하네스 엔지니어링 with Codex

$ NOTICE

저작권 안내

본 자료는 저작권법에 의거해 허가 받지 않은 복사,
전재, 편집, 재배포 등을 금지함을 알려 드립니다.

본 자료는 온/오프라인 또는 동영상 강의를 전제로 개발되었습니다. 강사의 부가 설명 없이 본 자료의 내용을 임의 해석할 경우 잘못된 결론에 이를 수 있음을 유념하십시오.

강의를 캡쳐, 녹음, 녹화하는 등 콘텐츠의 원천 제공 방식 이외의 저장 행위를
엄격히 금지하니 필요하신 내용은 수업 틈틈이 개별적으로 메모하시기 바랍니다.

강사 소개



최준혁 · 개발동생

CREATE	유튜브 '개발동생' · 코딩 에이전트와 하네스, 최신 AI 개발 트렌드를 직접 써보고 검증하는 채널 (구독자 3.8만)
BUILD	6년차 프로덕트 엔지니어 · 스타트업 창립 멤버로 제품·인프라·팀 문화·AI 도입까지 구축, 현재 자체 프로덕트 개발
TEACH	AI 개발 생산성 강의 · 개발자와 팀의 AI 네이티브 전환

학습 내용 안내

Part 구성

Part 1. 에이전틱 엔지니어링의 이해

AI 엔지니어링 5단계 패러다임 · Codex 소개 · 첫 체험 실습

Part 2. Rule + Skill 기초 - 기초 하네스

AGENTS.md 중첩 구조 · skill-creator로 스킬 생성 · TDD Rule

Part 3. Skill 고급 + 기능 구현 - 워크플로우

Frontmatter 고급 필드 · PR 스킬(디렉토리 구조) · TDD 관찰

Part 4. Hook + Sub-agent - 안전 장치 완성

Lethal Trifecta · Protection/Reminder Hook · Evaluator

Part 5. 스케일 - Worktree + 브라우저 검증

병렬 개발 · Bun 팩트체크 · 브라우저 검증 · 외부 메모리

Part 6. Codex 운영과 비용 최적화

캐시 · 컨텍스트 관리 · 권한 모드 · DevContainer

Part 7. 하네스 리팩토링 - 다이어트

레거시 하네스 · 검증 강도 결정표 · 스킬로 하네스 다이어트

Part 9. 복기 + 마무리

순서 완성표 · 레이어 완성 지도 · 내일 당장 적용할 2가지 · Q&A

강의 자료 다운로드

강의자료 · 개발환경 세팅

1

강의자료 다운로드

devbrothers.ai/lectures/codex/harness-engineering/

2

개발환경 세팅 가이드

devbrothers.ai/lectures/codex/harness-engineering/dev-setup/

Part 1. 에이전틱 엔지니어링 이해

Part 1 구성

이론

- AI 엔지니어링 패러다임의 진화: 5단계 Deep Dive
- Codex 소개: Desktop App의 특징과 조작법

실습 1: Codex 첫 체험

- 하네스 없이 프로젝트 생성
- 환경 확인
- 한계 관찰: 하네스의 필요성 체감

AI 엔지니어링 패러다임의 진화

STAGE 01 · 2022 **프롬프트 엔지니어링**: AI에게 좋은 질문을 하는 기술

STAGE 02 · 2025.02 **바이브 코딩**: AI에 전적으로 의존하여 코드 생성

STAGE 03 · 2025.04 **컨텍스트 엔지니어링**: 모델이 무엇을 알고 있게 할 것인가

STAGE 04 · 2025.11 **하네스 엔지니어링**: 에이전트의 실수를 반복 방지하는 시스템

STAGE 05 · 2026.02 **에이전틱 엔지니어링**: 에이전트 루프 전체를 설계

패러다임의 포함 관계



프롬프트 ⊂ 컨텍스트 ⊂ 하네스 ⊂ 에이전틱

2024	프롬프트를 잘 쓰는 시대
2025	맥락을 잘 주는 시대
2026	환경을 잘 만드는 시대

KEY INSIGHT · 각 계층은 대체가 아닌 확장

상위 계층은 하위 계층을 포함하며 더 넓은 범위를 다룸

1. 프롬프트 엔지니어링 (2022~)

DEFINITION

"AI에게 좋은 질문을 하는 기술"

CONTEXT · 시대적 배경

GitHub Copilot GA(2022.06), ChatGPT(2022.11)보다 5개월 앞서 등장. LLM이 자연어 지시만으로 실용적 코드를 생성하며, **"AI에게 어떻게 말해야 원하는 결과를 얻을까"**가 핵심 화두가 됨

TECHNIQUE 01

역할 부여

Role Prompting

✕ "이 코드 리뷰해줘"
✓ "10년 경력 시니어 백엔드 개발자로서 보안 취약점을 분석해주세요"

TECHNIQUE 02

Chain-of-Thought

CoT · 단계적 추론

"단계별로 천천히 풀어줘" 한 문장만 추가해도 정확도 급상승

TECHNIQUE 03

ReAct

Reasoning + Action

추론 → 도구 호출 → 관찰 → 다시 추론

오늘날 AI 에이전트의 원형

Codex · Cursor

TECHNIQUE 04 · 메타 프롬프팅

DEFINITION

"프롬프트를 직접 쓰지 않고, **AI에게 쓰게 하는 기술**"

WHY IT WORKS

좋은 프롬프트의 요건(역할 부여 · 단계적 추론 · 구조화 · 예시)은 이미 모델 안에 있다 · 메타 프롬프팅은 그 지식을 꺼내 쓰는 기법

EVIDENCE

Prompt Optimizer

OpenAI 공식 도구 (Playground)

메타-프롬프트가 기존 프롬프트를 모범 사례
기준으로 재작성 · 구조 정리 · 예시 표준화

→ 무엇을 왜 바꿨는지 변경 근거를 함께 제시

IN CODEX

프롬프트부터 위임

복잡한 작업일수록 프롬프트 먼저

- ① 작업 전, 프롬프트 작성을 먼저 요청
- ② 부족한 맥락은 AI가 나에게 질문하게
- ③ 완성된 프롬프트는 새 채팅(`/new`)에서 실행

참고 · **developers.openai.com/api/docs/guides/prompt-optimizer** · **learn.chatgpt.com/guides/best-practices** · **developers.openai.com/cookbook** · Meta prompting

메타 프롬프팅 사용 패턴 3가지

META PROMPTING

프롬프트를 직접 쓰지 않고, **AI에게 받는다**

언제 쓰나 · 비용과 기준

개선된 프롬프트는 더 길고 느리다 · 한 문장으로 설명되는 작은 작업엔 그냥 실행이 낫다

PATTERN 01

대필

작업 전에 프롬프트부터 받기

"이 작업을 직접 실행하지 말고, 한 번에 성공시킬 정밀한 프롬프트로 다시 써줘. 무엇을 왜 바꿨는지 설명해줘"

입력은 날것 그대로 OK · 정제는 AI에게 · 성공 조건까지 쓰게 하기

PATTERN 02

인터뷰

AI가 질문해서 스펙 완성

"만들고 싶은 것: ___. 세부사항을 인터뷰해줘. 뻔한 질문 말고 내가 놓쳤을 부분을 파고들어. 끝나면 SPEC.md로 정리해줘"

Codex 공식 Best Practices, Reverse Interview 패턴

PATTERN 03

개선 루프

기존 프롬프트를 피드백으로 개선

기존 프롬프트 + 문제 피드백 + 이상적 출력 예시 → 개선 요청

피드백 예: "리뷰가 스타일 지적뿐이야, 버그·보안 위주로 바꿔줘"

→ 만든 프롬프트는 `/new` (New chat) 후 새 채팅에서 실행 (컨텍스트가 찰수록 LLM 성능은 저하된다)

참고 · **learn.chatgpt.com/guides/best-practices** · **developers.openai.com/api/docs/guides/prompt-optimizer**

프롬프트 엔지니어링의 한계

PROBLEM · 구조적 한계

컨텍스트 윈도우 문제

컨텍스트 윈도우 = LLM이 한 번에 기억할 수 있는 텍스트 최대 크기

대화가 누적될수록 윈도우를 가득 채움 → **성능 저하**

아무리 프롬프트를 정교하게 작성해도, 대화가 길어지면 **효과 감소**

INSIGHT · 핵심 통찰

프롬프트를 넘어 시스템으로

단일 프롬프트 최적화의 한계

시스템 수준 설계가 필요

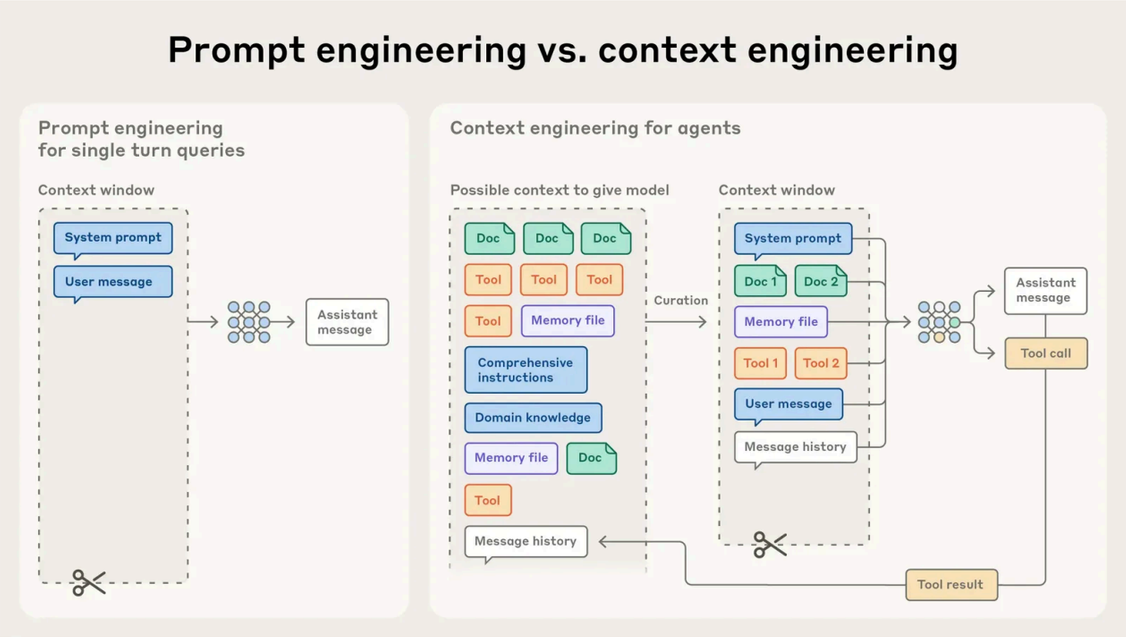
무엇을 기억하고 잊을지, 어떤 도구를 언제 호출할지 **전체 맥락 관리**

→ **컨텍스트 엔지니어링으로 진화**

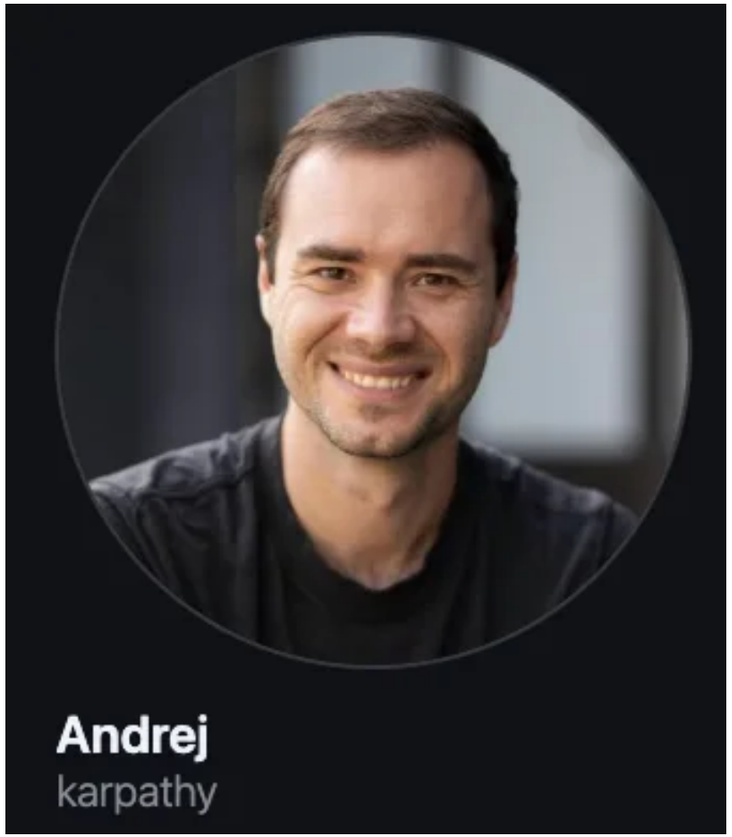
2. 컨텍스트 엔지니어링 (2025.04~)

"모델이 그 말을 들을 때 무엇을 알고 있게 할 것인가를 설계하는 것", Karpathy

컨텍스트 엔지니어링 다이어그램



참고 이미지



출처: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

2. 컨텍스트 엔지니어링 (2025.04~)

> 좋은 컨텍스트 엔지니어링은 **가장 작고 신호가 분명한 토큰 집합을 찾아내는 일**이다
>
> *Anthropic*

WHY · 컨텍스트 윈도우가 커졌지만 채울수록 성능이 저하된다 **(Context Rot)** → 채우는 게 아니라 **덜어내는** 설계 필요
(냉장고가 커졌다고 다 채우면 뒤쪽 음식이 썩듯이, 컨텍스트도 많이 채울수록 성능이 떨어집니다.)

METHOD · 01

적시 로딩

Just-in-Time

미리 다 넣지 않고, 필요할
때 동적으로 가져온다

EXAMPLE RAG · 파일 읽기 · 검색 툴 호출

METHOD · 02

외부 메모

Structured Note-taking

에이전트가 **실행 중에 스스로**
파일에 기록하며 상태를 외부화

EXAMPLE todo.md · scratchpad.md

METHOD · 03

멀티 에이전트

Multi-Agent

역할별로 컨텍스트를
분할해 각자 집중시킨다

EXAMPLE Planner · Worker · Reviewer

출처: **https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents**

Context Stuffing vs Curation

Context Stuffing = 문서를 통째로 넣기

- 추론 토큰 14~22% 증가, 성공률 하락
- `/init` 자동 생성 = stuffing

Context Curation = 필요한 정보만 설계하여 넣기

- 토큰 효율적, 에이전트 성능 향상
- 중첩 `AGENTS.md` = curation

연구 근거: /init은 왜 역효과인가

취리히 연방 공과 대학교 연구: "Evaluating AGENTS.md"

LLM 자동 생성 context 파일(/init)을 붙이면:

- 문제 해결률: -0.5% ~ -2% (오히려 하락)
- 추론 토큰: +14% ~ +22% 증가
- 추론 스텝: +2.45 ~ +3.92 스텝 증가

연구 결과 차트

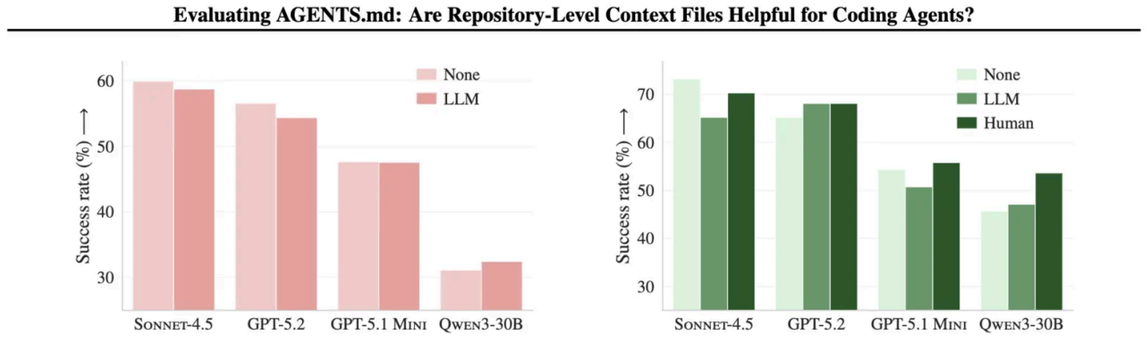


Figure 3. Resolution rate for 4 different models, without context files, with LLM-generated context files, and with developer-written context files, on SWE-BENCH LITE (left) and AGENTBENCH (right).

