 교훈 색인만 읽는다 2. 공통·이번 분야 교훈 중 적용 조건이 맞는 것을 고른다 3. 걸린 0~5개의 본문만 읽어 행동 지침으로 요약한다
결과물	관련 교훈 N개와 지침, 또는 "관련 없음"
왜 있나	교훈이 늘어도 비용이 폭주하지 않게 — 색인은 늘, 본문은 걸린 것만

`debug-server-log` 읽기 전용 "안 돼요"가 오면 추측 전에 서버 기록부터

언제 쓰나	디버깅 규칙 1단계에서 반드시
하는 일	1. 오류 시각 앞뒤로 범위를 좁힌다 2. 오류 기록을 뽑고, 같은 시각의 접속 기록으로 어떤 요청이었는지 맞춘다 3. 코드 위치를 짚고, 기록에 직접 보이는 것만 보고한다(추정은 "가능성"으로 표시)
결과물	찾음 / 흔적 없음 / 기록 접근 불가

`llm-usage-doc` 어디서 어떤 AI 를 무엇에 쓰는지 한 장으로

언제 쓰나	"AI 사용 현황 갱신해 줘"
하는 일	1. AI 호출을 모두 찾고, 모델을 고르는 중간 단계까지 따라간다 2. 모델·용도·입출력·비용 수준을 표로 정리한다
결과물	`docs/llm_usage.md` (매번 전체를 새로)
왜 있나	표 일부만 고치다 표가 깨진 일이 있어 매번 통째로 다시 쓴다

하네스 안 ⑤ 에이전트 — 문서·할 일

문서 요청은 맞는 곳으로 보내고, 할 일은 파일 하나씩 관리한다

`write-doc` 문서 요청의 접수 창구

언제 쓰나	"~문서/가이드/안내서 만들어 줘"
하는 일	1. 맞는 문서 스킬이 있으면 "이 스킬로 할까요?" 확인 후 넘긴다 2. 없으면 시점·독자·목적·깊이·저장 위치 다섯 가지를 한 번에 묻고 쓴다
결과물	문서, 또는 맞는 스킬로 연결
왜 있나	고객 안내문에 독자에게 필요 없는 "무엇을 바꿨나"가 습관처럼 들어가는 것을 막는다

`project-intro` 가벼운 모델 코드를 분석해 비개발자용 기능 소개 Word

언제 쓰나	"소개 문서 만들어 줘"
하는 일	1. 기능과 화면 주소를 분석한다 2. 서비스 소개·기능 요약·기능별 "전(수동) vs 후(자동)"·기대 효과를 쓴다
결과물	Word 소개 문서

`backlog-manager` 할 일 목록을 등록·조회·추천·완료 처리

언제 쓰나	"백로그에 추가해 줘", "다음 뭐 할까". `/develop` 마지막에 자동
하는 일	1. 할 일 하나당 파일 하나로 관리한다 2. 등록할 때 중복을 확인한다 3. 조회 결과에는 어느 시점 기준인지 반드시 붙인다
결과물	할 일 파일, 조회 결과
왜 있나	여러 세션이 동시에 등록하다 충돌했고, 조회가 옛 상태로 보고된 일이 있었다

하네스 안 ⑤ 에이전트 — 테스트·파싱

확인이 오래 걸리는 일은 필요할 때만, 정해진 방식으로

api-tester 코드를 바꾼 뒤 API 가 제대로 응답하는지

언제 쓰나	`/develop` 에서 API 를 바꿨을 때
하는 일	1. 서버를 켜지 않고 테스트용 클라이언트로 호출한다 2. 응답·DB 저장·파일 올리기/내려받기를 확인하고 테스트 데이터를 지운다
결과물	확인 결과

pdf-parser · excel-parser 새 문서를 읽어야 할 때 어느 방법이 맞는지 먼저 시험

언제 쓰나	파싱 코드를 새로 쓸 때
하는 일	1. 샘플 앞부분을 글자로 뽑아 숫자·필드명·표·한글이 제대로 나오는지 본다 2. 글자로 충분하면 글자, 아니면 그림(Vision)으로 — 엑셀은 AI 없이 직접 읽는 길도 3. core 부품으로 코드를 쓴다
결과물	읽는 방법 결정과 코드
왜 있나	읽는 방법을 감으로 고르지 않는다. AI 모델은 늘 밖에서 넘겨받는다

ui-tester 가벼운 모델 실제 브라우저로 화면·입력·오류 메시지를 시험

언제 쓰나	사람이 요청할 때만 — "결재 화면 테스트해 줘"
하는 일	1. 화면을 열어 정상 입력·빈 칸·경계값을 넣어 보고 브라우저 오류를 확인한다 2. 확인한 내용을 자동 테스트 코드로 남긴다
결과물	통과/실패 표, 자동 테스트 파일
왜 있나	한 번에 몇 분씩 걸려 매 개발 흐름에는 넣지 않았다. 대신 `/develop` 이 끝나면 "화면 확인이 필요하면 ui-tester 로" 한 줄을 남긴다

하네스 안 ⑥ 하루의 흐름

사람은 말하고 결정하고,
스킬과 에이전트는 서로를 부른다

하루 동안 오가는 요청이 어느 스킬로 들어가고, 그 스킬이 어떤 에이전트를 부르는지 한 장에 모았다. 사람이 외워야 할 명령은 없다 — 말로 하면 규칙집이 맞는 흐름으로 보낸다.

```
[아침]       "다음 뭐 할까"            → backlog-manager

[기능 개발] "∞ 기능 만들어 줘"        → /develop (기본 적용)
              ├ 맥락 읽기 (큰 작업)      → lesson-finder ← 팀 교훈 색인
              ├ 사람 승인 1회 · 스펙 기록
              ├ 검증 (메인이 점검표로)   → api-tester · pdf/excel-parser
              │                              (오류 보고 시 debug-server-log)
              ├ 버그였다면 이슈 문서 → 교훈 → 다음 프로젝트의 lesson-finder
              ├ 합치기 전 점검 (점검표)  → 정본에 합치기·정리
              └ 할 일 완료                → backlog-manager

[서버]       "서버 켜 줘"               → /dev
[배포]       "배포해 줘"                → /release
[회원가입]   "회원가입 만들어 줘"       → /signup → /develop → /email-setup → /dev → /release
[오류]       "안 돼요"                   → debug-server-log → (수정) /develop
[문서]       "~문서 만들어 줘"          → write-doc → /feature-doc · /runbook · /pdf-doc · /manual
                                              · /client-usage-guide (→ ui-tester · create-docx) · ...
[화면 점검] "모바일 화면 확인"          → /screen-audit → screen-reviewer × N
[출발점]     좋아진 것 올리기           → /sync-to-template → 팀 위키 변경 이력
              최신 개선 받기              → /sync-from-template
```

처음 하는 사람이 알아야 할 것은 "무엇을 원하는지"뿐이다

스킬 17개와 에이전트 12개의 이름을 몰라도 된다. 그 대신 승인할 때 쉬운 말 요약을 읽고, 정책 질문에 답하고, 결과를 눈으로 확인하는 일 — 업무를 아는 사람만 할 수 있는 일을 한다.

04

벽 4 · 모르는 실수

모르고 한 실수도 기계가 먼저 막는다

처음 하는 사람은 무엇이 실수인지 모른다. 규칙으로 적어 두면 숙련자는 지키지만, 처음 하는 사람도 AI 도 잊는다. standarda 는 실수가 일어나는 **바로 그 순간**에 멈추고, 왜 멈췄는지와 대신 무엇을 하면 되는지를 알려 준다.

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 네 번째 장치.

벽 — 처음 하는 사람이 모르고 하는 실수

실수라는 걸 모르면, 조심할 수도 없다

AI 와 함께 처음 무언가를 만드는 사람은 아래 일들이 문제라는 것 자체를 모른다. 화면은 멀쩡하게 돌아가므로, 문제는 한참 뒤 다른 사람이 이어받을 때나 무언가 어긋났을 때 드러난다.

고치고 기록을 안 남긴다
오류를 고쳤지만 원인과 교훈이 남지 않아, 다른 프로젝트에서 같은 오류가 또 난다

만들고 스펙을 안 남긴다
무엇을 왜 만들었는지 없어서 다음 사람도, 다음 대화의 AI 도 의도를 모른다

오류의 흔적을 지운다
예외를 잡고 짧은 메시지만 남겨, 나중에 어디서 왜 났는지 찾을 수 없다

엉뚱한 곳에 올린다
복사해 온 프로젝트라 연결된 저장소가 남의 것인 줄 모르고 그대로 올린다

정본을 되돌린다
"방금 것 취소"로 되돌렸는데, 그 사이 정본에 합쳐진 다른 작업까지 사라진다

원격 서버에서 위치를 착각한다
서버에 접속해 보낸 명령이 엉뚱한 폴더에서 실행돼 같은 실패를 반복한다

목록을 안 고친다
새 스킬을 만들었는데 목록에 없어 아무도 그 스킬이 있는 줄 모른다

이름표를 대충 쓴다
스킬 설명을 "보고서" 한 단어로 써서, 말로 부를 때 엉뚱한 스킬이 불린다

규칙으로 적어 두는 것만으로는 안 된다

위 실수들은 모두 규칙집(CLAUDE.md)에 "하지 말라"고 적혀 있다. 그래도 일어났다 — 사람은 바빠서 잊고, AI 는 긴 대화 중에 놓친다. 그래서 규칙을 **읽는 것**이 아니라 **통과해야 하는 것**으로 바꿨다.

standarda 의 장치

실수하는 순간에 멈추는 검사기 11개가 처음부터 들어 있다

검사기(hook)는 정해진 순간에 자동으로 돌아 규칙에 어긋나면 그 동작을 멈춘다. 프로젝트를 만들 때 이미 설치돼 있어서, 쓰는 사람은 존재를 몰라도 보호를 받는다.

막는 실수	언제	막으면 알려 주는 것
AI 가 명령을 실행하거나 파일을 쓰기 직전 — Claude hook 6개		
오류를 고치고 원인 기록(이슈 문서)을 안 남김	커밋 직전	이슈 문서를 같은 커밋에 넣으라
기능을 만들고 스펙을 안 남김	커밋 직전	스펙 문서를 넣으라(`/develop` 이 만들어 준다)
스킬·에이전트를 바꾸고 목록을 안 고침	커밋 직전	목록 갱신 스킬을 먼저 돌리라
오류의 흔적(추적 정보)을 지우는 코드	커밋 직전	문제의 파일·줄과 바꿀 방법
원격 서버에서 위치에 의존하는 명령	실행 직전	절대경로로 바꾸는 법
대충 쓴 스킬 이름표	파일 쓰기 직전	무엇을·언제 쓰는지, 비슷한 스킬과의 경계를 적으라
git 이 커밋·올리기·되돌리기를 하는 순간 — git hook 5개 (사람이 직접 해도 막힌다)		
원인 기록 없는 오류 고침 커밋	커밋	이슈 문서를 넣으라
목록을 안 고친 스킬 변경	커밋	목록을 갱신하라
엉뚱한 저장소로 올리기	올리기(push)	올리려던 곳과 원래 저장소, 고치는 명령
정본(main) 되돌리기·덮어쓰기	정본이 움직이는 순간	사라질 커밋 수, 대신 쓸 방법(`git revert`)
정본 다시 쓰기(rebase)	rebase 시작 전	대신 쓸 방법(`merge`)

Claude hook 과 git hook 이 같은 실수를 두 번 막는 곳이 있다 — AI 가 아닌 사람이 직접 커밋해도 git 단계에서 한 번 더 걸리게 하기 위해서다.

