AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

하네스 엔지니어링
hooks · permissions · skill

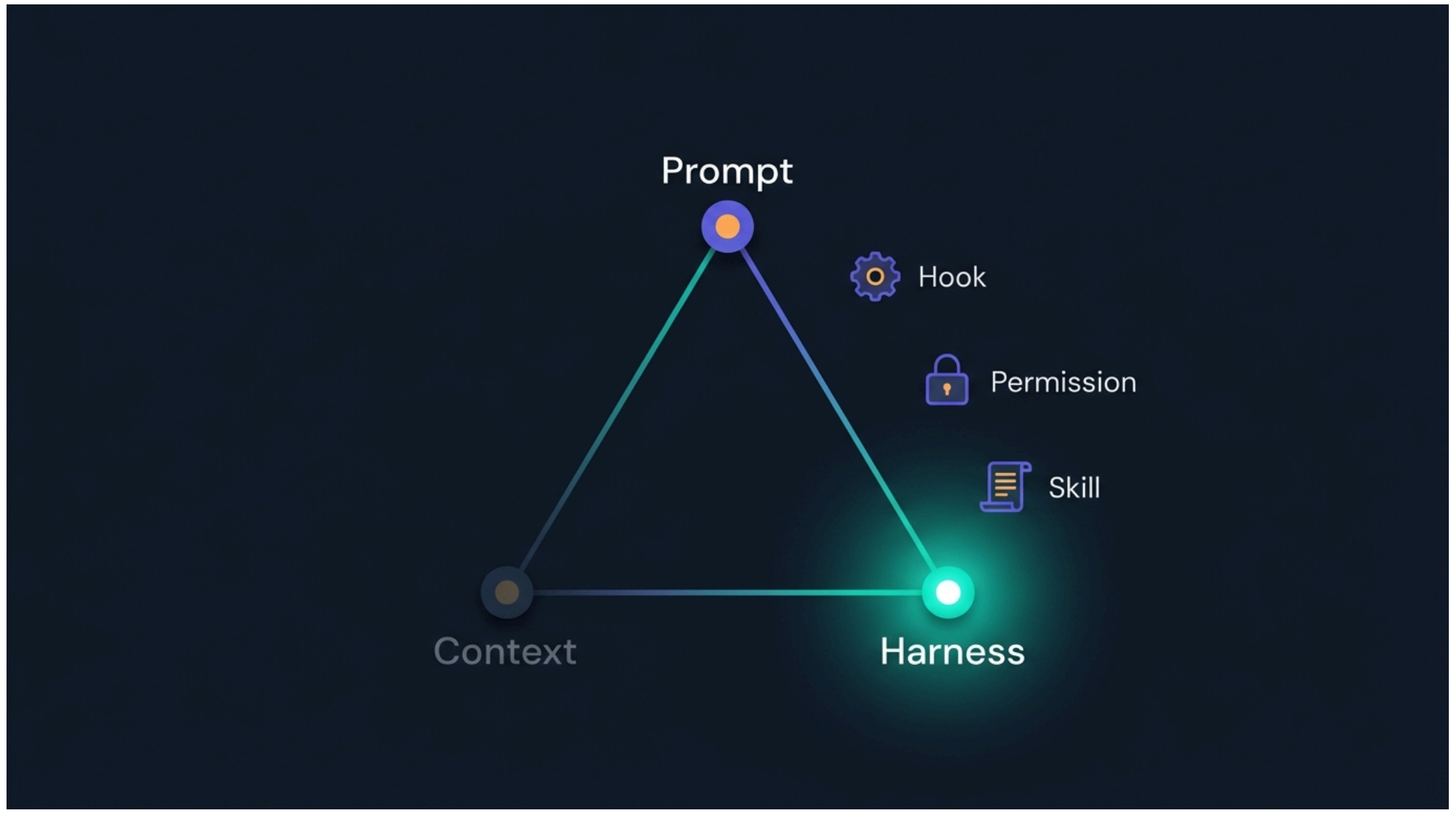
Session 4 · 하네스 엔지니어링 — 규약으로 강제하기

hooks 이벤트 6종 · permissions allow/deny · Statusline · Custom Skill · Slash Command · hook·slash·skill 각 1개 직접 제작

강사 박수현 · 🚀 젠아이랩스(GenAI Labs)

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

출처: Hooks | Permissions | Skill | Slash Command 문서



Session 4 · 하네스 엔지니어링 — 규약으로 강제하기

출처: Hooks | Permissions | Skill | Slash Command 문서

Claude Code 5계층 세팅 — 우선순위 순

1️⃣ CLAUDE.md

컨텍스트 · 규약 문서

- 3계층 (개인 · 프로젝트 · 서브)
- 지시 · 컨벤션 · 용어집
- **가장 넓은 신호**

🎯 **컨텍스트 축**

2️⃣ MCP

외부 도구 · 데이터 연결

- `.mcp.json` 서버 등록
- GitHub · Slack · DB · 사내 API
- **도구 카탈로그 확장**

🎯 **하네스 축 · 도구 연결**

3️⃣ Skills

자율 발동 지시 묶음

- `skills/<name>/SKILL.md`
- description 매칭 자동 로드
- **반복 노하우 재사용**

🎯 **하네스 축 · 오늘**

4️⃣ Hooks

결정론적 이벤트 훅

- `settings.json` 의 `hooks` 키
- Pre / Post / Session / Stop 등 6종
- **규칙 100% 강제 · 안전선**

🎯 **하네스 축 · 오늘**

5️⃣ Subagents

전문 페르소나 위임

- `agents/<name>.md`
- 검토 · 리뷰 · 검수 전담 인격
- **컨텍스트 격리 · 병렬화**

🎯 **컨텍스트 축**

확장 5계층 — *컨텍스트를 언제·얼마나 차지하는가*

같은 5계층을 **컨텍스트 점유** 축으로 다시 세우면 "무엇을 넉넉히 두고 무엇을 아껴야 하는지" 가 보인다.

계층	로딩 방식	컨텍스트 점유
CLAUDE.md + `@import`	세션 시작 시 **전량 상주** (eager)	항상 · 얇게 유지가 곧 절약
Skill	대기 땐 name·description만, 발동 시 본문 로드	대기 **~30–50토큰** · 조건부
MCP 서버	연결 내내 tool 스키마 **상주**	**수천–수만 토큰** · 상시·외부
Hook	셸에서 결정론적 실행	**컨텍스트 밖** · 0 (결과 주입 시만 소량)
Subagent	별도 컨텍스트 창에서 실행	**격리** · 요약만 부모로

📌 **Skill 이냐 MCP 냐 — 판단과 비용. 지식·절차** 는 Skill 로(대기 비용 거의 0 → **많이 둬도 싸다**), **외부 시스템 연결** 은 MCP 로(서버마다 상시 수천~수만 토큰 → **필요한 것만 적게**). 규칙 한 줄 — *"Skill 은 넉넉히, MCP 는 최소한."*

PART 1

하네스 엔지니어링이란 — *세 번째 축*

> 하네스의 정의 · Claude Code 하네스 5개 개념 · 개인·프로젝트·로컬 3층 스코프.