연구 근거: /init은 왜 역효과인가

ETH Zurich (취리히 연방 공과 대학교) 연구: "Evaluating AGENTS.md"

원인: "에이전트가 지시를 너무 잘 따른다"

- "context 파일에 "테스트 실행하세요" → 매번 전체 테스트 실행
- "관련 파일 확인하세요" → 관계 없는 파일까지 탐색
- "컨벤션 따르세요" → 컨벤션 확인만 반복

역설적이게도...
"성실한 에이전트 vs 현명한 에이전트"
→ 지시사항이 많으면 "성실"해지지만 "현명"해지지는 않는다

결론

직접 큐레이션한 AGENTS.md > LLM 자동 생성
→ Part 2에서 마스터 프롬프트로 "Curation"을 직접 체험

3. 바이브 코딩 (2025.02~)

"AI에 전적으로 의존해 코드를 이해하지 않고 수용하는 방법론"

2025년 2월 3일의 **트위터 포스트**에서
안드레 카파시가 처음 제시한 개념

"70% 문제"

- 프로토타입의 70%까지는 1시간 만에 도달
- 나머지 30%(에지케이스, 보안, 프로덕션 품질)에서 막힘
- 검증 없이 쌓인 코드 = 기술 부채 폭탄

교훈

- "주말 프로젝트엔 OK, 진지한 프로젝트엔 NO"
- AI에게 맡기되, 무엇을 알려줄지(컨텍스트)
  + 어떻게 제어할지(하네스)가 관건

밈: Vibe Coding vs Vibe Debugging



4. 하네스 엔지니어링 (2025.11~)

> **"에이전트가 실수를 저지르는 때마다, 다시는 같은 실수를 하지 않도록 해결책을 설계하는 것"**
>
> Mitchell Hashimoto, HashiCorp 공동 창업자 · Ghostty 개발자

하네스는 제한이 아니라 안전한 자유다

목줄은 한 점에 힘이 걸려 개를 다치게 한다. 하네스는 힘을 **몸 전체로 분산**시킨다.
개를 묶어두는 장치가 아니라, **마음껏 뛰되 차도로는 못 나가게** 하는 장치다.

영어 `harness` 의 다른 뜻: **harness the power of ~** : 힘을
억누르는 게 아니라 **쓸 수 있게 만든다** (수력을 harness해 발전하듯)

harness.jpg, 목이 아니라 가슴·어깨로



과거 직접 코드를 작성하는 것이 주된 업무

현재 에이전트가 안전하고 효율적으로 동작하는 **환경을 구축**하는 것이 주된 업무

그래서 어떻게 만드나: 하네스는 설계하는 게 아니라 누적된다

같은 **실수를 두 번** 하면 → 그때 **Rule**로 승격 · 같은 **절차를 세 번** 시키면 → 그때 **Skill**로 패키징 . 처음부터 완벽한 하네스를 설계하려 하지 않는다

출처: **mitchellh.com/writing/my-ai-adoption-journey**

$ 6가지 구성요소: 직접 구축

하네스 구성요소

Rule
에이전트가 따라야 할 행동 규칙
Part 2에서 구축 · AGENTS.md
한계 권고 수준: AI가 무시할 수 있음

Skill
반복 작업을 재사용 가능한 패키지로 묶기
Part 2~3에서 구축 · Agent Skills 오픈 표준

Hook
규칙을 프로그래밍 로직으로 강제
Part 4에서 구축
차별점 Rule은 권고, Hook은 강제

Sub-agent
컨텍스트 격리 + 작업 위임
Part 4에서 구축
핵심 작업자 ≠ 검증자 (Evaluator 패턴)

외부 도구 연동 (MCP)
혼자서는 못 보는 것, 화면·DB·
외부 시스템을 도구로 연결
Part 5(실습 7) · 내장 Browser(MCP와 같은 결)
원칙 코드로 못 보는 것만 외부 도구로 본다

외부 메모리
세션을 넘는 기억: 파일로 상태 외부화
Part 5에서 학습 · .agents/work/notes.md
· 마찰 로그(friction.md)
핵심 완료는 선언이 아니라 검증

Anthropic 실험 결과

동일 모델 + 동일 프롬프트에서 하네스만 변경 → 성공률 42% → 78%

"작업하는 에이전트와 판단하는 에이전트를 분리하는 것이 강력한 레버"

$ Anthropic 사례: 레트로 게임 메이커

하네스 구성요소

같은 Claude 모델로 레트로 게임을 개발시켰을 때,
하네스 구성만 바꿔서 결과가 극적으로 달라진 사례

A. 하네스 없이

Solo 단일 에이전트
시간 **20분**
비용 **$9**
결과 ⊗ **주요 기능 미작동**

B. 하네스 적용

Full Harness (3-에이전트)
시간 **6시간**
비용 **$200**
결과 ✓ **완전 기능 구현**

같은 모델 · 같은 프롬프트, 하네스 구성만 바꿔서 성패가 갈림

Anthropic: Harness Design for Long-Running Apps · **원문 링크** | 저자: Prithvi Rajasekaran (Anthropic Labs) | 2026년 3월 24일

$ Anthropic 에서 찾은 4가지 핵심 인사이트

하네스 구성요소

① 자기평가 편향

"에이전트에게 자기가 만든 결과를 평가하라고 하면, 자신 있게 자기 작업을 칭찬하는 경향이 있다."

② 작업자 ≠ 검증자 분리

"작업하는 에이전트와 그것을 판단하는 에이전트를 분리하는 것이 강력한 레버임이 드러났다."

③ 하네스 단순화 원칙

"가능한 가장 단순한 솔루션을 찾고, 필요할 때만 복잡도를 높여라."

④ 모델 진화에 따른 재검토

"모델이 좋아지면 하네스를 다시 점검하고, 더 이상 필요 없는 부품은 걷어내는 것이 좋은 습관이다."

Anthropic: Harness Design for Long-Running Apps · **원문 링크** | 저자: Prithvi Rajasekaran (Anthropic Labs) | 2026년 3월 24일

$ 이 강의의 진행 순서, 예고

무엇을 언제 붙이나: 문제와 반복이 기준이다

부품을 6개 봤다. 문제가 생기거나 같은 일이 반복되면, 그에 맞는 부품을 붙인다.

같은 설명을 반복한다

→ **Rule**

빈도 최고 · 비용 최저, Part 2

같은 절차를 반복시킨다

→ **Skill**

손이 반복될 때 패키징, Part 2~3

어기면 치명적이다

→ **강제**

권고를 겪은 뒤 강제를 안다, Part 4

5. 에이전틱 엔지니어링 (2026.02~)

> **"에이전트를 올바르게 작동시키는 시스템을 설계하는 것", Andrej Karpathy**
>
> 에이전트가 스스로 반복하는 **자율 루프**를 설계·운영

STEP 1 탐색
코드베이스 검색 · 요구사항 수집

→

STEP 2 계획
작업 분해 · 전략 수립

→

STEP 3 실행
코드 작성 · 도구 호출

→

STEP 4 검증
테스트 · 빌드 · 자기 교정

↻ **실패 시 다시 탐색으로, 사람 개입 없이 자율 반복**

인사이트

> **"엔지니어링 엄밀함은 사라지지 않는다. 더 높은 추상화 수준으로 이동할 뿐이다."**

설계 대상의 진화: 좁은 단위에서 시스템 전체로

~2023 · 프롬프트 단일 쿼리

↓

2024 · 컨텍스트 장기 맥락 설계

↓

2025 · 하네스 도구·규칙·제약

↓

2026~ · 에이전트 루프 자율 시스템 전체

각 단계에서 요구되는 엔지니어링 엄밀함의 본질은 같다, 대상만 커질 뿐

Codex 소개

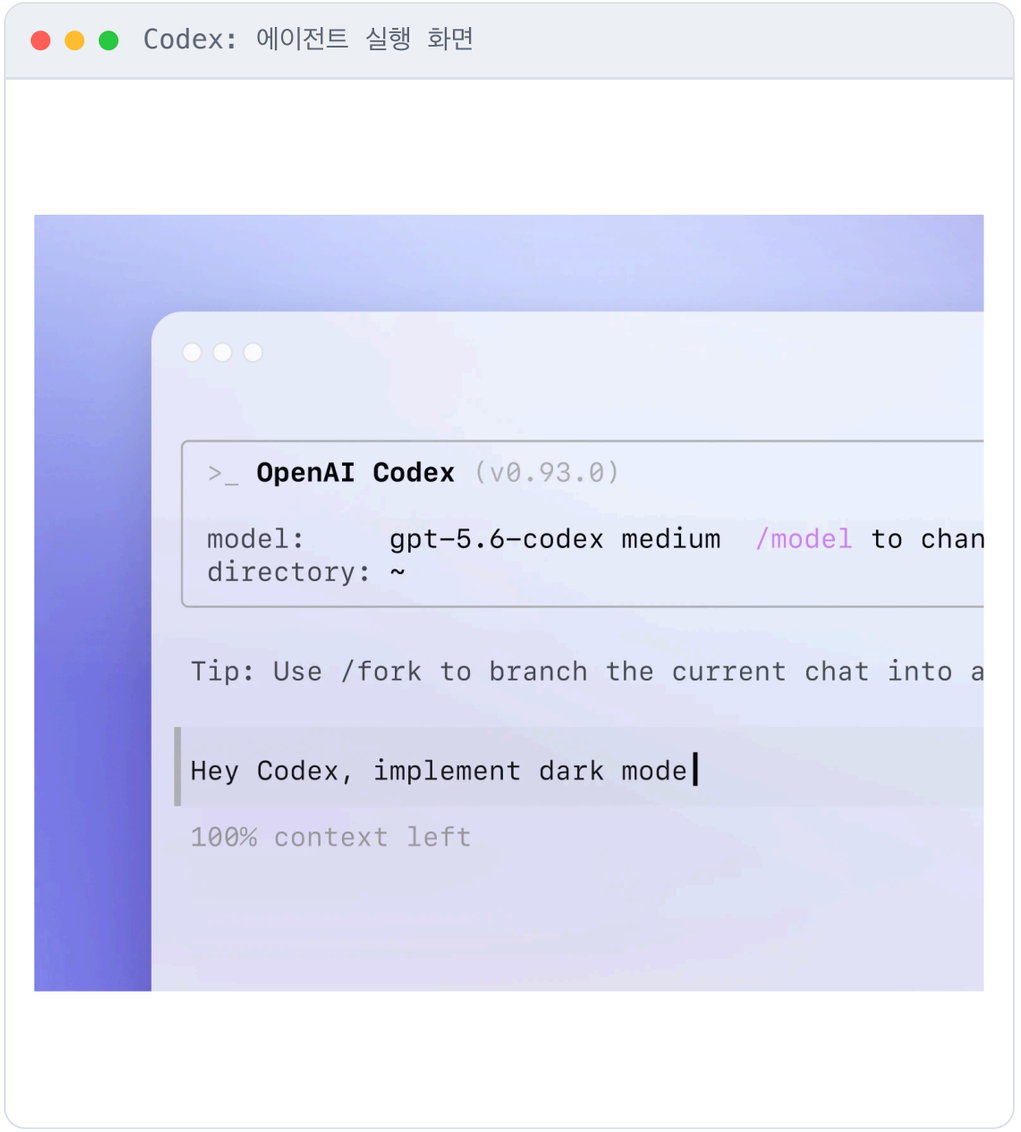
왜 에디터에서 에이전트 앱으로?

에디터: 사람이 코드를 읽고 편집하는 도구
에이전트 앱: AI가 코드를 읽고 편집하는 작업 공간

▸ 사용자가 사람 → AI로 바뀌면,
화면은 에이전트 앱으로 이동합니다

Codex Desktop App 특징

- 폴더를 열면 파일시스템·Git·셸에 바로 접근
- 특정 IDE 없이, 폴더만 열고 바로 작업
- 병렬 채팅·워크트리로 여러 작업을 동시에
- 같은 에이전트 코어가 CLI·CI에서도 동작



Codex 모델 + 주요 조작법

Codex 모델 클래스 · 4종 · 최신 GPT-6 세대와 롤아웃 동안 남는 GPT-5.6 세대

클래스	성격	최신 · 추천 (GPT-6 세대)	이전 (GPT-5.6 세대)
Astra	최고 성능. 가장 어려운 장기 작업·컴퓨터 사용	`gpt-6-astra` $10/$50	-
Sol	복잡한 코딩·에이전트 작업의 주력	`gpt-6.1-sol` $2/$10 **실습 메인** `gpt-6-sol` $2/$10 (이전 Sol)	`gpt-5.6-sol` $4/$20
Terra	지능과 비용의 균형	GPT-6 세대 없음	`gpt-5.6-terra` $2/$12
Luna	가장 효율적. 요약·추출·반복 작업	`gpt-6-luna` $0.10/$0.50	`gpt-5.6-luna` $0.20/$1.20

가격은 API 표준 요금, 100만 토큰당 입력/출력 · 6.1 Sol이 안 보이면 GPT-6 Sol, Free·Go는 Luna · GPT-5.6 세대는 롤아웃 동안만 남는다

그 외: 퇴역 · 특수 목적

- **GPT-5.5**: 2026-10-14 ChatGPT·Codex에서 퇴역(API는 유지) | 퇴역: **GPT-5.3 Codex Spark**(9/14), **GPT-5.4 / 5.4 mini**(8/31)
- **Daybreak Blue / Red**: 승인된 보안 작업용(Red는 별도 승인, 보안 연구 특화). **Trusted Access 승인 필요**라 `/model` 로 바로 전환되지 않는다

주요 조작법 · Desktop App

- `/model` 모델 선택 | `/reasoning` 추론 강도 (Light~Ultra) | `/status` 채팅 ID·컨텍스트·한도
- 권한은 입력창 아래 메뉴 (승인 요청 · 나 대신 승인 · 전체 권한 · 사용자 지정) | `/plan` 계획 모드
- `/compact` 컨텍스트 압축 | `/new` 새 채팅 (New chat · `Ctrl+N` / `Cmd+N`)
- `/init` AGENTS.md | 워크트리: 입력창 위 **워크트리** 스위치 | 이미지는 붙여넣기·드래그앤드롭

Codex 권한 모드

입력창 아래 메뉴 · 앱 제목은 **ChatGPT 액션은 어떻게 승인할까요?**

처음이면 **설정 → 일반**에서 나 대신 승인·전체 권한을 켜야 메뉴에 뜹니다

승인 요청

Ask for approval

외부 파일 수정 및 인터넷
사용 시 항상 확인

워크스페이스 안은 자동.
경계 밖은 사람이 승인

나 대신 승인

Approve for me

잠재적으로 위험한 것으로
감지된 작업에만 승인 요청

샌드박스는 승인 요청과 같음.
누가 승인하느냐만 바뀜

전체 권한

Full access

인터넷과 컴퓨터의 모든
파일에 제한 없이 액세스

샌드박스 해제. 강의 실습 기본값

사용자 지정

Custom (config.toml)

`config.toml`에
정의된 권한을 사용합니다

Part 4에서 권한 프로필로 다룸

⚠ 이 강의는 실습 속도를 위해 입력창 아래 권한을 `전체 권한` (Full access)으로 둡니다. 강의장 한정이며, 실무에서는 승인 요청 또는 나 대신 승인에서 시작하세요.

출처: OpenAI, **Permission modes** · 한국어 메뉴는 앱 실기

$ 실습

실습 1. Codex 첫 체험

실습 1 개요

하네스 0의 기준선을 직접 만들어보세요

미션: Codex를 처음 실행하고, 하네스 없이 간단한 프로젝트를 만들어보세요

이 실습에서 체험할 것

Codex Desktop App에서 기본 조작법 익히기

한 문장 프롬프트로 실행 가능한 프로그램 생성

"하네스 없는 상태"의 한계를 직접 관찰

→ Part 2부터 해결할 문제를 명확히 인식

Codex 첫 실행하기

Step 1 바탕화면에 `lumo-landing` 폴더를 만들고 앱에서 프로젝트로 열기 (Add project 또는 `Ctrl+O` / `Cmd+O`)

Step 2 입력창 아래 권한을 **전체 권한 (Full access)**으로 두기 (강의 실습 속도용, 앞 슬라이드 권한 모드 참고)

Step 3 `/new` 또는 New chat(`Ctrl+N` / `Cmd+N`)을 열고 `/status` 입력해보기

Step 4 `/model` 명령어 입력해보기

Step 5 정적 랜딩페이지 만들어보기: 아래 프롬프트 그대로 입력 (**프롬프트 gist**)

Step 6 추가 요청으로 수정 시도해보기

- ex) "핵심 기능 카드를 4개로 늘려줘"
- ex) "다크 모드 토글 버튼을 추가해줘"

한계 관찰

하네스가 없는 경우의 한계점

- 코드 스타일이 내 팀 컨벤션과 다르다
- 에러 핸들링이 없다
- 테스트가 없다
- 보안을 전혀 신경 쓰지 않는다

➔ "이걸 어떻게 체계적으로 해결할까?" = 하네스 엔지니어링

▸ Part 2부터 이 한계를 하나씩 해결해나간다:

Rule → **Skill** → **Hook** → **Sub-agent**

방금 만든 것이 "하네스 0"의 기준선이다.