실제로 막힌 장면 (1) — 사람이 git 으로 직접 해도

오류 고침·엉뚱한 저장소·정본 되돌리기, 세 가지 모두 실행되지 않았다

standarda 로 새로 만든 프로젝트(settle_helper)에서 실제로 시도한 결과다. 아래 출력은 손대지 않은 그대로다.

① 원인 기록 없이 "오류 고침" 커밋 — 커밋되지 않음

```
$ git commit -q -m "fix: 목록 정렬 오류 수정"
⚠️ 버그 수정 커밋에 이슈 문서가 포함되지 않았습니다.
docs/issues/{번호}_{slug}.md 파일을 작성하여 같은 커밋에 포함해주세요.
(단순 오타나 자명한 수정이면 git commit --no-verify 로 건너뛸 수 있습니다)
[exit 1]  ← 실행되지 않음
```

② 다른 팀의 저장소로 올리기 — 아무것도 올라가지 않음

```
$ git push other master
ERROR: Pushing to unexpected remote!
  Remote URL: ~/other-team-repo.git
  Expected repo: brie-org/settle_helper

Fix with: git remote set-url origin https://github.com/brie-org/settle_helper.git
error: failed to push some refs to '~/other-team-repo.git'
[exit 1]  ← 실행되지 않음
```

③ 정본(master)을 한 단계 되돌리기 — 커밋 유실 0 (안내 일부 생략)

```
$ git reset --hard HEAD~1

✋ 정본 브랜치 되감기 차단 — 정본 브랜치(main/master)는 fast-forward 로만 움직입니다.

  refs/heads/master: 0aac6c7 → 9c935ea (커밋 1개 유실)


대신:
  - 내용을 되돌리려면          git revert <sha>
  - 커밋 메시지·분리가 틀렸으면 그대로 두고 후속 커밋을 추가
  - 여러 번 다듬고 싶으면      main 에 직접 커밋하지 말고 워크트리 브랜치에서 작업

범위를 직접 확인했고 정말 되감아야 한다면:
  SETTLE_HELPER_ALLOW_MAIN_REWIND=1 <명령>

fatal: ref updates aborted by hook
[exit 128]  ← 실행되지 않음
```

실제로 막힌 장면 (2) — AI 가 하려던 일

AI 가 실수하려는 순간, 이유와 고칠 방법이 같이 돌아온다

AI 가 명령을 실행하거나 파일을 쓰기 직전에 Claude Code 가 검사기에 넘기는 것과 같은 입력을 넣어 받은 응답이다. AI 는 이 응답을 읽고 스스로 고쳐 다시 시도한다.

④ 스펙 없이 기능 커밋

```
AI 가 하려던 일: "feat: 결재 상태 칸 추가" 로 커밋 (스펙 문서 없음)

⚠️ 기능 개발 커밋인데 스펙/기능 문서가 포함되지 않았습니다.
docs/specs/{app}.md (사전 의도) 또는 docs/features/{app}.md (구현 후 기록) 를
같은 커밋에 포함해주세요. develop 스킬 또는 /feature-doc 로 생성할 수 있습니다.
(자명한 변경으로 오탐이면 커밋 메시지에서 feat/기능/구현 키워드를 빼면 통과합니다)
[exit 2]  ← 실행되지 않음
```

⑤ 오류의 흔적을 지우는 코드를 커밋

```
AI 가 하려던 일: 예외를 잡고 logger.error 만 남긴 projects/sync.py 를 커밋

⚠️ except 블록에서 logger.error/.warning 사용 — traceback 이 로그에 안 남습니다.
logger.exception(...) 으로 바꿔 traceback 을 보존하세요 (CLAUDE.md ## Code Patterns › Error
Handling).
  - projects/sync.py:8
(예외가 아닌 논리적 실패 기록이거나 의도적이면, 로그 호출을 except 밖으로 옮기세요)
[exit 2]  ← 실행되지 않음
```

⑥ 운영 서버에 접속해 위치를 가정한 명령

```
AI 가 하려던 일: ssh brie-prod "git pull && source bin/activate && python manage.py migrate"

⛔ SSH 원격 명령이 원격 cwd(원격 $HOME)에 의존합니다 — 그대로 보내면 깨집니다.
→ git 명령에 작업 경로가 없습니다 → 'git -C <절대경로> ...' 로 바꾸거나 'cd <경로> &&' 를 앞에 두세요
→ 원격 셸은 항상 원격 $HOME 에서 시작합니다(로컬 cwd 전달 안 됨). release 는 $RSH "cd $PROOT && ..." 헬
퍼를 쓰세요.
→ 같은 에러가 났다면 '고쳤다'는 산문/description 말고 명령 문자열이 실제로 달라졌는지 확인하세요.
[exit 2]  ← 실행되지 않음
```

⑦ 스킬 이름표를 "보고서" 한 단어로

```
AI 가 하려던 일: .claude/skills/report/SKILL.md 에 description: 보고서

⛔ SKILL.md description 규칙 위반: description 이 너무 짧습니다. 무엇을·언제 쓰는지 구체적으로 적으세요
→ frontmatter 에 'name:' 과 구체적인 'description:'(무엇을·언제 쓰는지 포함)을 작성하세요.
→ 비슷한 skill 과 헷갈리지 않도록 'Use when ... / 이런 경우엔 /other-skill' 형태로 경계를 적으세요.
→ 작성 표준 전문: /home/ubuntu/wiki/development/skill-policy.md
[exit 2]  ← 실행되지 않음
```

정본 보호

정본은 앞으로만 움직이고, 되돌리는 것도 기록으로 남긴다

여러 사람과 여러 AI 대화가 같은 정본에 작업을 합친다. 그중 하나가 정본을 되돌리면, 이미 "끝났다"고 보고된 다른 작업이 사라진다. 그래서 정본은 앞으로만 움직이게 했다.

장치	하는 일
정본은 앞으로만	main·master(와 원격 기본 브랜치)가 **뒤로 가는** 순간만 막는다. 명령어를 금지하는 게 아니다 — 작업 브랜치의 되돌리기·다시 쓰기는 자유, 정본에 새 커밋·합치기·되돌리기 기록(`revert`)도 자유
rebase 는 시작 전에	정본을 다시 쓰는 rebase 는 시작하기 전에 막아, 중간에 멈춘 뒷정리가 남지 않게
올리는 곳 확인	프로젝트를 만들 때 정해진 저장소가 아니면 올리기를 막는다
검사기가 지워지는 우회까지	"되돌리기" 명령이 검사기 파일부터 지워 보호를 우회한 일이 있었다(점검 중 발견). 그래서 git 이 실제로 읽는 검사기는 작업 폴더 **밖**에 두고, 작업 폴더의 원본이 사라지면 설치 때 남긴 사본이 대신 지킨다
막는 쪽과 열어 두는 쪽	정본 보호 검사기는 고장 나면 **막는** 쪽(fail-closed), 문서 검사 같은 품질 검사기는 고장 나면 **통과시키되 알리는** 쪽(fail-open) — 설치가 깨져도 일이 멈추지 않게

권한을 뺏는 게 아니라 "실수로"를 "알고서"로 바꾼다

정말 되돌려야 하면 프로젝트 이름이 들어간 표시(`SETTLE_HELPER_ALLOW_MAIN_REWIND=1`)를 붙여 실행하면 된다. 목적은 금지가 아니라, 모르고 한 되돌리기를 알고 하는 되돌리기로 바꾸는 것이다.

기본값으로 막기

고칠 줄 몰라도,
처음부터 맞게 되어 있다

검사기가 "하면 안 되는 것"을 막는다면, 기본값은 "해야 하는데 모르는 것"을 미리 해 둔다. 아래는 모두 누군가 한 번 겪은 일을 출발점의 기본값으로 바꾼 것이다.

모르고 하는 실수	실제로 겪은 일	지금 출발점의 기본값
날짜를 그냥 저장	저장한 시각과 화면에 보이는 시각이 9시간씩 어긋남	시간대를 지켜 저장하고 서울 시각으로 보여준다(설정과 규칙)
계정에 딸린 정보를 빠뜨림	관리자 도구로 만든 계정으로 로그인하자 오류 — 교육 리허설에서 세 번 반복	계정이 어떤 방법으로 생기든 프로필이 함께 만들어진다
등록 화면만 만들고 끝	견본에 등록 기능 하나뿐이라 AI 가 그대로 따라 만듦	견본 업무 앱이 목록·상세·등록·수정·삭제를 모두 갖춤
화면과 서버가 다른 규칙	화면이 안내한 비밀번호 `Abcd-1234` 를 서버가 거부	같은 규칙을 쓰고, 둘이 같은지 자동 테스트가 확인
개발·운영 설정을 섞음	운영 설정 파일이 늦게 읽혀 개발 값으로 돌던 함정	서버가 스스로 판단해 맞는 설정 파일을 먼저 읽는다
관리자 계정을 손으로	셋업 중 입력에서 멈추고, 팀마다 관리자 아이디가 제각각, 운영 서버엔 0명	개발·운영 서버 모두 관리자 계정이 자동으로 만들어진다
파일을 엉뚱한 곳에	작업 공간에서 서버를 띄우면 올린 파일이 다른 폴더에 쌓이고 합친 뒤 사라짐	파일 위치를 "어느 환경으로 실행했나" 기준으로 계산 — 어디서 띄워도 같은 곳

검사기와 기본값은 짝이다 — 기본값은 처음부터 맞게 해 두고, 검사기는 그 기본값에서 벗어나려는 순간을 잡는다.

이 장치가 "스스로 하게" 만드는 이유

안전장치가 조용히 꺼져 있으면 안전장치가 아니다

실수를 막는 장치가 있어도, 그 장치가 고장 난 걸 아무도 모르면 없는 것과 같다. standarda 는 검사기 자체도 검사한다.

1. 규칙은 문서가 아니라 통과해야 하는 문으로

처음 하는 사람에게 규칙집을 다 읽고 기억하라고 할 수 없다. 대신 실수하는 순간에 멈추고, 그 자리에서 이유와 고칠 방법을 알려 준다. 앞 쪽 장면들의 메시지는 모두 "무엇이 문제인지 → 대신 어떻게 하면 되는지"로 되어 있다. 사람이든 AI 든 그 메시지를 읽고 바로 고칠 수 있다.

시작하는 사람에게는 — 규칙을 몰라도 규칙대로 하게 된다.

2. 검사기도 검사한다

2026년 9월, 서버 정리 중 검사기들이 쓰던 도구 하나(jq)가 지워졌다. 그 뒤로 **모든 검사기가 아무것도 막지 않고 통과**했고, 이 상태가 두 달간 이어졌다 — 오류 메시지도 없었다. 발견한 뒤 검사기가 그 도구 없이 돌게 바꾸고, 그 도구를 일부러 가린 채 검사기들을 시험하는 회귀 테스트를 만들었다. 지금 22개 시험이 모두 통과한다.

시작하는 사람에게는 — 보이지 않는 보호가 실제로 켜져 있다는 것을 믿을 수 있다.

3. 막는 장치도 다듬는다

검사기가 엉뚱한 것을 막으면 일이 멈춘다. 그래서 오탐이 나오면 그것도 개선 거리로 모아 고친다(예: 문서만 바꾼 커밋의 메시지에 "수정"이 들어가 막히는 경우).