출처: Hooks | Permissions | Skill | Slash Command 문서

세 축의 지도 — *프롬프트 · 컨텍스트 · 하네스*

AI 코딩을 다루는 기술은 세 축으로 정리된다. 축마다 초점이 다르고, 뒤의 축이 앞의 축을 감싼다.

🎯 프롬프트 엔지니어링

무엇을 말할 것인가

- 핵심 질문 — *요청을 어떻게 정확히 표현하나*
- 제로샷 · 퓨샷 · 역할 부여 · CoT
- 배경 — 모델은 강한데 지시가 약했다

📚 컨텍스트 엔지니어링

무엇을 알려줄 것인가

- 핵심 질문 — *판단에 필요한 정보를 어떻게 공급하나*
- RAG · 메모리 · 롱 컨텍스트 · 도구 결과
- 배경 — 모델이 모르는 것은 지시로 못 채운다

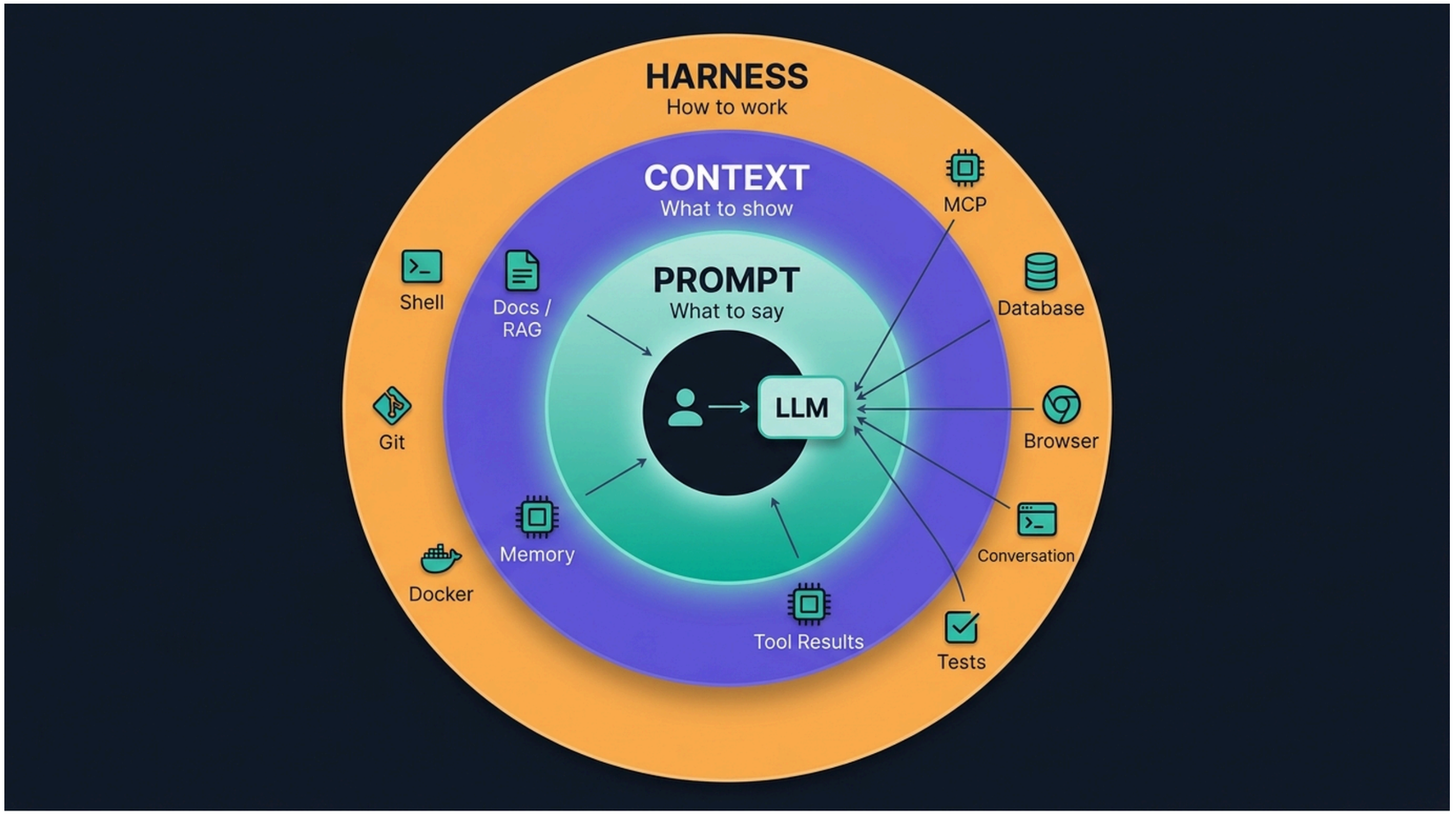
⚙️ 하네스 엔지니어링

어떻게 일하게 할 것인가

- 핵심 질문 — *어떤 절차·도구·규약 위에서 실행되나*
- 에이전트 루프 · 툴 유즈 · MCP · 훅 · 권한
- 배경 — 알아도 실행 환경이 없으면 못 끝낸다

🟢 **오늘 다루는 축**

📌 **대체가 아니라 포함이다. 프롬프트 ⊂ 컨텍스트 ⊂ 하네스** — 하네스가 실행 환경을 세우고, 그 안에서 컨텍스트가 공급되고, 그 위에서 프롬프트가 실행된다. 새 축이 나왔다고 앞 축이 낡는 게 아니라 **다루는 범위가 커진 것**이다. 등장 순서는 대략 **2022~ 프롬프트 → 2023~2024 컨텍스트 → 2025~ 하네스** 로 흘렀다 (경계는 흐릿하다).



오늘 서 있는 자리는 가장 바깥 고리 — 셸 · Git · Docker · 테스트 · 브라우저 · DB · MCP 가 걸리는 실행 환경

하네스란 무엇인가 — *묶어서 함께 일하게 하는 장치*

harness 는 원래 **여러 장비를 하나로 묶어 함께 동작하게 하는 장치** 를 뜻한다. 말 마구, 등반·작업용 안전벨트 하네스, 소프트웨어의 테스트 하네스 — 전부 같은 발상이다. AI 코딩의 하네스도 두 층위로 나눠 보면 정확해진다.