Part 1 체감 포인트

"한 문장으로 실행 가능한 프로그램이 만들어진다"

- → Codex의 Power

"하지만 팀 규칙도 모르고, 테스트도 없고, 보안도 신경 안 쓴다"

- → 하네스 없는 상태의 한계

Part 2 예고

- Rule: AGENTS.md로 에이전트 행동 규칙 정의
- Skill 기초: 반복 작업을 재사용 가능한 스킬로 패키지화
  - → 기초 하네스 구축 시작

Part 2. Rule + Skill - 기초 하네스

Part 2 구성

순서 예고 회수 같은 **설명**을 반복한다 → **Rule** · 같은 **절차**를 반복시킨다 → **Skill**. 빈도가 가장 높고 도입 비용이 가장 낮아서 여기가 첫 칸이다.

이론

- Rule: 에이전트가 항시 참고하는 행동 규칙
- Skill: **Agent Skills** 오픈 표준 기반의 재사용 패키지

실습 2: AGENTS.md + 간단한 Rules + Skill

- 프로젝트 포크 + Codex 탐색
- skill-creator로 디자인 스킬 + 커밋 스킬 생성
- AGENTS.md 마스터 프롬프트로 중첩 구조 생성
- TDD Rule 추가 + 기능 구현으로 효과 관찰

AGENTS.md란 무엇인가

- **Rule이란?**
  - AI 에이전트가 매 세션마다 자동으로 읽는 프로젝트 규칙 파일
  - 코딩 컨벤션, 아키텍처 원칙, 금지 행동 등을 정의
  - 팀원과 함께 Git으로 버전 관리하자
- **AGENTS.md 표준**
  - **AGENTS.md**: 에이전트 도구 표준 (Codex, Cursor, Gemini CLI, Jules 등 공통) . Claude Code는 이 표준을 네이티브로 읽지 않는다(CLAUDE.md 사용)
  - 거버넌스: **Linux Foundation 산하 Agentic AI Foundation**이 스튜어드십을 맡는다 . 특정 벤더 단독 표준이 아니다
  - Codex는 **AGENTS.md를 별도 변환 없이 그대로 읽는다**: 프로젝트 루트에 두면 곧바로 적용된다

```
# AGENTS.md
프로젝트 루트에 두면 Codex가 자동으로 읽는다
```

AGENTS.md 계층 + 스코프

하네스 구성 요소 스코프

프로젝트 스코프

· 이 실습 기준

`./AGENTS.md` · `.agents/skills/`

개인 스코프 (글로벌)

`~/.codex/AGENTS.md` · `~/.agents/skills/`

AGENTS.md 계층, 중첩 구조

`~/.codex/AGENTS.md` 전역: 모든 프로젝트 공통 규칙

↓

`루트 AGENTS.md` 전체 프로젝트 규칙

↓

`src/AGENTS.md` 프론트엔드 전용 규칙 (하위 디렉토리)

↓

`AGENTS.override.md` 해당 스코프의 AGENTS.md를 완전히 대체

AGENTS.md 핵심 섹션

AGENTS.md에 포함할 핵심 섹션

Operational Commands

빌드/실행/테스트 명령어

Golden Rules

절대 타협 불가한
보안/아키텍처 제약

Do's & Don'ts

명확한 행동 수칙

Maintenance Policy

규칙과 코드의 괴리
발생 시 업데이트 제안

Rule의 한계: Rule은 "권고" 수준으로 AI가 판단에 따라 무시할 수 있다

→ Part 4에서 Hook으로 강제하는 방법을 배운다

Skill: Agent Skills 오픈 표준

- **Skill이란?**
  - 반복되는 프롬프트 패턴을 재사용 가능한 패키지로 저장
  - 직접 호출(/스킬명) 또는 AI가 상황에 맞게 자동 호출
  - 필요할 때만 점진적 로드 (토큰 절약)
- **Agent Skills 오픈 표준 (agentskills.io)**
  - 35개+ 에이전트/도구가 채택
  - 한 번 만든 스킬이 어떤 에이전트에서든 동작

my-skill/ 폴더 구조

```
my-skill/
├── SKILL.md          # Required: instructions
├── scripts/        # Optional: executable code
├── references/       # Optional: documentation
└── assets/    # Optional: templates, resources
```

Skill: Agent Skills 오픈 표준

스킬은 점진적으로 로딩되어서 컨텍스트를 효율적으로 사용할 수 있습니다.

- 발견: 이름과 설명만 훑어봄 (~100토큰)
- 활성화: SKILL.md를 읽음 (<5,000토큰)
- 실행: references/를 그때 읽음

pdf-processing/SKILL.md

```
---
name: pdf-processing
description: Extract PDF text, fill forms, merge files. Use when handling PDFs.
---

# PDF Processing

## When to use this skill
Use this skill when the user needs to work with PDF files...

## How to extract text
1. Use pdfplumber for text extraction...

## How to fill forms
...
```

SKILL.md 기본 구조 + skill-creator

SKILL.md 기본 구조

SKILL.md 예시

```
---
name: frontend-design
description: "UI 디자인을 개선합니다. 디자인 개선, UI 수정 요청 시 활성화."
---
(본문: 스킬이 수행할 절차)
```

실습 2. AGENTS.md + 간단한 Skill + Rules

실습 2 개요

기초 하네스 구축 후 커밋

미션: React Component Generator 프로젝트에
AGENTS.md 규칙 시스템과 기본 스킬을 구축하세요

이 실습에서 구축할 것

skill-creator로 UI 디자인 스킬 + 커밋 스킬 생성

AGENTS.md 마스터 프롬프트로 중첩 구조 규칙(Rule) 시스템

TDD Rule 추가 후 기능 구현으로 효과 관찰

→ Rule이 진짜 효과가 있는지 직접 확인

프로젝트 셋업

Step 1 프로젝트 다운로드 + 압축 해제 + 의존성 설치

- **github.com/devbrother2024/react-component-generator** 접속 → **Code** 버튼 → **Download ZIP**
- 바탕화면에 압축 풀기
- Codex Desktop App에서 해당 폴더를 프로젝트로 열기

Step 2 Gemini API Key 발급 + 설정

- **Google AI Studio** → API Key 생성 (무료)
- `cp .env.example .env` → `GOOGLE_API_KEY` 입력 (앱 통합 터미널 또는 Codex에게 요청)
- 의존성 설치하기: `bun install`
- 개발서버 실행하기: `bun run dev`

Step 3 Codex로 프로젝트 탐색

- "이 프로젝트의 전체 구조를 설명해줘"
- "이 프로젝트에서 개선할 점이 있다면?"
- **동봉된 스킬 구경하기**: `.agents/skills/` 에 뭐가 들어있는지 열어보기

Skill 기초: skill-creator

Step 4 skills.sh CLI로 frontend-design 스킬 설치하기

- **macOS · Linux**: `npx skills add anthropics/skills -s frontend-design -a codex`
- **Windows**: `cmd /c npx skills add anthropics/skills -s frontend-design -a codex`
- PowerShell에서도 `npx` 실행이 막히면 동일하게 `cmd /c`를 앞에 붙입니다.
- **https://github.com/anthropics/skills/blob/main/skills/frontend-design/SKILL.md**

Step 5 스킬 테스트: 디자인 개선 체험

- " 레트로 느낌으로 페이지 레이아웃과 디자인, 테마 등 전체적으로 개선해보고 싶어. 알아서 딱 깔끔하고 센스있게 개선해봐"
- => 브라우저에서 변경 결과를 직접 확인해보기
- **현재 트렌드 스킬 찾기**
- **https://www.skills.sh/**

Skill 기초: skill-creator

Step 6-1 `user.name` & `user.email` 세팅

- `git init`
- `git config user.name YOUR_NAME`
- `git config user.email YOUR_EMAIL`

Step 6-2 `$skill-creator`로 커밋 스킬 생성 (Codex 기본 내장, 설치 없이 바로 호출)

- **https://gist.github.com/devbrother2024/8d97fb6bcbc39d7c1dd03e938c8854f0/**

Step 7 스킬 테스트: 커밋

- "모든 변경사항 커밋해줘" → commit 스킬 자동 호출

skill-creator: 스킬을 만드는 스킬

- Codex 기본 내장 (OpenAI 공식), 별도 설치 없이 바로 사용
- "이런 스킬 만들어줘" → 적절한 SKILL.md를 자동 생성
- 호출: `$skill-creator`

Rule: 중첩 구조 AGENTS.md

Step 8-1 `/status`로 컨텍스트 윈도우 사용률 보기 & `/new`(New chat)로 컨텍스트 초기화

Step 8-2 /init → AGENTS.md 스캐폴드 생성 → 내용 확인

- → README와 중복, 불필요한 정보 포함

Step 9 **AGENTS.md 마스터 프롬프트** (.agents/skills/agents-md, 프로젝트에 이미 포함됨) 스킬 뜯어보고 학습하기

- 자동 생성 AGENTS.md 삭제 → $agents-md 재실행 (2단계 자동 실행)
- Step 1: 루트 AGENTS.md, Golden Rules에
  코드 근거 기반 팀 고유 규칙까지 자동 포함
- Step 2: 중첩 파일(src/AGENTS.md 등),
  Local Golden Rules도 동일하게 반영

Step 10~11 생성된 파일 확인 + 직접 다듬기

중첩 AGENTS.md 구조

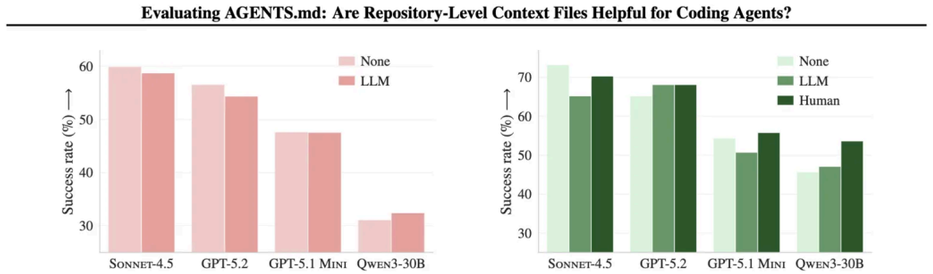


Figure 3. Resolution rate for 4 different models, without context files, with LLM-generated context files, and with developer-written context files, on SWE-BENCH LITE (left) and AGENTBENCH (right).

$ 지식 추출 절차

Golden Rules, 뭘 근거로 쓸까, 5착안점

방금 실행한 마스터 프롬프트는 **구조**만 잡지 않는다. 다음 5가지 신호로 코드를 직접 읽어 **이 팀만 아는 것**까지 Golden Rules에 채운다.

① 비대칭

같은 층위인데 **한쪽에만** 있는 처리
과거 장애를 겪고 수정한 흔적일 가능성이 높다

② 테스트 경계

테스트가 **있는 파일** vs **없는 파일**
팀이 위험하다고 판단한 영역을 보여준다

③ 하드 제약

어기면 **즉시 깨지는** 프레임워크·런타임 제약
AI가 자주 위반하는 지점이다

④ 이중 방어

같은 위험을 **두 곳에서** 막고 있다면
한쪽을 지우면 문제가 재발할 수 있다

⑤ 보안 경계

무엇이 **절대 클라이언트로 안 가는가**
명시하지 않으면 AI가 경계를 넘을 수 있다

전이되는 건 정답이 아니라 절차

같은 질문을 자기 회사 레포에도 던질 수 있다

💡 **정답지를 나눠주지 않는다, 방금 생성된 AGENTS.md에 이 5가지 신호가 실제로 어떻게 반영됐는지 직접 확인한다.**

$ Codex 공식 AGENTS.md 가이드

AGENTS.md에 넣을 것 / 뺄 것

✓ 넣을 것

- 프로젝트 고유 **아키텍처 결정**
- 흔한 함정 · **비직관적 동작**
- 테스트 지침
- 환경 특이사항 · 특수 명령어

✗ 뺄 것

- **코드를 읽으면 알 수 있는 것**
- 표준 관행 · 상세 API 문서
- **자주 바뀌는 정보**
- 파일별 설명

판단 기준: "이걸 지우면 Codex가 실수하게 되나? 아니라면 잘라낸다." · **권장 크기: 파일당 200줄 미만** · 32KiB 넘으면 하위 디렉토리 AGENTS.md로 분할

출처: **learn.chatgpt.com/docs/agent-configuration/agents-md** · **Codex Best Practices**

AGENTS.md 계층 구조: 전역 → 프로젝트 → 하위 디렉토리

- **별도 rules 폴더가 아니라, 하나의 표준을 계층으로 로드**
- **로딩 순서**
  - `~/.codex/AGENTS.md` (전역) → 프로젝트 루트 `AGENTS.md` → 작업 중인 하위 디렉토리 `AGENTS.md` 순서로 누적 로드
  - 세션 시작 시 항상 로드 → 총 용량 상한 `project_doc_max_bytes` 기본 32KiB, 넘으면 잘려서 로드된다
- **AGENTS.override.md**
  - 해당 스코프에 두면 그 스코프의 AGENTS.md를 완전히 대체(override)한다
  - 역할: 특정 디렉토리에만 다른 규칙을 적용하고 싶을 때
- 💡 **비유**
  - 전역 AGENTS.md = 모든 프로젝트 공통
  - 프로젝트·하위 디렉토리 AGENTS.md = 영역별 모듈화

AGENTS.md 계층 로딩 구조

```
~/.codex/                                  # 전역
├── AGENTS.md                              # 공통 지침
└── AGENTS.override.md                     # 스코프 override

your-project/                              # 프로젝트 루트
├── AGENTS.md                              # 프로젝트 규칙
└── src/
    └── AGENTS.md                          # 하위 규칙
```

AGENTS.md는 세션 내내 항상 컨텍스트에 남는다, 항상 필요하지 않은 작업별 절차는 AGENTS.md에 쌓지 말고 **Skill**로 분리해 필요할 때만 로드한다

TDD Rule + 기능 구현

Step 12 AGENTS.md에 TDD 규칙 섹션 추가

https://gist.github.com/devbrother2024/d9e184989144cd07215fd8f78b876547/

Step 13 규칙 적용 확인: 기능 구현 → **"프롬프트 500자 제한 검증 함수 추가해줘"** (유틸 함수 지정: TDD 적용 대상)

- ☆ 관찰: AGENTS.md의 TDD 규칙이 있으니 AI가 테스트를 먼저 작성하는가?
- ☆ 관찰: AGENTS.md 컨벤션을 따르는가?