처음 하는 사람이 혼자 해도 되는 이유

모르는 실수는 조심해서 피할 수 없다. 출발점의 기본값이 미리 맞게 해 두고, 검사기가 그 순간에 잡고, 그 검사기가 켜져 있는지까지 확인한다 — 처음 하는 사람이 혼자 일해도 실수가 쌓이지 않는 이유다.

04+

hook 안 들여다보기

검사기 11개는 각각 무엇을 보나

Claude hook 6개는 AI 가 명령을 실행하거나 파일을 쓰기 직전에, git hook 5개는 커밋·올리기·정본이 움직이는 순간에 돈다. 다음 세 쪽에서 하나씩 — 언제 돌고, 무엇을 검사하고, 막으면 무엇을 알려 주고, 왜 생겼는지 — 를 본다.

위치	내용
`.claude/hooks/`	Claude hook 6개 + 입력 읽기 도우미 + 검사기 회귀 테스트(`test-hooks.sh` , 22건)
`.githooks/`	git hook 5개 + 설치 스크립트(`install.sh` — 작업 폴더 밖에 설치하고 사본을 남김)
`.claude/settings.json`	어느 Claude hook 이 어느 순간에 도는지 연결

hook 안 ① Claude hook — 커밋 직전

AI 가 커밋하려 할 때 네 가지를 본다

check-bugfix-issue-doc Claude 오류 고침 커밋엔 원인 기록을

언제	AI 가 `git commit` 을 실행하기 직전
검사	1. 커밋 **메시지**에 오류 고침 단어(수정·fix·버그·오류 등)가 있는지 — 파일 이름이 아니라 메시지만 본다 2. 같은 커밋에 `docs/issues/` 문서가 들어 있는지
막으면	이슈 문서를 같은 커밋에 넣으라
왜 생겼나	교훈이 다른 프로젝트로 퍼지려면 원인·해결·교훈이 남아야 한다(교훈 전파의 출발점). 파일 이름 오탐·작업 공간 경로 미스매치를 겪은 뒤 메시지만 보도록 다듬었다

check-feature-doc Claude 기능 커밋엔 스펙을

언제	AI 가 `git commit` 을 실행하기 직전
검사	1. 메시지에 기능 단어(feat·기능·구현)가 있고 2. 앱 코드(.py)가 들어 있는데 3. 스펙 문서(`docs/specs/`)가 없으면
막으면	스펙 문서를 넣으라 — `/develop` 이 만들어 준다
왜 생겼나	`/develop` 을 거치지 않은 직접 커밋에서도 스펙이 빠지지 않게 하는 마지막 문

update-skill-agent-list Claude 스킬·에이전트를 바꾸면 목록도

언제	AI 가 `git commit` 을 실행하기 직전
검사	1. 스킬 추가·삭제나 이름표 변경, 에이전트 변경이 있는지(본문만 고친 건 통과) 2. 목록 문서(`docs/skill_agent_list/`)가 같이 바뀌었는지
막으면	목록 갱신 스킬을 먼저 돌리라
왜 생겼나	스킬이 있는데 아무도 모르는 일을 막는다

check-except-logger-exception Claude 오류의 흔적을 지우지 않게

언제	AI 가 `git commit` 을 실행하기 직전
검사	1. 이번 커밋에 **새로 추가된** 코드에서 2. 예외를 잡는 블록이 짧은 메시지(`logger.error`)만 남기는지 — 코드 구조를 분석해 정확히 판단
막으면	문제의 파일·줄, `logger.exception` 으로 바꾸라
왜 생겼나	회의록·받아쓰기 작업에서 오류가 났는데 원인 추적 정보가 로그에 없어 찾지 못한 일(popax)

hook 안 ② Claude hook — 실행·파일 쓰기 직전

원격 명령과 스킬 이름표는 보내기 전에 확인한다

`check-ssh-remote-cwd` Claude 원격 서버에서 위치를 착각하지 않게

언제 AI 가 `ssh` 명령을 실행하기 직전

검사
1. 원격에 보내는 명령 부분만 본다
2. 경로 없이 `git` 을 쓰거나, 상대 경로(`source bin/activate` , `../`)에 기대는지 — 원격 셸은 늘 홈 폴더에서 시작한다

막으면 절대경로· `git -C` 로 바꾸는 법, 그리고 "같은 오류가 났다면 명령이 실제로 바뀌었는지 확인하라"

왜 생겼나 운영 배포 중 같은 명령을 네 번 다시 보내며 같은 실패를 반복한 일 — 기억(메모)만으로는 재발해서 구조로 막았다

`check-skill-description` Claude 스킬 이름표를 대충 쓰지 않게

언제 AI 가 `SKILL.md` 를 쓰거나 고치기 직전

검사
1. 이름(name)과 설명(description)이 있는지
2. 설명이 너무 짧거나 모호하지 않은지 — 무엇을·언제 쓰는지

막으면 무엇을·언제 쓰는지, 비슷한 스킬과의 경계를 적으라 + 작성 표준 문서 위치

왜 생겼나 모든 스킬을 말로 부를 수 있게 했더니, 이름표가 모호하면 엉뚱한 스킬이 불렸다

도우미 · 회귀 테스트 검사기가 조용히 꺼지지 않게

언제 검사기를 고치거나 서버 환경이 바뀔 때

검사
1. `hook_input.py` — Claude Code 가 넘기는 입력을 외부 도구 없이 읽는다
2. `test-hooks.sh` — 막아야 할 입력은 막고 통과시킬 입력은 통과하는지 22건을 시험한다. 일부러 jq 를 가린 채 돈다

막으면 실패한 시험 이름

왜 생겼나 서버에서 jq 가 지워진 뒤 모든 검사기가 두 달간 아무것도 막지 않은 일(2026-09)

hook 안 ③ git hook

사람이 직접 해도 걸리는 다섯 개 — 그중 셋은 정본을 지킨다

commit-msg `git` 오류 고침 커밋엔 원인 기록을

언제	커밋할 때(사람·AI 모두)
검사	1. 실제 커밋 메시지에 오류 고침 단어가 있는지 2. 실제로 커밋될 파일에 `docs/issues/` 가 있는지
막으면	이슈 문서를 넣으라
왜 생겼나	Claude hook 과 같은 규칙을 사람의 직접 커밋에도

pre-commit `git` 스킬을 바꾸면 목록도

언제	커밋할 때
검사	1. 목록에 영향을 주는 스킬·에이전트 변경이 있는데 목록 문서가 없는지
막으면	목록 갱신을 먼저
왜 생겼나	위와 같음

pre-push `git` 엉뚱한 저장소에 올리지 않게

언제	올리기(push) 직전
검사	1. 올리려는 주소에 프로젝트를 만들 때 정해진 저장소 이름이 들어 있는지
막으면	올리려던 곳·원래 저장소·고치는 명령
왜 생겼나	복사해 온 프로젝트가 다른 저장소에 연결된 채 올라간 일(wooha)

reference-transaction `git · 고장 나면 막음` 정본은 앞으로만

언제	정본 브랜치가 움직이는 모든 순간
검사	1. main·master·원격 기본 브랜치가 **뒤로 가는** 업데이트인지(새 커밋·합치기·revert 는 통과)
막으면	사라질 커밋 수, 대신 쓸 방법, 이미 바뀐 작업 파일의 뒷정리, 의식적 우회 방법
왜 생겼나	합친 직후 작업 공간을 지우는 흐름이라, 정본을 되돌리면 다른 대화가 끝낸 작업이 사라질 수 있다

pre-rebase `git · 고장 나면 막음` 정본을 다시 쓰지 않게

언제	rebase 시작 전
검사	1. rebase 대상이 정본 브랜치인지(작업 브랜치는 자유)
막으면	대신 `merge` 를 쓰는 법
왜 생겼나	끝에서 막으면 중간 상태가 남아 뒷정리가 필요해, 시작 전에 막는다

부록 — 시연 방법

"실제로 막힌 장면"은 이렇게 만들었다

항목	내용
일시·장소	2026-11-14, Brie 개발 서버(brie-box)
프로젝트	standarda-template 최신판(2026-11-13 #47 기준)으로 만든 새 프로젝트 `settle_helper`. git 저장소를 만들고 `.githooks/install.sh` 로 검사기를 설치 — 실제 프로젝트 생성 때와 같다
git hook 장면	실제 git 명령을 실행하고 출력을 그대로 옮겼다. "다른 팀 저장소"는 외부에 올리지 않도록 서버 안의 빈 저장소로 만들었다
Claude hook 장면	AI 가 명령·파일 쓰기를 하려 할 때 Claude Code 가 검사기에 넘기는 것과 같은 형식의 입력을 넣고 응답을 옮겼다. "AI 가 하려던 일" 줄은 그 입력을 사람 말로 적은 것
결과	시도 9가지 모두 실행되지 않았다. 정본 되돌리기 시도 뒤 정본의 커밋은 그대로였다. 검사기 회귀 테스트 22건 통과
뒷정리	시연 프로젝트는 서버 임시 폴더에서만 만들었고 GitHub·DNS 등 바깥에는 아무것도 만들지 않았다

05

벽 5 · 서버·배포·운영

서버를 몰라도 내 앱을 올리고, 망가뜨려도 되돌아온다

개발 서버에서 돌아가는 앱을 동료와 고객이 매일 쓰게 하려면 운영 서버에 올려야 한다. 처음 하는 사람에게 이 일은 서버·DB·도메인·인증서를 모두 알아야 하는 벽이다. standarda 에서는 **"배포해 줘" 한 마디**로 올리고, 실패하면 **스스로 되돌아오며**, 여러 사람이 한 서버를 써도 **서로의 것을 보지 못한다.**

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 다섯 번째 장치.

벽 — "운영 서버에 올려 줘" 뒤에 숨은 질문

개발 서버에서 되는 앱을 매일 쓰게 만드는 일은 따로 있다

개발 서버에서 잘 돌아가는 것과 동료·고객이 매일 쓰는 것은 다른 일이다. 운영에 올리는 순간 아래 질문들에 답해야 한다. 대부분 서버 관리자의 경험으로 답하던 것들이다.

"운영 서버에 올려 줘" → 그 뒤에 따라오는 질문들

어디에, 무엇이 달라야 하나
개발 서버와 같은 곳에 둬도 되나? 운영 설정은 무엇이 달라야 하나

주소·인증서·웹서버는 누가
도메인을 등록하고 HTTPS 인증서를 받고 웹서버를 연결하는 일

운영 DB 와 관리자는
운영 DB 를 따로 만들고, 비밀번호를 정하고, 첫 관리자 계정을 만드는 일

DB 구조가 바뀌면
새 칸을 추가하는 업데이트에서 기존 데이터는 어떻게 되나. 잘못되면 되돌릴 수 있나

배포하다 실패하면
사용자는 깨진 화면을 보나. 누가, 어떻게 되돌리나

꺼지면, 로그는
서버가 재부팅되면 앱이 다시 켜지나. 문제가 생기면 기록은 어디서 보나

여러 사람이 한 서버에
다른 사람의 앱이 메모리를 다 쓰거나, 내 설정·데이터를 볼 수 있지는 않나

안 쓰는 앱은
만들어 놓고 잊힌 앱이 자원을 계속 차지하면 누가 정리하나

보통은 이 질문마다 서버 관리자에게 부탁한다. 그러면 관리자가 병목이 되고, 처음 하는 사람은 "내가 만든 것을 내가 올리는" 경험을 하지 못한다.

standarda 의 장치 ① 두 대의 서버

개발과 운영을 나누고, 각자 자기 앱을 직접 다룬다

서버는 두 대다. 개발 중에 DB 구조를 바꾸거나 서버를 다시 켜도 운영 중인 앱과 데이터에는 아무 영향이 없다. 두 서버 모두 사람마다 자기 계정으로 들어가 자기 앱만 다룬다.

