

Vibe Handbook

Vibe Handbook

차례

Git & GitHub	5
버전 관리란 무엇인가 (그리고 왜 필요한가)	6
리포지토리, 커밋, 그리고 이력	6
브랜치: 두려움 없이 시도하기	7
실제로 무엇이 바뀌었는지 보기 (diff)	7
“이런”버튼: undo, revert, reset	8
리모트와 GitHub: 클라우드 속 내 프로젝트	8
풀 리퀘스트와 이슈	9
.gitignore 와 절대 비밀을 커밋하지 않기	9
첫 repo, 처음부터 끝까지	9
AI 에게 운전대를 맡기되 (여전히 내가 책임지기)	10
정리와 실습	10
The Command Line	11
터미널과 shell 이 실제로 무엇인가	11
프롬프트 읽기	12
매일 쓰는 명령어들	12
프로그램과 dev 서버 실행하기	13
출력과 exit 코드 읽기	13
Ctrl-C 로 프로세스 멈추기	13
PATH 와 “command not found”	14
AI 가 주는 명령어를—안전하게—실행하기	14
만나게 될 패키지 매니저	14
정리와 실습	15
How the Web Works	16
두 등장인물: 클라이언트와 서버	16
URL 이란 실제로 무엇인가	17
요청과 응답의 순환	17
HTTP 메서드: 동사들	17
상태 코드: 작동했는가?	18
헤더와 쿠키	18
HTTPS 와 자물쇠	18
이름에서 서버로: DNS	19
이것이 당신의 앱에 어떻게 대응되는가	19
정리와 실습	20
API, JSON 그리고 데이터	21
API 가 실제로 무엇인가	21
엔드포인트와 REST, 쉽게 풀어서	22
JSON 이 무엇인가	22
당신이 마주칠 데이터 타입들	23

실제 요청과 응답	23
API key 와 인증 헤더	24
속도 제한과 요금 폭탄의 함정	24
API 문서를 읽는 법	25
AI 에게 호출을 맡기되—당신은 모양을 점검하기	25
정리와 실습	25
앱의 해부학	26
큰 그림	27
frontend, backend, 데이터베이스, 호스팅	27
HTML, CSS, JavaScript	27
패키지와 의존성	28
환경 변수와 설정	29
프로젝트의 생김새	29
요청은 어떻게 흐르는가	30
정리 & 연습	31
바이브 코딩 마인드셋	32
바이브 코딩이란 실제로 무엇인가	32
마인드셋 전환	33
루프: 의도 □ 생성 □ 검토 □ 개선	33
빛을 발하는 곳	35
실패하는 곳	35
흔한 함정	35
안목이 새로운 핵심 역량이다	36
현실적인 기대치	36
정리와 실습	36
AI 작업 공간 구축하기	38
AI 네이티브 에디터 또는 어시스턴트 선택하기	38
AI 가 탐색할 수 있도록 프로젝트 구조 잡기	39
컨텍스트 파일 작성하기 (AGENTS.md)	40
버전 관리를 반사 신경으로 만들기	41
피드백 루프를 촘촘하게 유지하기	41
시크릿을 신중하게 관리하기	42
온전한 로컬 환경이 AI 변경을 안전하게 만든다	42
실시간 진단 (LSP)	42
에이전트 확장하기	43
macOS vs Windows	43
TUI vs GUI	44
하네스 엔지니어링	44
비용 안전: CLI vs API	45
정리와 실습	45
엔지니어처럼 프롬프트하기	46
작업을 주기 전에 맥락을 줘라	47
입력, 출력, 엡지 케이스를 명시하라	47
작은 단계로 작업하라	48
코드 전에 계획을 요청하라	48
재작성이 아니라 diff 로 반복하라	49
예시를 제공하라	49
AI 에게 하지 말아야 할 것을 말하라	50
컨텍스트를 자원처럼 관리하라	50
피해야 할 흔한 안티패턴	51
핵심 마인드셋	51

정리와 실습	51
아이디어에서 명세까지, 몇 분 안에	52
왜 그냥 느낌이 아니라 명세인가	53
1 단계: 문제를 명확히 하라	53
2 단계: 범위와 비목표를 정의하라	54
3 단계: 사용자 스토리를 작성하라	54
4 단계: 가벼운 PRD 초안을 작성하라	54
5 단계: 데이터 모델을 스케치하라	55
6 단계: 명세를 바이브 크기 작업으로 쪼개라	56
명세를 살아 있게 유지하라	57
흔한 함정: 문제가 아니라 해결책을 명세화하기	57
보상	57
정리와 실습	57
스택 고르기: 한눈에 보는 언어들	59
JavaScript / TypeScript	60
Python	60
Go	60
Rust	61
나머지 언어들에 대한 짧은 한마디	61
한눈에 보는 비교	62
이국적인 선택의 진짜 비용	62
목표로 고르기	63
복습과 연습	64
기능 만들기, 바이브 하나씩	65
빌드 루프	65
“수직 슬라이스”의 의미	66
실전 예제: “즐거찾기”기능	67
모든 줄을 이해하지 않고 diff 읽기	68
AI 를 풀어둘 때 vs. 직접 조종할 때	69
앱을 항상-동작 상태로 유지하기	69
작은 단위로 커밋하기	69
슬라이스가 너무 클 때	70
핵심 정리	70
복습과 연습	70
AI 로 디버깅하기	71
핵심 마음가짐	71
디버깅 체크리스트	72
AI 에게 올바른 맥락 먹이기	73
이해 안 되는 에러 읽기	74
가설, 로깅, 그리고 이진 탐색	74
근본 원인 대 증상	75
AI 가 빙빙 돌 때	75
간직하기 전에 수정을 이해하라	75
핵심 정리	76
복습과 연습	76
서버 인프라	77
코드가 살 수 있는 다섯 곳	77
풀어 쓴 몇 가지 전문 용어	78
트레이드오프 표	78
언제 무엇을 고를까	79

AI 가 배포와 설정을 엮는 법	79
1 인 빌더를 위한 권장 기본 경로	80
“매니지드”가 진짜 의미하는 것 (그리고 당신이 여전히 소유하는 것)	81
빠른 출시 전 체크리스트	81
핵심 정리	82
정리 & 연습	82
데이터베이스 인프라	83
데이터 저장의 세 가지 형태	83
매니지드 vs 셀프 호스팅	84
스키마 설계 기초	85
마이그레이션: 스키마를 안전하게 바꾸기	86
인덱싱	86
캐싱	87
큐	87
흔한 초보자 실수	87
데이터 계층 구축을 AI 에게 지시하기	88
정리 & 연습	88
가드레일, 테스트 & 리뷰	89
AI 를 지시할 때 가드레일이 필요한 이유	90
테스트: 당신의 안전망	90
타입 체크와 린팅: 공짜로 잡는 실수	91
AI 변경의 범위 좁히기	92
디프를 비판적으로 읽기	92
리뷰 게이트: 머지 전 체크리스트	92
CI 기초: 점검을 자동으로 만들기	93
기본이 안전한 습관	94
가드레일이 비명을 지를 때: 고치기 전에 분류하라	94
정리 & 연습	94
프로덕션으로 출시하기	95
호스트를 고르고 배포하기	96
커스텀 도메인과 DNS	97
환경 변수와 비밀	98
결제 연결하기	98
스테이징과 롤백	99
모니터링과 에러 추적	100
성능과 캐싱	100
출시 전 체크리스트	100
출시 후	100
정리와 연습	101
사례 연구: 0 에서 출시까지	102
사례 연구 1: 다국어 랜딩 + 결제 페이지	102
사례 연구 2: Auth + 데이터베이스가 있는 작은 SaaS 도구	104
사례 연구 3: 개인 자동화 유틸리티	106
사례 연구 4: 손 갈 필요 없는 콘텐츠 사이트	107
넷의 공통점	109
정리와 연습	109
바이브 코더를 위한 보안	110
AI 가 생성한 코드가 기본적으로 안전하지 않은 이유	111
인젝션: 입력이 코드가 될 때	111
비밀값과 API 키: 노출된 키의 함정	112

인증 vs 인가: 누구나 호출할 수 있는 엔드포인트	113
파일 처리와 업로드	114
의존성 위험: AI 가 지어낸 패키지	114
보안 리뷰 게이트	114
자동화 안일, 보안 편	115
정리와 연습	115
에이전트 오케스트레이션	117
세 가지 계층	117
언제 풀어놓고 언제 직접 조종할 것인가	118
해매지 못하도록 과업의 범위를 정하기	118
백그라운드 및 병렬 에이전트	119
멀티 에이전트: 팬아웃, 적대자, 종합	120
규모에 맞춰 출력 검토하기	121
비용과 폭주 루프	121
정직한 한계	122
정리와 연습	122

Git & GitHub



바이브 코딩을 시작하기 전에 딱 하나의 도구만 배운다면, 바로 이것을 배우세요. Git 은 여러분을 용감하게 만들어 주는 도구입니다. AI 에게 “이 파일 전체를 다시 써줘”라고 말하고도 마음 편할 수 있는 이유가 바로 Git 입니다. 언제든지 잘 작동하던 버전으로 되돌아갈 수 있으니까요.

솔직하게 말하면 이렇습니다. AI 에게 소프트웨어를 만들라고 시키면, AI 는 반드시 무언가를 망가뜨립니다—그것도 자주. 자신 있게 파일을 다시 쓰고, 잘 작동하던 부분을 지우고, 엉망진창을 안겨줍니다. 짜증나는 오후 하루로 끝나느냐, 일주일을 통째로 날리느냐의 차이는 안전망이 있었는지에 달려 있습니다. Git 이 그 안전망입니다. 이 장은 사전 코딩 지식 없이 처음부터 Git 을 가르칩니다.

버전 관리란 무엇인가 (그리고 왜 필요한가)

저장 지점이 있는 비디오 게임을 떠올려 보세요. 까다로운 구간을 깨고, 저장한 다음, 위험한 시도를 해봅니다. 잘못되면 게임 전체를 처음부터 다시 하는 대신 마지막 저장 지점을 불러옵니다. **버전 관리**는 **프로젝트의 저장 지점**입니다.

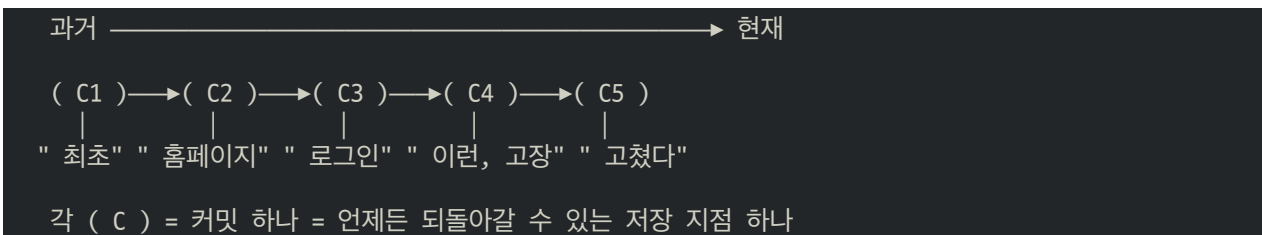
버전 관리가 없으면 프로젝트는 단 하나의 상태만 갖습니다. 바로 지금 이 순간의 모습뿐이죠. AI 가 코드를 망가뜨리면, 멀쩡했던 버전은 사라집니다. 버전 관리가 있으면 저장된 모든 상태의 전체 타임라인을 유지하며, 그중 어느 지점으로든 되돌아갈 수 있습니다.

Git 은 가장 널리 쓰이는 버전 관리 도구입니다. 여러분의 컴퓨터에서 돌아가며, 요청할 때마다 프로젝트의 스냅샷을 조용히 기록합니다. Git 이 공짜로 주는 몇 가지를 보면:

- **안전망**—몇 시간이 지난 뒤에도 잘못된 변경을 되돌릴 수 있습니다.
- **이력**—무엇이, 언제, 왜 바뀌었는지 정확히 볼 수 있습니다.
- **자신감**—아무것도 진짜로 사라지지 않으니 마음껏 실험할 수 있습니다.

마지막 항목이 진짜 보상입니다. 바이브 코딩은 빠르게 움직이며 이것저것 시도할 때 가장 잘 됩니다. Git 은 그 빠른 움직임을 무모함이 아닌 안전함으로 만들어 줍니다.

프로젝트의 이력을 언제든지 되돌아갈 수 있는 저장 지점들의 줄로 그려 보세요:



리포지토리, 커밋, 그리고 이력

계속 듣게 될 세 단어입니다. 하나씩 풀어 봅시다.

리포지토리 (줄여서 “repo”) 는 그냥 Git 이 지켜보고 있는 폴더입니다. 평범한 폴더를 한 번만 repo 로 바꿔주면, 그때부터 Git 이 그 안의 모든 것을 추적합니다.

커밋은 하나의 저장 지점입니다. 특정 시점의 모든 파일을 담은 스냅샷에, 무엇이 바뀌었는지 적은 짧은 메시지가 더해진 것입니다 (“로그인 페이지 추가”, “고장난 회원가입 버튼 수정”). 커밋은 끊임없이 만들게 됩니다—각각을 되돌아가고 싶을 수 있는 체크포인트라고 생각하세요.

이력은 모든 커밋을 순서대로 나열한 목록입니다. 곧 타임라인이죠. 쪽 훑어보며 각 변경이 무엇이었는지 읽고, 어느 지점으로든 시간 여행을 할 수 있습니다.

실제로는 이렇게 생겼습니다. 외울 필요는 없습니다—대부분 AI 에게 시켜서 실행하게 될 테니까요—하지만 단어들이 더 이상 수수께끼가 아니도록 같이 읽어 보세요:

```
# 현재 폴더를 Git 리포지토리로 만들기 (한 번만)
git init

# 마지막 저장 지점 이후 어떤 파일이 바뀌었는지 확인
git status

# 변경 사항을 스테이징한 뒤, 메시지와 함께 커밋으로 저장
git add .
```

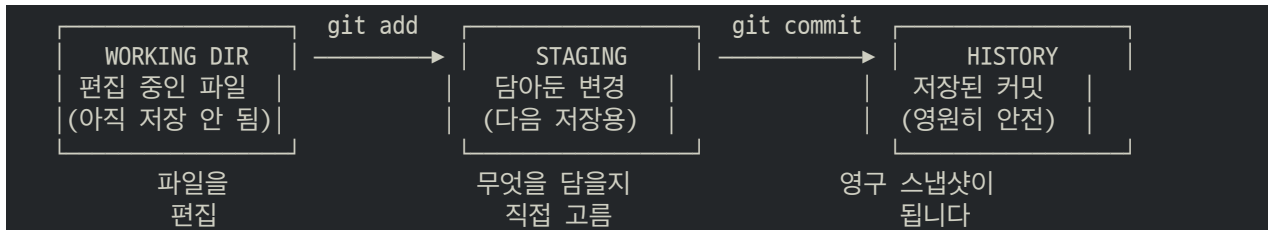
```
git commit -m "Add the homepage layout"
```

```
# 이력 보기 —모든 커밋, 최신순
```

```
git log --oneline
```

git add . 는 “내 모든 변경을 다음 저장 지점에 포함시켜줘”라는 뜻입니다. git commit 이 실제 저장이고요. -m “...” 은 미래의 나에게 남기는 메모입니다. 한 달 뒤에도 이해할 수 있는 메모를 쓰세요.

이 두 명령은 작업을 세 단계로 옮깁니다. 편집한 파일은 맨 오른쪽까지 도달해야 비로소 저장됩니다:



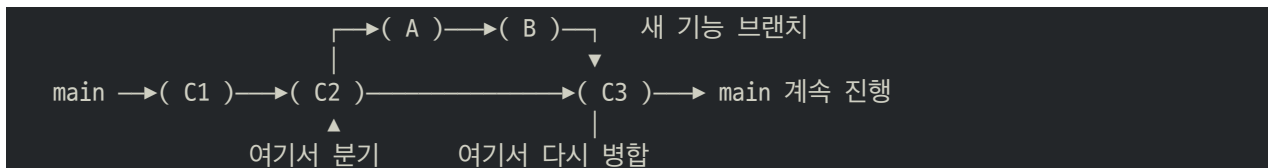
브랜치: 두려움 없이 시도하기

브랜치는 실험할 수 있는 프로젝트의 평행 사본입니다. 주된 작업은 보통 main 이라 불리는 브랜치에 있습니다. 무언가 시도하고 싶을 때—새 기능이든, 위험한 AI 재작성이든—새 브랜치를 만들어 거기서 작업하면, main 은 손대지 않은 안전한 상태로 남습니다.

실험이 잘 되면 main 에 병합합니다. 잘 안 되면 브랜치를 버리면 그만이고, main 은 그런 일이 있었는지조차 모릅니다.

이것이 큰 AI 변경을 다루는 정확한 방식입니다. “결제 흐름 전체를 만들어줘”는 자신만의 브랜치를 가질 자격이 있습니다. AI 가 괜찮은 걸 내놓으면 병합하세요. 재앙을 내놓으면 브랜치를 지우면 됩니다. 잃은 것은 아무것도 없습니다.

그림으로 보면, 브랜치가 main 에서 갈라져 나와 따로 자란 뒤 다시 합쳐집니다:



```
# 새 브랜치를 만들고 그쪽으로 전환
```

```
git checkout -b new-feature
```

```
# ... 작업하고, 커밋을 만들고...
```

```
# 안전한 main 브랜치로 돌아가기
```

```
git checkout main
```

```
# 브랜치의 좋은 작업을 main 에 병합
```

```
git merge new-feature
```

실제로 무엇이 바뀌었는지 보기 (diff)

변경을 저장하기 전에—그리고 특히 AI 가 만든 변경을 받아들이기 전에—정확히 무엇이 달라졌는지 보고 싶을 겁니다. 그게 **diff** 입니다. 한 줄 한 줄, 무엇이 추가되었고 (보통 초록색) 무엇이 제거되었는지 (보통 빨간색) 보여줍니다.

이건 가장 중요한 습관 중 하나입니다. 바이브 코딩의 철학 전체가 내가 출시하는 것을 이해하라는 겁니다. 코드를 직접 타 이핑할 필요는 없지만, 무엇이 바뀌었는지 보고 대략 이해는 해야 합니다. diff 가 바로 그걸 하는 곳입니다.

```
# 변경했지만 아직 커밋하지 않은 내용 보기
git diff

# 스테이징되어 커밋 준비가 된 내용 보기
git diff --staged
```

AI 가“끝났어요, 파일 업데이트했어요”라고 하면, 다음 동작은 diff 를 보는 것입니다. 필요했던 무언가를 지운 게 보이면, 지금 잡습니다—출시된 뒤가 아니라.

“이런”버튼: undo, revert, reset

이 부분이 Git 을 귀찮은 일에서 초능력으로 바꿔줍니다. 무언가 잘못되었을 때—분명 그럴 겁니다—이렇게 되돌립니다.

- **저장하지 않은 변경 버리기**—여러분 (또는 AI) 이 파일을 편집해 상황을 악화시켰지만 아직 커밋하지 않았을 때. 마지막으로 멀쩡했던 커밋으로 전부 되돌릴 수 있습니다.
- **커밋 되돌리기 (revert)**—변경을 저장했는데 나쁜 것이었음을 깨달았을 때. git revert 는 그 나쁜 커밋을 취소하는 새 커밋을 만들어, 이력을 정직하게 유지합니다.
- **이전 지점으로 리셋 (reset)**—프로젝트를 여러 커밋 전의 모습으로 되감고 싶을 때.

```
# 커밋하지 않은 모든 변경을 취소하고 마지막 커밋으로 돌아가기
git restore .

# 특정 나쁜 커밋을 안전하게 취소 (새 " 취소" 커밋을 만들)
git revert <commit-id>

# 이전 커밋으로 되감기 (강력함 —사용 전 AI 에게 설명을 요청하세요)
git reset --hard <commit-id>
```

한 가지 주의: git reset --hard 는 작업을 영구적으로 버립니다. 가끔은 맞는 도구지만, 조심해야 할 도구입니다. 확신이 안 서면 실행 전에 AI 에게“이 명령이 내 파일에 정확히 무슨 일을 하니?”라고 물어보세요.

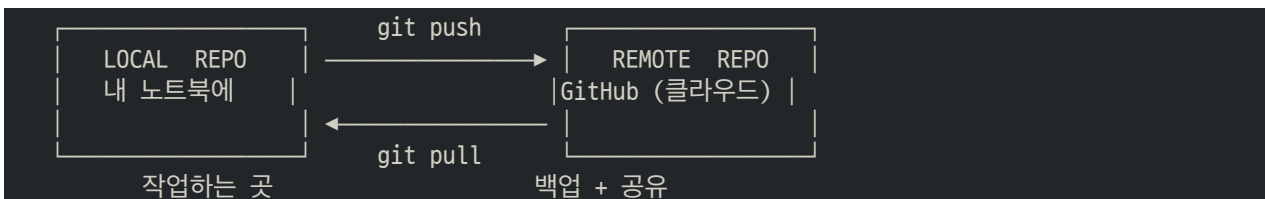
리모트와 GitHub: 클라우드 속 내 프로젝트

지금까지의 모든 일은 여러분의 컴퓨터에서 일어납니다. **GitHub** 은 여러분의 repo 사본을 클라우드에 저장하는 웹사이트입니다. 그 사본을 **리모트**라고 부릅니다.

왜 굳이? 세 가지 이유입니다. 노트북이 죽었을 때의 백업이고, 다른 사람 (또는 AI 도구와 배포 서비스) 과 코드를 공유하는 방법이며, 협업이 일어나는 곳입니다.

로컬 repo 를 리모트에 한 번 연결한 뒤, **push**(커밋을 GitHub 로 올리기) 와 **pull**(GitHub 에서 커밋을 내려받기) 을 합니다.

프로젝트 사본 두 개—하나는 노트북에, 하나는 클라우드에—가 push 와 pull 로 동기화된다고 생각하세요:



```
# 로컬 repo 를 GitHub repo 에 연결 (한 번만)
git remote add origin https://github.com/yourname/yourproject.git

# 커밋을 GitHub 로 올리기
git push origin main
```

```
# GitHub 에는 있지만 내 컴퓨터에는 없는 커밋을 내려받기
git pull origin main
```

push 는 “클라우드에 저장”, pull 은 “클라우드에서 최신 가져오기”라고 생각하세요.

풀 리퀘스트와 이슈

금방 마주치게 될 두 가지 GitHub 기능입니다.

풀 리퀘스트 (PR) 는 한 브랜치를 다른 브랜치에 병합하자는 제안으로, 들어가기 전에 변경을 검토할 수 있는 공간이 함께 달려 있습니다. 혼자 일할 때도 PR 은 유용합니다. 모든 변경을 보여주는 하나의 깔끔한 화면을 제공해, main 에 병합하기 전에 AI 의 작업을 검토할 수 있게 해줍니다. diff 습관을 공식화한 것이죠.

이슈 는 해야 할 일이나 고칠 일—버그, 기능 아이디어, 할 일 (TODO) —에 대한 추적 메모입니다. 이슈는 할 일 목록을 머릿속 여기저기가 아니라 프로젝트 자체에 붙여두는 방법입니다.

.gitignore 와 절대 비밀을 커밋하지 않기

어떤 파일은 절대 Git 에 들어가면 안 됩니다. 대표적인 것들:

- **비밀**—API(애플리케이션 프로그래밍 인터페이스) 키, 비밀번호, 데이터베이스 자격 증명. 비밀을 GitHub 에 커밋했다면, 나중에 지워도 유출된 것으로 취급하세요. 이력은 기억합니다.
- **쓰레기**—임시 파일, 내려받은 의존성 (node_modules 같은), 빌드 결과물. 거대하고 추적할 이유가 없습니다.

이건 .gitignore 라는 파일로 제어합니다. 그 안에 패턴을 적으면, Git 은 그 파일들이 존재하지 않는 척합니다. 이것은 바이브 코더에게 매우 중요합니다. AI 도구는 진짜 키가 든 설정 파일을 즐겨 만들고, 그런 걸 실수로 커밋하기가 너무 쉽기 때문입니다.

```
# 간단한 .gitignore 파일
.env
node_modules/
*.log
```

.env 줄이 핵심입니다—비밀은 보통 거기에 들어 있습니다. 첫 커밋 전에 .gitignore 를 추가하고, AI 에게 “이 커밋에 공개되면 안 될 게 있어?”라고 묻는 습관을 들이세요.

첫 repo, 처음부터 끝까지

전체 흐름을 한곳에 모았습니다. 폴더를 만들고, repo 로 만들고, 작업을 저장하고, GitHub 에 push 하기:

```
# 1. 새 프로젝트 시작
mkdir my-first-project
cd my-first-project
git init

# 2. 무엇보다 먼저 비밀을 보호
echo ".env" > .gitignore

# 3. 첫 저장 지점 만들기
git add .
git commit -m "Initial commit"

# 4. GitHub 에 연결하고 push
git remote add origin https://github.com/yourname/my-first-project.git
git push -u origin main
```

이게 전부입니다. 이제 안전망과 클라우드 백업을 갖춘 프로젝트가 생겼습니다. 진전이 있을 때마다 `add` `commit` `push` 리듬을 반복하세요.

AI 에게 운전대를 맡기되 (여전히 내가 책임지기)

여기 해방감을 주는 부분이 있습니다. 이 명령들을 외울 필요가 전혀 없습니다. AI 에게 “적절한 메시지로 이거 커밋해줘” 또는 “새 기능용 브랜치를 만들고 전환해줘”라고 말하면, AI 가 알맞은 명령을 실행합니다. 그것도 충분히 좋은 작업 방식입니다.

다만 함정은—이 책 나머지와 똑같은 함정인데—직접 타이핑하지 않았더라도 무엇을 했는지 이해할 책임은 여전히 여러분에게 있다는 점입니다. 실제로 통하는 중간 길:

- **AI 에게 명령을 실행하게 하되**, 처음 몇 번은 각 명령이 무엇을 하는지 쉬운 말로 설명해 달라고 요청하세요.
- **diff 는 직접 읽으세요**, 커밋 전에 반드시. 타협 불가입니다. AI 가 지우면 안 될 것을 지우는 걸 잡는 방법입니다.
- **자주, 작게 커밋하세요**. 작은 저장 지점 여러 개가 거대한 하나보다 낫습니다. 되감기가 전부-아니면-전무가 아니라 정밀해지니까요.
- **큰 작업 전에는 브랜치를 만드세요**. 대대적인 재작성 전에 AI 에게 브랜치를 만들라고 해서 main 을 안전하게 두세요.
- **명령이 파괴적으로 들릴 때** (`reset --hard`, 강제 push, 브랜치 삭제 같은 것), 승인하기 전에 “이게 정확히 뭘 하는 거야?”라고 물어보세요.

여러분이 테크 리더입니다. AI 는 빠르고 유능하지만 가끔 무모한 팀원이고요. Git 은 그 팀원이 진짜 피해를 입히지 못하게 막아주는 프로세스입니다. 자신 있게 지휘할 만큼 Git 을 잘 익혀두면, “전부 다시 써”버튼이 더는 두렵지 않을 겁니다—언제나 돌아갈 길이 있을 테니까요.

정리와 실습

핵심 요약

- Git 은 **프로젝트의 저장 지점**입니다—언제든 되돌아갈 수 있는 완전한 타임라인이고, 덕분에 두려움 없이 AI 를 풀어놓을 수 있습니다.
- 핵심 리듬은 `add` `commit` `push` 입니다: 변경 사항을 stage 하고, 명확한 메시지를 단 commit 으로 저장하고, 백업과 공유를 위해 GitHub 에 올립니다.
- **큰 작업 전에는 브랜치를 만들고, 커밋할 때마다 diff 를 읽으세요**—AI 가 지우면 안 될 것을 지우는 걸 잡아내는 방법입니다.
- `.gitignore` 는 비밀 (`.env` 같은 것) 과 쓰레기 (`node_modules/` 같은 것) 를 히스토리에서 빼냅니다. GitHub 에 커밋된 비밀은 나중에 지워도 영원히 유출된 것입니다.
- AI 에게 명령을 실행시켜도 되지만, 각 명령이 무엇을 했는지 이해할 책임은 여러분에게 있습니다—특히 `reset --hard` 같은 파괴적인 명령은요.

직접 해보기

완전히 새 폴더를 만들고 전체 흐름을 손으로 직접 돌려보세요: `git init`, `.env` 가 들어간 `.gitignore` 만들기, 간단한 `notes.txt` 파일 추가, 그다음 `git add .` 와 `git commit -m "Initial commit"`. 이제 `git log --oneline` 을 실행해 저장 지점을 보고, 파일을 편집한 뒤 `git diff` 를 실행해 Git 이 무엇이 바뀌었는지 정확히 보여주는 걸 지켜보세요. 아무것도 잘못될 수 없는 프로젝트에서 이 장의 모든 핵심 습관을 방금 다 써본 것입니다.

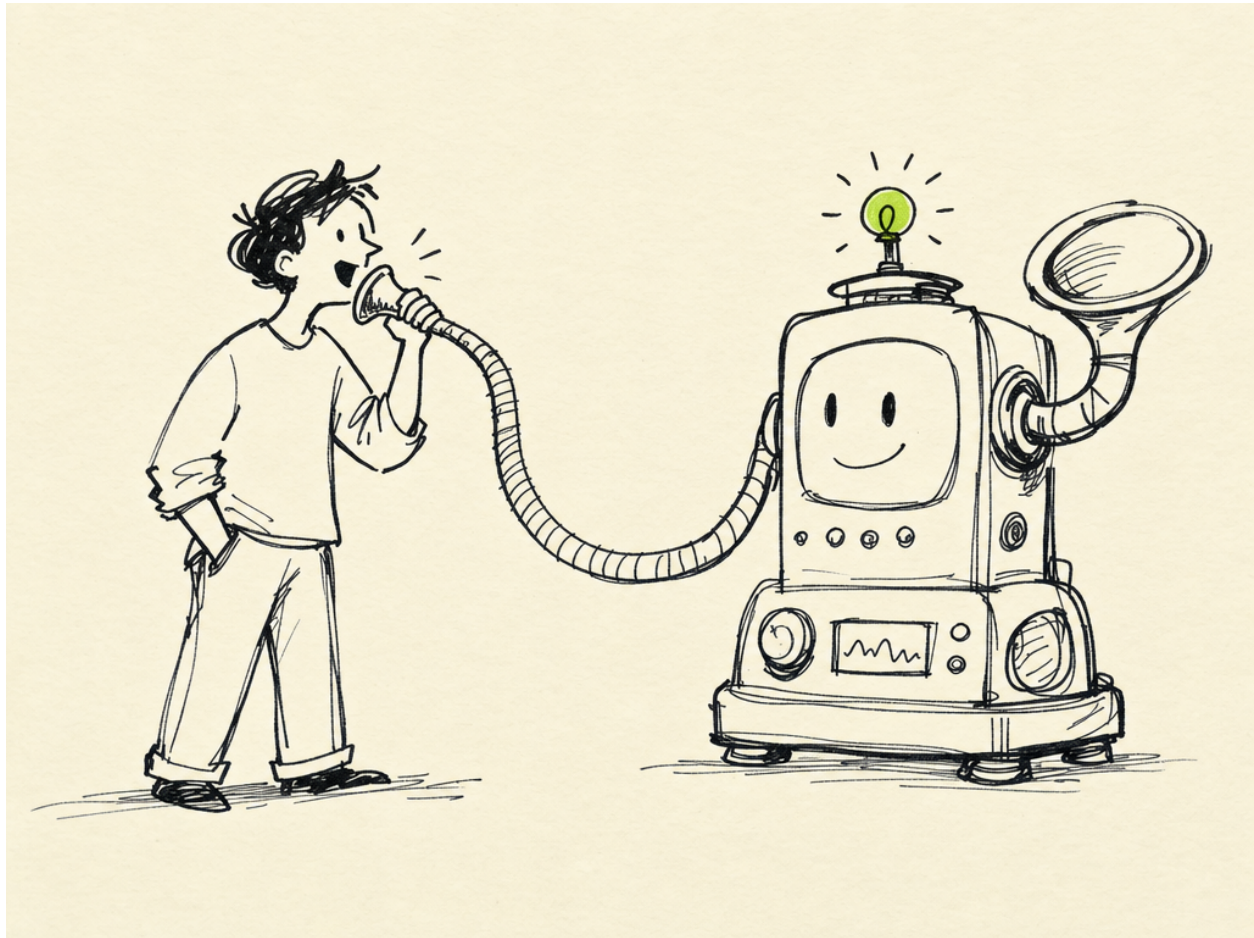
이 장의 프롬프트

나는 Git 을 처음 배우는 초보자이고, 네가 명령을 대신 실행해 주되 실행하면서 가르쳐 주면 좋겠어. 아래 작업에 대해:
< 하고 싶은 일을 설명 —예: " 지금까지의 작업 저장하기" 또는 " 위험한 변경을 안전하게 시도하기">

- 어떤 Git 명령을 실행할지, 각 명령이 무슨 일을 하는지 쉬운 말로 알려줘.
- 파괴적인 명령이라면 (`reset --hard`, 강제 push, 브랜치 삭제 같은 것), 실행 전에 분명히 경고하고 내가 무엇을 잃게 되는지 설명해줘.

- 어떤 커밋이든 하기 전에, diff 를 보여주고 무엇이 바뀌었는지 요약해줘.
- 변경 안에 공개되면 안 될 게 있으면 (비밀, 키) 알려줘.

The Command Line



AI 코딩 도구가 작동하는 모습을 처음 보면, 검은 창에 텍스트가 위로 스크롤되며 지나가고, 깜빡이는 커서가 당신이 입력하기를 기다린다. 그 창이 바로 커맨드 라인이다. 코딩을 해본 적 없는 많은 사람에게 이것은 여정 전체에서 가장 무서운 부분이다. 영화에서 본 것처럼, 한 번만 잘못 누르면 하드 드라이브가 통째로 날아갈 것처럼 보인다. 그렇지 않다. 커맨드 라인은 그저 컴퓨터에게 무엇을 할지 알려주는 또 다른 방식일 뿐이다. 클릭 대신 글자로 말이다.

이 장은 그 두려움을 걷어낸다. 백 개의 명령어를 외울 필요는 없다. 무슨 일이 일어나고 있는지 이해하고, 매일 쓰는 몇 안 되는 명령어를 알아보고, AI 가 건네주는 명령어를 안전하게 실행하는 법만 알면 된다. 그 정도면 두려움을 멈추고 효과적으로 일하기 시작하기에 충분하다.

터미널과 shell 이 실제로 무엇인가

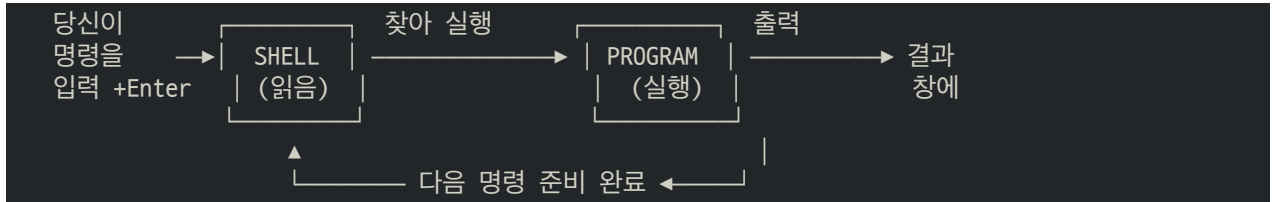
평범한 앱을 쓸 때, 당신은 버튼을 클릭하고 앱은 그 클릭을 컴퓨터용 명령어로 번역한다. 커맨드 라인은 버튼을 건너뛴다. 당신이 명령을 입력하고 Enter 를 누르면 컴퓨터가 그것을 실행한다.

자주 듣게 될 두 단어가 있다:

- **터미널 (Terminal)** —당신이 입력하는 창. 그저 기록이 남는 텍스트 상자다. (Mac 에서는“Terminal”이나“iTerm”, Windows 에서는“PowerShell”이나“Windows Terminal”, Linux 에서는 보통 그냥“Terminal”이다.)

- **shell** —그 창 안에서 당신이 입력한 것을 읽고 실행하는 프로그램. 가장 흔한 것은 **bash** 라 불리고, Mac 은 **zsh** 라는 것을 쓴다. 이 책에서 다루는 모든 것에 대해 둘은 거의 똑같이 동작한다.

Enter 를 누를 때마다 일어나는 순환은 이렇다: shell 이 당신이 입력한 것을 읽고, 일치하는 프로그램을 찾아 실행하며, 그 프로그램이 만들어낸 무엇이든 창에 다시 출력한다.



AI 코딩 도구가 왜 여기에 사는가? 커맨드 라인이 개발의 보편적인 제어판이기 때문이다. 도구 설치, 앱 실행, 테스트 돌리기, 인터넷에 배포하기—이 모든 것이 입력된 명령을 통해 일어난다. AI 에이전트는 당신이 입력할 똑같은 명령을 입력하며 작동한다. 다만 더 빠르고 오차가 없을 뿐이다. 에이전트가 일하는 것을 볼 때, 당신은 그것이 당신도 쓰게 될 바로 그 문을 사용하는 모습을 보고 있는 것이다.

프롬프트 읽기

무언가를 입력하기 전에, shell 은 당신에게 **프롬프트 (prompt)** —어떤 기호로 끝나는 짤막한 텍스트—를 보여주며 입력을 기다린다. 이렇게 생겼을 수 있다:

```
mini@laptop ~/projects/my-app $
```

왼쪽에서 오른쪽으로 읽으면 이렇게 알려준다: 당신의 사용자 이름 (mini), 당신의 컴퓨터 (laptop), 당신이 현재 있는 위치 (~/projects/my-app 폴더), 그리고 “당신 차례”라는 뜻의 \$. ~ 는 당신의 홈 폴더를 가리키는 약칭이다. 끝의 기호는 흔히 \$ 나 % 이고, 때로는 > 이기도 하다. 그 기호는 입력하지 마라—그저 shell 이 준비됐다고 말하는 것일 뿐이다.

이건 보이는 것보다 중요하다. 커맨드 라인에서 겪는 혼란의 대부분은 내가 어디에 있는지 모르는 데서 온다. 프롬프트는 매번 그것을 알려준다.

매일 쓰는 명령어들

여기 당신이 끊임없이 보게 될 명령어들이 있다. 문장처럼 읽어보라: 동사, 그리고 가끔 그 뒤에 따라오는 대상.

```

pwd                # "print working directory" —나는 지금 어디 있는가?
ls                 # 여기 있는 파일과 폴더를 나열한다
ls -la             # 숨김 파일까지 전부, 상세히 나열한다
cd projects        # change directory — "projects" 폴더 안으로 들어간다
cd ..              # 폴더 하나 위로 올라간다 (".." 는 " 한 단계 위 폴더" 를 뜻한다)
cd ~               # 홈 폴더로 돌아간다
mkdir my-app       # "my-app" 이라는 새 폴더를 만든다
cp notes.txt backup.txt # 파일을 복사한다 (원본은 그대로, 복사본이 만들어진다)
mv old.txt new.txt  # 파일을 옮기거나 이름을 바꾼다
rm draft.txt        # 파일을 삭제한다 (remove)
  
```

진짜 고생을 덜어줄 몇 가지 메모:

당신의 파일은 중첩된 폴더의 트리 안에 살고, cd 는 그 트리를 걷는 방법이다. 여기 작은 예가 있다. pwd 는 당신이 my-app 으로 cd 했을 때 서 있게 될 위치를 보여준다:

```

/                               ← 맨 위 ("root")
├── Users/
│   ├── mini/                  ← 당신의 홈 폴더 (~ 약칭)
│   │   ├── projects/
│   │   │   ├── my-app/       ← 여기로 cd → pwd 결과: /Users/mini/projects/my-app
│   │   │   └── src/
  
```



```
| └─ notes.txt
└─ old-app/
```

- **cd 는 이동하는 방법이다.** pwd, ls 와 합치면, 당신이 평생 하게 될 일의 80% 다. 올바른 폴더로 들어가서, 둘러보고, 무언가를 실행한다.
- **경로는 상대 경로일 수도, 절대 경로일 수도 있다.** cd projects 는 현재 위치 기준의 상대 경로이고, cd /Users/mini/projects 는 맨 위에서부터의 전체 주소다.
- **rm 은 영구적이다.** 커맨드 라인에는 휴지통이 없다. 파일을 rm 하면, 사라진다—“정말입니까?”도 없고, 되돌리기도 없다. 폴더와 그 안의 모든 것을 묻지도 않고 삭제하는 rm -rf 는 특히 조심하라. rm 명령이 무엇을 삭제할지 조금이라도 확신이 안 서면, 실행하지 마라. 먼저 AI 에게 설명을 청하라.

프로그램과 dev 서버 실행하기

소프트웨어를 만든다는 것은 그것을 실행한다는 뜻이다. 대부분의 경우 프로그램 이름을 입력하고 그 뒤에 하고 싶은 것을 적게 된다.

```
node app.js      # Node 로 JavaScript 파일을 실행한다
python script.py # Python 파일을 실행한다
npm run dev      # 프로젝트의 개발 서버를 시작한다
```

마지막 것은 특별하다. **dev 서버**는 당신의 앱을 당신 컴퓨터에서 돌려서 http://localhost:3000 같은 주소의 브라우저로 볼 수 있게 해주는 프로그램이다. 꼭 이해해야 할 핵심은 이것이다: dev 서버는 **계속 돌아간다**. 터미널이 “멈춘” 것처럼 보일 것이다—새 프롬프트를 주지 않고 대신 활동 내역이 줄줄이 표시된다. 그건 멈춘 게 아니라 일하고 있는 것이다. 그 명령이 돌아가는 동안 당신의 앱은 살아 있다. 브라우저를 열고, 명령이 출력한 주소로 가면, 당신의 앱이 보일 것이다.

다 끝나면, 그것을 멈춘다 (다음 절에서).

출력과 exit 코드 읽기

명령이 실행된 뒤에는 보통 무언가를 출력한다. 그 출력을 읽는 법을 익히는 것은 진짜 기술이다. 일이 잘됐는지를 당신과 AI 가 알아내는 방법이 바로 이것이기 때문이다.

- **정상 출력**은 그저 정보다—나열된 파일들, 진행 메시지, 서버가 시작됐다는 알림.
- **에러**는 보통 대놓고 말해준다: Error:, command not found, permission denied, cannot find module. 콜론 뒤의 단어들이 단서다. 에러 전체를 복사해 AI 에게 붙여 넣어라—무언가 망가졌을 때 건넬 수 있는 가장 유용한 단 하나다.

뒤에서는, 모든 명령이 보이지 않는 **exit 코드**로 끝난다: 0 은 성공을, 그 외의 값은 문제를 뜻한다. 손으로 확인할 일은 드물지만, 언급되는 것을 듣게 될 것이다. 마지막 코드는 이렇게 볼 수 있다:

```
echo $?      # 마지막 명령의 exit 코드를 출력한다 (0 = 정상)
```

명령이 조용히 실패했는데 확신이 안 서면, echo \$? 가 정말 성공했는지를 알려준다.

Ctrl-C 로 프로세스 멈추기

그 dev 서버는 당신이 멈출 때까지 계속 돌아간다. 보편적인 “멈춤”은 **Ctrl-C** 다—Control 키를 누른 채 C 를 누른다. 돌아가고 있는 무엇에게든 “지금 멈춰 주세요”라는 정중한 신호를 보내고, 당신은 프롬프트를 돌려받는다.

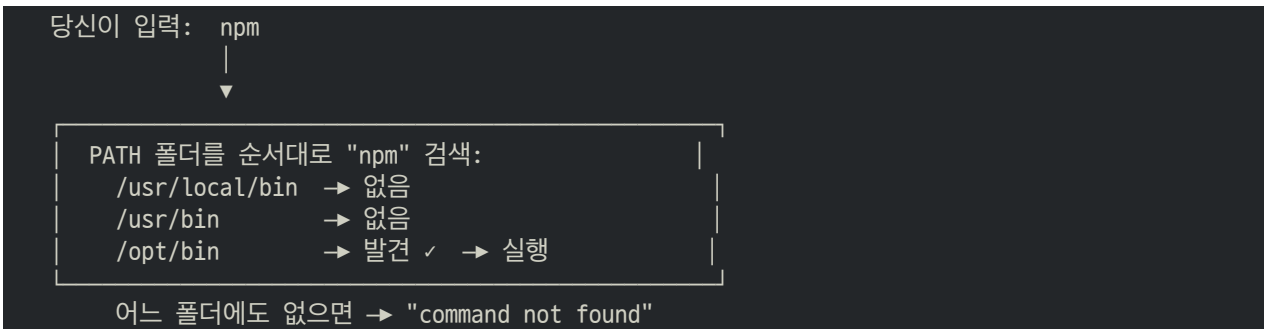
- **Ctrl-C** —이 터미널에서 지금 돌아가는 프로그램을 멈춘다. dev 서버를 끝내거나 너무 오래 걸리는 명령을 취소할 때의 단골 수단이다.
- **터미널 창을 닫는 것도** 그 안에서 돌아가는 모든 것을 멈추지만, Ctrl-C 가 더 깔끔하다.

Ctrl-C 를 기억하면 “멈춘” 터미널에 갇힌 기분을 다시는 느끼지 않을 것이다.

PATH 와 “command not found”

언젠가는 명령을 입력했는데 command not found 가 뜰 것이다. 이건 당신이 멍청해서가 거의 아니다. 그 프로그램이 어디에 있는지 컴퓨터가 모른다는 뜻이다.

shell 은 **PATH** 라 불리는 폴더 목록을 확인하며 프로그램을 찾는다. 당신이 npm 을 입력하면, shell 은 PATH 안의 각 폴더를 돌며 npm 이라는 프로그램을 찾는다. 그중 어디에도 없으면 command not found 가 뜬다.



```
echo $PATH # shell 이 검색하는 폴더 목록을 보여준다
```

실제로 command not found 는 보통 둘 중 하나를 뜻한다: 그 도구가 아직 설치되지 않았거나, PATH 에 없는 어딘가에 설치됐거나. 둘 다 흔하고 고칠 수 있다. 에러를 AI 에게 붙여 넣고 무엇을 실행하려 했는지 말하라—PATH 문제를 고치는 건 AI 가 차근차근 안내해주기에 딱 좋은 종류의 자잘한 설정 작업이다.

AI 가 주는 명령어를—안전하게—실행하기

바이트 코딩의 큰 부분은 AI 가 “이 명령을 실행하세요”라며 붙여 넣을 한 줄을 건네는 것이다. 이건 정상이고 괜찮다. 다만 당신을 영원히 지켜줄 습관 하나를 들여라: **실행하기 전에 읽어라.**

모든 플래그를 이해할 필요는 없다. 그 명령이 무엇을 하는지, 특히 맨 앞의 동사에 대한 대략적인 감만 있으면 된다:

- ls, pwd, cd, cat, echo —무해하다. 무언가를 들여다보거나 이동할 뿐이다. 자유롭게 실행하라.
- npm install, pip install —도구를 다운로드하고 설정한다. 대체로 안전하지만, 당신의 프로젝트를 바꾼다.
- rm, mv, sudo 가 붙은 것 무엇이든, 다운로드를 공장 shell 로 흘려보내는 것 무엇이든 (curl ... | bash) —**속도를 늦춰라.** 이것들은 무언가를 삭제하거나 당신의 기계에 대한 완전한 권한으로 코드를 실행할 수 있다.

명령이 무섭게 보이거나 동사를 알아볼 수 없으면 물어라: “이 명령은 무엇을 하고, 실행해도 안전한가요?” 좋은 AI 는 그것을 쉽게 설명해줄 것이다. 그 확인 없이는 아무 웹사이트의 명령도 붙여 넣지 마라. 커맨드 라인은 시키는 대로 정확히 한다—위험한 것까지 포함해서—그러니 먼저 읽는 작은 습관이 당신의 안전벨트다.

만나게 될 패키지 매니저

현대의 소프트웨어는 미리 쓰여진 작은 조각 수천 개로 만들어진다. **패키지 매니저 (package manager)** 는 그 조각들을 다운로드하고 정리하는 도구다. 끊임없이 둘을 만나게 될 것이다:

```
npm install # 이 JavaScript 프로젝트가 필요로 하는 모든 것을 설치한다
npm install react # "react" 라는 특정 패키지를 추가한다
pip install requests # "requests" 라는 Python 패키지를 설치한다
```

- **npm**(노드 패키지 매니저) 은 JavaScript 와 Node 프로젝트를 위한 패키지 매니저다. npm install(흔히 npm i 로 표기) 은 프로젝트가 무엇을 필요로 하는지 적힌 파일을 읽고 그것을 전부 다운로드한다.
- **pip** 은 Python 을 위해 같은 일을 한다.

프로젝트를 처음 열 때 npm install 을 한 번 돌리고 나서는 잊고 지내는 경우가 많다. AI 가 새 패키지를 필요로 하는 기능을 추가하면, 그것을 설치하라고 알려줄 것이다. 이제 당신은 그게 무슨 뜻인지 안다: 그저 building block 하나를 가져오는 것이지, 두려워할 무엇이 아니다.

이것이 커맨드 라인이다. 해커의 무기가 아니라—그저 당신이 청하는 그대로를 하는 텍스트 상자다. 프롬프트를 읽고, 열두어 개의 명령어를 알아보고, Ctrl-C 로 폭주하는 프로세스를 멈추고, 실행하기 전에 읽는 법을 익혀라. 그 외의 모든 것은, 물어보면 된다.

정리와 실습

핵심 요약

- 명령줄은 그저 시키는 대로 정확히 실행하는 텍스트 상자입니다—클릭 대신 단어죠. AI 에이전트는 여러분이 칠 법한 바로 그 명령들을 타이핑하며 작동합니다.
- **프롬프트**는 매번 지금 어디 있는지 알려줍니다. 혼란의 대부분은 현재 폴더를 모르는 데서 옵니다. `pwd`, `ls`, `cd` 가 여러분이 할 일의 80% 입니다.
- **rm** 은 **영구적입니다**—휴지통도, 실행 취소도 없습니다. `rm`, `sudo`, 그리고 다운로드를 셀로 흘려보내는 것 (`curl ... | bash`) 앞에서는 속도를 늦추세요.
- **개발 서버**는 계속 돌아가며 일부러 “멈춘” 것처럼 보입니다—그게 작동 중인 모습입니다. **Ctrl-C** 는 실행 중인 어떤 프로그램이든 멈추고 프롬프트를 돌려줍니다.
- **실행 전에 읽으세요.** 모든 플래그를 알 필요는 없고, 동사의 대략적인 감만 있으면 됩니다. 의심스러우면 AI 에게 “이 명령이 뭘 하고, 실행해도 안전해?”라고 물어보세요.

직접 해보기

터미널을 열고 아무것도 바꾸지 않으면서 탐색해 보세요: `pwd` 로 어디 있는지 보고, `ls` 로 뭐가 있는지 보고, `cd` 로 폴더에 들어가 다시 `ls`, 그다음 `cd ..` 로 다시 올라오세요. 매번 프롬프트가 갱신되는 걸 보세요. 이제 일부러 실패하는 명령을 실행해 보세요—`lss`(오타) 를 치고 `command not found` 메시지를 읽어보세요. 파일 하나 건드리지 않고, 가장 안전하고 가장 많이 쓰는 두 명령을 연습하고 에러 읽는 법까지 익힌 것입니다.

이 장의 프롬프트

저는 커맨드 라인 초보라서 무언가를 실행하는 게 조금 겁이 납니다.
당신 (또는 어떤 튜토리얼) 이 실행하라고 알려준 명령은 이것입니다:
< 여기에 명령을 붙여 넣으세요 >

- 이 명령이 무엇을 하는지 쉬운 말로 한 단계씩 설명해 주세요.
- 이 명령이 안전한지, 아니면 파일을 삭제하거나 제 시스템을 바꾸거나 완전한 권한으로 코드를 실행할 수 있는지 분명하게 알려 주세요.
- 위험하다면, 같은 일을 더 안전하게 하는 방법이나 먼저 확인해야 할 것을 제안해 주세요.
- 실행하기 전에 제가 어느 폴더에 있어야 하는지 알려 주세요.

How the Web Works



웹을 위한 무언가를 만들기 전에, 웹이 실제로 무엇인지 아는 것이 도움이 된다. 컴퓨터 과학 학위가 필요한 것도 아니고, 무언가를 외울 필요도 없다. 그저 작동하는 멘탈 모델 하나면 된다—주소를 입력하면 페이지가 나타나는 그 순간 무슨 일이 벌어지는지를 머릿속에 그린 그림 말이다.

이 챕터는 그 그림을 만든다. 로그인 폼부터 결제까지, 바이트 코딩으로 만든 앱이 하는 모든 일은 결국 같은 단순한 춤의 한 형태다: 한 컴퓨터가 묻고, 다른 컴퓨터가 답한다. 이 춤을 또렷이 볼 수 있게 되면, 이 책의 나머지는 마법처럼 느껴지기를 멈추고 배관 작업처럼 느껴지기 시작한다.

두 등장인물: 클라이언트와 서버

웹의 거의 모든 것은 두 역할 사이의 대화다.

- **클라이언트**는 묻는 쪽이다. 대부분의 경우 그것은 당신의 휴대폰이나 노트북에서 실행되는 웹 브라우저—Chrome, Safari, Firefox —다. 카운터로 다가가는 손님인 셈이다.
- **서버**는 답하는 쪽이다. 어딘가의 데이터 센터에 놓인, 하루 종일 실행되며 요청을 기다리는 컴퓨터다. 카운터 뒤의 주방인 셈이다.

클라이언트는 결코 서버의 일을 하지 않고, 서버도 결코 클라이언트의 일을 하지 않는다. 브라우저는 “홈 페이지를 줘”라고 묻는다. 서버는 “여기 있어”라고 답한다. 이 관계가 전부다. 서버는 특별한 마법 상자가 아니다—그저 질문을 기다렸다가 답을 보내는 것이 일인 평범한 컴퓨터일 뿐이다.

계속 듣게 될 단어가 하나 더 있다: **request(요청)**(클라이언트가 보내는 질문) 와 **response(응답)**(서버가 돌려보내는 답). 이 두 단어를 손에 쥐고 있어야; 아래의 모든 것의 등뼈다.