Step 14 TDD로 구현한 것 커밋으로 마무리하기

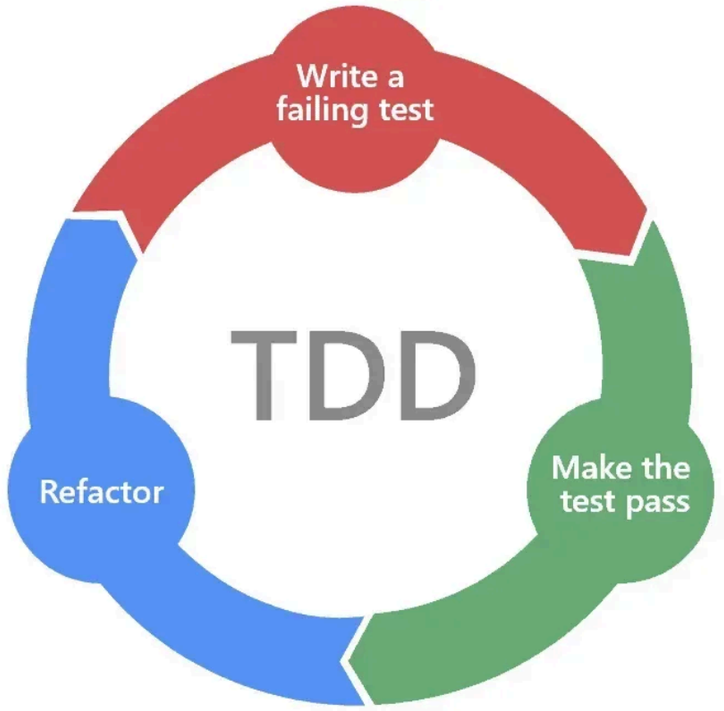
- "변경사항 커밋해줘" → commit 스킬 자동 호출

TDD: 왜 하네스에서 중요한가

TDD (Test Driven Development, 테스트 주도 개발)

- **AI 시대에 TDD가 더 중요한 이유**
  - AI가 코드를 작성하면 사람이 한 줄씩 검증하기 어렵다
  - TDD는 구현 전에 정답을 먼저
    정의 → AI 코드를 자동 검증
- **RED → GREEN → REFACTOR**
  - RED: 실패하는 테스트를 먼저 작성
  - GREEN: 테스트를 통과하는 최소한의 코드만 작성
  - REFACTOR: 코드 정리, 테스트는 계속 통과
- **하네스에서 TDD를 강제하는 2계층**
  - Rule (Part 2): 권고 | Hook (Part 4): 강제

RED-GREEN-REFACTOR 다이어그램



현재 하네스 상태

현재까지 구축한 하네스

- frontend-design plugin
  UI 디자인 개선
- `.agents/skills/commit.md`
  커밋 자동화
- AGENTS.md (루트)
  프로젝트 전체 규칙
- src/AGENTS.md
  프론트엔드 규칙
- server/AGENTS.md
  백엔드 규칙

AGENTS.md의 TDD 규칙 섹션까지 추가 완료, 단, 아직 "권고" 수준

Part 2 리뷰

skill-creator = "의도만 전달하면 스킬이 만들어진다"

→ 스킬 제작의 진입 장벽을 낮춤

"AGENTS.md" = 오픈 표준 · 전역~하위 디렉토리 계층으로 로드

어떤 AI 도구를 쓰든 동일한 규칙 적용을 위해 AGENTS.md 권장

중첩 구조 AGENTS.md = Context Curation (Context Stuffing X)

/init 자동 생성(Stuffing) 대비 토큰 효율적

Rule은 "권고"일 뿐 강제할 수 없음

AGENTS.md에 TDD 규칙을 정의해도 AI가 테스트를 건너뛸 수 있다

→ Part 4에서 Hook으로 강제

Part 3. Skill 고급 + 기능 구현 - 워크플로우

Part 3 구성

이론

- Skill 심화: Frontmatter
  고급 필드와 디렉토리 구조

실습 3: PR 생성 스킬 + 기능 구현

- 디렉토리 구조 스킬
  (SKILL.md + references/)
- references/는 필요 시에만 로드, 토큰 절약
- localStorage 영속화 기능을 TDD로 구현
- `$commit` → `$create-pr` →
  GitHub 자동 리뷰 워크플로우

Skill 심화: 기본 vs 고급

이번 Part: 고급 스킬

- 디렉토리 구조 (SKILL.md + references/)
- 지식을 파일로 분리: 필요할 때만 로드해 토큰 절약
- 같은 스킬이지만 규모가 커져도 정리된 상태 유지

skill 구조 예시

```
my-skill/
├── SKILL.md          # Required: instructions
├── scripts/        # Optional: executable code
├── references/       # Optional: documentation
└── assets/    # Optional: templates, resources
```

Frontmatter 고급 필드

스킬의 핵심 Frontmatter 옵션

- **description**: 자동 호출 조건 (AI가 호출 시기 판단)
- **allowed-tools**: 스킬별 도구 제한

Codex 공식 skill-creator 문서: SKILL.md엔 `name` · `description` 외 필드를 넣지 말라고 명시합니다. 호출 제어나 격리 실행이 필요할 때 Codex가 쓰는 실제 방식은 다음 장에서 다룹니다.

SKILL.md frontmatter 예시

```
---
name: my-skill
description: 스킬이 뭘 하는지?
allowed-tools: Read Grep
---

Your skill instructions here...
```

호출 제어: Codex의 실제 방식

Claude Code라면 SKILL.md 한 줄

- `disable-model-invocation: true` : frontmatter에 한 줄 추가

Codex는 별도 파일입니다

- 스킬 폴더 안 `agents/openai.yaml`에 `policy.allow_implicit_invocation: false`를 씁니다
- SKILL.md frontmatter가 아니라 **스킬 폴더 옆의 정책 파일**입니다
- 이렇게 하면 모델이 자동으로 스킬을 불러오지 않고, `$스킬이름` 명시 호출에만 실행됩니다

context: fork(격리 실행)는?

- Codex엔 아직 대응 기능이 없습니다, 격리 실행이 필요하면 Part 4의 Sub-agent를 직접 spawn하세요

my-skill/agents/openai.yaml

```yaml
policy:
  allow_implicit_invocation: false

# SKILL.md는 name·description만
# 유지: 나머지 필드는 넣지 않습니다
```

실습 3. PR 생성 스킬 + 기능 구현

실습 3 개요

◎ 미션: 디렉토리 구조의 PR 생성 스킬을 만들고, localStorage 영속화 기능을 TDD로 구현한 뒤 PR까지 생성하세요

이 실습에서 구축할 것

- .agents/skills/create-pr/ 디렉토리 구조 스킬
- references/로 PR 템플릿 분리, 필요할 때만 로드
- localStorage 영속화를 TDD로 구현
- `$commit` → `$create-pr` → GitHub 자동 코드 리뷰 워크플로우 완성

☆ 이번 실습의 진짜 관전 포인트, 하네스 4종이 동시에 작동한다

TDD Rule · commit 스킬 · create-pr 스킬 · GitHub 자동 코드 리뷰, 기능 하나 만드는 동안 넷이 각자 다른 시점에 끼어든다. 무엇이 언제 발동하는지 지켜보세요.

실습 3 사전 준비사항

실습 3 시작 전 아래 2가지를 먼저 완료해주세요

1. gh auth login

- GitHub CLI 인증
- `gh auth login`
- GitHub.com 선택
- HTTPS 프로토콜 선택
- 브라우저 인증 완료
- 확인:
  `gh auth status`

2. GitHub repo 만들기

- "GitHub에 'react-component-generator' 이름으로 repo 만들고 지금 코드 push해줘"
- 이후 PR·자동 리뷰가 이 repo를 기준으로 동작
- 확인:
  `gh repo view`

GitHub Connector + 자동 코드 리뷰 켜기

3 **Codex Settings → Connectors**에서 **GitHub 연결**

4 GitHub 승인 화면에서 계정과 대상 repo 접근 허용 . Select repositories라면 방금 만든 repo를 포함

5 **Code review 설정**에서 방금 만든 repo의 **Code review** 켜기

6 **Automatic reviews**도 켜기 . PR을 열면 코멘트 없이 자동 리뷰

출처: **learn.chatgpt.com/docs/third-party/github**

테스트 환경 + PR 생성 스킬 생성

Step 1~2를 직접 진행해보세요

Step 1 PR 생성 스킬 요청

- **Frontmatter**: `name` · `description` 만 (Codex SKILL.md 표준)
- **PR 본문 템플릿**: `references/pr-template.md`

Step 2 생성된 스킬 구조 확인

- Part 2 스킬과의 차이점
- **references/ 폴더의 역할**: 필요 시에만 로드 → **토큰 절약**

💡 **잘 모르겠으면 → 프롬프트 참고하세요**

기능 구현: TDD 관찰

Step 3~5를 진행해보세요 · TDD 흐름을 직접 관찰해보세요

Step 3 localStorage 영속화 기능 구현

- "브랜치 생성해서 기능 구현해줘", main 직접 커밋 방지, PR은 브랜치 단위로 생성됨
- API 키 저장 · Provider 선택 유지 · 프롬프트 히스토리 · 생성된 컴포넌트 목록 보존
- **프롬프트 gist**

Step 4 ☆ TDD 관찰 포인트

- 테스트를 먼저 작성했는가? **(RED)**
- 최소한의 구현만 했는가? **(GREEN)**
- 테스트 없이 바로 구현?

Step 5 테스트 실행 → 통과 확인

- 모든 테스트 통과 확인 · **GREEN** 상태 달성

커밋 + PR + 코드 리뷰

Step 6~8을 진행해보세요 · `$commit` → `$create-pr` → 코드 리뷰 자동화를 경험해보세요

Step 6 $commit 스킬 호출

- 커밋 요청해서 $commit 스킬 호출
- Part 2에서 만든 스킬 활용

Step 7 $create-pr 스킬로 PR 생성

- $create-pr 스킬 호출
- **PR 본문 자동 작성 및 PR 생성**

Step 8 GitHub PR 위에서 자동 코드 리뷰 확인

- ☆ **검증 2계층**: **TDD**(자동 테스트), **Codex 코드 리뷰**(GitHub PR 위에서 자동 실행)
- 사전 준비 단계에서 Automatic reviews를 미리 켜뒀기 때문에, PR을 열면 코멘트 없이 바로 리뷰가 달린다

$commit → $create-pr 흐름

```
❯ 모든 변경사항 커밋해서 main 브랜치 대상으로 PR 생성해줘.
```

Part 3 리뷰

Skills 디렉토리 구조

references/로 PR 템플릿 참조 (필요 시만 로드)

$commit → $create-pr 연쇄 호출

스킬 두 개를 이어 붙여 PR까지 자동 생성,
격리 실행은 Part 4 Sub-agent에서

검증이 2계층

TDD(테스트 코드) + GitHub
Codex 코드 리뷰(AI 코드 리뷰)

TDD Rule 관찰: "규칙을 넣었는데 AI가 정말 따랐나?"

→ Rule의 한계를 직접 목격 → Part 4에서 Hook으로 해결

Part 4. Hook + Sub-agent - 안전 장치 완성

Part 4 구성

"Codex 생명주기 동안 결정론적 스크립트를 실행한다" · "Run deterministic scripts during the Codex lifecycle."

이론

- 왜 Hook이 필요한가
- Hook: 프로그래밍 로직으로 강제하는 안전 장치
- Sub-agent: 컨텍스트 격리 위임

실습 4: Permission + Hook + Sub-agent

- permissions.deny: 민감파일(.env) 정적 차단
- Protection Hook: 커밋 전 테스트 강제
- Reminder Hook: 테스트 파일 누락 경고
- 커스텀 evaluator 서브에이전트
  → 개발 하네스 완성

출처: OpenAI, **Codex Hooks 문서**

Hook: 프로그래밍 강제

Hook이란?

Codex 라이프사이클의 특정 지점에서
자동으로 실행되는 사용자 정의 명령어

결정론적 실행
LLM 판단 없이 100% 실행 보장

이벤트 기반
라이프사이클 특정 시점에만 발동

유연한 핸들러
command · http · prompt · agent

왜 Hook이 필요한가?

프롬프트로 지시 vs 코드로 강제

AGENTS.md

· 프롬프트 방식

자연어로 규칙 지시
LLM이 읽고 판단
가끔 무시될 수 있음

⚠ '아마도' 실행

Hook

· 코드 방식

이벤트에 직접 연결
Shell · HTTP 실행
조건 만족 시 무조건 실행

✓ '반드시' 실행

핵심 인사이트

규제 환경과 프로덕션 코드베이스에서 '아마도 실행'과 '반드시 실행'의 차이는 결정적입니다

주요 훅 이벤트 한눈에 보기

① 세션 & 프롬프트

SessionStart
세션 시작 시
환경 로드 · 컨텍스트

UserPromptSubmit
차단 가능
프롬프트 제출 시
프롬프트 검증·보강

SessionEnd
세션 종료 시
세션 정리 · 로그 기록

② 도구 실행 라이프사이클

PreToolUse
차단 가능
도구 실행 직전
위험한 명령 차단

PostToolUse
도구 실행 직후
자동 포매팅·린트

PermissionRequest
차단 가능
권한 요청 시
안전 명령 자동 승인

③ 종료 · 서브에이전트 · 컴팩션

Stop
응답 완료 시
테스트 통과 강제

SubagentStop
서브에이전트 종료
서브 결과 검증

PreCompact
컨텍스트 압축 전
중요 정보 백업

출처: learn.chatgpt.com/docs/hooks

Hook 3가지 패턴

Protection

보안·안전 차단

위험한 작업을 아예 막는다

PreToolUse

적용 예시

- **.env 파일**: 편집 차단
- **main 브랜치**: 푸시 금지
- **rm -rf /**: 실행 거부

exit 2 · 차단

Reminder

규칙 망각 상기

차단하진 않지만 경고를 남긴다

Stop · PostToolUse

적용 예시

- **TDD 규칙**: 테스트부터?
- **커밋 메시지**: 컨벤션 준수?
- **타입 선언**: any 사용 확인

exit 0 + stderr

Workflow Trigger

조건부 자동 실행

특정 시점에 자동으로 작업을 돌린다

PostToolUse · Stop

적용 예시

- **파일 편집 후**: Prettier 실행
- **커밋 완료**: Slack 알림
- **세션 시작**: Git 상태 주입

exit 0 · 통과

세 패턴을 상황에 맞게 조합한다

차단(Protection) · 상기(Reminder) · 자동화(Workflow Trigger), 한 프로젝트에 여러 개를 함께 쓴다

Sub-agent: 컨텍스트 격리

① 왜 Sub-agent가 필요한가?

컨텍스트 유한
대화가 길수록 품질 저하

메인 에이전트 보호
서브는 자체 컨텍스트

요약만 반환
핵심만 돌려받음

② 내장 서브에이전트 유형

explorer
코드베이스 탐색
(Luna, 읽기 전용)

default
분석 · 계획
(부모 설정 상속)

worker
코드 작성
(전체 도구)

③ Evaluator 패턴: "작업자 ≠ 검증자"

작업자 Agent
코드 작성 · 구현

→

검증자 Agent
독립 검토 · 피드백

→

검증된 결과
편향 제거 · 품질 ↑

자기 코드를 스스로 리뷰하는 것은 본질적으로 편향됩니다. 검증자를 분리하면 동일 모델이라도 훨씬 엄격한 리뷰가 가능합니다

Cross-Model Evaluator

한 모델의 블라인드 스팟을, 다른 모델이 본다

동일 모델 검증은 같은 블라인드 스팟을 반복할 수 있다 → 구현과 검증을 다른 모델 계열로 분리한다

① 실전 적용: Codex Sol(계획·검증) + Claude Code / Fable 5.1(구현)

Step 1
Codex Sol
계획·검증 기준 수립

→

Step 2
Claude Code
Fable 5.1로 구현

→

Step 3
Codex Sol
diff·테스트 결과 검증

Claude Code가 구현한 diff와 테스트 결과만 Codex Evaluator에 전달 → 구현 맥락과 분리된 독립 검증

② 왜 다른 모델이어야 하나?

× 같은 모델끼리 검증

같은 가정·추론 습관을 반복할 가능성

"내가 쓴 코드를 내가 리뷰하는 것과 유사"

다른 모델로 적대적 리뷰

모델 계열·도구 맥락이 달라 다른 실패를 포착

"Adversarial Review로 블라인드 스팟 제거"

독립 컨텍스트 + 다른 모델 계열 = 편향 감소
계획의 검증 기준은 고정하고, 구현 모델과 검증 모델의 시각을 분리한다

실습 4. Hook + Sub-agent

실습 4 개요

정비 B: 기능은 만들지 않는다. 권한·Hook·Sub-agent로 안전장치를 추가한다.

미션: 권한과 Hook으로 Rule의 한계를 해결하고, 커스텀 Sub-agent를 만들어 검증을 분리하세요

이 실습에서 구축할 것

- Self-Evaluation: 에이전트가 자기 코드를 스스로 평가하면...
- permissions.deny: 민감파일(.env) 정적 차단
- Protection Hook: 커밋 전 테스트 강제
- Reminder Hook: 테스트 파일 누락 경고
- 커스텀 evaluator 서브에이전트
- **→ Rule의 한계를 강제로 해결하는 것을 직접 체감**

실습 4 개요

실습 4 · 개요

자기 평가 편향을 제거하는 Generator-Evaluator 패턴을 구축한다

Codex가 "다 됐습니다"라고 말할 때, 정말 다 된 건지 어떻게 확인하지?

문제: Self-Evaluation 편향

에이전트가 자기 코드를 스스로 평가하면...

- 과대평가: "완료했습니다" 선언
- 맹점: 엣지 케이스 미처리
- 부족: 테스트 커버리지 누락

"내가 쓴 글을 내가 교정 보는 것과 똑같다"

해법: Generator-Evaluator 패턴

작업자와 검증자를 다른 컨텍스트로 분리

- **Generator**: 작업자 (메인)
- **Evaluator**: 검증자 (서브)
- **객관적 피드백 반환**

"편집자를 따로 두는 것과 같은 효과"

"Generator-Evaluator"라는 이름은 **GAN**(2014 Ian Goodfellow · 딥페이크·StyleGAN의 기반)에서 왔다, "혼자서는 발전하기 어렵지만, 비판자가 있으면 급격히 좋아진다"

$ 보완 패턴

Advisor Strategy: 비용까지 최적화하는 상위 패턴

6.1 Sol이 실행 → 막힐 때만 Astra 자문 (한 번의 호출, 공유 컨텍스트)

비용까지 최적화하는 상위 패턴, 작업자는 빠르고 저렴한 모델, 자문은 막혔을 때만 상위 모델을 부른다

실측 근거(Anthropic, Claude 모델 기준, Codex 자체 실측치는 아직 없음): SWE-bench **+2.7%p** · 비용 **-11.9%** · BrowseComp에서 Haiku+Opus 조합 **19.7%→41.2%**(2배 이상), 비용은 Sonnet 대비 **85% 절감**

Generator-Evaluator가 **"누가 판정하나"**였다면, Advisor는 **"막혔을 때 누구에게 묻나"**다.