🌐 실행 하네스 — 넓은 뜻

LLM 을 중심으로 도구를 엮어 만든 하나의 작업 시스템

- 셸 · Git · Docker · 테스트 러너 · 브라우저 · DB · MCP 서버
- 도구를 잡고 → 결과를 읽고 → 다시 판단하는 **전체 루프**
- **Claude Code 라는 제품 자체가 이 하네스다** — 모델 하나만으로는 파일 한 줄도 못 바꾼다

🔒 규약 하네스 — 오늘 직접 만드는 것

그 루프를 강제·자동화하는 얇은 계층

- `hooks` · `permissions` · `skill` · `slash` · `statusline`
- 커밋 전 lint · 파일 수정 후 test · `.env` 접근 차단
- 루프를 새로 만드는 게 아니라 **이미 도는 루프에 규칙을 건다**

📌 **넓은 뜻을 먼저 잡아야 좁은 뜻이 산다.** 실행 하네스가 "무엇을 할 수 있는가" 의 판이라면, 규약 하네스는 그 판 위에 **"무엇을 반드시 하고 · 무엇은 못 하게 할지"** 를 새기는 일이다. 오늘 손대는 5개 개념은 전부 후자에 속한다.

`"Flask 할 일 앱 만들어줘"` — *한 줄 뒤에 벌어지는 일*

이 한 줄이 들어가면 실제로 일어나는 일을 펼쳐 보면, 사람이 준 몫과 하네스가 한 몫이 갈린다.

🧱 준비

- 디렉토리 생성
- `requirements.txt` 작성
- `pip install` 로 의존성 설치

⌨️ 구현

- `app.py` 작성
- `git init` 으로 저장소 초기화
- 서버 실행

🔁 검증 · 기록

- 브라우저로 화면 확인
- 오류가 나면 수정 후 재실행
- `git commit` 으로 마감

📌 **사람이 준 건 프롬프트 한 줄이고, 나머지 전부가 하네스다.** 셸·파일시스템·pip·Git·브라우저를 모델이 도구로 잡고 도는 이 루프가 실행 하네스이며, 그래서 손잡이는 둘로 좁혀진다 — **훅** (언제 무엇을 강제할지) 과 **권한** (무엇을 못 하게 할지). 오늘 파트 2·3 이 정확히 이 두 손잡이다.

출처: Hooks | Permissions | Skill | Slash Command 문서

Claude Code 하네스 5개 개념 — *오늘 목차*

개념	위치	하는 일
Hooks	`settings.json`	특정 이벤트에 셸 명령 트리거
Permissions	`settings.json`	도구·명령 allow / deny 리스트
Statusline	`settings.json` + 스크립트	하단 상태표시줄 커스터마이즈
Custom Skill	`~/.claude/skills/<slug>/SKILL.md`	자율 발동 재사용 지시 묶음
Slash Command	`~/.claude/commands/<name>.md`	`/` 로 호출하는 개인 명령

🎯 Slash Command

- 사용자가 `/name` 을 **쳐야 발동**
- **결정론적** — 매번 같은 결과
- 예측 가능성 우선

🤖 Custom Skill

- Claude 가 `description` 매칭해 **자율 발동**
- **LLM 판단 기반** — 문맥 맞으면 발동, 애매하면 건너뜀
- 예 · `"한글 주석 정리해줘"` → `korean-comment-cleaner` 매칭 성공
- 자동성 우선

📌 **같은 마크다운·같은 슬래시 호출 형태 — 자율 발동 여부만 다르다.**

출처: Hooks · Permissions · Skill · Slash Command 문서

스코프 5축 지도 — *파일별 위치 · 좁은 스코프가 이긴다*

파일	개인	프로젝트	로컬	무엇
CLAUDE.md	✅	✅	—	프롬프트 · 컨텍스트
settings.json	✅	✅	✅	hooks · permissions · statusline
.mcp.json	—	✅	✅	MCP 서버 등록
skills/	✅	✅	—	자율 발동 지시 묶음
commands/	✅	✅	—	slash 명령
agents/	✅	✅	—	서브에이전트 정의
hooks/	✅	✅	—	훅 스크립트 소스

🏆 좁은 스코프가 이긴다

로컬 > 프로젝트 > 개인. 같은 키가 여러 층에 있으면 **가장 좁은 스코프** 가 이긴다. - hooks · permissions · statusline — 좁은 스코프가 **완전히 대체** - skills · commands · agents — 같은 이름이면 좁은 스코프 우선 - .mcp.json — 로컬 우선 · 같은 이름 서버는 로컬이 이긴다

📐 어느 층에 둘 것인가

- **팀 공통** → 프로젝트 · **나만의 편의** → 개인 · **실험·시크릿** → 로컬
- `.gitignore` 에 `.claude/settings.local.json` 필수
- **CLAUDE.md 만 예외 — 누적된다** (덮어쓰기 아님)

PART 2

Hooks — *세션 이벤트에 셸을 매다는 장치*

> 훅의 정의 · 발동 카덴스 · 실습에서 다루는 이벤트 6종 (공식 30+ 중 실무 핵심) · JSON 구조 · 실전 예 2개 (lint · 커밋 게이트).

출처: Hooks | Permissions | Skill | Slash Command 문서

훅이란 — *세션 이벤트에 셸을 매다*

🪝 훅은 세 가지 중 하나로 동작한다

1. **실행 허용** — 통과 (기본)
2. **차단** — `PreToolUse` 훅이 **exit code 2** 반환 시 도구 호출 차단, Claude에 이유 전달
3. **컨텍스트 삽입** — stdout JSON 으로 후속 지시

⏱️ 발동 시점 — 6 이벤트

- `SessionStart` — 세션 시작 시 · `세션당 1회`
- `UserPromptSubmit` — 프롬프트 제출 후 · `턴당 1회`
- `PreToolUse` — 도구 실행 전 · `도구마다`
- `PostToolUse` — 도구 실행 후 · `도구마다`
- `Notification` — 알림 시 · `이벤트마다`
- `Stop` — 응답 종료 시 · `턴당 1회`

📌 **한 문장 정의.** Claude 가 파일을 수정하기 직전, 셸을 돌린 직후, 세션이 시작될 때 — **특정 시점에 내가 정한 스크립트가 자동 실행되는** 것이 훅이다.

프롬프트 vs 훅 — *확률적 규칙 vs 결정론적 규칙*

🎲 프롬프트 (CLAUDE.md · 시스템 지시)

확률적으로 발동한다.

- 규칙은 컨텍스트에 실려 **LLM 이 참고할 뿐**
- 세션이 길어지면 규칙이 **주의에서 밀려난다**
- 우선순위 다툼 · 요약 압축 · 노이즈에 취약
- "잘 지키다가 30 턴째 슬쩍 무시" 가 실제로 일어난다

⚙️ Hook (`settings.json`)

결정론적으로 발동한다.