	개발 서버 (brie-box)	운영 서버 (brie-prod)
용도	만들고 시험한다	동료·고객이 매일 쓴다
주소	`이름-dev.brie.co.kr`	`이름.brie.co.kr`
여는 법	`/dev` — 주소·인증서·웹서버 연결을 헬퍼 한 줄이 처리	`/release` — 처음 한 번 자리를 만들고, 그 다음부터 배포 한 줄
DB	자기 이름의 DB 를 직접 만든다	배포 헬퍼가 무작위 비밀번호로 만들고 본인만 읽을 수 있게 보관
앱 실행	개발 서버(작업 중) 또는 본인 계정의 상시 서비스	본인 계정으로 도는 서비스, 재부팅되면 자동으로 다시 켜짐
자원	사람마다 메모리 상한 3GB	앱마다 메모리 상한 512MB, 서버 전체 용량 감시
접속	바깥에는 HTTPS(443)만 열려 있음	관리용 접속은 개발 서버에서만 가능

사람이 하는 일

개발 서버: "서버 켜 줘" → 확인 한 번. 운영 서버: "배포해 줘" → 상태 확인이 실패했을 때만 판단. 서버 설정 파일을 직접 고치거나 관리자에게 부탁할 일이 없다.

standarda 의 장치 ② 배포 한 마디

"배포해 줘" 한 마디 안에서 스냅샷·확인·되돌리기가 돈다

운영 배포는 `/release` 스킬이 이끌고, 실제 작업은 서버의 배포 헬퍼(`brie-prod-deploy`)가 한다. 헬퍼는 정해진 순서를 매번 똑같이 밟고, 중간에 문제가 생기면 스스로 직전 상태로 돌아간다.

처음 한 번 — 운영 서버에 자리 만들기

운영 서버 접속 확인 → 코드 받기·설치 → `brie-prod-init`(DB·서비스·주소·인증서·등록을 한 번에, 실패하면 만든 것을 스스로 정리) → 운영 설정 파일 자동 생성 → 운영 관리자 계정 생성(비밀번호는 한 번만 보여 줌)

그다음부터 — 배포할 때마다

1 **이번에 올라갈 변경을 보여 준다**
무엇이 운영에 반영되는지 커밋 목록으로 확인한다.

2 **코드를 받고 필요한 프로그램을 설치한다**

3 **DB 구조가 바뀌는 배포면 먼저 스냅샷을 찍는다** — 되돌릴 수단
본인만 읽을 수 있는 파일로, 최근 5개를 보관한다. **스냅샷을 못 찍으면 DB 를 바꾸지 않고 멈춘다.**

4 **DB 구조 반영 → 정적 파일 모으기 → 앱 재시작**

5 **앱이 실제로 응답하는지 확인한다** — 확인
2초 간격으로 20번까지 물어본다.

6 **응답이 없으면 직전 버전으로 되돌리고 다시 켠다** — 자동 롤백
코드는 자동으로 되돌린다. DB 까지 되돌려야 하면 3단계 스냅샷으로 복원하는 명령을 알려 준다(사람이 확인 후 실행).

배포·정지·삭제는 모두 운영 서버의 기록(로그)에 누가·언제·무엇을 남긴다. 다음 쪽의 기록이 그것이다.

실제 기록 — 10월 검증

일부러 실패시킨 배포는 스스로 되돌아왔다

셀프 배포를 만들며 테스트 계정으로 시험한 운영 서버의 실제 배포 기록이다(가독성을 위해 칸만 맞췄다).

```
2026-10-11 15:01 deploy devtest devtest-app  8a35ca66→e7922429  OK
2026-10-19 16:04 deploy devtest snaptest     8fbc5d7b→8fbc5d7b  FAILED rolled-back
2026-10-19 16:04 deploy devtest snaptest     d0694aba→d0694aba  OK
2026-10-19 16:04 deploy devtest snaptest     84914e58→84914e58  OK
snap=…/snaptest_db_snapshots/snaptest-20261019-160426-84914e58.dump
2026-10-19 18:41 alert sent: [brie-prod] 방치 앱 자동 정지 — idletest
2026-10-19 18:41 start  devtest idletest
2026-10-19 18:42 remove devtest idletest
2026-10-26 11:16 deploy sergei  erp3         59f0913d→59f0913d  OK
snap=…/erp3_db_snapshots/erp3-20261026-111651-59f0913d.dump
```

기록	무엇을 보여 주나
`FAILED rolled-back`	일부러 깨뜨린 배포(`snaptest`)가 상태 확인에서 실패하자 직전 버전으로 스스로 돌아가 다시 켜졌다 — 사람이 손대지 않았다
바로 다음 줄 `OK`	고친 버전은 같은 명령 한 줄로 정상 배포
`snap=….dump`	DB 구조가 바뀌는 배포에서 바꾸기 직전 스냅샷이 자동으로 찍혔다
`방치 앱 자동 정지`	오래 안 쓴 앱(`idletest`)이 자동으로 멈추고 알림이 갔다 → 다시 켜기 → 정리까지 본인이 직접

기록의 `devtest`·`sergei`는 팀원 권한을 시험하는 테스트 계정이다 — 관리자가 아니어도 이 모든 것을 본인이 직접 했다는 뜻이다.

운영 장치 — 여러 사람의 앱이 한 서버에

한 서버를 같이 써도 서로의 자원을 빼앗지 않는다

여러 사람이 각자 만든 앱이 한 운영 서버에 모인다. 한 앱이 문제를 일으켜도 다른 앱이 함께 쓰러지지 않게, 그리고 잊힌 앱이 자원을 계속 차지하지 않게 했다.

장치	하는 일
앱마다 메모리 상한	운영 서버의 앱 하나는 512MB, 개발 서버의 사람 한 명은 3GB 까지. 한 앱이 메모리를 다 써도 다른 앱은 영향을 받지 않는다
서버 용량 감시	남은 메모리가 400MB 아래거나 상한 합계가 75% 를 넘으면 경고하고 관리자에게 메일 — 작업은 멈추지 않는다. 서버를 키우는 일은 관리자 담당
내 앱만 조작	배포·재시작·정지·삭제 명령은 등록부에 적힌 **본인 앱만** 받아 준다
방치 앱 자동 정리	실제 사용(사람의 접속·입력)이 30일 없으면 경고, 44일이면 자동 정지 — 코드·DB·주소는 남겨 두어 한 줄로 다시 켠다. 검색 로봇의 방문은 사용으로 치지 않는다
개발 서버 야간 정지	밤에 아무도 쓰지 않으면 백업 후 서버를 끈다. 밤샘 작업은 "켜 둠" 표시로 막는다
매일 DB 백업	전체 DB 를 7일치 보관하고 서버 바깥(S3)에도 사본을 둔다
지울 때도 남긴다	운영 앱을 지워도 DB 와 코드는 남긴다. DB 삭제는 관리자에게 따로 요청

보안 — 서로의 것을 보지 못하게

같은 서버를 써도
내 설정과 데이터는 나만 본다

여러 사람이 한 서버에서 일하면 실수로라도 남의 비밀번호·키·데이터를 볼 수 있으면 안 된다. standarda 의 서버는 사람마다 따로 들어가고, 할 수 있는 일을 검증된 명령으로만 제한하고, 비밀 파일은 주인만 읽게 한다.

장치	하는 일
사람마다 자기 계정	공용 계정을 나눠 쓰지 않는다. 누가 무엇을 했는지 계정 단위로 남는다. 새 사람은 관리자의 명령 한 줄로 개발·운영 계정이 함께 만들어진다
관리자 권한 대신 검증된 명령	팀원은 서버 전체 관리자 권한이 없다. 대신 주소 열기·운영 배포 같은 **검증된 헬퍼 몇 개만** 쓸 수 있다. 헬퍼는 입력을 검사하고, 남의 것이면 거부하고, 실패하면 원래대로 돌려놓는다. 파일 쓰기·셸 실행·임의 인자를 허용하는 명령은 넣지 않는다는 원칙
남의 폴더·DB 는 닫힘	각자의 홈 폴더는 다른 사람이 들어갈 수 없다. 기존 DB 는 소유자 외 접속을 막았다
새 파일은 처음부터 나만	새로 만드는 파일은 기본으로 다른 사람이 읽을 수 없다(접속·예약 작업 모두에 적용)
웹서버도 필요한 곳만	웹서버는 사람들의 그룹에 넣지 않고, 정적 파일 폴더만 읽게 한다
비밀 파일 매시간 점검	설정 파일(.env)·토큰·키·DB 덤프가 주인 외에게 읽히면 매시간 자동으로 바로잡고 기록한다 — 파일 복사·압축처럼 기본 설정을 거치지 않는 경로의 안전망
문은 최소로	개발 서버는 바깥에 HTTPS 만, 운영 서버의 관리 접속은 개발 서버에서만

점검에서 실제로 찾아 고친 것 (2026-10)

팀원 권한을 시험하는 테스트 계정으로 확인해 보니, 오래된 파일 중 다른 사람이 읽을 수 있는 것이 있었다. 개발 서버 75만여 건·운영 서버 7만여 건의 권한을 한꺼번에 정리하고, 비밀 파일 96건을 주인만 읽게 바꾼 뒤, 다시 생기지 않도록 위의 "새 파일은 처음부터 나만"과 "매시간 점검"을 넣었다. 모든 장치는 같은 테스트 계정으로 **막혀야 할 것이 막히는지** 다시 확인했다.

이 장치가 "스스로 하게" 만드는 이유

확인을 묻는 대신,
되돌릴 수단을 남긴다

처음 하는 사람을 배포하게 만드는 것은 서버 지식이 아니다. "잘못돼도 되돌릴 수 있다"는 사실이다.

1. 묻지 않고, 되돌릴 수 있게 한다

예전에는 배포 중 "DB 구조를 바꿀까요? (y/N)"를 물었다. 이 질문은 아무것도 되돌려 주지 못하고 배포만 멈췄다 — 처음 하는 사람은 무엇을 기준으로 답할지 모른다. 그래서 질문을 없애고, 바꾸기 직전에 스냅샷을 찍게 했다. 스냅샷을 못 찍으면 그때만 멈춘다.

시작하는 사람에게는 — 판단할 수 없는 질문 대신, 잘못돼도 돌아갈 곳이 있다.

2. 관리자가 병목이 되지 않는다

주소 열기·DB 만들기·인증서·배포를 관리자에게 부탁하던 일을 각자 검증된 명령으로 직접 한다. 관리자는 서버를 키우거나 정책을 정하는 일에만 남는다. 10월 기록의 배포·정지·재시작은 모두 관리자가 아닌 계정이 직접 했다.

시작하는 사람에게는 — 내가 만든 것을 내가 올리고, 내가 고치고, 내가 정리한다.

3. 같이 써도 안전하다

자원은 앱마다 나뉘고, 데이터와 비밀은 주인만 본다. 한 사람의 실수가 다른 사람의 앱과 데이터로 번지지 않는다.

시작하는 사람에게는 — 다른 사람에게 피해를 줄까 봐 망설이지 않아도 된다.