URL이란 실제로 무엇인가

URL(유니폼 리소스 로케이터)은 당신이 요청하는 주소다. 하나의 긴 문자열처럼 보이지만, 사실은 이름표가 붙은 몇 개의 부분이 이어 붙여진 것이다. 이것을 분해해 보자:

`https://shop.example.com/products/shoes?size=10`

<code>https</code>	<code>scheme</code>	— how to talk (use HTTPS, the secure way)
<code>shop.example.com</code>	<code>host</code>	— which server to talk to
<code>/products/shoes</code>	<code>path</code>	— which page or thing you want from it
<code>?size=10</code>	<code>query</code>	— extra details ("the size-10 ones")

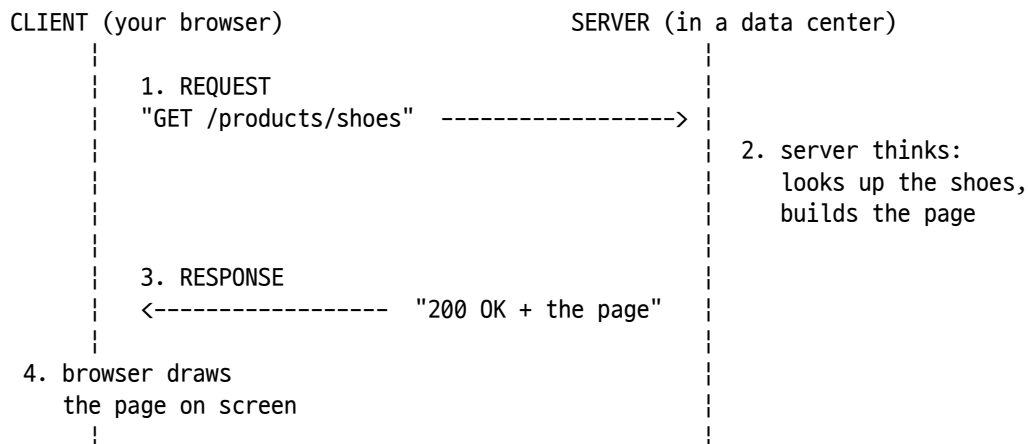
거꾸로 된 우편 주소처럼 읽어라:

- **scheme**(https)는 프로토콜이다—대화의 규칙. 오늘날 거의 항상 https다.
- **host**(shop.example.com)는 어느 서버에 연락할지를 가리킨다.
- **path**(/products/shoes)는 그 서버에서 어느 것을 원하는지를 가리킨다.
- 선택적인 **query**(?size=10)는 필터나 검색어 같은 추가 매개변수를 담는다.

누군가 “엔드포인트”나 “라우트”라고 말하면, 거의 언제나 특정 host의 특정 path를 뜻한다. 그 무서워 보이는 단어들이 뜻하는 건 그게 전부다.

요청과 응답의 순환

여기 그 춤이, 처음부터 끝까지 있다. 당신이 주소를 입력하면, 보이지 않는 곳에서 1초도 안 되는 사이에 이런 일이 벌어진다:



매번 네 단계다:

1. 클라이언트가 URL로 **request**를 보낸다.
2. 서버가 그것을 **처리**한다—무언가를 찾아보거나, 로직을 돌리거나, 당신이 누구인지 확인할 수도 있다.
3. 서버가 **response**를 돌려보낸다: 상태 (작동했는가?)와 약간의 콘텐츠 (페이지, 이미지, 또는 데이터).
4. 클라이언트가 그 응답으로 무언가를 한다—보통은 당신의 화면에 그린다.

웹 페이지 하나를 불러오는 것은 요청 하나가 아니다—수십 개다. 브라우저는 페이지를 요청하고, 그다음 이미지·폰트·스타일이 필요하다는 것을 보고, 각각에 대해 새로운 요청을 쏘아 보낸다. 같은 춤을, 여러 번, 아주 빠르게.

HTTP 메서드: 동사들

모든 요청은 **메서드**를 지닌다—당신이 어떤 종류의 동작을 원하는지 말하는 동사다. 여러 가지가 있지만, 네 개가 거의 전부를 덮고, 일상적인 관념에 깔끔하게 들어맞는다:

- **GET** —“이걸 줘.”읽기, 가져오기, 보기. 페이지나 목록을 불러오기. GET 은 결코 아무것도 바뀌서는 안 된다; 그저 가져올 뿐이다.
- **POST** —“여기 새로운 게 있어.”생성하기. 폼 제출, 가입, 댓글 달기.
- **PUT** —“이걸 갱신해.”프로필 수정처럼 이미 존재하는 것을 바꾸기. (부분 갱신에는 **PATCH** 라는 말도 듣게 된다 —같은 발상, 더 좁은 범위.)
- **DELETE** —“이걸 제거해.”들리는 그대로다—레코드나 파일을 삭제하기.

다른 건 다 잊더라도 이것만은: **GET** 은 읽고, **POST** 는 생성하고, **PUT** 은 갱신하고, **DELETE** 는 제거한다. 앱이 말썽을 부릴 때, 어떤 동사가 보내졌는지 아는 것이 첫 단서인 경우가 많다.

상태 코드: 작동했는가?

모든 응답은 세 자리 **상태 코드**와 함께 돌아온다—요청이 어떻게 됐는지에 대한 서버의 한눈 판정이다. 첫 자리 숫자로 묶인다:

- **2xx** —성공. 작동했다. 200 OK 는“여기 네가 요청한 게 있어”라는 일상적인 응답이다.
- **3xx** —리다이렉트. “여긴 없어, 대신 저쪽을 봐.”흔하고 해롭지 않다; 브라우저가 자동으로 따라간다.
- **4xx** —당신 (클라이언트) 이 실수했다. 유명한 것은 **404 Not Found** —존재하지 않는 path 를 요청했다. (401/403도 있다—로그인하지 않았거나, 허락되지 않았다.)
- **5xx** —서버가 실수했다. **500 Internal Server Error** 는 요청을 처리하던 중 서버 자신의 코드가 충돌하거나 깨졌다는 뜻이다.

4xx/5xx 의 구분은 머리에 새겨둘 가치가 있다. 어디를 봐야 할지를 알려주기 때문이다:

- **404** 는 요청이 잘못됐다는 뜻이다—보통 잘못된 URL 이나 오타 난 path. 무엇이 요청되고 있는지를 보라.
- **500** 은 서버가 쓰러졌다는 뜻이다. 당신의 요청은 멀쩡했다; 백엔드 코드가 깨졌다. URL 이 아니라 서버 로그를 보라.

바이브 코딩으로 만든 앱에서 뭔가 깨졌을 때, “이게 4xx 야 5xx 야?”는 분류에 쓸 수 있는 가장 빠른 질문 중 하나다.

상태 코드			
2xx	→ 성공	작동했다 (200 OK)	
3xx	→ 리다이렉트	" 저쪽을 봐" (자동으로 따라감)	
4xx	→ 클라이언트 오류	당신의 요청이 잘못됐다	→ URL/path 확인
5xx	→ 서버 오류	백엔드가 깨졌다	→ 서버 로그 확인

헤더와 쿠키

본문 콘텐츠와 나란히, 모든 요청과 응답은 **헤더**를 지닌다—추가 정보가 담긴 이름표 붙은 작은 메모들이다. 안에 든 편지와 별개인, 봉투에 적힌 글씨 같은 것이다. 헤더는 콘텐츠가 어떤 형식인지 (Content-Type: text/html), 또는 누가 묻고 있는지 같은 것을 말한다.

쿠키는 헤더의 한 가지 구체적이고 중요한 쓰임이다. 웹은 기억력이 짧다: 기본적으로 각 요청은 홀로 서 있고, 서버는 답하는 순간 당신을 잊는다. 쿠키가 그것을 고친다. 당신이 로그인한 뒤, 서버는 당신의 브라우저에게 작은 토큰—쿠키—을 건네고, 당신의 브라우저는 이후의 모든 요청에 그것을 자동으로 함께 보낸다. 그것이 사이트가 여러 페이지에 걸쳐 당신이 로그인되어 있음을“기억하는”방식이다. 쿠키가 없으면, 기억도 없다; 클릭할 때마다 매번 로그인해야 할 것이다.

HTTPS 와 자물쇠

주소창의 작은 자물쇠를 본 적이 있을 것이다. 그것은 연결이 **HTTPS**(하이퍼텍스트 트랜스퍼 프로토콜 시큐어) 를 쓰고 있다는 뜻이다—**TLS** 라 불리는 암호화 계층으로 감싼 HTTP(하이퍼텍스트 트랜스퍼 프로토콜) 다.

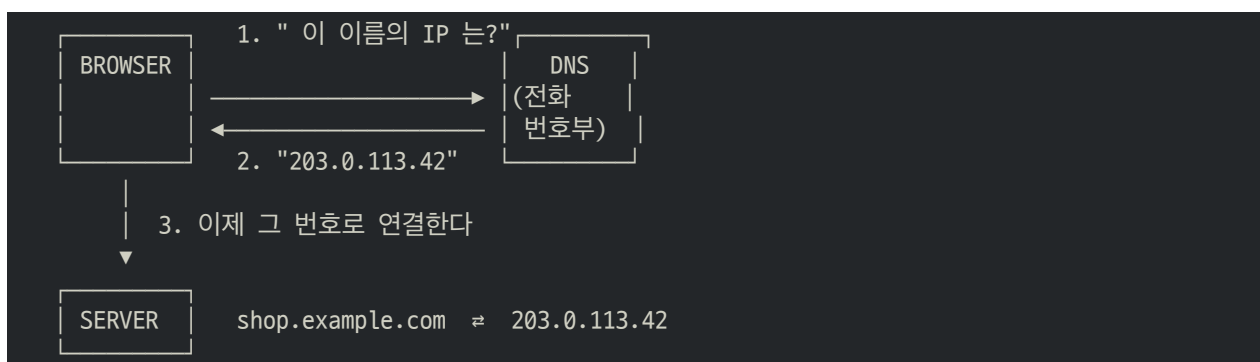
평범한 HTTP 는 모든 것을 읽을 수 있는 텍스트로 보낸다. 당신과 서버 사이에 앉은 누구든—가령 공용 Wi-Fi 에서—당신의 비밀번호가 지나갈 때 그것을 읽을 수 있다. 우편 경로의 누구나 읽을 수 있는 엽서처럼 말이다. HTTPS 는 그 대화를 당신과 서버만 열 수 있는 봉투에 봉인한다. 자물쇠는 두 가지를 확인해 준다:

- **프라이버시**—중간에 있는 누구도 당신이 보내는 것을 읽을 수 없다.
- **신원**—당신은 정말로 당신이 생각하는 그 서버와 이야기하고 있는 것이지, 사칭자가 아니다.

실전 규칙은 짧다: 진짜 앱은 HTTPS 를 쓴다, 항상, 예외 없이. 로그인이나 결제를 평범한 HTTP 로 다루는 것은 설계부터 망가진 것이다.

이름에서 서버로: DNS

마무리되지 않은 한 가닥. 당신은 `shop.example.com` 을 입력하지만, 컴퓨터는 사실 이름으로 서로를 찾지 않는다—`203.0.113.42` 같은 숫자로 된 **IP 주소**를 쓴다. **DNS**(Domain Name System) 는 웹의 전화번호부다: 당신의 브라우저가 요청을 보낼 수 있기 전에, 그것은 조용히 DNS 에게 친근한 **도메인** 이름을 숫자 주소로 번역해 달라고 묻고, 그다음 거기에 연결한다.



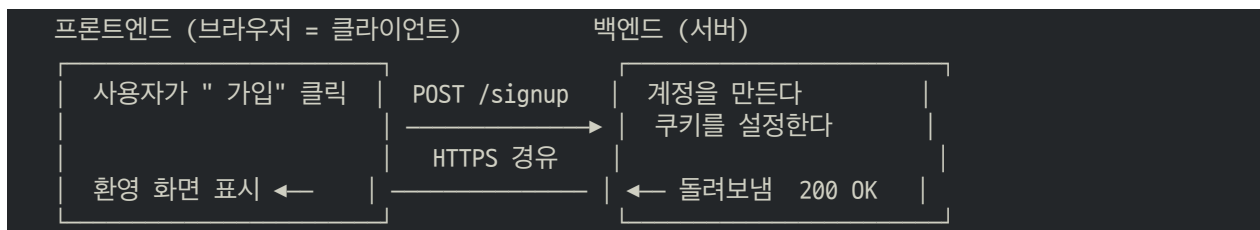
당신은 도메인을 사서, 그것의 DNS 레코드가 당신의 앱을 실행하는 서버를 가리키도록 하고, 그때부터 그 이름은 사람들을 당신의 기계로 이끈다. 거의 볼 일 없는 조회 단계지만, 기억하기 쉬운 이름이 숫자의 나열을 대신할 수 있는 이유가 바로 이것이다.

이것이 당신의 앱에 어떻게 대응되는가

이제 이 그림 전체를 바이트 코딩으로 만든 앱 위에 놓아 보자. 당신이 만들 모든 조각이 이 춤의 한쪽에 살고 있기 때문이다:

- **프론트엔드**는 브라우저에서 실행되는 모든 것이다—사용자가 보고 클릭하는 버튼, 폼, 레이아웃. 그것이 클라이언트다. 대체로 요청을 보내고 응답을 표시한다.
- **백엔드**는 서버에서 실행되는 당신의 코드다—요청을 어떻게 처리할지 결정하고, 데이터베이스와 이야기하고, 응답을 만드는 로직. 그것이 서버다.

가입 여정 전체를 한 장의 그림에 담으면 이렇다—이 챕터의 모든 굵은 단어가 제자리에 놓여 있다:



진짜 상호작용은 그 모든 것을 엮는다. 사용자가“가입”을 클릭한다(프론트엔드). 브라우저는 폼 데이터를 실은 채 `/api/signup` 같은 path 로 **HTTPS** 를 통해 **POST** 요청을 보낸다. 당신의 백엔드가 그것을 받아 계정을 만들고, 사용자가 로그인된 상태를 유지하도록 **쿠키**를 설정하고, **200** 을 돌려보낸다. 프론트엔드는 성공을 보고 환영 화면을 보여준다. 거기서 굵게 표시된 모든 단어는 당신이 방금 배운 것이다—그리고 그것이 당신이 만들 거의 모든 것의 전체 모양이다.

이 모든 것을 한 번에 머릿속에 담고 있을 필요는 없다. 하지만 뒤의 챕터가“요청은 서버에 닿았는데 500 으로 돌아왔다” , 또는“이 헤더를 설정해라”, 또는“이 라우트가 POST 를 처리한다”라고 말할 때, 당신은 이제 그것이 춤의 어느 부분을 가리키는지 정확히 알게 된다. 그 멘탈 모델이 진짜 선행조건이다—그리고 당신은 이제 그것을 가졌다.

정리와 실습

핵심 요약

- 웹은 끝없이 반복되는 하나의 단순한 춤입니다: **클라이언트** (보통 브라우저) 가 **요청**을 보내고, **서버**가 **응답**을 돌려 줍니다. 나머지는 전부 그 위에 얹힌 세부일 뿐입니다.
- **URL** 은 이어 붙인 이름표가 달린 부분들입니다—scheme, host, path, query. 메서드는 동사입니다: **GET** 은 **읽고**, **POST** 는 **만들고**, **PUT** 은 **갱신하고**, **DELETE** 는 **지웁니다**.
- **상태 코드**는 서버의 한눈 판정입니다. 4xx/5xx 구분이 어디를 봐야 할지 알려줍니다: **4xx** 는 요청이 잘못된 것 (잘못된 URL), **5xx** 는 서버 자신의 코드가 깨진 것입니다.
- **HTTPS**(자물쇠) 는 대화를 봉인해 중간의 누구도 읽지 못하게 합니다—진짜 앱은 항상 씁니다. **쿠키**는 여러 요청에 걸쳐 사이트가 로그인 상태를 기억하는 방법입니다.
- **프론트엔드**는 클라이언트 (브라우저에서 실행) 이고, **백엔드**는 서버 (로직을 실행하고 데이터베이스와 대화) 입니다. 여러분이 만드는 모든 기능은 이 춤의 한쪽 편에 있습니다.

직접 해보기

아무 웹사이트나 열고, 브라우저 개발자 도구를 여세요 (우클릭 ▢ “검사”, 그다음“Network”탭). 페이지를 새로고침하세요. 요청 목록이 채워지는 걸 보세요—페이지 요청 하나에 이어 이미지, 폰트, 스타일을 위한 수십 개의 요청이 더 나오는데, 설명한 그대로입니다. 하나를 클릭해 **메서드** (GET), **상태 코드** (잘하면 200), **헤더**를 보세요. 이제 요청/응답 춤이 실시간으로 일어나는 걸, 그것을 읽을 어휘와 함께 보고 있는 것입니다.

이 장의 프롬프트

저는 초보자로서 웹이 어떻게 작동하는지 배우고 있습니다. 제 앱을 예시로 사용해서:

< 당신의 앱을 설명하거나, URL 이나 지금 보고 있는 오류를 붙여넣으세요 >

- 하나의 구체적인 동작 (페이지 불러오기나 폼 제출 같은) 에 대해 요청/응답 순환을 단계별로 차근차근 설명해 주세요.
- 어느 부분이 클라이언트 (프론트엔드) 이고 어느 부분이 서버 (백엔드) 인지 알려주세요.
- 제가 상태 코드나 오류를 보고 있다면, 그것이 4xx(제 요청이 잘못된) 인지 5xx(서버가 깨짐) 인지, 그리고 어디를 봐야 하는지 설명해 주세요.
- 구체적이고 초보자 친화적으로 —가정된 지식 없이 설명해 주세요.

API, JSON 그리고 데이터



당신이 만드는 앱 중 혼자 동작하는 건 거의 없다. 지도를 보여주거나, 카드를 결제하거나, 이메일을 보내거나, 날씨를 조회하려는 순간, 당신의 앱은 다른 누군가의 앱과 대화해야 한다. 그 앱이 쓰는 언어를 **API**(애플리케이션 프로그래밍 인터페이스)라고 부르고, 주고받는 말들은 거의 항상 **JSON**(자바스크립트 오브젝트 노테이션)이라는 형식으로 쓰여 있다. 이 챕터는 그 둘에 관한 것이다—당신이 직접 배관 작업을 손으로 짜기 위해서가 아니라, 그걸 읽고, AI가 만든 걸 점검하고, 돈이 새어나가기 전에 뭔가 잘못됐다는 걸 알아채기 위해서다.

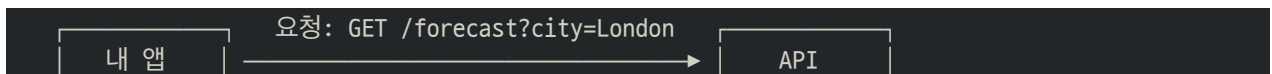
여기서 뭔가를 외울 필요는 없다. 모양을 알아보면 된다. AI가 “Stripe API를 호출해서 JSON 응답을 파싱하겠습니다”라고 말할 때, 그게 대략 무슨 뜻인지, 어디서 어긋날 수 있는지 그림을 그릴 수 있어야 한다.

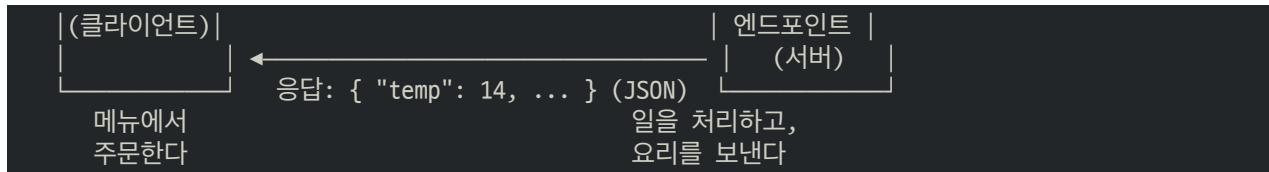
API가 실제로 무엇인가

API를 생각하는 가장 깔끔한 방법은 **한 프로그램이 다른 프로그램에게 제공하는 요청들의 메뉴**라고 보는 것이다.

식당은 당신이 주방에 들어와 요리하게 해주지 않는다. 메뉴판을 건넨다. 여기 주문할 수 있는 것들이 있고, 이렇게 주문하면 되고, 이런 걸 받게 된다고 알려준다. 주방이 어떻게 돌아가는지 알 필요는 없다. 메뉴판만 있으면 된다. API가 정확히 그것이다—회사가 “우리 서비스에 할 수 있는 요청들은 이것이고, 각각이 무엇을 돌려주는지 알려드립니다”라고 말하는 것이다.

AI가 “날씨 API를 호출하는” 코드를 쓸 때, 그건 그 메뉴에서 주문을 넣는 것이다. 이 도시의 예보를 줘. 날씨 회사의 서버가 일을 처리하고 답을 돌려준다. 당신의 앱은 그들의 데이터베이스나 코드를 절대 보지 못한다. 그냥 요리만 받는다.





중요한 사고방식의 전환이 있다. API 는 **계약**이다. 상대 회사는 당신이 특정한 방식으로 요청하면 특정한 모양의 답을 받게 될 거라고 약속한다. 당신의 일—그리고 AI 의 일—은 올바르게 요청하고 답을 올바르게 읽는 것이다. 대부분의 API 버그는 어느 한쪽이 그 계약을 깨뜨리는 것이다.

엔드포인트와 REST, 쉽게 풀어서

엔드포인트는 메뉴의 한 항목이다. 그저 한 가지 특정한 일을 하는 URL(유니폼 리소스 로케이터) 일 뿐이다. 예를 들면,

- `https://api.weather.com/forecast` —예보 가져오기
- `https://api.weather.com/cities` —이용 가능한 도시 목록

같은 주방, 다른 요리. 각 엔드포인트는 하나의 기능이다.

당신이 다루게 될 대부분의 API 는 **REST**(레프리젠테이션얼 스테이트 트랜스퍼) 라고 불리는 느슨한 스타일을 따른다. 학술적인 정의는 필요 없다. 두 가지 개념만 알면 된다.

- **URL 은 어떤 것의 이름을 가리킨다** (하나의 “리소스”) —사용자, 주문, 예보.
- **동사는 그것으로 무엇을 하고 싶은지를 말한다.** 이 동사들은 HTTP(하이퍼텍스트 트랜스퍼 프로토콜) 메서드인데, 웹이-어떻게-동작하는가에서 본 그것들과 같다.
 - GET —무언가를 읽기 (예보 가져오기). 안전하다. 아무것도 바꾸지 않는다.
 - POST —무언가를 생성하기 (주문 넣기, 사용자 가입시키기).
 - PUT / PATCH —이미 존재하는 무언가를 수정하기.
 - DELETE —무언가를 제거하기.

그래서 GET `/orders/42` 는 “42 번 주문을 읽어라”는 뜻이고, DELETE `/orders/42` 는 “그걸 삭제해라”는 뜻이다. 같은 명사, 다른 동사, 완전히 다른 효과. API 문서나 diff 를 읽을 때, 그 두 단어—**메서드**와 **URL** —가 요청이 무슨 일을 하는지 거의 다 알려준다.

JSON 이 무엇인가

당신의 앱이 API 에게 질문을 던지면, 답은 **JSON** 이라는 형식의 텍스트로 돌아온다. JSON 이 기본값이 된 이유는 사람도 읽을 수 있고 동시에 프로그램이 다루기 쉽기 때문이다. 끊임없이 보게 될 테니, 읽을 줄 아는 게 가치가 있다.

JSON 은 **키와 값**으로 만들어진다. 키는 따옴표 안의 라벨이고, 값은 거기에 붙은 데이터다. 다음은 한 사용자를 묘사하는 실제 예시다.

```

{
  "id": 42,
  "name": "Ada Lovelace",
  "email": "ada@example.com",
  "isAdmin": false,
  "loginCount": 137,
  "tags": ["beta", "early-access"],
  "profile": {
    "city": "London",
    "avatarUrl": null
  }
}
  
```

누군가 작성한 양식처럼 읽어보자. “name” 은 항목이고, “Ada Lovelace” 는 거기에 적은 내용이다. 두 가지 구조가 모든 일을 한다.

- **object** 는 중괄호 { } 로 감싸여 있다. 키/값 쌍의 묶음—하나의 레코드 같은 것이다. 위 예시 전체가 하나의 object 다.
- **array** 는 대괄호 [] 로 감싸여 있다. 값들의 순서 있는 목록이다. 위에서 "tags" 는 두 string 의 array 를 담고 있다.

"profile" 이 object 안의 object 라는 점에 주목하자. JSON 은 중첩된다—값 자체가 object 나 array 일 수 있고, 필요한 만큼 깊이 들어갈 수 있다. 그래서 하나의 응답이 사용자와 그의 주소, 그리고 최근 열 개의 주문을 한꺼번에 묘사할 수 있는 것이다.

같은 사용자 object 를 트리로 그리면 그 모양이 한눈에 보인다—위에는 평평한 항목들, 그리고 거기서 목록 하나와 중첩된 object 가 가지처럼 뻗어 나간다.

```
{ } user (객체)
├─ id ..... 42 (숫자)
├─ name ..... "Ada Lovelace" (문자열)
├─ isAdmin ..... false (불리언)
├─ tags ..... [ ] (배열)
│   └─ "beta"
│   └─ "early-access"
└─ profile ..... { } (객체)
    ├─ city ..... "London" (문자열)
    └─ avatarUrl ..... null (null = 값 없음)
```

당신이 마주칠 데이터 타입들

JSON 의 모든 값은 작은 타입 집합 중 하나다. 이것들을 알면 값이 엉뚱한 종류일 때를 알아챌 수 있는데, 이게 버그의 흔한 원천이다.

- **string** —텍스트, 항상 큰따옴표 안에 있다. "Ada Lovelace", "ada@example.com".
- **number** —그냥 숫자, 따옴표 없음. 42, 3.14. (따옴표가 의미를 바꾼다. "42" 는 사-이라는 텍스트이지 숫자가 아니다.)
- **boolean** —true 아니면 false, 따옴표 없음. "isAdmin" 같은 예/아니오 플래그에 쓰인다.
- **null** —의도적인 "여기엔 값 없음". "avatarUrl": null 은 사용자에게 아바타가 없다는 뜻이다—키가 아예 빠진 것과는 다르다.
- **array** —순서 있는 목록. ["beta", "early-access"].
- **object** —키/값 쌍의 묶음. "profile" 블록.

이게 어휘 전부다. 현실에서 가장 흔한 실수는 string-대-number 함정이다. 코드는 number 42 를 기대하는데 API 는 string "42" 를 보내고, 비교가 조용히 실패한다. 뭔가 이상하게 동작할 때, 값의 타입을 확인하는 게 종종 해결책이다.

실제 요청과 응답

다음은 실제 API 호출이 처음부터 끝까지 어떻게 생겼는지다—앱이 보내는 요청과, 돌려받는 JSON 이다.

```
GET /forecast?city=London HTTP/1.1
Host: api.weather.com
Authorization: Bearer sk_live_8f4b2c...
Accept: application/json
```

```
{
  "city": "London",
  "unit": "celsius",
  "current": {
    "temp": 14,
    "condition": "cloudy"
  },
}
```

```

"forecast": [
  { "day": "Mon", "high": 16, "low": 9 },
  { "day": "Tue", "high": 18, "low": 11 }
]
}

```

저 Authorization 줄이 다음 주제다—그리고 제대로 짚고 넘어가야 할 가치가 가장 큰 것이다.

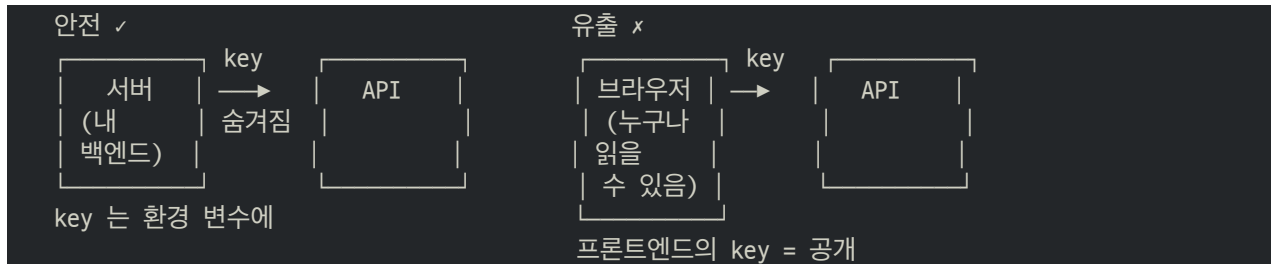
API key 와 인증 헤더

쓸모 있는 API 는 대부분 세상 누구에게나 열려 있지 않다. 회사는 누가 요청하는지를 알아야 사용량을 추적하고 당신에게 요금을 청구할 수 있다. 그래서 **API key** —당신의 계정을 식별하는 길고 비밀스러운 문자열—를 준다. 당신은 그걸 모든 요청에 붙이는데, 보통 Authorization 이라 불리는 **HTTP header** 에 붙인다. 위의 Bearer sk_live_... 줄처럼 말이다.

API key 는 정확히 비밀번호처럼 다뤄야 한다. 실제로 그게 비밀번호이기 때문이다. 그걸 가진 사람은 누구든 당신인 척 요청을 보낼 수 있다—그리고 당신의 청구서를 불러놓거나 당신의 데이터를 읽을 수 있다. 이건 보안 챕터와 공장 이어지는데, 가장 중요한 규칙 하나가 있다.

- **API key 를 브라우저로 전달되는 코드에 절대 넣지 마라.** 당신의 프론트엔드에 있는 건 무엇이든 공개된 것이다—사용자가 개발자 도구를 열어서 읽을 수 있다. AI 에게 “페이지에서 API 를 호출해줘”라고 하면, 그놈은 신나게 당신의 비밀 key 를 바로 거기에 붙여넣을 것이다. 막아라. 비밀 key 는 서버에 있어야 한다.
- **key 를 git 에 절대 커밋하지 마라.** 나중에 지우더라도, 그건 히스토리에 영원히 살아남고, 봇들은 몇 분 안에 공개 저장소를 훑어 key 를 찾아낸다.
- **key 는 환경 변수에 저장하라,** 코드 안이 아니라—보안 챕터에서 다루는 바로 그 패턴이다.

key 는 안전한 곳에서 출발해야 하며, 브라우저에서 출발해선 절대 안 된다. 두 갈래 길을 그려보라—하나는 안전하고, 하나는 유출이다.



만약 실제 key 가 diff 안에, 프론트엔드 파일 안에, 또는 채팅 메시지 안에 딱하니 있는 걸 본다면, 이미 유출된 것으로 취급하라. 즉시 교체하라 (새로 발급하고, 기존 것을 비활성화하라).

속도 제한과 요금 폭탄의 함정

메뉴가 큰소리로 외치진 않지만 당신을 반드시 물어버릴 두 가지가 더 있다.

속도 제한 (Rate limits). API 는 일정 시간 창 안에 보낼 수 있는 요청 수에 상한을 둔다—가령 분당 100 개. 넘기면 속도를 늦출 때까지 429 Too Many Requests 에러로 거절당한다. 이건 한 고객이 서비스를 압도하는 걸 막기 위해 존재한다. 함정은 이렇다. AI 가 짠 코드는 종종 일시정지 없이 반복문 안에서 API 를 호출해서, 순식간에 제한을 뚫어버린다. 어떤 기능이 항목 하나에는 잘 돌아가는데 천 개를 돌리면 망가진다면, 속도 제한이 유력한 용의자다.

종량제 및 유료 API. 많은 API 는 요청당 요금을 매긴다—각각 1 센트의 몇 분의 일. 호출 몇 번이면 아무것도 아니지만, 뭔가가 통제 불능으로 반복되면 진짜 청구서가 된다. 전형적인 공포 이야기가 있다. 버그가 실패한 유료 호출을 영원히 재시도하거나, 공개된 품이 낯선 사람들이 유료 호출을 일으키게 놔두고, 그러다 네 자릿수 청구서를 보며 잠에서 깬다. 스스로를 보호하라.

- 유료 API 를 연결하기 전에 요금을 읽어라. 호출당 비용을 알아두라.
- 제공업체 대시보드에 **지출 한도나 예산 알림**을 설정하라—그런 기능이 있다면. 대부분은 있다.

- 유료 API 를 반복문 안에서나 매 페이지 로드마다 호출하는 코드는 어떤 것이든 의심하라.

API 문서를 읽는 법

당신은 API 문서 속에서 살게 될 테니, 전부 읽지 않고도 필요한 걸 빨리 얻는 법을 알아두자. 어떤 엔드포인트든, 다섯 가지를 찾아라.

- **메서드와 URL**—GET /forecast. 무슨 일을 하는지, 두 토큰으로.
- **인증**—어떤 header 나 key 가 필요한지. 보통 API 전체에 대해 공유되는 한 섹션이다.
- **파라미터**—보낼 수 있는 입력값들 (여기서는 city), 무엇이 필수인지, 그리고 그것들의 타입.
- **예시 응답**—돌려받게 될 JSON 모양. 이건 금덩이다. 복사해두라.
- **예러**—뭔가 잘못됐을 때 무엇이 돌아오는지, 그리고 상태 코드 (401 권한 없음, 404 찾을 수 없음, 429 속도 제한).

좋은 문서에는 복사-붙여넣기 가능한 예시 요청과 응답이 들어 있다. 그 두 토막은 문단으로 된 산문보다 더 많은 걸 알려준다—그것들을 챙겨서 AI 에게 건네라.

AI 에게 호출을 맡기되—당신은 모양을 점검하기

여기 실용적인 분업이 있다. AI 는 API 를 호출하는 코드를 정말 잘 쓴다. 라이브러리도 알고, header 문법도 알고, 예러 처리도 안다. 맡겨라. 이 배관은 당신이 손으로 짜지 않는다.

하지만 AI 가 못하는 것 하나는 당신이 책임진다. **데이터 모양이 실제로 맞는지 아는 것이다.** 좋은 작업 흐름은 이렇다.

- **AI 에게 진짜 문서를 줘라.** 문서의 예시 요청과 응답을 프롬프트에 붙여넣어라. AI 가 API 모양을 기억으로 추측하는 건 버그의 으뜸가는 원천이다—그렇듯해 보이지만 존재하지 않는 키를 지어낸다. 진짜 예시가 그걸 현실에 묶어둔다.
- **먼저 원본 응답을 보여달라고 해라.** “파싱하기 전에, API 가 실제로 돌려주는 JSON 을 로그로 찍어줘.” 그런 다음 코드가 기대하는 것과 눈으로 대조하라. 코드가 읽는 키 (response.current.temp) 가 응답에 실제로 존재하는가? API 버그의 절반은 키 이름이 어긋난 것이다.
- **타입을 점검하라.** 그건 number 인가 string 인가? 항상 있어야 할 항목이 어쩌다 null 이진 않은가? 이제 JSON 을 읽을 수 있으니—써먹어라.
- **불행한 경로를 처리하라.** 물어보라. “이 호출이 실패하거나, 시간 초과되거나, 속도 제한에 걸리면 어떻게 되지?” 답이 “앱이 죽는다”라면, 그건 기능이 아니라 버그다.

요청을 직접 쓸 필요는 없다. 응답을 읽고 “이게 내가 기대한 모양인가?”라고 물으면 된다. 그 하나의 습관—데이터를 계약과 대조하는 것—이 사용자가 보기도 전에 통합 버그 대부분을 잡아낸다.

정리와 실습

핵심 요약

- **API** 는 한 프로그램이 다른 프로그램에 제공하는 요청 메뉴이며, **계약**입니다: 합의된 방식으로 물으면 합의된 형태의 답을 받습니다. 대부분의 API 버그는 한쪽이 그 계약을 깨는 것입니다.
- **JSON** 은 답이 돌아오는 형식입니다—키와 값으로 이뤄지고, 객체 { } 와 배열 [] 로 만들어지며 중첩됩니다. 데이터 타입은 string, number, boolean, null, array, object 입니다.
- **문자열 대 숫자 함정** (42 대 "42") 과 누락 대 null 이 가장 흔한 형태 버그입니다. 이제 JSON 을 읽을 수 있으니—뭔가 이상하게 굴면 타입을 확인하세요.
- **API 키** 는 비밀번호입니다. 절대 브라우저로 내보내지 말고, 절대 git 에 커밋하지 말고, 환경 변수에 저장하세요. 유출된 키는 즉시 교체 (rotate) 해야 합니다.
- **레이트 리밋** (429) 과 **사용량 과금**을 조심하세요—유료 API 를 험 없이 호출하는 AI 작성 루프가 네 자리 숫자 청구서를 만드는 길입니다. 먼저 요금을 읽고 예산 알림을 설정하세요.

직접 해보기

브라우저에서 키가 필요 없는 무료 API 를 찾아보세요—예를 들어 새 탭에서 <https://api.github.com/users/octocat> 을 바로 열어보세요. 원시 JSON 응답이 보일 겁니다. 양식처럼 읽으세요: 객체 { } 를 골라내고, 배열이 있을

면 찾고, 서로 다른 값 세 개의 타입을 이름 붙여 보세요 (어느 게 문자열? 숫자? 불리언? null 인 건 없나?). 이 장이 AI가 연결하는 모든 API 응답에 대해 하라고 하는 바로 그 점검을 방금 한 것입니다.

이 장의 프롬프트

나는 API 를 호출하고 있고, 초보자로서 데이터 모양을 직접 점검해 보고 싶어.

여기 진짜 문서야 (예시 요청 + 예시 응답):

<API 문서의 예시 요청과 JSON 응답을 여기에 붙여넣어라 >

- 파싱 코드를 작성하기 전에, 이 호출이 돌려주는 원본 JSON 을 보여주고 키들, 그 타입들, 그리고 null 이거나 빠질 수 있는 것을 짚어줘.
- 내 코드가 읽는 키들이 그 응답에 실제로 존재하는지 확인해줘.
- API key 가 서버의 환경 변수에서 읽히도록 하고, 코드에 하드코딩하거나 브라우저로 보내지 않도록 해줘.
- 호출이 실패하거나, 시간 초과되거나, 속도 제한에 걸리면 어떻게 되는지 알려줘.

앱의 해부학



AI 에게 “로그인 페이지를 추가해줘”라거나 “이걸 데이터베이스에 저장해줘”라고 요청하면, AI 는 조용히 여섯 개쯤 되는 구성 요소에 대한 결정을 내립니다. 그런 요소들이 존재한다는 사실조차 모른다면, 그 답변들은 외국어처럼 읽힐 테고—좋은 답인지 헷갈린 답인지조차 구별할 수 없습니다.

이 장은 지도입니다. 여러분을 엔지니어로 만들어주지는 않지만, 현대적인 앱의 어떤 조각이든 가리키며 그게 무슨 일을 하고 어디에 있는지 말할 수 있게 해줍니다. 일단 지도를 손에 쥐면, 다른 모든 장이 더 이상 추상적으로 느껴지지 않습니다.

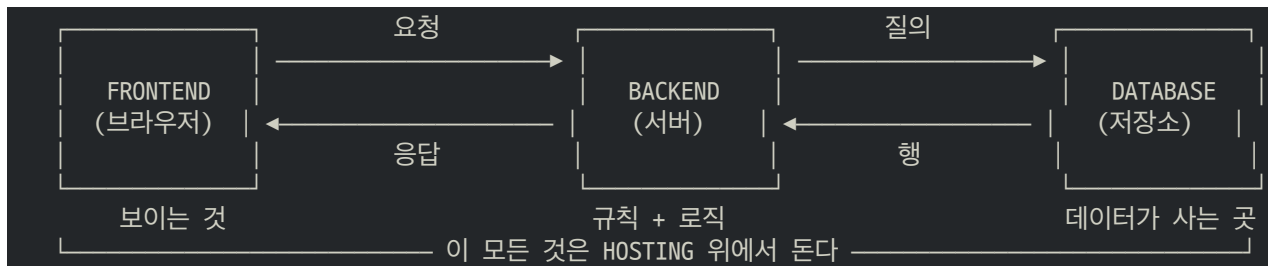
니다. 문제를 설명하기 시작하기도 전에 그 문제가 어느 칸에 들어있는지 알게 되니까요. 우리는 쉽게 설명할 겁니다—정의 없는 전문 용어는 쓰지 않고, 하나의 예시 (사용자가 버튼을 클릭하는 상황) 를 모든 계층을 따라 추적하면서요.

큰 그림

여러분이 만들게 될 거의 모든 앱은 서로 대화하는 네 부분으로 이루어져 있습니다:

- **frontend(사용자가 보는 화면)** —사용자가 보고 클릭하는 것. 브라우저나 휴대폰 앱 안의 화면, 버튼, 글자입니다.
- **backend(서버에서 도는 로직)** —여러분이 관리하는 서버에서 돌아가는 로직. 규칙을 강제하고, 일을 처리하고, 누가 무엇을 할 수 있는지 결정합니다.
- **데이터베이스**—정보가 머무는 곳, 그래서 내일도 그 정보가 그대로 있게 해줍니다. 계정, 게시물, 주문 같은 것들.
- **호스팅**—세상 어딘가에 있는, 위의 모든 것을 실제로 돌려서 다른 사람들이 접근할 수 있게 해주는 컴퓨터들입니다.

이것들이 어떻게 연결되는지 봅시다:



frontend 는 절대 데이터베이스와 직접 대화하지 않습니다. 항상 규칙이 머무는 backend 를 거칩니다. 이 한 가지 사실이 앱이 어떻게 만들어지는지에 대해 많은 것을 설명해줍니다—그리고 왜 “그냥 페이지가 데이터베이스를 읽게 하자”가 거의 정답이 아닌지도요.

frontend, backend, 데이터베이스, 호스팅

예시를 가지고 한 계층 더 깊이 들어가 봅시다: 사용자가 “**프로필 저장**”을 클릭합니다.

frontend. 웹 브라우저나 모바일 앱으로 돌아가는, 눈에 보이는 앱입니다. 폼을 그리고, 클릭을 알아채고, 사용자가 입력한 내용을 모읍니다. 설계상 이것은 신뢰할 수 없습니다. 누구나 열어보고, 들여다보고, 바꿀 수 있으니까요. 그래서 frontend 의 일은 보기 좋고 반응이 빠르게 느껴지는 것이지, 중요한 무언가를 강제하는 게 아닙니다.

backend. 버튼이 클릭되면, frontend 는 데이터를 backend 로 보냅니다—서버에서 돌아가는 코드죠. backend 는 사용자가 로그인되어 있는지 확인하고, 데이터를 검증하고 (“이게 정말 이메일 주소가 맞나?”), 비즈니스 규칙을 적용한 뒤, 그제서야 저장합니다. 사용자가 함부로 손댈 수 없기에, 신뢰는 바로 여기에 머뭅니다.

데이터베이스. backend 는 데이터를 데이터베이스에 넘겨 저장합니다. 마법 상자가 아닙니다—즉시 검색할 수 있고 절대 잊어버리지 않는, 아주 엄격한 스프레드시트들 (테이블이라고 부릅니다) 의 모음이라고 생각하세요. 내일이 되면 backend 는 프로필을 다시 읽어 내서 또 보여줍니다.

호스팅. prod(실제 서비스) 환경에서는 이 중 어느 것도 여러분의 노트북에서 돌지 않습니다. Cloudflare, Vercel, 또는 어떤 클라우드 플랫폼 같은 호스팅 제공업체에서 빌린 컴퓨터에서 돌아갑니다. 호스팅이 있어야 여러분의 앱이 자기 컴퓨터에서만 아니라 진짜 웹 주소에서 접근 가능해집니다.

쓸모 있는 직감 하나: 무언가 망가지면 어느 계층인지 물어보세요. “버튼이 아무 반응이 없어”는 frontend 입니다. “영똥한 걸 저장해”는 보통 backend 죠. “내 데이터를 잊어버렸어”는 데이터베이스를 가리킵니다. 계층을 짚어내는 것이 버그를 고치는 일의 절반입니다—그리고 AI 가 실행에 옮길 수 있는 프롬프트를 쓰는 일의 절반이기도 하죠.

HTML, CSS, JavaScript

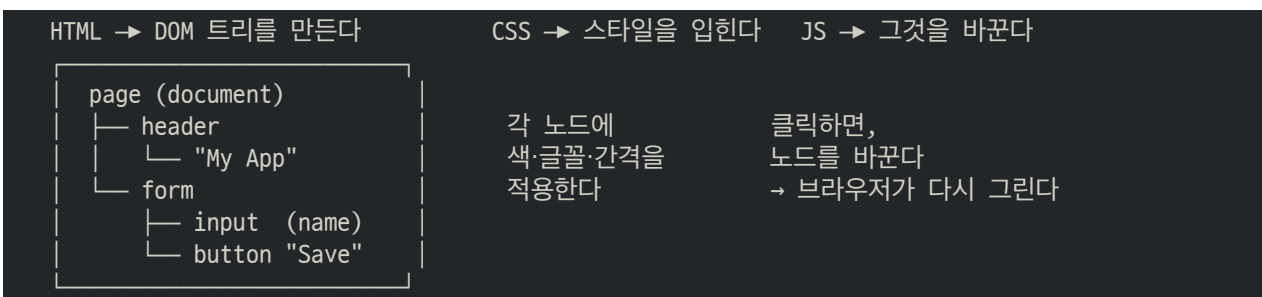
브라우저 안의 frontend 는 정확히 세 가지 언어로 만들어집니다. 손으로 직접 쓸 필요는 없지만, 각각이 무엇을 담당하는지는 알아야 합니다.

HTML(하이퍼텍스트 마크업 랭귀지) 은 구조입니다—내용과 그 뼈대죠. “여기 제목이 있다, 여기 문단이 있다, 여기 버튼이 있다”라고 말합니다. 페이지의 명사인 셈입니다. HTML 만 있으면 꾸밈없는 맛있는 문서, 라벨이 붙은 텍스트 파일 같은 모습입니다. 브라우저는 여러분의 HTML 을 **DOM**(Document Object Model, 문서 객체 모델) 이라 불리는, 메모리 안에 살아있는 객체 트리로 바꿉니다. 코드가 “페이지를 바꾼다”고 할 때, 실제로는 DOM 을 바꾸는 것이고, 브라우저는 거기에 맞춰 다시 그립니다.

CSS(캐스케이딩 스타일 시트) 는 스타일입니다—모든 것이 어떻게 보이게 하죠. 색깔, 글꼴, 간격, 배치, 왼쪽이나 가운데냐, 작은 화면이나 큰 화면이나. CSS 는 페이지에 무엇이 있는지를 바꾸지 않고, 그것이 어떻게 보여지는지만 바꿉니다. HTML 이 명사라면, CSS 는 형용사입니다.

JavaScript(JS) 는 동작입니다—여러분이 상호작용할 때 무슨 일이 일어나는거죠. 클릭에 귀 기울이고, 데이터를 backend 로 보내고, 스피너를 보여주고, 응답이 돌아오면 DOM 을 업데이트합니다. HTML 과 CSS 는 정적입니다. **JavaScript** 가 페이지로 하여금 무언가를 하게 만듭니다. 동사인 셈이죠. (같은 언어가 흔히 backend 도 돌리는데, 그래서 이 언어를 어디서나 보게 됩니다.)

이 분담을 그려보는 한 가지 방법: HTML 은 트리 (DOM) 를 만들고, CSS 는 거기에 색을 칠하고, JavaScript 는 그 안으로 손을 뻗어 그것을 바꿉니다:



패키지와 의존성

아무도 앱 전체를 처음부터 다 쓰지 않습니다. 어렵고 흔한 문제들—날짜 형식 맞추기, 결제 처리하기, 달력 그리기—은 이미 다른 사람들이 해결해서 묶어 놓았습니다. **패키지**는 여러분이 직접 쓰는 대신 프로젝트에 끌어와 쓰는, 재사용 가능한 코드 덩어리입니다. 프로젝트의 **의존성**은 그 프로젝트가 의지하는 패키지들의 목록입니다.

JavaScript 세계에서 이것을 관리하는 도구가 **npm**(Node Package Manager) 입니다. 두 개의 파일이 장부 역할을 합니다:

- **package.json** —프로젝트에 무엇이 필요한지를 사람이 읽을 수 있게 적은 목록에, 실행 명령어까지 더한 것입니다. 여러분 (또는 AI) 이 이 파일을 편집합니다.
- **잠금 파일 (lockfile)** (package-lock.json 같은 것) —모든 패키지와 그 정확한 버전을 기계가 생성한 그대로 기록한 것입니다. 그래서 여러분의 컴퓨터에서든, 동료의 컴퓨터에서든, 서버에서든 앱이 똑같이 빌드됩니다. 이 파일은 손으로 편집하지 않고, 도구가 관리하도록 맡깁니다.

```

{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
  "dependencies": {
    "next": "16.0.0",
    "react": "19.0.0"
  }
}
    
```

왜 신경 써야 할까요? AI 가 “이걸 위해 라이브러리를 하나 추가하자”고 할 때, 그것은 **package.json** 을 편집하는 것이고 여러분은 남의 코드를 믿는 겁니다—잘 알려진 의존성이 적을수록, 정체불명의 의존성이 잔뜩 쌓인 것보다 안전합니다.

다. 그리고 빌드가 영문도 모르게 망가질 때, 버전이 어긋난 의존성이 흔한 용의자이고, “패키지를 지우고 다시 설치한다”가 실제로 자주 통하는 해결책입니다.

환경 변수와 설정

어떤 값들은 코드 안에 적어 넣을 수 없습니다: 비밀 키, 데이터베이스 비밀번호, 어떤 서비스의 주소 같은 것들이죠. 그리고 어떤 값들은 앱이 어디서 도는가에 따라 바뀌어야 합니다. 여러분이 노트북에서 테스트하는 코드 (**dev** 환경) 는 테스트용 데이터베이스와 대화해야 하고, 사용자들이 실제로 쓰는 살아있는 앱 (**prod**, 즉 프로덕션 환경) 은 진짜 데이터베이스와 대화해야 합니다—같은 코드를 쓰면서요.

그 답이 **환경 변수** (흔히 “env vars”라고 부르죠) 입니다: 코드 바깥에 머물다가 코드가 실행될 때 코드에 공급되는, 이름 붙은 값들입니다. 코드는 “DATABASE_URL 을 줘”라고 말하고, 환경이 어느 것을 건네줄지 결정합니다.

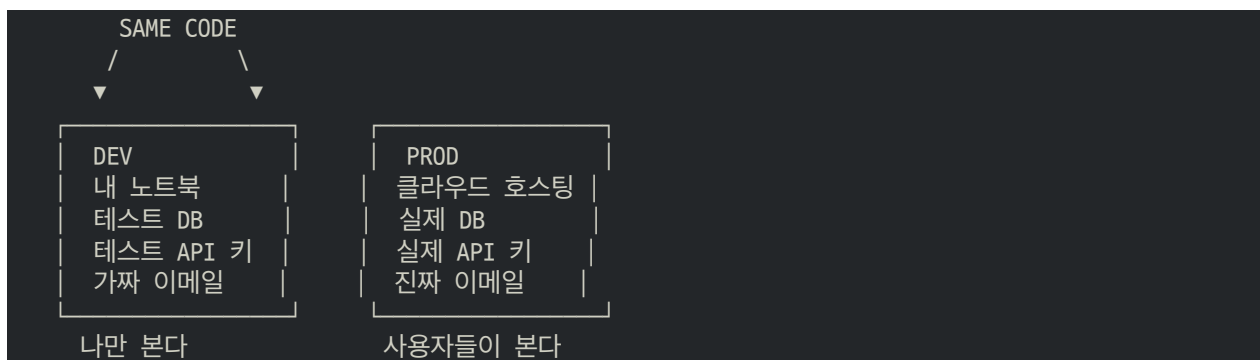
설정 파일 (예: .env) —값은 코드가 아니라 여기에 산다

```
DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...    # dev = 테스트용 키
SEND_REAL_EMAILS=false               # 테스트 중에는 진짜 사용자에게 이메일 보내지 말 것
```

두 가지 규칙이 진짜 고통을 덜어줄 겁니다:

- **비밀은 절대 코드에 넣지 마세요.** 코드 안에 있는 것은 무엇이든 공개될 수 있습니다 (특히 공유된 저장소에서요). 그리고 유출된 결제 키나 데이터베이스 비밀번호는 진짜 비상사태입니다. 비밀은 env vars 에 속하며, 버전 관리 바깥에 뒤야 합니다.
- **dev 와 prod 는 다릅니다.** 핵심은 같은 코드가 양쪽 모두에서 올바르게 동작한다는 것입니다. 어느 데이터베이스와 키를 쓸지 결정하는 것이 코드가 아니라 환경이기 때문이죠.

같은 코드, 두 개의 집—다른 것은 env vars 뿐입니다:



프로젝트의 생김새

전형적인 프로젝트를 열어보면 폴더 목록이 잡음처럼 느껴질 수 있습니다. 하지만 잡음이 아닙니다—대부분의 프로젝트는 알아볼 수 있는 패턴을 따릅니다. 여기 군더더기를 뺀 웹 앱이 있습니다:

```
my-app/
├─ package.json      # 의존성 + 실행 명령어 (위 참고)
├─ package-lock.json # 잠금 파일 —정확한 버전
├─ .env              # 로컬 비밀/설정 —공유 안 함
├─ .gitignore        # Git 이 무시해야 할 파일 (예: .env)
├─ README.md         # 이 프로젝트가 무엇인지, 어떻게 실행하는지
├─ public/           # 있는 그대로 제공되는 정적 파일 (이미지, 아이콘)
├─ src/              # 여러분의 실제 코드가 사는 곳
│   └─ components/   # 재사용 가능한 frontend 조각 (버튼, 카드)
│   └─ pages/ (or app/) # 사용자가 이동하는 화면/경로
```

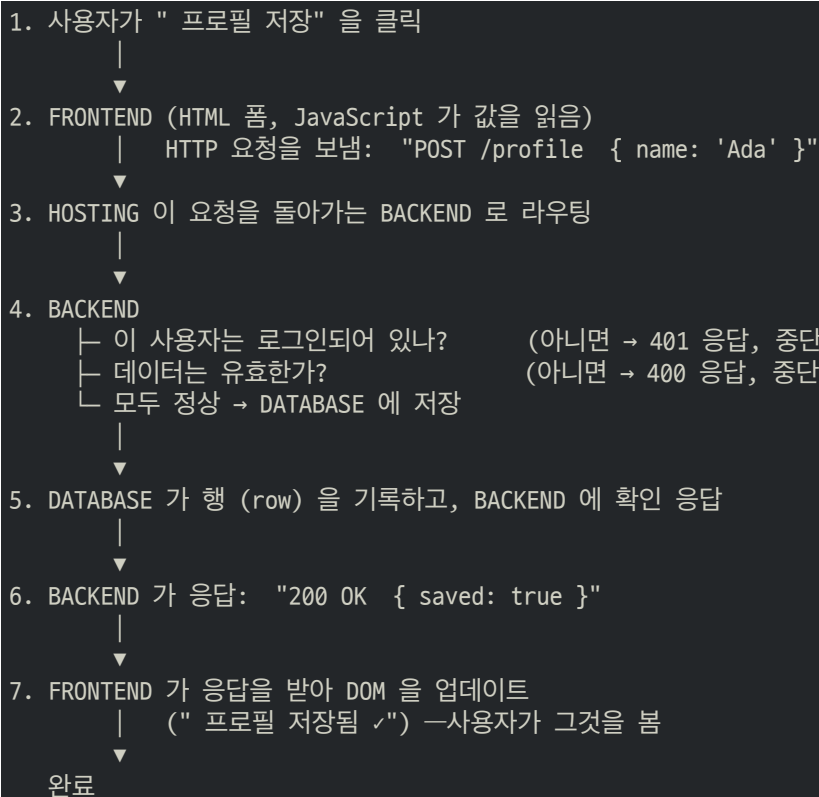
```
└─ lib/           # 공유 헬퍼와 backend 로직
└─ styles/        # CSS
```

이걸 외울 필요는 없습니다—그저 알아보기만 하면 됩니다. 그래야 AI가 “이걸 src/lib에 추가할게”라고 할 때, 그게 대략 어디인지 그리고 왜인지 알 수 있죠. 무게를 지탱하는 몇 가지 관례:

- src/는 여러분이 쓰는 코드를 담습니다. 최상위 단계의 어수선한 것들은 대부분 설정입니다.
- **루트의 설정 파일들** (package.json, dotfile 등)은 도구를 설정하는 것이지, 여러분 앱의 동작을 설정하는 게 아닙니다.
- .gitignore는 버전 관리에 저장되지 말아야 할 파일들을 나열합니다—여러분의 .env가 여기에 속하며, 바로 이것이 비밀이 공유 저장소 바깥에 머무는 방식입니다.

