AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

프롬프트 엔지니어링 카드로 남기는 개발 자산

Session 2 · 카드로 남기는 개발 자산

원칙 4가지 · CO-STAR 6요소 · 실전 카드 5장 · 나쁜/좋은 프롬프트 대결

강사 박수현 · 🚀 젠아이랩스(GenAI Labs)

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

Session 2 · 카드로 남기는 개발 자산

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

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

🟢 **오늘 다루는 축**

📚 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

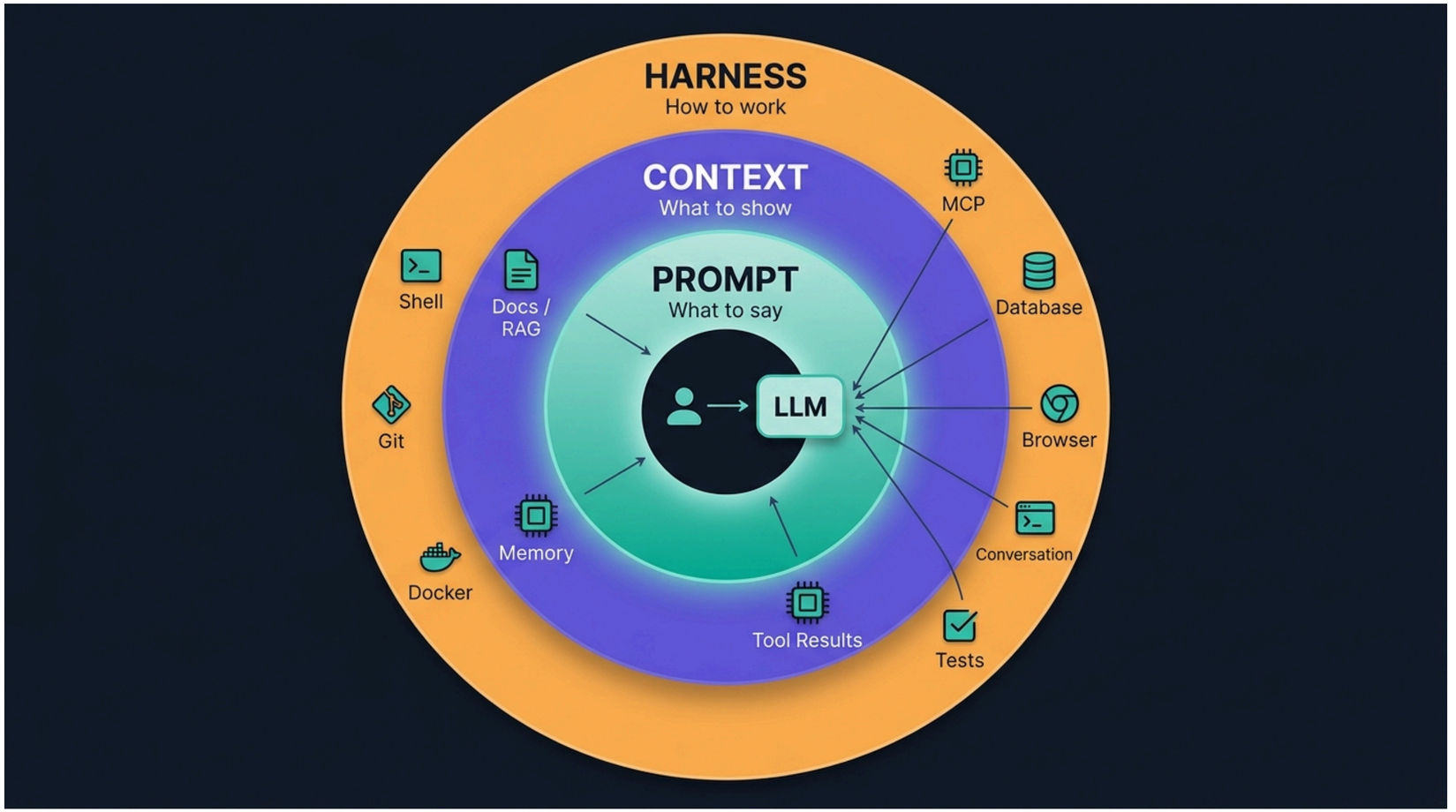
⚙️ 하네스 엔지니어링

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

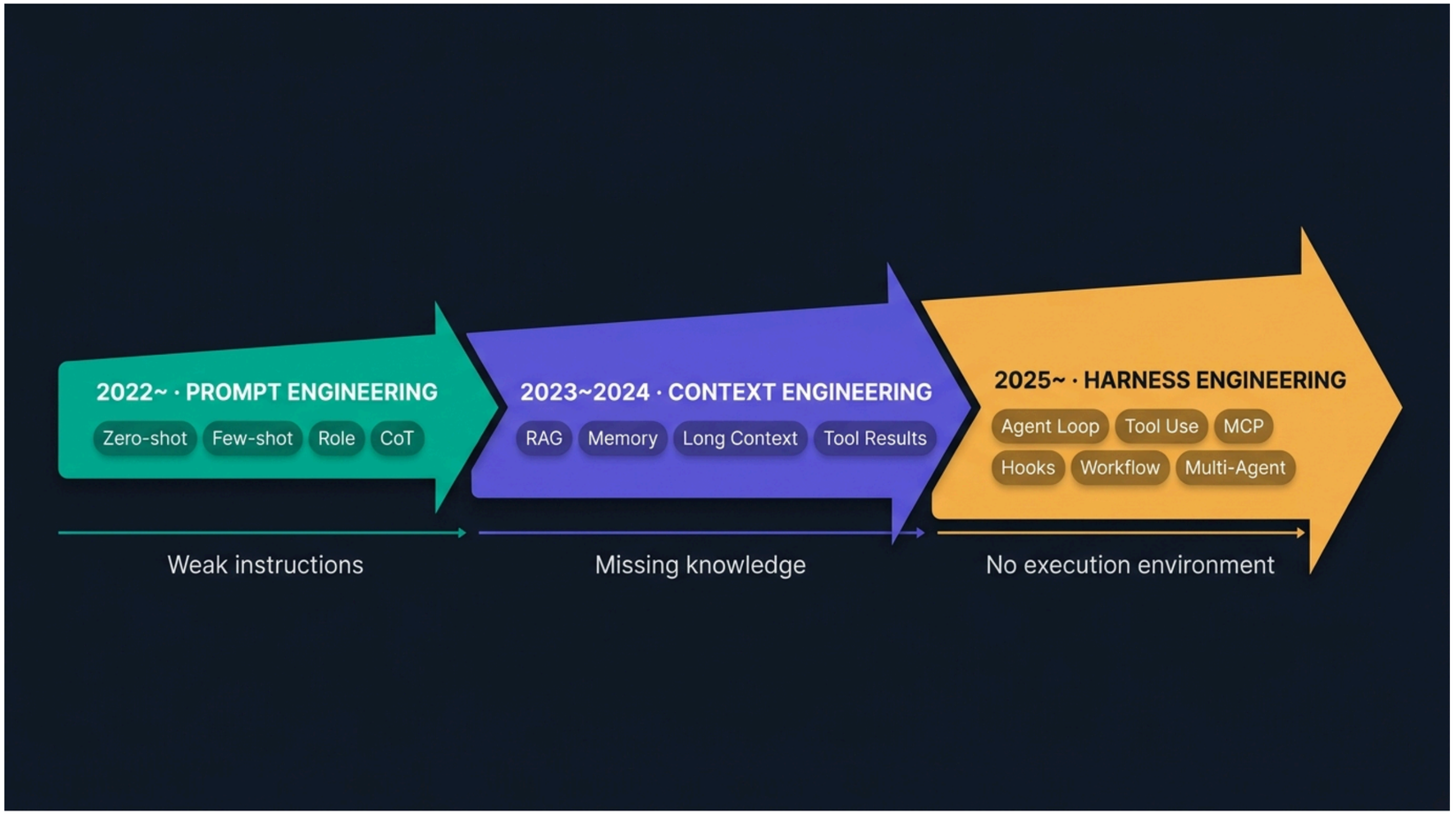
📌 **대체가 아니라 포함이다. 프롬프트 ⊂ 컨텍스트 ⊂ 하네스** — 하네스가 실행 환경을 세우고, 그 안에서 컨텍스트가 공급되고, 그 위에서 프롬프트가 실행된다. 새 축이 나왔다고 앞 축이 낡는 게 아니라 **다루는 범위가 커진 것**이다. 등장 순서는 대략 **2022~ 프롬프트 → 2023~2024 컨텍스트 → 2025~ 하네스** 로 흘렀다 (경계는 흐릿하다).

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드



프롬프트 ⊂ 컨텍스트 ⊂ 하네스 — 나란한 셋이 아니라, 뒤의 축이 앞의 축을 감싼다