05+

운영 안 들여다보기

서버 헬퍼 명령은 각각 무엇을 하나

사람과 AI 가 서버에서 쓰는 명령은 모두 `brie-` 로 시작하는 헬퍼다. 원본은 팀 wiki(`scripts/devteam/`)에 있고 서버에 설치된다 — 문서와 실행 코드가 한곳에 있어 서버를 다시 세워도 같은 결과가 나온다. 다음 세 쪽에서 하나씩 본다.

운영 안 ① 개발 서버·계정

자리를 고르고, 주소를 열고, 사람을 들이는 명령

brie-ports 읽기 전용 빈 포트 찾기·확인

누가·언제	누구나, 관리자 권한 없이 — 프로젝트를 만들 때(`create-project`)
하는 일	1. 웹서버 설정·예약 목록·운영 등록부·지금 쓰는 포트를 합쳐 실제 사용 현황을 만든다 2. `--next` 로 비어 있는 포트 하나를 고른다(개발 8000~8199, 운영 9000~9499)
안전장치	정본은 서버 실물 — wiki 의 포트 표는 이 명령이 자동으로 만드는 사본이라 손으로 고치지 않는다

brie-dev-expose · unexpose 팀원 허용 개발 앱에 HTTPS 주소 열기·닫기

누가·언제	본인 — `/dev` 가 처음 서버를 켤 때
하는 일	1. 도메인 등록 후 실제로 퍼졌는지 확인(최대 3분) 2. 공유 인증서에 주소 추가(기존 주소는 모두 유지) 3. 웹서버 연결 생성(소유자 기록) → 설정 검사 → 반영
안전장치	이름·포트 범위 검사, 남의 주소·포트·관리자 예약은 거부, 설정 검사 실패 시 원상 복구, 닫기는 본인 것만

brie-onboard-member · add-dev-user 관리자 새 사람 들이기

누가·언제	관리자 — 한 줄
하는 일	1. 개발 서버 계정(그룹·키·DB 역할·상시 서비스 권한) 생성 2. 운영 서버 계정 생성 3. 개발 → 운영 접속까지 확인
안전장치	여러 번 실행해도 결과가 같다(멱등), 키 형식 검사, 서버 역할(개발/운영)은 설정 파일로만 판정

brie-autostop · brie-keep 자동 밤에 안 쓰면 개발 서버 끄기

누가·언제	자동 — 밤 8시 반부터 아침 6시까지 30분마다
하는 일	1. 접속·예약 작업·켜 둠 표시가 있는지 본다 2. 없으면 백업 → 알림 → 끄기
안전장치	"켜 둠"(`brie-keep`) 표시로 밤샘 작업을 지킨다, 켜질 때도 알림

운영 안 ② 운영 서버 — 셀프 배포

자리 만들기·배포·관리·정리, 네 개의 명령

brie-prod-init 팀원 허용 운영 서버에 내 앱 자리 만들기(처음 한 번)

누가·언제	본인 — `/release` 첫 배포 때
하는 일	1. DB 와 DB 계정(무작위 비밀번호를 본인만 읽는 파일에 보관) 2. 서비스(본인 계정으로 실행, 메모리 512MB 상한, 권한 상승 금지) 3. 도메인 → 웹서버 연결 → HTTPS 인증서 4. 등록부에 소유자 기록
안전장치	바꾸기 **전에** 포트·서비스·주소 중복을 모두 검사, 중간에 실패하면 만든 것을 스스로 정리, 메모리가 부족하면 경고와 관리자 메일(작업은 진행)

brie-prod-deploy 팀원 허용 배포 — 스냅샷·확인·자동 롤백

누가·언제	본인 — `/release` 할 때마다
하는 일	1. 코드 받기 → 설치 2. DB 구조가 바뀌면 스냅샷(최근 5개) → 반영 3. 정적 파일 → 재시작 → 상태 확인(2초 간격 20번) 4. 실패하면 직전 버전으로 되돌리고 다시 켬
안전장치	스냅샷을 못 찍으면 DB 를 바꾸지 않음, DB 복원 명령을 알려 줌, 본인 앱만

brie-prod-svc 팀원 허용 내 앱 상태 보기·재시작·기록 보기

누가·언제	본인 — 언제든
하는 일	1. 상태 · 재시작 · 기록 보기 · 정지 · 시작 · "켜 둠" 표시
안전장치	본인 앱만

brie-prod-remove 팀원 허용 내 앱 운영에서 내리기

누가·언제	본인 — 더 이상 운영하지 않을 때
하는 일	1. 서비스 · 웹서버 연결 · 인증서 · 도메인을 지운다
안전장치	**DB 와 코드는 남긴다**(DB 삭제는 관리자에게 따로), 확인 표시(`--yes`) 필수

운영 안 ③ 감시·정리·보호

아무도 신경 쓰지 않아도 서버가 스스로 챙기는 일

brie-prod-idle 자동 방치 앱 정리

누가·언제	자동 — 매일 아침
하는 일	1. 자기 주소에서 온 실제 사용(사람의 화면 이동·입력)만 센다 — 검색 로봇은 제외 2. 30일 없으면 경고, 44일이면 자동 정지
안전장치	코드·DB·주소는 남겨 한 줄로 다시 켬, 알림 메일

brie-prod-capacity 자동 서버 용량 감시

누가·언제	자동 매시간 · 누구나 조회
하는 일	1. 남은 메모리와 앱 상한 합계를 본다 2. 기준(남은 메모리 400MB 미만 또는 상한 합계 75% 초과)을 넘으면 경고
안전장치	작업은 막지 않고 관리자에게 알림(하루 한 번)

brie-secret-perm-guard 자동 비밀 파일 점검

누가·언제	자동 — 매시간
하는 일	1. 모든 홈에서 설정 파일·토큰·키·인증서·DB 덤프·접속 정보 파일을 찾는다 2. 주인 외에게 읽히면 주인만 읽게 바로잡고 기록
안전장치	기본 설정을 거치지 않는 복사·압축 경로의 안전망

db_backup · 배포 기록 자동 백업과 기록

누가·언제	자동 — 매일 / 모든 운영 명령
하는 일	1. 전체 DB 를 매일 백업해 7일 보관, 서버 바깥(S3)에도 사본 2. 운영 서버의 배포·시작·정지·삭제를 누가·언제·무엇을로 기록
안전장치	앞의 "실제 기록" 쪽이 이 기록이다

06

벽 6 · 혼자 배운 건 혼자만 안다

한 사람이 겪은 일이 다음 사람의 출발점이 된다

혼자 만들면 시행착오는 그 사람 머릿속에만 남는다. 다음 사람은 같은 곳에서 또 넘어지고, AI 는 대화가 끝나면 잊는다. standarda 는 한 프로젝트에서 배운 것을 **기록하고, 모으고, 다음 작업이 시작될 때 자동으로 꺼내 주고, 기본값으로 올린다.**

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 여섯 번째 장치.

벽 — 혼자 만들면 생기는 일

고친 사람만 알고,
다음 사람은 또 넘어진다

여러 사람이 각자 업무 자동화를 만들기 시작하면 같은 함정이 프로젝트마다 반복된다. 시행착오는 쌓이는데 지식은 쌓이지 않는다.

같은 함정에 각자 빠진다
엑셀 업로드, 메일 줄바꿈, AI 답 해석 — 프로젝트마다 처음부터 다시 겪는다

고친 사람만 안다
원인을 찾느라 한나절을 썼지만 그 답은 그 사람 기억과 그 프로젝트 코드에만 있다

AI 는 대화가 끝나면 잊는다
어제 AI 와 함께 고친 이유를 오늘의 AI 는 모른다. 같은 실수를 같은 방식으로 다시 만든다

적어 둬도 아무도 안 찾는다
위키에 교훈을 적어도, 작업할 때 꺼내 보지 않으면 없는 것과 같다

잘 만든 것도 퍼지지 않는다
한 프로젝트에서 잘 만든 기능을 다른 프로젝트는 모르고 다시 만든다

고친 것이 다시 사라진다
출발점을 고쳐도 옮기거나 정리하는 과정에서 빠지면, 고쳤던 결함이 새 프로젝트에 돌아온다

개인의 경험은 팀의 자산이 되지 않는다 — 장치가 없으면

처음 하는 사람이 스스로 해내려면, 앞 사람들이 이미 겪은 일을 **공부하지 않아도** 갖고 시작할 수 있어야 한다. 그러려면 배운 것이 저절로 도는 고리가 필요하다.

standarda 의 장치

배운 것이 도는 고리 다섯 개

오류를 고친 순간부터 그 교훈이 다른 프로젝트의 출발점이 될 때까지, 각 단계를 맡는 장치가 있다. 사람이 기억해서 챙기지 않아도 고리가 돈다.

① 기록
고친 그 자리에서
원인·해결·교훈을 이슈 문서로. 교훈을 어디까지 퍼뜨릴지 4층 중 하나로 분류. 빠뜨리면 검사기가 커밋을 막는다(벽 4)

② 모음
팀 wiki
교훈 436개, 53개 프로젝트에서 나왔다. 한 줄 색인마다 "언제 보면 되나"를 적어 둔다

③ 자동 조회
다음 작업 시작 때
`/develop` 이 큰 작업을 시작하면 `lesson-finder` 가 이번 작업에 맞는 교훈만 골라 온다

④ 기본값 승격
두 번 겪으면
규칙집·검사기·출발점(template)·공용 부품(core)으로 올린다. 변경 기록 114건, 건마다 교육 포인트

⑤ 다시 퍼뜨리기
기존 프로젝트로
출발점 → 기존 프로젝트 반영 스킬, core 버전 올리기. 끊기는 곳은 백로그로 모아 매주 정리

교훈이 가는 곳	언제	효과
이슈 문서(이 프로젝트)	모든 비자명한 오류 고침	이 프로젝트의 다음 사람·다음 AI 가 읽는다
wiki 교훈 + 색인	다른 프로젝트도 겪을 만한 것(공통·분야별)	다른 프로젝트의 `lesson-finder` 가 찾아 준다
규칙집·검사기	모든 프로젝트가 반드시 지켜야 할 것, 재발 2회	모든 AI 가 처음부터 따르고, 어기면 막힌다
출발점·공용 부품	코드로 해결할 수 있는 것	새 프로젝트는 처음부터 갖고, 기존 프로젝트는 버전을 올려 받는다

실제로 돌려 본 장면 — lesson-finder

"엑셀을 AI 로 분석해 시트에 쓰는 기능"을 시작하자, 7개 프로젝트의 교훈 10개가 따라왔다

standarda 로 만든 새 프로젝트에서 실제 `lesson-finder` 에이전트(가장 가벼운 모델)에게 작업 하나를 주고 결과를 받았다. 걸린 시간 103초. 아래는 돌려준 10개를 한 줄씩 옮긴 것이다(원문은 교훈마다 "이번 작업에 적용할 방법"과 원문 경로가 함께 온다).