- **셸 명령** — OS 가 실행한다
- 이벤트가 오면 **100% 발동**, 모델 상태 무관
- `exit 2` 는 예외 없이 도구 호출을 **차단**
- LLM 의 "깜빡함" 에 의존하지 않는 안전선

📌 **한 문장 슬로건. LLM 워크플로우에서 규칙을 100% 신뢰성으로 강제하는 유일한 방법은 훅이다.** "반드시 lint 통과 후 커밋" 같은 규칙을 CLAUDE.md 에만 적어 두면 뚫린다 — `PreToolUse` 로 매달아야 뚫리지 않는다.

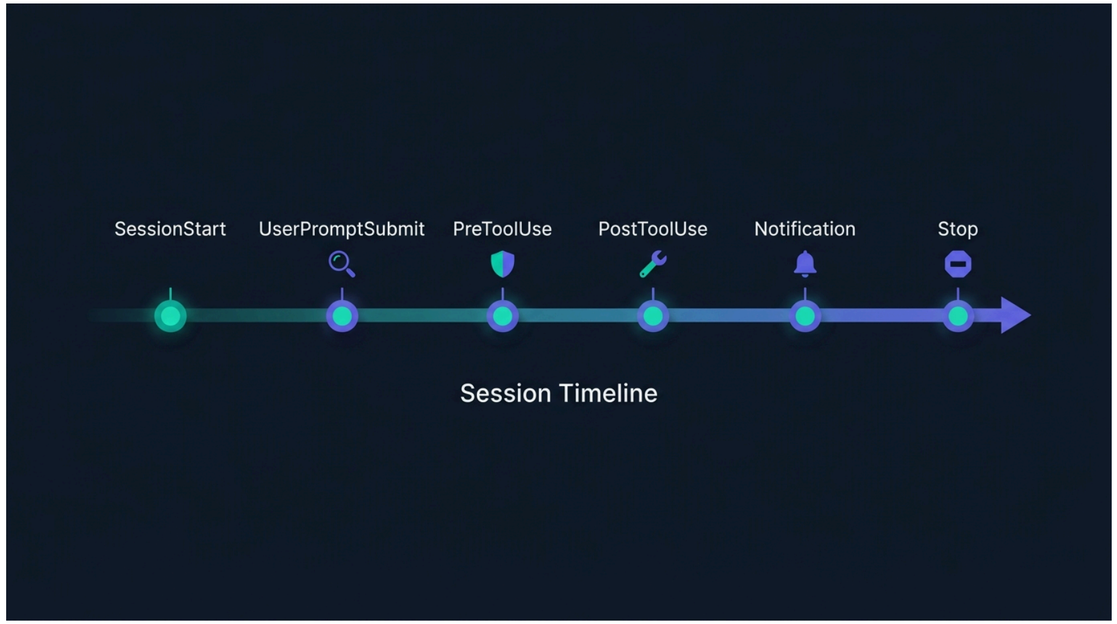
⚠️ **오해 방지.** 프롬프트가 열등하다는 뜻이 아니다. **의도 · 설계는 프롬프트**, **강제 · 게이트는 훅** — 역할이 다르다. "방향 제시" 와 "가드레일" 의 조합이 하네스의 본질.

실습에서 다루는 훅 이벤트 6종 — *실무 핵심 6개 + 소수 확장*

이벤트	발동 시점	대표 용도
`SessionStart`	세션 시작 시 (재개 포함)	환경 진단 · 컨텍스트 파일 로드
`UserPromptSubmit`	프롬프트 제출 후	프롬프트 검열 · 추가 컨텍스트 주입
`PreToolUse`	도구 실행 전	`.env` 차단 · 위험 명령 검문 · dry-run
`PostToolUse`	도구 실행 후	lint · format · 테스트 · 로깅
`Notification`	알림 시	데스크톱·슬랙 알림
`Stop`	응답 종료 시	세션 요약 · 정리

📌 **범위 안내.** 이 세션은 **실무에서 가장 자주 쓰는 6개**에 집중한다. 나머지는 `SessionEnd` · `PreCompact` · `SubagentStop` · `PermissionRequest` 등 **소수 확장 이벤트**로, 필요 시 공식 문서에서 같은 JSON 구조로 확장한다.

훅 이벤트 6종 — *세션 타임라인 위 어디에 붙는가*



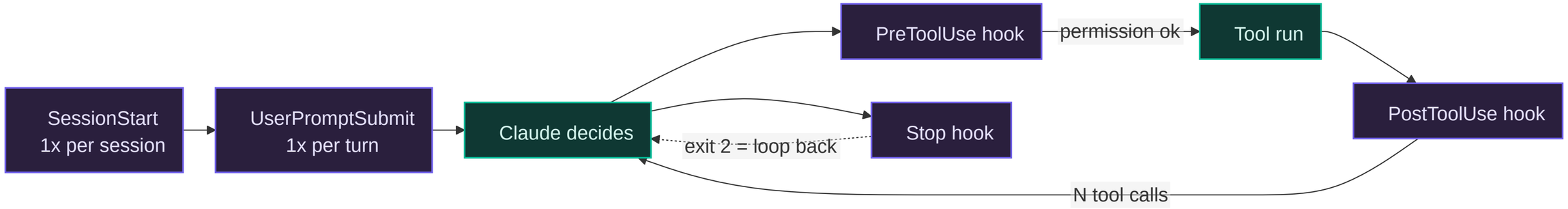
세션 한 사이클 · 왼쪽 → 오른쪽으로 흐른다

🧭 그림 읽기

- **왼쪽 → 오른쪽** = 세션 진행 시간
- **여섯 점** = 훅이 걸리는 여섯 자리
- **점 아래 아이콘** = 그 시점 대표 행동 (검열 · 차단 · 청소 · 알림 · 정지)

바로 앞 표와 짝지어 본다. 그림은 **언제**, 표는 **무엇**. 둘을 겹쳐 읽어야 여섯 이벤트의 자리를 기억할 수 있다. 이후 실습에서 `PreToolUse` 하나를 커밋 게이트에 매단다.

매 턴 실행 loop — *훅이 걸리는 자리를 한 장에*



📌 **한 턴의 형태.** `SessionStart` 는 세션당 1번, 그 뒤로는 사용자 프롬프트 하나마다 `UserPromptSubmit → (PreToolUse → 권한 → 도구 실행 → PostToolUse) × N → Stop` 이 반복된다. 훅은 이 파이프라인의 **다섯 자리** 에서만 걸린다 — 자리를 외우면 훅 설계가 쉬워진다.