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드



각 축은 앞 축의 한계에서 태어났다 — 약한 지시 · 없는 지식 · 없는 실행 환경

PART 1

프롬프트는 개발 행위다

> 자동완성과 대화의 차이 · 프롬프트 = 스펙+아키텍처+리뷰 · 실패의 대표 3 패턴.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

자동완성과 대화의 차이 — *지시가 성과물이다*

프롬프트가 개발 행위인 이유는 도구의 성격이 바뀌었기 때문이다. Copilot 계열의 자동완성은 다음 몇 줄을 추측한다. Claude Code · Codex 계열의 대화는 **무엇을 만들지 자체를 사람이 지시**한다. 지시의 품질이 결과의 품질을 결정한다.

※ **Codex** = OpenAI 의 대화형 개발 페어.

⌨️ Copilot 자동완성

- **입력 단위** — 편집 중인 커서 주변 몇 줄
- **결정권** — AI 가 다음 코드를 제안 → 사람이 승낙
- **실패 지점** — 문맥 부족한 자동완성 (오답률 낮음)
- **개발 행위** — 타이핑 가속

💬 Claude Code · Codex 대화

- **입력 단위** — 세션 전체 대화 · 파일 참조 · 셸 결과
- **결정권** — 사람이 무엇을·왜·어디까지를 지시
- **실패 지점** — 지시가 모호 · 과잉 · 미완성
- **개발 행위** — 스펙·설계·리뷰 지시

📌 **핵심 대비.** 자동완성 시대에는 "어떻게 쓸지" 가 개발자 손에 있었다. 대화형 페어에서는 **"무엇을 시킬지" 가 개발자 머리에 있다.** 코드가 아니라 지시가 성과물이다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

프롬프트 = *스펙 + 아키텍처 + 리뷰 지시*

잘 쓰인 프롬프트 한 통에는 세 가지가 담긴다.