#	분야	골라 온 교훈	처음 겪은 프로젝트
1	sheets	업로드 엑셀은 구형 `.xls` 도 받아야 한다 — openpyxl 단독은 실서비스에서 깨짐	receipt-tracker
2	sheets	엑셀 열기 실패(`BadZipFile` 등)를 사용자용 오류로 바꿔라 — 파일 앞머리 검사만으론 부족	cp_revshare
3	sheets	사용자가 올리는 엑셀은 열 **위치**가 아니라 **머리글 이름**으로 매핑하라	cp_revshare
4	general	건당 비용이 드는 AI 처리 대상은 와일드카드가 아니라 허용 목록으로	knowclaw
5	general	사용자가 다시 만들 수 없는 입력(올린 파일)은 외부 호출 전에 먼저 저장하라	english_study
6	general	AI 에 표를 넘길 땐 구조(머리글·행·열)를 살려 보내라	codepresso_marketing_agent
7	llm	여러 행을 한 번에 보내고 되돌릴 때 개수·번호가 어긋나면 묶음 전체를 버려라 — 한 칸씩 밀려 조용히 저장된다	english_study
8	llm	출력이 상한에 닿으면 잘린 것 — 정상 종료로 보고되니 직접 감지하라	golden
9	general	AI 답의 JSON 해석 실패는 정상 케이스 — 정해진 방식으로 복구하고 부분 성공을 버리지 마라	codepresso_marketing_agent
10	google	구글 토큰 경로를 설정에 명시하라 — 안 하면 프로세스마다 다른 계정으로 시트에 쓴다	popax

원문 한 개 그대로 — 7번째 교훈

```
- [llm] 여러 항목을 한 번에 LLM 에 보내고 되매핑할 때, 개수·번호가 어긋나면 부분 저장하지 말고 묶음 전체를 버려라
  · 적용: Excel 행들을 묶음(batch)으로 보내 분석 결과를 행별로 되돌려 시트에 쓸 경우, 응답 개수가 입력과 다르거나 번호(`n`)가 `i+1` 과 어긋나면 그 묶음은 통째로 실패 처리하고 사유를 남긴다. 앞에서부터 채우는 방식은 한 칸씩 밀린 결과를 조용히 시트에 쓴다. 되매핑 검증은 에러 없이 통과하므로 반드시 명시적으로 확인한다.
  · 원문: wiki/lessons/llm/batch-item-remap-refuse-on-misalignment.md (출처 english_study, 본문 기준)
```

에이전트는 끝에 "보류한 후보 5개"와 "제목 훑기로 좁혔으므로 누락 가능성이 있다"까지 밝혔다. 처음 하는 사람은 이 10가지 중 대부분을 **문제가 생긴 뒤에야** 알았을 것이다.

실제로 한 바퀴 돈 사례 (1·2)

한 프로젝트의 문제가 모든 프로젝트의 부품이 되기까지

① AI 캐시가 거의 안 먹던 문제 → 공용 부품이 알아서 처리

겪음 · popax#39
같은 지시문을 반복해 보내는 AI 도구 호출에서 캐시 적중률이 3% 미만 — 비용이 그대로 나감. 원인은 캐시 표시를 붙이는 위치

기록 · wiki 교훈
"도구 호출 반복에서 캐시는 지시문뿐 아니라 쌓이는 대화 끝에도 경계를 둬야 한다"(llm-chat)

승격 · 2026-06-21 core v0.13.0
캐시 표시와 비용 집계를 묶은 공용 부품 — 함께 출발점에 "쓰는 법" 안내(template)

승격 · 2026-07-07 core v0.16.0
AI 도구 호출 뼈대를 공용 부품으로 — 캐시 경계를 **내장**. 복사해 쓰던 코드는 고쳐도 퍼지지 않으므로 공용 부품을 정본으로

퍼짐
이 뼈대를 쓰는 모든 프로젝트가 캐시를 따로 신경 쓰지 않는다 (벽 2)

② 메일 줄바꿈이 사라지던 문제 → 두 번째에 규칙이 됐다

겪음 · popax#49
알림 메일 본문이 한 줄로 뭉개짐 — HTML 메일에 일반 줄바꿈을 넣은 것

또 겪음 · popax#57
같은 일이 다른 곳에서 다시

기록 · wiki 교훈
"메일 본문이 HTML 이면 줄바꿈은 <br>, 사용자 입력은 escape"

승격 · 2026-08-08 core v0.25.0
알림 메일 HTML 을 만드는 공용 부품 `render_email_html`

강제 · 2026-08-08 template
규칙집에 "알림 메일은 이 부품으로" — 교육 포인트: **재발 2회면 교훈을 코드 규칙으로**, 승격(core)과 강제(template)는 쌍

실제로 한 바퀴 돈 사례 (3·4)

리허설의 실수는 다음 날의 기본값이 되고, 빠진 개선은 다시 찾아 넣는다

③ 교육생이 등록 화면만 만들던 문제 → 견본이 곧 표준

겪음 · 2026-11 FDE 교육 리허설
교육생들이 만든 업무 앱에 등록 기능만 있고 목록·수정·삭제가 없었다. 출발점의 견본 앱이 등록 기능 하나뿐이었고, AI 는 그 견본을 따라 만들었다

승격 · 2026-11-09 template
견본 업무 앱을 목록·상세·등록·수정·삭제·검색·테스트까지 갖춘 형태로

교육 포인트
"견본이 곧 표준 — AI 는 기존 코드를 따라 만든다"

퍼짐
그 뒤 만드는 모든 프로젝트는 처음부터 완전한 견본을 갖고 시작한다 (벽 1·3)

④ 고친 것이 이전 중에 빠졌다 → 다시 넣고, 빠지지 않는 장치를 만들었다

겪음 · 2026-08-31 popax 외 5개 계정
작업 공간에서 서버를 띄우면 올린 파일이 엉뚱한 폴더에 쌓임 — 실측 10건·2,732파일. 출발점을 고쳤다(이전 저장소 #77)

빠짐 · 2026-09 저장소 이전
다른 회사 저장소에서 Brie 저장소로 옮길 때 이 변경이 브랜치로만 넘어와 새 출발점에서 빠졌다. 아무 신호가 없었다

발견 · 2026-11-11
이 소개 자료를 쓰려고 출발점을 해부하다 발견

다시 넣음 · 2026-11-11 template
변경을 다시 반영하고, 테스트를 기본 테스트 방식으로 바꿔 실제로 돌게 했다

장치 · 2026-11-13
개선 거리를 한곳에 모으는 백로그와 매주 금요일 정리, "저장소 이전 땐 머지된 변경과 새 정본을 대조" 체크리스트

④는 고리가 끊긴 사례다. 고리도 끊길 수 있다 — 그래서 끊긴 곳을 찾는 일(수집·정리·대조)까지 장치로 만들었다.

이 장치가 "스스로 하게" 만드는 이유

혼자 공부한 시간이 팀의 기본값이 된다

처음 하는 사람이 혼자 해낼 수 있는 이유는 그가 특별해서가 아니라, 앞 사람들의 시행착오를 갖고 시작하기 때문이다.

1. 공부하지 않아도 갖고 시작한다

앞의 시연에서 "엑셀을 AI 로 분석하는 기능"을 시작하자 7개 프로젝트의 교훈 10개가 103초 만에 따라왔다. 처음 하는 사람이 혼자 겪으며 알았다면 몇 주가 걸렸을 것들이다. 이 교훈들은 사람에게 읽히려고가 아니라 **AI 가 작업에 반영하도록** 쓰여 있다.

시작하는 사람에게는 — 53개 프로젝트의 시행착오가 내 첫 작업에 들어온다.

2. AI 는 잊어도 시스템은 기억한다

AI 는 대화가 끝나면 잊는다. 그래서 기억을 AI 에게 맡기지 않고 이슈 문서·교훈·규칙집·출발점·공용 부품에 남긴다. 다음 대화의 AI 는 그것부터 읽는다.

시작하는 사람에게는 — 어제의 AI 와 오늘의 AI 가 같은 것을 안다.

3. 끊기는 곳도 장치로 잇는다

교훈을 적어 놓고 색인에 안 올리면 아무도 못 찾는다(이 장을 쓰다 11개를 발견해 백로그에 올렸다). 고친 것이 이전 중에 빠지기도 한다(사례 ④). 그래서 개선 거리를 한곳에 모으고, 매주 금요일 정리하고, 완료 기준에 "기존 프로젝트까지 판단"을 넣었다.

시작하는 사람에게는 — 누군가 겪은 일은 결국 내 출발점까지 온다.

06+

고리 안 들여다보기

교훈은 어떤 모양으로 남고, 찾히고, 올라가나

다음 세 쪽에서 고리의 각 단계를 실제 문서 모양으로 본다 — 이슈 문서와 전파 4층, 교훈 파일과 색인, 그리고 출발점·공용 부품으로 올라가 기존 프로젝트까지 가는 길.

고리 안 ① 기록 — 이슈 문서와 전파 4층

오류를 고치면 교훈이 갈 곳까지 정한다

비자명한 오류를 고치면 같은 커밋에 `docs/issues/번호_제목.md` 를 넣는다(빠지면 검사기가 막는다 — 벽 4). 형식은 정해져 있다.

```
# Issue #{번호}: {제목}
## 증상       사용자가 겪은 현상
## 원인       근본 원인 — 코드 수준
## 해결       고친 내용
## 수정 커밋  커밋 번호와 요약
## 교훈       같은 실수를 반복하지 않기 위한 원칙
## 전파       이 교훈을 어디까지 퍼뜨릴지 — 아래 4층 중 하나
```

층	기준	교훈이 가는 곳
1 코드 규칙	모든 프로젝트가 반드시 지켜야 하는 것	규칙집(CLAUDE.md)의 코드 규칙 — 필요하면 검사기로 강제
2 공통 교훈	분야와 상관없이 다른 프로젝트도 겪을 것	wiki `lessons/general/` + 색인 한 줄
3 분야 교훈	엑셀·메일·AI 대화·배포 등 특정 분야에서 겪을 것	wiki `lessons/<분야>/` + 색인 한 줄
4 이 프로젝트만	이 프로젝트의 업무에만 해당	이슈 문서에만 남긴다

wiki 는 관리자만 고친다. 팀원 프로젝트에서 2·3층이 나오면 이슈 문서에 "wiki 교훈 후보"로 남기고 관리자에게 알린다 — 관리자가 옮긴다.

고리 안 ② 모음과 조회 — 교훈 파일·색인·lesson-finder

교훈은 AI 가 찾을 수 있게 "언제 보면 되나"와 함께 남는다

분야	교훈 수	분야	교훈 수
general(공통)	350	llm-chat(AI 대화)	26
deploy(배포)	24	llm(AI 호출)	13
sheets·google·audio	5·4·4	그 밖 5개 분야	10

합계 436개 · 출처 프로젝트 53개(가장 많은 곳 popax 45개)

교훈 파일 하나

```
---
domain: sheets                          ← 분야
title: 업로드 엑셀은 머리글 이름으로 매핑
trigger: 사용자가 올리는 엑셀을 읽는 기능을 만들 때    ← 언제 보면 되나 (조회의 열쇠)
source: cp_revshare#10                  ← 처음 겪은 곳
promoted_to: (규칙집으로 올라갔으면 적음)
---
## 무슨 일이 있었나 / ## 일반화된 교훈 / ## 적용 방법 / ## 출처
```

색인 한 줄 → lesson-finder

교훈마다 색인(`lessons/INDEX.md`)에 **분야 · 제목 · 적용 조건 · 출처 · 경로** 한 줄이 붙는다. `lesson-finder` 는 가장 가벼운 모델로 돌며 ① 색인만 읽고 ② 공통 + 이번 작업 분야 행 중 적용 조건이 맞는 것만 골라 ③ 걸린 교훈의 본문만 열어 ④ "이번 작업에 적용할 방법"으로 바꿔 돌려준다. 색인은 늘, 본문은 걸린 것만 — 교훈이 늘어도 비용이 폭주하지 않는다.