출처: **Anthropic, "The advisor strategy" (2026.04.09)** · SWE-bench Multilingual · 300문제 × 9개 언어 × 5회

실습 4 개요

실습 4 · 개요 ②

컨텍스트가 꽉 차면 에이전트도 조급해진다, Sub-agent로 격리하자

긴 대화 후반부의 Codex는 처음 Codex와 같은 에이전트가 아니다

대화 길이에 따른 Codex의 상태 변화

초반: 신선한 상태

정교한 추론, 엣지케이스 고려

중반: 품질 저하 시작

Context Drift: 맥락 이탈

후반: 신뢰성 불안정

지름길 · 주석 스킵 · TODO

/new

세션 리셋

문맥이 통째로 날아감

스프린트 분해

작업을 작게 쪼개기

컨텍스트 수명 연장만

Sub-agent 격리

메인은 계속 신선하게 유지

◎ Sub-agent가 두 가지 문제를 한 번에 해결한다, ①Self-Evaluation 편향 ②Context Anxiety → 다음 페이지부터 evaluator를 직접 만들어본다

권한 프로필로 민감파일 차단

목표: 민감파일(.env · credentials · secrets) 접근을 권한 레벨에서 차단한다

Step 1 **권한 프로필 선언 + deny 규칙 추가**: 정적 패턴 차단은 Hook보다 Permission이 우선 (permissions)

config.toml, **필수**: default_permissions(활성화 스위치) + filesystem 규칙. extends로 내장 :workspace
프로필의 기본 권한을 물려받고 deny 3줄만 얹는다 (특수 토큰·필수/선택 전체 정리는 다음 슬라이드)

```
default_permissions = "harness"
[permissions.harness]
extends = ":workspace"
[permissions.harness.filesystem.":workspace_roots"]
"**/.env*" = "deny"
"**/*credentials*" = "deny"
"**/*secrets*" = "deny"
```

Step 2 **차단 테스트**: 아래 두 프롬프트가 모두 `Operation not permitted`로 차단되는지 확인한다 (Denied)

테스트 ① Read 도구

```
> .env 파일의 내용을 보여줘
Error: .env: Operation not permitted
```

테스트 ② Bash 도구

```
> cat .env 를 실행해줘
cat: .env: Operation not permitted
```

정적 차단 = Permission · 동적 검증·경고 = Hook

출처: OpenAI, **Codex Permissions 문서**

$ 필수 · 선택

권한 프로필 문법 사전 (1/2)

앞 슬라이드 config.toml이 왜 이렇게 생겼는지, 한 줄씩

필수: 없으면 프로필이 아예 안 먹힌다

- `default_permissions` : 어떤 프로필을 켤지 고르는 최상위 스위치. 없으면 `[permissions.*]` 는 정의만 되고 무시된다.
- `[permissions.<name>.filesystem]` : 파일 접근 규칙 테이블. 비어있거나 아예 없으면 "전부 차단"이 기본값이다 (공식 문서: "Missing or empty filesystem tables keep filesystem access restricted")

선택: 안 써도 동작한다

- `description` : 사람이 읽는 설명. 동작엔 영향 없음
- `extends` : 부모 프로필 상속. 내장 3종 `:read-only` · `:workspace` (오늘 harness가 상속)· `:danger-full-access` 중 하나로 시작하면 나머지는 안 써도 된다
- `workspace_roots` : 작업 폴더를 추가로 더 등록할 때만
- `network` : 기본값 꺼짐(`enabled = false`). 안 건드리면 안 적어도 된다

$ 특수 토큰

권한 프로필 문법 사전 (2/2)

filesystem 안의 특수 토큰: 실제 경로가 아니라 경로 "묶음"을 가리키는 예약어

:minimal

셸·도구 최소 동작에
필요한 시스템 경로

:workspace_roots

지금 세션이 작업 중인 폴더(들)

:root

파일시스템 전체(/): 거의 안 씀

:tmpdir · :slash_tmp

임시 폴더: 기본값으로 충분

오늘 실습은 `extends = ":workspace"` 로 시작하기 때문에 이 특수 토큰들을 직접 쓸 필요가 없다, 자기만의 프로필을 처음부터 만들 때만 필요한 지식이다.

출처: OpenAI, **Codex Permissions 문서**

Protection Hook: 커밋 전 테스트 강제

Protection Hook

목표: git commit 전에 bun run test가 통과해야만 커밋 허용

Step 3 **커밋 전 테스트 강제 Hook 추가**: Bash(git commit) 호출 전에 bun run test를 실행 → 실패 시 커밋 차단 (PreToolUse)

.codex/hooks.json, 프로젝트 로컬(전역 ~/.codex 아님). 훅 기능은 기본 활성이지만, 개별 훅은 trust 전까지 실행 안 됨 ↓

```
{
  "matcher": "^Bash$",
  "hooks": [{ "type": "command",
    "command": "case \"$(jq -r .tool_input.command)\" in *'git commit'*) bun run test || exit 2;; esac"
  }]
}
```

⚠ 추가만으로는 안 켜진다, `/hooks` 실행 → 목록에서 검토 → trust해야 활성화

Step 4 **커밋 요청 → Hook이 자동으로 테스트 실행**: 테스트가 실패하면 Codex가 "커밋 차단됨" 피드백을 받고 재시도/수정

```
> 변경사항 커밋해줘
▶ bun run test … FAIL Error: Tests failed (exit 2) ← 커밋 차단
```

Rule은 권고, Hook은 강제 · 테스트 통과가 "습관"이 아니라 "시스템"이 된다

Reminder Hook: 테스트 파일 누락 경고

발동 조건: `src/` 아래 파일을 **수정·생성**했는데 같은 이름의 `*.test.*` 파일이 **없을 때** → stderr로 경고 1줄 (작업은 그대로 진행된다)

Step 6 **Reminder Hook: 테스트 파일 누락 경고** (**참고 프롬프트**): src/ 수정 시 대응 테스트 파일 존재 여부 확인 → 없으면 stderr로 경고만 (PostToolUse)

.codex/hooks.json (프로젝트 로컬, PostToolUse) → check-test.sh 호출

```
FILE=$(jq -r '.tool_input.command' \
  | grep -oE '\*\*\* (Update|Add) File: .*' \
  | sed -E 's/.*File: //' | head -1)

if [ -n "$FILE" ]; then
  E="${FILE##*.}"; T="${FILE%.$E}"
  [ -f "$T.test.$E" ] || echo "⚠ 테스트 파일 없음: $FILE" >&2
fi
```

Step 7 **문자 수 카운터 추가, Hook 동작 확인**: 테스트 없이 새 컴포넌트 생성 시 경고 출력, 코드는 계속 생성되지만 TDD 규칙 상기 (exit 0 통과)

Codex 프롬프트

```
> 프롬프트 입력창 아래에 글자 수 카운터를 추가해줘
```

관찰 포인트

⚠ 테스트 파일 없음: CharCounter.tsx, 파일은 생성됨(exit 0 통과)

exit 0 + stderr = Reminder · exit 2 = Protection (같은 Hook, 다른 역할)

Sub-agent: evaluator 생성

목표: 읽기 전용 서브에이전트로 독립 평가, 작업자 ≠ 검증자 편향 제거

Step 8 **evaluator 서브에이전트 생성 (참고 프롬프트)**: 읽기 전용 도구 + Sol 모델 → 메인 오염 없이 독립 평가 (Sub-agent)

.codex/agents/evaluator.toml (프로젝트 스코프)

```
name = "evaluator"
description = "기능 구현이나 버그 수정을 마친 뒤 코드를 읽기 전용으로 독립 검증하는 서브에이전트"
sandbox_mode = "read-only"
developer_instructions = "코드를 읽고 평가만 하라. 수정하지 마라."
# 강의 환경은 기본 모델(6.1 Sol) · 실전 권장: Astra (아래 설명 참조)
```

Step 9 **자동 위임 테스트: "evaluator"라는 이름을 부르지 않는다**: 작업자(메인) ≠ 검증자(서브), 자기 평가 편향 제거 ☆

```
> 방금 구현한 localStorage 저장 기능, 검증까지 마쳐줘
```

관찰: 이름을 안 불렀는데도 **evaluator에게 스스로 위임하는가**: 안 되면 Step 10에서 Hook으로 강제한다.

Step 10 **실전 권장: 6.1 Sol(작업) × Astra(검증) 분리**: 비용 대비 편향 제거 극대화

6.1 Sol(작업) → Stop Hook 자동호출 → Astra Evaluator → 검증 피드백

Step 11 **커밋** : 여기까지 만든 것 커밋해줘

Stop / SubagentStop Hook으로 검증자 자동 호출 → 사람 개입 없이 "작업→검증" 파이프라인 완성

Sub-agent, 실행 중에도 개입할 수 있다

spawn하면 끝이 아니다, 실행 중 **조회·추가 지시·중단**이 모두 된다



💡 **실측 확인 3가지: ① "끝나는 대로 보고해줘" 지시가 그대로 작동한다 ② 대기 중 사용자 입력이 오면 즉시 응답한다(wait 인터럽트) ③ 턴이 끝나도 남은 서브에이전트는 계속 돌고, 다음 턴에서 결과를 회수할 수 있다**

⚠ 공식 경고: 여러 에이전트가 동시에 코드를 수정하면 충돌·조율 비용이 늘어난다, 파일 소유권을 나눠서 맡기는 운영이 필요하다

출처: OpenAI, **Codex Subagents 문서**

개발 하네스 완성!

Part 2~4에서 구축한 전체 하네스

AGENTS.md + 중첩 구조 (Rule)

AGENTS.md TDD 규칙 섹션 (Rule, 권고)

.agents/skills/, frontend-design, commit, create-pr/

.codex/agents/evaluator.toml (Sub-agent)

config.toml, 민감 파일 차단 규칙 + Hook 2개 (Protection 1 + Reminder 1)

Part 4 체감 포인트

☆ Part 4에서 완성한 안전장치

권고만으로 안 되던 것을 구조로 강제했다

Permission · Protection Hook

사람 또는 에이전트가
실수해도 시스템이 막는다

정적 패턴은 Permission
· 동적 조건은 Hook

Reminder Hook

AI가 규칙을 잊어도 시스템이 상기시킨다

Sub-agent

Generator가 쓴 코드를
Evaluator가 검증한다

Part 5. 스케일 - Worktree + 에이전트 아키텍처 + 브라우저 검증

$ PART 5

Part 5 구성

실습 5에서 Worktree로 스트리밍을 구현하고, 실습 6에서는
Bun 사례의 못다 한 질문을 **독립적으로** 조사합니다.

이론	**Git Worktree**, 병렬 개발 · **Sub-agent fan-out으로 딥리서치**, 배운 부품을 조합한다
실습 5	**Worktree 스트리밍 구현**. 독립 환경 생성, 원하는 UX를 직접 지시해 구현, 하네스 상속 확인
실습 6	**Bun 사례 팩트체크**. researcher fan-out으로 앞선 사례의 남은 의문 4가지 조사, 출처·신뢰도 붙여 문서로 정리
실습 7	**내장 Browser 검증 루프**. 구현 → 화면 확인 → 문제 있으면 수정
이론	**장기 실행 에이전트 하네스**. 외부 메모리를 어디에 둘 것인가 · 규칙의 SSOT · 마찰 로그

Git Worktree: 병렬 개발

- **왜 Worktree가 필요한가**
  - 일반: 브랜치 전환 시 stash → checkout → 수정 → checkout → stash pop
  - Worktree: 여러 브랜치를 동시에 별도 디렉토리로 체크아웃
- **Worktree + Codex = 병렬 AI 개발**
  - 앱에서 입력창 위 **워크트리** 스위치를 켜면 프로젝트 복사본이 만들어집니다
  - 각 워크트리에서 독립된 채팅 · 세션 간 컨텍스트 완전 분리

뒤에서 볼 Bun 재작성 사례가 이 구조입니다. 워크트리 4개 × 각 16개 = 최대 64 인스턴스 동시 실행.
워크트리는 병렬의 상한을 올리는 실행 환경입니다. 내부는 git worktree입니다.

Worktree 기본 사용법

Worktree 기본

앱에서는 입력창 위 **워크트리** 스위치, 아래 git 명령은 원리와 정리용

Worktree는 한 저장소의 여러 브랜치를 각각 다른 디렉토리로 "체크아웃"하는 git 기능. 앱이 대신 만들어 줍니다

생성 (add)

새 브랜치·디렉토리를 한 번에

```
$ git worktree add -b feature/streaming ../streaming
```

-b 는 새 브랜치 생성. 기존 브랜치 체크아웃이면 -b 생략

목록 확인 (list)

현재 만들어진 모든 worktree 보기

```
$ git worktree list
```

경로 · HEAD 커밋 · 브랜치명이 출력됨

제거 (remove)

작업 끝난 worktree 정리

```
$ git worktree remove ../streaming
```

디렉토리 삭제 + git 등록 해제 (브랜치는 유지)

프루닝 (prune)

수동 삭제된 worktree 흔적 청소

```
$ git worktree prune
```

rm -rf로 지웠을 때 git 내부 메타데이터 정리

`git worktree add` 로 만들면 `main-project/` (원본) ↔ `streaming/` (feature/streaming 브랜치) · 앱이 만들면 `~/.codex/worktrees/` 아래에 브랜치 없이(분리된 HEAD) 생긴다. 어느 쪽이든 같은 .git을 공유하는 독립 워킹트리

병렬 작업 패턴: 2가지 비교

⤮ 병렬 작업 패턴 비교

같은 "여러 에이전트"여도 구조와 쓰임이 다릅니다

Subagent · Worktree 차이

Subagent

위임형 · Part 4 실습

메인 → 하위 (1:N)

- **통신**: 결과 보고만 (단방향)
- **브랜치**: 같은 세션 · 독립 컨텍스트
- **상태**: ⊘ 정식 기능
- **비용**: 1.5~2×

☆ 적합한 경우. 코드 탐색 · 검증 요약 · 단일 태스크 위임

Worktree

(실습)

분산형 · Part 5 실습

독립 개발자 N명 (N:0)

- **통신**: 없음 · 완전 독립
- **브랜치**: 각자 별도 브랜치 · 디렉토리
- **상태**: ⊘ git 표준 기능
- **비용**: N× (병렬 개수만큼)

☆ 적합한 경우. 독립 기능 동시 개발 · 실험 브랜치 병렬 시도

◎ 핵심: "통신이 필요한가?" 로 선택합니다. 보고만 받으면 Subagent, 완전 독립이면 Worktree

실습 5. Worktree 스트리밍 구현

실습 5 개요

- **미션**: Git Worktree에서 메인 브랜치를 건드리지 않고,
  원하는 스트리밍 UX를 직접 지시해 기능을 구현하세요
- **지시할 UX: 생성 과정을 눈으로 보여 준다**
  - 컴포넌트를 만드는 동안 "생성된 컴포넌트" 카드의 **코드 탭**에 AI가 만드는 코드가 **실시간으로** 채워집니다
  - 코드가 다 만들어지면 **미리보기 탭으로 자동 전환**해 완성된 컴포넌트를 보여 줍니다
- **관찰**: 프롬프트에 안 적은 함정(코드 블록 표시 ```, 모델 폴백, provider별
  스트림 형식)까지 Codex가 코드와 AGENTS.md를 읽고 챙기는지 봅니다
- **이 실습에서 체험할 것**: 앱에서 워크트리를 켜고 프롬프트 한
  번으로 구현까지, 오전에 정비한 하네스가 따라오는지 확인

25분 · 무엇을 만들지는 프롬프트로 말하고, 지켜야 할 규칙은 하네스가 챙깁니다

Worktree 셋업

목표: 첫 프롬프트를 보내기 전에 워크트리를 켠다 · 메인 폴더는 그대로

Step 1 **워크트리 켜기**: 새 채팅 입력창 위 오른쪽 **워크트리** 스위치를 켠다. "이 채팅은 어떤 브랜치에서 시작할까요?"에서 **현재 브랜치**를 고르면 커밋 안 한 오전 변경까지 따라온다 (첫 프롬프트를 보낼 때 워크트리가 생긴다)

Step 2 ⚠ **환경 파일(.env)**: gitignore된 파일은 워크트리에 안 따라온다. 저장소 루트 `.worktreeinclude`에 `.env`를 적어 두면 앱이 복사한다 (실습 저장소에 적혀 있다 · Claude Code도 같은 파일을 읽는다)

> 💡 **.env만의 얘기가 아닙니다. 로컬 캐시·인증서·머신 전용 설정도 마찬가지입니다. 반복되면 `.worktreeinclude`에 패턴을 넣으세요.**

Step 3 **권한은 전체 권한 (Full access)**: 워크트리의 git 기록은 원래 폴더의 `.git`에 있어, 기본 권한으로는 커밋이 막힐 수 있다 (알려진 문제)

> 💡 **터미널에서는 `codex --worktree "프롬프트"` 한 줄로도 시작한다 (소개만) · CLI 워크트리는 아직 `.worktreeinclude`와 setup 스크립트를 적용하지 않는다**