요청은 어떻게 흐르는가

이제 모든 걸 하나로 엮어봅시다. 방금 만난 모든 조각을 거치며, 사용자가 “**프로필 저장**”을 클릭하는 과정을 따라가 봅시다. 이것이 이 책에서 가장 쓸모 있는 단 하나의 사고 모델입니다. 다이어그램 하나를 내 것으로 만든다면, 이걸로 하세요.



각 계층이 무엇을 했는지 짚어봅시다: **frontend**는 입력을 모아서 보냈고, **호스팅**은 요청을 올바른 곳에 전달했고, **backend**는 규칙 (로그인, 검증)을 강제하고 결정을 내렸고, **데이터베이스**는 그것을 영구적으로 만들었고, 응답은 다시 끝까지 흘러가 **frontend**가 화면을 업데이트할 수 있게 했습니다. 모든 앱은 초당 수천 번씩 돌아가는 이 반복의 변주일 뿐입니다.

검사가 모두 backend에서 일어나고, frontend에서는 절대 일어나지 않는다는 점에 주목하세요—첫 번째 절의 규칙이 실제로 작동하는 모습입니다: frontend는 사용자를 위한 것이고, backend는 진실을 위한 것입니다. 각 단계가 어디서 일어나는지 알면, 어떤 기능을 모호하게 (“폼이 동작하게 해줘”)가 아니라 정확하게 (“저장하기 전에 backend에서 이메일을 검증해줘”) 설명할 수 있습니다—그리고 정확함이 이 모든 게임의 전부입니다.

정리 & 연습

핵심 요약

- 모든 앱은 네 부분입니다: **frontend**(사용자가 보는 것), **backend**(규칙과 로직), **데이터베이스** (데이터가 지속되는 곳), 그리고 **호스팅** (이 모든 게 돌아가는 곳). frontend 는 절대 데이터베이스에 직접 손대지 않습니다—backend 를 거칩니다.
- 브라우저에서 **HTML** 은 구조, **CSS** 는 스타일, **JavaScript** 는 동작입니다. 코드는 **DOM** 을 바꿈으로써 페이지를 바꿉니다.
- 모든 걸 직접 쓰지 않습니다—**패키지** (package.json 과 잠금 파일을 통해 **npm** 이 관리) 가 해결된 문제들을 **의존성**으로 공급합니다.
- **환경 변수**는 비밀을 코드 바깥에 두고, 같은 코드가 dev 와 prod 에서 올바르게 동작하게 해줍니다.
- 요청은 frontend → 호스팅 → backend → 데이터베이스로 흘렀다가 되돌아옵니다. 문제가 머무는 계층을 짚어내는 것이 해결의 절반입니다.

해보기

매일 쓰는 앱 하나를 골라 한 가지 동작을 계층을 거치며 소리 내어 풀어보세요: “내가 전송을 누르면 (frontend) → 그게 그쪽 서버로 가고 (backend) → 내가 권한이 있는지 확인하고 메시지를 저장한 뒤 (데이터베이스) → 채팅이 업데이트된다 (frontend).” 그런 다음 각 문제가 어느 계층에 사는지 추측해보세요: 버튼이 어긋나 보인다; 앱이 “로그인되어 있지 않습니다”라고 한다; 어제 온 메시지가 사라졌다. 이렇게 계층을 짚어내는 데 능숙해지는 것이 이 책의 나머지가 그 위에 쌓아 올리는 바로 그 기술입니다.

이 장의 프롬프트

나는 앱이 어떻게 맞물려 돌아가는지 배우고 있어. 내가 원하는 기능은 이거야:
< 사용자가 무엇을 할 수 있어야 하는지 한 문장으로 설명 >

코드를 쓰기 전에, 계층별로 나눠서 설명해줘:

- Frontend: 사용자가 보는 것, 그리고 무엇이 전송되는지
- Backend: 어떤 규칙/검사가 돌아야 하는지, 그리고 어디서
- Database: 어떤 데이터를 저장하거나 읽어야 하는지
- 관련된 env vars 나 설정 (특히 비밀)

그런 다음 클릭부터 결과까지 요청 흐름을 번호 매긴 단계로 설명해줘.
쉬운 말로 해줘 —코드 전에 전체 생김새를 이해하고 싶어.

바이브 코딩 마인드셋



바이브 코딩은 의미가 과하게 실린 용어다. 누군가에게는 한 줄짜리 프롬프트를 입력하고, 모델이 뱉어내는 것을 그대로 받아들이고, 한 줄도 읽지 않은 채 출시하는 것을 뜻한다. 그런 버전은 장난감이다. 실제 사용자가 손을 대는 순간 무너지는 데모를 만들어낼 뿐이다.

이 책은 다른 버전에 관한 것이다. 시니어 엔지니어가 팀을 이끌듯이 AI 를 지휘해 프로덕션 소프트웨어를 만드는 것 말이다. 무엇을 만들지, 그리고 왜 그것이 옳은지는 당신이 책임진다. 타이핑은 위임한다. 그것이 바로 그 전환이며, 들리는 것보다 훨씬 큰 변화다.

바이브 코딩이란 실제로 무엇인가

바이브 코딩은 **자연어로 의도를 표현하고 AI 가 구현을 생성하게 하되**, 당신은 긴밀한 루프 안에서 방향을 잡고, 검토하고, 다듬는 것이다.

이것은 다음이 아니다:

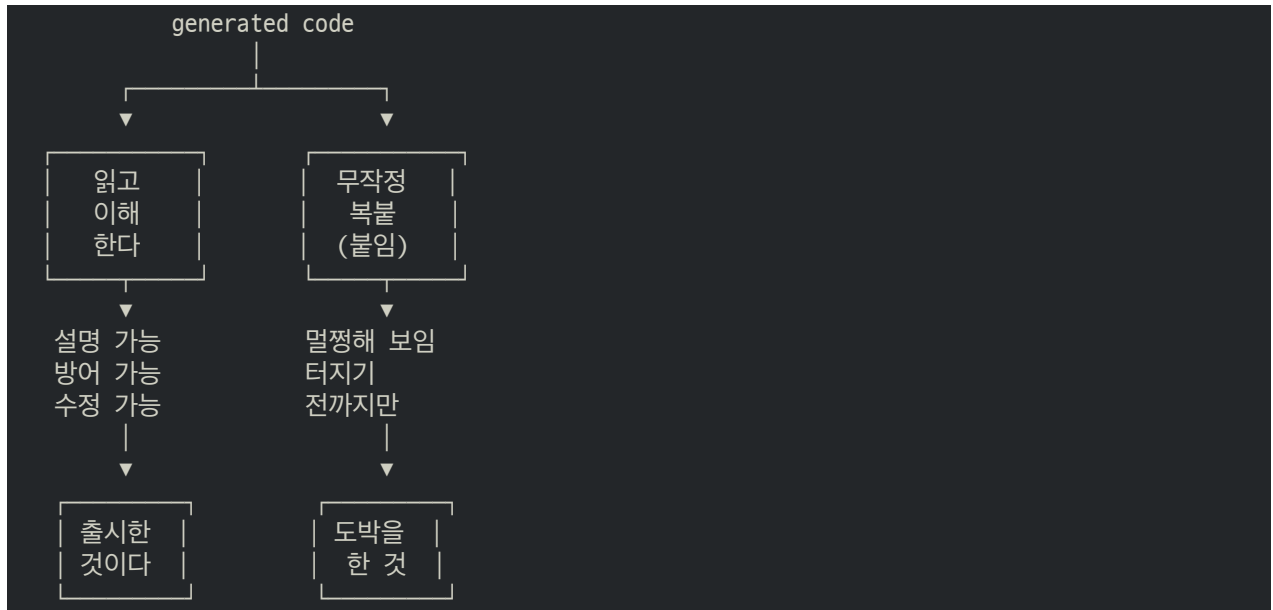
- 이해하지 못하는 생성된 코드를 맹목적으로 받아들이는 것.
- 엔지니어링 판단을 그저 느낌으로 대체하는 것.
- 한 문장을 출시 가능한 제품으로 바꿔주는 마법의 버튼.

이것은 다음이 맞다:

- 모델을 빠르고, 박식하고, 약간 과신하는 주니어 엔지니어로 대하는 것.
- 문법 대신 아키텍처, 정확성, 그리고 안목에 당신의 주의를 쏟는 것.
- 예전에는 몇 시간 걸리던 일을 몇 분 안에 반복하는 것.

가장 잘 작동하는 멘탈 모델은 이것이다: 당신은 테크 리드이고, AI 는 당신의 팀이다. 좋은 테크 리드는 모든 줄을 직접 쓰지 않는다. 방향을 정하고, 제약을 설정하고, 풀 리퀘스트 (pull request —승인을 받기 위해 제출하는 변경 묶음) 를 검토하고, 나쁜 아이디어가 출시되기 전에 잡아낸다. 당신은 그 일을 하고 있다—다만 대화의 속도로.

이 두 갈래 길은 똑같은 생성된 코드에서 출발하지만 전혀 다른 곳에서 끝난다:



이 정의에서 무거운 짐을 지는 단어가 하나 있다: **이해**다. 이 접근 전체는 단 하나의 규칙에 기반다—당신이 출시하는 것을 이해하라. 직접 타이핑할 필요는 없지만, 그것을 설명할 수 있어야 하고, 코드 리뷰에서 방어할 수 있어야 하고, 새벽 2 시에 호출이 올릴 때 고칠 수 있어야 한다. 그러지 못한다면 당신은 바이브 코딩을 한 게 아니라 도박을 해서 운이 좋았던 것뿐이다. 그 둘 사이의 간극이 이 책 전체의 주제다.

마인드셋 전환

모든 글자를 직접 타이핑하던 출신이라면, 가장 어려운 부분은 도구를 배우는 것이 아니다. 키보드를 놓는 것이다.

당신의 예전 병목은 생산이었다—얼마나 빨리 정확한 코드를 쓸 수 있는가. 당신의 새로운 병목은 명세와 검토다—원하는 것을 얼마나 명확하게 설명할 수 있고, 돌아온 결과를 얼마나 잘 판단할 수 있는가.

이것은 당신이 무엇을 최적화하는지를 바꾼다:

- **이전:** API(애플리케이션 프로그래밍 인터페이스—도구가 제공하는 명령 메뉴) 를 외우고, 문법과 씨름하고, 빠르게 타이핑한다.
- **지금:** 의도를 정확하게 설명하고, 문제를 분해하고, 출력을 비판적으로 평가한다.

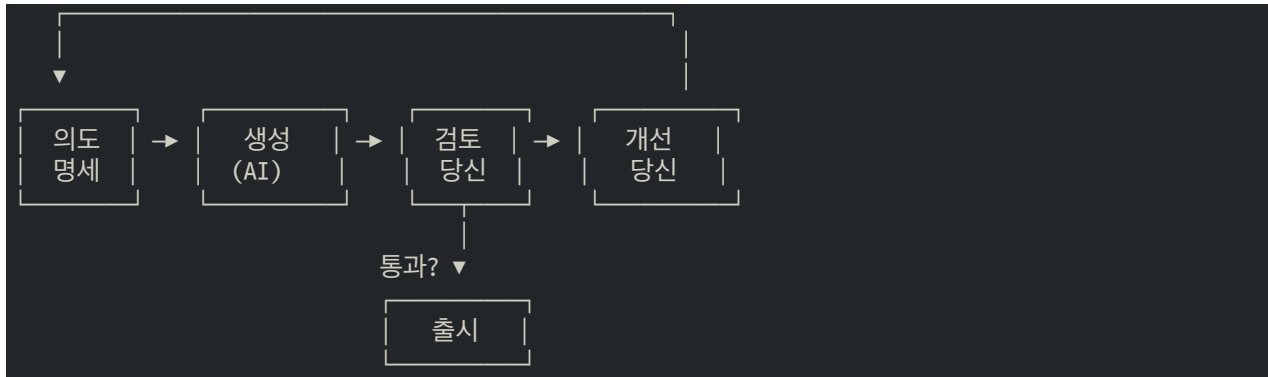
이 일에서 번창하는 엔지니어는 가장 빠른 타이피스트가 아니다. “좋은 것”이 어떤 모습인지에 대해 강한 의견을 가지고 있고, 기준을 충족하지 못하는 출력을 거부하는 규율을 갖춘 사람들이다.

사람들을 당황하게 만드는, 더 조용한 두 번째 전환이 있다: 모른다는 것과 당신의 관계다. 예전 본능은 API 를 건드리기 전에 멈춰서 그것을 배우는 것이었다. 이제는 익숙하지 않은 스택에서 첫 시도를 생성하고, 그것을 되읽으면서 배우는 경우가 많다. 그것은 진짜 초능력이다—그리고 함정이다. 당신이 직접 쓸 수 없었을 코드를 읽는 것은 괜찮다. 당신이 평가할 수 없는 코드를 출시하는 것은 괜찮지 않다. 경계선은 그 출력이 옳다고 자신 있게 말할 수 있느냐다. 당신이 쓸 수 있는 것보다 빠르게 쓰는 데에는 모델에 기대라. 당신이 검증할 의지가 있는 것보다 더 많이 아는 데에는 결코 기대지 마라.

루프: 의도 □ 생성 □ 검토 □ 개선

이 책의 모든 것은 하나의 사이클로 귀결된다. 이것을 체화하라.

루프는 출력이 당신의 기준을 충족할 때까지 돈다—그리고 검토는 매번 통과해야 하는 관문이다:



1. **의도 (Intent).** 원하는 것과 중요한 제약을 진술하라—입력, 출력, 옛지 케이스, 스택, 스타일. 모호한 의도는 모호한 코드를 낳는다.
2. **생성 (Generate).** 모델이 구현을 작성하게 하라. 문법을 일일이 붙잡지 말고, 동작을 설명하라.
3. **검토 (Review).** 돌아온 결과를 읽어라. 실제로 그 일을 하는가? 빈 리스트, null, 실패한 네트워크 호출을 처리하는가? 작동하는 것 중 가장 단순한 버전인가?
4. **개선 (Refine).** 무엇이 잘못됐는지 가리키고, 구체적인 수정을 제시하고, 다시 생성하라. 기준을 충족할 때까지 반복하라.

대부분의 초보자는 3 단계를 건너뛴다. 생성하고, 돌아가고, 넘어간다. 그러다 설명한 적 없는 케이스에서 프로덕션이 깨진다. 검토 단계가 바로 엔지니어링이 일어나는 곳이다.

이 루프에서 좋은 첫 프롬프트는 다음과 같은 모습이다:

TypeScript 함수 ``parseDuration(input: string): number`` 를 작성해줘.
`"1h30m"`, `"45s"`, `"2d"` 같은 문자열을 총 초 (seconds) 로 변환한다.

요구사항:

- 단위 지원: d (일), h (시간), m (분), s (초).
- 여러 단위 조합 가능: `"1h30m"` = 5400.
- 잘못된 입력 (빈 값, 알 수 없는 단위, 음수) 은 명확한 메시지와 함께 `Error` 를 throw 해서 거부한다.
- 외부 라이브러리 금지. 옛지 케이스를 다루는 단위 테스트 5 개 포함.

함수와 테스트만 보여줘.

구조에 주목하라: 정확한 시그니처, 명시적인 규칙, 명시된 옛지 케이스, 진술된 제약, 그리고 테스트 요청. 당신은 모델이 알아서 맞추기를 바라는 게 아니다—잘못된 추측을 만들어내는 모호함을 제거하는 것이다.

개선 단계도 첫 프롬프트만큼이나 하나의 기술이다. 모호한 피드백은 모호한 수정을 낳는다. 이 두 가지 수정 요청을 비교해보라:

나쁜 예: " 이거 틀렸어, 고쳐줘"

좋은 예: `"parseDuration('1h30m')` 가 5400 대신 90 을 반환해 —숫자는 더하면서 단위 배수를 무시하고 있어. 더하기 전에 각 값에 해당 단위의 초 (`d=86400`, `h=3600`, `m=60`, `s=1`) 를 곱해야 해. 그리고 조합 케이스 `'2d3h'` 에 대한 테스트도 추가해줘."

좋은 쪽은 증상, 원인, 그리고 수정 방법을 명시한다. 당신은 동료의 풀 리퀘스트를 검토하듯 검토하고 있는 것이다—그리고 더 정확하게 가리킬수록 더 적은 라운드를 쓴다. 각 루프는 수렴해야 한다. 제자리를 맴돌고 있다는 느낌이 든다면, 그것은 의도가 명세 부족이었다는 신호다. 다시 생성하기를 멈추고 명세를 다시 써라.

빛을 발하는 곳

바이브 코딩은 다음에 대해 진정으로 변혁적이다:

- **그린필드 스캐폴딩.** 새 앱을 띄우고, 라우트를 연결하고, 설정을 구성하는 일—지루하지만 잘 닦여 있는 작업.
- **보일러플레이트와 글루 코드.** CRUD 엔드포인트, 폼 검증, 데이터 변환, 두 API 사이의 어댑터.
- **익숙하지 않은 영역.** 잘 모르는 언어나 프레임워크. 모델은 관용구를 알고, 당신은 판단을 제공한다.
- **리팩터링과 마이그레이션.** “이 프로미스들을 async/await 로 변환해줘”, “이 400 줄짜리 파일을 책임별로 나눠줘”
- **버릴 탐색.** 어차피 다시 쓸 프로토타입. 다듬음보다 속도가 더 중요하다.

이런 영역에서는 명확한 프롬프트와 신중한 검토가 매번 손 타이핑을 이긴다.

공통된 맥락은 이 작업들이 **잘 닦여 있고 검증 가능하다**는 점이다. 모델은 만 개의 CRUD 엔드포인트를 봐왔기에 관용적인 것을 만들어낸다. 그리고 당신은 결과가 옳은지 한눈에 판별할 수 있다. 이 둘이 모두 참일 때—흔한 패턴, 검증 비용이 저렴함—바이브 코딩은 거의 순수한 이득에 가깝다.

실패하는 곳

한계에 대해 정직하라. 감독 없는 바이브 코딩이 사람들을 태워버리는 곳이 바로 여기다:

- **깊고 새로운 로직.** 미묘한 알고리즘, 동시성, 95% 맞다는 것이 곧 망가진 것을 의미하는 모든 것. 모델은 틀렸을 때조차 자신만만하다.
- **여러 파일에 걸친 방대한 변경.** 큰 코드베이스를 가로질러 맥락의 실을 놓치고, 보지 못하는 것들을 조용히 망가뜨린다.
- **보안과 돈.** 인증, 결제, 권한, 그럴듯해 보이는 버그가 실제 결과를 초래하는 모든 것. 당신의 일자리가 달린 것처럼 검토하라. 실제로 그럴 수도 있으니까.
- **명세가 부족한 문제.** 당신이 무엇이 옳은지 모른다면, AI 는 당신의 마음을 읽을 수 없다. 쓰레기 명세가 들어가면 쓰레기 코드가 나온다.

실패 양상은 거의 항상 같다: 맞아 보이고, 정상 경로에서는 돌아가고, 읽었더라면 잡아냈을 방식으로 틀린 코드. 해결책은 검토 단계다. 건너뛰 방법은 없다.

두 가지 질문이면 어떤 작업이든 이 지도 위에 놓인다—패턴이 흔한가, 그리고 결과를 점검하기 쉬운가:

	검증 쉬움	검증 어려움
흔한 패턴	순수 이득 (스캐폴딩, 보일러플레이트)	천천히, 테스트 요구
드문 / 새로운	읽어야 하지만 가능	위험 구역 (새 로직, 인증, 결제)

위 규칙의 역을 주목하라: 이것들은 드물거나, 미묘하거나, 검증 비용이 비싼 경우들이다. 패턴이 흔치 않아서 모델이 추측하고 있거나, 버그가 정상 경로에서는 보이지 않아서 빠른 점검으로는 잡히지 않는다. 그렇다고 여기서 AI 를 피하라는 뜻이 아니다—속도를 늦추고, 단계를 잘게 쪼개고, 테스트를 요구하고, 모든 줄을 적대적인 것처럼 읽으라는 뜻이다.

흔한 함정

몇 가지 함정은 첫 달에 거의 모두를 잡는다:

- **돌아가는 첫 결과를 받아들이는 것.** “돌아간다”와 “옳다”는 서로 다른 주장이다. 앞것은 정상 경로가 죽지 않았다는 뜻이고, 뒤것은 옛지를 점검했다는 뜻이다. 둘을 혼동하지 마라.
- **메가 프롬프트.** 기능 전체를 한 방에 요청하면 모델에게 표류할 여지를 너무 많이 주고, 당신에게는 한 번에 검토할 것을 너무 많이 안긴다. 분해하라. 가장 작은, 검토 가능한 조각을 출시하고, 그 위에 쌓아라.

- **컨텍스트가 썩게 두는 것.** 세션이 길어질수록 모델은 앞선 결정과 모순되기 시작하고, 이미 고친 버그를 다시 들어 온다. 그게 시작되면 혼란스러운 스레드와 싸우는 대신 현재 상태를 요약하고 새로 시작하라.
- **자신감을 신호로 신뢰하는 것.** 모델의 어조는 맞을 때나 환각일 때나 똑같다. 그 확신은 당신에게 아무것도 말해주지 않는다. 당신의 검증이 모든 것을 말해준다.
- **타이핑만이 아니라 결정까지 위임하는 것.** “X에 대한 내 선택지가 뭐야?”라고 묻는 것은 괜찮다. 그 답이 당신의 판단을 루프에 넣지 않은 채 당신의 아키텍처를 고르게 두는 것은 실수다.

안목이 새로운 핵심 역량이다

누구나 코드를 생성할 수 있게 되면, 희소한 역량은 어떤 생성된 코드가 좋은지 아는 것이다.

안목은 당신에게 이렇게 말해준다:

- 이 추상화는 시기상조다. 더 단순한 버전을 요청하라.
- 이건 데모는 처리하지만 실제 입력은 처리하지 못한다.
- 이건 기술적으로는 맞지만 유지보수의 악몽이다.
- 이건 거의 맞다—그리고 바뀌야 할 구체적인 지점은 여기서다.

안목은 옛날 방식으로 쌓는다: 많은 코드를 읽고, 무언가를 출시하고, 무엇이 깨지는지 지켜보면서. AI는 그 경험을 대체하지 않는다—오히려 더 가치 있게 만든다. 이제 당신의 판단이 병목이자 차별점이기 때문이다. 안목 있는 사람은 견고한 소프트웨어를 빠르게 출시한다. 안목 없는 사람은 빠르게 출시한 다음 버그에 빠져 허우적댄다.

불편한 따름정리가 여기 있다: AI는 모두의 바닥을 끌어올리고, 이는 곧 문법을 아는 것의 이점을 평평하게 만들고 판단을 갖춘 것의 이점을 날카롭게 만든다. 배움을 건너뛰려고 AI에 기대는 주니어는 정체될 것이다. 출력이 틀렸을 때를 알려주는 안목을 결코 쌓지 못하기 때문이다. AI를 더 많이 출시하고 더 많이 읽는 데 쓰는 사람은 복리로 성장한다. 같은 도구, 정반대의 결과—그리고 그 차이는 전적으로 돌아온 것에 계속 관여하느냐에 달려 있다.

현실적인 기대치

나중에 실망하지 않도록 지금 이것들을 설정하라:

- **당신은 많은 코드를 읽게 될 것이다.** 검토가 곧 일이다. 코드 읽기가 싫다면, 이것은 그것을 우회하는 지름길이 아니다.
- **사소하지 않은 것은 한 방에 안 된다.** 루프를 몇 번 돌 것을 예상하라. 그것은 정상이지, 실패가 아니다.
- **여전히 당신의 책임이다.** AI가 쓴 코드가 프로덕션에서 깨지면, “모델이 썼다”는 변명이 되지 않는다. 출시한 건 당신이다.
- **빨라지는 것이지, 공짜가 되는 것이 아니다.** 도구는 놀랍다. 마법은 아니다. 판단은 여전히 당신의 몫이다.

그리고 이 책이 무엇이 아닌지에 대해 직설적으로 말하자면: 이것은 당신이 이해하지 못하는 소프트웨어를 출시하는 길이 아니고, 일확천금 비책도 아니다. 바이브 코딩을 “코딩을 배우지 않고 앱 만들기”로 파는 사람은 누구든 장난감 버전을—실제 사용자와 접촉하는 순간 무너지는 그 버전을—파는 것이다. 이 기술은 그것을 엔지니어링 판단을 대체하는 것이 아니라 그 위에 없는 지렛대로 다루는 사람에게 복리로 쌓인다.

바이브 코딩의 약속은 진짜다: 혼자일 때보다 더 많이, 더 빠르게, 더 많은 영역에서 만들 수 있다. 하지만 이것은 계속 관여하는 엔지니어—지휘하고, 검토하고, 다듬는 사람—에게 보상하고, 손을 떼고 나타나는 것을 그대로 받아들이는 사람을 벌한다.

구경꾼이 아니라 테크 리드가 되어라. 이 책의 나머지는 그 방법에 관한 것이다.

정리와 실습

핵심 요점

- 바이브 코딩은 무엇을 왜 만드는지에 대한 주도권을 줬 채 AI에게 소프트웨어 구축을 지시하는 것이지, 설명할 수 없는 코드를 무작정 출시하는 것이 아니다.
- 이 모든 것을 지탱하는 규칙은 출시하는 것을 이해하라이다. 직접 타이핑할 필요는 없지만, 변호하고 고칠 수는 있어야 한다.
- 병목은 생산 (빠른 타이핑)에서 명세와 검토 (명확하게 기술하고 비판적으로 판단하기)로 옮겨갔다.

- 의도 □ 생성 □ 검토 □ 개선의 루프 안에서 살되, 실제 엔지니어링이 일어나는 검토 단계를 절대 건너뛰지 마라.
- 모델의 자신감은 신호가 아니다. 당신의 검증이 신호다. 안목과 판단력이 희소하고 복리로 쌓이는 역량이다.

해보기

진짜로 이해하는 작은 함수 하나를 골라라—“비밀번호가 정책을 충족하는지 검증”또는“바이트 수를 사람이 읽을 수 있는 문자열로 포맷”같은 것. 정밀한 프롬프트를 작성하고 (시그니처, 규칙, 엣지 케이스, 테스트 요청), 생성한 뒤, 의도적으로 검토 단계를 실행하라. 최소한 하나의 엣지 케이스를 손으로 따라가며 코드가 처리하는지 확인하라. 처리하지 못한다면 “고쳐줘” 대신 정밀한 개선 프롬프트 (증상 + 원인 + 해법) 를 작성하라. 목표는 완전히 평가할 수 있는 코드 위에서 전체 루프를 처음부터 끝까지 한 번 느껴보는 것이다.

이 장의 프롬프트

내 페어 프로그래밍 파트너 역할을 해줘. 작고 독립적인 함수 하나를 설명할게. 코드를 작성하기 전에, 내가 확인할 수 있도록 요구사항을 다시 정리하고 네가 처리할 엣지 케이스를 나열해줘. 그런 다음 그 엣지 케이스를 다루는 단위 테스트와 함께 함수를 작성해줘.

함수: < 한 줄 설명 >

입력 / 출력: < 타입과 형태 >

제약: < 스택, 사용하거나 피할 라이브러리, 성능 제한 >

코드 다음에, 내가 검증해야 할 가정이 있으면 짚어줘.
요구사항을 만족하는 가장 단순한 버전으로 유지해줘.

AI 작업 공간 구축하기



바이브 코딩은 파트너십이다. 당신은 의도와 판단을 가져오고, AI 는 속도와 지치지 않는 타이핑을 가져온다. 하지만 어떤 파트너십이든 그렇듯, 공유된 환경, 명확한 기본 규칙, 그리고 실수해도 안전한 공간이 있을 때 가장 잘 작동한다. 이 장은 단 하나의 기능을 작성하기 전에 그 환경을 구축하는 것에 관한 것이다.

좋은 작업 공간은 세 가지를 한다: AI 가 똑똑한 변경을 하는 데 필요한 맥락을 제공하고, 그 변경의 결과를 빠르게 볼 수 있게 하며, 모든 변경을 쉽게 되돌릴 수 있게 한다. 이것들을 제대로 갖추면 AI 는 시니어 협업자처럼 느껴진다. 잘못 갖추면 슬롯머신처럼 느껴진다. 그 차이는 좀처럼 모델 때문이 아니다—당신이 모델을 들여놓은 방 때문이다. 테스트도, 컨텍스트 파일도, 버전 관리도 없는 혼란스러운 리포에 들어간 유능한 에이전트는 허우적댄다; 깨끗하고 잘 계획된 프로젝트에 들어간 평범한 에이전트는 당신을 놀라게 할 것이다. 셋업이 곧 배울기다.

이 가운데 어떤 것도 한 번 하고 잊어버리는 잡일이 아니다. 작업 공간을 끊임없이 튜닝하는 무언가로 다뤄라: AI 가 나쁜 쪽으로 당신을 놀라게 할 때마다, 빠진 맥락 조각이나 느린 피드백 루프, 혹은 안전하지 않은 기본값이 그 일을 허용한 건 아닌지 물어라—그런 다음 코드 한 줄이 아니라 방을 고쳐라.

AI 네이티브 에디터 또는 어시스턴트 선택하기

당신의 첫 번째 결정은 AI 가 어디에 사는가다. 오늘날 대략 세 가지 범주가 있으며, 영원히 묶일 필요는 없다.

- **AI 네이티브 에디터** (Cursor, Windsurf): AI 가 깊이 통합된 익숙한 코드 에디터. AI 가 옆에서 작업하는 동안 코드를 보고 만지고 싶다면 최선이다.
- **터미널/CLI 에이전트** (Claude Code, Codex CLI, Gemini CLI, Aider): 버튼을 클릭하는 대신 명령을 입력하는 텍스트 전용 제어판인 CLI(커맨드라인 인터페이스)에서, AI 가 파일을 읽고, 편집하고, 명령을 실행하는 에이전트로 동작한다. AI 가 여러 파일에 걸쳐 더 자율적으로 작업하게 하는 데 최선이다.

- **IDE 플러그인** (GitHub Copilot, Continue, JetBrains AI): 프로그래머가 코드를 작성하는 올인원 앱인 IDE(통합 개발 환경) 안에, 이미 좋아하는 에디터에 덧붙이는 보조 도구.

실제 소프트웨어를 출시하려면 **프로젝트 전체를 읽고, 여러 파일을 편집하고, 명령 (테스트, 빌드, 린터) 을 스스로 실행할 수 있는 것**을 골라라. 자신의 작업을 실행하고 점검할 수 있는 능력이 바로 유용한 어시스턴트와 자동완성 장난감을 가르는 지점이다. 다음 줄만 제안할 수 있는 도구는 모든 검증을 당신 손으로 하게 만든다; 방금 바꾼 테스트를 직접 실행할 수 있는 도구는 스스로 루프를 닫는다.

그 핵심 역량을 넘어, 정착하기 전에 따져볼 만한 몇 가지가 있다:

- **컨텍스트 창과 프로젝트 인식**—관련 파일을 실제로 불러올 수 있는가, 아니면 작업 중반에 대화의 시작을 잊어버리는가?
- **권한 모델**—명령을 실행하고 파일을 편집하기 전에 묻는가, 아니면 먼저 행동하는가? 둘 다 쓸 자리가 있다; 당신이 어느 모드에 있는지 알아둬라.
- **비용과 한도**—자율 에이전트는 사용량을 빠르게 태울 수 있다. 한 시간씩 돌리기 전에 가격 정책을 이해하라.

선택을 너무 고민하지 마라. 이 책의 기술들은 도구 간에 옮겨간다. 노련한 바이브 코더 대부분은 결국 두 개를 설치해 둔다—대량 작업용 빠른 터미널 에이전트와 리뷰용 GUI 에디터—그리고 이 장을 닫는 TUI vs GUI 절이 바로 그 쪽에 다시 돌아온다. 여기서 GUI(그래픽 사용자 인터페이스)는 창과 버튼으로 클릭하는 익숙한 프로그램이고, TUI(텍스트 사용자 인터페이스)는 터미널에서 동작하는 텍스트 전용 사촌이다.

AI 가 탐색할 수 있도록 프로젝트 구조 잡기

AI 모델은 당신의 코드를 읽음으로써 추론한다. 지저분하고, 산만하고, 일관성 없는 프로젝트는 당신에게 어려운 만큼 그들에게도 어렵다. 완벽한 아키텍처가 필요한 건 아니지만, 몇 가지 습관은 즉시 보상을 준다:

- **예측 가능한 레이아웃을 유지하라.** 소스는 한 곳에, 테스트는 소스 옆이나 소스를 거울처럼 따르는 위치에, 설정은 루트에.
- **명확하고 서술적인 이름을 써라.** `calculateMonthlyInvoiceTotal` 이 `calc2` 보다 낫다. AI 는 이름을 단서로 쓴다.
- **더 작은 파일을 선호하라.** 200 줄짜리 파일은 2,000 줄짜리보다 정확하게 바꾸기 쉽다. 둘 모두에게.
- **관련된 것들을 한곳에 모아라.** 한 기능의 코드, 스타일, 테스트가 함께 있으면, AI 는 한 번 읽기로 맥락을 모은다.
- **명확한 진입점을 유지하라.** 앱이 시작되는 분명한 한 곳은 AI 가 프로젝트를 파악할 때 당겨볼 실마리를 준다.

전형적이고 AI 친화적인 레이아웃은 다음과 같은 모습일 수 있다:

```
my-app/
├── AGENTS.md           # AI 를 위한 프로젝트 규칙 & 컨텍스트
├── README.md           # 프로젝트 소개, 실행 방법
├── .env.example         # 필요한 시크릿 문서화 (실제 값 없음)
├── .gitignore           # .env, 빌드 산출물, node_modules 제외
├── package.json         # 스크립트: dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/    # 한 기능의 코드 + 테스트, 함께
│   ├── lib/             # 공용 헬퍼
│   └── index.ts         # 진입점
└── tests/              # 횡단 / 통합 테스트
```

그 보상은 복리로 불어난다: 프로젝트가 AI 가 탐색하기 쉬울수록, AI 의 변경은 더 작고 더 외과적으로 된다. 잘 정돈된 리포는 AI 가 서른 개를 헤집어 추측하는 대신 관련된 세 개의 파일만 건드리게 한다—그러니 더 작은 diff, 더 적은 사고, 그리고 당신이 실제로 읽을 수 있는 리뷰가 된다. AI 가 광범위하고 초점 없는 편집을 하고 있다면, 진짜 범인은 프롬프트가 아니라 구조인 경우가 많다.

컨텍스트 파일 작성하기 (AGENTS.md)

당신이 할 수 있는 가장 지렛대 효과가 큰 일은 AI 가 자동으로 읽는 **컨텍스트 파일**을 작성하는 것이다. 도구마다 찾는 이름이 다르다—AGENTS.md, CLAUDE.md, .cursorsrules, 또는 .github/copilot-instructions.md —하지만 발상은 동일하다: 당신의 프로젝트가 어떻게 동작하고 당신이 AI 가 어떻게 행동하길 원하는지 알려주는 살아있는 문서.

이것이 없으면 AI 는 추측한다. 이것이 있으면 AI 는 당신의 코드 재포매팅을 멈추고, 쓰지도 않는 라이브러리를 지어내기를 멈추고, 이미 내린 결정을 다시 설명하기를 멈춘다. 이를 뛰어나지만 세션 사이에 완전한 기억상실에 걸리는 새 팀원을 위한 온보딩 문서라고 생각하라—그들이 생산적이 되는 데 필요한 모든 것은 그들이 매번 읽을 어딘가에 있어야 한다.

실용적인 시작 파일:

```
# Project: Invoice Manager

## What this is
A small web app for freelancers to create and track invoices.

## Tech stack
- TypeScript + React (Vite)
- Tailwind for styling
- SQLite via Prisma
- Vitest for tests

## Commands
- Install: `npm install`
- Run dev server: `npm run dev`
- Run tests: `npm test`
- Lint: `npm run lint`

## Conventions
- Use functional components and hooks; no class components.
- Keep components small; one component per file.
- Write a test for every new function in `src/lib`.
- Never edit files in `src/generated/` — they are auto-generated.

## Don'ts
- Don't add new dependencies without flagging it first.
- Don't commit secrets; use environment variables.
```

짧게 유지하고 최신으로 유지하라. 천 줄까지 불어난 컨텍스트 파일은—당신에게도 모델에게도—더 이상 꼼꼼히 읽히지 않으니, 추가하는 만큼이나 공격적으로 쳐내라. AI 가 같은 잘못된 가정을 반복할 때, 그것이 한 줄을 추가하라는 신호다; 규칙이 더 이상 현실과 맞지 않으면, 그것이 에이전트를 적극적으로 오도하기 시작하기 전에 지워라.

작동하는 컨텍스트 파일과 무시되는 컨텍스트 파일을 가르는 몇 가지 패턴이 있다:

- **구체적이고 명확하게 써라.** “우리 컨벤션을 써라”는 쓸모없다. “검증에는 zod 를 써라; 우리는 yup 을 쓰지 않는다”는 AI 가 실제로 따를 수 있는 규칙이다.
- **하지 말 것을 할 것만큼 크게 말하라.** 모델은 돕고 싶어 안달이고, 그래서 당신이 말리지 않으면 기꺼이 의존성을 추가하거나 설정을 다시 쓴다.
- **예시를 가리켜라.** “src/features/invoices 의 스타일에 맞춰라”는 해석할 모호한 지시 대신 베껴올 구체적인 모델을 AI 에게 준다.

컨텍스트 파일을, 그렇지 않았다면 매번 반복해야 했을 모든 교훈의 기록으로 다뤄라. 그것은 이 장 전체에서 가장 값싸고 가장 오래가는 투자다.

버전 관리를 반사 신경으로 만들기

버전 관리는 당신의 안전망이며, AI 와 함께라면 필수가 된다. AI 는 5 초 만에 열 개의 파일을 바꿀 수 있다—그 작업물을 점검하고, 보관하고, 버릴 깔끔한 방법이 필요하다.

- **자주, 작은 단위로 커밋하라.** 큰 변경을 요청하기 전에, 작동하는 것을 커밋하라. 깨끗한 작업 트리는 언제나 그 지점으로 돌아갈 수 있음을 뜻한다.
- **커밋 전에 diff 를 읽어라.** 이것이 당신의 진짜 검토 단계다. 모든 줄을 이해할 필요는 없지만, 무엇이 왜 바뀌었는지는 이해해야 한다.
- **위험한 작업에는 브랜치를 써라.** 브랜치를 띄우고, AI 가 마음껏 날뛰게 하고, 결과가 마음에 들 때만 머지하라. 아니면 삭제하라—손해 없다.
- **정직한 커밋 메시지를 써라.** 미래의 당신과 AI 둘 다 의도를 이해하려고 히스토리를 읽는다.

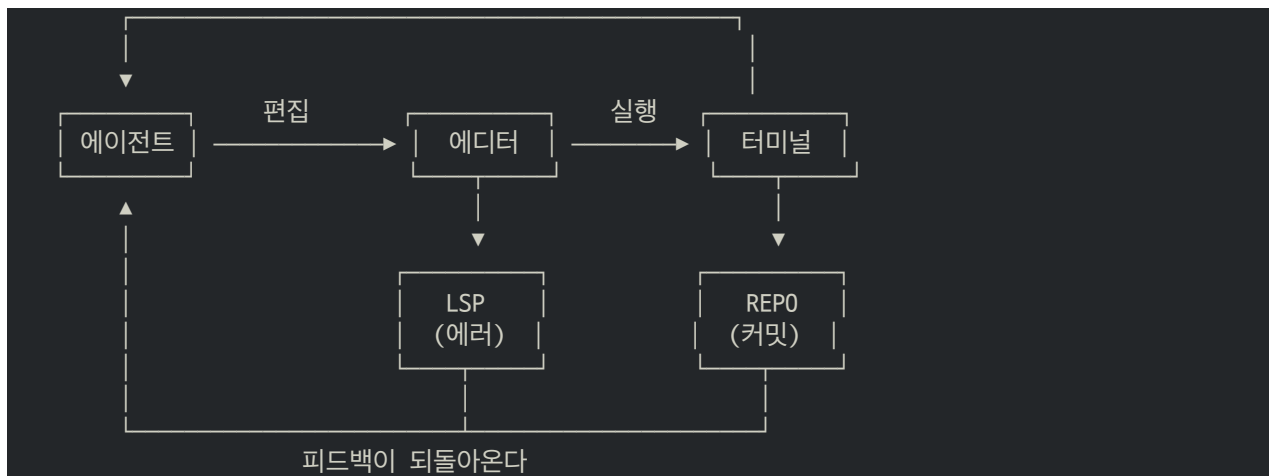
멘탈 전환: 커밋은 관료주의가 아니라 게임 속 체크포인트다. 보스전 전에 저장하라. 그리고 깨끗한 히스토리를 지킬 가치가 있게 만드는 두 개의 탈출구를 익혀라—커밋하지 않은 실험을 버리는 `git checkout .` (또는 `git restore .`), 그리고 이미 만든 커밋을 되돌리는 `git revert`. 언제나 알려진-양호한 상태로 돌아갈 수 있다는 걸 아는 것, 바로 그것이 불안 없이 AI 에게 긴 고삐를 내줄 수 있게 한다.

피드백 루프를 촘촘하게 유지하기

변경이 작동하는지 더 빨리 볼 수 있을수록, 당신과 AI 는 더 빨리 올바른 것으로 수렴한다. 느린 루프—무언가 깨졌는지 알기까지 몇 분을 기다리는—은 모든 실수를 조용히 배가시킨다.

- **“실행”을 단일 명령으로 만들어라.** `npm run dev`, `make test`, 무엇이든, 컨텍스트 파일에 문서화해 AI 도 실행할 수 있게 하라.
- **빠른 테스트에 기대라.** 단위 테스트 몇 개만 있어도 AI 가 당신의 감시 없이 자신의 작업을 검증할 방법이 생긴다.
- **린터와 포매터를 써라.** 이들은 한 부류의 오류를 즉시 잡아내고, 스타일 논쟁을 당신의 검토에서 빼준다.
- **오류를 AI 에게 되돌려 보여줘라.** 무언가 깨지면 실제 오류 메시지를 붙여 넣어라. 구체적인 피드백은 구체적인 수정을 낳고, “안 돼요”는 추측을 낳는다.

AI 작업 공간의 조각들은 하나의 연결된 루프를 이룬다—에이전트가 리포를 편집하고, 에디터와 언어 서버가 문제를 보고하고, 터미널이 코드를 실행하며, 모든 결과가 에이전트에게로 되돌아온다:



이 발상의 가장 깊은 버전은 AI 가 스스로 루프를 닫게 하는 것이다. 당신이 명령을 실행하고, 실패를 읽고, 그것을 전달하는 대신, 에이전트에게 변경마다 테스트와 타입체커를 실행하고 본 것에 반응할 권한을 줘라. 사이클은 이렇게 된다: 편집하고, 실행하고, 실패를 읽고, 고치고, 다시 실행한다—모두 당신이 중간에 끼지 않은 채로. 당신의 역할은 배달부에서 감독으로 옮겨간다: 목표와 가드레일을 정하고, 루프가 도는 것을 지켜본다. AI 가 편집하고, 실행하고, 실패를 보고, 스스로 고칠 수 있게 되면, 당신은 감독석에 머무는 동안 진짜 일을 해내는 루프를 구축한 것이다.

시크릿을 신중하게 관리하기

AI 도구는 당신의 파일을 읽고, 때로는 원격 서비스와 맥락을 공유한다. 그래서 시크릿 유출은 실제 위험이다. 첫날부터 좋은 습관을 들여라:

- **실제 키를 코드에 절대 넣지 마라.** 환경 변수와 .env 파일을 써라.
- **항상 .env 를 gitignore 하라.** 빈 자리표시자가 있는 .env.example 을 커밋해 AI 가 값을 보지 않고도 무엇이 필요인지 알게 하라.
- **키의 범위를 좁게 잡고** 노출됐을 수 있는 것은 무엇이든 교체하라.
- **컨텍스트 파일에서 AI 에게** 시크릿을 절대 하드코딩하거나 커밋하지 말라고 일러줘라.

AGENTS.md 에 Never read or print the contents of .env 같은 강력한 규칙은 값싼 보험이다. 키처럼 보이는 무언가가 담긴 커밋을 거부하는 pre-commit 훅이나 시크릿 스캐닝 도구도 마찬가지다—유출의 비용은 비대칭적이라 여기서는 다층 방어가 중요하다. 공개 리포에 푸시된 키는, 당신이 몇 초 뒤에 지우더라도, 그것이 도착하는 순간 이미 탈취된 것이다. 봇이 실시간으로 커밋을 긁어가기 때문이다. 의심스러우면 노출된 키는 폐기된 것으로 간주하고 교체하라; 키 교체는 1 분이면 되지만, 유출된 키의 뒷수습은 며칠이 걸릴 수 있다.

온전한 로컬 환경이 AI 변경을 안전하게 만든다

실수가 값싸고 복구 가능한 환경으로 한데 묶어라:

- main 에 직접이 아니라 **브랜치**에서 작업하라.
- 큰 변경 **전에** 작업 트리를 **깨끗하게** 유지해 diff 가 의미 있게 하라.
- 테스트와 개발 서버를 단일 명령으로 **실행 준비** 상태로 두어라.
- 시크릿을 **리포지토리 밖에** 두고 컨텍스트 파일을 **최신으로** 유지하라.

이 토대가 있으면, 어떤 AI 변경에서든 최악의 경우는 git checkout . 과 새 출발이다. 그 안전함이 바로 당신이 빠르게 움직이고 대담하게 지휘하게 해주는 것이다—AI 가 하는 어떤 것도 진정으로 되돌릴 수 없는 것은 없음을 알기에. 다음 장에서는 이 작업 공간을 실제로 무언가를 만드는 데 쓰기 시작할 것이다.

실시간 진단 (LSP)

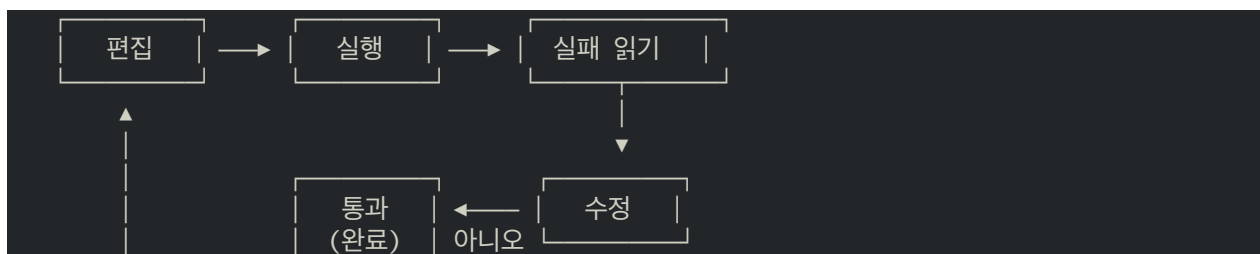
가장 빠른 피드백 루프는 코드를 실행할 때까지 기다리지 않는다—타이핑하는 순간 실수를 잡아낸다. 그것이 바로 **언어 서버 프로토콜 (LSP)** 이 제공하는 것이다: 에디터 뒤의 실시간 “지능”으로, 실시간 **에러**, **타입 정보**, **정의로 이동**, **자동완성** 을 포함한다. 언어 서버는 코드가 바뀌는 대로 읽어들이고 무언가 실행되기 전에 문제를 보고한다.

바이브 코딩에서의 지렛대는 이 **진단 결과를 AI 에게 되먹이는 것**이다. 이게 없으면 루프가 느리다: AI 가 코드를 쓰고, 당신이 실행하고, 오타나 타입 에러로 죽고, 당신이 직접 메시지를 붙여넣는다. 진단이 연결되어 있으면, 에이전트는 그 줄을 쓰는 순간 당신이 보는 것과 똑같은 빨간 밑줄을 보고—코드가 실행되기도 전에 타입이나 린트 에러를 진행하면서 고친다.

에이전트에게 이 시야를 주는 몇 가지 방법:

- 당신의 스택에 맞는 언어 서버를 실행하라 (tsserver, rust-analyzer, pyright 등) —대부분의 AI 네이티브 에디터는 이를 알아서 해준다.
- 에이전트가 타입체크나 린트 명령을 실행해 출력을 읽게 하거나, MCP 서버를 통해 진단을 노출하라.
- 깨끗한 진단 결과를 나중 일이 아니라 “완료”의 일부로 취급하라.

이렇게 연결하면 에이전트는 스스로 자기 교정 사이클을 돈다—편집하고, 실행하고, 실패를 읽고, 고치고, 진단이 깨끗하게 돌아올 때까지 다시 실행한다:



여전히 실패

이것이 또한 타입이 있는 언어가 바이트 코딩에서 조용한 초능력이 될 수 있는 이유다: 모든 타입 주석은 AI 의 실수가 실행되기도 전에 잡아내는, 작고 기계가 검사하는 명세다. 실행-크래시-붙여넣기 사이클이 조용한 자기 교정 사이클이 된다. AI 는 덜 헤매고 더 빨리 수렴한다.

에이전트 확장하기

기본 상태의 AI 어시스턴트는 범용적이다. 같은 베이스 모델도 주위에 툴링을 두르면 훨씬 나은 협업자가 된다—평범한 챗봇을 당신 프로젝트에 맞춘 **하니스**로 바꾼다. 알아둘 주요 확장 지점:

- **스킬 (Skills)** —에이전트가 필요할 때 불러오는 재사용 가능한 패키화된 역량 (지침 + 스크립트) 으로, 매 세션 다시 설명할 필요 없이 도메인 노하우를 쓸 수 있게 한다.
- **플러그인 (Plugins)** —스킬, 명령, 훅, 커넥터를 하나의 설치 가능한 단위로 묶어 배포한다.
- **MCP(Model Context Protocol)** —에이전트를 외부 도구·데이터 (데이터베이스, API(애플리케이션 프로그래밍 인터페이스—다른 소프트웨어가 제공하는 표준 명령 메뉴), 파일 시스템) 에 하나의 공통 인터페이스로 연결하는 개방형 표준이다.
- **서브에이전트 (Subagents)** —메인 에이전트가 집중적인 작업 (검색, 리뷰, 긴 결다리 작업) 을 위해 띄우는 특화된 보조 에이전트로, 각자 자기 컨텍스트 창을 가진다.
- **훅 (Hooks)** —생명주기 이벤트 (명령 실행 전, 파일 편집 후) 에 자동으로 발동하는 스크립트로, 규칙을 강제하거나 컨텍스트를 결정론적으로 주입할 수 있게 한다.
- **슬래시 명령 (Slash commands)** —/이름으로 호출하는 짧은 이름의 워크플로우, 반복되는 다단계 요청을 한 번의 입력으로 줄인다.

첫날부터 이 전부가 필요하진 않다. 평범하게 시작하고, 같은 셋업을 반복하고 있다는 걸 알아챈 순간 스킬이나 슬래시 명령을 추가하라. 유용한 어림법: 같은 다단계 지시를 세 번째 타이핑하는 순간, 그것을 슬래시 명령으로 만들어라; 에이전트가 닿지 못하는 도구에 닿았으면 하고 처음 바라는 순간, 그것을 위한 MCP 서버를 찾아라. 특히 훅은 가드레일 장에서 AI 가 기억하기를 바라는 대신 규칙을 강제하는 방법으로 다시 등장한다.

macOS vs Windows

대부분의 AI 툴링—CLI 에이전트, 스크립트, 튜토리얼—은 은연중에 Unix 셸을 가정한다. 그것이 개발 환경 선택의 현실을 좌우한다.

- **macOS** 는 **Unix 네이티브**다. 진짜 POSIX 셸과 나머지 개발 세계가 쓰는 것과 같은 git, ssh, curl 을 기본 제공한다. 대부분의 에이전트와 스크립트가 “그냥 된다”.
- **Windows** 도 충분히 유능하지만, 가장 매끄러운 길은 **WSL2**(Windows Subsystem for Linux) 다: Windows 안의 진짜 리눅스 환경이다. 네이티브 PowerShell 의 괴벽과 싸우는 대신 WSL2 안에서 AI 코딩을 하라.
- **Linux** 는 이 가족의 세 번째 일원이며 이 목적에 한해서는 거의 정확히 macOS 처럼 동작한다—그 자체가 Unix 이므로, 같은 에이전트와 스크립트가 손대지 않고 돈다. 이미 그 위에 있다면, 당신은 준비된 샘이다.

이 모두에서 작업을 이식 가능하게 유지하기 위한 몇 가지 함정:

- **경로**—/home/you/project 대 C:\Users\you\project; 백슬래시와 드라이브 문자가 스크립트를 깨뜨린다. WSL2 가 대부분을 피해 간다.
- **줄 끝**—Windows 는 CRLF, Unix 는 LF 를 쓴다. 불일치는 시끄러운 diff 와 깨진 셸 스크립트를 낳는다; core.autocrlf 와 .gitattributes 로 LF 로 정규화하라.
- **대소문자 구분**—macOS 와 Windows 는 흔히 File.ts 와 file.ts 를 같은 것으로 취급하지만, 리눅스와 CI 는 아니다. 그래서 로컬에서 빌드되는 코드가 클라우드에서 실패할 수 있다. import 와 파일명의 대소문자를 일관되게 하라.
- **권한**—Unix 파일 모드 (chmod +x) 는 Windows 에 깔끔하게 대응되지 않는다.

어느 쪽에 있든, AI 에게 명령을 이식 가능하게—슬래시는 정방향, LF 줄 끝, 일관된 대소문자로—유지하라고 하라. 그래야 그 스크립트가 당신 노트북과 CI 에서 똑같이 돈다. AI 가 당신에게 없는 도구를 가정한 한 줄짜리 명령을 건넬 때, 가

장 빠른 해결책은 컨텍스트 파일에 당신의 OS 와 셸을 일러줘 더 이상 추측하지 않게 하는 것인 경우가 많다.

TUI vs GUI

AI 코딩 도구는 두 가지 형태로 오고, 생산적인 바이브 코더는 둘 다 손에 쏜다. **터미널/TUI 에이전트** (Claude Code, Codex CLI, Aider) 는 터미널에서 돈다: 빠르고, 스크립트 가능하고, 조합 가능하다—자동화와 AI 가 여러 파일에 걸쳐 자율적으로 일하게 하는 데 좋다. **GUI/에디터 도구** (Cursor, Windsurf, IDE 확장) 는 시각적 에디터 안에 산다: 발견하기 쉽고, 인라인 diff, 클릭 수락, 완만한 진입로—변경을 리뷰하기 더 편하다.