📐 스펙

무엇을 만들지 · 어떤 입력·출력·제약을 두는지

- 파일·엔드포인트·자료 구조
- 성공 판정 기준

🏗️ 아키텍처

어떤 구조로 나눌지 · 어느 지점에서 결정을 미룰지

- 파일 배치 · 계층
- 2 턴 구조·다음 턴 예약

🔍 리뷰 지시

결과를 어떤 기준으로 판정할지 · 회귀 시 어떻게 반응할지

- 판단 축·심각도
- 회귀 게이트·롤백

📌 세 축을 담지 않은 프롬프트는 결과가 매번 다르다. **결과가 매번 다르면 개발 행위가 아니라 도박이다.** 이어지는 원칙 4가지가 세 축을 채우는 재료다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

실패의 대표 3 패턴 — *현장 관찰*

현장에서 반복 관찰되는 프롬프트 실패는 세 종류로 좁혀진다.

🌫️ 모호

증상 — 결과가 매번 다름 · 기대와 어긋남

원인 — 성공 판정 기준 없음 · 범위 불명확

🌀 과잉

증상 — 응답이 산만 · 결정을 안 함

원인 — 요구가 너무 많음 · 우선순위 없음

✂️ 미완성

증상 — 중간에 끊김 · 파일 절반만 수정

원인 — 완료 조건 없음 · 다음 단계가 모호

📌 **세 패턴의 공통 처방.** 프롬프트에 **"성공 판정 기준"** 을 못 박는다. "다음을 만족하면 완료" 라는 한 줄이 위 세 실패를 동시에 잡는다.

PART 2

원칙 4가지 — *명시성 · 역할 · 예시 · 재현성*

> Anthropic 공식 프롬프트 가이드가 소개하는 여러 기법 중 실전에서 반복되는 네 축.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

원칙 개요 — *네 축이 반복된다*

이 네 축을 카드에 담으면 대부분의 페어링 상황을 커버한다.

📌 명시성 (Explicit)

무엇을·왜·어디까지·성공 판정 기준까지 프롬프트 앞머리에 스펙 블록.

🎭 역할 (Role)

"너는 X 다" 로 페르소나 부여. 첫 줄에 시니어·리뷰어·아키텍트.

🎯 예시 (Few-shot)

좋은 예·나쁜 예 페어링. 프롬프트 중간에 입출력 샘플.

🔁 재현성 (Reproducible)

온도·모델·컨텍스트 고정. 세션 시작 시 `/model` · `--model` 명시.

📖 **참고.** Anthropic 공식 문서 — Prompt engineering overview · Prompting best practices

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

원칙 ① 명시성 — *무엇을·왜·어디까지*

명시성은 네 원칙 중 가장 무겁다. **성공 판정 기준까지 프롬프트 안에 들어가야 결과가 재현된다.**

🧱 명시성이 담아야 할 것

- **대상** — 파일 · 함수 · 엔드포인트
- **현재값** — 현재 지표 · 프로파일
- **목표값** — 낮추려는 값 · 응답 시간
- **응답 형태** — 표 · 3행 이내 · 4열
- **완료 조건** — 다음 턴 예약 · 코드 금지 시점

🎯 왜 무거운가

- 나머지 세 원칙(역할·예시·재현성)은 명시성 위에서 작동한다
- 명시성이 없으면 역할 부여도 예시도 응답 편차를 못 줄인다
- 완료 조건 한 줄이 대화를 2 턴으로 쪼갠다 → 재현성 상승

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

명시성 — *나쁜 예 vs 좋은 예*

🔴 나쁜 예

```
routes/feed.py 좀 빠르게 해줘
```

문제. "빠르게" 의 기준이 없다. 목표값·응답 형태·완료 조건 어느 것도 담기지 않았다.

결과. 매번 다른 부분을 손댄다. 재현이 안 된다.

🟢 좋은 예

```
routes/feed.py 의 GET /feeds 응답을
200ms 이하로 낮춰라.
현재 p50 380ms · p95 720ms · N+1 의심.

병목 3개를 우선순위 순으로
· 각 병목마다 개선안
· 위험도(하/중/상)
· pytest 회귀 가능성을 표로.

완료 조건: 3행 표 · 코드 수정은 다음 턴.
```

차이. 현재값·목표값·응답 형태·완료 조건을 모두 담았다.