$ Pro Tip · 자동화

.env 자동 복사, Pro Tip

새 워크트리마다 .env 복사와 의존성 설치를 반복하지 않게 한다

추천 · Codex 앱·Claude Code 공통

.worktreeinclude

```
# 저장소 루트 .worktreeinclude
# .gitignore 문법, gitignore된 파일만 복사
.env
```

Codex 앱·Claude Code가 워크트리를 만들 때 자동 복사:
파일 하나로 두 도구 모두, 커밋해 두면 팀 전체에 적용

setup 스크립트: 앱의 로컬 환경에 `bun install` 을 넣어
두면 새 워크트리마다 자동 실행 (설정은 `.codex/` 에 저장)

Part 4 연결 · Hook으로 할 때

SessionStart Hook

```
# 원래 폴더 위치를 git으로 찾는다
M=$(git rev-parse --path-format=absolute \
      --git-common-dir)
[ -f .env ] || cp "$(dirname "$M")/.env" .
```

앱 워크트리는 `~/.codex/worktrees/` 에 생겨서
`../main/.env` 같은 상대 경로로는 원본을 못 찾는다

◎ `.worktreeinclude` 는 Codex 앱과 Claude Code가 **모두 지원**합니다. 단, Git 기능이 아니라 도구가 **자기 워크트리**를 만들 때만 복사합니다.
`git worktree add` ·Codex CLI·원격 워크트리에는 안 먹습니다

$ 구현 + 커밋

스트리밍 구현 + 커밋

목표: 스트리밍 기능 완성 → PR → 메인 복귀

❹ 구현

구현

워크트리 채팅에서 한 번에 · **gist**

```
> bun install 하고
  스트리밍 기능 추가해보자.
  생성 중엔 코드 탭에서 실시간으로,
  완료되면 미리보기 탭으로 전환.
  끝나면 개발 서버 띄우고
  주소 알려줘.
```

❺ 검증

브라우저 확인

구현 후 실제 동작 검증

```
# 메인 폴더 개발 서버는
# 먼저 끈다 (API 포트 공유)
# 알려 준 주소로 접속

# 체크포인트:
# - 코드 탭 실시간 표시 ✓
# - 완료 후 미리보기 전환 ✓
# - Hook 자동 실행 ✓
```

❻ 스킬

커밋 + PR

Part 3 스킬 재사용해 마무리

```
# 분리된 HEAD라 브랜치부터
# (채팅 상단 브랜치 생성)
> $commit

> $create-pr
# 승인 후
# PR 본문 작성 + 생성
```

💡 **워크트리 켜고 원하는 UX 지시 → Codex가 설치·구현·서버 → 브라우저 확인 → 브랜치 만들고 스킬로 마무리**

$ 개념 정의

딥리서치란 무엇인가

여러 에이전트가 주제를 나눠 조사하고, 결론마다
출처를 남기고, 검증을 거쳐 하나의 리포트로 합칩니다.

fan-out 조사
주제를 항목으로 쪼개 병렬로 파고듭니다

→

출처 추적
결론마다 근거를 남기고, 확신 없으면 미검증 표시

→

검증
근거가 결론을 실제로 뒷받침하는지 재확인

→

리포트 합성
출처·신뢰도 표기를 유지한 채 하나로 정리

OpenAI Deep Research

`o3-deep-research` 모델로 서비스되는
별도 API 제품. 웹 대상 멀티스텝 조사 + 인용

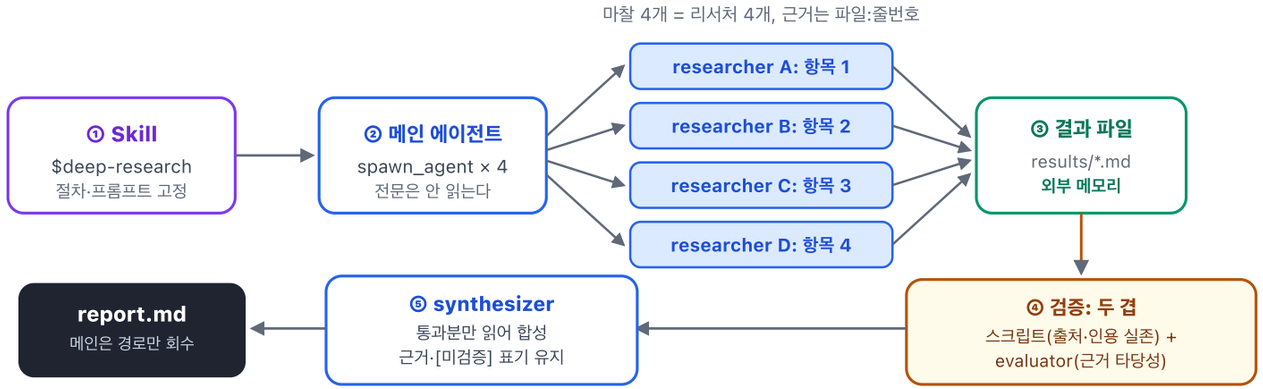
Claude Code 내장 `/deep-research`

Dynamic Workflow에 번들된 기능.
Claude Code 제품 안에만 있습니다

제품마다 구현은 다르지만 fan-out → 출처 추적 → 검증 → 합성이라는 절차 자체는 같습니다. Codex엔 이걸 제품으로 주는 기능이 없습니다. 다음 장에서 직접 조립합니다.

딥리서치, 부품을 조합해 직접 만들어 보겠습니다

Codex에 딥리서치 제품 기능은 없습니다. 오늘 배운 **Skill + Sub-agent + 파일 메모리 + 검증**으로 조립합니다



💡 **5개 부품 전부 Part 2·4에서 이미 배웠습니다, 딥리서치는 신기능 소개가 아니라 배운 부품의 조립입니다**

💻 모범 답안: **github.com/devbrother2024/codex-deep-research**

macOS · Linux: `npx skills add devbrother2024/codex-deep-research`

Windows: `cmd /c npx skills add devbrother2024/codex-deep-research`

$ 검증의 분업

기계로 확인할 것, 판단이 필요한 것

"정적 차단 = Permission · 동적 검증 = Hook"과 같은 원칙이 검증에도 적용됩니다

✓ 스크립트: 기계 검증

- 결론마다 `근거: 파일:줄번호` 표기가 있는가
- 인용한 파일·줄번호가 실제로 존재하는가
- **FAIL이면 해당 항목만 재실행**. 파일 기반이라 완료분은 안 건드립니다

🔍 evaluator: 판단 검증

- 결론이 그 근거로 **실제 뒷받침되는가**
- 과장·근거 범위를 넘는 일반화가 없는가
- Part 4에서 만든 그 evaluator를 **그대로 재사용**

📖 참고한 구조: **Weizhena/Deep-Research-skills**(파일 기반 조율 패턴)

사례 깊이 보기: Bun 재작성

실사례

Bun 창립자가 혼자서 JS 런타임 코어 53만 줄을 11일 만에 Rust로 다시 쓴 사례

◎ 왜 했나

(동기)

메모리 관리를 사람이 하다 계속 사고

Zig는 해제를 직접 챙겨야 해서, 놓치면 그대로 크래시 버그

Rust는 컴파일러가 막아준다

같은 실수가 컴파일 단계에서 걸립니다. 버그 부류를 통째로 제거

예전엔 엄두도 못 낼 일

저자 표현으로 "엔지니어 3명이 1년 걸릴 일". AI가 리팩토링의 계산을 바꿈

규모: 핵심 수치 4가지

(수치)

기간

11일 (사람 1명)

동시 실행

Claude Code(Fable 5)를 워크트리 4개 × 각 16개 = 최대 64개 동시 실행

결과

Zig 53.5만 줄 → Rust diff +100만 줄

비용

API 환산 ~$165,000

출처: **bun.com/blog/bun-in-rust**

머지 조건은 하나. 기존 테스트 약 6만 개 전부 통과 (테스트 삭제·스킵 0건)

사례 깊이 보기: 어떻게 굴렸나

📖 운영기

프롬프트 한 방이 아니라 3가지 준비와 11일간의 운영으로 만든 결과

📖 어떻게 굴렸나: 3가지

구조

① 규칙 문서부터 (사람이 쓴 시간: 3시간)
"Zig를 Rust로 이렇게 번역하라" 가이드를 먼저 만들고, 그 문서부터 검증

② 만드는 역할 · 트집 잡는 역할 분리
구현자 1명 + "이 코드는 틀렸다"만 찾는 리뷰어 2명 + 수정자 1명

③ 컴파일 에러 목록 = 할 일 목록
에러 16,000개를 모듈 단위로 묶어 에이전트들이 나눠 갖습니다. 같은 파일을 두 에이전트가 건드릴 일이 없고, 남은 에러 수가 곧 진행률입니다

🔥 사고가 날 때마다 규칙이 늘었다

(사람의 역할)

에이전트끼리 작업을 지워버림
→ "커밋 말곤 git 명령 쓰지 마" 규칙 추가

빈 함수로 눈속임 통과
→ "변명 주석이 길면 틀린 코드" 리뷰 규칙 추가

결국 사람의 일 = 감시하고 규칙 추가하기
11일 내내 실패 패턴을 발견할 때마다 규칙으로 전환

📖 출처: **bun.com/blog/bun-in-rust**

실습 6. 딥리서치 직접 구현
Bun 재작성 사례, 팩트체크

$ 실습 6 개요

researcher 서브에이전트를 직접 만들어, 딥리서치를 완성한다

딥리서치 스킬을 직접 만들어보겠습니다.

이 실습에서 체험할 것

- 앞서 본 구조를 목표로, **researcher.toml과 그걸 부를 미니 스킬을 직접 작성**합니다. Part 2·4에서 배운 형식 그대로입니다
- 완성된 구조로 바로 fan-out을 실행하고, 검증 방식도 직접 정합니다
- 모범 답안(codex-deep-research) 코드를 확인해 내가 쓴 파일과 비교합니다, 이 도구가 **코드베이스 인용에 맞춰 설계됐다는 것**도 함께 확인합니다

$ 직접 만들기

researcher.toml, 직접 써본다

방금 본 구조를 목표로, **researcher 서브에이전트를 직접 정의**합니다. Part 4 evaluator.toml과 같은 형식입니다.

✎ 작성

researcher.toml + 미니 스킬

오른쪽 골격의 빈칸을 채웁니다, 서브에이전트 하나(researcher.toml) + 그걸 부를 딥리서치 스킬 하나(트리거 이름 + 절차 한 줄). evaluator.toml과 필드가 같습니다

💡 **출처는 웹입니다.** `근거: 경로:줄번호` **대신 "출처: URL + 인용문"을 요구하도록 developer_instructions을 직접 바꿔 적어야 합니다, 모범 답안과 갈리는 지점이 바로 여기입니다**

채워야 할 골격: 빈칸은 직접

```toml
# agents/researcher.toml
name = "researcher"
description = "___(언제 부를지)___"
default_permissions = ":workspace"
developer_instructions = """
  담당 항목은 인자로 받는다 · 출력 형식 · "출처: URL"
  ___(직접 채우기)___
"""

# skills/deep-research/SKILL.md
---
name: deep-research
description: ___(트리거 조건)___
---
"$deep-research [주제] 조사해줘"
→ researcher 4개를 그 주제로 fan-out 호출
```

📖 전체 프롬프트 스켈레톤: **gist.github.com/devbrother2024/codex-deep-research-diy-prompt.md**

$ 직접 만들기

완성된 구조로 돌리고, 검증까지 직접 정한다

방금 쓴 researcher.toml로 fan-out을 실행하고, 검증 방식도 직접 정해봅니다.

3 실행

내 스킬을 호출해 fan-out

researcher를 직접 부르지 않습니다. 스킬이 부릅니다, 방금 본 Bun 사례를 검증 대상으로 넣어서 호출합니다

```
> $deep-research 다음 4가지를 조사해줘:
1. Zig 창시자 "unreviewed slop" 비판의 정확한 근거
2. "버그 128개 해소·메모리 11배 개선" 수치의 출처
3. Claude Code가 Rust판 Bun 위에서 구동한다는 근거
4. 이 리라이트에 대한 커뮤니티의 실제 반응
```

☆ 4개가 한 번에 안 뜰 수 있습니다. 동시 실행 한도에 걸리면 메인이 배치로 이어서 띄웁니다. Part 4(slide-88b)에서 이미 본 패턴입니다

4 검증

evaluator로 검증, 한계는 관찰만

Part 4에서 만든 evaluator를 그대로 부릅니다

```
> 방금 나온 결과 파일들, evaluator로 검증해줘.
```

- 결과 파일 4개, **형식이 서로 같은가**
- 기계적 검증(출처·인용 실존)은 **누가 했나**
- 다시 실행하면 **처음부터 다시 도나요?**

리서치 품질은 검증 단계에서 갈립니다. 다음 스텝에서 이 관찰들을 모범 답안 코드와 하나씩 대조합니다

$ 모범 답안 살펴보기

모범 답안 코드를 확인한다

설치하지 않습니다. researcher.toml, SKILL.md, validate_research.py를 직접 읽으며 내가 만든 것과 견줍니다.

5 확인

모범 답안 코드 확인

내 스킬 폴더는 그대로 두고, 별도 위치에서 코드만 봅니다

```
$ git clone https://github.com/devbrother2024/codex-deep-
research /tmp/codex-deep-research
$ cat /tmp/codex-deep-research/SKILL.md
$ cat /tmp/codex-deep-research/agents/researcher.toml
$ cat /tmp/codex-deep-research/scripts/validate_research.py
```

☆ researcher.toml은 Part 4에서 만든

evaluator.toml과 형식이 같습니다. 낯설지 않을 겁니다

6 견주기

Step 4의 세 관찰, 이제 답을 찾는다

내가 만든 것에 없던 게 여기엔 있는지 확인합니다

- **형식**: 내 developer_instructions과 이 researcher.toml, 출력 형식 규칙을 어떻게 다르게 적었나요?
- **기계 검증**: 내 정의엔 검증 스크립트가 없었습니다. 여긴 validate_research.py로 분리했습니다
- **재개**: 내 정의에 완료 항목 스킵 규칙을 넣었나요? SKILL.md는 이걸 절차로 못 박아뒀습니다

Part 5 체감 포인트

- **Worktree = 독립 환경에서 실험**
  · 메인 브랜치를 건드리지 않고 기능 개발
- **/plan = 코드 수정 없이 분석 먼저**
  · 영향 범위 파악 후 승인 → 안전한 구현
- **Sub-agent fan-out = 배운 부품의 조합**
  · Skill+Sub-agent+파일 메모리+검증을 엮어 딥리서치를 직접 만듭니다 (실습 6)
- 내장 Browser = 실제 동작 확인
  · 필요한 변경만 구현 직후 화면에서 확인합니다 (실습 7)

실습 7. 내장 Browser 검증 루프

지금까지 쓰던 그 레포에, 구현 직후 화면을 직접 확인하는 습관을 **하네스로 굳힙니다**

브라우저 검증은 테스트를 대신하지 않는다

브라우저 검증은 마지막 레이어입니다.
빠른 검사부터 실행하고 필요한 경우에만 실제 화면을 봅니다.

STEP 1
Lint · Type
빠른 정적 검사
매 변경마다 실행

→

STEP 2
Unit Test
코드 동작 확인
회귀 방지

→

STEP 3
적용 판단
브라우저가
필요한가?