	TUI / 터미널	GUI / 에디터
속도	매우 빠름, 낮은 오버헤드	더 느림, UI 많음
스크립트 가능	예—파이프, 체인, 자동화	제한적
시각적 diff	평문	풍부, 인라인
발견 용이성	낮음 (명령을 알아야)	높음 (클릭 가능)
적합한 용도	자동화, 자율성, 파워 유저	리뷰, 온보딩, 시각 작업

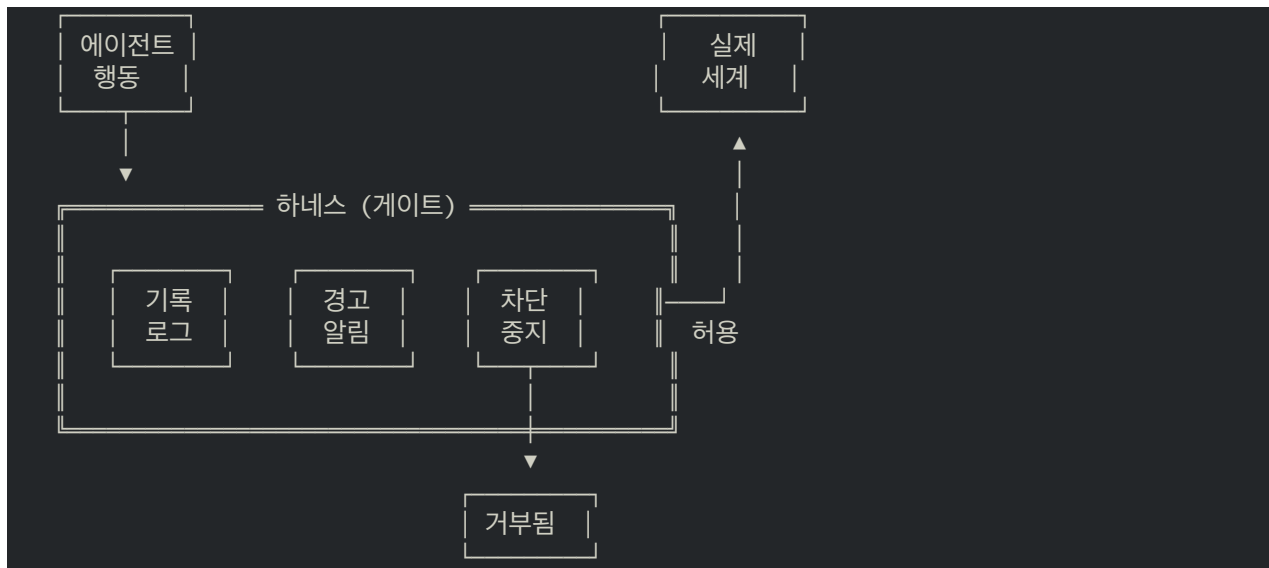
영원히 하나만 고를 필요는 없다. 조합이 핵심이다: 작업의 대부분은 빠른 TUI 에이전트로 몰고, 시각적 리뷰나 세밀한 편집을 위해 GUI 로 넘어가라. 흔한 워크플로는 터미널 에이전트가 여러 파일에 걸쳐 광범위한 변경을 하게 한 다음, GUI 의 좌우 비교 diff 를 열어 그것이 한 일을 실제로 읽고 승인하는 것이다—함에는 속도를, 점검에는 명료함을. 편한 주력을 고르되, 이 책의 기술은 둘 사이를 넘나든다는 걸 기억하라.

하네스 엔지니어링

습관은 당신이 기억해서 해야 하는 일이다. 작업 공간을 갖추는 다음 단계는 그 점검들을 AI 가 **건너뛸 수 없게** 만드는 것이다. 그게 하네스다.

하네스를 에이전트와 실제 세계 사이에 놓인 보안 검색대라고 생각하라. 위험한 단계마다 셋 중 하나를 한다: 무슨 일이 있었는지 기록하거나, 경고하거나, 그 행동을 아예 차단한다. 명령이 실행되기 전, 파일이 쓰이기 전, 프롬프트가 제출될 때, 커밋할 때, 푸시할 때—게이트가 있다. 새벽 2 시에 테스트 돌리는 걸 깜빡할 수 있는 당신과 달리, 에이전트는 그것을 우회할 수 없다.

에이전트가 하는 모든 위험한 행동은 먼저 하네스를 통과하며, 각 게이트는 그것이 실제 세계에 닿기 전에 기록하거나, 경고하거나, 차단할 수 있다:



왜 필수일까? 바이브 코딩의 핵심 베팅은 “AI 는 질주하게 두고, 당신은 조종하라”는 것이다. 그건 당신이 눈 깜빡하는 사이에 AI 가 절벽으로 달려가지 못할 때에만 통한다. 하네스는 안전을 경계심 (모든 디프를 읽고, 모든 규칙을 기억하는

것) 에서 코드 (자동으로 강제되는 규칙) 로 옮긴다. 진짜 사용자 앞에 내놓을 무언가라면, 그건 선택적 마감이 아니다—자율성을 안전하게 만드는 바로 그것이다.

좋은 하네스는 알아둘 만한 몇 가지 원칙을 따른다:

- **성공은 조용히, 실패는 시끄럽게**—괜찮을 때는 아무 말도 안 하므로, 경고에 의미가 남는다.
- **절대 자동 수정하지 않는다**—제안하거나, 차단하거나, 경고한다; 에이전트를 몰래 “고쳐서” 문제를 숨기지 않는다.
- **새 규칙을 더하기 전에 잠든 규칙을 폐기한다**—군더더기 없이 신뢰받는 상태를 유지하고, 낡은 점검의 더미가 되지 않게 한다.
- **설정 주도**—엔진은 하나, 프로젝트별 설정으로, 어느 저장소에든 끼워 넣을 수 있다.
- **추가 전용 (append-only) 로그**—에이전트가 실제로 한 일의 변조-방지 기록.

이걸 처음부터 만들 필요는 없다. 기성품으로 끼워 넣을 수 있는 하네스가 있다—dancinlab/harness는 코딩 에이전트의 도구 수명 주기에 연결할 수 있는 프로젝트 비종속 엔진 하나다. 솔직히 말하자면: 하네스가 AI 를 옹계 만들어주지는 않는다. AI 의 실수를 값싸게 잡아내게 해줄 뿐이다—빠르게 움직일 때 그것이 싸움의 대부분이다.

비용 안전: CLI vs API

청구서 하나가 프로젝트를 망칠 수 있고, 그건 호스팅에서 오는 일은 드물다. 종량제 AI API 에서 온다. 당신이 매일 코딩에 쓰려고 갖춰둔 도구—Claude Code, Cursor 같은—는 대개 **정액 구독제**다: 프롬프트 하나 더 보내는 비용은 사실상 0 이다. 그런데 당신의 앱이 원시 AI **API** 나 **SDK**(소프트웨어 개발 키트—제공업체가 주는 사전 포장된 코드 라이브러리) 를 호출하는 순간, 경제 구조가 뒤집힌다: **토큰당, 요청당** 청구되고, 상한이 없다. 버그 있는 재시도, 제한 없는 배치 작업, 혹은 루프를 도는 에이전트 하나가 0 달러짜리 유희 청구서를 하룻밤 사이에 네 자릿수 깜짝 청구서로 조용히 바뀔 수 있다. 그게 “청구 쇼크 (bill shock)”이고, 1 인 빌더가 데이는 가장 흔한 경로다.

유료 API 는 출시 전에 우리에게 가뭄라:

- **제공자 대시보드에 하드 지출 상한을 설정하라**—이메일 알림이 아니라 진짜 천장.
- **개발에는 정액제 CLI 를 선호하라**; 종량제 API 는 정말로 필요한 프로덕션 기능을 위해 아껴라.
- **자기 호출에 레이트 리미트를 걸어라**—동시성과 분당 요청 수를 제한하고—첫 며칠은 사용량 대시보드를 지켜봐라.
- **대량 작업에는 싸고 작은 모델을 쓰고, 먼저 아주 작은 입력으로 테스트하라** (전체 테이블이 아니라 레코드 하나).
- **유료 API 를 호출하는, 스로틀 없는 루프를 절대 출시하지 마라**. 모든 루프에 경계를 뒤라.

처음부터 규칙으로 삼아라: 모든 유료 API 에 하드 상한과 스로틀이 있다.

정리와 실습

핵심 요점

- 좋은 작업 공간은 AI 에게 컨텍스트, 빠른 피드백 루프, 값싼 되돌리기를 준다—설정이 곱셈기이며, 종종 모델 자체보다 더 중요하다.
- 컨텍스트 파일 (AGENTS.md / CLAUDE.md) 은 작성할 수 있는 가장 레버리지 높은 것이다. 구체적인 규칙, 큰 목소리의 금지 사항, 예시에 대한 포인터를 담아라.
- 버전 관리를 반사 신경으로 만들어라—작게 커밋하고, 디프를 읽고, 위험한 작업은 브랜치로—그러면 AI 가 하는 어떤 것도 진정으로 되돌릴 수 없는 것이 아니다.
- 루프를 타이트하게 유지하고 에이전트가 스스로 닫게 하라: 편집, 실행, 실패 읽기, 수정. 진단 (LSP, 린트, 타입체크) 을 AI 에게 되먹여라.
- 모든 유료 API 를 하드 상한과 스로틀로 가뭄라. 비밀을 저장소 밖에 두고, 하네스를 써서 안전 검사를 AI 가 건너 뛸 수 없게 만들어라.

해보기

작업 중인 프로젝트를 열거나 (또는 빈 프로젝트를 스캐폴딩하고) AGENTS.md 초안을 손으로 작성하라: 프로젝트가 무엇인지, 기술 스택, dev/test/lint 실행 정확한 명령어, 실제 컨벤션 세 가지, 그리고 최소 두 개의 큰 목소리 금지 사항 (예: “절대 .env 를 읽거나 출력하지 말 것”, “먼저 알리지 않고 의존성을 추가하지 말 것”). 그런 다음 AI 도구에 사소한 변경을 요청하고 파일을 존중하는지 지켜보라. 원하지 않은 일을 할 때마다 한 줄을 추가하거나 다듬어라. 코드만이 아니라 방식을 튜닝하는 것이다.

이 장의 프롬프트

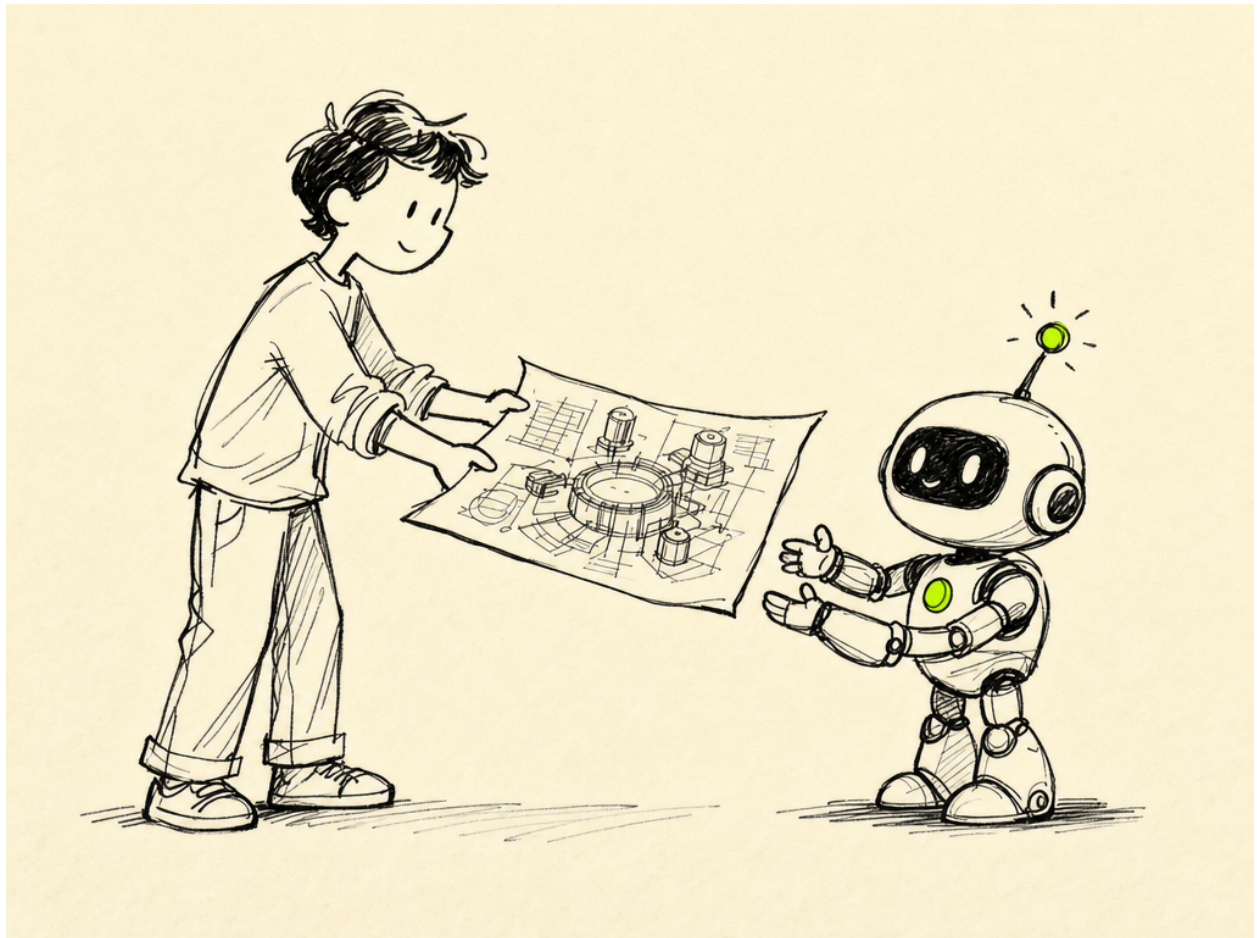
Read my project and help me write a concise AGENTS.md context file for it.

First, inspect the repo and tell me:

- the tech stack and versions you detect
- the commands for install / dev / test / lint / build
- the layout and where the entry point is

Then draft an AGENTS.md with these sections: What this is, Tech stack, Commands, Conventions, and Don'ts. Be specific and concrete (name the exact libraries and rules, not "follow our conventions"). State the don'ts as loudly as the do's, and point to one existing file as a style example. Keep it under ~40 lines — I'll prune from there.

엔지니어처럼 프롬프트하기



답답한 AI 세션과 생산적인 AI 세션의 차이는 좀처럼 모델에 있지 않다. 프롬프트에 있다. 대부분의 초보자는 AI 를 검색 엔진처럼 대한다—모호한 소원을 입력하고 최선을 바란다. 시니어 엔지니어는 그것을 날카롭지만 곧이곧대로 받아들이는 주니어 동료처럼 대한다: 빠르고 유능하지만, 당신의 실수를 포함해 당신이 설명한 그대로를 만들어내는 사람.

이 장은 그 간극을 메우는 것에 관한 것이다. 요령을 외우거나 “마법” 프롬프트 템플릿 폴더를 모을 필요는 없다. 좋은 엔지니어가 작업을 전달하는 방식으로 소통하면 된다: 맥락, 제약, 그리고 명확한 완료 정의와 함께. 아래의 모든 것은 그 하나의 기술의 변주다.

작업을 주기 전에 맥락을 줘라

AI 는 당신이 보는 방식으로 당신의 프로젝트를 볼 수 없다. 당신의 프레임워크 버전도, 파일 구조도, 이미 확립한 컨벤션도 모른다. 이것을 건너뛰면, AI 는 추측한다—그것도 형편없이. 더 나쁜 것은, 자신만만하게 추측한다는 점이다. 그래서 출력은 당신의 실제 코드베이스와 충돌하기 전까지는 그럴듯해 보인다.

같은 요청에 대한 이 두 프롬프트를 비교해보라.

이메일을 검증하는 함수를 추가해줘.

이것은 검증에 Express 와 Zod 를 쓰는 TypeScript 백엔드야.
우리는 이미 src/schemas/ 에서 Zod 스키마로 입력을 검증해.
그 스타일로 이메일 검증 스키마를 추가해줘. 이메일은
소문자여야 하고, 최대 254 자이며, src/config.ts 의 기존
BLOCKED_DOMAINS 목록에 있는 일회용 도메인을 거부해야 해.

첫 번째 프롬프트는 당신이 원하지 않는 정규식을 쓸 수도 있는, 당신의 코드베이스와 맞지 않는 스타일의 일반적인 코드를 낳는다. 두 번째는 그대로 붙여 넣을 수 있는 코드를 낳는다. 규칙은 이것이다: **스택, 컨벤션, 그리고 AI 가 재사용해야 할 기존 요소들을 진술하라.**

좋은 프롬프트는 이 조각들을 아래에서 위로 순서대로 쌓는다—맨 아래에 맥락, 맨 위 가까이에 실제 요청:

예시	입력 하나 → 출력
제약	하지 말 것
엣지 케이스	빈 값, 무효, 에러
입력 / 출력	타입과 형태
목표	명확한 작업 하나
맥락	스택, 파일, 패턴

기초가 먼저 ▲

엔터를 치기 전에 쓸모 있는 마음속 체크리스트: 언어와 프레임워크 (중요하다면 버전까지), 관련 파일이나 모듈, 프로젝트에 이미 있는 패턴, 그리고 AI 가 선호하거나 피해야 할 라이브러리. 매번 넷 다 필요한 건 아니다—하지만 답이 그중 하나에 따라 달라진다면, 그것을 명명하라. 확신이 없으면, 새 코드가 맞춰야 할 실제 시그니처, 타입, 또는 설정을 붙여 넣어라. 몇 줄의 기존 코드가 한 단락의 설명보다 당신의 컨벤션을 AI 에게 훨씬 잘 가르친다.

입력, 출력, 엣지 케이스를 명시하라

작업은 그 경계가 정의되기 전까지 정의된 것이 아니다. 시니어 엔지니어는“무엇이 들어가고, 무엇이 나오고, 일이 잘못 되면 무슨 일이 일어나는가”의 관점에서 생각한다. 그것을 프롬프트에 녹여라.

날짜 문자열을 파싱하는 함수를 작성해줘.

parseDate(input: string): Date | null 함수를 작성해줘.

입력: ISO 8601 문자열 ("2026-06-14", "2026-06-14T10:30:00Z").

출력: Date 객체, 문자열이 무효하면 null.

처리할 엣지 케이스:

- 빈 문자열 또는 공백 -> null
- 형식은 유효하나 불가능한 날짜 (2026-02-30) -> null
- 뒤에 쓰레기 값이 붙은 경우 ("2026-06-14xyz") -> null

throw 하지 마. 모든 무효 입력에 대해 null 을 반환해.

옛지 케이스를 명명하면, AI 는 그것들을 처리한다. 명명하지 않으면, AI 는 정상 경로를 작성하고 당신은 프로덕션에서 그 구멍을 발견한다. 같은 규율이 순수 함수 너머에도 적용된다: 엔드포인트라면, 성공 응답 과 함께 에러 응답도 명시하라 (중복 이메일에는 어떤 상태 코드? 잘못된 본문에는?). UI 컴포넌트라면, 채워진 상태뿐 아니라 빈 상태와 로딩 상태도 명시하라. 대부분의 소프트웨어에서 흥미로운 동작은 그 가장자리에 산다. 그러니 거기에 당신의 프롬프트가 단어를 써야 한다.

옛지 케이스가 무엇인지 확신이 없다면, AI 에게 먼저 물어라: “날짜 파서가 처리해야 할 옛지 케이스는 무엇인가?”—그런 다음 그 답을 제약으로 되먹여라. 이것은 모델을 코드 생성기에서 체크리스트 생성기로 바꾼다. 그리고 체크리스트를 검토하는 것이 당신이 잊은 케이스를 디버깅하는 것보다 빠르다.

작은 단계로 작업하라

가장 흔한 실수는 한 번에 너무 많은 것을 요청하는 것이다. “인증, 차트, 설정 페이지가 있는 사용자 대시보드를 만들어줘”는 검토할 수 없는 코드의 벽을 놓고, 읽지도 않은 파일들에 버그가 흩어진다.

작은 단계는 당신을 통제 안에 둔다:

- 프롬프트당 하나의 함수, 컴포넌트, 또는 엔드포인트를 출시하라.
- 넘어가기 전에 각 조각이 작동하는지 검증하라.
- 검증되지 않은 출력의 tap이 아니라, 작동이 확인된 코드 위에 쌓아라.

로그인, 가입, 비밀번호 재설정, 세션, 이메일 인증이 있는
전체 인증 시스템을 만들어줘.

인증 흐름의 1 단계: 가입 엔드포인트만 작성해줘.

{ email, password } 를 받는 POST /signup. bcrypt 로 비밀번호를
해시하고, 사용자를 저장하고, user id 와 함께 201 을 반환해.
User 모델과 db 클라이언트는 이미 있다고 가정해 (형태는 내가
보여줄게). 로그인은 다음에 처리하자.

더 작은 프롬프트는 더 작은 diff 를 뜻하고, 더 작은 diff 는 무엇이 바뀌었는지 실제로 읽을 수 있음을 뜻한다. 그것이 게임의 전부다. 두 번째 이득도 있다: 무언가 깨졌을 때, 작은 단계는 이분 탐색으로 잡아내기가 사소하다. 가입 엔드포인트를 생성하고 검증한 다음, 세션 로직, 그다음 비밀번호 재설정을 했다면, 3 단계 이후에 나타난 버그는 거의 확실히 3 단계에 산다. 빅뱅 생성은 그 신호를 버린다—버그가 당신이 한 번도 읽지 않은 400 줄 어디에든 있을 수 있다.

코드 전에 계획을 요청하라

사소하지 않은 무엇이든, 생각하기를 타이핑하기와 분리하라. AI 에게 접근 방식을 먼저 개요로 잡아달라고 요청하라. 이것은 잘못된 가정이 200 줄의 잘못된 코드가 되기 전에 잡아내며, 풀 리퀘스트를 고치는 것보다 계획을 고치는 것이 훨씬 싸다.

내 API 에 속도 제한을 추가하고 싶어. 코드를 쓰기 전에,
짧은 계획을 줘: 리미터가 어디에 있어야 하는지, 어떤 저장소가
필요한지, 제한에 걸리면 무슨 일이 일어나는지, 그리고 트레이드오프.
아직 코드는 쓰지 마 —접근 방식을 내가 먼저 승인할게.

계획이 맞아 보이면 “좋아, 1 단계를 구현해줘”라고 말하라. 아니라면, 당신은 코드를 풀어내느라 한 시간을 쓰는 대신 방향을 바로잡는 데 30 초를 쓴 것이다. 계획 먼저 세우기야말로 엔지니어식 프롬프트와 소원식 프롬프트를 가장 크게 가르는 습관이다.

계획 단계는 또한 AI 가 필요 이상으로 무거운 해법을 고르는 것을 잡아내는 곳이다—인메모리 맵으로 충분한데 Redis 를 끌어오거나, 한 줄짜리 라이브러리 호출로 이미 해결되는 것을 순수 짜거나. 한 가지 질문을 마음에 두고 계획을 읽어라: 이것이 작동할 수 있는 가장 단순한 것인가? 접근이 과잉 설계처럼 냄새난다면, 한 줄도 쓰이기 전에 평범한 말로 반박하라 (“이건 서버 한 대짜리 취미 프로젝트야—분산 저장소는 건너뛸 수 있을까?”).

재작성이 아니라 diff 로 반복하라

코드가 일단 존재하면, 전체를 다시 요청하려는 충동을 참아라. “더 낮게 만들어줘”같은 모호한 후속 요청은 작동하는 코드를 버리고 이미 고친 버그를 다시 들어온다. 원하는 구체적인 변경을 가리켜라.

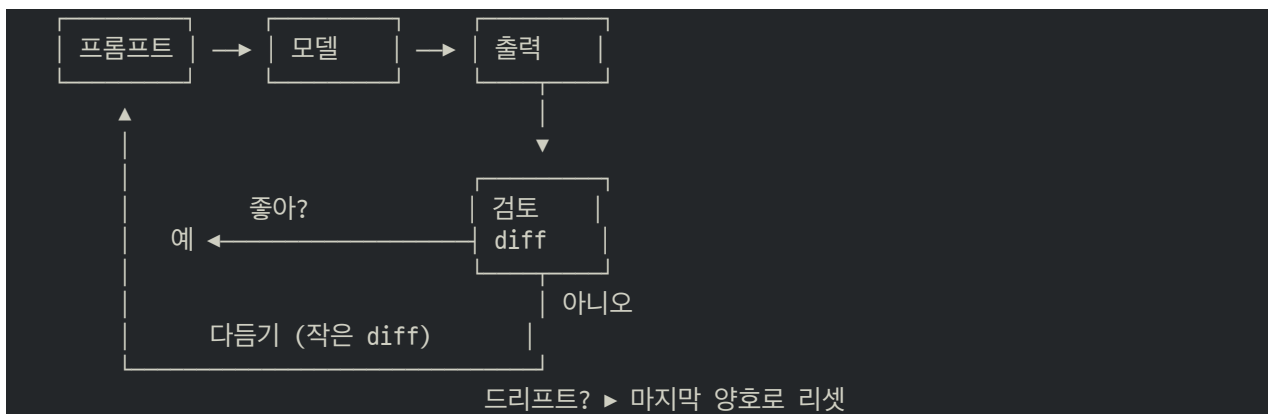
틀렸어, 전체를 다시 작성해줘.

함수는 작동하는데, 두 가지 문제가 있어:

1. `throw new Error`` 가 있는 줄의 에러는 `throw` 하는 대신 `Result` 타입을 반환해야 해 —위의 `validateUser` 함수의 패턴에 맞춰줘.
2. 루프가 매 반복마다 `items.length`` 를 다시 읽어; 밖으로 끌어내.

그 두 변경에 대한 diff 만 보여줘.

건강한 사이클은 촘촘한 루프다: 모델에 프롬프트하고, 출력을 읽고, 작고 표적화된 diff 로 다듬어라—절대 전체 재작성이 아니라—맞을 때까지:



(전체 재작성이 아니라) diff 를 요청하면 변경이 검토 가능하게 유지되고 이미 작동하는 부분이 보존된다. AI 의 출력을 동료의 풀 리퀘스트처럼 다뤄라: 특정 줄에 코멘트하고, 표적화된 편집을 요청하라. 변경이 었나가고 답변마다 코드가 원하던 것에서 점점 더 멀어진다면, 계속 땀질하지 마라—신뢰했던 마지막 버전으로 되돌리고 더 날카로운 설명으로 거기서부터 다시 프롬프트하라. 깨진 토대 위에서 앞으로만 반복하면 엉망진창이 가중될 뿐이다.

예시를 제공하라

구체적인 예시 하나가 한 단락의 설명만큼 값지다. 출력이 특정 형태이길 원한다면, 그 형태를 보여줘라. 이것이 모호함을 없애는 가장 빠른 방법이다.

데이터를 보기 좋게 포맷해줘.

이 사용자 배열을 요약 객체로 변환해줘.

입력:

```
[{ name: "Ann", role: "admin" }, { name: "Bo", role: "user" }]
```

원하는 출력:

```
{ total: 2, byRole: { admin: ["Ann"], user: ["Bo"] } }
```

변환 함수와 이 예시에 맞는 테스트를 작성해줘.

예시는 말만으로는 흐릿하게 남는 명명, 구조, 대소문자를 못 박는다. 가능하다면 예시를 테스트로 포함하라—그러면 “테스트를 통과시켜줘”가 모호하지 않은 지시가 된다. 이 요령은 일반화된다: 샘플 API(애플리케이션 프로그래밍 인터페이스—한 프로그램이 다른 프로그램에 데이터를 요청하는 방식) 응답은 파서를 못 박고, 전/후 한 쌍은 리팩터링을 못 박고,

스크린샷이나 ASCII 스케치는 레이아웃을 못 박는다. 어떤 형태에 대한 긴 설명을 쓰고 있는 자신을 발견할 때마다, 멈추고 그 형태의 한 사례를 대신 보여줘라.

AI 에게 하지 말아야 할 것을 말하라

제약은 당신이 원하는 결과에 관한 것만이 아니다—당신이 AI 가 피하길 원하는 수에 관한 것이기도 하다. 제약 없이 두면, 모델은 의존성을 추가하고, 당신이 건드려달라고 하지 않은 코드를 “친절하게” 리팩터링하고, 당신 요청의 가장자리에 기능을 지어내는 쪽으로 흘러간다. 몇 개의 부정 제약이 출력을 단단하게 유지한다.

검색 입력창에 디바운스를 추가해줘.

검색 입력창에 디바운스를 추가해줘.

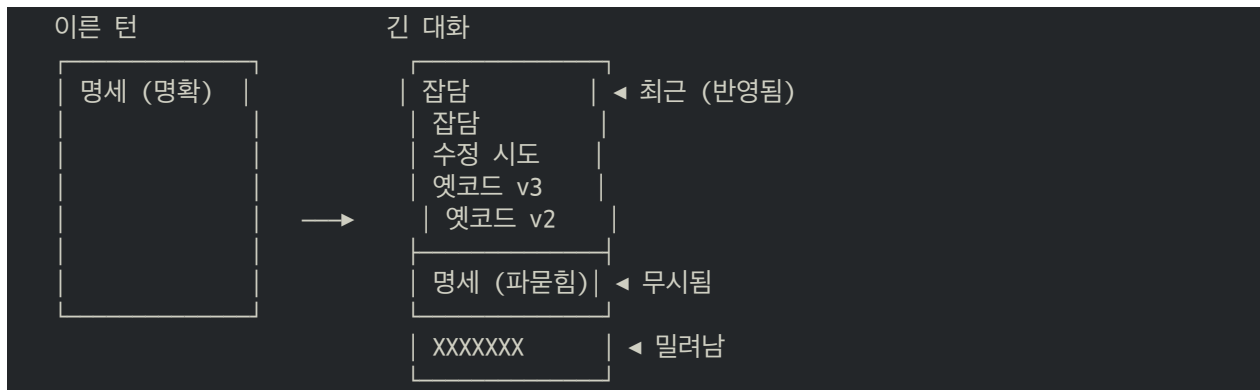
제약:

- 새 의존성 금지 —우리는 이미 lodash 가 있으니 그걸 써.
- SearchBar.tsx 만 건드려. 부모는 리팩터링하지 마.
- 기존 prop 이름과 타입은 그대로 유지해.
- 300ms 지연.

두 번째 프롬프트는 첫 번째가 잘못될 수 있는 열두 가지 길을 미리 막는다. 부정 제약은 기존 코드베이스에서 특히 값지다. 거기서 요청하지 않은 변경의 비용은 그것을 검토하는 것만이 아니다—머지 충돌, 놀란 동료, 그리고 잘 작동하던 코드의 회귀다. 넘지 말았으면 하는 선이 있다면, 명시적으로 말하라. AI 는 예의에서 당신의 경계를 추론하지 않는다.

컨텍스트를 자원처럼 관리하라

긴 대화는 공짜가 아니다. 스레드가 커질수록, 이전 지시는 파묻히고, 모델은 최근 메시지에 더 기대고, 스무 턴 전에 진술한 세부 사항은 조용히 중요하지 않게 된다. 컨텍스트 윈도우를 고정 크기의 상자로 그려보라: 대화가 커질수록 상자가 차오르고, 가장 이른 지시는 모델이 가장 덜 주목하는 바닥 쪽으로 밀려난다:



컨텍스트 윈도우를 무한한 기억이 아니라, 깨끗하게 유지할 책임이 당신에게 있는 작업 표면으로 다뤄라.

몇 가지 습관이 도움이 된다:

- **방향을 틀 때 명세를 다시 진술하라.** 열 메시지 전의 요구사항이 여전히 유효하다고 가정하지 마라. 현재 목표를 짧은 한 단락으로 요약하고 AI 가 그것으로부터 작업하게 하라.
- **새 문제에는 새로 시작하라.** 인증 코드에서 CSS(캐스케이딩 스타일 시트—페이지의 겉모습을 제어하는 언어) 버그로 전환할 때는, 새 대화를 열어라. 무관한 컨텍스트를 끌고 다니면 모델은 더 나아지는 게 아니라 더 나빠진다.
- **히스토리가 아니라 현재 코드를 붙여 넣어라.** 함수가 다섯 번 바뀌었다면, AI 는 다섯 개의 옛 버전이 필요하지 않다—지금 존재하는 하나가 필요하다. 현재 상태와 다음 변경을 줘라.
- **채팅 밖에 짧은 진실의 원천을 뒤라.** 더 긴 작업에는, 파일 안 몇 줄 (“스택, 핵심 결정, 끝난 것, 다음 것”) 이 스크롤 백에서 재구성하는 대신 몇 초 만에 새 대화를 다시 심게 해준다.

목표는 이 장 전체를 관통하는 그것과 같다: 매 순간, AI 는 더듬어 추측해야 하는 방대한 기록이 아니라, 정확하고 압축된 작업 그림으로부터 작업해야 한다.

피해야 할 흔한 안티패턴

이것들은 당신의 시간을 조용히 낭비시키는 프롬프트 습관이다. 당신 자신의 세션에서 이것들을 경계하라.

- **모호한 소원.** “좋은 로그인 페이지를 만들어줘.” 누구의 기준으로 좋은가? 형용사를 구체로 바꿔라: 어떤 필드, 어떤 검증, 어떤 프레임워크.
- **메가 프롬프트.** 기능 전체를 한 메시지에 쏟아붓기. 한 번에 하나씩 검증할 수 있는 단계로 쪼개라.
- **침묵하는 맥락.** AI 가 당신의 프로젝트를 기억하거나 안다고 가정하기. 관련 파일, 타입, 컨벤션을 다시 진술하라—특히 새 대화에서는.
- **눈감고 받아들이기.** 생성된 코드를 읽지 않고 붙여 넣기. AI 는 틀렸을 때조차 자신만만하다. 책임 있는 엔지니어는 여전히 당신이다.
- **끝없는 재시도.** 다른 출력을 바라며 “다시 해줘”를 거듭 실행하기. 두 번 시도해서 실패하면, 문제는 프롬프트다—그냥 다시 굴리지 말고 맥락이나 제약을 추가하라.
- **움직이는 표적.** 요구사항을 스레드 중간에 바꾸면서 다시 진술하지 않기. 방향을 틀 때 현재 명세를 요약해, AI 가 낡은 그림에서 작업하지 않게 하라.
- **아침 루프.** “당신 말이 전적으로 옳습니다!”를 수정이 맞다는 증거로 받아들이기. 동의는 검증이 아니다—코드를 직접 실행하고 동작을 확인하라.

핵심 마인드셋

여기 모든 기법은 하나의 발상으로 귀결된다: **당신의 마음을 읽을 수 없는 유능한 동료에게 말해야 할 것을 말하라.** 맥락은 앞에. 명확한 입력과 출력. 작고 검증 가능한 단계. 코드 전에 계획. 재작성보다 표적화된 편집. 말이 부족한 곳에는 예시. 건드리지 말아야 할 것에 대한 명시적 한계. 매 순간 깨끗하고 최신인 작업 그림.

당신은 마법 주문을 쓰는 게 아니다. 명세를 간결하고 정확하게 쓰고, 그 결과를 검토하는 것이다. 그것을 일관되게 하면 AI 는 슬롯머신처럼 느껴지기를 멈추고, 실제로 될 수 있는 그 빠르고 지치지 않는 엔지니어처럼 느껴지기 시작한다—그리고 그만큼 중요하게, 당신은 무엇이 출시되었는지 이해하는 사람으로 남는다.

정리와 실습

핵심 요점

- 작업 전에 컨텍스트를 줘라: 스택, 컨벤션, 재사용할 기존 조각을 명시하거나, 코드가 맞물려야 할 실제 시그니처를 붙여넣어라.
- 경계를 정의하라: 입력, 출력, 옛지 케이스. 소프트웨어의 흥미로운 동작은 가장자리에 살기 때문에, 거기에 단어를 써라.
- 작고 검증 가능한 단계로 작업하고, 코드 전에 계획을 요청하라. 풀 리퀘스트보다 계획을 고치는 게 훨씬 싸다.
- 전체 재작성이 아니라 표적화된 디프로 반복하고, 단어로 모호하게 남는 것은 구체적인 예시 (이상적으로는 테스트) 로 못 박아라.
- 컨텍스트 원도를 자원으로 관리하고, 하지 말아야 할 것을 말하라—명시적인 부정 제약이 출력을 타이트하게 유지한다.

해보기

평소에 입력했을 법한 모호한 한 줄 요청 (“리스트에 검색 추가”) 을 가져와, 무엇을 보내기 전에 엔지니어의 프롬프트로 다시 써라. 컨텍스트 (스택, 건드릴 파일), 입력/출력, 두세 개의 옛지 케이스, 원하는 결과의 예시 하나, 그리고 부정 제약 하나 (건드리지 말 것) 를 추가하라. 다시 쓴 버전을 보내고, 게으른 버전이 만들어냈을 것과 비교하라. 실제 요청 세 개에 대해 이렇게 하면 이 업그레이드가 자동이 된다.

이 장의 프롬프트

Context: < 언어 + 프레임워크 + 버전 >, < 파일/모듈 > 에서 작업 중.
We already < 재사용할 관련 기존 패턴 또는 라이브러리 >.

Task: < 만들 명확한 한 가지 >.

Inputs / outputs: < 타입과 형태 >.

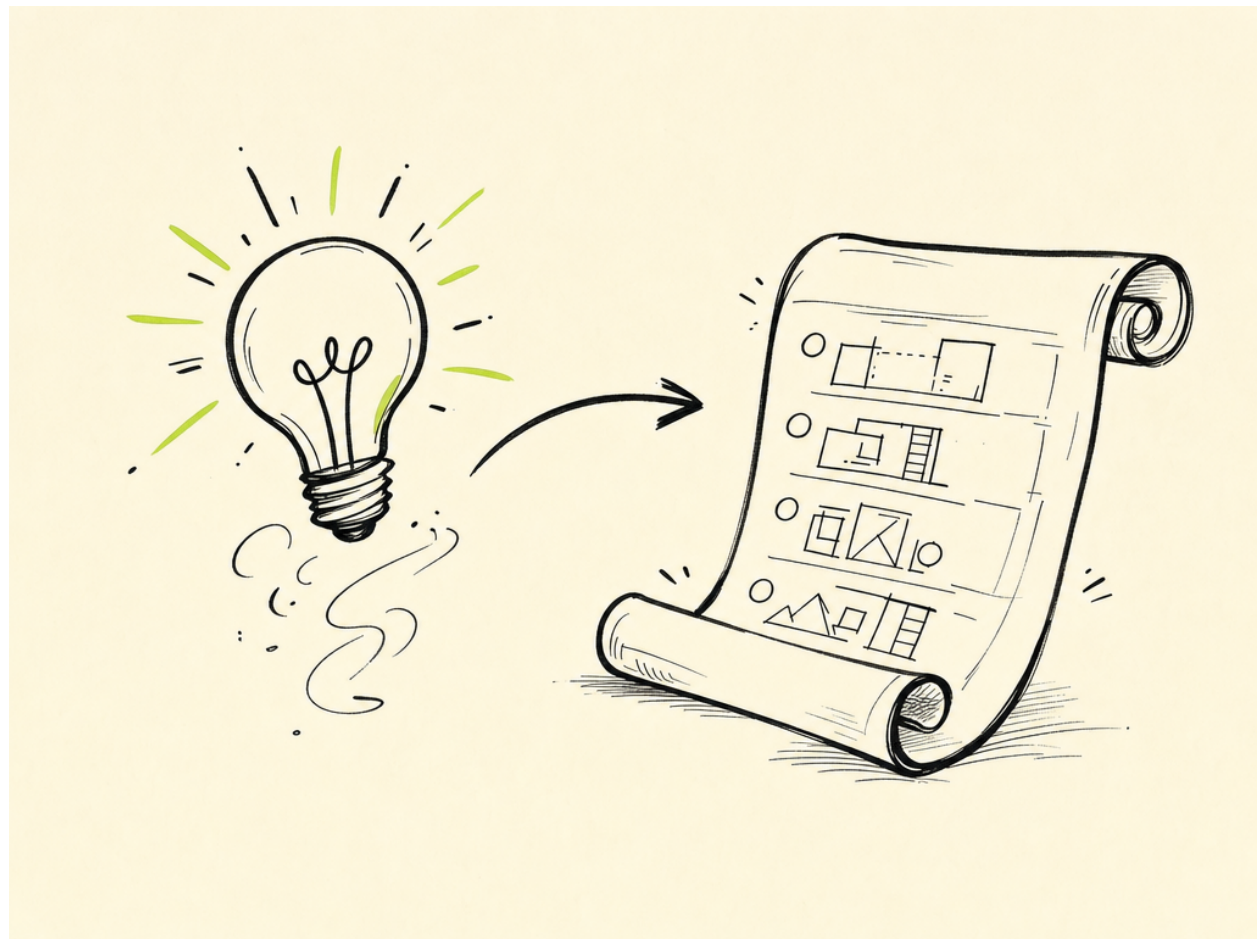
Edge cases to handle: < 중요한 것들을 나열 >.
Example: < 구체적인 입력 하나 -> 기대 출력 >.

Constraints:

- No new dependencies — < 이미 가진 것 > 을 써.
- < 이 파일 > 만 건드려; 다른 건 리팩터링하지 마.
- 기존 이름과 시그니처는 그대로 유지해.

Before coding, give me a 3-4 line plan and wait for my "go".

아이디어에서 명세까지, 몇 분 안에



모든 프로젝트는 흐릿한 생각에서 시작한다: “내 독서를 추적하는 데 도움이 되는 앱을 갖고 싶어.” 그것은 괜찮은 씨앗이지만, 거기서 공장 만들 수는 없다—당신의 AI 도 마찬가지다. 모호한 아이디어를 AI 에게 건네면 모호한 소프트웨어가 나온다: 요청하지 않은 기능들, 요청한 빠진 조각들, 그리고 이해할 수 없는 코드 더미.

해결책은 **명세 (spec)** 다—무엇을 만들고 무엇을 만들지 않는지를 말하는 짧고 평이한 언어의 문서. 좋은 소식: AI 를 사고 파트너로 두면, 한 줄짜리 아이디어에서 만들 수 있는 명세까지 며칠이 아니라 몇 분 만에 갈 수 있다. 이 장은 그 방법을 보여준다.

명세가 무엇이 아닌지도 분명히 해두자: 그것은 계약서가 아니고, 회의에서 변호해야 할 설계 문서도 아니며, 한 번 쓰고 올려버리는 것도 아니다. 그것은 당신 자신과 당신의 AI 에게 쓰는 작업용 메모다. 배워가면서 고쳐 쓰게 된다. 목표는 완벽한 문서가 아니다—다음 한 시간의 작업을 지난 시간보다 더 명확하게 만들어 주는 충분히 좋은 문서다.

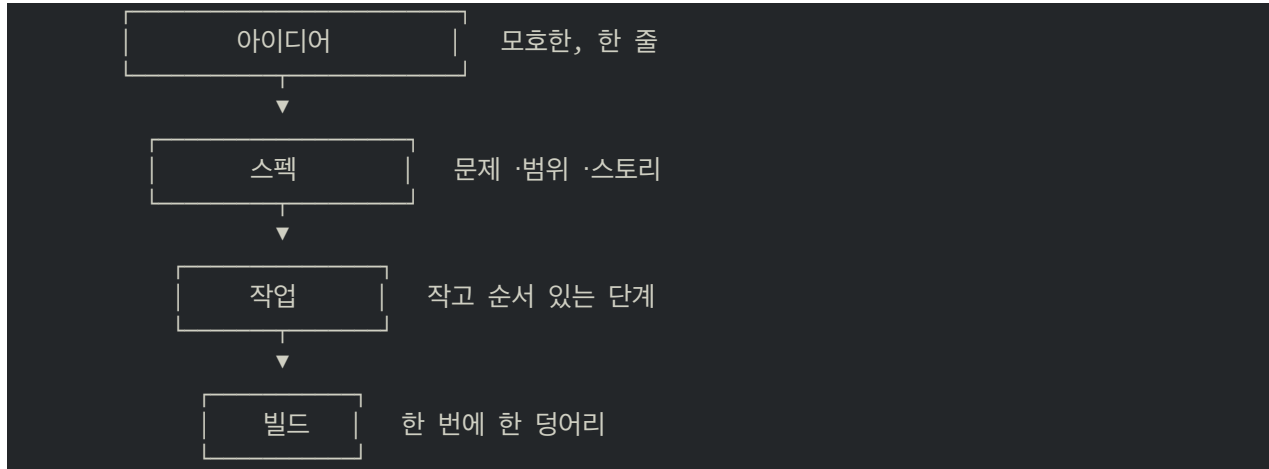
왜 그냥 느낌이 아니라 명세인가

바이브 코딩은 계획을 건너뛰다는 뜻이 아니다. 계획을 빠르고 가볍게 한 다음, AI 에게 실행을 지휘한다는 뜻이다. 명세는 행들이다. 그것 없이는 AI 가 모든 프롬프트에서 당신의 의도를 추측하고, 작은 오해들이 쌓여 엉망이 된다.

명세는 당신에게 세 가지를 준다:

- **공유된 맥락**—어떤 AI 세션에든 붙여 넣어 무엇을 만들고 있는지 즉시 다시 확립할 수 있다.
- **범위 경계**—“안”과“밖”사이의 글로 쓴 선, 기능 부풀림을 멈춘다.
- **작업 목록**—AI 가 한 번에 하나씩 완료할 수 있는, 만들 수 있는 덩어리들.

명세는 깔때기처럼 작동한다: 넓고 흐릿한 아이디어를 받아, 단계마다 좁혀서, AI 가 실제로 만들 수 있는 작은 덩어리로 만든다.



길 필요는 없다. 대부분의 프로젝트에는 한 페이지면 충분하다. 사실, 두 페이지를 넘겨 자라는 명세는 보통 경고 신호다: 너무 멀리까지 계획하고 있거나, 슬그머니 범위를 넓히고 있는 것이다. 의심스러우면 잘라내라. 지운 부분은 그리워질 일이 드물고, 남긴 부분은 더 날카로워진다.

1 단계: 문제를 명확히 하라

기능 이전에, 문제와 그것을 가진 사람을 명명하라. 해결책으로 건너뛰려는 충동을 참아라. 명확한 문제 진술은 이후의 모든 결정을 정직하게 유지한다.

스스로에게 물어라 (또는 AI 에게 당신을 인터뷰하게 하라):

- 이걸 누구를 위한 것인가? (구체적으로—“나”, “인디 메이커들”, “내 러닝 클럽”.)
- 이것 없이 그들은 오늘 어떤 고통스러운 일을 하고 있는가?
- 성공은 한 문장으로 어떤 모습인가?

AI 가 파헤치게 둘 수 있다. 이 프롬프트를 시도해보라:

나에게 막연한 아이디어가 있어: 내가 읽는 책을 기록하는 웹 앱.
제품 기획 파트너 역할을 해 줘. 날카로운 질문 5 개를 한 번에
하나씩 던져서 문제, 타겟 사용자, " 완료" 가 어떤 모습인지 명확히
해 줘. 아직 기능을 제안하지 말고 —아이디어가 구체적으로
잡힐 때까지 캐묻기만 해.

좋은 질문 다섯 개에 답하면 보통 모호한 아이디어가 진짜인 무언가로 날카로워진다. 요령은 AI 가 모든 것에 동의하는 대신 밀어붙이게 두는 것이다. AI 가“기기 간 동기화가 필요한가요?”라고 물었을 때 당신의 정직한 답이“아니, 그냥 내 노트북만”이라면, 당신은 방금 백엔드 하나와 인증 시스템 하나, 그리고 몇 주의 작업을—코드 한 줄 쓰기도 전에—지워 버린 것이다. 그것이 바로 핵심이다: 기능을 잘라내기 가장 싼 곳은 코드베이스가 아니라 대화다.

당신의 답이“음, 그런 것 같은데”에서 멈추고“그래, 정확히 그거야”로 바뀌는 순간을 살펴라. 그 전환이 진짜 문제를 찾았고 다음으로 넘어가도 된다는 신호다.

2 단계: 범위와 비목표를 정의하라

이것은 가장 가치 있으면서도 가장 많이 건너뛰는 단계다. **비목표 (Non-goals)** —당신이 의도적으로 만들지 않는 것들—는 “도움이 되는” 추가 기능을 사랑하는 AI로부터 당신을 보호한다.

우리의 책 트래커에서 범위는 이럴 수 있다:

- **범위 안:** 책 추가, 읽는 중/완료 표시, 목록 보기, 완독한 책 평점 매기기.
- **범위 밖 (비목표):** 소셜 기능, 추천, Goodreads 에서 가져오기, 모바일 앱, 사용자 계정.

두 개의 열을 그려보라: 왼쪽에는 만들기로 약속한 모든 것, 오른쪽에는 의도적으로 건너뛰는 모든 것. 그 사이의 선이 v1 을 작게 유지하는 경계다.

작업 범위	비목표
책 추가 읽는중/완료 표시 목록 보기 완독 책 평점	소셜 기능 추천 Goodreads 가져오기 모바일 앱 사용자 계정
지금 만들기 ←	→ 나중에

비목표를 적어두는 것은 두 가지 일을 한다. 첫 버전을 실제로 끝낼 만큼 작게 유지하고, AI 가 벗어날 때 붙여 넣을 한 문장을 준다: “기억해, 계정은 v1 의 비목표야—로그인을 추가하지 마.”

비목표는 “절대 안 함”이 아니다—“지금은 안 함”이다. 사용자 계정을 비목표로 적는다고 해서 당신의 앱이 영영 그것을 갖지 못한다는 뜻은 아니다. 이번 버전의 일부가 아니라는 뜻이고, 그래서 오늘 그 비용을 치르지 않는다는 뜻이다. 이 재구성은 중요하다: 좋은 아이디어를 버리는 기분 없이 기능에 “아니”라고 말할 수 있게 해준다. 당신은 그저 순서를 매기고 있을 뿐이다. 비목표 목록의 항목 대부분은 사실 변장한 v2 백로그이고, 그래도 괜찮다—어딘가에 적어두고 넘어가라.

3 단계: 사용자 스토리를 작성하라

사용자 스토리는 범위를 테스트 가능한 동작으로 바꾼다. 형식은 간단하다:

[사용자] 로서, 나는 **[행동]** 하고 싶다, 그래서 **[이점]** 을 얻기 위해.

책 트래커의 경우:

- 독자로서, 나는 제목과 저자로 책을 추가하고 싶다, 그래서 추적을 시작할 수 있도록.
- 독자로서, 나는 책을 완독으로 표시하고 싶다, 그래서 내 진행 상황을 볼 수 있도록.
- 독자로서, 나는 완독한 책에 1~5 점 별점을 매기고 싶다, 그래서 무엇이 좋았는지 기억하도록.

각 스토리는 독립적으로 만들고 검증할 수 있는 단위다. 앱을 클릭해 돌아다니며 스토리를 어떻게 테스트할지 그릴 수 없다면, 너무 모호한 것이다—쪼개거나 다시 써라.

그래서 (so that) 절은 사람들이 떨어뜨리는 부분이고, 가장 중요한 부분이다. “독자로서, 나는 검색창을 원한다”는 AI 에게 무엇을 만들지는 알려주지만 왜는 알려주지 않아서, AI 가 좋은 절충을 할 수 없다. “독자로서, 나는 제목 일부를 입력해 내 목록에서 책을 찾고 싶다, 그래서 수십 개의 항목을 스크롤하지 않아도 되도록”은 AI 에게 실제 할 일을 알려준다—그리고 이제 입력하는 대로 거르는 간단한 필터가 묵직한 검색 엔진을 이길 수도 있다. 이점은 명세를 위한 명세다: AI 가 (그리고 미래의 당신이) 좋은 구현과 잘못된 구현을 구분할 수 있을 만큼 그 기능을 충분히 설명해준다.

4 단계: 가벼운 PRD 초안을 작성하라

PRD(Product Requirements Document, 제품 요구사항 문서) 는 그저 당신의 결정들을 한곳에 모은 것이다. 거기에는 예측 가능한 뼈대가 있다—매번 똑같은 몇 개의 섹션:

PRD
└ 문제 왜 존재하는가

```

├─ 사용자      누구를 위한 것인가
├─ 목표        무엇을 하는가, 한 줄로
├─ 작업 범위   v1 에 포함되는 것
├─ 비목표      v1 에서 빼는 것
├─ 사용자 스토리 ~ 로서, ~ 하고 싶다, ~ 하기 위해
├─ 데이터 모델 저장하는 필드
└─ 완료 기준   인수 테스트

```

아주 작게 유지하라. 다음은 AI 에게 건넬 수 있는 완전한 시작용 PRD 다:

```

# Book Tracker — Mini PRD

## Problem
I read a lot but lose track of what I've finished and what I thought of it.

## User
Me (a single user). No accounts needed for v1.

## Goal
A simple web app to add books, track reading status, and rate finished ones.

## In Scope (v1)
- Add a book (title, author)
- Set status: To Read / Reading / Finished
- List all books, filterable by status
- Rate finished books 1-5 stars

## Non-Goals (v1)
- User accounts / login
- Social features, sharing, recommendations
- Importing from other services
- Mobile / native apps

## Tech (keep it simple)
- Single-page web app
- Local browser storage (no backend for v1)

## Done When
I can add a book, change its status, rate it, and see my list
persist after a page refresh.

```

“Done When”줄에 주목하라. 그것이 당신의 인수 테스트다—v1 을 출시했는지 알 수 있는 구체적인 방법. 항상 하나 포함하라. 좋은 “Done When”은 당신이 물리적으로 수행할 수 있는 것이다: 여기를 클릭하고, 이것을 입력하고, 새로고침하고, 저것을 본다. 만약 완료 조건이 느낌처럼 읽힌다면 (“다듬어졌을 때”, “느낌이 좋을 때”), 그것은 영영 완료되지 않는다. 대조할 것이 없기 때문이다. 낯선 사람에게 건넬 수 있는 대본으로 만들어라.

PRD 를 손으로 직접 쓸 필요는 없다. 유용한 수는 앞 단계들의 답을 AI 에게 다시 먹이고 초안을 조립하게 한 다음, 당신이 편집하는 것이다. 통째로 받아들이는 대신 편집하는 이유는, 당신이 모든 줄을 이해하고 동의해야 하기 때문이다—이 문서는 당신이 계속 붙여 넣게 될 진실의 원천이라, 여기에 든 오류는 이후의 모든 세션으로 복사된다.

5 단계: 데이터 모델을 스케치하라

대부분의 앱은 실은 데이터에 관한 것이다: 무엇을 저장하고 조각들이 어떻게 관련되는가. 빠른 데이터 모델 스케치는 나중에 혼란스럽고 일관성 없는 코드로부터 당신을 구해준다. 데이터베이스 다이어그램은 필요 없다—필드 목록이면 충분하다.

책 트래커의 경우, 주된 것은 하나다—Book:

```
Book
- id: 고유 문자열
- title: 문자열
- author: 문자열
- status: "to_read" | "reading" | "finished"
- rating: 숫자 1-5 (완독한 경우만)
- addedAt: 날짜
```

이 작은 스케치조차 결정을 명시적으로 만든다: 평점은 완독한 책에만 존재하고, 상태는 정해진 세 값 중 하나이며, 모든 책에는 id 가 있다. 이것을 AI 에게 건네면, AI 는 필드 이름 추측을 멈추고 일관되게 만든다.