📎 **용어.** `p50` · `p95` 는 응답 시간의 50·95 백분위 · `N+1` 은 목록 조회에서 자식 쿼리가 항목 수만큼 반복되는 패턴.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

원칙 ② 역할 — *페르소나와 강도*

"너는 시니어 파이썬 개발자다" 한 줄이 응답 밀도를 바꾼다. 역할 부여는 두 축으로 조절한다. **누구인지**와 **얼마나 엄격한지.**

🎭 역할 부여 예

- **시니어 개발자** — 일반 개발 · 리팩토링
- **보안 리뷰어** — 인증·권한·인젝션 관련
- **아키텍트** — 미들웨어 vs 별도 서비스 결정
- **코드 리뷰어 (엄격 모드)** — PR 최종 검토 · 회귀 방지
- **주니어 튜터** — 학습 목적 · 개념 설명

📌 **강도 조절.** "엄격하게" · "실용성 우선" · "일단 완성 우선" 같은 강도 지시가 응답 문체와 결정 기준을 바꾼다. 같은 시니어라도 강도에 따라 다른 리뷰가 나온다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

역할 — *나쁜 예 vs 좋은 예 (코드 리뷰)*

🔴 나쁜 예

```
이 코드 리뷰해줘.
```

문제. 역할이 없다. 어떤 관점·어떤 강도·어떤 축으로 볼지 지시가 비었다.

결과. 문법·스타일 같은 얕은 지적만 나온다.

🟢 좋은 예

```
너는 15년 차 백엔드 리뷰어다.
엄격하게 검토하되 팀 컨벤션은
CLAUDE.md 를 따른다.

검토 축: 성능 · 보안 · 테스트 · 가독성.
축별 발견 사항 3개 이내
· 심각도(하/중/상) 표기.
```

차이. 역할·강도·기준 문서·검토 축·응답 밀도를 모두 지정했다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

원칙 ③ 예시 (few-shot) — *좋은 예 + 나쁜 예 페어링*

언어 모델은 예시에서 패턴을 잡는다. **좋은 예 하나**보다 **좋은 예 + 나쁜 예 페어**가 응답 편차를 훨씬 좁힌다. 대조가 있으면 모델이 경계선을 좁게 잡는다.

📉 좋은 예만 넣을 때

- 종종 나쁜 예 스타일이 섞여 나온다
- 경계선이 흐릿하다
- 모델이 "어디까지가 좋은 예인지" 를 자기 분포로 넓힌다

📈 좋은 예 + 나쁜 예 페어

- 모델이 "이런 식은 피하라" 를 명시적으로 인식
- 경계선이 선명해진다
- 응답 편차가 눈에 띄게 준다

📖 **참고.** Anthropic 공식 — Use examples (multishot prompting)

예시 — *커밋 메시지 컨벤션 리뷰*

🔴 나쁜 예 (좋은 예만)

```
너는 커밋 메시지 리뷰어다.

[좋은 예] feat(auth): JWT 만료 시간
  15분 → 8시간

아래 3개를 좋은 예 스타일로 다시 써라:
"fix bug" · "refactor code" · "add feature"
```

문제. 좋은 예만 있어 경계선이 흐릿하다. 결과에 나쁜 예 스타일이 섞이곤 한다.

🟢 좋은 예 (좋은+나쁜 페어)

```
너는 커밋 메시지 리뷰어다.

[좋은 예] feat(auth): JWT 만료 시간
  15분 → 8시간
  - 짧은 만료로 세션 끊김 사고 3건
  - Refresh 토큰 도입 이전 임시 조치

[나쁜 예] update auth — 뭐 좀 고침

아래 3개를 좋은 예 스타일로 다시 써라:
"fix bug" · "refactor code" · "add feature"
```

차이. 나쁜 예가 경계선을 선명하게 그린다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

원칙 ④ 재현성 — *온도·모델·컨텍스트 고정*

같은 프롬프트에서 결과가 매번 흔들리면 재현이 아니다. 재현성은 세 축으로 확보한다.

🧠 모델 고정

- `--model sonnet` · `/model sonnet`
- 카드 상단에 권장 모델 명시
- 모델별로 응답 성향이 다르다

📚 컨텍스트 고정

- `@` 파일 참조로 컨텍스트 명시 — 세션 안에서 항상 인용된다 (예: `@src/foo.py` · `@~/.claude/prompts/refactorLoop.md`)
- `CLAUDE.md` 로 프로젝트 규약 자동 주입
- 세션 시작 시 컨텍스트를 못 박는다

🎲 온도 · 시드

- API 사용 시 `temperature=0` · `seed`
- CLI 세션은 온도 제어 제한적
- **프롬프트 자체를 결정론적으로** 쓴다

📌 **개인 프롬프트 저장 경로.** 재현성은 프롬프트를 파일로 남기는 데서 시작한다. `~/.claude/prompts/` 나 개인 노트에 카드를 축적한다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

재현성 — *휘발 vs 저장*

🔴 나쁜 예 (휘발 흐름)

- 세션에서 즉흥으로 프롬프트를 만든다
- 잘 나온 결과에 만족하고 세션을 닫는다
- 프롬프트는 세션 로그 안에 남을 뿐 검색되지 않는다
- 다음 주에 비슷한 상황이 오면 다시 처음부터 조립
- 결과 품질이 매번 다르다

🟢 좋은 예 (저장 흐름)

```
~/.claude/prompts/
├── specToArch.md
├── refactorLoop.md
├── regressionGuard.md
├── archDecision.md
└── incidentReview.md
```

- 카드마다 상단에 권장 모델·상황·변수 명시
- 세션 시작 시 `@~/.claude/prompts/refactorLoop.md` 로 참조
- 같은 카드를 그대로 복붙 → 응답 편차가 줄어든다

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

네 축 위에 얹는 것 — *생각의 사슬 (Chain-of-Thought)*

역할·예시·명시성이 **지시를 정돈**한다면, CoT 는 **지시를 받은 뒤의 순서를 정돈**한다. "단계별로 생각해 보라" 고 유도해 모델의 추론 과정을 응답 밖으로 꺼내는 기법이다.