⚠️ `Stop` **훅에서** `exit 2` 를 던지면 Claude 는 세션을 끝내지 않고 **한 턴 더 도는** 방식으로 loop-back 이 가능하다 — 자동 검토 · 자기 반박 루프의 출발점.

핵심 대비 — `PreToolUse` vs `PostToolUse`

🛑 `PreToolUse`

막을 수 있다.

- 도구가 실행되기 **전**에 발동
- **exit 2 로 차단** — Claude 에 이유 전달
- 위험 명령 · 시크릿 접근 방지
- **예시.** `git commit` 직전 pytest, `.env` 접근 사전 차단

✅ `PostToolUse`

반응만 한다.

- 도구가 실행된 **직후** 발동
- 이미 실행됐으므로 **후처리** 위주
- lint · format · 테스트 · 로깅
- **예시.** 파일 편집 후 `ruff format`, 저장 후 `prettier`

⚠️ **막을 목적이면 `PreToolUse`, 청소할 목적이면 `PostToolUse`.** `PostToolUse` 에 exit 2 를 넣어도 도구는 이미 실행된 뒤다.

훅 JSON 구조 — `settings.json` 아래 `hooks` 키

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write|MultiEdit",
        "hooks": [
          {
            "type": "command",
            "command": "npx prettier --write \"$CLAUDE_TOOL_INPUT_file_path\""
          }
        ]
      }
    ]
```

🔑 필드 3개

- `matcher` — 정규표현식. 위 예는 `Edit` · `Write` · `MultiEdit` 에만 발동 (빈 문자열이면 전체)
- `hooks[].type` — 실전은 `command` (셸) 하나로 해결
- `hooks[].command` — 셸 명령. 이벤트 JSON 이 stdin, 도구 인자가 `$CLAUDE_TOOL_INPUT_*`

📥 stdin JSON · exit code

- **stdin** — `{ "tool_name", "tool_input", ... }`, `jq` 파싱이 표준
- **exit 0** — 성공 (stdout JSON 이면 후속 제어)
- **exit 2** — 차단, Claude 에 이유 전달
- **그 외** — 오류, 로그만 남기고 진행

실전 훅 예 ① — *파일 수정 후 자동 lint*

목표. Python 파일 수정 시 즉시 `ruff` lint · format. `~/.claude/settings.json` (개인 스코프).

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write|MultiEdit",
        "hooks": [
          {
            "type": "command",
            "command": "path=$(jq -r '.tool_input.file_path' <<< \"$CLAUDE_HOOK_INPUT\"); if [[ \"$path\" == *.py ]]; then ruff check --fix \"$path\" && ruff format \"$path\"
          }
        ]
      }
    ]
  }
}
```

📖 해설

- **모든 편집 후** 발동, `jq` 로 경로를 뽑아 `.py` **일 때만** `ruff` 실행
- 결과를 Claude 에 참고시키려면 `echo '{"additionalContext":"..."}'` 로 stdout JSON

⚠️ 주의

- `--fix` 로 코드가 바뀌면 diff 가 어긋난다 — **"포맷만 자동, 로직은 사람 확인"**
- matcher 를 좁히지 않으면 무거운 훅이 매 편집마다 돌아 느려진다

실전 훅 예 ② — *커밋 시 테스트 자동 실행 (`PreToolUse`)*

목표. `git commit` 직전 pytest 실행, 실패 시 커밋 차단. `<repo>/.claude/settings.json` (프로젝트 · 팀 공유).

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "cmd=$(jq -r '.tool_input.command' <<< \"$CLAUDE_HOOK_INPUT\"); if [[ \"$cmd\" == *'git commit'* ]]; then pytest -x --tb=short || exit 2; fi"
          }
        ]
      }
    ]
  }
```

📖 해설

- `PreToolUse` — **커밋 실행 전** 발동
- Bash 명령을 `jq` 로 뽑아 `git commit` 포함 확인
- 실패 시 **exit 2 로 차단**, Claude 가 이유 수신

🤝 팀 하네스로서의 의미

- 프로젝트에 커밋 → **팀 누구든 같은 게이트 통과**
- 사람마다 잊는 pre-commit 을 **세션 훅**으로 대체
- git 훅과 병행 — Claude 훅은 **AI 세션 안에서만** 발동

Hook 실용 예 ③ — *위험 명령 사전 차단 (`PreToolUse` Bash)*

목표. AWS 리소스 삭제 · 프로덕션 DB 파괴 명령을 Claude 가 실행하기 직전에 차단. Bash 도구에 훅을 걸어 외부 파이썬 검문 스크립트를 태운다.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "~/.claude/hooks/danger_guard.py" }
        ]
      }
    ]
  }
}
```

📌 `command` 에 스크립트 경로만 넣으면, 그 스크립트가 이벤트 JSON 을 stdin 으로 받아 판단한다. 검문 로직은 다음 슬라이드의 `danger_guard.py`.

Hook 실용 예 ③ — `~/.claude/hooks/danger_guard.py`

```
#!/usr/bin/env python3
import json, sys, re

cmd = json.load(sys.stdin).get("tool_input", {}).get("command", "")
DANGER = [
    r"\brm\s+-rf\s+/",
    r"\bterminate-instances\b",
    r"\bDROP\s+(TABLE|DATABASE)\b",
    r"\bkubectl\s+delete\s+ns\b",
]
for pat in DANGER:
    if re.search(pat, cmd, re.I):
        print(f"차단: 위험 명령 감지 → {pat}", file=sys.stderr)
        sys.exit(2)
sys.exit(0)
```

⚠️ **exit 2 + stderr 는 Claude 에 이유가 전달되는 유일한 통로.** 사유를 명확히 남기면 Claude 는 "왜 막혔는지" 문맥을 받아 대안을 제시한다. permissions.deny 는 「규칙」, 이 훅은 「런타임 검문」 — **둘을 겹쳐** 잡는 것이 실무 표준.

Hook 실용 예 ④ — *커밋 전 스테이지 쓰레기 검사*

목표. `git commit` 을 감지하면 스테이지된 diff 를 훑어 API 키 · "TODO remove" · 디버그 print 등 커밋되면 안 될 것들을 잡아낸다.