이것을 일찍 못박아 두어야 하는 이유는 필드 이름이 사방으로 새어 나가기 때문이다. 일단 AI 가 book.rating 을 읽는 코드를 쓰면, 그 이름은 폼에도, 목록 뷰에도, 저장 계층에도, 그리고 나중에 추가하는 어떤 기능에도 나타난다. 그것이 떠다니게 두면—한 곳에서는 rating, 다른 곳에서는 stars, 또 다른 곳에서는 score —쫓기 성가신 버그가 생긴다. 여섯 줄짜리 데이터 모델은 그것에 대한 가장 싼 보험이다.

짧고 복사해 붙일 수 있는 명세 블록은 AI 에게 산문보다 더 잘 통하는 경향이 있다. 다음은 실제로 보낼 프롬프트로 쓴 같은 모델이다:

책에 정확히 이 데이터 형태를 사용해. 내가 적지 않은 필드는 추가하지 말고, 이 이름들을 바꾸지도 마:

```
Book {
  id      —고유 문자열, 생성 시 자동 발급
  title   —문자열, 필수
  author  —문자열, 필수
  status  —다음 중 하나: "to_read" | "reading" | "finished"
  rating  —정수 1-5, status 가 "finished" 일 때만 존재
  addedAt — ISO 날짜 문자열, 책 추가 시 설정
}
```

필드가 빠졌다고 생각되면, 추가하기 전에 나에게 물어봐.

마지막 줄—추가하기 전에 나에게 물어봐—가 진짜 일을 하고 있다. 그것은 AI 를 추측꾼에서 협력자로 바꾸고, 데이터 모델을 당신의 것으로 유지한다.

6 단계: 명세를 바이브 크기 작업으로 쪼개라

마지막 수는 명세를 AI 가 **한 번에 하나씩** 실행할 수 있는 작업으로 잘게 나누는 것이다. 바이브 크기 작업은 결과를 1 분 안에 검토하고 작동했는지 알 수 있을 만큼 작다. 한 작업이 AI 에게 여러 파일이 걸리고 그게 맞는지 알 수 없다면, 너무 큰 것이다.

책 트래커를 위한 좋은 작업들:

1. 헤더와 빈 책 목록이 있는 기본 페이지 레이아웃을 구성한다.
2. 목록에 메모리상으로 추가하는 “책 추가” 폼 (제목 + 저자) 을 만든다.
3. 상태 (To Read / Reading / Finished) 와 그것을 바꿀 드롭다운을 추가한다.
4. 새로고침에도 살아남도록 목록을 브라우저 로컬 스토리지에 저장한다.
5. 완독한 책에만 나타나는 평점 컨트롤을 추가한다.
6. 목록 위에 상태 필터 버튼을 추가한다.

각 작업은 사용자 스토리나 PRD 줄에 매핑되고, 직전 것 위에 쌓이며, 작동하는 것을 볼 수 있는 무언가로 끝난다. 그것이 이 바이브 코딩의 리듬이다: 작은 작업, 실행, 검증, 다음.

이 분해를 AI 가 생성하게 할 수도 있다:

여기 내 미니 PRD 와 데이터 모델이 있어 [여기에 붙여넣기].
이걸 작고 순서 있는 빌드 작업 6~8 개로 쪼개 줘. 각 작업은
독립적으로 테스트 가능하고 이전 작업 위에 쌓여야 해. 각 작업은
앱을 클릭해 보면서 내가 확인할 수 있는 수준으로 유지해 줘.

명세를 살아 있게 유지하라

명세는 한 번 짓고 감상하는 기념비가 아니다. 그것은 살아 있는 문서이고, 가장 흔한 실수는 그것을 고정된 것으로 취급하는 것이다. 만들어 가면서, 계획 단계가 알려줄 수 없었던 것들을 배우게 된다: 어떤 기능이 예상보다 어렵고, 어떤 비목표가 알고 보니 필수적이며, 어떤 사용자 스토리가 실은 두 개의 스토리였다는 식으로. 그런 일이 생기면, 먼저 명세를 고친 다음, AI 에게 새 버전을 따르라고 말하라. 명세가 이끌고, 코드가 따른다.

몇 가지 습관이 그것을 정직하게 유지한다:

- **결정이 바뀌는 순간 갱신하라.** 코드는 다른 것을 하는데 명세는 한 가지를 말하고 있다면, 그것은 명세가 없는 것보다 나쁘다—다음 AI 세션을 능동적으로 오도하기 때문이다.
- **끝낸 작업은 지워라.** 작업 목록은 진행 추적기 역할도 겸한다. 여덟 개 중 여섯 개가 체크된 것을 보는 것은 그 자체로 일종의 추진력이다.
- **놀라운 것은 그냥 해버리는 대신 비목표나 “나중에” 목록으로 옮겨라.** 솔깃한 기능이 작업 도중 나타나면, 적어두고 계속 가라. 그것이 v1 에 자리를 얻을 자격이 있는지는 나중에 결정할 수 있다.

이것이 “나는 무언가를 만들었다”와 “나는 내가 의도한 것을 만들었다”를 가르는 규율이다. 명세는 당신이 프로젝트의 승객이 아니라 저자로 남는 방법이다.

흔한 함정: 문제가 아니라 해결책을 명세화하기

한 가지 실패 양상은 따로 짚어둘 만한데, 너무 쉽게 빠지기 때문이다. 명세를 쓰려고 앉으면, 이미 머릿속에 그려둔 해결책을 묘사하고 싶은 유혹이 든다—“탭이 있는 사이드바”, “튀어나오는 모달”, “Postgres 테이블”. 하지만 해결책으로 가득 찬 명세는 아직 얻어내지 못한 결정들을 슬그머니 못박아 버린다.

이 두 줄을 비교해보라:

나쁨: 책 상태별로 접을 수 있는 섹션이 있는 왼쪽 사이드바를 추가해.
좋음: 전체 목록을 잃지 않으면서 책을 상태별로 묶어서 보게 해 줘.

첫 문장은 이미 사이드바를, 접히는 섹션을, 그리고 레이아웃을 골랐다—그것들이 옳은 선택인지 아무도 확인하기 전에. 둘째 문장은 해야 할 일을 진술하고, AI(그리고 당신)가 그것을 만족시키는 가장 단순한 것을 찾도록 자유롭게 둔다, 그것이 그냥 필터 버튼 세 개일 수도 있다. 원하는 결과를 묘사하라; 구현은 가능한 한 오래 협상 가능한 채로 두어라. 해결책은 명세에서는 바꾸기 싸고 코드에서는 바꾸기 비싸다.

보상

이제 당신은 문제 진술, 범위와 비목표, 사용자 스토리, 한 페이지짜리 PRD, 데이터 모델, 그리고 순서가 매겨진 작업 목록을 가졌다—실제 프로젝트의 토대 전체를, AI 를 의견 상대 삼아 몇 분 만에 초안 잡은 것이다.

이 가운데 어느 것도 코드를 짤 줄 알아야 하지는 않았다. 당신이 무엇을 원하는지 알고 몇 분 동안 명료하게 생각하기만 하면 됐다—나머지 생각은 AI 가 당신과 함께 소리 내어 해주었다. 그것이 진짜 전환이다: 당신은 아직 소프트웨어를 쓰고 있는 게 아니라 의도를 쓰고 있고, 좋은 의도가 싸움의 대부분이다.

이 명세를 프로젝트 안의 파일에 보관하라 (많은 사람이 SPEC.md 나 PRD.md 를 쓴다). 그것은 새 AI 세션에 붙여 넣어 맥락을 다시 고정하는 문서가 되고, 바이브 크기 작업을 하나씩 처리해 나가는 체크리스트가 된다. 다음 장에서는 첫 번째 작업을 가져와 실제로 만들어볼 것이다.

정리와 실습

핵심 요점

- 명세는 짧고 평이한 언어로 된 작업 노트다—무엇을 만들고 무엇을 안 만드는지. 동결하는 계약이 아니며, 한 페이지면 충분하다.
- 기능 이전에 문제와 그 문제를 가진 사람을 명확히 하고, 논골 (non-goals) 을 적어라. AI 가 “도움이 되는” 추가 기능을 넣는 것에 맞서는 가장 날카로운 방어다.
- so that 절이 있는 유저 스토리와 구체적인 “Done When” 줄이 있는 작은 PRD 는 의도를 테스트 가능하고 구축 가능한 동작으로 바꾼다.
- 데이터 모델을 일찍 못 박아라—필드 이름은 어디에나 새어 나간다—그리고 명세를 한 번에 하나씩 검증할 수 있는 바이브 크기 작업으로 쪼개라.
- 해법이 아니라 문제를 기술하라. 결과는 명세에서는 바꾸기 싸고 코드에서는 비싸다. 배워가면서 명세를 살아있게 유지하라.

해보기

묵혀둔 아이디어 하나를 골라 단일 AI 세션 안에서 한 페이지 명세로 바꿔라. 먼저 AI 가 다섯 개의 날카로운 질문으로 당신을 인터뷰하게 한 다음—당신 자신의 언어로—문제, 사용자, 세 개의 유저 스토리 (각각 so that 포함), 명시적인 논골 목록, 여섯 줄짜리 데이터 모델, 그리고 물리적으로 수행할 수 있는 “Done When” 줄 하나를 써라. SPEC.md 로 저장하라. 좋은 명세의 시험: 내일 완전히 새로운 대화에 붙여 넣어도 AI 가 정확히 무엇을 만들어야 할지 알 수 있어야 한다.

이 장의 프롬프트

나에게 막연한 아이디어가 있어: < 한 줄 설명 >.

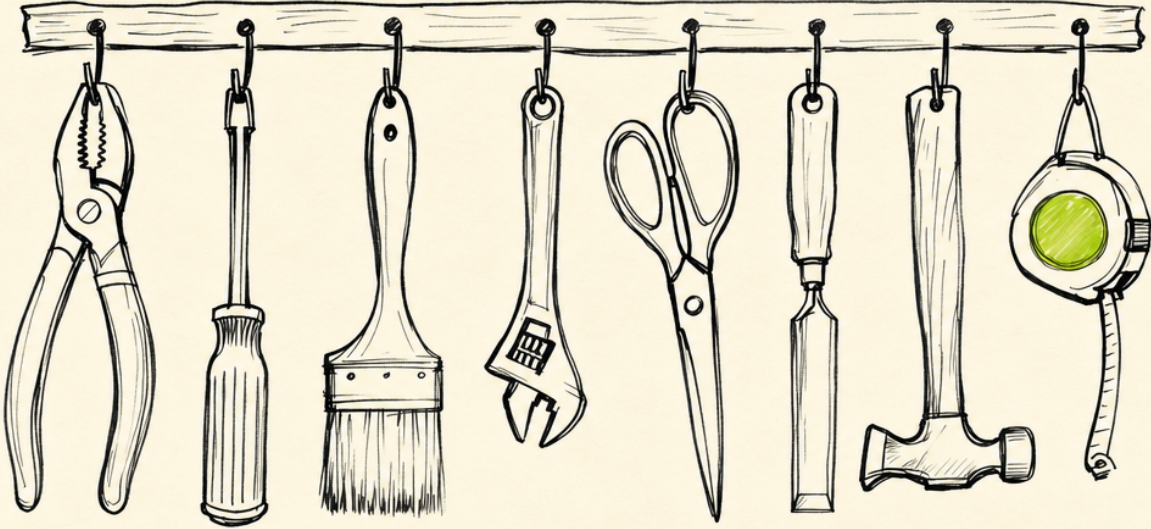
내 제품 기획 파트너 역할을 해 줘. 먼저, 날카로운 질문 5 개를 한 번에 하나씩 던져서 문제, 타겟 사용자, " 완료 " 가 어떤 모습인지 명확히 해 줘. 아직 기능을 제안하지 마.

내가 답하면, 다음 섹션으로 한 페이지짜리 스펙 초안을 작성해 줘:

- 문제 - 사용자 - 목표
- 작업 범위 (v1) - 비목표 (v1)
- 사용자 스토리 (각각 "~ 로서, ~ 하고 싶다, ~ 하기 위해")
- 데이터 모델 (필드의 짧은 목록)
- 완료 기준 (구체적이고 직접 수행 가능한 인수 테스트)

간결하게 —한 페이지 이내로. 모호한 게 있으면, 가정하기 전에 물어봐. 모든 줄에 동의할 때까지 초안을 고칠게.

스택 고르기: 한눈에 보는 언어들

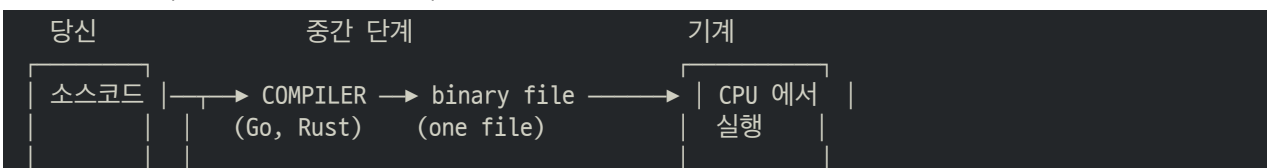


AI 에게 소프트웨어를 만들라고 지시할 때도, 어떤 언어를 고르느냐는 여전히 중요하다—그것도 아주 많이. 코드는 AI 가 쓰지만 그 결과를 감당하는 건 당신이다. 얼마나 빨리 출시하는지, 배포가 얼마나 쉬운지, 얼마나 자주 벽에 부딪히는지 가 거기 달려 있다. 다행인 점은 이 언어들 중 어느 것도 통달할 필요는 없다는 것이다. 합리적인 선택을 내리고, AI 가 당신을 고통스러운 곳으로 끌고 갈 때 그것을 알아차릴 정도만 알면 된다.

이 장에서는 신경 쓸 만한 언어들을 하나씩 살펴봄, 각 언어가 진짜로 잘하는 것, 발목을 잡는 지점, 그리고—바이브 코딩에서 결정적으로 중요한—AI 어시스턴트가 그 언어를 얼마나 잘 다루는지를 짚는다. 마지막에는 목표에 따라 고르는 아주 간단한 가이드로 마무리한다.

시작하기 전에 한 가지: AI 어시스턴트는 공개된 코드가 산더미처럼 쌓인 인기 언어를 틈새 언어보다 훨씬 잘 다룬다. 이건 사소한 차이가 아니다. 프로젝트가 매끄럽게 굴러가느냐 마느냐를 가르는 가장 큰 단일 요인인 경우가 많다. 인기는 그 자체로 하나의 기능이다. 어떤 언어와 그 주요 프레임워크가 공개 저장소에 수백만 번 등장하면, AI 는 흔한 패턴, 전형적인 에러, 관용적인 해결책을 전부 본 셈이다. 그렇지 않으면, 그 희소함의 대가를 실패한 시도와 그럴듯하게 들리는 헛소리로 치르게 된다.

언어 이름을 보기 전에 머릿속에 담아둘 그림 하나: 당신이 쓰는 코드는 기계가 실제로 돌리는 것이 결코 아니다. 그 사이의 무언가가 당신의 텍스트를 돌아가는 소프트웨어로 바꾼다—컴파일러 (미리 전부 번역해 실행 파일로 만든다) 이거나 인터프리터 (한 줄씩 읽으며 실행한다) 다. 당신이 고르는 언어가 어느 경로를 탈지 정한다.





JavaScript / TypeScript

대부분의 바이브 코딩에서 이것이 기본 선택이며, 거기엔 그럴 만한 이유가 있다. 어디서나 돌아간다—브라우저, 서버 (Node.js), 엣지 함수—그리고 웹 프론트엔드 전체의 언어다. TypeScript 는 JavaScript 에 타입 시스템을 덧붙인 것인데, 거의 언제나 TypeScript 를 택하는 게 좋다. 타입은 실수를 일찍 잡아주고, 무엇보다 AI 에게 안전 가드레일을 제공해 결과물을 더 믿을 만하게 만든다. 타입이 붙은 함수 시그니처는 어시스턴트가 읽을 수 있는 계약서다. 그것이 없으면 AI 는 당신의 데이터가 어떤 형태인지 추측하게 되고, 추측이야말로 버그가 숨는 곳이다.

- **잘하는 것:** 웹 앱 (프론트엔드와 백엔드), 브라우저에서 사용자가 클릭하는 모든 것, 빠른 프로토타입, 한 언어로 만드는 풀스택 프로젝트, 서버리스/엣지 배포.
- **아픈 지점:** 생태계가 빠르게 움직이며 자주 깨진다. 의존성이 불어나는 문제가 실재한다 (작은 앱 하나가 수백 개의 패키지를 끌어올 수 있다). 순수 JavaScript 의 느슨함은 미묘한 버그가 슬쩍 빠져나가게 한다—TypeScript 를 써야 할 또 하나의 이유다.
- **대표적인 사용처:** SaaS(서비스형 소프트웨어—설치하지 않고 온라인으로 빌려 쓰는 소프트웨어) 대시보드, 랜딩 페이지, 전자상거래, 실시간 앱, 브라우저 확장 프로그램, Vercel·Netlify·Cloudflare 에 배포하는 모든 것.
- **AI 처리 능력:** 탁월함. 학습 데이터에 가장 많이 등장하는 언어다. 어시스턴트는 React, Next.js, Node 를 자신 있게 작성하며 막히는 일이 드물다. 굳이 다른 걸 골라야 할 강력한 이유가 없다면 여기서 시작하라.

알아둘 만한 구체적인 트레이드오프 하나: JavaScript 를 프로토타이핑에 훌륭하게 만드는 그 속도가, 동시에 끊임없이 변하게 만드는 속도다. 18 개월 전 튜토리얼이 이미 폐기된 API 를 쓰고 있을 수 있고, 과거의 스냅샷으로 학습된 AI 는 때때로 어제의 패턴을 건네준다. 그것이 작성한 코드가 작동하지 않는다면, 원인은 라이브러리가 앞으로 나아가버렸기 때문인 경우가 많다. 해결책은 보통 당신이 쓰는 버전을 알려주고 최신 문서를 확인하라고 시키는 것이다. 이것은 지구 상에서 가장 활발한 생태계에 사는 대가로 내는 세금이며, 대부분의 프로젝트에서는 널 만한 값이다.

Python

Python 은 거의 영어처럼 읽혀서, 초보자에게 가장 친근한 언어이자 데이터·스크립팅, 그리고 AI 자체의 공용어다. 프로젝트가 머신러닝, 데이터 분석, 자동화, 과학 계산에 맞는다면 보통 답은 Python 이다.

- **잘하는 것:** AI/ML, 데이터 처리, 자동화 스크립트, 백엔드 API(애플리케이션 프로그래밍 인터페이스—한 프로그램이 다른 프로그램과 대화할 때 쓰는 출입구), 접착제 코드, 이미 라이브러리가 존재하는 모든 것 (그리고 Python 에선 대개 존재한다).
- **아픈 지점:** 내부적으로 C 로 작성된 라이브러리에 기대지 않으면 순수 수치 연산이 느리다. 가상 환경과 패키지 버전이 얽힌“의존성 지옥”이 입문자를 혼란스럽게 한다. 브라우저 프론트엔드에는 맞지 않는다.
- **대표적인 사용처:** 스크래핑, 데이터 파이프라인, ML 실험, Discord/Telegram 봇, 내부 도구, REST(표현 상태 전송—그런 API 출입구를 만드는 흔한 방식) API(FastAPI, Django, Flask).
- **AI 처리 능력:** 탁월하며 JavaScript 와 어깨를 나란히 한다. 어시스턴트는 Python 과 그 주요 라이브러리에 대단히 능숙하다. 비 (非) 웹 프로젝트의 훌륭한 기본 선택이다.

Python 이 입문자를 어김없이 걸려 넘어지게 하는 단 하나의 지점은 언어가 아니라 환경이다. AI 에게는 한 폴더에서 잘 돌던 스크립트가 다른 폴더에서는 실패하는데, 패키지가 활성화되지 않은 가상 환경 안에 들어 있거나, 두 프로젝트가 같은 라이브러리의 서로 다른 버전을 원하기 때문이다. 이 중 어느 것도 당신 코드의 잘못이 아니고, 한 번 겪고 나면 어렵지도 않다—하지만 Python 에서“왜 그냥 안 돌아가지”하는 순간으로 가장 흔하다. 처음부터 AI 에게 가상 환경과 requirements.txt 를 설정하라고 (또는 uv 나 Poetry 같은 도구를 쓰라고) 시키면, 고통의 대부분이 사라진다.

Go

Go 는 단순하고 빠르며 배포가 쉽도록 Google 에서 설계되었다. 컴파일하면 런타임을 따로 설치할 필요 없이 서버에 그냥 던져 넣을 수 있는 단일 자립형 바이너리가 나온다. 화려한 기능을 일부러 빼버려서 코드베이스가 좋은 의미로 지

루하게 유지된다—그리고 지루함은 사람과 AI 둘 다 추론하기 쉽다.

- **잘하는 것:** 빠른 백엔드 서비스, 명령줄 도구, 네트워킹, 수많은 동시 연결을 처리해야 하는 모든 것, 단순한 배포.
- **아픈 지점:** 장황한 에러 처리 (if err != nil 을 사방에서 보게 된다). 프론트엔드 언어가 아니다. 의도된 단순함은 표현력 있는 지름길을 원할 때 제약처럼 느껴질 수 있다.
- **대표적인 사용처:** API, 마이크로서비스, CLI(명령줄 인터페이스—명령어를 입력해 프로그램을 실행하는 텍스트 전용 방식), 인프라 도구, 트래픽이 많은 백엔드.
- **AI 처리 능력:** 매우 좋음. Go 의 작고 일관된 기능 집합은 대부분의 일에 “올바른 방식”이 하나 있음을 뜻하고, 어시스턴트는 그것을 잘 따른다. JS/Python 보다 학습 데이터는 적지만 단순함이 이를 보완한다.

1 인 개발자에게 Go 의 결정적 강점은 그 단일 바이너리다. “서버에 Node 와 패키지 마흔 개를 설치하기”같은 춤도, 당신 컴퓨터와 프로덕션 사이의 런타임 버전 불일치도 없다. 빌드하고, 파일 하나를 복사하고, 실행한다. 그것만으로도 배포가 극적으로 덜 오류가 나게 된다—바이브 코딩 중이고 인프라를 디버깅하고 싶지 않을 때 이건 큰 의미가 있다. 장황함이 그 대가다. Python 에서 세 줄일 코드가 Go 에서는 여덟 줄이 된다. 하지만 그 여분의 줄들은 예측 가능하고, 예측 가능해야말로 AI 가 잘 다루는 바로 그것이다.

Rust

Rust 는 메모리 버그 부류 전체를 아예 작성하지 못하게 막는 컴파일러와 함께 C 수준의 속도를 준다. 그 대가는 악명 높게 엄격한 컴파일러와 가파른 학습 곡선이다. 바이브 코딩에서 Rust 는 양날의 검이다. AI 가 코드를 쓰긴 하지만, Rust 의 엄격한 때문에 컴파일러가 코드를 더 자주 거부하고, 그 결과 수정과 재시도를 오가는 사이클이 늘어난다.

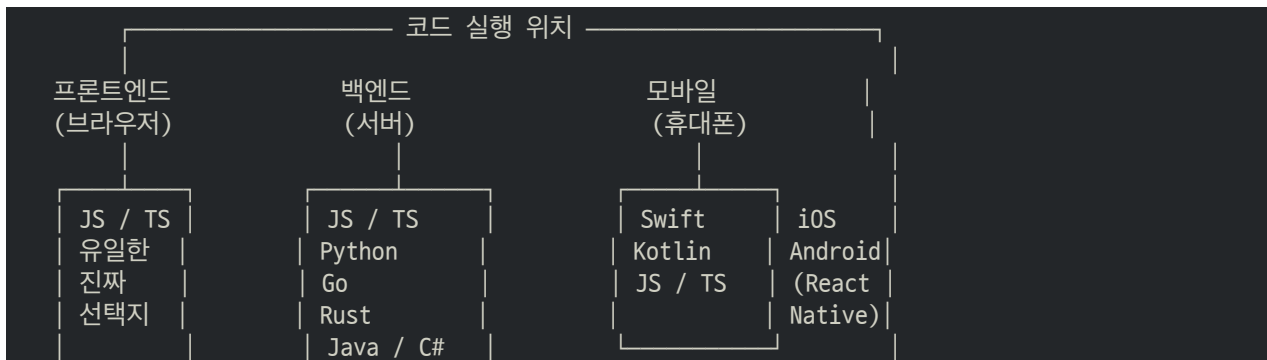
- **잘하는 것:** 성능이 결정적인 소프트웨어, 시스템 프로그래밍, 게임 엔진, WebAssembly, 속도가 중요한 명령줄 도구, 정확성과 안전성이 최우선인 모든 것.
- **아픈 지점:** 가파른 학습 곡선. 느린 컴파일 시간. “borrow checker”가 초보자를 좌절시킨다. 대부분의 웹 앱과 스크립트에는 과하다.
- **대표적인 사용처:** 고성능 백엔드, CLI 도구, WASM(WebAssembly) 모듈, 임베디드에 가까운 시스템, 1 밀리초가 아쉬운 인프라.
- **AI 처리 능력:** 좋지만 울퉁불퉁함. 어시스턴트는 Rust 를 잘 알지만, 엄격한 컴파일러가 반복을 요구하는 에러를 드러낸다. Python 이나 JS 보다 “에러 고치고 다시 시도”루프가 더 많을 것으로 예상하라. 성능이나 안전성이 그 마찰을 정말로 정당화할 때 Rust 를 택하라—기본값으로는 아니다.

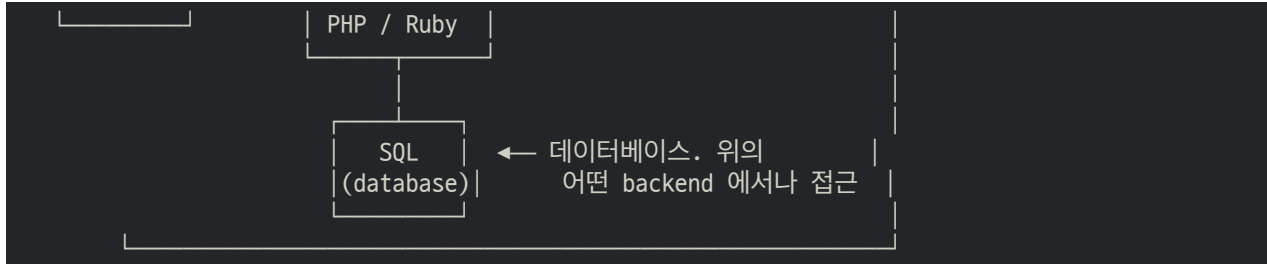
놓치기 쉽지만 Rust 의 엄격함에는 미묘한 장점이 있다. 컴파일러가 공짜로, 가차 없이 코드를 리뷰해 준다. AI 가 엉성한 JavaScript 를 쓰면 그 버그는 그대로 출시되어 사용자를 문다. AI 가 엉성한 Rust 를 쓰면 컴파일러가 보통 프로그램이 실행되기도 전에 그것을 잡아낸다. 늘어나는 수정 루프는 마찰처럼 느껴지지만, 그중 상당수는 컴파일러가 진짜 버그를 막아 주는 것이다. Rust 에 손을 뻗기 전에 던질 정직한 질문은, 당신이 실제로 그 속도나 안전성이 필요한지, 아니면 그걸 원해야 한다고 들었을 뿐인지다. 첫 프로젝트에서는 거의 언제나 필요 없다.

나머지 언어들에 대한 짧은 한마디

이것들을 마주치게 될 텐데, 때로는 그중 하나가 정확히 맞는 답이다:

각 언어가 주로 어디에 사는지 그려 보면 도움이 된다. 소프트웨어에는 층이 있다—브라우저에서 도는 것 (프론트엔드), 서버에서 도는 것 (백엔드), 폰에서 도는 것 (모바일)—그리고 대부분의 언어에는 자기 텃밭이 있다:





- **Java / C#:** 대기업의 일꾼들. 성숙하고 안정적이며 도구가 풍부하게 갖춰져 있어, 큰 기업 백엔드나 Android(Java/Kotlin) 또는 Windows/.NET 생태계 (C#) 에 이상적이다. AI 는 둘 다 잘 다룬다. 1 인 바이브 코더로서는 기존 엔터프라이즈 스택 안에서 일하거나 Android/Windows 를 겨냥해 만드는 게 아니라면 이것들이 필요한 경우가 드물다.
- **Swift / Kotlin:** iOS(Swift) 와 Android(Kotlin) 앱을 위한 네이티브 언어. 진짜 모바일 앱을 App Store 나 Play Store 에 출시한다면 이것들을 건드리게 될 가능성이 높다. AI 지원은 탄탄하지만 웹 언어보다는 약하고, 모바일 개발에는 추가적인 마찰 (기기, 앱 스토어 규칙, 빌드 도구) 이 따른다.
- **PHP / Ruby:** 웹의 구세대이지만, 여전히 웹의 거대한 한 조각을 조용히 돌리고 있다. PHP 는 WordPress 를 돌리고, 따라서 전체 웹사이트의 큰 비율을 떠받친다. Ruby on Rails 는 데이터베이스 기반 앱을 빠르고 일관된 의견으로 만드는 방식으로 남아 있다. 둘 다 학습 데이터가 풍부해서 AI 가 잘 다룬다—하지만 오늘날 입문자가 가장 먼저 손을 뻗는 것은 드물고, 보통은 새로 시작하기보다 기존 코드베이스 안에서 일하기 때문에 고르게 된다.
- **SQL(구조적 질의 언어):** 범용 언어는 아니지만, 앱이 데이터를 저장하는 순간 마주하게 된다. 데이터베이스에 질문을 던지는 언어라고 생각하면 된다—“이번 주에 가입한 사용자를 전부 보여줘”같은. SQL 은 데이터베이스에 질의한다. 다른 언어들 대신 SQL 을 고르는 게 아니라—무엇을 고르든 그 옆에 함께 산다. 다행인 점: AI 는 SQL 작성에 탁월해서, 원하는 데이터를 평범한 말로 묘사하면 AI 가 쿼리를 만들어 준다.

한눈에 보는 비교

언어	가장 잘 맞는 곳	주의할 점	AI 능숙도
JavaScript/TypeScript	웹 앱, 풀스택, 엣지	의존성 폭증, 빠른 변동	탁월
Python	AI/ML, 데이터, 스크립트, API	순수 연산이 느림, 환경 설정	탁월
Go	빠른 백엔드, CLI, 서비스	상황함, 프론트엔드에 부적합	매우 좋음
Rust	성능, 안전성, WASM	가파른 곡선, 수정 루프 증가	좋음 (울퉁불퉁)
Java/C#	엔터프라이즈, Android, .NET	무거움, 1 인 작업엔 흔히 과함	매우 좋음
Swift/Kotlin	네이티브 iOS / Android 앱	모바일 마찰, 약한 AI	좋음
PHP/Ruby	WordPress, Rails, 기존 웹앱	낮은 기본값, 레거시 짐	매우 좋음
SQL	데이터베이스와 대화하기	대신이 아니라 옆에 함께 산다	탁월

이국적인 선택의 진짜 비용

언어를 우아하다는 이유로, 빠르다는 이유로, 혹은 당신이 존경하는 누군가가 극찬한다는 이유로 고르고 싶은 유혹이 있다. 특히 바이브 코딩에서는 그 본능이 프로젝트를 조용히 가라앉힐 수 있다. 이유는 이렇다. 틈새 언어의 도구와 씨름하는 데, 또는 모델이 본 예제가 너무 적어서 미묘하게 틀린 AI 출력을 풀어내는 데 쓰는 모든 시간은, 출시에 쓰지 못한 시간이다. 이국적인 선택은 좀처럼 요란하게 실패하지 않는다. 늘어난 수정 루프, 없는 라이브러리, 존재하지 않는 Stack Overflow 답변이라는 느린 물방울로 실패한다.

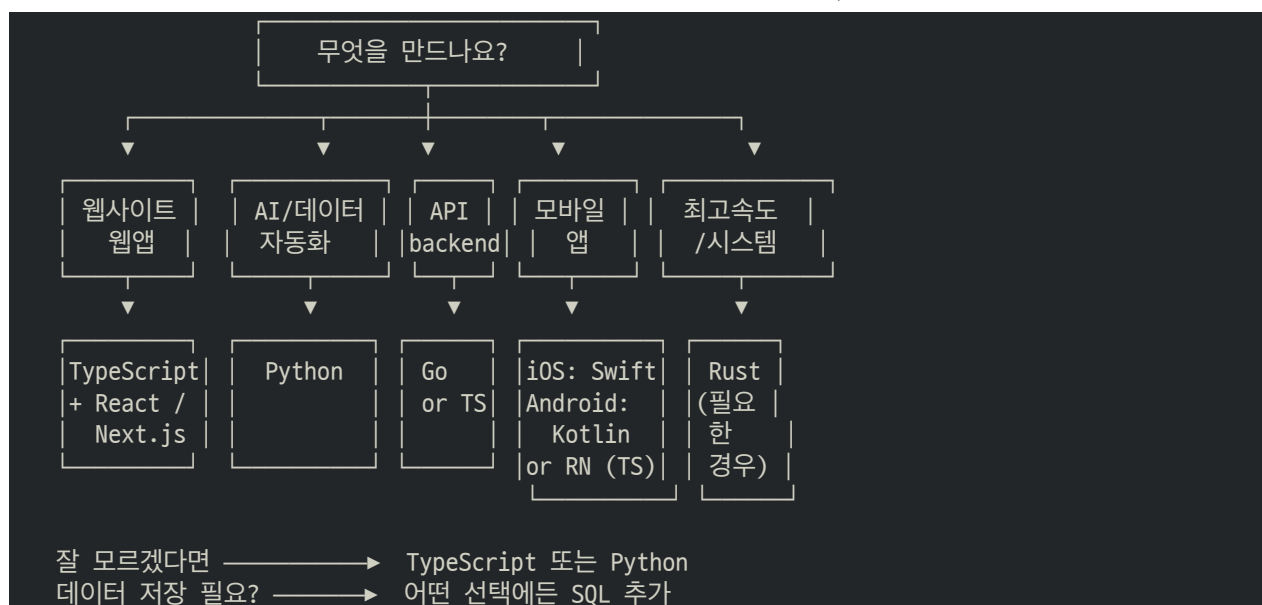
구체적으로, 그 페널티는 세 곳에서 나타난다. 첫째, AI 가 덜 능숙해서 실수를 더 많이 하고 고치는 것도 더 못한다—그리고 그 언어를 모르는 당신은 좋은 코드와 나쁜 코드를 구별할 수 없다. 둘째, 생태계가 더 얇아서 필요한 라이브러리가 없거나, 버려졌거나, 복사할 예제가 없을 수 있다. 셋째, 막혔을 때 웨이 더 조용하다. 튜토리얼도, 답변된 질문도, 당신과 똑같은 벽에 부딪혀 본 사람도 더 적다. Python 이나 TypeScript 에서는 이 셋 모두 문제가 아니다.

이 중 어느 것도 이국적인 언어가 나쁘다는 뜻은 아니다. 입증 책임이 이국적인 선택 쪽에 있다는 뜻이다. 구체적이고 명확한 이유가 있다면—Rust 의 속도가 진짜로 필요하거나, App Store 때문에 Swift 로 떠밀렸거나, 기존 Go 서비스를 확장하고 있다면—눈을 뜬 채 그 비용을 치러라. 이유가 “멋져 보여서”라면, 인기 있는 쪽을 골라 무언가를 출시하라. 나중에 언제든지 다시 쓸 수 있지만 거의 그러지 않게 되는데, 인기 있는 선택이 이미 일을 해냈을 것이기 때문이다.

목표로 고르기

고민으로 시간을 낭비하지 마라. 당신의 목표를 한 행에 맞추고 넘어가라—나중에 언제든지 방향을 바꿀 수 있고, 첫 프로젝트에서는 최적화보다 추진력이 이긴다.

전체 결정을 하나의 트리로 그리면 이렇다. 맨 위에서 시작해 각 질문에 답하고, 가지를 따라가 당신의 스택에 도달하라:



- **웹사이트나 웹 앱을 만들고 싶다.** □ TypeScript(React/Next.js 와 함께). 명백하고 지원이 탄탄한 선택.
- **AI, 데이터, 또는 자동화를 하고 싶다.** □ Python. 도구들이 사는 곳이다.
- **빠른 API 나 백엔드 서비스가 필요하다.** □ Go(단순하고 빠름) 또는 TypeScript(웹 스택의 일부라면). Python 의 FastAPI 도 충분히 잘 작동한다.
- **모바일 앱을 만들고 있다.** □ iOS 는 Swift, Android 는 Kotlin —또는 TypeScript 에 머무르기 위해 크로스플랫폼 웹 기반 접근 (React Native) 을 고려하라.
- **순수한 속도나 시스템 수준의 제어가 필요하다.** □ Rust, 단 실제로 필요하다고 확인한 경우에만. 대부분의 사람에게겐 필요 없다.
- **이미 존재하는 사이트에 무언가를 추가하고 있다.** □ 그것이 이미 작성된 언어로. WordPress 플러그인이라면 PHP, 오래된 Rails 앱이라면 Ruby 다. 코드베이스에 맞추고, 거기에 맞서 싸우지 마라.
- **그냥 배우면서 무언가를 빨리 출시하고 싶다.** □ Python 이나 TypeScript. 둘 다 너그럽고, 둘 다 AI 가 가장 잘 아는 언어이며, 둘 다 끝없는 튜토리얼과 예제가 있다.
- **내 앱이 데이터를 저장해야 한다.** □ 위에서 무엇을 고르든, 거기에 데이터베이스용 SQL 을 더하라. 둘 사이에서 고르는 게 아니다.

메타 교훈: 헛갈리면 인기 있는 쪽을 택하라. TypeScript 와 Python 은 1 인 바이브 코더가 평생 만들 것의 대부분을 커버하며, AI 어시스턴트가 가장 안정적으로 다루는 언어다. 바이브 코딩에 “최고”인 언어는 보통 AI 가 가장 잘 아는 언어—즉 인기 있는 언어다. 이국적인 선택은 구체적인 이유가 생겼을 때를 위해 아껴 두고, 어느 쪽이든 문법은 AI 에게 맡겨라.

복습과 연습

핵심 요약

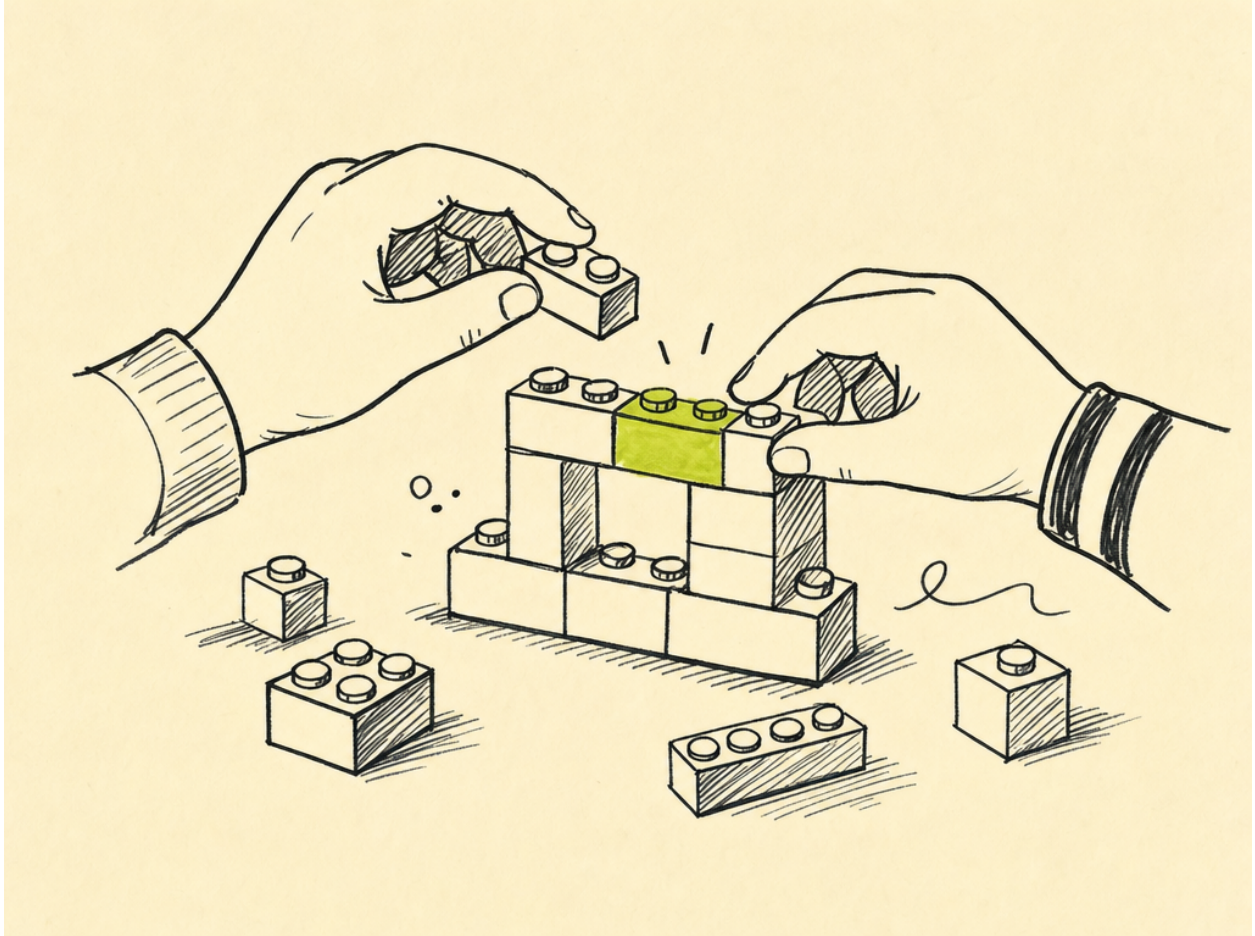
- AI 는 인기 없는 언어보다 인기 있는 언어를 훨씬 유창하게 다룬다—인기는 하나의 기능이며, 프로젝트가 순조롭게 풀릴지를 좌우하는 가장 큰 단일 요인일 때가 많다.
- TypeScript 와 Python 은 안전한 기본값이다: 웹/풀스택은 TypeScript, AI/데이터/자동화는 Python 이며, 둘 다 AI 지원이 뛰어나다.
- Go(단일 바이너리 백엔드), Rust(속도/안전성, 다만 수정 루프가 더 많음), 그리고 나머지는 각자 좁은 강점 영역이 있다—구체적인 이유가 있을 때만 손을 뻗어라.
- SQL 은 양자택일의 선택이 아니다. 앱이 데이터를 저장하는 순간, 어떤 언어를 고르든 그 옆에 함께 산다.
- 그럴듯하게 들리는 것이 아니라 당신의 목표에 언어를 맞춰라—첫 프로젝트에서는 추진력이 최적화를 이긴다.

직접 해보기

정말로 만들고 싶은 프로젝트 아이디어 하나를 골라 그 목표를 한 문장으로 적어 보라. 그것을 “목표로 고르기” 목록의 한 줄에 맞춘 다음, AI 어시스턴트에게 그 선택을 확인하거나 반박해 달라고 요청하라. 기본 스택 (언어, 프레임워크, 어디에 배포할지) 을 세 개의 불릿으로 정리하게 하고, 다른 언어가 더 잘 맞을 한 가지 경우를 짚어 달라고 하라. 코드 한 줄 쓰기 전에 근거 있는 스택 결정을 손에 쥐고 끝나게 된다.

다음 프로젝트를 만들고 싶어요: [한두 문장으로 설명].
이것을 vibe coding 하기에 가장 좋은 언어와 프레임워크를 추천해줘. 기준은 순수 성능이 아니라 AI 어시스턴트가 얼마나 매끄럽게 코드를 작성하고 고칠 수 있는가야. 알려줘: (1) 언어와 주요 프레임워크, (2) 어디에 배포할지, (3) 주의할 현실적인 함정 하나, (4) 다른 언어가 오히려 더 나은 선택이 되는 시나리오 하나. 짧게 답해줘.

기능 만들기, 바이브 하나씩



이제 당신은 AI 와 대화해 코드를 쓰게 할 수 있다. 하지만 진짜 기능은 프롬프트 하나보다 크다. 출시하는 사람과 막혀 버리는 사람을 가르는 기술은, 기능을 작은 조각으로 쪼개고 AI 를 그 조각들 사이로 한 번에 하나씩 몰아가는 법—매 단계마다 앱이 동작하도록 유지하는 법—을 아는 것이다.

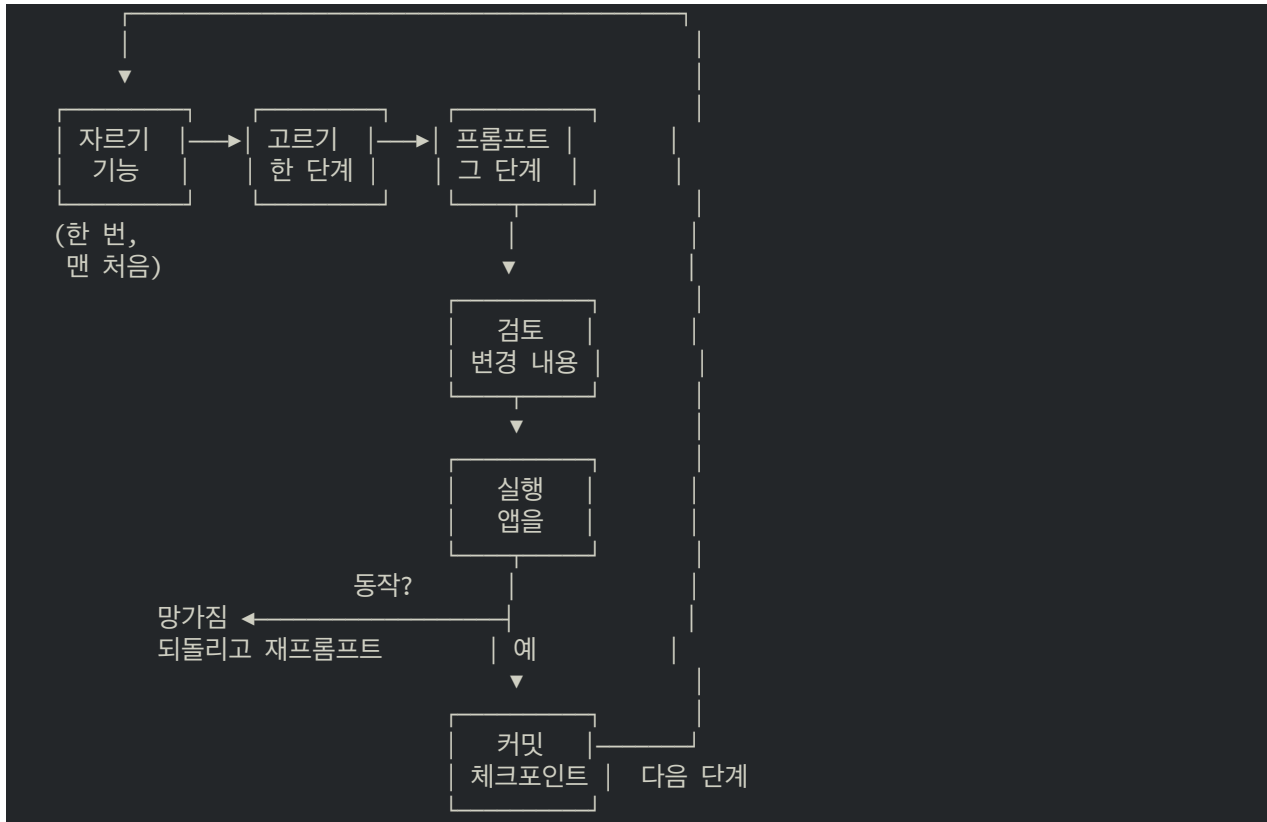
이 장은 핵심 빌드 루프다. 한 번 익히면 앞으로 만들 모든 기능에 재사용하게 된다. 이것이 꾸준히 자라는 프로젝트와, 재미있어지는 바로 그 순간 제 무게에 무너지는 프로젝트를 가르는 차이다.

빌드 루프

여기 당신이 반복해서 돌릴 루프가 있다. 일부러 작게 만들었다.

1. **기능을 작은 수직 단계들로 잘라라.** 각 단계는 몇 분 안에 만들고, 실행하고, 동작하는 것을 볼 수 있는 무언가여야 한다.
2. **한 단계만 골라라.** 딱 하나. 지금은 나머지 전부를 무시하라.
3. **그 한 단계만 프롬프트하라.** 무엇이 바뀌고 무엇이 그대로인지 구체적으로 말하라.
4. **변경 내용을 검토하라.** AI 가 쓴 것을 읽어라. 모든 줄을 이해할 필요는 없지만, 명백히 잘못된 게 있는지 훑어보라.
5. **앱을 실행하라.** 실제로 버튼을 누르고, 페이지를 불러오고, 엔드포인트를 쳐라. 그 단계가 동작하는지 확인하라.
6. **커밋하라.** 짧은 메시지와 함께 버전 관리에 체크포인트를 저장하라.
7. 다음 단계로 2 번부터 **반복하라**—기능이 완성될 때까지.

이 루프를 하나의 순환으로 그리면 이렇게 생겼다—한 바퀴에 작은 단계 하나씩, 기능이 끝날 때까지 계속 돈다:



핵심은, 당신이 동작하는 앱에서 결코 멀리 떨어지지 않는다는 것이다. 4 번 단계에서 뭔가 망가진 게 나오면, 되돌릴 작은 변경 하나뿐이지 영킨 덩어리가 아니다.

4 번부터 6 번 단계를 건너뛰는 사람은 약 20 분간은 더 빠른 듯 느낀다. 그러다 찾을 수 없는 버그에 부딪힌다. 마지막으로 확실히 동작하던 버전이 100 개의 변경 뒤에 있기 때문이다. 이 루프가 느리게 느껴지는 건, 정확히 그런 종류의 빛을 결코 쌓이게 두지 않기 때문이다.

“수직 슬라이스”의 의미

수직 슬라이스는 한 계층씩 통째로 만드는 대신, 앱 전체를 관통하는 기능의 얇은 한 조각이다—UI 조금, 로직 조금, 데이터 조금.

솔깃하지만 틀린 접근 (수평):

- 데이터베이스 스키마 전체를 만든다.
- 그다음 백엔드 로직 전부를 만든다.
- 그다음 UI 전체를 만든다.

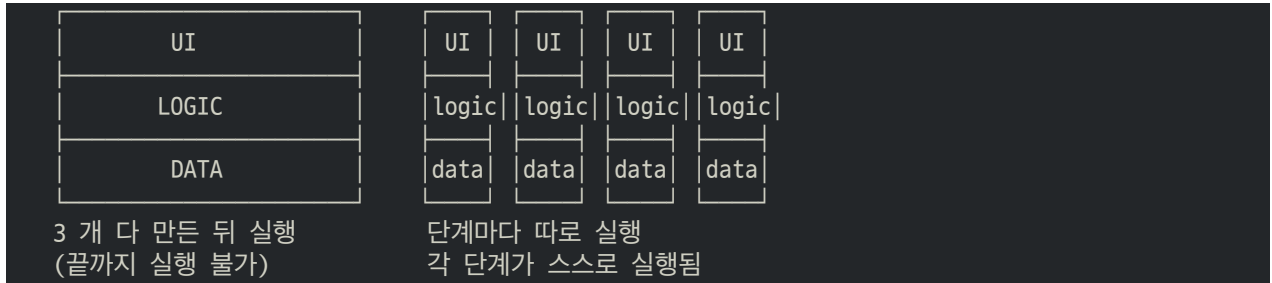
셋이 다 끝날 때까지 아무것도 실행할 수 없으니, 문제를 일찍 잡아낼 수 없다.

더 나은 접근 (수직):

- 1 단계: 가장 단순한 버전의 동작을 가짜 데이터로 수행하는 버튼.
- 2 단계: 실제 데이터를 쓰게 만든다.
- 3 단계: 엡지 케이스를 처리한다.

차이는 나란히 놓고 보면 가장 잘 보인다. 수평은 한 번에 한 계층 전체를 만들고 끝까지 아무것도 실행하지 않는다. 수직은 매 단계마다 모든 계층을 관통하는 얇고 실행 가능한 띠를 만든다:

수평 (위험)	수직 (안전)	실행? ▼
---------	---------	-------



각 슬라이스는 실행 가능하다. 그것이 앱을 항상-동작 상태로 유지하는 비결이다.

좋은 슬라이스를 가능하는 유용한 시험: **시연할 수 있는가?** 화면을 가리키며 “이거 봐”라고 말할 수 있다면, 그것은 진짜 수직 슬라이스다. 정직한 시연이 “음, 이제 스키마가 자리 잡았어”뿐이라면, 당신은 수평 계층을 만든 것이고 그게 맞는지는 훨씬 나중에—보통 최악의 순간에—가서야 알게 된다.

실전 예제: “즐거찾기”기능

작은 기능 하나를 끝에서 끝까지 만들어 보자. 사용자가 목록의 항목을 즐겨찾기로 표시하고, 몇 개를 즐겨찾기했는지 개수를 볼 수 있다.

자르기

내가 이걸 수직 단계로 쪼갠다면 이렇게 한다:

1. 각 항목 옆에 별 버튼을 추가한다 (아직 아무 동작도 안 하고, 그냥 보이기만).
2. 별을 클릭하면 켜짐/꺼짐 모양이 토글된다 (상태만, 저장은 안 됨).
3. 토글을 반영하는 “3 개 즐겨찾기”카운터를 보여준다.
4. 페이지를 새로고침해도 유지되도록 즐겨찾기를 저장한다.

각 단계가 독립적으로 실행 가능하고 눈에 보인다는 점에 주목하라. 어느 단계 다음에서든 멈춰도 여전히 동작하는 앱을 갖게 된다.

1 단계: 버튼

좋은 프롬프트는 파일, 변경 내용, 그리고 건드리지 말아야 할 것을 명시한다:

ItemCard 컴포넌트(src/components/ItemCard.jsx)에서 각 카드의 우상단 모서리에 별 버튼을 추가해줘. 외곽선 별 아이콘을 렌더링해야 해. 아직 클릭 동작은 연결하지 마 — 그냥 보이기만 하게 해. 나머지 카드 레이아웃은 그대로 유지해줘.

앱을 실행하라. 별이 보이는가? 좋다. 커밋하라:

```
git add -A && git commit -m "Add star button to item cards"
```

2 단계: 상태 토글하기

별 버튼을 클릭하면 채워진 모양과 외곽선 모양 사이를 토글하게 만들어줘. 이걸 로컬 컴포넌트 상태로 추적해 — 아직 어디에도 저장하지 마. 채워진 별은 즐겨찾기됨을 의미해.

상태 (state) 는 앱의 단기 기억이라고 생각하면 된다—지금 이 순간 무엇이 참인지, 예컨대 특정 별이 지금 채워져 있는지 같은 것. 이제 **상태와 UI 를 함께** 다루고 있는데, 이건 정상이다. (UI, 즉 사용자 인터페이스는 사람이 보고 클릭하는 앱의 부분일 뿐이다.) AI 는 아마 이런 걸 추가할 것이다:

```
const [isFavorite, setIsFavorite] = useState(false);

<button onClick={() => setIsFavorite(!isFavorite)}>
  {isFavorite ? <StarFilled /> : <StarOutline />}
</button>
```

별 몇 개를 클릭해 보라. 토글되는가? 커밋하라.