→

STEP 4
브라우저 검증
실제 사용자 흐름
결과·근거 보고

변경에 맞는 검증만 실행하고, 결과와 근거를 함께 보고합니다

브라우저 검증은 필요한 변경에만 실행한다

브라우저 의존도와 실패 영향을 기준으로 고릅니다

실행 · 생략 · 강제를 나눈다

▶ 실행

브라우저 확인 필요

- **사용자 동작 변경**
- **폼**: 입력 검증 · 포커스
- **상태**: 저장소 · 화면 이동
- **통신**: 네트워크 결과
- **화면**: 접근성 · 반응형

이번 실습: 폼 · 포커스 · 요청 차단

– 생략

코드로 충분 (절약)

- **내부 변경**
- **문구**: 동작 없는 텍스트
- **스타일**: 동작 없는 시각 수정
- **코드**: 리팩터링 · 타입
- **서버**: 백엔드 전용 변경

생략 기준: 브라우저로 새로 확인할 동작이 없음

! 강제

실패 영향이 큼 (Hook·CI)

- **반드시 확인**
- **권한**: 접근 제어
- **결제**: 금전 흐름
- **삭제**: 복구 어려운 변경
- **외부**: 실제 시스템 변경

강제 기준: 자동 호출에만 맡기지 않습니다

◎ 핵심: 브라우저 검증자를 추가했다면 실행 기준도 함께 적습니다
description과 AGENTS.md로 유도하고, 중요 흐름은 Hook·CI에서 강제

$ 확인 규칙

AGENTS.md에 확인 규칙을 반영한다

이미 있는 하네스를 중복으로 만들지 않습니다, 내장 Browser라는 기존 하네스에 언제 쓸지 규칙만 얹습니다

Step 1 이렇게 의도를 설명하면, 코딩 에이전트가 실습 2의 AGENTS.md에 확인 규칙을 반영합니다

```
AGENTS.md에 규칙을 추가해줘.

폼·포커스, 저장소 상태(Cookie·localStorage·로그인), 화면 이동, 네트워크 반영, 권한·결제·삭제처럼 실패 비용이 큰 변경을
만들었으면, 구현 후 내장 Browser로 실제 화면을 확인하고 문제가 있으면 그 자리에서 고치고 다시 확인한다는 규칙을 Do's에 추가해줘.

문구·스타일처럼 새 동작이 없는 변경은 확인하지 않는다는 것도 Don'ts에 추가해줘.

기존 구조와 규칙은 그대로 두고 항목만 추가해줘.
```

⚠ **전제 확인**: 실습 2의 AGENTS.md가 루트에 남아 있어야 이 요청이 반영됩니다

◎ 규칙(언제) + 링킹(읽히는가): 독립 검증이 아니라 실행 검증입니다

$ 구현 + 확인

실습 · 미리보기 로딩 오버레이 추가

목표: 생성·재생성 중 미리보기에 로딩 오버레이를 띄우고,
방금 만든 하네스가 스스로 화면을 확인하는지 관찰

3 · 구현 아래 프롬프트를 태그 없이 그대로 복사해서 씁니다, 확인 방법은 적지 않고, Step 1에서 쓴 규칙이 판단하게 합니다

```
컴포넌트를 생성하거나 재생성하는 동안 미리보기 카드 위에 반투명
오버레이와 로딩 스피너를 보여줘.
생성이 끝나면 오버레이가 사라지고 새로 생성된 결과가 그대로 보여야 해.
재생성·삭제 같은 기존 기능은 그대로 동작해야 하고, 관련 테스트도 추가해줘.
프로젝트 규칙에 따라 동작을 확인한 뒤 변경 내용과 근거를 알려줘.
```

4 · 확인 **하네스 관찰**: 구현 뒤 내장 Browser 자동 확인

```
(내장 Browser가 자동으로 열려 화면을 확인합니다)
```

자동으로 확인이 안 되면
"방금 구현한 기능, 내장 Browser로 직접 확인해줘."

전체 프롬프트 ·
gist.github.com/devbrother2024/ab9c16708b21bcbf8d4d81a240073d73

기능을 일부러 깨뜨리지 않습니다 · 문제가 있으면 확인된 것만 수정하고 다시 확인합니다

구현이 끝나면 하네스가 화면을 확인한다

> 하네스가 변경 내용을 보고 확인 대상을 판단합니다.
> 같은 에이전트가 확인하고, 문제가 있으면 그 자리에서 고칩니다.

STEP 1
요청 확인
완료 기준을 먼저 정리합니다

→

STEP 2
테스트 · 구현
메인 에이전트가 기능을 만듭니다

→

STEP 3
실행 기준 판단
브라우저 확인 대상인가?

→

STEP 4
화면 확인
내장 Browser로 직접 확인합니다

🔄 **문제없으면 완료 · 문제 있으면 그 자리에서 수정 후 다시 확인**

$ PROJECT · BROWSER VERIFICATION

프로젝트에 맞는 브라우저 검증 기준을 정한다

① 입력 동작인가?

폼 검증 · 포커스 · 제출처럼 실제 사용자 조작을 확인합니다.

② 브라우저 상태가 바뀌나?

저장소 · 로그인 · 화면 이동처럼 코드만으로 판단하기 어려운 상태를 봅니다.

③ 실패 영향이 큰가?

권한 · 결제 · 삭제 · 외부 변경은 Hook · CI · 전용 E2E로 강제합니다.

④ 브라우저가 꼭 필요한가?

문구 · 내부 리팩터링 · 타입처럼 새 동작이 없으면 생략합니다.

$ 심화 모듈

장기 실행 에이전트 하네스

외부 메모리 · 세션을 잇는 기록 · 마찰 로그

세션이 끝나면 기억도 끝난다

컨텍스트 윈도우는 유한합니다 · 복잡한 프로젝트는 한 세션 안에 끝나지 않습니다

실패 패턴 ① One-shotting

한 번에 전체를 구현하려다 **컨텍스트가 중간에 소진**: 기능이 반만 구현된 채 세션 종료. 다음 세션은 "무슨 일이 있었는지" **추측**부터 시작합니다

실패 패턴 ② 조기 완료 선언

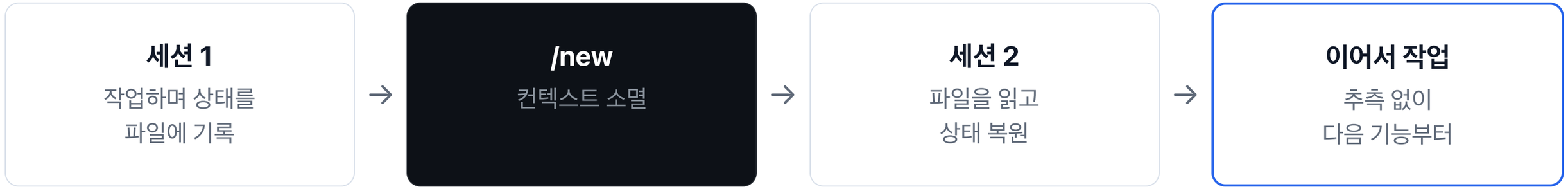
계획된 기능 중 일부만 구현하고 **"작업 완료"라고 선언**: 검증 없이 완료 표시. 남은 작업이 어디까지인지 아무도 모릅니다

💡 지금까지 만든 하네스(Rule · Skill · Hook · Sub-agent · MCP)는 전부 한 세션 안의 장치: 세션을 넘는 기억은 아직 없습니다

출처: **Anthropic: Effective harnesses for long-running agents**

파일시스템이 세션을 잇는 기억이다

> **"새 컨텍스트 윈도우로 시작한 에이전트가 작업 상태를 빠르게 파악할 방법을 찾는 것, 그것이 핵심이었다"**
>
> "The key insight here was finding a way for agents to quickly understand the state of work when starting with a fresh context window, which is accomplished with the claude-progress.txt file alongside the git history."
>
> Anthropic, Effective harnesses for long-running agents



Part 1에서 본 "외부 메모(Structured Note-taking)": 개념으로만 소개했던 그 패턴이 이것입니다. DB도 전용 서버도 아닙니다. **그냥 파일과 git**입니다.

$ .agents

그 파일들을 어디에 둘 것인가

`.agents/`: 세션이 리셋돼도 살아남는 **중앙 통제소**

```
.agents/
├── context/ 조사 결과 · 사실 수집
├── specs/ 명세 · 구현 계획
├── work/ 작업 이력 · 인계 노트
├── reviews/ 검증 실패 기록
└── friction.md 마찰 로그

.agents/skills/ 반복 절차 (스킬)
AGENTS.md 항상 지켜야 하는 규칙
```

디렉토리 이름은 팀 합의일 뿐입니다. 중요한 것은 **"세션 밖에 있고, 다음 세션이 반드시 들르는 자리"**가 하나로 정해져 있다는 것

세 가지가 한자리에 모인다

- **작업 이력**: 무엇을 왜 그렇게 했는지. git log가 "무엇을 바꿨나"라면 이쪽은 "무엇을 알아냈나"
- **인계 노트**: 다음 세션(또는 다른 에이전트)이 읽고 이어받는 상태
- **스킬**: 세 번 반복된 절차가 파일로 굳은 자리

컨텍스트 윈도우는 휘발되고 저장소는 남습니다. 남는 쪽에 통제소를 두면 세션 수명이 하네스 수명을 정하지 못합니다

$ 규칙의 출처

규칙은 어디서 오는가

산출물은 프로젝트 `.agents/` 에 남습니다, 그 산출물을 쓰게 만드는 **규칙은** `~/.codex/AGENTS.md` **한 곳**에서 옵니다

글로벌 규칙

`~/.codex/AGENTS.md`

슬러그 · work notes · 마찰 로그 계약

수정은 이 한 곳에서만

→

모든 프로젝트 세션

어느 레포를 열어도

같은 규칙이 로드된다

프로젝트 `AGENTS.md` (팀의 계약)와 글로벌 `~/.codex/AGENTS.md` (내 에이전트 운영 계약)는 층이 다릅니다: 팀 전체가 공유할 규칙은 레포에, 나 혼자 쓰는 운영 습관은 글로벌에 둡니다

$ 사람이 붙어 있는 일반 작업

일반 작업은 기록 두 줄이면 충분하다

사람이 세션에 붙어 함께 판단하는 보통의 작업, 무거운 구조 없이 이 루프면 세션이 이어집니다

시작 `git log` + `.agents/work/notes.md` 읽기: **직전 세션이 남긴 상태 파악** (추측 없이 시작)

작업 평소처럼 진행: **완료 판정은 사람이 그 자리에서** 합니다. 별도의 기능 명세 파일이 필요 없는 이유입니다

종료 **커밋 +** `notes.md` **갱신**: 무엇을 했고 · 무엇을 알아냈고 · 다음에 뭘 할지 (다음 세션의 시작 지점)

이 수준으로 /new를 넘습니다. 더 무거운 구조(기능 명세 파일 · 초기화 전용 세션)는 사람이 자리를 비우는 무인 장기 실행에서만 필요합니다, 부록(Initializer 패턴)에서 다룹니다

$ ✎ .agents/friction.md

실수가 저절로 쌓이는 구조, 마찰 로그

사람 기억이 아니라 **파일이 백로그**를 들고 있어야 합니다, 기록 트리거를 규칙으로 정해 에이전트가 스스로 append하게

재설명
이미 알려준 것을 다시
설명하게 만들었을 때

되돌림
산출물을 되돌리거나
재작업 지시를 받았을 때

막힘
도구·권한·구조 문제로
우회가 필요했을 때

반복실수
같은 실수를 또 했다고 지적받았을 때

```
## 2026-08-13 | feature-2769 | 재설명
- 무엇: 한 줄 사실
- 패턴 후보: 어떤 규칙 부재가 원인인지 가설
- 발생: 2회째 (이전: 2026-08-05)
- 상태: open
```

- 트리거 발생 시 **그 세션의 에이전트가 즉시 append**: 사용자 요청 불필요
- 같은 패턴 재발이면 새 항목 대신 **발생 카운트만 올린다**
- 승격되면 한 줄로 압축 · 기각되면 삭제, **파일은 항상 한눈에 읽히는 크기**

같은 패턴 2회 → Rule 후보 · 3회 → Skill 후보: 마찰 로그가 Rule/Skill 승격 판단의 원천이 된다

◎ **Part 1에서 본 Hashimoto의 정의 그대로, 실수는 짜증이 아니라 하네스의 백로그입니다. 백로그를 쌓을 자리가 있어야 그 정의가 실행된다**

핵심 요약

외부 메모리는 파일이면 충분하다

DB도 프레임워크도 아닌 마크다운 파일과 git, 단순할수록 에이전트가 잘 지킵니다

일반 작업은 기록 두 줄이면 된다

시작할 때 읽고(git log · notes.md), 끝낼 때 남긴다, 완료 판정은 사람이 그 자리에서

실수는 하네스의 백로그다

최소 구조로 시작 → 실제 실패가 관찰될 때만 늘립니다 (Simple is the best)

근거: **Anthropic: Effective harnesses for long-running agents** · **Anthropic: Harness design for long-running application development**

Part 6. Codex 운영과 비용 최적화

무엇을 비용 최적화라고 부를까?

> **MEASURE**
> "총 실행 비용 ÷ 검증 통과 결과 수"

COMPARE FAIRLY: 같은 성공 조건을 충족한 실행끼리 비교한다.
짧은 답변보다 재작업 없이 통과한 결과가 더 저렴할 수 있다.

NUMERATOR: 총 실행 비용

- 입력 토큰
- 캐시 쓰기 · 읽기
- **출력 토큰 · 재시도 비용**

DENOMINATOR: 검증 통과 결과 수

- 테스트 통과
- 요구사항 충족
- 범위 준수
- 종료 조건 충족

비교 원칙 · 같은 성공 조건 · 같은 작업 범위 · 같은 검증 기준

$ 비용은 세 곳에서 샌다

비용이 새는 세 지점

무엇을 읽혔는지, 왜 다시 했는지, 왜 나눴는지를 확인한다

01 · Context: 컨텍스트 누수

끝난 작업과 불필요한 파일이 다음 요청에도 따라온다.

#무엇을 읽혔나?

02 · Retry: 재작업 누수

범위와 완료 조건이 없어 탐색·수정·설명을 반복한다.

#왜 다시 했나?

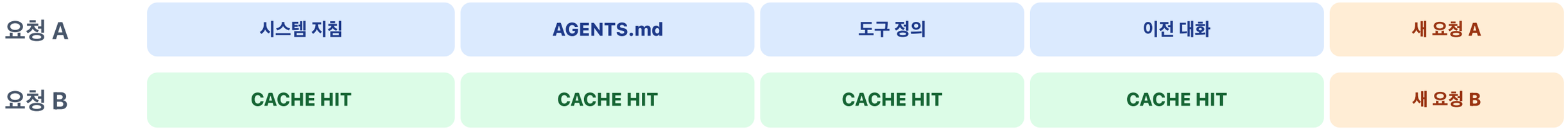
03 · Parallel: 병렬 실행 누수

독립 컨텍스트가 필요하지 않은데
여러 에이전트를 동시에 실행한다.

#왜 나눴나?

자동 캐시는 안정적인 prefix를 재사용

> 캐시는 프롬프트 앞부분이 정확히 같을 때 재사용된다.
> 모델·Effort 변경은 전체 캐시를 무효화한다.



✓ 앞부분 유지

같은 모델·Effort·규칙·도구 → prefix 재사용

✕ 앞부분 변경

모델·Effort·규칙 순서 변경 → 그 지점부터 재처리

파란 prefix를 고정하면, 두 번째 요청은 초록 구간을 캐시에서 읽고 주황 suffix만 새로 처리

$ 세션 경계 3가지

계속 · /compact · /new 선택 기준

지금까지의 맥락이 다음 요청에도 필요한지 먼저 판단한다

01 · 계속

- 목표와 작업 범위가 같다.
- 지금까지의 맥락이 다음 요청에도 필요하다.
- 후속 요청은 같은 세션에서 이어간다.

02 · /compact

- 목표는 같지만 대화가 길어졌다.
- 자연스러운 작업 경계에서 세부 기록을 요약한다.
- 요약을 기준으로 다음 단계를 진행한다.

03 · /new

- 목표·저장소·작업 범위가 바뀌었다.
- 이전 맥락이 새 작업에 필요하지 않다.
- 새 작업은 깨끗한 세션에서 시작한다.

/status · /compact · New chat

먼저 읽고, 필요할 때만 세션을 나누거나 요약한다

/status

· 세션 상태

채팅 ID, 컨텍스트 사용량, rate limit을 확인한다. 앱 공식 slash 명령.

/compact

· 같은 목표

대화가 길어지면 요약을 기준으로 이어간다. 세션을 버리지 않는다.

/new

· 새 채팅

New chat 또는 `Ctrl+N` / `Cmd+N`. 목표가 바뀌면 여기서 분리. /clear는 CLI 전용.

판단

· 결과와 함께 보기

테스트 통과·재시도·탐색 범위를 함께 본다. 토큰 수치만 낮추지 않는다.

$ NEXT SESSION · 4 CHECKS

다음 세션부터 네 가지만 확인한다

① 같은 목표인가?

같은 목표·범위면 세션을 이어간다.

작업이 바뀌면 `/new` (New chat)로 분리한다.

② prefix를 흔들었나?

규칙·도구·지침의 앞부분을 안정적으로 유지한다.

모델·Effort 변경은 새 캐시를 만든다.

③ 재작업이 반복되나?

목표·범위·검증·종료를 프롬프트에 표시한다.

같은 실패가 반복되면 멈추고 원인을 본다.

④ 병렬화가 이득인가?

독립적인 작업만 병렬로 실행한다.

같은 파일을 함께 수정하면 조율 비용이 커진다.