`~/.claude/hooks/commit_scan.py`.

```
#!/usr/bin/env python3
import json, sys, subprocess, re

cmd = json.load(sys.stdin).get("tool_input", {}).get("command", "")
if "git commit" not in cmd:
    sys.exit(0)
diff = subprocess.run(["git", "diff", "--cached"], capture_output=True, text=True).stdout
PATTERNS = {
    "OpenAI key":  r"sk-[A-Za-z0-9]{20,}",
    "AWS key":     r"AKIA[0-9A-Z]{16}",
    "TODO remove": r"TODO.*remove",
    "print debug": r"^\+\s*print\(.+debug",
}
hits = [n for n, p in PATTERNS.items() if re.search(p, diff, re.M)]
if hits:
    print(f"차단: 스테이지에 {', '.join(hits)} 흔적", file=sys.stderr)
    sys.exit(2)
sys.exit(0)
```

📌 **git pre-commit 훅과의 차이.** git pre-commit 은 사람이 셸에서 `git commit` 을 칠 때만 발동. 이 훅은 **Claude Code 세션 안** 에서 Claude 가 커밋할 때 발동한다. **둘을 병행 배치** 가 표준 — 사람 커밋 · AI 커밋 모두 같은 게이트를 통과.

Hook 실용 예 ⑤ — *긴 작업 완료 알림 (`Stop`)*

목표. 응답을 마치는 순간 데스크톱 알림. 긴 작업을 맡겨 두고 다른 일을 하다 알림을 받는다 — 체감 효과가 가장 큰 훅.

```
{
  "hooks": {
    "Stop": [
      { "hooks": [ { "type": "command", "command": "~/.claude/hooks/notify.sh" } ] }
    ]
  }
}
```

`~/.claude/hooks/notify.sh` — **OS 분기.**

```
#!/usr/bin/env bash
title="Claude Code"; msg="응답 완료"
case "$(uname -s)" in
  Darwin) osascript -e "display notification \"$msg\" with title \"$title\"" ;;
  Linux)  notify-send "$title" "$msg" ;;
  MINGW*|MSYS*|CYGWIN*)
    powershell -Command "[reflection.assembly]::LoadWithPartialName('System.Windows.Forms'); \
      [System.Windows.Forms.MessageBox]::Show('$msg','$title')" ;;
esac
```

📌 `Stop` 훅은 matcher 가 필요 없다 — 응답이 끝날 때마다 무조건 발동. `uname` 으로 분기해 macOS·Linux·Windows 어디서든 같은 알림.

Hook 실용 예 ⑤ — *완료 알림 · 왜 유용한가 · 확장*

📢 왜 유용한가

- **blocking 없이도** 완료 시점을 놓치지 않는다
- 다른 리포 · 다른 창을 보고 있어도 알림으로 완료를 파악한다
- **매 응답마다** 조용히 발동 — UX 가 크게 바뀐다

⚙️ 확장 아이디어

- Slack 웹훅 · Discord 웹훅 — 원격 근무 시 자기에게 DM
- `say "완료"` (macOS) · `spd-say` (Linux) — 음성 알림
- 로그 파일에 완료 시각 기록 → 세션 통계 축적

PART 3

Permissions — *allow · deny 리스트로 사전 차단*

> 훅이 "이벤트 반응" 이라면 permissions 는 "규칙 리스트로 사전 차단" · 세밀 문법 · 팀 규약 3원칙 · 프로젝트 초안 템플릿.

Permissions 세 축 — `allow` · `deny` · `ask`

```
{
  "permissions": {
    "allow": [
      "Bash(git status)",
      "Bash(git log:*)",
      "Bash(pytest:*)",
      "Read(./src/**)",
      "WebFetch(domain:docs.python.org)"
    ],
    "ask": [
      "Bash(git push:*)",
      "Bash(gh pr create:*)"
    ],
    "deny": [
      "Bash(rm -rf:*)",
      "Bash(sudo:*)",
      "Read(./.env)",
      "Read(./.env.*)",
      "WebFetch(domain:*.internal.company.com)"
    ]
  }
}
```

📌 **세 리스트 · 한 파일.** `allow` 는 자동 통과, `ask` 는 매번 확인, `deny` 는 무조건 차단. 평가 순서는 **deny → ask → allow** — 문법·원칙은 다음 슬라이드.

Permissions 문법·원칙 — `Tool(spec)` · deny → ask → allow

🔤 규칙 문법

- `Tool(spec)` — 툴 이름 + 세부 스펙. Bash — 명령 패턴, Read — 경로 glob, WebFetch — 도메인
- **인자 와일드카드 두 형태** — 스페이스 `Bash(git log *)` (승인 다이얼로그 기본형) · 끝 `:*` `Bash(git log:*)` (`:*` 는 **패턴 끝에서만** 유효 · 중간은 리터럴)
- ✓ `Bash(git log:*)` — `git log --oneline -5` 매칭 성공 (끝에 위치)
- × `Bash(git:* push)` — `git status push` 조차 매칭 실패 (`:*` 가 중간이라 리터럴로 해석)
- **우선순위** — **deny → ask → allow** 순 평가. 먼저 매칭되는 규칙이 결정

🎯 운영 원칙

- **deny 는 안전** — 두껍게 쌓기 (시크릿 · `sudo` · `rm -rf`)
- **ask 는 확인** — 파괴·발신 명령 (`git push` · `gh pr create` · `curl`) 은 사람이 한 번 승인
- **allow 는 편의** — 자주 쓰는 read-only 명령을 얹어 승인 프롬프트 감축

세밀 제어 — `Bash` · `WebFetch` · `MCP` · `Read/Edit/Write`

도구군	문법	예
Bash	명령 패턴 (스페이스 `git *` 또는 끝 `:*`)	`Bash(git commit *)` · `Bash(rm -rf:*)`
WebFetch	도메인	`WebFetch(domain:github.com)` · `WebFetch(domain:*.internal.*)`
MCP 도구[1]	서버:도구	`mcp__github__create_pull_request` · `mcp__slack__*`
Read / Edit / Write	경로 glob	`Read(./src/**)` · `Edit(./.env)`

📌 **세션 단위 예외.** `--add-dir` · `--allowed-tools` CLI 플래그로 이번 세션만 허용. `settings.json` 을 계속 편집하면 리포 diff 가 지저분해진다. **일회성은 CLI, 상시는 파일.**

---

1. **MCP (Model Context Protocol)** — 외부 도구 서버 규약. 이번 세션은 **서명 문법(`mcp__server__tool`)만** 다룬다. ↩

출처: Hooks | Permissions | Skill | Slash Command 문서

팀 규약 3원칙 — *permissions 로 팀 안전선 긋기*

1️⃣ 시크릿 경로 deny 필수

- `.env` · `credentials.json`
- `.ssh/*` · `id_rsa*`
- **사고 한 번이** 팀 전체 키 재발급으로 이어진다
- Read · Edit 양쪽에 걸기

2️⃣ `sudo` · `rm -rf` 기본 deny

- 기본은 무조건 차단
- 예외는 **로컬 설정에서만**
- 팀 리포에 개별 예외 안 넣기

3️⃣ MCP 는 최소 권한

- 쓰기 도구 (create · delete · post) 는 팀 리뷰 후 allow
- 읽기만 우선 열고, 쓰기는 신중
- 특히 GitHub · Slack · DB MCP

⚠️ `.env` **접근 사고는 개인이 아니라 조직 사고다.** deny 3중 방어 (permissions.deny · PreToolUse 훅 · git-ignore) 를 모두 걸어 둘 가치가 충분하다.