3 단계: 카운터

항목 목록 상단에 "★ N개 즐겨찾기" 카운터를 추가해줘. 현재 별이 채워진 항목이 몇 개인지 세어야 해. 카운터가 볼 수 있도록 필요하면 즐겨찾기 상태를 목록 컴포넌트로 끌어올려도 돼.

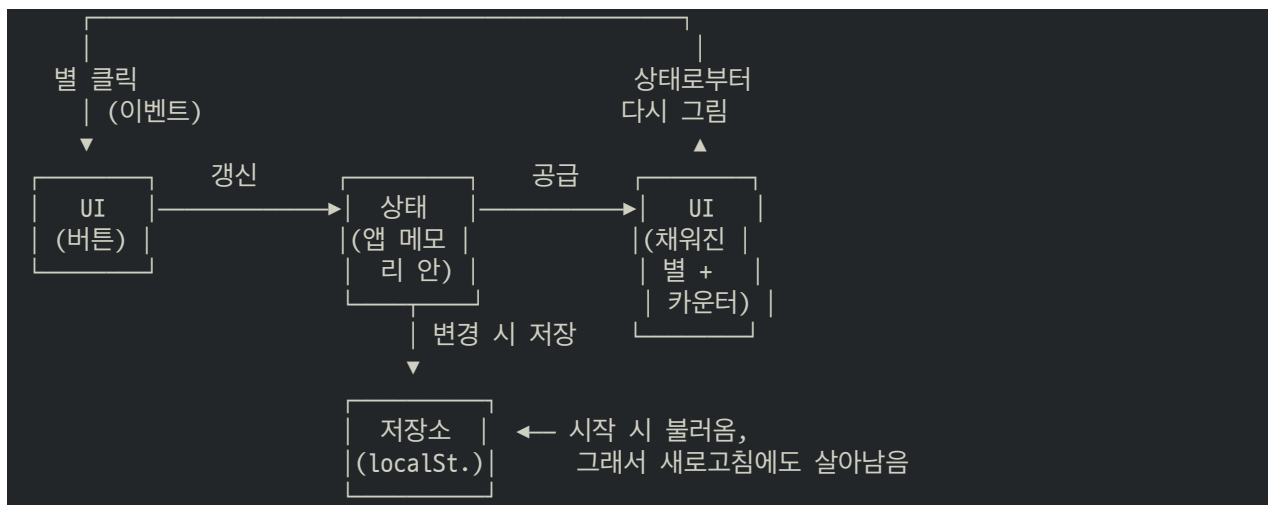
이 단계에서는 AI가 **상태가 사는 위치를 재구성해야** 할 수도 있다—그건 예상된 일이고, 바로 그래서 이걸 별도의 작은 단계로 하는 것이다. 실행하고, 별 몇 개를 토글하고, 숫자가 바뀌는 걸 지켜보라. 커밋하라.

4 단계: 유지되게 만들기

페이지를 새로고침해도 즐겨찾기가 살아남도록 유지해줘. 지금은 localStorage를 써 — 즐겨찾기된 항목 ID 목록이 바뀔 때마다 저장하고, 시작할 때 불러와.

(localStorage는 브라우저가 방문자 본인의 컴퓨터에 두는 작은 메모장 같은 것이라, 앱이 뭔가를 적어 두었다가 새로고침 후에도 다시 찾을 수 있다.) 페이지를 새로고침하라. 여전히 즐겨찾기되어 있는가? 당신은 방금 기능 하나를, 실행 가능한 슬라이스를 하나씩 쌓아 출시했다.

완성된 기능의 조각들이 서로 어떻게 대화하는지 보자. 클릭이 상태로 흘러 내려가고, 화면은 상태에서 다시 그려지며, 상태는 저장소에 거울처럼 비쳐 새로고침에도 살아남는다:



모든 줄을 이해하지 않고 diff 읽기

루프의 4 번 단계는 "변경 내용을 검토하라"이고, 사람들이 가장 많이 얼버무리는 단계다. 코드를 직접 쓸 수 있어야 유용하게 검토할 수 있는 건 아니다. 작은 위험 신호 몇 가지를 훑어볼 줄만 알면 된다.

AI가 변경을 건네주면, 가장 깔끔하게 보는 방법은 diff다—추가되고 제거된 정확한 줄들:

```
git diff
```


+ 로 시작하는 줄은 추가된 것, -로 시작하는 줄은 제거된 것이다. 컴파일러가 아니라 리뷰어처럼 읽어라. 당신이 찾는 것은 이런 것들이다:

- **건드릴 이유가 없는 파일을 건드렸는가?** 카운터를 요청했는데, 왜 auth.js 가 diff 에 있는가?
- **중요해 보이는 걸 지웠는가?** 알아보지 못하는 제거된 줄 (-) 은 질문할 가치가 있다.
- **요청하지 않은 동작을 살짝 바꿨는가?** 새 기본값, 바뀐 조건, “친절한”덤들.
- **모양이 요청과 맞는가?** 한 문장 요청이 200 줄을 만들어냈다면 다시 볼 만하다.

뭔가 이상하면 직접 고칠 필요 없다—그냥 물어라: “왜 auth.js 를 바꿨어? 나는 카운터만 요청했는데.”절반은 AI 가 “잘 잡았다”며 되돌릴 것이다. 그 습관 하나가, 완전히 이해하지 못하는 코드를 아무리 노력하는 것보다 더 많은 버그를 잡아 낸다.

AI 를 풀어줄 때 vs. 직접 조종할 때

모든 단계가 똑같이 핸들을 꼭 잡아야 하는 건 아니다.

AI 를 자유롭게 풀어줄 때:

- 단계가 작고 잘 정의되어 있다 (“버튼 추가”).
- AI 가 분명히 수천 번 봤을 흔한 패턴이다.
- 실수가 값싸고 명백하다 (앱을 실행하는 순간 바로 보인다).

직접 조종할 때—더 작은 프롬프트, 더 면밀한 검토:

- 변경이 쉽게 복구할 수 없는 데이터를 건드린다 (레코드 삭제, 마이그레이션).
- AI 가 같은 것을 빙빙 돌며 계속 “고친다”.
- 동작이 미묘해서 화면을 훑듯 보는 것만으로 드러나지 않는다 (금액 계산, 권한, 인증).
- 두 번 시도했는데 나아지기는커녕 더 나빠지고 있다.

믿을 만한 규칙: 폭발 반경이 클수록, 단계는 작게.

앱을 항상-동작 상태로 유지하기

이것이 나머지 모든 것을 가능하게 하는 규율이다:

- **매 단계마다 실행하라.** 세 단계마다가 아니라. 매번.
- **동작하는 상태만 커밋하라.** 커밋은 돌아갈 수 있는 저장 지점이다. 어떤 단계가 1~2 분 안에 고칠 수 없는 무언가를 망가뜨렸다면, 그것을 버리고 (git restore . 또는 revert) 더 명확한 요청으로 다시 프롬프트하라—망가진 토대 위에 수정을 쌓지 마라.
- **단계당 한 가지 관심사.** “저것도 빨리 고쳐버릴까”하는 유혹이 들면 참아라. 현재 슬라이스를 끝내고, 커밋하고, 그 다음 새 단계를 시작하라.
- **버그를 진단하지 말고 묘사하라.** 뭔가 망가지면 본 것을 정확히 AI 에게 말하라: “두 번째 별을 클릭한 후 카운터가 2 가 아니라 1 을 보여줘.”원인은 AI 가 찾게 하라.

작은 단위로 커밋하기

작은 커밋은 당신의 안전망이다. 각 커밋은 하나의 일관된 단계여야 한다:

```
git commit -m "Add star button to item cards"
git commit -m "Toggle favorite state on star click"
git commit -m "Add favorites counter to list header"
git commit -m "Persist favorites in localStorage"
```

4 단계가 엇나가도 처음 세 단계는 안전하고 잃는 게 거의 없다. 크고 드문 커밋은 사람들이 자기 코드를 건드리기 두려워하게 되는 길이다.

슬라이스가 너무 클 때

가끔은 “한 단계”를 골랐는데도 일이 틀어진다—AI가 앱의 절반을 다시 쓰고, diff는 읽을 수 없고, 실행하면 새 버그가 한꺼번에 셋이나 튀어나온다. 그건 거의 운이 나빠서가 아니다. 슬라이스가 너무 컸다는 신호다.

해법은 더 얇게 자르는 것이다. “유지되는 즐겨찾기 추가”가 터졌다면, 그건 한 슬라이스가 아니었다. 단일 프롬프트를 변장으로 쓴 기능 전체였다. 마지막 좋은 커밋으로 돌아가 쪼개라:

이걸 한 방에 요청하지 마:

"카운터와 localStorage 유지가 있는 즐겨찾기 기능을 추가해줘."

대신 이렇게, 커밋마다 하나씩 요청해:

"각 항목 카드에 별 버튼을 추가해줘. 아직 동작은 없이."

"별이 로컬 상태로 채워짐/외곽선을 토글하게 만들어줘."

"채워진 별을 반영하는 카운터를 추가해줘."

"즐거찾기된 ID를 localStorage에 유지하고 시작 시 불러와."

좋은 어림 규칙: 한 단계가 하는 일을 짧은 한 문장으로 묘사할 수 없다면, 그건 두 단계다. 그리고 어떤 단계의 diff가 1분 안에 훑어보기엔 너무 길다면, 다음 단계는 더 작아야 한다. 자르는 직전 슬라이스가 어디서 너무 두꺼웠는지 감을 잡으며 점점 더 잘하게 되는 기술이다.

핵심 정리

기능은 하나의 거대한 프롬프트가 아니다—작고 실행 가능한 슬라이스들의 더미다. 잘라라, 한 단계만 프롬프트하라, 검토하라, 실행하라, 커밋하라, 반복하라. 쉬운 것은 AI를 풀어두고, 위험한 것 근처에서는 단단히 조종하라. 매 체크포인트마다 앱이 동작하게 유지하면, 당신은 늘 작은 되돌리기 하나로 단단한 땅에 닿을 수 있다.

다음 장에서는 한 단계가 계획대로 풀리지 않을 때 무엇을 해야 하는지—바이브로 디버깅하기—를 살펴본다.

복습과 연습

핵심 요약

- 기능은 하나의 거대한 프롬프트가 아니라 작은 수직 슬라이스들의 더미다—다음으로 넘어가기 전에 각 슬라이스를 만들고, 실행하고, 동작하는 것을 직접 보라.
- 매번 빌드 루프를 돌려라: 슬라이스 → 한 단계 고르기 → 프롬프트 → diff 검토 → 앱 실행 → 커밋 → 반복.
- diff는 컴파일러가 아니라 리뷰어처럼 읽어야—건드리지 말았어야 할 파일, 말없이 바뀐 동작, 요청보다 훨씬 큰 diff를 살펴라.
- 작고, 비용이 싸고, 흔한 단계는 AI를 풀어줘라. 파급 범위가 큰 곳에서는 직접 조종하라 (더 작은 프롬프트, 더 면밀한 검토).
- 동작하는 상태만 작은 단위로 커밋하라—슬라이스가 어긋나면, 망가진 토대 위에 수정을 쌓지 말고 버리고 다시 프롬프트하라.

직접 해보기

프로젝트에 넣고 싶은 작은 기능 하나를 골라 3~5개의 수직 단계로 잘라라. 각 단계를 짧은 한 문장으로 적어라. 첫 번째 단계만 만들어 보라: 그것을 프롬프트하고, git diff를 실행해 변경을 훑고, 앱을 실행해 동작을 확인한 뒤, 한 줄 메시지로 커밋하라. 지금은 거기서 멈춰라. 목표는 루프가 한 번 깔끔하게 도는 것을 느끼는 것이다—실행 가능한 슬라이스 하나, 실제로 읽은 diff 하나, 돌아갈 수 있는 커밋 하나.

이 기능을 추가하고 싶어: [기능을 한두 문장으로 설명].
아직 코드는 쓰지 마. 먼저 이걸 3~5개의 수직 단계로 잘라줘. 각 단계는 내가 직접 만들고, 실행하고, 스스로 동작하는 걸 볼 수 있는 무언가여야 해. 매 단계 후에도 앱이 계속 동작하도록 순서를 정해줘. 각 단계마다 한 문장 설명을 주고, 네가 가장 먼저 바꿀 파일 이름을 말해줘. 그런 다음 기다려 — 내가 하나씩 만들어 달라고 요청할게.

AI 로 디버깅하기



당신이 만드는 모든 프로그램은 망가질 것이다. 그것은 당신이 뭔가 잘못하고 있다는 신호가 아니다—소프트웨어의 정상 상태다. 버그는 실력의 실패가 아니다. 진짜 코드가 진짜 데이터, 진짜 사용자, 그리고 아무도 상상하지 못한 진짜 예외 상황과 만났을 때 일어나는 일이다. 좌절스러운 오후와 5 분 만의 수정을 가르는 차이는 방법이다. 바이브 코딩으로 디버깅할 때, 스택 트레이스가 이해될 때까지 그것을 노려보는 사람은 당신이 아니다—스택 트레이스란 프로그램이 멈췄을 때 출력하는 오류 보고서인데, 뒤에서 자세히 풀어본다. 당신은 감독이다. 문제를 재현하고, 올바른 증거를 모으고, AI 가 무거운 일을 처리할 수 있도록 깔끔하게 잘 정리된 사건을 건넨다. 이 장은 바로 그것을 위한, 반복 가능한 절차를 알려준다—버그가 한 줄짜리 오타든, 세 겹 깊이로 얽힌 경쟁 상태 (race condition) 든 통하는 절차를. (경쟁 상태란 타이밍 버그다. 두 일이 당신이 예상하지 못한 순서로 일어나는 것—마지막 한 자리를 두 사람이 동시에 잡아채는 것처럼.)

핵심 마음가짐

초보자가 저지르는 가장 큰 실수는 채팅에 “안 돼요, 고쳐주세요”를 붙여넣고 기대하는 것이다. AI 는 당신의 화면도, 데이터도, 당신이 무엇을 기대했는지도 볼 수 없다. 당신이 말해주는 것만 안다. 당신이 아무것도 주지 않으면, AI 는 할 수 있는 유일한 일을 한다. 추측이다—그리고 우연히 틀린 자신만만한 추측은 답이 없는 것보다 더 위험하다. 당신이 그 위에서 행동할 것이기 때문이다. AI 로 디버깅한다는 건 대부분 **올바른 맥락을 AI 에게 먹여서** 추측 대신 추론하게 하는 일이다.

좋은 버그 리포트는—사람에게든 AI 에게든—세 가지 질문에 답한다:

- 무엇을 했는가? (그것을 유발하는 단계들)
- 무엇이 일어나길 기대했는가?
- 실제로 무엇이 일어났는가? (에러, 잘못된 출력, 충돌)

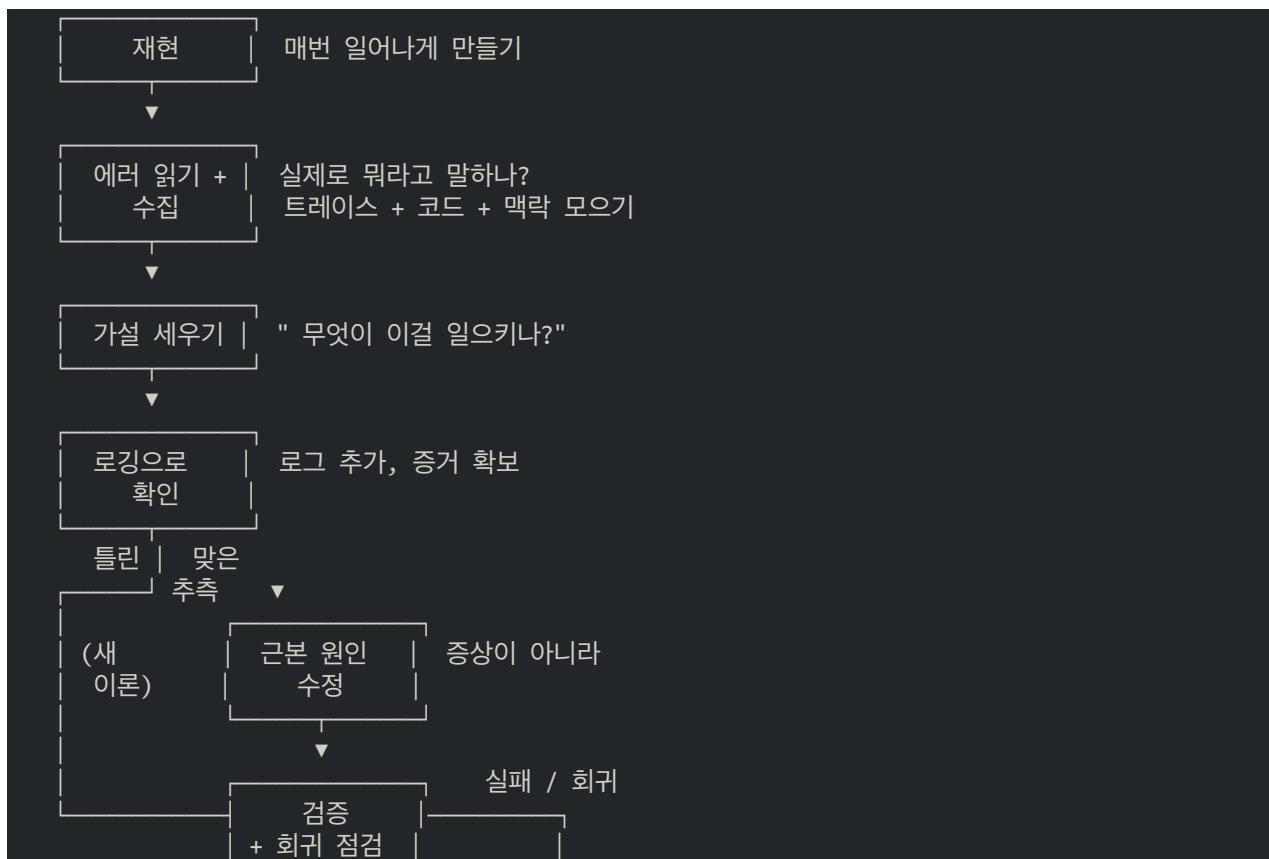
이 셋에 답할 수 있다면, 당신은 이미 대부분의 사람보다 앞서 있다. 이 셋이 중요한 이유는, 버그란 정의상 당신이 기대한 것과 실제로 일어난 것 사이의 간극이기 때문이다. AI 는 양쪽을 다 볼 수 없는 간극을 찾을 수 없다. “망가졌어”는 한 쪽만 설명한다. “환영 화면을 기대했는데, 빈 페이지와 이 에러가 나왔어”는 양쪽을 다 설명한다—그러면 간극이 스스로 이름을 댄다.

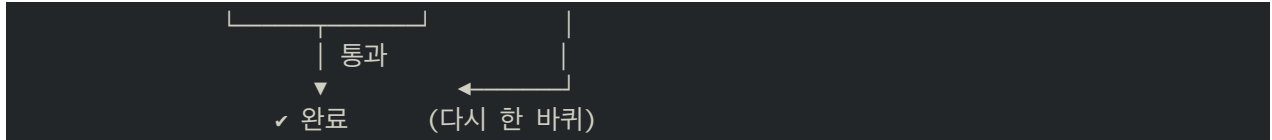
디버깅 체크리스트

이것들을 순서대로 밟아라. 곧장 “고쳐줘”로 건너뛰는 것이 AI 를 빙빙 돌게 만드는 원인이다.

1. **안정적으로 재현하라.** 매번 버그를 일으키는 정확한 단계를 찾아라. 재현할 수 없는 버그는 고쳤다고 확인할 수도 없는 버그다—사라졌는지 영원히 추측만 하게 된다. 간헐적으로 일어난다면, 그것을 켜고 끄는 변수를 사냥하라. 특정 입력, 로그인 대 로그아웃 상태, 느린 네트워크 같은 것.
2. **실제 에러를 읽어라.** 대충 훑지 마라. 에러 메시지와 스택 트레이스의 맨 위는 보통 무엇이 잘못된 파일과 줄을 가리킨다. 초보자는 종종 화면에서 가장 유용한 문장을, 무섭게 보인다는 이유로 그냥 지나친다.
3. **맥락을 모아라.** 에러 텍스트, 스택 트레이스, 관련 코드, 그리고 무엇을 기대했는지에 대한 한 줄 설명을 모아라.
4. **AI 에게 전체 그림을 건네라.** 위의 모든 것을 한 메시지에 붙여넣어라. AI 가 가설을 세우게 하라.
5. **수정이 아니라 가설을 세워라.** “이거 고쳐줘”보다 먼저 “무엇이 이것을 일으킬 수 있어?”라고 물어라. 틀린 이론 위의 틀린 수정은 시간을 낭비한다—그리고 때로는 첫 번째 버그 위에 두 번째 버그를 얹는다.
6. **가설을 시험할 로깅을 추가하라.** 바꾸기 전에 코드가 실제로 무엇을 하는지 확인하라.
7. **문제를 이진 탐색하라.** 의심 영역을 반복해서 절반으로 잘라 어디서 깨지는지 좁혀라.
8. **증상이 아니라 근본 원인을 고쳐라.** 수정이 눈에 보이는 결함만이 아니라 왜 깨졌는지를 다루는지 확실히 하라.
9. **수정을 검증하고 회귀를 점검하라.** 재현 단계를 다시 돌려라. 그다음 주변 기능이 여전히 동작하는지 확인하라—회귀란 수정이 잘 돌아가던 무언가를 슬그머니 망가뜨리는 것이다. 한 곳의 누수를 막았더니 바로 옆 파이프가 터지는 것처럼.

그 체크리스트는 사실 버그가 사라질 때까지 타고 도는 하나의 루프다. 각 상자가 다음에게 증거를 건네고, 검증에 실패하면 배운 것을 가지고 다시 한 바퀴 돌게 된다:





순서는 임의가 아니다. 각 단계는 다음 단계가 필요로 하는 증거를 만들어낸다. 재현은 읽을 것을 주고, 읽기는 모을 맥락을 주고, 맥락은 AI가 가설을 세울 거리를 준다. 한 단계를 건너뛰면 다음 단계에 구멍을 먹이는 셈이다.

AI에게 올바른 맥락 먹이기

여기에 당신의 지렛대 대부분이 있다. 버그를 보고할 때 다음을 포함하라:

- **에러 메시지를 그대로.** 정확히 복사하라—의역하지 마라. 에러를 의역하는 것은 전화로 사진을 설명하는 것과 같다. 정작 중요했던 하나의 디테일을 빠뜨리게 된다.
- **스택 트레이스**, 특히 위쪽 몇 줄.
- **관련 코드**—에러를 던진 함수, 그리고 그것이 호출하는 모든 것.
- **무엇을 기대했고 무엇이 일어났는지.**

강력한 디버깅 프롬프트는 실제로 이렇게 생겼다:

회원가입 폼을 제출할 때 에러가 나. 전체 출력은 다음과 같아:

```
TypeError: Cannot read properties of undefined (reading 'email')
    at validateUser (src/auth/validate.js:14:23)
    at handleSignup (src/routes/signup.js:31:10)
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
```

재현 단계: 폼을 채우고 "Sign up"을 클릭한다.

기대: "Welcome" 메시지.

실제: 페이지가 빈 화면이 되고 위 에러가 콘솔에 나타난다.

여기 validate.js야 (14번 줄 근처):

[함수 붙여넣기]

그리고 이것 호출하는 handleSignup 함수야:

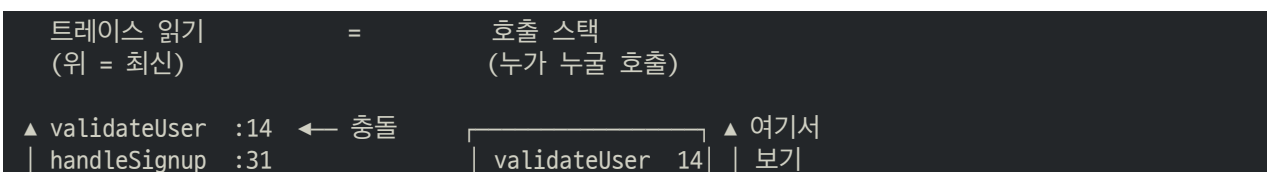
[함수 붙여넣기]

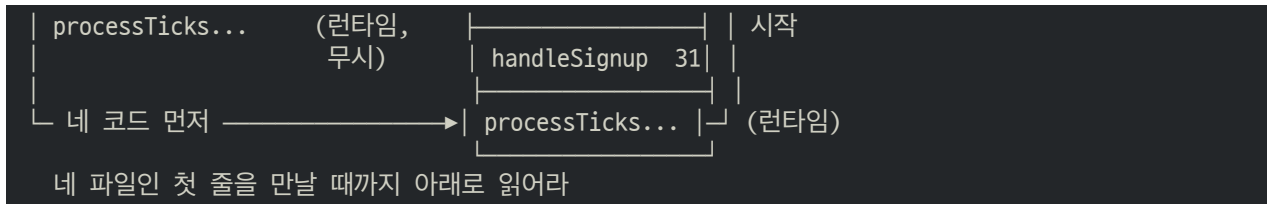
무엇이 이것 일으킬 수 있고, 무언가를 바꾸기 전에 어떻게 확인할 수 있을까?

마지막 줄에 주목하라. 당신은 즉각적인 수정이 아니라 가설과 확인 단계를 요청하고 있다. 그것이 AI를 정직하게 유지한다.

당신이 붙여넣는 트레이스를 읽는 법을 아는 것도 도움이 된다. 스택 트레이스는 최신순으로 정렬된 호출 이력이다. 맨 위 줄은 프로그램이 실제로 터진 곳이고, 그 아래 각 줄은 그 위의 함수를 호출한 함수다. 그래서 예시에서 validateUser는 email이 undefined인 무언가에서 읽힌 곳이고, handleSignup은 validateUser를 호출한 쪽이다. 당신 코드 아래의 프레임—processTicksAndRejections 같은—은 언어 런타임의 것이다. 보통 무시해도 된다. 당신의 파일을 가리키는 첫 줄을 만날 때까지 위에서 아래로 읽어라. 그곳이 거의 항상 살펴보기 시작할 자리다.

같은 트레이스를 호출 더미로 읽으면 순서가 분명해진다. 충돌은 맨 위에 있고, 그 아래 각 프레임은 바로 위 프레임을 호출한 쪽이다:





이해 안 되는 에러 읽기

머지않아 알파벳 수프처럼 보이는 에러를 만날 것이다—긴 타입 이름, 죄다 프레임워크 내부인 스택, 본 적 없는 전문 용어로 된 메시지. 그것에 발이 묶이지 마라. 에러는 데이터지, 당신의 능력에 대한 판결이 아니다. 메시지 자체가 불투명할 때는, 그것을 붙여넣고 AI 에게 고치기 전에 번역하라고 요청하라:

이 에러가 이해가 안 돼. 어떤 수정도 제안하기 전에, 쉬운 말로 설명해줘:
이게 실제로 무슨 말을 하는 거고, 어느 줄을 가리키고 있고,
정확히 이런 에러를 보통 무엇이 일으키는지.

[에러와 스택 트레이스 전체 붙여넣기]

쉬운 말로 의미를 먼저 묻는 것은 두 가지를 한다. 그것은 당신에게 패턴을 가르쳐서, 다음에 그 에러를 보면 스스로 알아보게 한다—그리고 몇 달이 지나면 이것이 당신이 실제로 자신의 스택을 읽는 법을 배우는 방식이 된다. 그리고 그것은 AI 가 일반적인 패치로 건너뛰는 대신 에러가 하는 말에 수정을 근거하게 강제한다. 가장 흔한 에러 함정을 경계하라. 14 번 줄을 가리키는 에러는 종종 완전히 다른 곳에서 일어난 것이다. 트레이스의 줄은 프로그램이 문제를 알아챈 곳이지, 항상 문제가 시작된 곳은 아니다.

가설, 로깅, 그리고 이진 탐색

AI 가 원인을 제시하면 그냥 받아들이지 마라. **값싸게 시험하라**. 가장 값싼 시험은 보통 로그 한 줄이다.

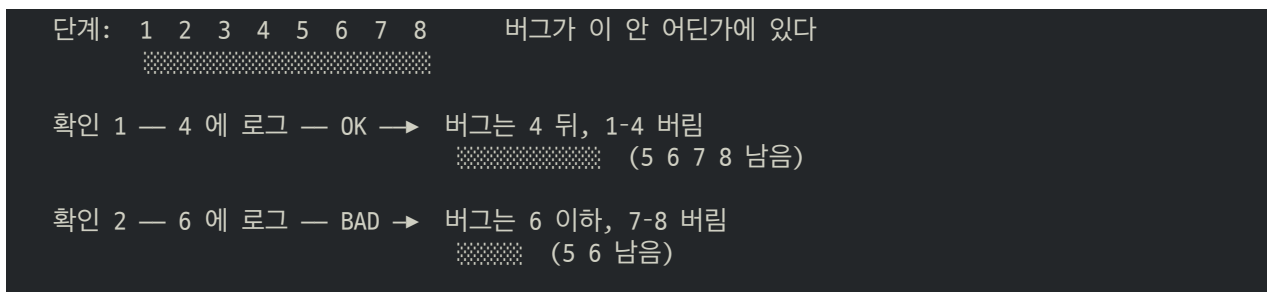
AI 에게 임시 로깅을 추가하라고 요청하라:

“무언가 바뀌기 전에, 14 번 줄 바로 앞에서 user 의 값을 보여주는 console.log(또는 print) 를 추가해줘.
그게 실제로 undefined 인지 볼 수 있게.”

실행하고, 출력을 다시 붙여넣어라. 이제 당신은 이론이 아니라 증거를 가졌다. user 가 undefined 일 수도 있다—가설을 확인하는 것이다. 멀쩡하고 진짜 범인이 한 단계 위에 있을 수도 있다. 어느 쪽이든 당신은 앞으로 나아갔다. 조사하는 동안에는 로그를 넉넉히 찍어라. 변수의 값, 그리고 그게 어느 변수이고 로그가 어디 있는지 알려주는 라벨까지. undefined 를 찍는 맨 console.log(user) 는 console.log("user at validate.js:14 =", user) 보다 훨씬 덜 유용하다. 그리고 버그가 고쳐지면 임시 로그를 다시 빼내는 걸 잊지 마라—남겨진 디버그 잡음은 그 자체로 작은 어질러짐이다.

버그가 긴 과정 어딘가에 숨어 있고 어디인지 알 수 없을 때는 **이진 탐색**을 써라. 의심 구간을 절반으로 잘라라: 중간 지점에 로그를 추가하라.

핵심 아이디어: 각 확인이 남은 것의 절반을 버리므로, 거대한 탐색 공간이 빠르게 줄어든다. 여기 버그가 1 단계와 8 단계 사이 어딘가에 숨어 있다—세 번의 확인으로 그것을 몰아넣는다:



확인 3 — 5 에 로그 — OK → 버그는 6 단계 ✓ 찾음

8 단계 → 3 번 확인. 1000 단계 → 약 10 번 확인.

거기서 값이 올바르게 보였는가? 그렇다면 버그는 그 뒤에 있다—그 절반을 탐색하라. 이미 잘못 보였는가? 버그는 그 앞에 있다—그 절반을 탐색하라. 각 단계가 탐색 공간을 반으로 줄이므로, 천 줄짜리 경로라도 약 열 번의 확인으로 좁혀진다. AI 에게 명시적으로 말하라: “이걸 이등분하게 도와줘—로그를 찍어야 할 중간 지점이 어디야?”같은 기법은 코드 뿐 아니라 이력에도 통한다. 어떤 기능이 지난주에는 동작했는데 오늘 깨졌다면, 커밋을 이등분해서 버그를 들어온 정확한 변경을 찾아라.

근본 원인 대 증상

디버깅에서 가장 솔깃한 함정은 원인을 고치지 않고 에러를 사라지게 만드는 패치다. `user` 가 `undefined` 인데 `if (user) { ... }` 로 그것을 “고치면” 에러는 멈춘다—하지만 이제 회원가입이 조용히 아무것도 안 하게 되어 더 나쁘다. 요란한 충돌은 적어도 뭔가 잘못됐다고 말해줬는데, 이제는 아무것도 말해주지 않는다. 진짜 질문은 왜 여기서 `user` 가 `undefined` 인가? 이다. 데이터베이스 조회가 실패했을 수도 있다. 필드 이름이 바뀌었을 수도 있다. 상류에서 `await` 가 빠졌을 수도 있다.

AI 가 수정을 제안하면, 이렇게 물어라:

- “이게 원인을 고치는 거야, 아니면 증상을 가리는 거야?”
- “그 값이 잘못된 원래 이유가 뭐였어?”
- “이 같은 근본 원인이 다른 무언가도 깨뜨릴 수 있어?”

근본 원인을 고치면 보통 코드가 더 단순해지거나 더 명확해진다. 증상 패치는 보통 가드 하나, 에러를 삼키는 `try/catch`, 또는 진짜 문제를 덮어버리는 기본값을 추가한다. 두 번째 종류를 의심하라. 배워둘 만한 신호가 있다. 어떤 수정이 올바른 일을 하는 대신 충돌을 막는 것이 유일한 임무인 코드를 더한다면, 당신은 아마 증상을 패치하고 있는 것이다. 진짜 수정은 불가능한 상태를 단지 견디는 게 아니라 아예 제거하는 경향이 있다.

AI 가 빙빙 돌 때

때때로 AI 가 수정을 제안하고, 그게 안 되고, 또 다른 걸 제안하고, 그것도 안 되고, 당신은 “대신 이것 시도해봐” 루프에 세 번째로 빠져 있다. 이건 막다른 길이 아니라 신호다. 이렇게 하라:

- **멈추고 맥락을 초기화하라.** 시도한 모든 것이 아니라, 이제 확실히 아는 모든 것 (로그로 확인된 것) 을 요약하는 새 메시지로 시작하라. 실패한 시도로 가득한 긴 대화는 AI 의 다음 답을 적극적으로 오염시킨다—AI 는 자기 이력 속의 막다른 길에 계속 패턴 매칭을 한다.
- **무엇을 배제했는지 진술하라.** “`user` 가 `undefined` 이고 DB 쿼리가 `null` 을 반환하는 걸 확인했어. 쿼리 문자열은 맞아 보여. 그러니 문제는 쿼리의 상류에 있어.” 이것은 AI 가 막다른 길을 다시 제안하지 못하게 막는다.
- **고치라고 말고, 설명하라고 시켜라.** “요청에서 14 번 줄까지 `user` 가 어떻게 값을 얻는지 단계별로 짚어줘.” 설명을 강제하면 종종 빈틈이 드러난다.
- **추측을 더하지 말고 로깅을 더하라.** 이론이 바닷나면 증거가 앞으로 나아가는 길이다.
- **문제를 줄여라.** 여전히 버그를 재현하는 가능한 가장 작은 조각을 만들도록 AI 에게 도움을 청하라. 잡음이 사라지면 버그가 종종 명백해진다—그리고 작은 버전에서 재현되지 않는다면, 원인이 당신이 제거한 부분에 있다는 걸 배운 것이다.

간직하기 전에 수정을 이해하라

동작하지만 당신이 이해하지 못하는 수정은 해결책이 아니라 빛이다—그리고 근처의 무언가가 다음에 깨질 때 이자가 돌아온다. AI 가 타이핑을 하니, 초록색 체크 표시를 받아들이고 왜 코드가 이제 초록색인지 결코 알지 못한 채 넘어가기 쉽다. 그것에 저항하라. 수정을 간직하기 전에, AI 가 그것을 해명하게 하라:

- “이게 왜 고치는지 **한두 문장으로 설명해줘.**” 설명이 모호하면—“이게 더 견고하게 만들어”, “이게 예외 케이스를 처리해”—되받아쳐라. 진짜 설명은 구체적인 원인과 변경이 그것을 어떻게 제거하는지 짚는다.
- “그 버그가 **뭘었는지, 원인과 결과로 말해줘.**” 당신은 그 답을 자기 말로 되웁을 수 있어야 한다. 못 한다면, 당신은 아직 자기 프로그램을 이해하지 못한 것이다.

- “이 코드 경로를 건드리는 다른 무엇을 내가 다시 테스트해야 해?” 이것은 수정을 회귀 점검으로 바꾼다.

이것은 자신이 출시하는 것을 이해하라는 철학을 가장 어려운 경우에 적용한 것이다. 수정을 맨바닥에서 작성할 수 있어야 하는 건 아니다—그건 AI 의 일이다. 하지만 원인과 결과의 이야기는 머릿속에 쥐고 있어야 한다. 그것이 다시 깨질 때 새벽 2 시에 거기 있을 사람은 당신이고, 그때 유일하게 도움이 되는 건 그 물건이 실제로 어떻게 동작하는지 아는 것이기 때문이다.

핵심 정리

AI 로 디버깅하는 것은 답을 아는 일이 아니다. 규율 있는 절차를 돌리는 일이다: 재현하고, 맥락을 모으고, 가설을 세우고, 로깅으로 확인하고, 이등분하고, 근본 원인을 고친다—그다음 그 수정을 정직하게 간직할 만큼 충분히 이해한다. 감독으로서 당신의 일은 조사를 증거에 단단히 묶어두고, 루프가 정제되었을 때 그것을 알아차리는 것이다. 그렇게 하면 대부분의 버그는 더 이상 미스터리가 아니라, 진짜 소프트웨어를 만드는 또 하나의 단계가 된다.

복습과 연습

핵심 요약

- 버그는 기대한 것과 실제로 일어난 것 사이의 간극이다—AI 에게 양쪽 모두와 정확한 재현 단계를 줘라. 절대“안 돼요”만 던지지 마라.
- 에러는 글자 그대로 붙여 넣고, 스택 트레이스의 맨 위를 함께 줘라. 당신의 파일을 가리키는 첫 줄이 거의 언제나 출발점이다.
- 바꾸기 전에 확인하라: 가설을 검증하는 라벨 붙은 로그 한 줄을 추가하고, (코드나 커밋 이력에서) 이등분으로 매 단계 탐색 공간을 절반으로 줄여라.
- 증상이 아니라 근본 원인을 고쳐라—크래시만 막는 가드나 try/catch 는 보통 진짜 문제를 덮어버린다.
- AI 가 빙빙 돌면, 맥락을 확실히 아는 것으로 리셋하고, 고치라고 하지 말고 설명하라고 하며, 추측 대신 증거를 더 하라.

직접 해보기

다음에 무언가 망가지면, 수정을 요청하기 전에 버그 리포트를 써라. 새 대화에서 AI 에게 세 가지를 줘라: 재현 단계, 기대한 것, 그리고 실제로 일어난 것—에러는 글자 그대로 복사하고 스택 트레이스 맨 위 프레임을 포함해서. 그런 다음“고쳐줘”가 아니라 가설과 그것을 싸게 확인할 방법 (보통 로그 한 줄) 을 요청하라. 그 확인을 실행하고 출력을 다시 붙여 넣은 다음, 그제야 AI 가 수정을 제안하게 하라. “안 되는데 도와줘”보다 길이 얼마나 더 곧은지 느껴 보라.

버그가 있어. 수정을 제안하기 전에, 가설 하나와 그것을 싸게 확인할 방법을 (로그 한 줄 같은) 줘 —아직 코드는 고치지 마.

재현 단계: [무엇을 했는지]
기대: [무엇이 일어나야 했는지]
실제: [무엇이 일어났는지]

전체 에러와 스택 트레이스:
[그대로 붙여넣기]

관련 코드:
[실패한 함수, 그리고 그것이 호출하는 모든 것 붙여넣기]

무엇이 이걸 일으킬 수 있고, 바꾸기 전에 확인하려면 무엇을 로그로 찍어야 할까?

서버 인프라



당신은 노트북에서 동작하는 무언가를 바이브 코딩했다. 이제 다른 사람들이 그것을 쓰게 하고 싶다. 그 간극—“내 컴퓨터에서 돌아감”과 “인터넷에서 돌아감” 사이의 간극—이 바로 인프라가 다루는 것이다. 다행인 점: 그것을 건너기 위해 DevOps 역량이 필요한 건 아니다. 작동하는 멘탈 모델과, 자잘한 설정은 AI가 처리하게 맡길 의지만 있으면 된다.

이 장은 호스팅 지형의 안개를 걷어내고, 트레이드오프를 평이한 말로 알려주며, 1인 빌더를 선택의 늪에 빠지지 않고 프로토타입에서 프로덕션까지 데려가는 기본 경로를 건넨다.

코드가 살 수 있는 다섯 곳

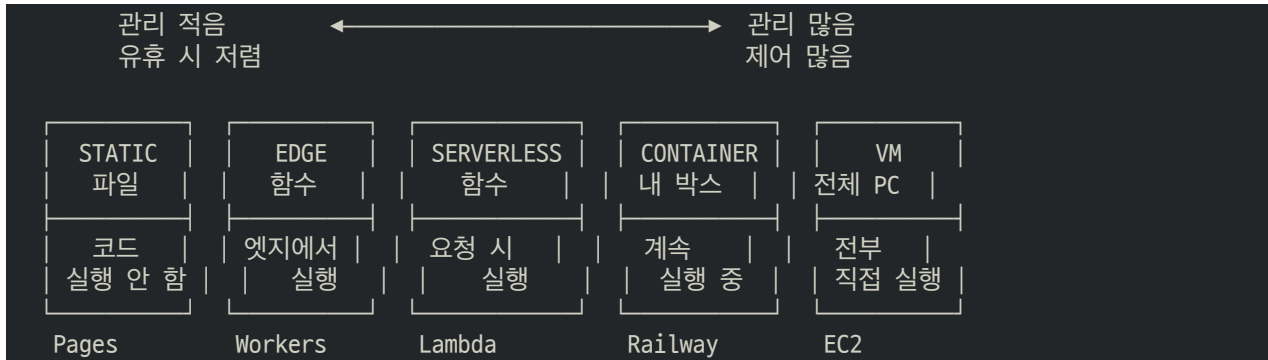
지구상의 거의 모든 호스팅 제품은 이 다섯 범주 중 하나의 변형이다. 이것들을 익히면 마케팅 페이지가 더 이상 헛갈리지 않는다.

- **정적 호스팅**—순수 파일 (HTML(HyperText Markup Language, 페이지의 뼈대), CSS(Cascading Style Sheets, 꾸미는 스타일), JavaScript, 이미지)을 브라우저에 곧장 제공한다. 어떤 서버도 당신의 코드를 “실행”하지 않는다. 그저 파일을 넘겨줄 뿐이다. 인터넷에 올려둔 폴더라고 생각하라. 망가질 것도, 패치할 것도 없기에, 인터넷에 올릴 수 있는 가장 안정적인 것이다. 예: GitHub Pages, Cloudflare Pages, Netlify, Vercel(정적 모드).
- **엣지 함수**—사용자와 물리적으로 가까운, 전 세계 곳곳의 서버에서 돌아가는 작은 코드 조각들. 거의 즉시 시작되며 가벼운 로직에 훌륭하다: 인증 확인, 리다이렉트, 작은 API(애플리케이션 프로그래밍 인터페이스—한 프로그램이 다른 프로그램과 대화하는 통로) 호출. 함정: 기능이 축소된 환경에서 돌아가기에, 모든 라이브러리가 거기서 작동하지는 않는다. 예: Cloudflare Workers, Vercel Edge Functions, Deno Deploy.
- **서버리스 함수**—요청 시에 실행되고 유휴 상태가 되면 사라지는 코드. 시간당이 아니라 요청당 과금된다. 당신은 함수를 작성하고, 플랫폼이 기계를 처리한다. 단일 요청이 몇 분을 돌아야 하기 전까지는 훌륭하다—대부분의

플랫폼은 타임아웃 (흔히 10~60 초) 후에 함수를 끊어버린다. 예: AWS Lambda, Google Cloud Functions, Vercel Functions.

- **컨테이너**—당신의 앱과 그 환경 전체 (올바른 언어 버전, 라이브러리, 시스템 도구) 를 컨테이너라 불리는 휴대 가능한 상자에 담은 것. 플랫폼이 그 상자를 계속 돌려준다. 제어권이 더 많고, 설정은 살짝 더 든다. 그 상자를 정의하는 단 하나의 파일이 Dockerfile 이고, AI 는 그것을 잘 쓴다. 예: Fly.io, Railway, Render, Google Cloud Run.
- **가상 머신 (VM)** —데이터센터에서 빌리는 컴퓨터 한 대 전체. 모든 것—운영체제, 보안 업데이트, 웹 서버, 전부—을 직접 설치하고 관리한다. 최대의 제어권, 최대의 책임. 예: AWS EC2, DigitalOcean Droplets, Hetzner.

다섯 범주를 나란히 놓고 보면, 아래로 갈수록 관리할 것이 적음을 제어권이 많음과 맞바꾼다:

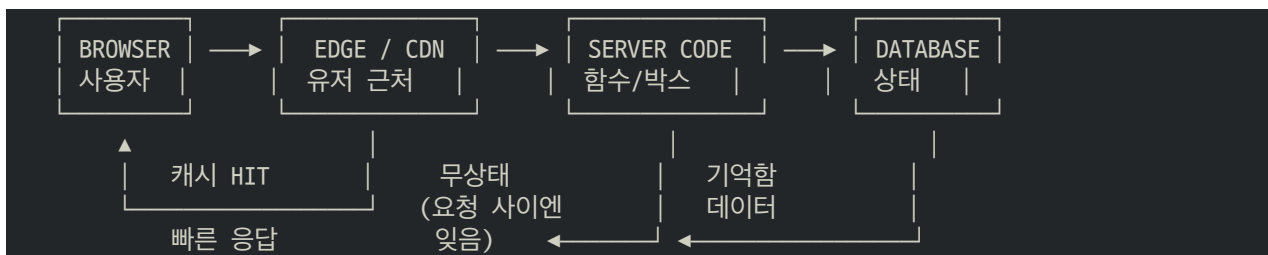


비용에 관한 대략의 어림 규칙: 플랫폼이 당신을 위해 더 많이 관리할수록, 아무도 앱을 쓰지 않을 때 더 저렴하고, 모두가 쓸 때 더 많은 걱정을 대신해 준다.

풀어 쓴 몇 가지 전문 용어

- **콜드 스타트**—코드가 한동안 실행되지 않았을 때, 플랫폼은 그것을 깨워야 한다. 그 첫 요청은 더 느리다 (1 초 미만에서 몇 초까지). 엣지 함수는 이게 거의 없고, 서버리스 함수는 가끔 있으며, 항상 켜져 있는 컨테이너와 VM 은 없다. 대부분의 앱에서는 눈에 띄지 않지만, 빠릿해야 하는 결제 흐름에서는 문제가 될 수 있다.
- **스케일링**—더 많은 사용자를 처리하면서도 쓰러지지 않는 것. 자동 스케일링은 플랫폼이 당신을 위해 용량을 추가 한다는 뜻이다. 수동 스케일링은 당신이 버튼을 클릭하거나, 아니면 부하에 그냥 무너진다는 뜻이다. “0 으로 스케일 (scales to zero)”은 자주 보게 될 관련 표현이다: 아무도 앱을 쓰지 않을 때 비용이 0 이라는 뜻이다.
- **상태 (State)** —계속 남아 있어야 하는 데이터 (데이터베이스, 업로드된 파일, 사용자 세션). 정적·서버리스 계층은 흔히 “무상태 (stateless)”다—요청 사이에 모든 것을 잊는다—그래서 상태는 별도의 데이터베이스나 스토리지 서비스에 산다. 거의 모든 진짜 앱이 하나의 마법 상자가 아니라 “호스팅 더하기 어딘가의 데이터베이스”인 이유가 이것이다.
- **리전 (Region)** —당신의 코드와 데이터가 물리적으로 위치한 세계의 어느 부분. 속도 (가까울수록 빠르다) 에, 때로는 법 (어떤 데이터는 특정 국가에 머물러야 한다) 에 영향을 준다. 첫 프로젝트라면 사용자와 가장 가까운 리전을 고르고 넘어가라.

요청이 거치는 전체 여정과 각 조각이 사는 곳은 이렇다. 브라우저는 가까운 엣지와 대화하고, 엣지는 캐시된 파일을 내 주거나 당신의 코드를 깨우며, 코드는 기억해야 할 것이 있으면 데이터베이스에 손을 뻗는다:



트레이드오프 표

호스팅 유형	비용 (유휴 1분 주)	스케일링	콜드 스타트	복잡도	가장 잘 맞는 곳
정적	무료 1 매우 저렴	자동, 손쉬움	없음	가장 낮음	마케팅 사이트, 문서, 블로그, 프론트엔드
엣지 함수	무료 티어 1 저렴	자동, 글로벌	미미함	낮음	인증, 리다이렉트, 가벼운 API, 개인화
서버리스	유휴 시 \$0 1 사 용량에 따라 증가	자동	가끔 체감됨	낮음 ~ 중간	API, 웹훅, 폭발적/예측 불가 트래픽
컨테이너	실행 중 과금	자동에 가까움, 설정 가능	없음 (워밍 유지 시)	중간	완전한 백엔드, 장시간 작업, 커스텀 런타임
VM	24 시간 내내 과금	수동	없음	가장 높음	특수 하드웨어, 레거시 소프트웨어, 완전한 제어

패턴: 표를 아래로 내려갈수록 단순함과 낮은 유휴 비용을 제어권과 유연성과 맞바꾼다. 대부분의 바이브 코더는 표의 위쪽에 머무르려 애써야 한다.

언제 무엇을 고를까

- **그냥 웹사이트나 프론트엔드?** 정적 호스팅. 끝. 무료고, 빠르며, 거의 망가뜨릴 수 없다.
- **약간의 백엔드 로직이 필요—연락처 폼, 비밀번호 지킬 API 키, 인증 확인?** 엣지 또는 서버리스 함수. 너그러운 무료 티어에 머무르며 아무도 쓰지 않을 때 0 으로 스케일한다.
- **데이터베이스, 백그라운드 작업, 또는 장시간 도는 서버를 기대하는 프레임워크가 있는 진짜 앱?** Railway, Render, Fly.io 같은 컨테이너 플랫폼. 대부분의 “실제 제품” 프로젝트에 딱 좋은 지점이다.
- **특정 GPU, 특수 시스템 소프트웨어, 또는 완전한 제어가 필요?** VM. 위쪽의 무언가가 정말로 그 일을 해낼 수 없을 때만 여기로 가라. 이제 당신이 시스템 관리자이기 때문이다.

유용한 규칙: **그 일을 해낼 수 있는 가장 단순한 옵션을 고르고, 진짜 벽에 부딪혔을 때만 한 단계 아래로 내려가라**—상상 속의 벽이 아니라.

“진짜 벽”은 실제로 어떻게 생겼을까? 솔직한 예 몇 가지: 당신의 빌드가 계속 떠 있어야 하는 서버를 만들어 낸다 (실시간 연결을 유지하는 채팅 백엔드), 그래서 정적으로는 안 된다. 또는 함수가 타임아웃에 계속 걸린다, 한 요청이 진짜 일을 하기 때문에 (영상 리사이징, PDF 생성), 그래서 컨테이너로 올라간다. 또는 무료 티어의 요청 한도가 진짜 트래픽에 의해 실제로 소진된다. 이것들의 공통점에 주목하라: 각각은 짐작이 아니라 측정이다. “언젠가 커질지도 몰라”는 벽이 아니다.

AI 가 배포와 설정을 엮는 법

여기서 바이브 코딩이 빛난다. 배포 설정은 정밀하고, 잘 문서화되어 있으며, 보일러플레이트가 많은—바로 AI 가 잘하는 종류의 일이다. 당신이 지시하고, AI 가 쓴다.

효과적인 프롬프트는 이렇게 들린다:

- “이 Node 앱에 Dockerfile 과, 그것을 배포할 Railway 설정을 추가해줘.”
- “main 에 푸시할 때마다 dist/ 폴더를 Cloudflare Pages 에 배포하는 GitHub Actions 워크플로를 작성해줘.”
- “서버리스 함수에서 콜드 스타트 타임아웃이 나고 있어—무엇이 문제일 가능성이 높고 설정을 어떻게 고치면 될까?”