🕰️ 원래 형태

- `"Let's think step by step"` 한 줄로 중간 추론을 유도
- 산술·논리 문제에서 정답률이 눈에 띄게 올랐다
- 추론 모델이 보편화되며 **명시적 CoT 요구는 줄었다** — 모델이 알아서 사고 과정을 돈다

💻 코드 작업에서의 형태

- **"먼저 계획을 세워 보여 준 뒤 코드를 써라"**
- 🔴 `"이 모듈 리팩터링해줘"` → 곧장 파일 12개가 바뀐다
- 🟢 `"① 현재 구조 요약 ② 변경 후보 3개와 근거 ③ 착수 순서 를 먼저 제시하고, 승인 뒤 코드에 손대라"`

📌 **계획을 먼저 받으면 방향이 틀렸을 때 코드 한 줄 쓰기 전에 잡는다.** 되돌리는 비용이 가장 싼 지점이 계획 단계다. Claude Code 의 **Plan 모드** (Shift+Tab 순환) 가 이 형태를 제품으로 굳힌 것이다 — 계획을 내놓고 승인받기 전에는 파일을 쓰지 않는다.

PART 3

CO-STAR 프레임 — *6요소 순서*

> Sheila Teo 가 GovTech Singapore 프롬프트 대회에서 우승하며 정리한 프레임. 6 요소를 순서대로 채우면 원칙 4가지가 그대로 배치된다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

CO-STAR — *상위 3요소 (C · O · S)*

CO-STAR 는 2023년 GovTech Singapore 프롬프트 대회에서 우승한 Sheila Teo 가 정리해 공개한 프레임이다.

C — Context

배경 · 현재 상황 · 전제

무엇을 다루는지, 어떤 스택·환경인지.

예 — "FastAPI 앱 · Postgres · 하루 100만 요청"

O — Objective

하고 싶은 것 · 목표값

무엇을 이루고 싶은지, 수치가 있으면 명시.

예 — "GET /feeds 응답을 200ms 이하로"