프로젝트 `permissions` 실전 템플릿 — *`<repo>/.claude/settings.json`*

```
{
  "permissions": {
    "allow": [
      "Bash(git status)", "Bash(git diff:*)", "Bash(git log:*)", "Bash(git branch:*)",
      "Bash(ls:*)", "Bash(cat:*)", "Bash(pytest:*)", "Bash(ruff:*)",
      "Read(./**)", "Edit(./src/**)", "Edit(./tests/**)",
      "WebFetch(domain:docs.python.org)", "WebFetch(domain:github.com)"
    ],
    "deny": [
      "Bash(rm -rf:*)", "Bash(sudo:*)", "Bash(curl:*)", "Bash(wget:*)",
      "Read(./.env)", "Read(./.env.*)", "Read(./secrets/**)",
      "Edit(./.env)", "Edit(./.env.*)"
    ]
  }
}
```

📌 **해설.** 읽기는 리포 전체, 쓰기는 `src/` · `tests/` 만 (node_modules · dist 오염 방지). 네트워크 명령은 기본 deny, 필요 시 세션 플래그로 개별 허용. **시크릿** · `sudo` · `rm -rf` **3중 차단.**

PART 4

Statusline · Custom Skill · Slash Command

> 하단 상태표시줄 커스터마이즈 · 자율 발동 skill · 명시 호출 slash · 개인 규약 자동화 4선.

Statusline — *하단 상태표시줄* · `settings.json`

Statusline 은 세션 하단에 뜨는 상태 줄. 기본은 디렉토리·git 브랜치·모델명. 스크립트로 원하는 정보를 붙일 수 있다.

`~/.claude/settings.json` **등록.**

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh"
  }
}
```

🔌 동작 원리

- Claude Code 가 statusline 스크립트의 **stdin 으로 세션 상태 JSON** 전달
- 스크립트 **stdout 첫 줄**이 세션 하단에 표시
- 셸이든 파이썬이든 **실행 파일이면 된다**

💡 개인 활용 예 3

- **작업 중인 티켓 번호** — 브랜치명에서 `JIRA-123` 추출
- **API 잔여 크레딧** — `/cost` 캐시 표시
- **로컬 서비스 상태** — `docker ps` 로 필수 컨테이너 색 코딩

Statusline 스크립트 — `~/.claude/statusline.sh`

```
#!/usr/bin/env bash
input=$(cat)
model=$(echo "$input" | jq -r '.model.display_name')
dir=$(echo "$input" | jq -r '.workspace.current_dir' | sed "s|$HOME|~|")
branch=$(git -C "$(echo "$input" | jq -r '.workspace.current_dir')" branch --show-current 2>/dev/null)
echo "[$model] $dir ${branch:+@$branch}"
```

📖 해설

- `input=$(cat)` — stdin JSON 을 통째로 받기
- `jq -r '.model.display_name'` — 모델명 추출
- `jq -r '.workspace.current_dir'` — 현재 작업 디렉터리
- `${branch:+@$branch}` — 브랜치가 있을 때만 `@브랜치` 표시

⚠️ 주의

- statusline 은 **모든 프롬프트마다 실행** — 무거운 명령 넣으면 세션이 느려진다
- 오래 걸리는 조회 (API · docker) 는 **로컬 캐시 파일**이 표준
- 실행 권한 필수 — `chmod +x ~/.claude/statusline.sh`

Custom Skill — `SKILL.md` 스켈레톤 · 자율 발동

Custom Skill 은 재사용 지시 묶음. 마크다운 한 장에 "이런 상황에 이렇게 처리" 를 담아 두면 Claude 가 해당 상황에서 **자율 발동** 한다.

```
---
name: korean-comment-cleaner
description: >
  Python·JavaScript·TypeScript 코드의 한글 주석을 정리한다.
  띄어쓰기·문장 부호 통일, 오래된 TODO 정리, 중복 주석 제거.
  코드에 한글 주석이 섞여 있을 때 자동 발동.
---

# korean-comment-cleaner

## 발동 조건
- 사용자가 "한글 주석 정리" 를 요청
- Edit·Write 로 수정하는 파일에 한글 주석이 다수 포함

## 처리 규칙
1. **띄어쓰기** — 조사 앞은 붙이고, 관형·명사 사이는 띄운다.
2. **문장 부호** — `.` 은 문장 끝에만, 목록은 `-` 로 통일.
3. **TODO** — `TODO:` 뒤 6개월 이상 오래된 것은 로그로 옮기고 제거.
4. **중복** — 같은 함수 안 같은 뜻 주석 2개 이상이면 하나로 합친다.

## 안전 규칙
- 주석의 **의미** 를 바꾸지 않는다. 표현만 정리.
- 라이선스 헤더 · 저작권 문구는 건드리지 않는다.
- 수정 결과는 diff 로 사용자에게 먼저 보인다.
```

Skill 스켈레톤 3요소 · 파일 배치 · 스코프

🧩 스켈레톤 3요소

- **frontmatter** — `name` · `description` 필수. Claude 는 startup 시 이 두 필드만 읽어 (skill 당 ~100 토큰) 훑는다
- **본문 마크다운** — skill 발동 시 **전체가 컨텍스트로 로드**
- **부속 파일 (선택)** — 같은 폴더에 파이썬 헬퍼 · JSON 설정 · 참고 문서

🔀 발동 2경로

- **자율 발동** — Claude 가 태스크 서술과 `description` 매칭
- **명시 호출** — 사용자가 `/skill-name` 입력

📁 파일 배치

```
~/.claude/skills/
└── korean-comment-cleaner/
    ├── SKILL.md          # 필수
    ├── example.py        # 선택
    └── glossary.md       # 선택
```

🌐 스코프 2개

- **개인** — `~/.claude/skills/`
- **프로젝트** — `<repo>/.claude/skills/`
- 추가·수정·삭제는 **현재 세션에서 즉시 반영** (재시작 불필요)

출처: Hooks | Permissions | Skill | Slash Command 문서

Slash Command — `~/.claude/commands/<name>.md`

Slash command 는 사용자가 명시 호출하는 재사용 프롬프트. skill 과 달리 **자율 발동하지 않아** 예측 가능성이 필요할 때 유리. 최소 예 `~/.claude/commands/summarize.md` (`/summarize`).