그래서 색인 한 줄이 빠지면 그 교훈은 영영 안 찾힌다 — 이 장을 쓰며 11개를 찾아 정리 거리로 올렸다.

고리 안 ③ 승격과 전파 — 출발점·공용 부품·기존 프로젝트

기본값으로 올라간 것은 기록과 함께 모든 프로젝트로 간다

올리기

경로	무엇을
프로젝트 → 출발점	`/sync-to-template` — 한 프로젝트에서 좋아진 규칙·스킬·검사기·공통 코드를 출발점과 비교해 항목별로 골라 올린다(프로젝트 고유값은 자리표시로)
프로젝트 → 공용 부품	검증된 기능을 core 로 — 받아쓰기·슬라이드·결과 검증·메일 HTML 이 이렇게 올라왔다(벽 2)

기록하기 — 변경 기록 114건, 건마다 교육 포인트

```
| 2026-11-09 | template | #36 | projects 를 CRUD 견본 앱으로(목록·상세·등록·수정·삭제·테스트) …
|            |          |     | 📚 견본이 곧 표준(AI 는 기존 코드를 따라 만든다) | 발단: FDE 리허설 |
```

출발점 93 · 공용 부품 20 · 서버 헬퍼 1. "무엇이 바뀌었나"뿐 아니라 "왜·무엇을 배웠나"를 남겨 리뷰와 교육 자료의 원천이 된다.

기존 프로젝트로 퍼뜨리기

경로	무엇을
출발점 → 기존 프로젝트	`/sync-from-template` — 네 등급으로 골라 가져온다: 일하는 방식·도구(적극) / 환경 파일(판단) / 규칙집 공통 부분(줄 단위로 합침) / 데이터 구조(절대 덮지 않음)
공용 부품 → 기존 프로젝트	core 버전 번호만 올린다 — 쓰는 쪽 코드는 그대로(벽 2)
끊긴 곳 잇기	개선 거리를 한곳(standarda 백로그)에 모으고, 매주 금요일 정리, 완료 기준 5가지 중 하나가 "기존 프로젝트 반영 판단"

07

벽 7 · 우리 회사 인프라에서도?

우리 회사의 서버·계정·도메인 위에 같은 체계가 그대로 선다

여기까지 본 출발점·공용 부품·일하는 방식·검사기·서버 헬퍼·교훈은 Brie 의 서버에서만 도는 것이 아니다. 회사마다 다른 값은 **값 파일 하나**에 모으고, 코드는 하나로 둔다. 그래서 고객 회사가 **자기 명의의** AWS·GitHub·도메인 위에 같은 체계를 세우고, 업데이트를 계속 받으며, 혼자 운영한다.

> Brie 는 AX 를 대신 해 주지 않는다. 고객이 스스로 하게 만든다.
> — 그 일곱 번째 장치.

벽 — "우리 회사에서도 되나"

남의 서버에서만 도는 체계는 우리 것이 되지 않는다

업체의 서버와 계정 위에서 만든 업무 자동화는 업체와의 관계가 끝나는 날 같이 멈춘다. 그렇다고 우리 서버에 옮겨 세우려면 또 다른 전문가가 필요하다.

우리 명의로 가질 수 있나
서버·코드·데이터·도메인이 업체 계정에 있으면, 계약이 끝날 때 무엇이 남는지 알 수 없다

옮기면 어디가 깨지나
남의 도메인·계정·조직 이름이 설정과 문서 곳곳에 박혀 있다. 하나라도 놓치면 엉뚱한 곳을 가리킨다

옮긴 뒤 업데이트는?
복사해 간 순간부터 원본과 갈라진다. 원본이 고친 결함과 개선을 다시 받을 길이 없다

우리 사정은 다르다
밤에 서버를 꺼서 비용을 줄일지, 늘 켜 둘지. 회사마다 사정이 다른데 그때마다 코드를 고쳐야 하나

제대로 옮겨졌는지 어떻게 아나
겉으로 돌아간다고 같은 것이 아니다. 권한·백업·자동 정리 같은 보이지 않는 장치가 빠져도 모른다

처음 세우는 일이 너무 많다
AWS 계정, 서버, 계정·권한, GitHub 조직, 도메인과 인증서 — 처음 하는 회사는 어디서부터 해야 할지 모른다

"스스로 하게"는 서버까지 우리 것이어야 끝난다

앞의 여섯 장치가 아무리 좋아도 남의 서버에서만 돈다면 고객은 그 업체를 떠날 수 없다. 체계 전체가 **고객의 인프라로 옮겨 갈 수 있어야** 고객이 스스로 하는 것이다.

standarda 의 장치

코드는 하나, 값은 박스마다

회사마다 달라야 하는 것은 전부 박스마다 한 파일(`/etc/brie/site.env`)에 모았다. 출발점·개발 도구·서버 헬퍼는 모두 이 파일을 읽는다. 다른 회사에 세울 때 바꾸는 것은 이 파일뿐이다.

장치	하는 일	그래서
① 값 파일 하나 `site.env`	회사 이름, GitHub 조직, 도메인, 서버 주소, 알림 받을 곳, 밤에 끌지 늘 켤지 — 20개 값. 비밀값은 넣지 않는다	코드를 고치지 않고 값만 바꿔 다른 회사에 선다. 값이 비면 추측하지 않고 멈춘다
② 공용 자산 받기 `brie-upstream-mirror`	15분마다 Brie 의 출발점·공용 부품·개발 도구를 고객 조직으로 그대로 가져온다(기본 브랜치와 버전 태그만, 덮어쓰기 없이)	옮겨 간 뒤에도 개선이 계속 온다. 고객 쪽 값은 값 파일에 있으니 받는 코드를 손볼 필요가 없다
③ 같은지 검사 `brie-parity-check`	박스의 작동 방식(권한·계정·메모리 상한·헬퍼·예약 작업·웹서버·DB)을 회사 값과 무관한 목록으로 뽑는다. 두 박스를 나란히 비교한다	"돌아간다"가 아니라 "같다"를 확인한다. 다른 줄은 의도한 차이인지 하나씩 본다
④ 세우는 절차 FDE 온보딩 플레이북	고객 명의의 AWS·GitHub·도메인 위에 세우는 순서. 고객이 할 일 6가지와 우리가 할 일을 나누고, 실제로 겪은 함정 16개를 단계마다 붙였다	처음 하는 회사도 정해진 순서대로 간다. 같은 함정에 두 번 빠지지 않는다

Brie 의 박스

회사 `brie` · 도메인 `brie.co.kr`
GitHub 조직 `brie-org` · 밤에 자동 정지
공용 자산의 원본

다른 회사(다웰)의 박스

회사 `dawel` · 도메인 `dawelservice.com`
GitHub 조직 그 회사 쪽 조직 · 늘 켜 둠
Brie 에서 15분마다 받아 옴

두 박스의 프로그램은 같다. 위 값만 다르다.

실례 — 다른 회사에 세운 2박스

다른 회사의 dev·prod 서버를 하루 만에 같은 체계로 맞췄다

다웰은 자기 AWS 계정에 개발 서버와 운영 서버를 두고 standarda 로 일한다. 2026-08 첫 구축 때는 회사 값을 코드에 직접 바꿔 넣는 방식이었다. 10월에 값 파일이 생기자, Brie 와 같은 체계로 하루 만에 다시 맞췄다.

2026-08-24 · 첫 구축
다웰 교육생 5명이 자기 회사 서버에서 FDE 교육을 받았다. 이때는 회사 값을 파일마다 바꿔 넣는 방식

2026-08-27
운영(prod) 서버 신설

2026-10-26 17:09 · 맞추기 시작
기존 상태 보존(되돌릴 길 확보)

20:40 ~ 20:57
서버 설정·계정·권한을 Brie 방식으로

21:02 ~ 21:15
웹서버를 Brie 와 같은 것으로 교체 — 실제 전환은 6초

21:20 ~ 21:45
공용 자산을 그 회사 조직으로 옮김(공용 부품 버전 태그 29개), 팀원 프로젝트 5개 이전

같은 날 · 운영 서버도
운영 서버도 Brie 운영 서버 방식으로 다시 세움

그 뒤 지금까지
15분마다 자동으로 받아 옴 — 10-26 첫 반영부터 11-13 까지 31번 갱신이 들어왔다

20	10	11	32
회사 값(값 파일)	같은 서버 헬퍼	그 서버의 팀원 계정	돌아가는 개발 앱

2026-11-14 다웰 개발 서버 실측. 오늘 기준 그 회사 조직의 출발점·공용 부품·개발 도구는 Brie 의 최신과 같은 버전이다.

실례 — 같은지 확인

"돌아간다"가 아니라 "같다"를 숫자로 확인했다

맞추기를 끝낸 뒤 Brie 와 다웰의 박스에서 같은 검사(`brie-parity-check`)를 돌려 나란히 비교했다(2026-10-27). 다른 줄은 하나씩 이유를 확인했다.

비교	검사 항목	완전히 같은 줄	다른 줄은 무엇이었나
개발 서버 ↔ 개발 서버	84	56	밤 정지(Brie) vs 늘 켬(다웰)에 딸린 도구·예약 작업, 받아 오기 도구(다웰에만), 팀원 이름, 메모리 상한, 앱 수
운영 서버 ↔ 운영 서버	84	72	팀원, 알림 경로, 등록된 서비스 수

다른 줄은 전부 "회사 사정"이었다

서버 헬퍼 프로그램은 두 박스에서 지문(해시)까지 같았다. 남은 차이는 값 파일로 고른 운영 방식(밤에 끌지 늘 켤지)과 그 회사 사람·서비스 수뿐이다. 빠진 장치가 없다는 것을 눈대중이 아니라 검사로 확인한 것이다.

옮기면 빠지는 것 — 실제로 겪은 일

이 체계 자체도 2026-09 다른 회사의 인프라에서 Brie 의 AWS·도메인·GitHub 로 통째로 옮겨 왔다. 코드와 데이터는 다 왔는데, **옮긴 직후 DB 백업이 0개**였다. 백업·자동 서버 이미지·권한 역할 같은 클라우드 쪽 자동화는 복사로 따라오지 않는다. 문서에는 백업 절차가 있어 "있다"고 착각하기 쉬웠다.

그래서 — 옮긴 뒤 "다시 만들어야 하는 것" 체크리스트와 같은지 검사가 절차에 들어갔다.

이 장치가 "스스로 하게" 만드는 이유

체계가 고객 소유가 되어야 고객이 스스로 한다

고객이 스스로 하려면 도구뿐 아니라 그 도구가 도는 자리까지 고객의 것이어야 한다.

1. 처음부터 고객 명의다

AWS 계정·GitHub 조직·도메인·결제는 처음부터 고객이 만들고, 구축하는 쪽을 초대한다. 구축이 끝나면 초대받은 쪽이 조직에서 빠지기만 하면 된다 — 옮겨 줄 것이 없다.

고객에게는 — 업체와의 관계가 끝나도 서버·코드·데이터는 그대로 우리 것이다.

2. 옮긴 뒤에도 같이 좋아진다

값과 코드를 나눴기 때문에, Brie 가 출발점과 공용 부품을 고치면 15분 안에 고객 조직에 들어온다. 앞 장의 교훈 고리(벽 6)가 고객 회사까지 이어진다.

고객에게는 — 혼자 운영해도 다른 회사들의 시행착오가 계속 들어온다.