S — Style

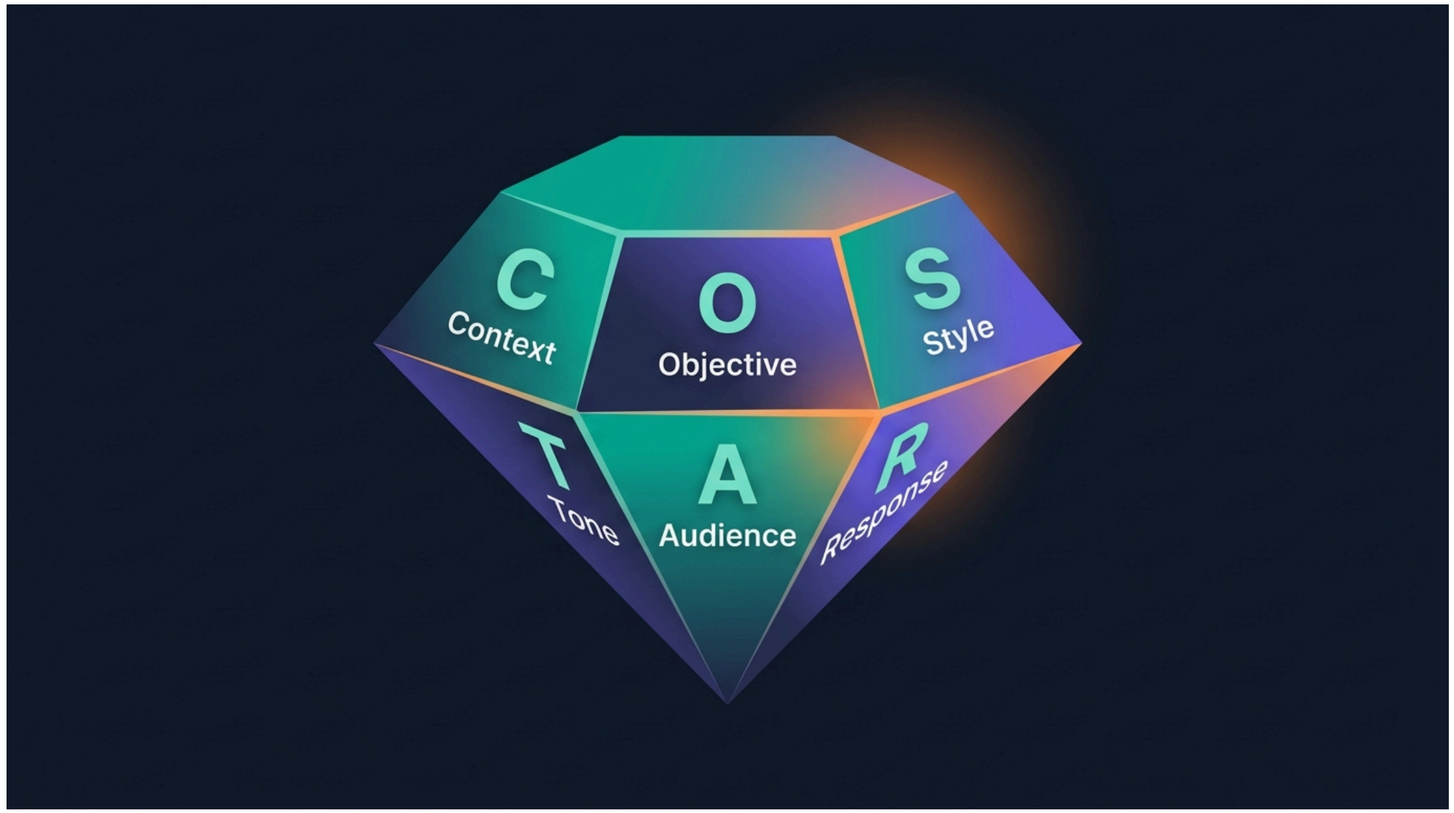
응답 형식 · 문체

표인지 서술인지, 몇 행·몇 열인지.

예 — "표 · 3행 이내 · 단정문"

📖 **참고.** Sheila Teo — How I Won Singapore's GPT-4 Prompt Engineering Competition · CO-STAR Framework 정리 (AI Advisory Boards)

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드



CO-STAR — 상위 3요소 (C · O · S)

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

CO-STAR — *하위 3요소 (T · A · R)*

T — Tone

어조 · 강도

엄격한지 실용적인지, 위험 우선인지 완성 우선인지.

예 — "엄격 · 위험 우선"

A — Audience

응답을 읽을 사람

누구를 대상으로 쓰는지. 배경 지식 수준까지.

예 — "팀 리드 · 5년 차 백엔드"

R — Response

산출 형태 · 완료 조건

무엇을 내놓고 어디서 멈출지. 2 턴 구조도 여기.

예 — "1차: 진단 표 / 2차: 코드 diff"

📌 **R (Response) 이 카드의 핵심 축.** 대화를 2 턴으로 쪼개는 지시가 여기 들어간다. "지금은 표만 · 승인하면 다음 턴에서 코드" 같은 구조가 재현성을 끌어올린다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

CO-STAR 실 예제 — *FastAPI 라우트 리팩토링*

앞서 나온 GET /feeds 리팩토링을 6요소에 맞춰 조립한다.

```
# Context
너는 15년 차 백엔드 리뷰어다.
대상 코드: FastAPI 앱 · Postgres 15 · 하루 100만 요청 · Python 3.11.
파일: routes/feed.py · models/feed.py · services/feed_service.py
현재 프로파일: GET /feeds p50 380ms · p95 720ms.

# Objective
GET /feeds 응답 시간을 p95 200ms 이하로 낮추는 개선안을 뽑는다.
기존 API 스펙은 유지 · 클라이언트 변경은 허용하지 않는다.

# Style
표 형식 · 각 행 3열 이내 · 단정문.

# Tone
엄격 · 회귀 위험을 우선 표기.

# Audience
팀 리드 (5년 차 백엔드) · Postgres 인덱스 개념 익숙 · MSA 경험 없음.

# Response
1차 응답 = 병목 3개 진단 표 (원인 · 개선안 · 위험도 · 소요).
표 검토 후 승인하면 2차 응답에서 코드 diff.
지금은 코드 작성 금지.
```

📌 **관찰.** 이 프롬프트는 "코드 짜줘" 가 아니라 **먼저 진단 표부터 요구**한다. 대화를 2 턴으로 쪼갠 게 R (Response) 파트다. 이 2 턴 구조가 재현성의 핵심이다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

카드 만드는 절차 — *4단계*

CO-STAR 를 매번 처음부터 쓰지 않는다. 자기 상황별 템플릿을 카드로 만들어 두고, 상황이 오면 **카드 안 변수만 바꾼다.**

1️⃣ 상황 정의

카드를 꺼낼 상황 한 문장.

예 — "라우트 하나를 성능 개선할 때"

2️⃣ 6요소 채우기

Context · Objective · Style · Tone · Audience · Response 를 뼈대로 초안.

3️⃣ 변수화

파일명·목표값 같은 부분은 `{file}` · `{target}` 같은 자리표시로.

4️⃣ 저장

`~/.claude/prompts/perfRoute.md` 같은 이름으로 개인 저장소에 커밋.

📌 **목표.** 이 카드를 15장 정도 확보하면 대부분의 페어링 상황이 **카드 한 장 + 변수 3~4개**로 해결된다. 시작은 5장.

PART 4

실전 프롬프트 카드 5장

> 그대로 복붙해 쓸 수 있는 완성본. 상단에 상황·권장 모델, 본문에 CO-STAR 6요소, 하단에 변수 목록. 파일명 규약: `~/.claude/prompts/{camelcase}.md`.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