```
---
description: 현재 리포의 최근 커밋 5개를 3줄로 요약
allowed-tools:
  - Bash(git log:*)
  - Read(./**)
---

다음을 실행한다:

1. `git log --oneline -5` 로 최근 5개 커밋 확인.
2. 각 커밋의 diff 를 훑고, 무엇을 바뀌는가 한 줄로 요약
```

🔑 frontmatter 2 필드

- `description` — 명령 목록에 표시될 짧은 설명
- `allowed-tools` — 실행 중 허용 도구 화이트리스트. **안전 게이트** (`/review` 가 편집 도구를 못 쓰게)

📝 본문 지시

- `$ARGUMENTS` — 명령 뒤 인자가 치환 · `@file` — 파일 참조
- **파일명이 곧 명령 이름** — `status.md` → `/status`
- **프로젝트 명령** — `<repo>/.claude/commands/deploy.md` → `/deploy staging`

개인 규약 자동화 4선 — *네 개념 조합 실전*

규약	도구	위치
커밋 메시지에 티켓 번호 자동 삽입	`PreToolUse` 훅 (Bash matcher, 브랜치명에서 추출)	프로젝트
`.env` **접근 완전 차단**	`permissions.deny` + `PreToolUse` 훅 이중 게이트	개인 + 프로젝트
한글 주석 정리	Custom Skill (`korean-comment-cleaner`)	개인
`/pr` **— 현재 브랜치 PR 초안**	Slash command + `allowed-tools` 로 read-only 강제	개인

🔄 자동 vs 명시

- **자동이면** hook · skill
- **명시면** slash command

🛡️ 금지 목적

- **permissions.deny** (규칙)
- **+ PreToolUse 훅** (동작)
- 이중 게이트가 실무 표준

🎯 자율 발동 필요

- **skill** — 상황 인식해서 자율 발동
- 매번 호출해도 되면 **slash command**

PART 5

미니 실습 — *hook · slash · skill 각 1개*

> 참여자 각자가 자기 홈에 hook · slash · skill 각 1개를 만들고 동작을 확인한다. 순서대로 진행. 실패해도 다음으로 넘어가고 마지막에 트러블슈팅.

실습 ① — *개인 hook*

목표. 커밋 직전 자동 게이트(테스트·lint), 실패 시 차단.

단계.

```
mkdir -p ~/.claude
touch ~/.claude/settings.json
```

PART 2 의 훅 JSON (`PreToolUse` · `matcher: "Bash"` · `git commit` 감지 시 pytest) 을 `~/.claude/settings.json` 에 병합. `pytest` 를 자기 리포의 lint · 테스트 명령으로 교체 가능. **세션 재시작 후** 리포에서 `> git commit -m "test"` 시도.

👀 관찰 포인트

- **통과 시** — 훅 메시지 뜨고 커밋 진행
- **실패 시** — exit 2 로 차단, Claude 가 이유 요약해서 대답

🩺 실패 시 처방

- `jq: command not found` → `brew install jq` · `apt install jq`
- 훅 미발동 → `/hooks` 로 등록 상태 확인, JSON 파싱 오류 가능성
- 세션 재시작 필수 — 설정 편집만으로 반영 안 됨

실습 ② — *개인 slash command*

목표. `/status` 호출 시 리포 상태·최근 커밋·브랜치·미커밋 파일을 3줄 요약.

```
mkdir -p ~/.claude/commands
```

`~/.claude/commands/status.md`.

```
---
description: 현재 리포의 상태를 3줄로 요약
allowed-tools:
  - Bash(git status)
  - Bash(git log:*)
  - Bash(git branch:*)
---

`git branch --show-current` · `git status --short` · `git log --oneline -3` 을 실행한다.
```

🚀 실행

세션 안에서 `> /status` 입력.

👀 관찰

- `allowed-tools` 로 read-only 강제 — Edit·Write 불가
- `/help` **목록**에 `status` 등장
- 매 리포에서 반복하던 프롬프트가 명령 하나로 대체된다

실습 ③ — *개인 skill*

목표. `korean-comment-cleaner` skill 을 만들어 한글 주석 정리 요청 시 자율 발동.

단계.

```
mkdir -p ~/.claude/skills/korean-comment-cleaner
```

PART 4 의 SKILL.md 를 그대로 위 폴더에 저장. 세션에서 다음을 입력.

```
> src/main.py 의 한글 주석을 정리해줘
```

👀 관찰 포인트

- 응답 앞에 **skill 로드 표시** (UI 버전 상이)
- **처리 규칙 4개** 가 diff 에 반영됐는지 확인
- SKILL.md 편집 후 다시 요청하면 **재시작 없이 즉시 반영**

🔀 명시 호출

- `/korean-comment-cleaner` 로도 호출 가능
- 자율 발동 안 하면 명시 호출로 강제
- description 매칭이 약하면 description 을 보강

실습 정리 · 체크리스트

✅ 오늘의 체크리스트

- [ ] `~/.claude/settings.json` 훅 1개 · `/hooks` 로 확인
- [ ] `~/.claude/commands/status.md` · `/help` 목록에 표시
- [ ] `~/.claude/skills/*/SKILL.md` · 자율 발동 관찰
- [ ] `.env` deny 규칙 확인 (permissions)
- [ ] `settings.json` 은 **git 커밋 안 함** · dotfiles 리포에 백업 (선택)

🚀 확장 아이디어

- 훅은 **실제 커밋 흐름**에 붙이기 (테스트 · lint · 티켓 삽입)
- slash 는 **자주 쓰는 프롬프트** 하나 대체 (`/pr` · `/review` · `/log`)
- skill 은 **팀 코드 컨벤션** 하나 고정 (`snake_case` , JSDoc, 로그 포맷)

⚠️ **세 실습 중 하나라도 성공하면 오늘 목표는 채운 것** — 남은 시간에 실패한 것부터 트러블슈팅.

PART 6

정리 — *한 문장 결론*

> 프롬프트 · 컨텍스트 · 하네스 세 축 — 오늘의 한 문장.

마무리 — *한 문장으로*

"매 세션 반복하는 손을 규약으로 옮기는 것 — 그게 하네스다."

🎯 오늘 확보한 것

- `settings.json` 이 개인 하네스의 **뿌리** 라는 사실
- **hook · slash · skill** 이 실제 자기 홈에 하나씩 있다
- **allow · deny** 로 안전선을 그을 수 있다는 확신
- 반복 동작이 하나라도 있으면 **세 도구 중 하나로 옮길 수 있다**

🧭 남은 시간에 할 것

- 세 실습이 안 됐다면 강사에게 손 → **트러블슈팅**
- 세 실습이 다 됐다면 자기 팀에 **어떤 프로젝트 하네스** 를 넣을지 30초 회고
- 오늘 만든 파일 3개를 dotfiles 리포에 백업 (선택)