3. 회사 사정은 값으로 고른다

비용을 아끼려 밤에 서버를 끌지, 교육이 많아 늘 켜 둘지, 알림을 어디로 받을지 — 코드를 고치지 않고 값 파일에서 고른다. 고객이 직접 바꿔도 다른 곳이 깨지지 않는다.

고객에게는 — 우리 회사에 맞게 바꾸는 일도 전문가 없이 할 수 있다.

07+

인프라 안 들여다보기

값 파일, 받아 오기, 세우는 순서는 실제로 어떻게 생겼나

다음 세 쪽에서 값 파일의 20개 값과 그것을 읽는 곳, 공용 자산을 받아 오는 방식과 같은지 검사, 그리고 고객 인프라 위에 세우는 순서와 실제로 겪은 함정을 본다.

인프라 안 ① 값 파일 — 20개 값과 읽는 곳

회사가 바뀌면 이 20줄만 바뀐다

값	뜻
`SITE_ID` · `BOX_ROLE`	회사 짧은 이름 · 개발/운영 서버 구분
`ALERT_TAG` · `SNS_TOPIC_ARN`	알림 메일 제목 표시 · 알림 보낼 곳(비면 다른 경로로)
`RUN_MODE`	밤에 자동 정지 / 늘 켬
`GITHUB_ORG` · `UPSTREAM_GITHUB_ORG`	이 회사의 GitHub 조직 · 공용 자산을 받아 올 상위 조직(비면 받아 오지 않음)
`ADMIN_GITHUB_USER` · `ADMIN_LINUX_USER`	관리자 GitHub 계정 · 서버 관리자 계정
`DEV_DOMAIN` · `PROD_DOMAIN` · `DEV_CERT_NAME`	개발 주소 `<앱>-dev.도메인` · 운영 주소 `<앱>.도메인` · 개발 인증서 이름
`ROUTE53_ZONE_ID` · `DNS_AWS_PROFILE`	도메인을 관리하는 곳 · 그곳을 고칠 때 쓰는 AWS 프로필 이름
`DEV_IP` · `PROD_IP` · `PROD_ADMIN_SSH_KEY`	서버 주소 · 개발 서버에서 운영 서버로 접속하는 키의 위치
`AWS_REGION` · `AWS_ACCOUNT_ID` · `DB_BACKUP_S3_BUCKET`	AWS 지역 · 계정 번호 · DB 백업을 서버 밖에 보관할 저장소

비밀값(키·비밀번호)은 넣지 않는다 — 누구나 알아도 되는 공개 정보만. 비밀은 관리자만 읽는 파일과 서버 권한 역할에 따로 둔다.

누가 읽나

읽는 곳	어떻게
출발점(새 프로젝트 만들기)	GitHub 조직을 값 파일에서 읽어 공용 부품 주소와 새 저장소 위치를 채운다. 비어 있으면 만들기를 멈춘다
개발 도구	프로젝트 만들기·지우기·가져오기가 조직·도메인을 값 파일에서 읽는다. 없으면 "값 파일 없음"으로 멈춘다
서버 헬퍼	개발 주소 열기·운영 배포·알림·백업이 첫 줄에서 값 파일을 읽고 필요한 값이 비었는지 검사한다 — 기본값으로 조용히 넘어가지 않는다

인프라 안 ② 받아 오기와 같은지 검사

고객 쪽은 15분마다 최신이 되고, 다른 점은 목록으로 드러난다

받아 오기 — `brie-upstream-mirror`

고객 서버의 관리자 예약 작업으로 15분마다 돈다. 상위 조직(Brie)의 출발점·공용 부품·개발 도구·인프라 안내서에서 기본 브랜치와 버전 태그만 고객 조직으로 옮긴다. **앞으로만 나아가고 덮어쓰지 않는다** — 고객 쪽이 갈라져 있으면 멈추고 알림 메일 한 통, 복구되면 한 통.

```
$ cat ~/logs/upstream-mirror.log        # 다웰 개발 서버, 바뀐 것이 있을 때만 한 줄
2026-10-26 23:07:07 standarda-template: ee30237 → b8ff80d     ← 첫 반영
2026-10-27 00:11:51 infra-handbook: 없음 → 830ac6d
…
2026-11-13 15:37:07 standarda-template: 74466d4 → dd74865
2026-11-13 17:22:08 standarda-template: dd74865 → 9dc956d     ← 31번째
```

출발점 안의 모든 회사 값은 값 파일에서 오므로, 받아 온 코드를 고객 쪽에서 고칠 일이 없다 — 그래서 "그대로 옮기기"만으로 충분하다.

같은지 검사 — `brie-parity-check`

박스의 작동 방식을 `항목=값` 목록으로 뽑는다. 회사 이름·주소 같은 값은 빼고 "어떻게 도는가"만 남기므로, 같은 역할의 두 박스는 목록이 같아야 한다.

검사 묶음	무엇을 보나
값 파일	20개 값이 다 있는가(값 자체는 비교하지 않음)
서버 설정 · 계정 · 권한	새 파일 기본 권한, 팀원 그룹, 허용된 관리 명령 목록, 사람마다 메모리 상한
서버 헬퍼	설치된 헬퍼 프로그램과 지문(해시) — 같은 버전인가
예약 작업 · 웹서버 · DB	백업·권한 점검·정리 작업, 웹서버 종류와 주소 수, DB 버전
운영 서버	배포 도구·알림 경로·등록된 서비스

인프라 안 ③ 고객 인프라 위에 세우는 순서

고객은 6가지만 하고,
나머지는 정해진 순서대로 간다

순서	고객이 하는 것	세우는 쪽이 하는 것
1	회사 영문 이름(조직·도메인의 뿌리) 철자 확정	후보 이름이 GitHub·도메인에서 비어 있는지 미리 조회
2	AWS 계정 만들고 관리 권한 위임, 팀원 GitHub 계정 이름, AI 좌석·키 — 한 번에 회신	안내서 묶음을 회사 값으로 채워 전달
3	확정한 이름으로 GitHub 조직을 만들고 초대	팀·저장소·팀원 초대 세팅
4	팀원 각자 초대 수락 + 2단계 인증	서버 생성 → 계정·권한·DB → 값 파일 설치 → 공용 자산 받아 오기
5	도메인 구매 + 알려 준 네임서버 4개 입력	도메인 관리 영역 생성, 개발 주소용 와일드카드 연결
6	각자 로그인해서 첫 프로젝트 만들기	검사 통과 확인 → 인수인계(조직에서 빠지기, 권한 축소)

실제로 겪고 절차에 박아 둔 함정 (16개 중)

겪은 일	지금 절차
회사 이름 철자 미확정(다웰)	조직·도메인·서버·문서를 전부 다시 만들었다 → 철자부터 확정하고 시작
"비어 있음" 확인했는데 생성 때 이미 선점	확인과 생성 사이에 틈을 두지 않는다 — 확정 즉시 조직 생성
새 AWS 계정은 무료 플랜이라 서버가 안 뜸	유료 전환을 첫 단계로
새 계정의 서버 크기 한도(5 vCPU)	큰 서버가 필요하면 첫날 한도 증설 요청, 작은 서버로 먼저 시작
새 계정은 AWS 에서 도메인 등록이 막힘	국내 등록처에서 고객 명의로 사고 네임서버만 연결
교육생 여럿이 AI 를 동시에 돌리자 메모리 부족으로 서버가 멈춤	교육용은 큰 서버 + 사람마다 메모리 상한
Windows 담당자의 접속 키 권한 오류	Windows 를 먼저, 창 여는 법부터 안내

처음부터 배포까지, 한 번에

사람이 하는 말은 네 문장,
나머지는 장치가 한다

7개의 장치는 따로 도는 것이 아니라 한 흐름으로 이어진다. 업무 앱 하나가 생겨서 동료가 매일 쓰기까지, 사람이 하는 일과 시스템이 하는 일을 나란히 놓았다.

1 **"settle helper 만들어 줘" — 그리고 "/dev"** — 3분 24초

이름·DB·빈 포트를 정하고 표준 골격(파일 212개)으로 프로젝트를 만든 뒤, 실행 준비와 개발 주소·인증서 연결까지. 사람이 넣는 값은 DB 비밀번호 하나. **벽 1**

2 **"정산 화면에 결재 상태 추가해 줘"** — 승인 1번

격리된 작업 공간 → 관련 스펙·코드와 **교훈**을 먼저 읽고(벽 6) → 쉬운 말 5~7줄 요약을 **사람이 한 번 승인** → 스펙 작성 → 규칙대로 구현(AI 기능은 공용 부품으로, 벽 2) → 스펙과 코드 대조 → 점검 후 정본에 합침. **벽 3**

3 **(그 사이 모르고 한 실수)** — 저장 전

엉뚱한 저장소에 올리거나, 남의 작업이 합쳐진 정본을 되돌리거나 다시 쓰려는 순간 검사기가 이유와 함께 막고 고칠 방법을 알려 준다. **벽 4**

4 **"배포해 줘"** — 한 줄

운영 서버에 자리를 만들고(처음 한 번), DB 구조가 바뀌면 먼저 스냅샷 → 반영 → 상태 확인. 실패하면 코드는 스스로 이전 버전으로 되돌아오고, DB 를 되돌릴 땐 스냅샷 복원 명령을 알려 준다. **벽 5**

이 과정에서 겪은 일은 다음 사람에게 간다

오류를 고치면 이슈 문서와 교훈이 남고(벽 6), 같은 일을 두 번 겪으면 출발점·부품·검사기로 올라가 다음 프로젝트의 기본값이 된다. 이 모든 것은 고객 회사의 서버·계정 위에서도 같은 모양으로 돈다(벽 7).

지금까지의 기록

이미 돌고 있는 체계다

아래 숫자는 이 자료를 쓰며 실제 서버·저장소에서 다시 센 것이다(2026-11 기준). 자세한 근거는 각 장에 있다.

무엇	숫자	뜻
교훈	436개 · 53개 프로젝트	각 프로젝트가 겪은 일을 일반화해 남긴 것. 작업 시작 때 관련된 것만 골라 온다 (벽 6)
기본값으로 올라간 변경	114건	출발점·공용 부품·서버 도구가 바뀐 기록. 건마다 "무엇을 배웠나"가 붙어 있다 (벽 6)
개발 요청 흐름	7단계 · 승인 1번	모든 "만들어 줘·고쳐 줘"가 같은 순서로 돈다 (벽 3)
검사기	11개	실제 차단 장면 9건을 이 자료에 그대로 실었다 (벽 4)
배포 안전장치	스냅샷 · 상태 확인 · 자동 되돌리기	실제 운영 로그로 실패 → 자동 되돌리기를 확인했다 (벽 5)
다른 회사 인프라	2박스 · 값 20개	개발·운영 서버를 하루 만에 같은 체계로 맞췄고, 15분마다 최신을 받는다 (벽 7)

standarda 는 한 번에 설계된 것이 아니다. 실제 프로젝트에서 넘어진 자리마다 장치를 하나씩 세웠고, 그 기록이 이 자료의 근거다.

끝으로

전문가 없이도 할 수 있게

전문가의 머릿속에 있던 것을 시스템에 옮겨 담았기 때문에, 처음 하는 사람이 혼자 만들고, 올리고, 운영할 수 있다. 그리고 그 사람이 겪은 일은 다시 시스템에 담겨 다음 사람의 출발점이 된다.

> Brie 는 AX 를 대신 해 주지 않는다.
> 고객이 스스로 하게 만든다.