카드 1 — *스펙 → 아키텍처*

상황. 새 기능·새 서비스의 요구사항을 받아 파일 트리와 데이터 모델부터 뽑고 싶을 때. **권장 모델.** Opus (설계 판단) · 응답 확정 후 Sonnet 으로 전환. **예시 스택.** FastAPI + SQLAlchemy + Pydantic.

```
# 상황
{feature_name} 기능을 신규 개발한다.
스택은 {stack} 이다.

# 요구사항
{requirements}

# 요청
1. 파일 트리 (디렉토리 3레벨까지) 만 먼저 제시
2. 데이터 모델 (SQLAlchemy · Pydantic) 도 이름·필드·관계까지
3. API 엔드포인트 목록 (메서드·경로·요청/응답 스키마 요약)
4. 위 세 개가 확정될 때까지 실제 코드는 작성 금지

# 결정 기준
- 파일 트리는 도메인 중심 (routes/ · services/ · repositories/)
- N+1 회피를 데이터 모델에 반영
- 각 엔드포인트는 인증·에러 처리 명시

# 완료 조건
- 파일 트리 · 데이터 모델 · 엔드포인트 표 세 블록으로 응답
- 아직 코드는 없음
```

변수. `{feature_name}` · `{stack}` · `{requirements}`

카드 2 — *반복 리팩토링*

상황. 특정 파일·모듈의 병목·중복을 진단하고 개선안을 표로 받고 싶을 때. **권장 모델.** Sonnet.

```
# 상황
대상 파일: {files}
프로파일 · 지표: {profile}
관찰된 증상: {symptoms}

# 요청
1. 병목 3개를 우선순위 순으로 나열
2. 각 병목마다 개선안 · 예상 개선폭 · 위험도(하/중/상) · 소요(1h/1d/1w)
3. 표 형식 · 3행 이내
4. 기존 pytest 회귀 가능성 열을 추가

# 결정 기준
- 우선순위 = 예상 개선폭 × (1 / 위험도) × (1 / 소요)
- 위험도 상은 배포 게이트 명시

# 완료 조건
- 표 한 개 · 4열 (원인 · 개선안 · 위험도 · 소요)
- 코드 수정은 다음 턴
```

변수. `{files}` · `{profile}` · `{symptoms}`

카드 3 — *회귀 방지*

상황. PR 이 기존 테스트를 깨는지 dry-run 결과와 함께 검증하고 싶을 때. **권장 모델.** Sonnet · CI 배치 시 Haiku.

```
# 상황
대상 PR · 브랜치: {branch}
변경 파일: {changed_files}
관련 테스트 스위트: {tests}

# 요청
1. 변경으로 영향을 받을 수 있는 테스트를 먼저 나열
2. 각 테스트에 대해 예상 결과 (통과 · 실패 · 판단 어려움)
3. 실패 예상 테스트는 원인 3줄 이내로 진단
4. 회귀 확인 시 수정 diff 를 최소 변경 원칙으로 제시

# 결정 기준
- 회귀 발견 = 배포 차단
- 판단 어려움 = 실제 pytest 실행 명령을 제안

# 완료 조건
- 테스트 목록 표 (파일 · 결과 · 원인)
- 필요 시 최소 diff 블록
```

변수. `{branch}` · `{changed_files}` · `{tests}`

카드 4 — *아키텍처 결정*

상황. 미들웨어 vs 별도 서비스, 캐시 계층 위치, 큐 도입 여부 등 장기 영향이 큰 결정을 앞뒀을 때. **권장 모델.** Opus. **용어.** TCO = 초기 개발 + 유지보수 + 장애 + 이관 총합.

```
# 상황
결정 대상: {decision}
현재 시스템: {current_arch}
제약 조건: {constraints}

# 요청
선택지 2~3개를 나열하고, 각 선택지에 대해 다음을 표로.
1. 개발 초기 비용 (1주 단위)
2. 5년 유지보수 비용 (연 단위 인력 · 인프라)
3. 장애 영향 반경
4. 롤백 난이도
5. 팀 학습 곡선

# 결정 기준
- 5년 관점에서 총 소유 비용 (TCO) 이 낮은 순으로 정렬
- 각 선택지에 "언제 이 선택이 옳은가" 조건 3개 이내

# 완료 조건
- 표 한 개 (선택지 × 5축)
- 마지막에 강사 추천 한 줄 · 근거 두 줄
```

변수. `{decision}` · `{current_arch}` · `{constraints}`

카드 5 — *실패 회고*

상황. 에러 로그 · 장애 이력을 받아 원인 분류 · 재현 조건을 정리하고 싶을 때. **권장 모델.** Sonnet · 로그 크면 Haiku.

```
# 상황
발생 시각: {timestamp}
증상: {symptom}
관련 로그 (원문): 아래 붙임
{logs}

# 요청
1. 원인 후보 3개 · 각 후보의 근거를 로그에서 인용
2. 각 원인의 재현 조건 (입력 · 순서 · 환경 변수)
3. 가장 유력한 원인을 지목하고, 반증 시나리오 1개
4. 재발 방지 체크리스트 3항목

# 결정 기준
- 근거는 반드시 로그 라인 번호 인용
- 추측성 서술 금지 (근거 없으면 "확인 필요" 로 표기)

# 완료 조건
- 원인 후보 3개 · 재현 조건 · 유력 원인 · 반증 시나리오 · 재발 방지 5블록
```

변수. `{timestamp}` · `{symptom}` · `{logs}`

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

카드 5장 정리 — *프로젝트 한 사이클*

다섯 카드는 각각 다른 국면을 커버한다. **다섯 카드로 프로젝트 한 사이클이 돈다.**

🧭 사용 국면

- **카드 1 · 스펙 → 아키텍처** — 신기능 착수
- **카드 2 · 반복 리팩토링** — 개발 중반
- **카드 3 · 회귀 방지** — 배포 직전
- **카드 4 · 아키텍처 결정** — 중장기 판단
- **카드 5 · 실패 회고** — 사후 · 인시던트

🎛️ 권장 모델

- **카드 1** — Opus
- **카드 2** — Sonnet
- **카드 3** — Sonnet · CI: Haiku
- **카드 4** — Opus
- **카드 5** — Sonnet · 로그 크면 Haiku

📌 **자산의 관점.** 카드는 곧 자산이다. 다음 파트 실습에서 카드 하나를 즉석에서 써 본다.