$ 격리 방식

Codex 권한·격리 방식

샌드박스부터 시작 → 한계 직면 → 안전한 대안 찾기

Step 1 **샌드박스 모드: 기본값이자 권장**

Codex가 기본으로 제공하는 격리, 위험 명령 필터링 + 사용자 확인 프롬프트 (approval policy + sandbox mode)

⚠ macOS · Linux · WSL2 에서만 지원 (순수 Windows는 불가, 강의에서도 미사용)

하지만 쓰다 보면... ↓

Step 2 **⚠ 전체 권한 (Full access · 예전 CLI YOLO)**

샌드박스 제약이 답답해서 풀고 싶은 유혹, 모든 권한 해제

⊗ Rule · Hook으로 안전장치 만들 수 있지만, 근본적으로 호스트 전체 접근 가능

안전하게 풀 수 있는 대안이 있다면? ↓

Step 3 · ☆추천 **DevContainer: 전체 권한을 안전하게 쓰는 방법**

컨테이너 안에서는 호스트와 분리되어 있으니 전체 권한을 켜도 호스트는 안전

컨테이너를 삭제해도 → 재생성만 하면 끝. 호스트 파일시스템은 영향 없음.

✓ **"전체 권한의 자유 + 호스트 격리", 두 마리 토끼**

개인적으로는 컨테이너 격리 방식(DevContainer) + Hooks(안전장치) 추가로 작업, 자유롭게 풀어놓고 쓸 수 있어서

$ DevContainer (2/2)

DevContainer 실전 설정

언어 달라도 구조는 같다, 이미지만 바꿔서 복붙

Node.js, .devcontainer/devcontainer.json

```
{
  "name": "Dev Container",
  "image": "...devcontainers/javascript-node:20",
  "postCreateCommand": "npm install -g @openai/codex &&
bun install",
  "remoteUser": "node"
}
```

Java (Maven): .devcontainer/devcontainer.json

```
{
  "name": "Dev Container",
  "image": "...devcontainers/java:17",
  "postCreateCommand": "npm install -g @openai/codex &&
./mvnw dependency:go-offline",
  "remoteUser": "vscode"
}
```

차이는 단 2줄: image (베이스 이미지) + postCreateCommand (설치 명령)

JS

Node.js

javascript-node:20

J

Java

java:17

Py

Python

python:3.12

Go

Go

go:1.22

컨테이너 기동: `devcontainer up --workspace-folder .` · 그다음 앱에서 그 폴더를 연다

Part 7. 하네스 리팩토링, 다이어트

비용을 정량화했으니, 이제 응답할 차례, **덜어내기**

$ 태도 전환

하네스는 낡는다, 이제 덜어낼 시점

- 한 번 만든 하네스는 영원하지 않다
- ① 모델이 점점 똑똑해진다
  - 예전엔 '계획부터', '바로 구현 금지'로 묶어야 했지만, 좋은 모델은 그냥 한다.
  - → 옛 규칙 = 과잉 제약
- ② 코딩 에이전트 제품 자체가 이미 하네스다
  - Codex·Cursor가 파일읽기·테스트·리뷰·메모리·MCP를 내장
  - → 내 하네스 = 제품 하네스 위에 또 올린 하네스
  - 제품이 업데이트되면, 내 규칙은 도움이 아니라 중복 지시가 될 수 있다.

◎ 쌓을 땐 필요해서 하나씩 쌓았다. 그런데 쌓다 보니 많아졌다, 지금은 그걸 줄일 차례다.

$ ⚠ 증상

도우려던 하네스가 방해가 된다

- **낡은 하네스의 증상**
  - ⚠ 간단한 수정에도 '계획부터' → 느려진다
  - ⚠ 무관한 문서를 매 세션 로드 → 컨텍스트 오염
  - ⚠ 작은 버그에 서브에이전트 소환 → 프로세스가 문제보다 커진다
  - ⚠ 제품 기본 기능을 내 규칙이 다시 지시 → 중복
  - ⚠ hook이 위험을 막기보다 매번 멈춤 → 마찰

- **→ '무엇을 더 붙일까?' 가 아니라 '무엇을 이제 빼도 될까?'**

$ 낡는 이유 ① 모델 진화 · 공식 근거

'레거시 스캐폴딩'은 공식 문서에 있는 말이다

강사의 관찰이 아니라, OpenAI가 GPT-5.6 프롬프트 마이그레이션 가이드에서 직접 권고한 상태다

> "Remove obsolete scaffolding, repeated instructions, and irrelevant tools." **configurations with leaner system prompts improved evaluation scores by roughly 10–15% while reducing total tokens by 41–66%.**

→ GPT-5.6은 예전 모델의 한계를 메우던 장황한 지시 없이도 스스로 재시도·계획·검증한다. 낡은 스캐폴딩을 지우면
평가 점수는 오히려 오르고 토큰은 줄어든다: 별도 검증 단계를 덧붙이는 '레거시 하네스 구조물'에도 똑같이 적용된다.

모델이 이미 하는 일

- 자기 결과 재확인 · 자기 교정
- "계획부터 세워라" 없이도 계획

같은 지시를 한 번 더 = 과잉 검증

그래서 지울 후보

- "반드시 다시 검증해라" 류의 상시 지시
- 모델이 이미 하는 일을 한 번 더 시키는 검증 단계

무엇을 남길지는 뒤의 결정표로 판정

출처: **developers.openai.com, Prompting guidance for GPT-5.6 Sol**

$ 낡는 이유 ② 제품 흡수 · 실증

실증: 제품이 내 하네스를 흡수한다

이론으로 배운 패턴, 실습으로 쌓은 하네스, 그리고 제품

이론 · Part 4~5

패턴으로 배웠다

Generator-Evaluator · Advisor

이름과 구조를 익혔다

→

실습 · Part 4~5

손으로 쌓았다

Sub-agent · Worktree로 병렬 하네스 직접 조립

내 손으로 만든 오케스트레이션

→

흡수

2026.5 · 제품

기능이 됐다

Dynamic Workflow · ultracode

제품 안의 하네스

◎ 패턴을 이해한 사람이, 무엇을 덜어낼지 판단할 수 있다

월간 감사에 추가: 내 하네스, 제품 기능과 중복은 없나?

$ 흡수 사례 컬렉션

내가 만든 하네스가 제품 기능이 되는 순간

직접 조립하던 것이 제품 안으로 들어오면, 내 버전은 그날부터 중복 후보다

조사 절차를 스킬로

직접 만들던 것

- 탐색 → 정리 → 문서화를 스킬로 묶어 굴렸다
- → **Workflow 번들**이 같은 절차를 제품 안에서 제공

제품 흡수

검증 루프를 Hook으로

직접 만들던 것

- Stop hook + 상태 파일로 "통과할 때까지" 루프를 자작
- → `/goal` : 별도 evaluator가 매 턴 조건을 재확인

제품 흡수 (전제조건 있음)

Ralph Loop

성격이 다른 사례

- 원 제안: Geoffrey Huntley, 2025-07, 프롬프트를 무한히 다시 먹이는 셸 한 줄
- → 저장소 1.7k stars, **플러그인 형태로 유통되는 커뮤니티 패턴**

공식 기능이 아니다

앞의 둘은 제품이 흡수한 것, 세 번째는 커뮤니티가 유통하는 패턴이다. 셋을 같은 층위로 말하면 틀린다, 그래서 다이어트 판정도 "제품 릴리스 노트"와 "커뮤니티 담론"을 나눠서 봐야 한다.

Ralph 원 제안: **ghuntley.com/ralph** · 채택 저장소: **github.com/ghuntley/how-to-ralph-wiggum**

$ ⚖ 다층 검증 ≠ 상시 다중 검증

검증 강도 결정표: 모델 × 작업 시간 × 실패 비용

공식 문서는 검증 게이트를 권고하면서, 동시에 과잉 검증을 경고한다, 둘 다 사실이다

✓ 공식 권고: 게이트를 두라

- 프롬프트 안의 즉시 검증 → `/goal` 조건
  → Stop hook → second opinion
- "작업한 에이전트가 자기 결과를 채점하지 않게 하라"

Codex Best Practices

⚠ 공식 경고: 전부 상시로 켜지 마라

- 불필요한 반복 탐색·과도한 tool-calling은
  피하라, 시키면 없는 문제도 찾아낸다
- 모델이 이미 하는 검증의 재지시 = 레거시 구조물

GPT-5 프롬프팅 가이드 취지 요약

축	검증을 두껍게	검증을 얇게
모델	구형 · 소형 모델	최신 상위 모델: 시키지 않아도 스스로 검증
작업 시간	장시간 미감독 실행 · 세션을 넘는 작업	한 세션에서 끝나고 눈으로 보는 작업
실패 비용	운영 반영 · 데이터 변경 · 보안 · 결제	되돌리기 쉬운 로컬 변경

◎ 완전히 끄는 선택지는 없다: 3축은 검증의 "두께"를 정하지, "유무"를 정하지 않는다.

$ 판정 질문

한 달에 한 번, 하네스 감사

- **이거, 아직도 필요한가?**
  - 이 규칙은 최근에도 실제 실수를 막았나?
  - 이 MCP는 자주 쓰나, 멋있어 보여서 켜둔 건가?
  - 이 Skill은 지금도 필요한가, 제품 기본 기능으로 대체됐나?
  - 이 hook은 안전하게 만드나, 매번 불필요하게 멈추나?
  - 이 문서는 매 세션 읽어야 하나, 필요할 때만 읽어도 되나?
  - 내 하네스, Codex 릴리스 노트와 대조했나, 제품 기능과 중복은 없나?
- **→ 좋은 하네스는 쌓이는 게 아니라 갱신된다.**

↓ 이 질문 목록 자체가 판정 체크리스트입니다, 사람이 매달 손으로 묻던 것을 그대로 스킬 프롬프트로 옮기면, harness-legacy-scan 같은 도구가 됩니다.

다음 파트: 복기 + 마무리

Part 9. 복기 + 마무리

$ 오늘 쌓은 것 · 줄인 것 · 옮긴 것

하네스 누적 과정 복기

Part	겪은 마찰 → 쌓은 것	실습
1 이해	하네스 0의 파워와 통제 불가, 순서·레이어 **티저**	1 Lumo
2 Rule + Skill	같은 설명·같은 절차의 반복 → **AGENTS.md · 스킬**	2 합류 (정비 커밋)
3 Skill 고급	쌓은 하네스 **4종이 동시에** 발동하는 장면	3 영속화
4 Hook + Sub-agent	권고 위반 · 자기 검증 오염 → **강제 · 격리**	4 안전장치 (정비 커밋)
5 스케일	혼자로는 못 본다 · 느리다 → **MCP · Worktree · 외부 메모리**	5 구현 · 6 리서치 · 7 검증
6 운영·비용	캐시 · 컨텍스트 · 권한 · 격리	-
7 다이어트	쌓은 것이 낡는다 → **판정하고 줄인다**	- (이론)
9 복기	지금: **두 장의 설계도로 접는다**	-

Part 1 티저의 회수 지점: 반복 설명 → Part 2 **Rule** / 반복 절차 → Part 2~3 **Skill** / **"강제"**는 Part 4의 **Permission · Hook**이었습니다. 나머지(격리 · 능력 결손 · 스케일)는 Part 4~5에서 실제로 겪은 뒤 이름을 붙였습니다.

실습 1 Lumo: 하네스 0의 기준선. 같은 모델, 같은 사람이었습니다. 그때와 지금, 달라진 것은 모델이 아니라 주변 구조입니다.

$ 첫 번째 축: 시간(무엇을 언제)

무엇을 언제 붙이나: 문제와 반복이 기준이다

다음 하네스는 **마찰 빈도 × 도입 비용 × 실패 비용**, 이 세 가지로 정한다

#	겪는 마찰	하네스 응답	이 순서인 이유
0	상시: 마찰을 기록한다	**외부 메모리** (마찰 로그)	기록이 없으면 판단은 느낌이 된다, 1번 이전에 켠다
1	같은 설명을 반복한다	**Rule** (AGENTS.md)	빈도 최고 + 도입 비용 최저
2	같은 절차를 반복시킨다	**Skill**	빈도: 설명이 아니라 손이 반복될 때
3	권고를 어긴다 · 어기면 치명적	**Permission · Hook**	교수법: **권고**를 겪은 뒤 **강제**를 안다 (실패 비용 크면 1번)
4	자기 검증 오염 · 탐색 오염	**Sub-agent**	격리: 검증 분리 + 탐색 격리
5	혼자로는 못 본다 (브라우저 · DB · 외부)	**MCP**	능력 결손: 실무에선 더 일찍 붙는 경우가 많다
6	혼자로는 느리다 (병렬 · 대규모)	**Worktree·Workflow**	스케일 패밀리: 셋의 마찰·비용은 서로 다르다

[상시 주기] 줄인다: 2번부터 병행되는 주기

※ 공식 문서에 순서 권고는 **없습니다**: 3축 공식은 **이 강의의 자체 프레임**입니다 (원리는 YAGNI + Rule of Three).

$ 두 번째 축: 구조(무엇이 어떤 역할)

레이어 완성 지도: 오늘 4개를 전부 채웠다

레이어	역할	부품 · 실물	발동 시점	오늘 실습
착수 Kickoff	목표·완료·검증 조건, 메모리·실패 정책 설계	Deep Interview · plan · 명세	**작업 시작 시**	5
기록 Memory	반복 실수·지식·진행을 구조에 남김	Rule · 외부 메모리 · 마찰 로그	상시 누적	2
절차 Procedure	반복 절차 패키징 + 외부 능력 연결	Skill · Workflow · MCP	호출 시	2·3·5·7
검증 Verification	산출물 확인: **결정표로 선택**	TDD · Hook · Sub-agent · 브라우저 · /review	변경 시	3·4·7

※ **Worktree는 이 지도에 없습니다, 의도된 공백.** 4분류는 강의 자체 프레임, 부품은 전부 공식 문서에 있습니다.

> 순서표는 **기록 → 절차 → 검증**을 채우는 순서, 착수는 순서가 아니라 **계기**입니다. Worktree는 이 전부를 여러 벌 돌리는 실행 환경입니다.

◎ **구축 순서(시간 축)와 레이어 지도(구조 축), 이 두 장이 여러분 프로젝트의 하네스 설계도입니다.**

$ ✓ 레이어 지도의 '검증' 행을 펼치면

검증 레이어 안의 6가지 수단

레이어는 4개, 그중 **검증 레이어 하나**에 오늘 체험한 수단이 6가지 있었습니다.

1. Rule

권고: AI가 참고하지만 무시할 수 있다

2. TDD · 테스트

자동: 코드 정확성을 기계가 판정

3. Permission · Hook

강제: 위반 시 차단, 결정적

4. Sub-agent

독립: 작업자 ≠ 검증자

5. 브라우저 검증 + MCP

실제 화면: 실습 7에서 직접 붙였다

6. /review

리뷰: 다면 지적, 사람이 선별

⚖ **이 6가지는 전부 상시가 아닙니다, 결정표로 고릅니다. 여섯을 다 켜면 검증이 아니라 비용이 됩니다. 변경의 성격(되돌릴 수 있나 · 화면에 보이나 · 실패하면 얼마나 위험한가)이 어느 수단을 켤지 정합니다.**

※ 무엇을 켤지의 판단 기준은 Part 7 **결정표**에서 다뤘습니다, 다층 검증이 핵심이지, 전부 켜는 것이 핵심이 아닙니다.

내일 당장 적용할 2가지

1 ✎ 마찰 로그를 켠다, 3일만

- 0번 상시 레이어. 무엇이 / 몇 번째 / 시간을 얼마나 / 실패하면 얼마나 위험한가
- 3일이면 1번으로 뭘 할지가 **추측이 아니라 기록**으로 나온다

2 📄 순서표에서 딱 한 칸만 실행한다

- 기록이 **설명 반복**을 지목하면 AGENTS.md 하나, 마스터 프롬프트(Gist)로 자동 생성
- **절차 반복**을 지목하면 스킬 하나 · **치명적 위반**이면 Hook을 1번으로 앞당긴다

🔨 그리고 90일 뒤 한 번, 오늘 만든 하네스를 다시 판정합니다. 스캔·다이어트 스킬을 들고 가십시오. 쌓기만 하는 하네스는 3개월이면 obsolete scaffolding이 됩니다.

핵심 인사이트

"엔지니어링 엄밀함은 사라지지 않는다"

"더 높은 추상화 수준으로 이동할 뿐이다"

- ▸ 프롬프트 텍스트 → 컨텍스트 윈도우 설계 → **에이전트 루프 전체 설계**
- ▸ 오늘 구축한 하네스가 바로 그 엄밀함입니다, 모델이 아니라 **주변 구조**가 결과를 갈랐습니다
- ▸ 완벽한 하네스를 한 번에 만들려 하지 마십시오. **문제나 반복이 생긴 것 하나**부터 시작하십시오
- ▸ 하네스 생애주기 = **쌓기 → 줄이기**, 그리고 다시 쌓기 . 한 번 하고 끝나는 계단이 아니라 계속 도는 주기입니다
- ▸ 손에 남는 것은 도구 목록이 아니라 **두 장의 설계도**입니다, 구축 순서(언제)와 레이어 지도(무슨 역할)

Q&A

"하네스 엔지니어링 with Codex"

▸ 질문이 있으시면 편하게 물어보세요