PART 5

미니 실습 — *프롬프트 대결*

> 각자 자기 리포에서 라우트·함수 하나를 골라 나쁜 프롬프트·좋은 프롬프트를 순서대로 던진다.

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

문제 세팅 — *5단계 절차*

각 참여자 노트북에서 동시에 진행. 강사는 순회하며 프롬프트 밀도를 확인한다.

🎯 실습 절차

1. **대상 코드 선정** — 자기 리포에서 라우트·함수 하나
2. **나쁜 프롬프트 던지기** — 한 줄짜리
3. **좋은 프롬프트 던지기** — 카드 2 + 변수
4. **결과 비교 · 옆 사람과 공유**
5. **회고 · 카드에 담을 항목**

📐 대상 코드 조건

- **30~80줄** 사이가 이상적
- 라우트 하나 · 함수 하나
- 프로파일 지표를 알고 있으면 더 좋음

🧰 준비물

- 자기 리포 (없으면 예제 리포 배포)
- Claude Code 세션
- 카드 2 (반복 리팩토링) 원문

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

대결 — *나쁜 프롬프트 vs 좋은 프롬프트*

🔴 나쁨

```
routes/feed.py 좀 리팩토링해줘
```

결과 관찰. - 결과가 산만하다 - 어디를 어떻게 바꿨는지 파악하기 어렵다 - 회귀 위험도 표시가 없다 - 다시 물어봐도 매번 다른 부분을 손댄다

🟢 좋음

Part 4 카드 2 (반복 리팩토링) 를 그대로 복붙하고 변수만 자기 리포에 맞춘다.

- `{files}` = `routes/feed.py`
- `{profile}` = `p50 380ms · p95 720ms · N+1 의심`
- `{symptoms}` = `페이지네이션 없음 · 응답 페이로드 과다`

결과 관찰. - 결과가 표 한 개로 온다 - 재현되고, 팀 리드에게 그대로 공유 가능 - 같은 프롬프트를 다시 던져도 열 구조·판단 축이 유지된다

출처: Anthropic Prompt Engineering Guide | CO-STAR | 실전 프롬프트 카드

결과 비교 · 공유 — *4개 축*

결과를 옆 사람과 교환한다. 두 응답의 차이를 다음 축으로 요약한다.

📊 비교 축 (1·2)

- **재현성** — 나쁜: 매번 다름 / 좋은: 열 구조 고정
- **판단 근거** — 나쁜: 서술문 · 산만 / 좋은: 표 · 위험도·소요 명시

📊 비교 축 (3·4)

- **팀 공유 적합성** — 나쁜: 낮음 / 좋은: 그대로 공유 가능
- **다음 액션** — 나쁜: 불명확 / 좋은: 표 승인 → 다음 턴 코드

📌 **공유 시 물어볼 것.** 옆 사람 카드에서 배울 부분이 있는지, 자기 카드에 넣을 항목이 있는지. **이 순간이 카드 자산이 확장되는 지점**이다.

PART 6

정리 — *카드 저장 · 개인화 · 한 문장 결론*

> 카드는 저장해야 자산이 된다. 저장 경로와 개인화 원칙을 짚고 마친다.

카드 저장 · 개인화

카드는 저장해야 자산이 된다. 저장 경로는 세 가지가 표준이다.

📁 `~/.claude/prompts/`

Claude Code 세션 안에서 `@` 로 바로 참조 가능.

- 개인 저장소로 가장 빠름
- 세션 시작과 함께 즉시 로드

📝 Notion · Obsidian

검색·태깅에 강함.

- 카드 수가 늘면 검색성이 관건
- 태그로 국면·모델별 정렬

🔧 dotfiles 리포

팀 · 여러 노트북 동기화.

- git 히스토리로 카드 진화 추적
- 노트북 여러 대 쓰는 사람에게 유리

📌 **개인화 원칙.** 카드는 다른 사람 것을 그대로 쓰지 않는다. 자기 리포의 파일명·팀 규약·검토 축을 반영해 매번 수정한다. **남의 카드 100장보다 자기 카드 10장이 강하다.**

마무리 — *한 문장으로*

S2 가 끝났다. 원칙 4가지 · CO-STAR 6요소 · 카드 5장을 정리했다. 나쁜/좋은 프롬프트의 격차를 자기 리포에서 확인했다.

🎯 이번 세션의 결론 한 줄

"프롬프트는 코드보다 오래 남는 자산이다. 카드로 만들어 저장하라."

프롬프트가 매번 사라지면 개발 행위가 아니라 노동이다. 카드로 만들어 저장하면 다음 작업의 시작 지점이 오늘 끝난 지점이 된다.