여기 AI 가 당신을 위해 만들어 줄 종류의 파일이 있다—코드를 푸시할 때마다 정적 사이트를 자동으로 배포하는 배포 워크플로:

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci && npm run build
      - name: Publish to Cloudflare Pages
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${ secrets.CF_API_TOKEN }
          command: pages deploy dist --project-name=my-app
```

이 문법을 외울 필요는 없다. 그것이 무엇을 하는지 알아보면 된다: main 에 푸시할 때마다, 코드를 체크아웃하고, 빌드하고, 결과를 게시한다. 무언가 깨지면 에러를 AI 에게 다시 붙여넣고 설정을 고쳐달라고 요청하라.

많은 빌더는 처음엔 GitHub Actions 파일을 아예 건너뛰고 그냥 자기 터미널에서 명령 하나로 배포한다. 그게 첫날에 무언가를 라이브로 보기에 가장 빠른 길인 경우가 많다. 같은 Cloudflare Pages 배포를 명령줄에서 하면 이렇게 생겼다:

```
# install the CLI once
npm install -g wrangler

# build your site, then push the output folder live
npm run build
wrangler pages deploy dist --project-name=my-app
```

첫 실행은 브라우저로 로그인하라고 요청할 것이고, 그 후로는 명령 하나면 된다. 트레이드오프는 단순하다: 터미널 명령은 “지금 당장 출시”에 훌륭하고, GitHub Actions 파일은 “영원히 자동으로 출시”에 훌륭하다. 대부분의 프로젝트는 전자로 시작하고 배포가 일상이 되면 후자를 더한다. 당신이 어느 지점에 있는 맞는 것을 AI 에게 요청하라.

진짜 고통을 덜어주는 몇 가지 지시 요령:

- **AI 가 행동하기 전에 설명하게 하라.** “설정을 쓰기 전에, 이 프로젝트에 어떤 호스팅 유형이 맞고 왜 그런지 말해줘.” 이것은 과도한 엔지니어링을 일찍 잡아낸다.
- **비밀 (secret) 을 코드 밖에 뒀라.** API 키와 비밀번호는 플랫폼의 환경 변수 / 시크릿 저장소에 넣고, 커밋하는 파일에는 절대 넣지 마라. 위의 CF_API_TOKEN 처럼 시크릿 참조를 쓰도록 AI 에게 요청하라.
- **일찍 배포하고, 자주 배포하라.** 첫날에 “hello world”를 라이브로 올려라. 처음부터 작동해 온 파이프라인은 출시 시점에 급조한 것보다 디버깅하기 훨씬 쉽다.
- **롤백 시나리오를 미리 물어라.** “이 배포가 망가지면, 작동하던 버전으로 어떻게 되돌리지?” 좋은 플랫폼은 이전 배포들을 클릭 한 번 거리에 둔다. 되돌리기 버튼이 존재한다는 걸 필요해지기 전에 알아두면 새벽 2 시의 패닉이 어께 으쓱으로 바뀐다.

1 인 빌더를 위한 권장 기본 경로

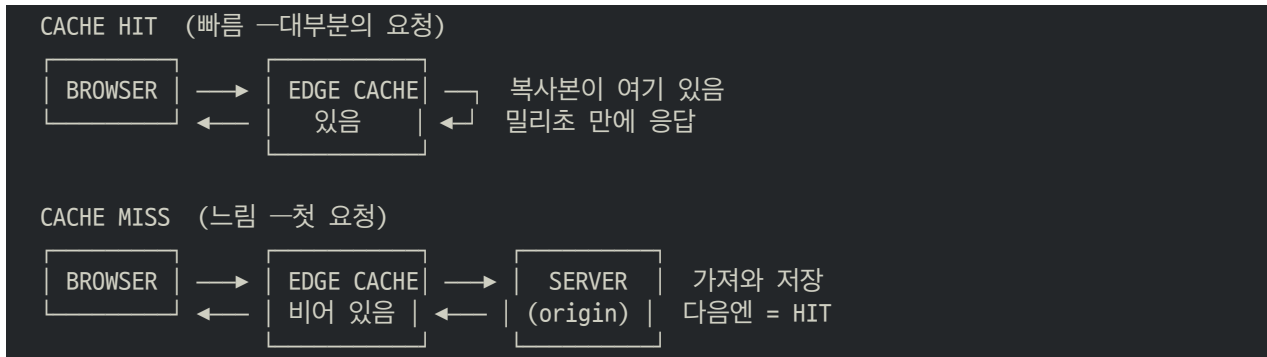
여기 수많은 1 인 프로젝트에서 통한 경로가 있다. 각 단계는 실제로 필요할 때만 밟는다.

1. **프로토타입: 정적 + 매니지드 백엔드 서비스.** 프론트엔드를 정적 호스팅 (Cloudflare Pages, Vercel, 또는 Netlify) 에 올려라. 데이터와 인증은 Supabase 나 Firebase 같은 매니지드 백엔드에 기대어 데이터베이스를 직접 돌리

는 일을 건너뛰어라. 비용: 사실상 0.

2. **초기 프로덕션: 엣지 또는 서버리스 함수 추가.** 서버 측 로직이 필요할 때—비밀 API 호출, 웹훅, 커스텀 인증—같은 플랫폼에 함수를 추가하라. 여전히 0 으로 스케일하고, 여전히 저렴하며, 여전히 한 번의 배포.
3. **진짜 제품: 백엔드를 컨테이너 플랫폼으로 옮겨라.** 앱이 함수의 한계를 넘어서면 (장시간 도는 프로세스, 영속적인 서버를 원하는 프레임워크, 백그라운드 작업) 컨테이너화해서 Railway, Render, 또는 Fly.io 에 배포하라. Dockerfile 과 플랫폼 설정은 AI 에게 쓰게 하라.
4. **스케일 또는 특수화: 그제서야 VM 이나 AWS 같은 클라우드 제공자를 고려하라.** 여기서 복잡도 (그리고 당신의 청구서) 가 켕충 쫘다. 대부분의 1 인 빌더는 이 단계가 결코 필요 없고, 그건 일을 제대로 하고 있다는 신호다—한계가 아니라.

CDN(Content Delivery Network) 은 이 경로 앞에 앉아, 이미 사본을 갖고 있는지에 따라 속도를 극적으로 바꾼다. 캐시 HIT 는 엣지에서 즉시 답하고, 캐시 MISS 는 먼저 당신의 서버까지 끝까지 다녀와야 한다:



피해야 할 실수는 “진짜 엔지니어”가 그걸 쓴다고들 해서 4 단계에서 시작하는 것이다. 진실은 그 반대다: 단순하게 시작하는 것이 시니어다운 수다. 더 많은 제어권으로 언제든 졸업할 수 있지만, 필요도 없던 인프라와 씨름하느라 잃을 몇 달은 쉽게 되찾을 수 없다.

“매니지드”가 진짜 의미하는 것 (그리고 당신이 여전히 소유하는 것)

매니지드 서비스는 플랫폼이 지루하고 위험한 부분—서버, 보안 패치, 스케일링, 백업—을 당신을 위해 처리하는 것이다. “서버리스”, “매니지드 데이터베이스”, “서비스로서의 플랫폼 (PaaS)”은 모두 같은 거래의 다른 맛이다: 당신은 운영을 넘기고 계속 만든다. 1 인 빌더에게 이것은 게으름이 아니라 지렛대다. Linux 보안 업데이트를 적용하지 않아 아낀 시간은 실제 제품에 쓰는 시간이다.

하지만 “매니지드”는 “마법”이 아니다. 플랫폼이 아무리 많이 처리해도 몇 가지는 여전히 당신의 책임으로 남는다:

- **당신의 데이터.** 플랫폼은 데이터베이스를 계속 돌려주지만, 실제로 복원을 테스트해 본 백업을 갖고 있는지는 당신 몫이다. “내보내기” 버튼을 한 번 눌러 파일이 비어 있지 않은지 확인하라.
- **당신의 시크릿.** API 키는 플랫폼의 시크릿 저장소에 살지만, 유출된 키를 교체하는 일과, 공개 저장소나 스크린샷에 절대 붙여넣지 않는 일은 당신의 일이다.
- **당신의 청구서.** 자동 스케일링은 당신의 청구액을 자동 스케일하기 전까지는 멋지다. 지출 알림과 상한을 설정하라—다음 절은 통째로 이것에 관한 것인데, 가장 아픈 실패이기 때문이다.
- **당신의 이해.** 자정에 무언가 깨질 때, “플랫폼이 처리한다”는 거기까지만 간다. 전문가일 필요는 없지만, 로그를 읽고 실패를 AI 에게 명확히 설명할 수 있어야 한다. 그게 “당신이 출시하는 것을 이해하라” 철학을 한 문장으로 담은 것이다.

건강한 마음가짐: 운영을 외주하되, 이해는 외주하지 마라. 플랫폼이 기계를 돌리게 하고, 당신은 그 기계가 무엇을 하는지 아는 사람으로 남아라.

빠른 출시 전 체크리스트

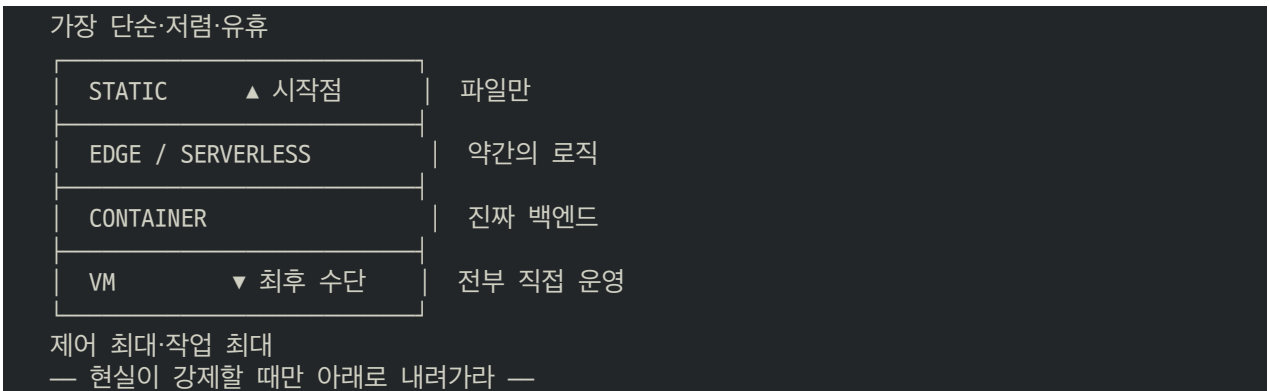
사람들에게 라이브 URL (Uniform Resource Locator —웹 주소) 을 방문하라고 말하기 전에, 이 목록을 따라가라. 어떤 것도 DevOps 전문성을 요구하지 않는다—대부분 AI 에게 보내는 프롬프트 하나거나 대시보드에서의 클릭 하나다.

- **실제로 로드된다.** 로컬 미리보기가 아니라 진짜 URL 을 새 브라우저 (또는 휴대폰) 에서 열어라. “내 컴퓨터에서는 됐는데”출시가 공개석상에서 404 를 내는 사례는 전설적으로 많다.
- **시크릿이 코드에 없다.** 푸시하기 전에 저장소에서 자신의 키를 검색하라. AI 에게 물어라: “이 프로젝트에서 하드코딩된 API 키, 비밀번호, 토큰을 스캔해줘.”
- **지출 상한이 설정돼 있다.** 모든 유료 API 에 하드 상한, 그리고 호스팅 계정에 결제 알림. 상한부터 걸고, 자축은 나중에.
- **HTTPS 가 켜져 있다.** HTTPS(HyperText Transfer Protocol Secure) 는 웹의 기본 전송 방식을 암호화한 버전으로, 주소창에 자물쇠를 띄워 주는 것이다. 현대 플랫폼은 이걸 자동으로 하지만, 작은 자물쇠 아이콘이 거기 있는지 확인하라. 브라우저는 암호화되지 않은 사이트에 벌을 준다.
- **롤백하는 법을 안다.** 아무것도 불타고 있지 않은 지금, 이전 배포 버튼을 찾아두라.
- **당신에게 닿을 길이 있다.** 기본적인 것이라도 로깅이나 에러 리포팅 도구를 뒤서, 문제의 첫 신호가 화난 사용자가 되지 않게 하라. 당신의 스택에 에러 로깅을 붙여달라고 AI 에게 요청하라.

이것은 포괄적인 프로덕션 감사가 아니다—창피하고, 비싸고, 쉽게 피할 수 있는 실수를 잡아내는 짧은 목록이다. 반사적이 될 때까지 매 출시마다 돌려라.

핵심 정리

인프라는 도약이 아니라 사다리다. 꼭대기에 정적, 바닥에 VM, 그리고 당신 작업의 대부분은 위쪽 가로대에 행복하게 산다.



다섯 범주를 알고, 그 일을 해내는 가장 단순한 것을 고르고, 설정은 AI 에게 쓰게 하고, 무언가를 가능한 한 일찍 라이브로 올려라. 나머지는 그저—현실이 강제할 때, 오직 그럴 때만—한 가로대 아래로 내려가는 일일 뿐이다.

정리 & 연습

핵심 요점

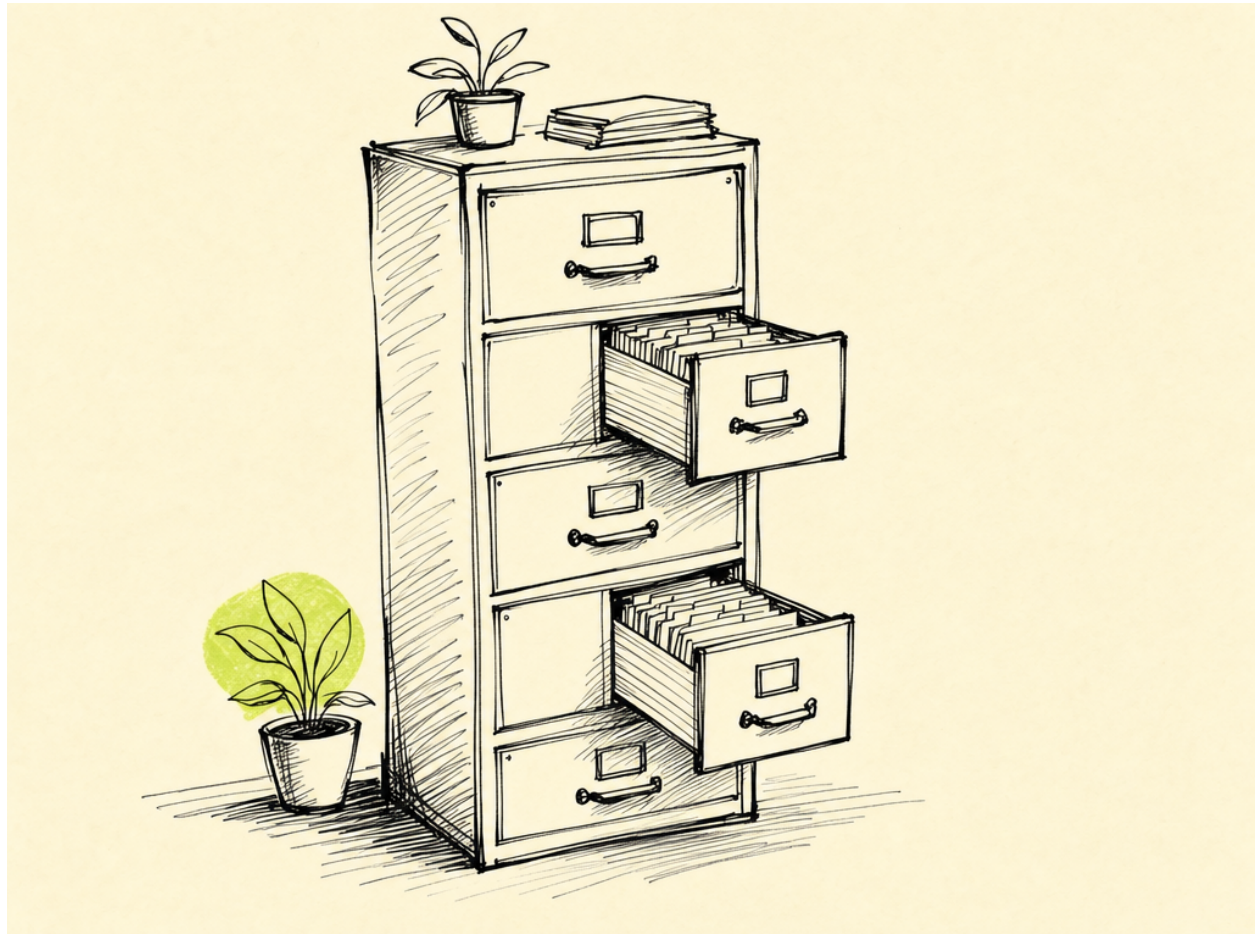
- 인프라는 다섯 단계의 사다리다—정적 호스팅, 서버리스, PaaS, 매니지드 컨테이너, 그리고 순수 VM —그리고 일을 해내는 가장 단순한 단계를 골라야 한다.
- **매니지드** 서비스를 우선하라: 운영 (서버, 패치, 스케일링, 백업) 은 맡기되 이해는 직접 챙겨라.
- AI 에게 배포 설정과 CI(Continuous Integration —코드를 푸시할 때마다 점검을 자동으로 돌려 주는 로봇) 워크플로를 쓰게 하되, 믿기 전에 먼저 읽어라.
- “매니지드”는 “마법”이 아니다—데이터, 시크릿, 청구서, 그리고 이해는 여전히 당신 책임이다.
- 출시 전에 지출 상한을 걸고, HTTPS 를 확인하고, 롤백하는 법을 알아 두어라.

직접 해보기

이미 가진 작은 프로젝트 (HTML 한 페이지라도 좋다) 를 정적 호스트에 배포해 보라. 그런 다음 AI 에게 push 때마다 재배포하는 GitHub Actions 워크플로를 만들어 달라고 하고, 생성된 YAML 을 한 줄씩 읽고, 호스트 대시보드에서 “이전 배포로 롤백”버튼을 찾을 수 있는지 확인하라.

```
I have a static site in this repo. Walk me through deploying it to a free static host, then generate a GitHub Actions workflow that rebuilds and redeploys on every push to main. Explain each step of the YAML, and tell me where in the dashboard I roll back a bad deploy.
```

데이터베이스 인프라



진짜 앱은 결국 무언가를 기억해야 한다—사용자, 주문, 메시지, 설정. 그 기억은 데이터베이스에 산다. 초보자에게 “데이터베이스”는 전문가들이 지키는 블랙박스처럼 느껴질 수 있다. 그렇지 않다. 이 장은 데이터 계층의 신비를 벗겨내, 당신이 좋은 선택을 내리고 무거운 작업은 AI에게 안전하게 맡길 수 있도록 돕는다.

일찍 체화해야 할 것: 코드는 대체 가능하지만 데이터는 보통 그렇지 않다. 버그 있는 함수는 오후 한나절이면 다시 쓸 수 있지만, 지난 화요일에 손상시킨 컬럼—혹은 한 번도 받지 않은 백업—은 영영 사라질 수 있다. 바로 이 비대칭성 때문에, 데이터 계층은 스택의 거의 어떤 부분보다도 약간의 선행 이해에 보답한다.

데이터 저장의 세 가지 형태

대부분의 데이터베이스는 세 가지 계열에 속한다. 제품을 외울 필요는 없다—어떤 형태가 당신의 문제에 맞는지 알아보면 된다.

유형	예시	적합한 용도	트레이드오프
관계형 (SQL)	PostgreSQL, MySQL, SQLite	관계가 있는 구조화된 데이터; 돈이나 정확성이 중요한 모든 것	스키마를 미리 설계해야 한다; 쓰기를 아주 크게 확장하기는 더 어렵다
문서형 / NoSQL	MongoDB, Firestore, DynamoDB	유연하거나 중첩된 데이터, 빠른 반복, 다양한 레코드 형태	일관성 없는 데이터를 만들기 쉽다; 관계와 리포팅이 어색해진다
키-값	Redis, Memcached, Cloudflare KV	캐싱, 세션, 카운터, 알려진 키로 빠른 조회	내용으로 쿼리할 수 없다; 보통 신뢰의 원천 (source of truth) 은 아니다

세 가지 형태는 데이터를 진짜로 다르게 저장한다. 나란히 그려 보면 이렇다:

관계형 (SQL)

행 + 열

id	email	name
1	a@...	Ana
2	b@...	Bob

테이블이 서로를 참조한다

문서형 (NoSQL)

독립된 데이터 덩어리

```
{
  id, email,
  name,
  posts: [ ... ]
}
```

모두 중첩되어 있다

키-값

키 1 개 → 값 1 개

key	value
session count	"x9f2" "42"

내부를 쿼리 못 함 — 키로만 조회한다

관계형 데이터베이스는 서로를 참조할 줄 아는 스프레드시트 묶음이라고 생각하라. 그것과 대화할 때 쓰는 언어가 SQL(Structured Query Language, 구조적 질의 언어) 이다. 프로젝트의 90% 에 적합한 기본값: **관계형 데이터베이스 (Postgres) 로 시작하라**. 성숙하고 예측 가능하며, 구조를 강제하고, 진짜 앱이 필연적으로 키워나가는 관계를 잘 다룬다. 데이터가 정말로 형태가 없을 때 문서형 저장소를 꺼내고, 키-값 저장소는 보조 계층 (캐싱) 으로 쓰되 주 저장소로는 쓰지 말라.

피해야 할 함정은 분위기로 고르는 것이다. “NoSQL(SQL 테이블을 쓰지 않는 데이터베이스—유연하고 자유로운 형태의 레코드를 저장한다) 이 확장성이 더 좋다”는 말은 대부분의 앱이 결코 닿지 않는 좁은 문제 집합에서만 참이고, 그조차도 데이터를 신뢰할 수 있게 만드는 일관성 보장을 포기하는 대가를 치른다. 시스템 설계 영상에서 가장 인상적으로 들리는 형태가 아니라, 당신의 데이터에 맞는 형태를 골라라. 확신이 서지 않으면 관계형이 안전한 선택이다—캐시나 문서형 저장소는 나중에 언제든지 더할 수 있지만, 엉킨 NoSQL 데이터 모델을 풀어내는 일은 고통스럽다.

매니지드 vs 셀프 호스팅

데이터베이스를 직접 서버에서 운영할 수도 있고, 공급자가 백업·패치·가동시간을 책임지는 **매니지드 서비스** (Supabase, Neon, PlanetScale, RDS, Cloudflare D1) 를 빌릴 수도 있다.

바이브 코딩에서는 **항상 매니지드로 시작하라**. 데이터베이스를 셀프 호스팅한다는 것은 이제 백업, 보안 업데이트, 디스크 공간, 그리고 새벽 3 시의 복구를 당신이 책임진다는 뜻이다. 그것은 빌드에서 주의를 빼앗는 진짜 운영 업무다. 매니지드 서비스는 약간의 비용이 들지만 엄청난 시간과 위험을 줄여준다. 셀프 호스팅으로 졸업하는 것은 구체적인 이유와 그것을 뒷받침할 기술이 있을 때만 하라.

매니지드 공급자를 고를 때는 실제로 무엇에 돈을 내는지 아는 게 좋다. 마케팅 페이지는 다 비슷해 보이기 때문이다. 진짜로 다른 점 몇 가지가 있다:

- **연결 모델.** 서버리스 플랫폼 (Cloudflare Workers, Vercel functions) 은 데이터베이스 연결을 끊임없이 열고 닫는데, 이는 고전적 Postgres 연결 풀을 고갈시킬 수 있다. Neon 이나 Supabase 같은 공급자는 바로 이를 위해 연결 풀러 (PgBouncer 또는 동등물) 를 제공한다—서버리스 코드에서는 풀링된 연결 문자열을 써라.
- **브랜칭과 백업.** Neon 과 PlanetScale 은 Git 처럼 데이터베이스를 “브랜치”할 수 있게 해, 마이그레이션을 사본에서 테스트할 수 있다. 시점 복구 (point-in-time recovery) 가 켜져 있는지 확인하고, 얼마나 과거까지 거슬러 가는지 알아두라.

- **중속 (lock-in).** D1 과 DynamoDB 는 훌륭하지만 하나의 클라우드에 당신을 묶는다. 일반 Postgres(Supabase, Neon, RDS) 는 이식성이 있다—당신의 스키마와 쿼리는 어떤 Postgres 호스트로도 옮겨간다. 대부분의 초보자에게는 이식 가능한 쪽이 장기적으로 더 안전한 선택이다.

스키마 설계 기초

스키마는 당신 데이터의 청사진이다: 어떤 테이블이 존재하고, 어떤 컬럼을 가지며, 어떻게 연관되는지. 좋은 스키마 설계는 대부분 상식에 몇 가지 습관을 더한 것이다.

- “**것**”**하나당 테이블 하나.** 사용자, 게시물, 주문—각각 자기 테이블을 갖는다.
- **모든 행에는 고유 ID 가 필요하다** (기본 키).
- **테이블은 외래 키로 연결하라.** 데이터를 복사해 다니지 말고.
- **정직한 타입을 골라라.** 돈은 float 가 아니라 decimal 이다. 날짜는 텍스트가 아니라 타임스탬프다.
- **중복하지 말라.** 사용자의 이메일이 다섯 테이블에 살면, 결국 다섯 개의 다른 이메일을 갖게 된다.

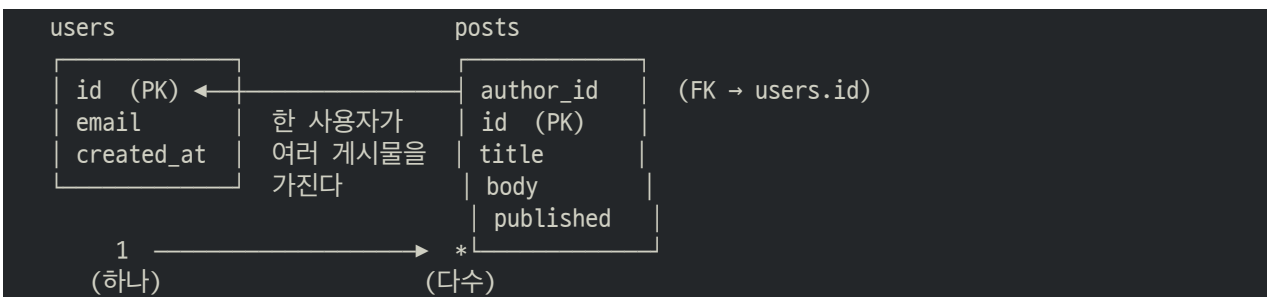
다음은 블로그를 위한 간단한 스키마로, AI 가 당신을 위해 생성해줄 종류다:

```
CREATE TABLE users (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email       TEXT NOT NULL UNIQUE,
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE posts (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  author_id   UUID NOT NULL REFERENCES users(id),
  title       TEXT NOT NULL,
  body        TEXT NOT NULL,
  published   BOOLEAN NOT NULL DEFAULT false,
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE INDEX idx_posts_author ON posts(author_id);
```

author_id ... REFERENCES users(id) 줄이 외래 키다. 각 게시물을 정확히 한 명의 실제 사용자에 엮어, 데이터베이스 스스로가 유효한 작성자 없는 게시물 저장을 거부한다:



REFERENCES(게시물을 실제 사용자에 묶는 외래 키) 와 NOT NULL, UNIQUE 제약에 주목하라. 이것들은 데이터베이스가 당신을 위해 강제하는 가드레일이라, 당신 코드에 버그가 있어도 잘못된 데이터가 몰래 들어올 수 없다.

의도적으로 추가할 가치가 있는 제약은 부모가 사라질 때 무슨 일이 일어나느냐다. 기본적으로, 게시물이 여전히 참조하는 사용자를 삭제하려 하면 실패한다—종종 그것이 당신이 원하는 바다. 하지만 명시적으로 정할 수 있다: REFERENCES users(id) ON DELETE CASCADE 는 게시물도 함께 삭제하고, ON DELETE RESTRICT 는 삭제를 막는다. 의도를 가지고 고르는 것이, 프로덕션에서 기본값을 호되게 깨닫는 것보다 낫다. 똑같은 “데이터베이스가 당신을 지키게 하라”는 본능은 CHECK 제약 (CHECK (price >= 0)) 에도 적용된다—모든 코드 경로가 검증하리라 믿는 대신, 말도 안 되는 데이터를 원천에서 거부한다.

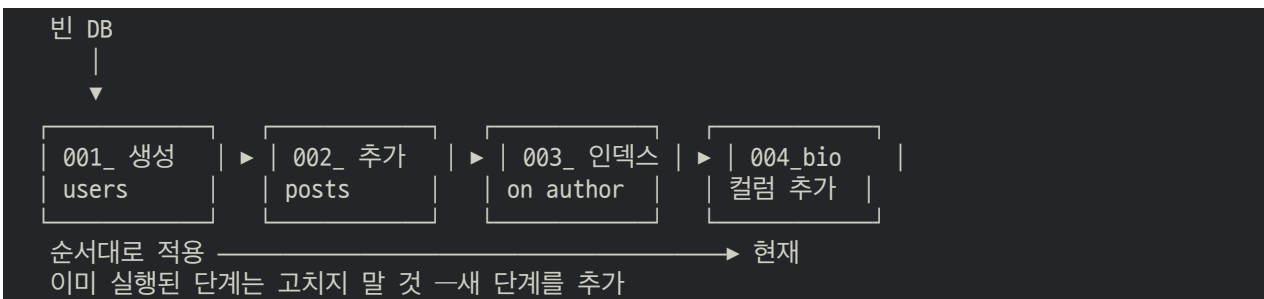
마이그레이션: 스키마를 안전하게 바꾸기

당신의 스키마는 변한다—컬럼을 추가하고, 테이블 이름을 바꾸고, 인덱스를 추가하게 된다. 공유 문서의 변경 이력 추적처럼 생각하라: 모든 변경이 순서대로 기록되어, 누구의 사본이든 같은 방식으로 최신 상태로 맞출 수 있다. **마이그레이션**은 그런 변경 하나를 기술하는 버전 관리되고 반복 가능한 스크립트다. 마이그레이션이 중요한 이유는, 당신의 노트북, 동료의 노트북, 그리고 프로덕션의 스키마가 동일하게 유지되어야 하기 때문이다.

마이그레이션을 안전하게 지키는 규칙:

- **프로덕션에서 데이터베이스를 손으로 절대 수정하지 말라.** 마이그레이션을 작성하라.
- **마이그레이션은 앞으로 나아간다.** 각각이 새 파일이며, 이미 실행된 옛것을 다시 쓰지 않는다.
- **파괴적 변경은 신중히 하라.** 컬럼이나 테이블을 드롭하면 데이터가 영구히 삭제된다. 먼저 백업을 가져라.
- **추가하는 변경이 가장 안전하다.** nullable 컬럼을 추가하는 것은 거의 아무것도 깨뜨리지 않지만, 이름을 바꾸거나 제거하는 것은 실행 중인 코드를 깨뜨릴 수 있다.

마이그레이션은 순서가 있는 사술을 이룬다: 각각은 번호 매겨진 단계로, 데이터베이스의 모든 사본을 한 알려진 상태에서 다음 상태로 옮긴다. 그래서 당신의 노트북, 동료의 노트북, 프로덕션이 모두 동일하게 끝난다:



Prisma, Drizzle, 또는 Rails 마이그레이션 같은 도구가 이것들을 생성하고 추적해준다. 도구에게 맡겨라.

마이그레이션은 두 부분으로 이루어진다: 적용할 변경 (up) 과 그것을 되돌리는 법 (down). 신뢰할 만한 도구는 둘 다 생성해주어, 잘못된 배포가 깔끔하게 롤백될 수 있다:

```
-- up: 선택적 bio 추가, 라이브에서 적용해도 안전
ALTER TABLE users ADD COLUMN bio TEXT;

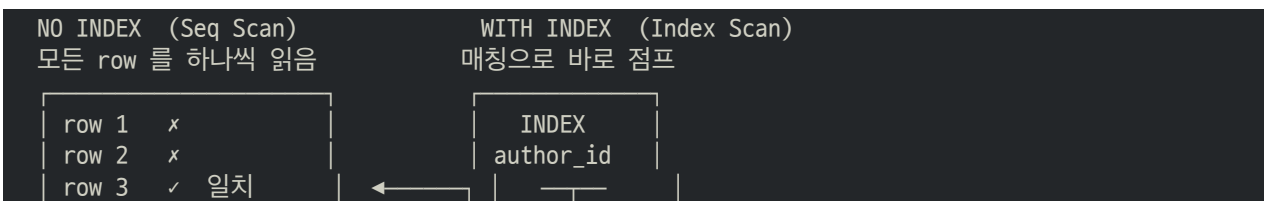
-- down: 되돌리기
ALTER TABLE users DROP COLUMN bio;
```

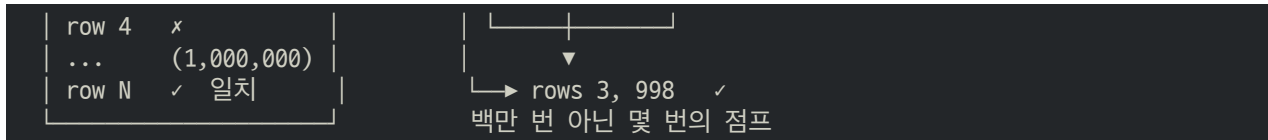
정말로 위험한 마이그레이션은 이름 변경과 타입 변경인데, 옛 코드와 새 스키마가 잠시 어긋나기 때문이다. 프로페셔널한 패턴은 그것들을 추가적 단계로 쪼개는 것이다: 새 컬럼을 추가하고, 백필하고, 코드가 그것을 읽도록 전환한 다음, 아무것도 참조하지 않게 된 나중의 배포에서야 옛 컬럼을 드롭한다. 느리게 느껴지지만, 매끄러운 릴리스와 배포 도중 모든 사용자에게 터지는 500 에러 급증 사이의 차이다.

인덱싱

인덱스는 책 뒤의 색인과 같다: 데이터베이스가 모든 행을 스캔하는 대신 일치하는 행으로 곧장 점프하게 해준다. 인덱스가 없으면 데이터가 늘수록 쿼리가 느려진다—100 행에서는 괜찮지만 1,000,000 행에서는 고통스럽다.

인덱스가 없으면 데이터베이스는 모든 행을 차례로 확인하고, 있으면 일치하는 곳으로 곧장 점프한다—책 색인이 처음부터 끝까지 읽는 대신 맞는 페이지로 보내주는 것과 같다:





실용적인 지침:

- 자주 **필터**하거나 **조인**하는 컬럼에 인덱스를 걸어라 (예: author_id, email).
- 기본 키는 자동으로 인덱싱된다.
- **모든 것에 인덱스를 걸지 말라**—인덱스 하나하나가 읽기는 빠르게 하지만 쓰기는 느리게 하고 공간을 쓴다.
- 모든 컬럼에 선제적으로 거는 대신, 느린 쿼리를 봤을 때 인덱스를 추가하라.

인덱스가 도움이 되는지 알아내는 정직한 방법은 추측이 아니라 데이터베이스에게 물어보는 것이다. 모든 SQL 엔진에는 사용할 계획을 보여주는 EXPLAIN(Postgres에서는 EXPLAIN ANALYZE)이 있다:

```
EXPLAIN ANALYZE
SELECT * FROM posts WHERE author_id = '...';
```

큰 테이블에서 출력이 Seq Scan 이라고 한다면, 데이터베이스가 모든 행을 읽고 있다는 뜻이다—인덱스가 도움이 되리라는 신호다. 하나를 추가하면 Index Scan 으로 바뀌어야 한다. 이것은 AI 에게 넘기기 좋은 작업이다: 느린 쿼리와 EXPLAIN 출력을 붙여 넣고, 어떤 인덱스를 추가해야 하는지와 그 이유를 물어라. 당신은 인덱스 이론을 외워서가 아니라 답을 이해함으로써 통제를 유지한다.

캐싱

캐시는 비싼 질문에 대한 답을 저장해서 다시 묻지 않아도 되게 한다. 홈페이지가 방문자마다 같은 무거운 쿼리를 돌린다면, 그 결과를 Redis 에 60 초 동안 캐시하고 즉시 제공하라.

캐싱은 성능 최적화이지 신뢰의 원천이 아니다. 초보자의 고전적 함정은 **캐시 무효화**를 잇는 것이다—캐시된 사본이 낡아져 사용자가 옛 데이터를 본다. 경험 법칙: 진짜 병목을 측정했을 때만 캐시하고, 캐시된 값이 언제 만료되거나 갱신되는지에 대한 명확한 규칙을 항상 가져라.

깔끔한 전략은 두 가지인데, 이 둘을 헷갈리는 것이 대부분의 캐시 버그를 일으킨다. **시간 기반 만료 (TTL —Time To Live, 캐시된 값에 찍힌 유통기한 같은 것)**는 “이 값은 60 초 동안 유효하고, 그다음엔 버려라”라고 말한다—단순하며, 약간 낡은 데이터가 무해할 때 (조회수, 홈페이지 피드) 괜찮다. **이벤트 기반 무효화**는 “기반 데이터가 바뀌는 순간 캐시된 값을 삭제하라”라고 말한다—정확하지만, 그것을 모든 쓰기 경로에서 기억해야 하고, 바로 거기서 사람들이 미끄러진다. 확신이 안 서면 짧은 TTL 을 선호하라: 길어야 1 분 낡은 값은, 무효화 하나를 잊어서 영원히 낡은 값보다 작은 버그다.

큐

큐는 “지금 해”대신 “이거 나중에 해”라고 말할 수 있게 해준다. 사용자가 가입할 때, 환영 이메일을 보내고 아바타 크기를 조정하는 동안 그들이 기다리길 원하지 않을 것이다. 그런 작업을 큐 (RabbitMQ, BullMQ, Cloudflare Queues, SQS) 에 밀어 넣으면 백그라운드 워커가 집어간다. 사용자는 즉각적인 응답을 받는다.

느리거나, 실패하고 재시도할 수 있거나, 사용자에게 응답하기 전에 끝나지 않아도 되는 모든 작업에 큐를 사용하라.

대부분의 초보자가 부딪히는 함정은, 큐에 넣은 작업이 한 번 이상 실행될 수 있다는 것이다—네트워크 끊김으로 워커가 재시도하고, 이제 환영 이메일이 두 번 나간다. 해결책은 작업을 **멥등적 (idempotent)** 으로 만드는 것이다: 같은 작업을 두 번 실행해도 한 번 실행한 것과 같은 효과가 나도록 설계하라 (보내기 전에 “이미 보냈나?”를 확인). 또한 계속 실패하는 작업을 어떻게 할지 미리 정하라; 성숙한 큐는 그것을 데드레터 큐로 보내, 독이 든 메시지 하나가 뒤에 줄 선 모든 것을 막지 않게 한다.

흔한 초보자 실수

- **현대적으로 들린다는 이유로 NoSQL 을 집어드는 것.** 대부분의 앱에는 관계가 있고, SQL 이 그것을 더 잘 다룬다.
- **돈을 부동소수점으로 저장하는 것.** 0.1 + 0.2 는 유명하게도 0.3 이 아니다. decimal 타입을 써라.
- **백업 없음.** 매니지드 서비스는 보통 자동으로 백업한다—가정하지 말고 확인하라.

- **비밀이나 거대한 파일을 데이터베이스에 넣는 것.** 파일은 오브젝트 스토리지 (S3, R2) 에 속하고, 데이터베이스는 그 링크를 저장한다.
- **AI 가 사용자 입력을 붙여 넣은 원시 쿼리를 쓰게 두는 것.** 이것이 SQL 인젝션을 일으킨다. 파라미터화된 쿼리를 써라.
- **N+1 쿼리.** 항목마다 쿼리 하나를 돌리는 루프 (게시물 100 개를 가져온 다음, 작성자 조회 100 번을 따로) 는 데이터베이스를 두들긴다. 관련 데이터를 조인으로 한 쿼리에 가져오거나, 조회를 배치로 묶어라.
- **부하 테스트를 절대 안 하는 것.** 당신 노트북의 50 행에서 즉각적인 쿼리가, 프로덕션의 500 만 행에서는 기어갈 수 있다. 성능을 신뢰하기 전에 현실적인 양의 가짜 데이터를 시드하라.

데이터 계층 구축을 AI 에게 지시하기

AI 는 스키마 설계와 쿼리 작성에 탁월하다—단, 설계할 제약을 줄 때 그렇다. “데이터베이스 만들어”라고 말하지 말라. 도메인, 관계, 그리고 타협 불가능한 것들을 묘사하라.

강력한 프롬프트는 이렇게 생겼다:

Design a PostgreSQL schema for a task-management app.

Entities and relationships:

- A user can own many projects.
- A project has many tasks.
- A task belongs to one project, has a status (todo / doing / done), and an optional due date.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL, plus a forward migration file.
- Explain any trade-offs you made.

그런 다음 이 장의 체크리스트로 **출력을 검토하라**: 정직한 타입, 외래 키와 제약이 존재하는지, 합리적인 인덱스, 중복 데이터 없음. 쿼리를 요청할 때는 사용자 입력이 SQL 에 절대 연결되지 않도록 **파라미터화된 쿼리**를 명시적으로 요구하라. 그리고 무언가를 드롭하거나 이름을 바꾸는 마이그레이션이라면, 먼저 백업할 수 있도록 AI 에게 그것을 짚어달라고 요청하라.

데이터베이스는 당신 앱에서 실수를 되돌리기가 가장 어려운 부분이다—데이터는 한번 손상되거나 잃으면 항상 돌아오지는 않는다. 바로 그래서 이 기본기를 알아둘 가치가 있다: 모든 줄을 직접 쓰기 위해서가 아니라, AI 를 자신 있게 지시하고 위험한 수가 출시되기 전에 잡아내기 위해서다.

정리 & 연습

핵심 요약

- 대부분의 앱은 세 가지 저장 형태—관계형, 문서형, 키-값형—중 하나만 필요하며, 관계형이 안전한 기본값이다.
- 매니지드 데이터베이스로 시작하라. 데이터와 스키마 결정은 당신이, 서버 운영은 플랫폼이 책임진다.
- 좋은 스키마 설계란 정직한 타입, 외래 키와 제약 조건, 합리적인 인덱스, 그리고 중복 없는 데이터를 뜻한다.
- 모든 마이그레이션을 일방통행 문으로 다뤄라: 무언가를 drop 하거나 이름을 바꾸기 전에 백업하고, AI 가 쓴 마이그레이션은 꼼꼼히 검토하라.
- 항상 파라미터화된 쿼리를 요구해 사용자 입력이 SQL 에 절대 그대로 이어 붙지 않게 하라.

직접 해보기

작은 앱 아이디어를 하나 고르고, 그 엔티티와 관계를 먼저 평범한 한국어로 적어 보라. 그런 다음 그 설명을 AI 에게 넘겨 스키마와 정방향 마이그레이션을 요청하고—무엇이든 실행하기 전에 이 장의 체크리스트로 결과를 검토하라.

Design a relational schema for a personal bookmarks app.

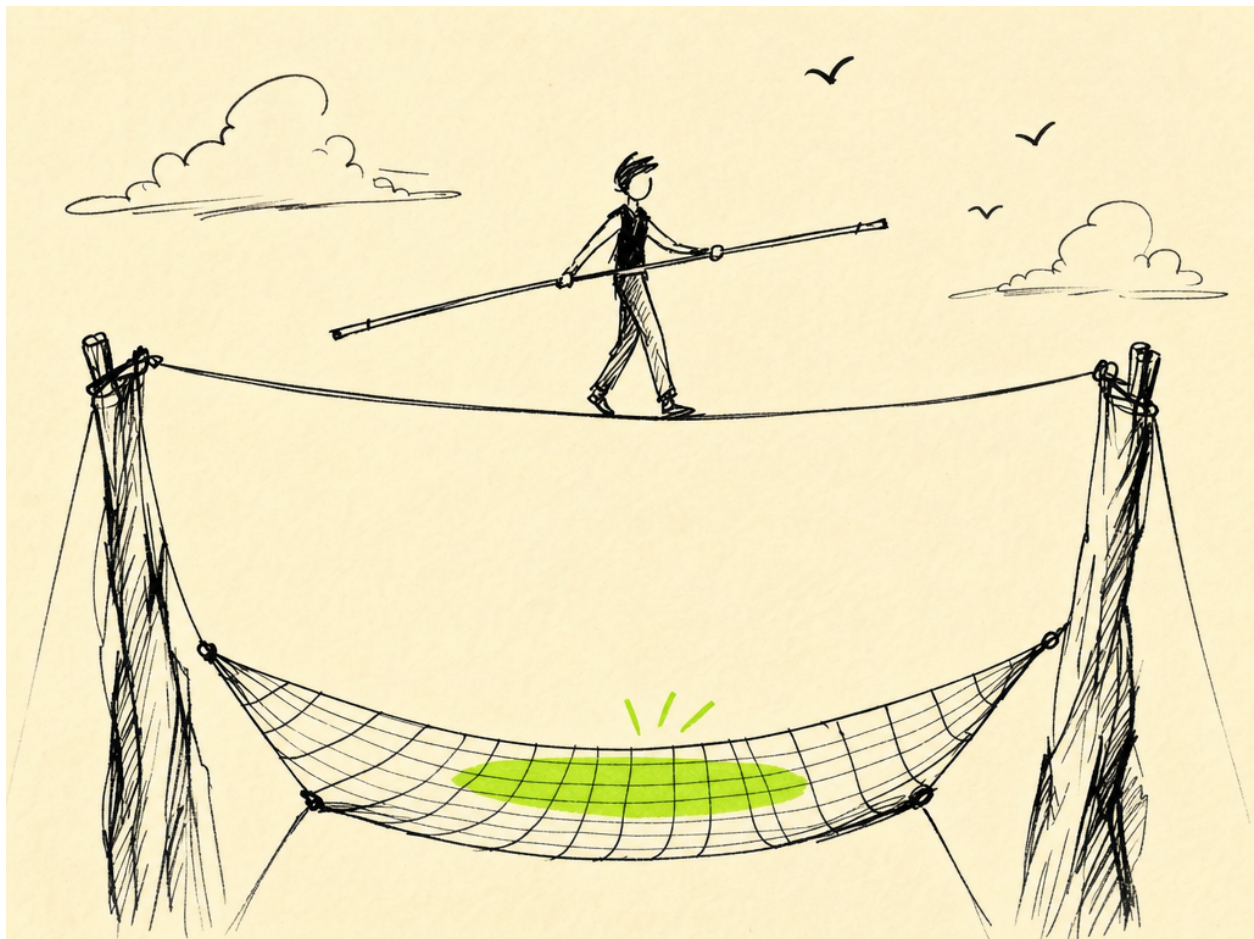
Entities and relationships:

- A user has many bookmarks.
- A bookmark belongs to one user, has a URL, a title, and many tags.
- A tag can apply to many bookmarks.

Requirements:

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL plus a forward migration file.
- Flag any migration step that drops or renames so I can back up first.

가드레일, 테스트 & 리뷰



바이브 코딩은 빠르게 움직이게 해준다. 위험은 당신이 잘못된 방향으로도 빠르게 움직인다는 것이다—그리고 AI가 자신만만하게 들리기 때문에, 진짜 사용자 앞에서 무언가 깨질 때까지 눈치채지 못할 수 있다. 가드레일은 충돌 없이 속도를 유지하는 방법이다.

이 장은 당신 프로젝트를 관료 조직으로 바꾸는 이야기가 아니다. AI의 실수 (그리고 당신의 실수)가 프로덕션에 도달하기 전에 잡아내는, 값싸고 자동적인 몇 가지 점검에 관한 것이다. 한 번 설정해두면 영원히 백그라운드에서 작동한다.

이 장의 철학 전체는 한 줄로 요약된다: **이해하지 못한 것은 절대 출시하지 말라.** 가드레일은 “이해하기”를 매년 실제로 할 만큼 충분히 값싸게 만들어주는 것이다.

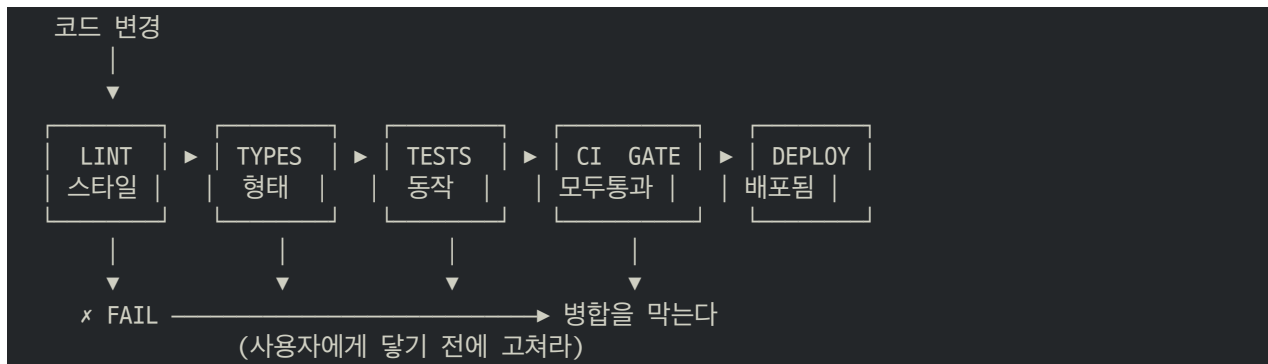
AI 를 지시할 때 가드레일이 필요한 이유

AI 는 당신 앱이 무엇을 해야 하는지 모른다. 그럴듯해 보이는 것에 패턴 매칭할 뿐이다. 그 말은 AI 가 기꺼이 다음을 한다는 뜻이다:

- 이해하지 못하는 점검을 “에러를 고치려고” 삭제한다
- 한 파일을 바꾸라고 했더니 세 파일을 리팩터링한다
- 정상 경로에서는 드러나지 않는 미묘한 버그를 들여온다
- 코드를 한 번도 실행하지 않았으면서 끝났다고 자신 있게 말한다

가드레일은 “날 믿어”를 “증명해봐”로 바꾼다. 목표는 실수로 무언가를 깨뜨리기가 어려운 시스템이라, 숨을 죽이지 않고도 계속 “그냥 해봐”라고 말할 수 있는 것이다.

가드레일은 모든 변경이 진짜 사용자에게 닿기 전에 통과해야 하는 파이프라인이라고 생각하라. 각 관문은 값싸고 자동적이며, 그중 어느 하나라도 나쁜 변경을 단번에 멈출 수 있다:

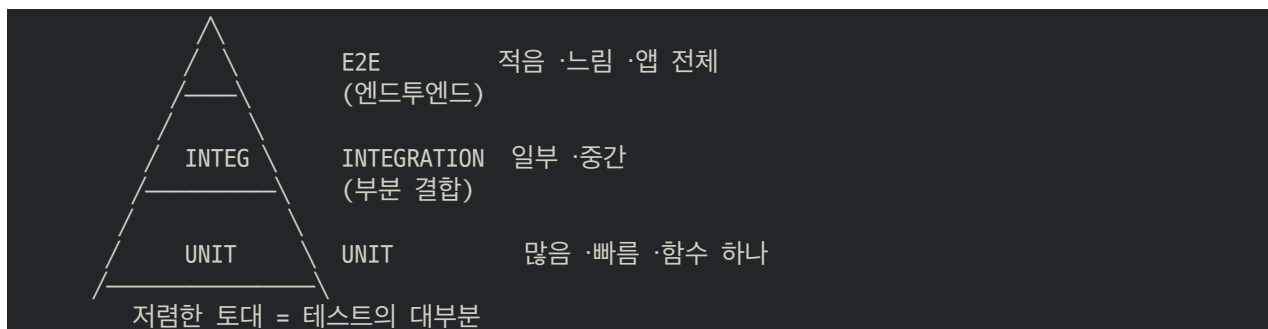


이것이 인간 동료보다 AI 에게 더 중요한 더 깊은 이유가 있다. 검증 점검을 삭제하는 인간은 보통 자기가 위험을 감수하고 있음을 알고, 그 무게를 느낀다. AI 는 아무것도 느끼지 않는다. AI 는 가드를 제거하고, 파일을 다시 쓰고, 성공을 보고한다—그 변경이 훌륭한 재앙이든 똑같이 명량한 어조로. 당신은 AI 의 몸짓을 읽을 수 없는데, 몸짓이라는 게 없기 때문이다. 가드레일이 바로 그 몸짓이다—작업이 실제로 건전한지에 대해 당신이 얻는 유일하게 정직한 신호다.

테스트: 당신의 안전망

테스트는 다른 코드가 당신이 기대하는 대로 동작하는지 점검하는 코드다. 한번 작성되면 몇 초 만에 돌아가며 무언가 퇴행했는지 즉시 알려준다.

건강한 테스트 묶음은 피라미드처럼 생겼다: 바닥에 작고 빠른 단위 테스트가 많이, 가운데에 통합 테스트가 좀 더 적게, 꼭대기에 느린 종단 간 테스트가 한 줄만. 안전의 대부분은 바닥에서 값싸게 온다:



100% 커버리지가 필요하지는 않다. 깨졌을 때 가장 아플 것들부터 테스트하라: 로그인, 결제, 재생성할 수 없는 데이터, 핵심 비즈니스 로직. AI 는 테스트 작성에 탁월하다—기대 동작이 구체적이기 때문에 종종 기능 작성보다 더 낫다.

다음은 잘 작동하는 프롬프트다:

src/pricing.js의 calculateDiscount 함수에 대한 테스트를 작성하라.

다음은 다뤄라: 정상 케이스, 0/빈 케이스, 최대 할인
경계, 그리고 잘못된 입력 하나. 이 저장소의 기존 테스트 설정을
사용하라 (다른 *.test.js 파일들이 어떻게 구성됐는지 확인하라).
끝내기 전에 테스트를 실행해서 통과하는 것을 내게 보여라.

그 마지막 줄이 중요하다. AI에게 테스트를 작성만 하지 말고 항상 실행하라고 요청하라. 한 번도 실행되지 않은 테스트
는 추측이다. 어떤 케이스를 다룰지 구체적으로 말하는 것도 가치가 있는데, 내버려두면 AI는 정상 경로의 변수 세 개를
써놓고 철저하다고 부르는 경향이 있기 때문이다. 실제로 버그를 잡는 케이스는 지루한 것들이다: 빈 입력, 경계값, 던져
야 마땅한 것.

AI가 만들어내야 할 간단한 예시:

```
import { calculateDiscount } from "../pricing.js";
import { describe, it, expect } from "vitest";

describe("calculateDiscount", () => {
  it("applies a normal percentage discount", () => {
    expect(calculateDiscount(100, 0.2)).toBe(80);
  });

  it("returns the full price when discount is zero", () => {
    expect(calculateDiscount(100, 0)).toBe(100);
  });

  it("never discounts below zero", () => {
    expect(calculateDiscount(100, 1.5)).toBe(0);
  });

  it("rejects a negative price", () => {
    expect(() => calculateDiscount(-10, 0.2)).toThrow();
  });
});
```

나중에 기능을 바꿀 때는 테스트를 먼저 돌려라. 빨간불이 들어오면, 사용자보다 먼저 당신이 그 깨짐을 찾은 것이다. AI
가 버그를 고칠 때는, 옛 동작에서 실패하는 테스트를 추가하라고 요청하라—그래야 버그가 조용히 다시 돌아올 수 없다.
이 습관에는 이름이 있다: 회귀 테스트 (regression test). 당신이 마주친 모든 버그는 코드베이스가 대가를 치르고 얻은
교훈이다; 회귀 테스트는 그 교훈을 지키는 방법이다.

한 가지 경고: 테스트도 틀릴 수 있다. AI가 버그가 있는 동작이 올바르다고 단언하는 테스트를 쓰면, 그 테스트는 영원
히 통과하면서 당신 앱은 깨진 채로 남는다. 그러니 초록 체크 표시만이 아니라 단언 (assertion)을 읽어라. 통과하는 테
스트는 “코드가 테스트가 말하는 대로 동작한다”만 의미한다—그 테스트가 옳은 것을 말하는지는 여전히 당신이 확인해
야 한다.

타입 체크와 린팅: 공짜로 잡는 실수

두 도구는 당신 쪽에서 거의 노력 없이도 통째로 한 부류의 에러를 잡아낸다:

- **타입 체크** (TypeScript, mypy 등)는 “숫자가 기대되는 자리에 문자열을 넘겼다”와 “이 값은 undefined 일 수 있다”
를 잡는다. AI는 타입이 불가능하게 만드는 엇지 케이스를 잇는 것을 좋아한다.
- **린팅** (ESLint, Ruff 등)은 사용되지 않는 변수, 도달 불가능한 코드, 스타일 표류를 잡는다. AI가 손댄 코드베이
스가 서서히 죽처럼 멍게지는 것을 막아준다.

이것들을 일찍 설정하고 AI에게 따르라고 말하라: “타입 체커와 린터를 돌리고, 끝났다고 말하기 전에 그것들이 보고하
는 모든 것을 고쳐라.” 깨끗한 타입 체크를 완료의 정의 일부로 취급하라.

하지만 게으른 탈출구를 경계하라. 타입 체커가 불평할 때, 진짜로 올바른 수정은 그것이 찾아낸 케이스—undefined 값, 잘못된 타입—를 처리하는 것이다. 빠른 수정은 `// @ts-ignore, any`, 혹은 `# type: ignore` 주석으로 그 불평을 입막음하는 것인데, AI 는 빨간 글씨를 사라지게 하기 때문에 끊임없이 이것들에 손을 뻗는다. 그것들은 또한 점검이 도우려던 바로 그 자리에서 점검을 무가치하게 만든다. 디프에 억제 (suppression) 주석이 나타나면, 그것을 해결책이 아니라 질문으로 취급하라: “이게 왜 필요한가? 무시하지 말고 근본 타입을 고쳐라.”

AI 변경의 범위 좁히기

AI 가 일으키는 피해의 가장 큰 단일 원인은 한 번에 너무 많은 것을 바꾸게 두는 것이다. 아홉 파일에 걸친 400 줄 디프는 리뷰가 불가능하고 버그를 숨기기 쉽다.

변경을 작고 한정되게 유지하라:

- **한 번에 하나의 논리적 변경**을 요청하고, “온 김에 이것도…”는 하지 말라.
- 명시적으로 말하라: “src/auth.js 만 건드려라. 다른 것은 아무것도 리팩터링하지 말라.”
- 동작하는 상태를 자주 커밋해서, 항상 롤백할 깨끗한 지점을 가져라.
- AI 가 큰 재작성을 제안하면, 하나씩 리뷰할 수 있는 단계로 작업을 쪼개달라고 요청하라.

작은 범위는 다음 단계—디프 읽기—를 실제로 가능하게 만든다. 여기에는 복리 효과가 있다: 작은 커밋은 폭발 반경 (blast radius) 도 작다. 한정적인 변경이 틀린 것으로 판명되면, 커밋 하나를 되돌리고 10 분을 잃는다. 거대한 변경이 틀리면, 어느 부분이 망가뜨렸는지 풀어내느라 한 시간을 쓰거나, 오후 작업 전체를 버리게 된다. 범위 규율은 필요할 줄 알기도 전에 사두는 값싼 보험이다.

디프를 비판적으로 읽기

디프는 정확히 무엇이 바뀌었는지의 목록이다: 빨간 줄은 제거됨, 초록 줄은 추가됨. 디프 읽기는 바이브 코딩에서 가장 레버리지가 큰 습관인데, 바로 거기서 AI 가 당신이 요청하지 않은 무언가를 하는 것을 잡아내기 때문이다.

모든 줄을 이해할 필요는 없다. 다음의 위험 신호를 훑어라:

- **바뀔 거라 예상하지 않은 파일.** 왜 설정 파일이 건드려졌나?
- **삭제된 코드.** 검증 점검, 에러 핸들러, 혹은 “통과시키려고” 테스트를 제거했나?
- **비밀이나 하드코딩된 값.** 코드에 있으면 안 되는 API(애플리케이션 프로그래밍 인터페이스) 키, 비밀번호, URL(Uniform Resource Locator —웹 주소).
- **비활성화된 점검.** 건너뛴 테스트, 주석 처리된 가드, `// @ts-ignore`, 혹은 느슨해진 권한.
- **범위 확장 (scope creep).** 요청한 것과 멀리 떨어진 변경들.

뭔가 이상해 보이면 물어라: “X 를 왜 바꿨는지 설명하라—난 그걸 요청하지 않았다.” 특히 삭제는 AI 가 정당화하게 만들 어려. “에러를 일으키고 있었다”는 그 에러를 이해할 이유이지 점검을 삭제할 이유가 아니다.

습관을 들이는 실용적인 방법: 변경을 받아들이기 전에 디프를 읽어라, 절대 후에 말고. 코드가 한번 머지되고 앱이 동작하는 듯 보이면, 당신은 절대 돌아가서 그것을 읽지 않을 것이다—강제하는 마찰이 없으니까. 리뷰의 순간이 결정의 순간이어야 한다. 여기서 도구가 도움이 된다: `git diff --stat` 은 파일당 한 줄 요약을 줘서 예상치 못한 파일이 즉시 튀어나오게 하고, 풀 리퀘스트 페이지에서 리뷰하면 에디터 여기저기 흩어진 대신 변경 전체를 스크롤 가능한 한 화면에 드러낸다.

리뷰 게이트: 머지 전 체크리스트

리뷰 게이트는 변경이 라이브로 가기 전에 반드시 참이어야 하는 것들의 짧은 목록이다. 직접 훑거나, AI 가 각 항목을 확인하게 하라:

- ☐ 디프를 읽었고 무엇이 왜 바뀌었는지 이해한다
- ☐ 변경이 내가 실제로 요청한 것으로 한정되어 있다
- ☐ 테스트가 통과한다 (그리고 새 동작에는 테스트가 있다)
- ☐ 타입 체크와 린터가 깨끗하다
- ☐ 남겨진 비밀, 디버그 로그, 주석 처리된 코드가 없다
- ☐ 안전 점검이 조용히 제거되지 않았다

□ 적어도 정상 경로만이라도 직접 변경을 써봤다

박스 하나를 체크할 수 없다면, 끝난 게 아니다.

체크리스트가 의도적으로 짧은 이유는, 당신이 건너뛰는 게이트는 게이트가 아니기 때문이다. 특히 빠르게 움직이고 변경이 “괜찮아 보일” 때, 그냥 통과시키고 싶은 유혹이 있다. 사용자에게 노출되는 무엇에 대해서든 그 유혹에 저항하라. 유용한 요령은 AI가 당신을 위해 채우게 하는 것이다—목록을 붙여넣고 물어라: “방금 한 변경에 대해 이 리뷰 게이트를 훑어라. 각 항목에 대해 PASS 또는 FAIL 을, 한 줄짜리 증거와 함께 말하라.” AI 는 테스트를 돌리지 않고서는 정직하게 “테스트 통과”라고 쓸 수 없고, 줄마다 증거를 대게 강제하면 막연한 “좋아 보임”이 감사 가능한 무언가로 바뀐다.

CI 기초: 점검을 자동으로 만들기

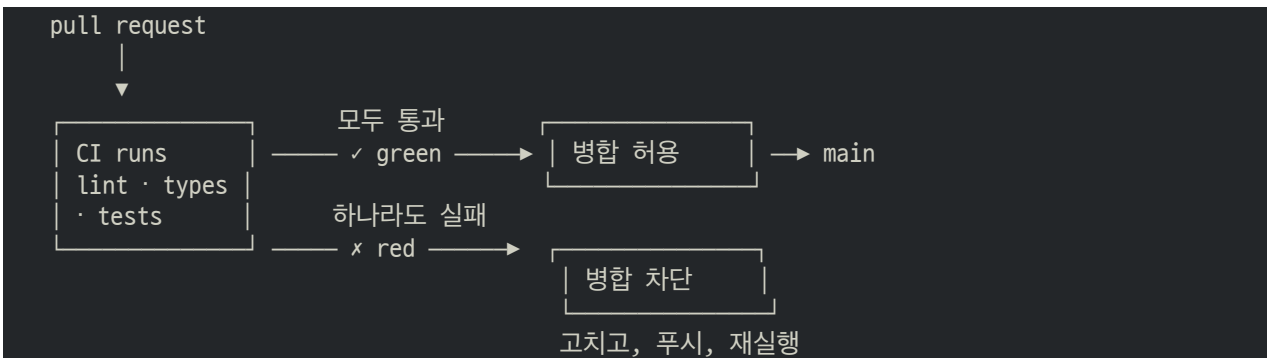
CI(Continuous Integration) 는 코드를 푸시할 때마다 당신의 점검을 돌리는 로봇이라, 점검 돌리는 것을 잊을 수 없다. 대부분의 플랫폼은 이것을 파일 하나로 만들어준다. 다음은 최소한의 GitHub Actions 예시다:

```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm test
```

이제 모든 푸시가 자동으로 린팅, 타입 체크, 테스트를 거친다. 무언가 실패하면 코드가 머지되기 전에 빨간 표시를 본다. 당신의 스택에 맞게 이 파일을 AI 에게 작성하라고 요청할 수도 있다—“모든 풀 리퀘스트에서 우리의 린트, 타입 체크, 테스트를 돌리는 GitHub Actions 워크플로를 추가하라.”

브랜치 보호를 켜두면, 빨간 점검은 경고만 하는 게 아니라 빌드가 다시 초록이 될 때까지 머지 버튼을 물리적으로 잠근다:



CI 의 핵심은 자동화 그 자체가 아니라, 점검을 당신의 기억에서 파이프라인으로 옮긴다는 것이다—거기서는 피곤한 밤에도 건너뛸 수 없다. 기본이 초록불로 돌아가면 할 만한 업그레이드가 둘 있다. 첫째, 점검들을 메인 브랜치의 필수 상태 (GitHub 의 “브랜치 보호 규칙”) 로 만들어서, 빨간 빌드가 무시할 수 있는 경고가 아니라 말 그대로 머지 버튼을 막게 하라. 둘째, CI 가 실패하면 실패 로그를 곧장 AI 에게 붙여넣어라: “이 실행에서 CI 가 실패했다, 여기 출력이다—진단하고 고쳐라.” 로그는 정밀한, 기계가 생성한 버그 리포트이고, 그것은 바로 AI 가 가장 잘 다루는 종류의 입력이다.

기본이 안전한 습관

빠르게 움직이되 두려움 없이 움직이게 해주는 반사 신경으로 한데 모아보자:

- **프로덕션이 아니라 브랜치에서 작업하라.** 브랜치는 공짜지만, 망가진 프로덕션은 아니다.
- **작고 동작하는 상태를 자주 커밋하라.** 잦은 커밋은 롤백을 사소하게 만든다.
- **항상 AI 가 작성한 것을 실행하게 요청하라**—테스트, 앱, 타입 체커.
- **터미널 하나는 AI 용, 하나는 당신용으로 유지하라,** 그래야 독립적으로 검증할 수 있다.
- **읽지 않은 변경을 절대 출시하지 말라.** 속도는 리뷰를 건너뛰는 데서가 아니라 좋은 가드레일에서 나온다.

가드레일은 당신을 느리게 하지 않는다—속이 줄이지 않고도 “그래, 출시해”라고 말할 수 있게 해주는 것이 가드레일이다. 한 번 설정해두면, 안전망이 추락을 받아내는 동안 AI 는 전력 질주할 수 있다.

가드레일이 비명을 지를 때: 고치기 전에 분류하라

가드레일은 당신이 잘 반응할 때에만 유용하고, 실패 양상은 예측 가능하다: 빌드가 빨개지고, 당신은 약간 당황하고, 에러를 “고치라”와 함께 AI 에게 붙여넣는다. 때로는 그게 통한다. 종종 그것은 실제로 무엇이 잘못됐는지 다루지 않으면서 빨간 글씨만 사라지게 하는 수정을 만들어낸다—AI 가 테스트를 약화시키거나, 타입을 넓히거나, 실패하는 호출을 문제를 삼켜버리는 try/catch 로 감싼다.

AI 가 무엇이든 “고치게” 두기 전에, 10 초를 분류에 써라. 가드레일 실패에는 사실상 세 종류뿐이다:

- **당신이 방금 들어온 진짜 버그.** 점검이 제 일을 하고 있다. 점검이 아니라 코드를 고쳐라.
- **이제 남아버린 테스트.** 당신이 의도적으로 동작을 바꿨고, 옛 단언이 더는 맞지 않는다. 새로운 올바른 동작을 단언하도록 테스트를 갱신하라—반사적으로가 아니라 의도적으로.
- **불안정하거나 잘못된 점검.** 정말로 드물다. 여기서는 자신을 의심하라; 이것은 사람들이 진짜 버그를 떠넘기는 통인데, 그게 그냥 넘어가게 해주는 유일한 통이기 때문이다.

AI 를 정직하게 유지하는 지시는, 무엇이든 바꾸기 전에 진단부터 하게 만드는 것이다:

"never discounts below zero" 테스트가 실패하고 있다.

먼저, 왜 실패하는지 말하라 — 지금 코드가 무엇을 하는지 대

테스트가 무엇을 기대하는지. 그런 다음 어느 쪽이 옳은지 말하라:

코드인가 테스트인가. 내가 확인하기 전까지는 아무것도 바꾸지 말라.

그 한 번의 멈춤이, 당신을 보호하는 가드레일과 당신이 AI 에게 조용히 무력화하도록 훈련시킨 가드레일의 차이이다. 빨간 점검은 정보다; 당신이 할 수 있는 최악은 그것이 무엇을 말하고 있었는지 읽지 않고 초록으로 만들려고 대가를 치르는 것이다.

정리 & 연습

핵심 요점

- 가드레일은 안전망이 추락을 받아 주는 동안 AI 가 전력 질주하게 해 주는 장치다—속도를 늦추는 게 아니라 높여 준다.
- 테스트, 타입 체크, 린팅은 공짜로 실수를 잡아 준다. CI 에 엮어 검사가 자동으로 돌게 하라.
- AI 변경의 범위를 좁게 한정하고, 머지하기 전에 모든 diff 를 비판적으로 읽어라.
- 빨간 검사는 정보다—AI 에게 “고치라”고 시키기 전에 분류하라 (진짜 버그 / 낡은 테스트 / 불안정한 검사).
- 읽지 않은 변경은 절대 출시하지 마라.

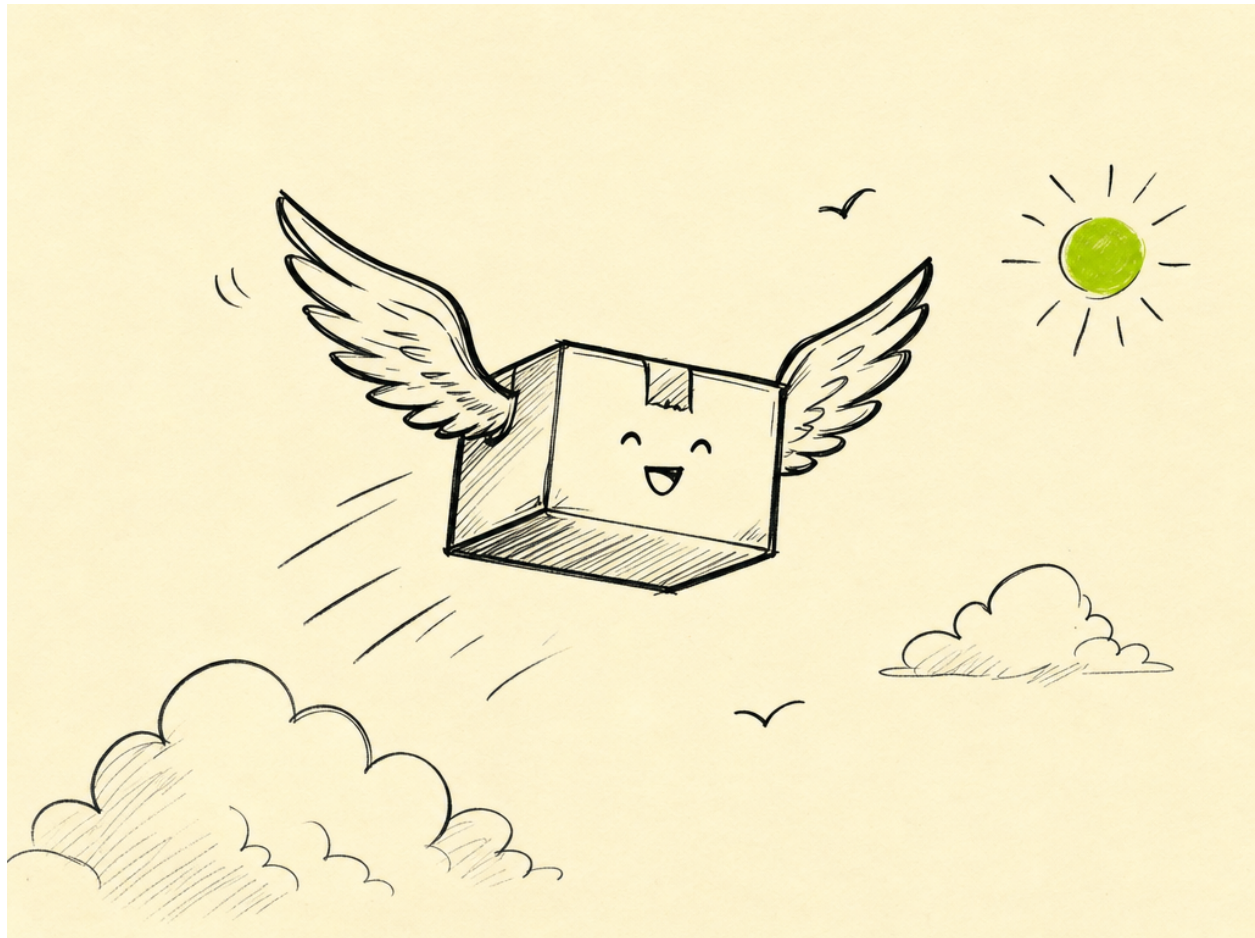
직접 해보기

당신의 프로젝트에서 아직 없는 자동 검사 하나—테스트 하나, 린터, 또는 타입 체크 단계—를 설정하고, push 때마다 되도록 CI 워크플로에 추가하라. 그런 다음 일부러 사소한 것을 망가뜨려, 고치기 전에 검사가 빨강게 되는지 확인하라.

```
Add a CI workflow to this repo that runs the test suite, the linter,
and the type checker on every push and pull request. Generate the
```

```
config, explain each step, and tell me how to read the output when a
check fails so I can tell a real bug from an out-of-date test.
```

프로덕션으로 출시하기



당신은 계속 빌드해왔다. 앱은 당신 머신에서 동작하고, 데모는 친구를 감탄시켰으며, 이걸 세상에 내놓고 싶은 근질거림은 진짜다. 이 순간이 많은 1인 빌더가 멈춰서는 지점이다—코드가 망가져서가 아니라, “라이브로 가기”가 그 나름의 흑마법(DNS(도메인 네임 시스템, yourdomain.com 같은 이름을 서버가 사는 숫자 주소로 바꿔주는 인터넷의 주소록), 서버, 비밀, 결제 웹훅)을 가진 다른 분야처럼 느껴지기 때문이다. 좋은 소식: 이것이야말로 AI가 가장 잘하는, 기계적이고 잘 문서화된 종류의 작업이다. 당신의 일은 지시하는 것. 그들의 일은 명령을 타이핑하는 것.

하지만 “지시한다”가 핵심 단어다. 이 책 전체의 주제는 당신이 출시하는 것을 이해하라는 것이다. 플래그를 외울 필요는 없지만, 각 단계가 무엇을 위한 것인지는 알아야 한다—웹훅이 당신을 무엇으로부터 지키는지, 왜 비밀이 절대 저장소에 들어가면 안 되는지, 지출 상한이 무엇을 가두는지. 그것의 형태를 이해하면, AI의 자신만만한 답이 슬그머니 틀렸을 때를 알아챌 수 있고, 첫 배포가 폭발할 때 복구할 수 있다. 출시에서 데이는 빌더는 거의 항상 자기가 읽지 못하는 명령을 붙여 넣은 사람들이다.

이 장은 동작하는 로컬 앱에서 사람들이 당신에게 결제할 수 있는 진짜 URL(유니폼 리소스 로케이터, 사람들이 브라우저에 타이핑하는 웹 주소)까지, 마지막 1마일 전체를 걷는다.

호스트를 고르고 배포하기

대부분의 바이브 코딩 앱에는, 당신이 돌보아야 하는 서버가 필요 없다. 현대 플랫폼은 당신의 코드를 받아 몇 분 만에 라이브 URL 을 준다. 흔한 선택지:

- **정적 사이트와 프론트엔드:** Cloudflare Pages, Vercel, Netlify, GitHub Pages.
- **풀스택 앱과 API(애플리케이션 프로그래밍 인터페이스, 한 프로그램이 다른 프로그램과 대화하는 데 쓰는 출입구):** Cloudflare Workers, Vercel, Render, Fly.io, Railway.
- **더 무거운 백엔드:** 컨테이너 플랫폼이나 작은 VPS(버추얼 프라이빗 서버, 데이터 센터에 늘려져 있는 컴퓨터의 한 조각을 빌려 쓰는 당신만의 몫).

올바른 수는 당신이 무엇을 만들었는지 AI 에게 말하고 추천하게 두는 것이다. 시도해보라: “이건 *Postgres* 데이터베이스를 쓰는 *Next.js* 앱이다. 넉넉한 무료 티어가 있는, 가장 싸고 믿을 만한 호스트를 원한다. 배포 단계와 설정 파일을 달라.” 선택을 실제로 제약하는 것들에 대해 구체적으로 말하라: 당신의 프레임워크, 데이터베이스가 있는지, 백그라운드 작업이나 파일 업로드가 필요한지, 그리고 예산. 정적 마케팅 페이지와 큐를 가진 풀스택 앱은 사는 집이 아주 다르다.

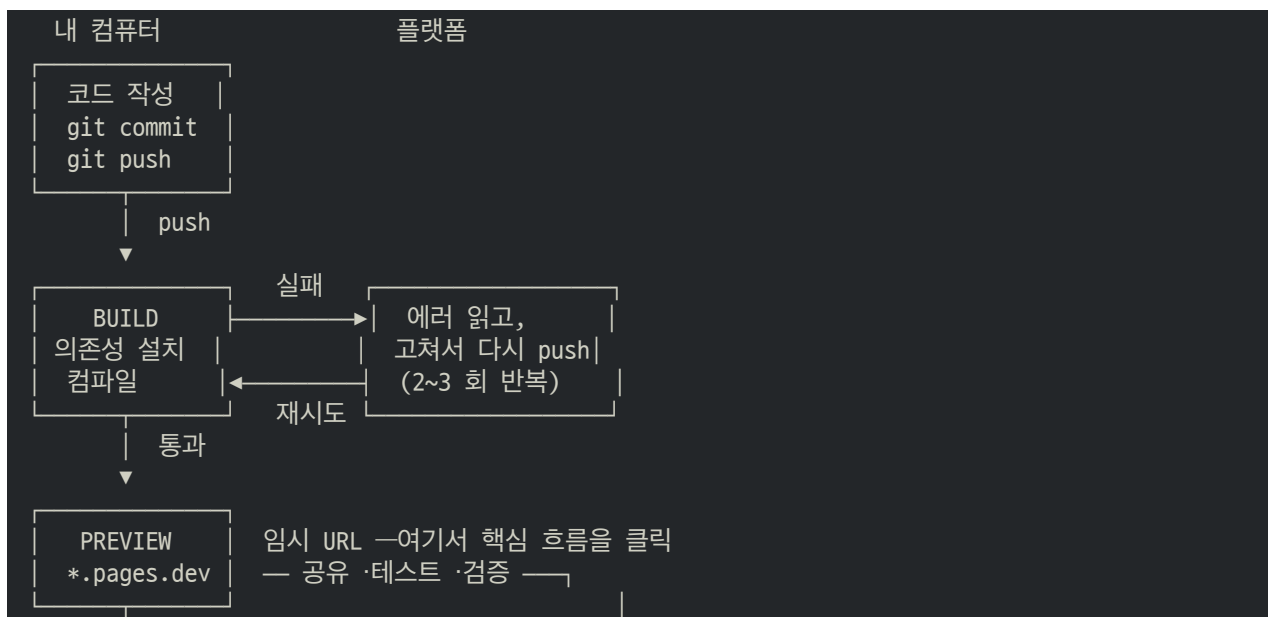
하나를 고르고 나면, 배포는 보통 명령 몇 개다:

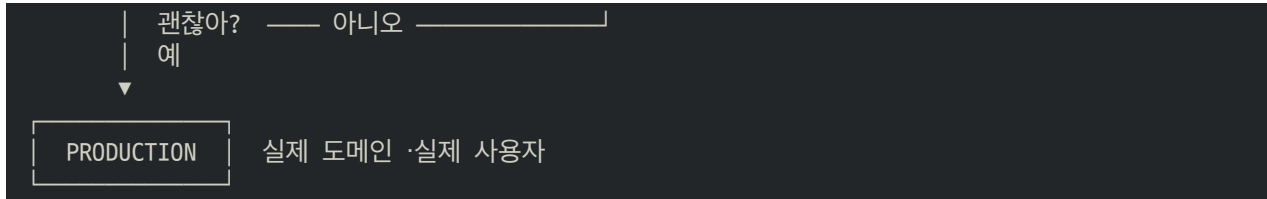
```
# 예시: Wrangler 를 통한 Cloudflare Workers
npm install -g wrangler
wrangler login
wrangler deploy

# 예시: Vercel
npm install -g vercel
vercel --prod
```

플랫폼 설정 (wrangler.toml, vercel.json 등) 은 AI 가 생성하게 하고, 첫 배포가 던지는 에러를 설명하게 하라. 첫 배포는 거의 항상 무언가를 던진다—누락된 빌드 명령, 잘못된 출력 디렉터리, Node 버전 불일치, 빌드가 필요로 하는데 찾지 못하는 환경 변수 (코드 바깥에서 앱에 건네지는, 이름이 붙은 설정값—현관 매트 밑에 둔 열쇠처럼 앱이 어디서 찾아야 하는지 알고 있는 것). 이 중 어느 것도 당신이 명칭한 짓을 했다는 뜻이 아니다; 플랫폼에 한 가지를 더 말해줘야 한다는 뜻이다. 에러를 그대로 다시 붙여 넣고 수정을 요청하되, 왜 일어났는지도 물어서 다음 번엔 덜 놀라게 하라. 두세 번 돌면 임시 URL 에서 라이브가 된다. 축하하기 전에 그 URL 을 열어 핵심 흐름을 클릭해 보라—“배포가 성공했다”와 “앱이 동작한다”는 같은 주장이 아니다.

모든 배포는 같은 파이프라인을 걷는다. 프리뷰 단계가 당신의 안전망이다: 모든 변경이 진짜 사용자에게 닿기 전에, 클릭해 볼 수 있는 자기만의 URL 을 갖는다.





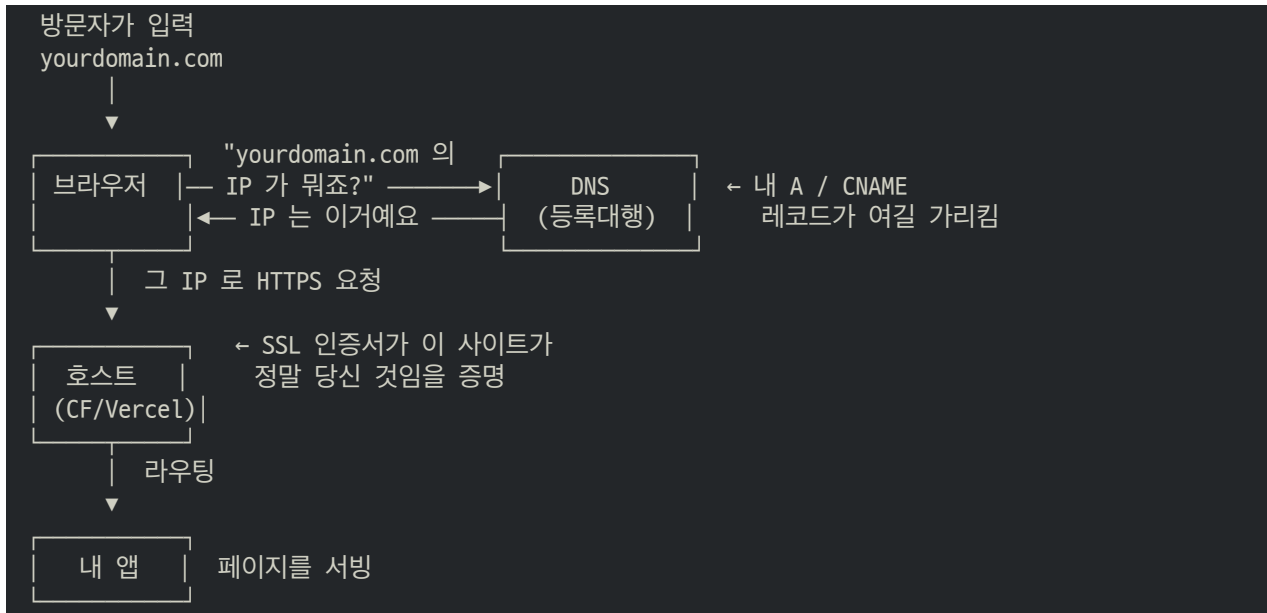
커스텀 도메인과 DNS

*.workers.dev 나 *.vercel.app URL 은 테스트에는 괜찮지만, 당신은 자기 이름을 원한다. 흐름:

1. 도메인을 산다 (Namecheap, Cloudflare Registrar, Porkbun —하나 골라라).
2. 그 DNS 를 당신 호스트로 향하게 한다. 보통 CNAME 이나 A 레코드를 추가하거나, 도메인의 네임서버를 통째로 바꾸는 것을 뜻한다.
3. 호스트의 대시보드에서 도메인을 추가하고, SSL 인증서 (사이트가 정말 당신 것임을 증명해 자물쇠 표시와 HTTPS(웹의 전송 프로토콜의 안전하게 암호화된 버전) 를 켜주는 디지털 신분증) 를 자동으로 발급하게 둔다.

레코드가 마법처럼 느껴지지 않도록 빠른 멘탈 모델을 잡자: **A 레코드**는 이름을 IP 주소로 곧장 향하게 한다; **CNAME** 은 한 이름을 다른 이름으로 향하게 한다 (호스트의 IP 가 당신 모르게 바뀔 수 있을 때 유용하다); 그리고 **네임서버**는 도메인 전체를 제공자에게 넘겨, 모든 레코드를 대신 관리하게 한다. 대부분의 현대 호스트는 CNAME 이나 전체 네임서버 전환을 선호한다—SSL 인증서와 엣지 라우팅을 직접 관리하고 싶어 하기 때문이다.

방문자의 요청이 거치는 전체 사슬과, 당신이 설정한 각 조각이 그 안 어디에 사는지는 다음과 같다:



DNS 는 초보자에게 가장 헷갈리는 단 하나의 부분인데, 대부분 전문 용어와 전파 지연 때문이다. AI 에게 번역하게 하라. 물어라: “내 도메인은 *Namecheap* 에 등록되어 있고 *Cloudflare Pages* 에 호스팅하고 있다. 정확히 어떤 레코드를, 어떤 값으로 추가해야 하는지, 그리고 작동했는지 어떻게 검증하는지 차근차근 안내하라.” 그런 다음 브라우저를 새로고침만 하지 말고, 빠른 점검으로 결과를 직접 검증하라:

```

# 세상이 실제로 보는 레코드를 조회
dig yourdomain.com
nslookup yourdomain.com

# 인증서가 살아 있고 만료되지 않았는지 확인
curl -I https://yourdomain.com
  
```

DNS 변경은 전파에 몇 분에서 하루까지 걸릴 수 있고, 당신 자신의 브라우저 캐시가 그중 가장 큰 거짓말쟁이다—인터넷의 나머지가 다 업데이트된 뒤에도 한참 동안 낡은 답을 기꺼이 보여줄 것이다. 즉시 라이브가 되지 않아도 그건 정상이다: 무언가 망가졌다고 가정하기 전에, 셀룰러 데이터로 폰에서 확인하거나 온라인 “DNS 전파”체커를 써보라.

환경 변수와 비밀

당신의 코드는 API 키, 데이터베이스 URL, 토큰이 필요하다. 이것들은 당신 저장소에 **절대** 커밋되어서는 안 된다. 비밀이 git 히스토리에 한 번 들어가면 사실상 영원히 공개된 것이다—나중 커밋에서 그 줄을 지워도 히스토리에서 사라지지 않으며, 봇들은 새 공개 저장소를 몇 분 안에 긁어 유출된 키를 찾는다. 두 규칙이 멀리까지 데려다준다:

- 비밀을 로컬에서 `.env` 파일에 두고, `.env` 가 `.gitignore` 에 있는지 확인하라.
- 프로덕션을 위해 같은 값들을 호스트의 대시보드에서 (또는 CLI(커맨드라인 인터페이스, 클릭 대신 명령을 타이핑하는 텍스트 프롬프트 창) 로) 환경 변수로 설정하라.

```
# .env (로컬 전용 —이건 절대 커밋하지 말 것)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# 프로덕션 대응 값을 호스트에 설정
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

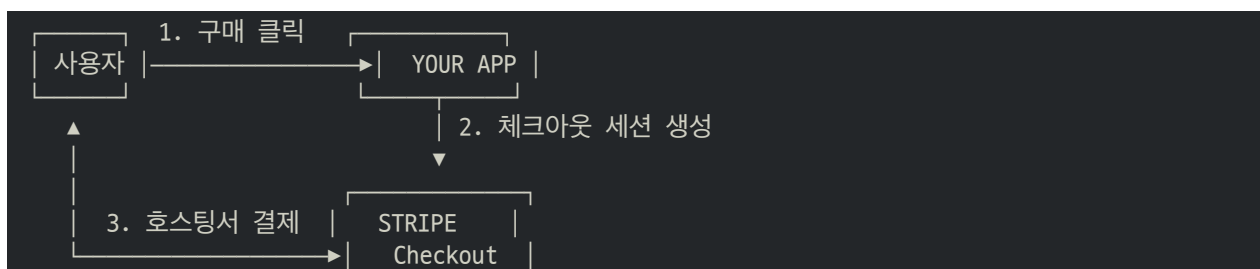
모든 변수의 (값이 아닌) 이름을 나열하는, 커밋된 `.env.example` 을 유지하라. 그래야 미래의 당신과 AI 둘 다 앱이 무엇을 기대하는지 안다. 위의 `sk_test_` 접두사를 보라—테스트 키와 라이브 키를 분명히 구별되게 유지하고, 라이브 키가 채팅에 붙여 넣을 수도 있는 로컬 `.env` 에 절대 놓이지 않게 하라. 채팅에 비밀을 붙여 넣거나 실수로 하나를 커밋하게 되면, 그것을 유출된 것으로 취급하고 즉시 교체하라: 제공자 대시보드에서 새 키를 생성하고, 모든 곳에서 갱신한 뒤, 옛 키를 폐기하라. 교체는 5 분이 들지만, 유출된 데이터베이스 URL 은 데이터베이스 안의 모든 것을 잃게 할 수 있다.

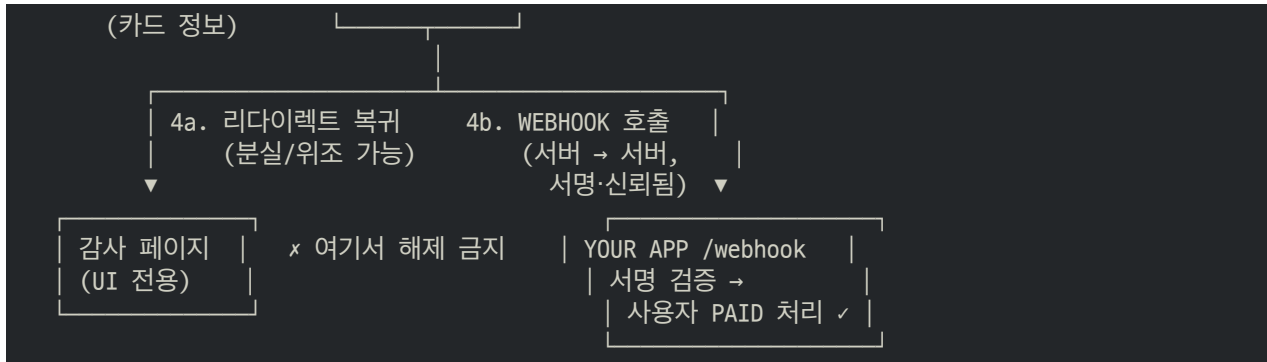
결제 연결하기

돈을 받고 싶다면, 1 인 빌더를 위한 표준 경로는 Stripe 다 (판매세를 대신 처리하길 원한다면 Lemon Squeezy / Paddle —등록 판매자 (merchant of record) 로서, 더 큰 몫을 떼어가지만 세금 골치를 사라지게 한다). 그 메커니즘은 반복적이고 잘 문서화되어 있어서, 위임하기에 이상적이다. 필요한 조각들:

- **체크아웃** 흐름—Stripe Checkout 이 가장 빠르다; 호스팅된 결제 페이지를 건네줘 당신이 원시 카드 정보를 절대 만지지 않게 한다.
- **웹훅**은 당신 앱이 “결제가 됐나?”라고 거듭 물어보는 대신, 결제가 성사되는 순간 결제 회사가 당신 앱에 전화를 걸어주는 것이라고 생각하면 된다. **웹훅** 엔드포인트가 바로 그것이다: 결제 제공자가 호출하는 엔드포인트 (당신 서버의 URL) 로, 결제가 실제로 성공했을 때 당신 앱이 알게 해준다. 이것이 초보자가 건너뛰고는 왜 주문이 절대 이행되지 않는지 궁금해하는 부분이다. 당신 사이트로 돌아오는 리다이렉트는 닫히거나, 막히거나, 위조될 수 있다; 웹훅이 서버 대 서버의 진실의 원천이다. 리다이렉트만 절대 믿지 말라; 웹훅을 믿어라.
- **이용 권한 (entitlement) 을 저장하는** 방법—데이터베이스에서 사용자를 결제 완료로 표시하라. 그래야 새로고침이나 새 기기에서도 그들이 산 것을 여전히 본다.

핵심 통찰은 어느 화살표를 믿느냐다. 당신 사이트로 돌아오는 사용자 리다이렉트는 사라지거나 위조될 수 있다; 웹훅이 실제로 제품을 잠금 해제하는 서버 대 서버의 선이다:





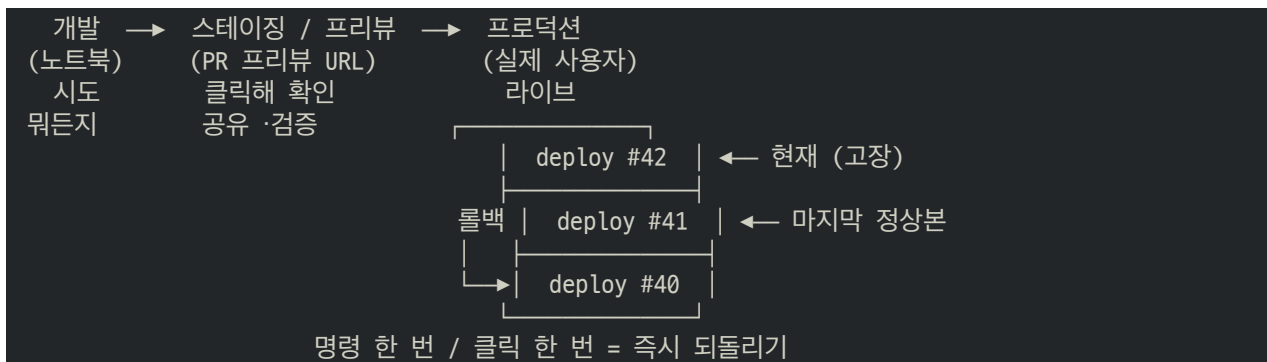
AI 에게 세 가지 모두를 스캐폴드하고 웹훅 서명 검증을 설명하게 하라. 그것이 보안에 민감한 단 하나의 단계이기 때문이다—그것이 없으면 당신의 웹훅 URL 을 찾은 누구라도 “결제 성공” 이벤트를 위조해 제품을 공짜로 풀 수 있다. 라이브 키로 전환하기 전에 Stripe 의 테스트 모드에서 그들의 테스트 카드 번호 (4242 4242 4242 4242 가 유명하다) 로 모든 것을 테스트하고, Stripe CLI 를 써서 웹훅 이벤트를 당신 로컬 머신으로 포워딩해 그것들이 발화하는 것을 지켜보라. 테스트 모드 리허설을 건너뛰지 말라. 당신 앱을 통과하는 첫 진짜 돈은, 이미 열 번은 동작하는 것을 본 거래여야 한다.

스테이징과 롤백

우아하게 복구하는 출시와 한밤중 패닉으로 변하는 출시를 가르는 두 가지 습관: 변경이 진짜 사용자에게 닿기 전에 안전한 곳에서 테스트하는 것, 그리고 나쁜 배포를 즉시 되돌릴 수 있는 것.

스테이징 환경은 그저 당신 앱의 두 번째 복사본이다—같은 코드, 별도의 URL, 별도의 (또는 일회용) 데이터베이스—프로덕션 전에 시도해 보는 곳이다. 대부분의 호스트는 이것을 거의 공짜로 만든다: 모든 풀 리퀘스트가 자동으로 자기만의 **프리뷰 URL** 을 받아서, 머지하기 전에 변경을 클릭해 보고 누군가와 공유할 수 있다. 데이터베이스 마이그레이션처럼 쉽게 되돌릴 수 없는 것에 특히 써라.

세 환경을 변경이 올라가는 사다리로 그려보라. 롤백은 프로덕션을 마지막 정상 단으로 곧장 떨어뜨리는 밧줄이다:



롤백은 앱 전체에 대한 당신의 실행 취소 버튼이다. 이 플랫폼들은 이전의 모든 배포를 보관하기 때문에, 마지막 정상 버전으로 돌아가는 것은 보통 명령 하나나 대시보드 클릭 하나다:

```
# Cloudflare Pages: 정상으로 알려진 빌드를 재배포
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: 이전 배포를 프로덕션으로 승격
vercel rollback <DEPLOYMENT_URL>
```

이 명령을 필요할 때가 아니라 필요해지기 전에 알아두라. 바이브 코딩 속도의 핵심은 출시가 싸다는 것이다—하지만 싼 출시는 출시 취소도 똑같이 쌀 때만 안전하게 느껴진다.

모니터링과 에러 추적

사람들이 당신 앱을 쓰기 시작하면, 언제 깨지는지 알아야 한다—이상적으로는 그들이 말해주기 전에. 가벼운 스타터 키트:

- **에러 추적:** Sentry 는 스택 트레이스와 함께 예외를 잡아 그룹화하고, 어느 사용자와 어느 줄이 각각을 일으켰는지 알려준다. 설치는 몇 줄.
- **가동시간 모니터링:** 무료 핑거 (UptimeRobot, Better Stack) 가 몇 분마다 당신 사이트를 때려보고 다운되면 이메일로 알린다.
- **로그:** 호스트가 로그를 어디에 두는지 알아라 (wrangler tail, Vercel 대시보드 등). 그래야 무언가 잘못돼 보일 때 조사할 수 있다.

AI 에게 Sentry 를 추가하고 기본 헬스 체크 엔드포인트를 연결하게 하라—200 OK 를 반환하고 데이터베이스가 닿는지 확인하는 /health 같은 작은 라우트로, 가동시간 모니터가 핑할 수 있다. 목표는 엔터프라이즈급 관측성이 아니다—단지 화난 트윗으로 장애를 알게 되지 않는 것이다.

성능과 캐싱

느린 앱은 사용자를 잃고 검색에서 더 낮게 랭크된다. 첫날부터 미세 최적화할 필요는 없지만, 값싼 승리는 땀겨라:

- 당신의 파일 복사본을 전 세계 곳곳의 창고에 두어 방문자마다 자기에게 가장 가까운 곳에서 받게 한다고 상상해 보라. 그것이 **CDN**(콘텐츠 전송 네트워크) 이다—정적 자산을 이것으로 제공하라 (대부분의 호스트가 자동으로 해준다).
- 자주 바뀌지 않는 비싼 응답과 데이터베이스 쿼리를 **캐시**하라.
- 이미지를 압축하고 지연 로딩하라; JavaScript 를 덜 보내라.
- Chrome DevTools 에서 **Lighthouse** 를 돌리고, 우선순위가 매겨진 수정을 위해 그 리포트를 AI 에게 먹여라.

물어라: “여기 내 *Lighthouse* 리포트다. 가장 적은 노력으로 가장 큰 영향을 내는 세 가지 변경을 달라.” 추측이 아니라 측정이 당신이 최적화할 것을 골게 하라—느낌상 느리다는 이유로 엉뚱한 것을 하루 종일 최적화하는 일은 정말 흔하다. 진짜 비용이 인덱스 없는 쿼리 하나나 압축을 잊은 4 MB 짜리 히어로 이미지였을 때 말이다.

출시 전 체크리스트

무언가를 발표하기 전에, 이 목록을 걸어라. 당신의 기억을 믿는 대신 AI 가 각 항목을 검증하도록 도와주게 하라.

- ☐ 앱이 깔끔하게 배포되고 프로덕션 URL 이 로드된다
- ☐ 커스텀 도메인이 유효한 HTTPS 로 해석된다
- ☐ 모든 비밀이 프로덕션에 설정됨; 저장소에 커밋된 것은 없음
- ☐ .env 가 gitignore 되어 있고 .env.example 이 최신이다
- ☐ 결제가 테스트 모드에서 (웹훅 포함) 처음부터 끝까지 테스트됨
- ☐ 가입, 로그인, 핵심 정상 경로가 모두 프로덕션에서 동작한다
- ☐ 에러 추적과 가동시간 모니터링이 라이브다
- ☐ 롤백하는 법을 알고, 한 번 테스트해 보았다
- ☐ 모바일 레이아웃을 실제 폰에서 확인했다
- ☐ 기본 개인정보 처리방침과 이용약관 페이지가 존재한다
- ☐ 데이터베이스 백업이 활성화되어 있다
- ☐ 사용자가 당신에게 연락할 명확한 방법 (이메일이나 폼)

출시 후

출시는 시작이지 끝이 아니다. 첫 버전은 가설이고, 진짜 사용자가 실험이다. 에러 대시보드를 지켜보고, 모든 메시지를 읽고, 작은 수정을 빠르게 출시하라. 마찰의 첫 신호에 모든 것을 다시 만들고 싶은 충동을 참아라—날카로운 모서리를 패치하고, 사람들이 실제로 쓰는 것에 집중하며, 나머지는 기다리게 두어라.

여기서 가장 강력한 습관은 촘촘한 루프다: 문제나 아이디어를 알아채고, AI 에게 묘사하고, 변경을 출시하고, 결과를 지켜본다. 기계적 작업이 위임되기에, 당신은 몇 주가 아니라 몇 시간 만에 반복할 수 있다. 그 속도가—어떤 단일 기능이 아니라—바이브로 빌드하는 것의 진짜 이점이다. 당신은 라이브다. 이제 계속 가라.

정리와 연습

핵심 요약

- 배포는 한순간이 아니라 워크플로다: 배포하고, 관찰하고, 고치고, 반복하라—그리고 롤백은 미리 한 번 테스트해 둔 것으로 만들어라.
- 무언가를 알리기 전에 출시 전 체크리스트를 돌리고, 기억에 의존하지 말고 AI 에게 각 항목을 검증하게 하라.
- 측정이 최적화를 이끌게 하라—“느린 것 같다”는 직감보다 Lighthouse 리포트가 항상 낫다.
- 비밀값은 프로덕션 설정에 두고 절대 저장소에 두지 마라. .env 는 gitignore 되고 .env.example 은 최신으로 유지한다.
- 첫 버전은 가설이고 실제 사용자가 실험이다. 작고 빠른 수정으로 반복하라.

직접 해보기

배포한 앱 (또는 지금 작은 앱을 하나 배포) 을 골라 출시 전 체크리스트를 한 줄씩 대조하라. 정직하게 체크할 수 없는 항목마다 실패한 항목을 AI 에 붙여 넣고 그 차이를 메우는 가장 작은 변경을 요청하라. 마지막으로 홈페이지가 아니라 라 이브 사이트의 딥 링크를 클릭해, 프로덕션이 개발 환경처럼 동작하는지 확인하며 마무리하라.

이 장의 프롬프트

```
Here is my deployed app and its stack: [describe app + hosting + services].
Act as my launch reviewer. Walk this pre-launch checklist item by item
and, for each, tell me how to verify it on the LIVE site (not in dev):
- production URL loads over valid HTTPS on the custom domain
- all secrets set in production; none committed to the repo
- core happy path (sign-up, login, main action) works in production
- payments tested end-to-end including the webhook (if any)
- error tracking and uptime monitoring are live
- I can roll back, and have tested it once
- mobile layout works on a real phone
- database backups are enabled
For every item that fails, give me the SMALLEST change that fixes it.
Do not assume anything passes — make me prove each one.
```

사례 연구: 0 에서 출시까지



이론은 유용하지만, 바이브 코딩을 가르치는 데는 그것이 실제로 일어나는 것을 보는 것만 한 게 없다. 이 장은 AI 를 지시하여 처음부터 끝까지 만들고 출시한, 작지만 진짜인 네 프로젝트를 걸어본다. 어느 것도 장난감이 아니다. 각각에는 사용자, 도메인 이름, 그리고 걸린 돈이나 시간이 있다.

각 사례에서 당신은 아이디어, AI 에게 건넨 스펙, 우리가 고른 스택, 실제로 판세를 바꾼 프롬프트, 우리를 거의 탈선시킬 뻔한 장애물, 그리고 그것이 어떻게 라이브가 됐는지를 보게 된다. 이것들을 하이라이트 영상이 아니라 런북으로 읽어라. 핵심은 이 정확한 프로젝트들이 중요하다는 게 아니라—그 수 (手) 들이 반복된다는 것이다. 네 번째쯤 가면 당신은 리듬을 알아보기 시작할 것이다: 짧은 스펙, 매니지드 스택, 증거 기반 디버깅, 그리고 현실에 대고 시험하는 출시.

사례 연구 1: 다국어 랜딩 + 결제 페이지

아이디어

한 친구가 한국으로 이주하는 외국인을 위한 29 달러짜리 PDF 가이드를 판다. 그녀는 전 세계에서 빠르게 로드되고, 영어·한국어·중국어로 말하며, 서버를 만지지 않고도 카드 결제를 받는 단일 페이지를 원했다.

스펙

코드를 한 줄이라도 쓰기 전에 스펙을 한 문단으로 유지했다:

29 달러짜리 디지털 제품을 위한 단일 페이지 마케팅 사이트를 만들어라.
세 언어 (en/ko/zh) 를, 브라우저에서 자동 감지하되
토글로 전환 가능하게. Stripe Checkout 을 여는 " 구매 " 버튼 하나와

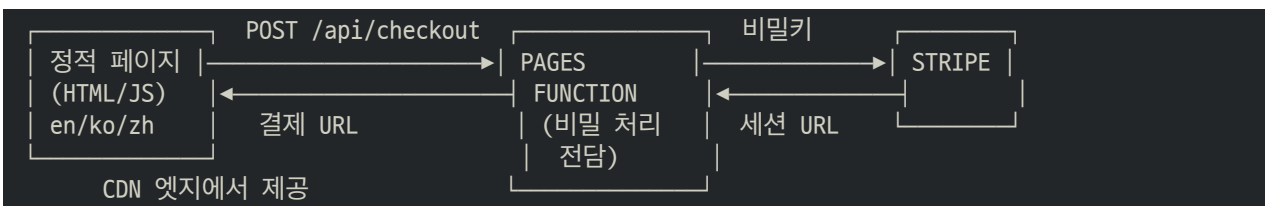
성공 시 감사 페이지로 리다이렉트. 내가 유지보수해야 하는 백엔드 서버는 없이. 어디서든 빠르게 로드되어야 한다.

스펙이 무엇을 말하지 않는지 주목하라: 프레임워크 없음, 디자인 시스템 없음, 애널리틱스 없음, 이메일 수집 없음. 그 하나하나가 합리적인 기능이었을 것이고, 그 하나하나가 첫 출시를 늦췄을 것이다. 한 문단짜리 스펙의 규율은 대부분 무언가를 빼는 규율이다.

스택

“내가 유지보수하는 백엔드 없음”에 “전 세계에서 빠름”을 더하면, 비밀을 필요로 하는 단 하나 (Stripe 세션 생성) 를 위한 작은 서버리스 함수가 딸린 엣지 CDN(콘텐츠 전송 네트워크—당신의 파일 복사본을 전 세계 데이터 센터에 두어 방문자마다 가까운 곳에서 받게 하는 것) 위의 정적 호스팅을 골장 가리켰다. 우리는 Cloudflare Pages 위의 정적 사이트에 단일 Pages Function 을 선택했다. 전체 아키텍처가 한 문장에 들어간다—함수 하나를 호출하고 그 함수가 Stripe 와 통신하는 정적 페이지—그리고 그 단순함이 나중에 디버깅 가능하게 만든 것이다.

전체 시스템은 박스 세 개이고, 비밀을 건드리는 것은 가운데 하나뿐이다:



핵심 프롬프트

우리는 넓게 시작해서 AI 가 구조를 제안하게 했다:

정적 랜딩 페이지 (순수 HTML/CSS/JS, 프레임워크 없음) 를 en/ko/zh 언어 토글과 함께 스캐폴드하라. 카피를 언어를 키로 하는 단일 JS 객체에 저장하라. 기본값을 navigator.language 에서 감지하라. index.html 하나에 main.js 하나로 유지하라.

그런 다음 비밀이 사는 결제 경로:

/api/checkout 에 Cloudflare Pages Function 을 추가하라. 단일 29 달러 제품에 대한 Stripe Checkout 세션을 생성하고 리다이렉트 URL 을 반환하는 것이다. STRIPE_SECRET_KEY 를 환경에서 읽고, 절대 하드코딩하지 말라. 구매 버튼은 이 함수에 POST 한 다음 반환된 URL 로 window.location 해야 한다.

“절대 하드코딩하지 말라”는 문구는 제 자리를 한다. 제 기본값에 맡겨두면, AI 는 예제를 “동작”하게 만들려고 플레이스홀더 키를 인라인으로 즐겁게 떨어놓고, 플레이스홀더는 어느새 출시되는 진짜 키가 되는 법이다. 프롬프트에서 한 번 소리 내어 말하는 것이 리뷰에서 잡아내는 것보다 싸다.

장애물

첫 라이브 테스트가 실패했다: 구매 버튼이 아무것도 하지 않았고, 브라우저 콘솔에 CORS(크로스 오리진 리소스 셰어링—어떤 사이트가 어떤 사이트를 호출해도 되는지에 대한 브라우저의 규칙) 에러가 나왔다. 우리는 추측하지 않았다. 정확한 에러를 그대로 다시 붙여 넣었다:

구매를 클릭하면 이렇게 로그된다: "Access to fetch at '/api/checkout' blocked by CORS policy." 함수와 페이지는 같은 도메인에 있다. 여기 함수 코드다: [붙여넣음]. 실제로 뭐가 잘못됐나?

AI 가 즉시 그것을 잡아냈다: 함수가 Stripe URL 을 JSON(자바스크립트 오브젝트 노테이션—프로그램들이 구조화된 데이터를 주고받는 데 쓰는, 단순하고 읽기 쉬운 텍스트 형식) 으로 반환하고 있었지만, 함수의 OPTIONS 프리플라이트가

처리되기 전에 fetch 가 만들어지고 있었다. 우리가 실수로 정적 페이지와 함수를 서로 다른 두 Pages 프로젝트에 배포 했기 때문이었다. 진짜 수정은 코드가 아니라 배포 토폴로지였다. 함수를 같은 프로젝트의 /functions 디렉터리로 옮기자 에러가 사라졌다.

이 교훈은 곱씹을 가치가 있다: 증상은 코드에 있었지만 (fetch 에서 나온 CORS 에러), 원인은 인프라에 있었다 (하나가 아니라 두 프로젝트). 만약 우리가 설정을 붙여 넣지 않고 AI 에게 “CORS 에러를 고쳐”라고 시켰다면, AI 는 Access-Control-Allow-Origin 헤더를 덧붙였을 것이다—컴파일되고, 증상을 덮고, 진짜 버그를 살려두는 수정. 문자 그대로의 에러와 함께 그 주변 맥락을 붙여 넣고, 당신의 설정과 맞지 않으면 첫 번째 그럴듯한 설명을 받아들이지 말라.

출시

우리는 라이브 Stripe 키를 Pages 대시보드에 암호화된 환경 변수로 추가하고 (저장소에는 절대 넣지 않고), 그녀 도메인의 DNS(도메인 네임 시스템, 이름을 서버에 매핑해주는 인터넷의 주소록) 를 Pages 로 향하게 한 뒤, 카드로 진짜 29 달러 테스트 구매를 한 번 돌리고 환불했다. 우리는 또 실패 경로도 일부러 클릭해봤다—거절되는 테스트 카드—그녀가 청구되지 않고 빈 화면 대신 멀쩡한 메시지를 보는지 확인하기 위해. 오후 한나절 만에 라이브. 그녀는 그날 저녁 첫 판매를 했다.

사례 연구 2: Auth + 데이터베이스가 있는 작은 SaaS 도구

아이디어

한 프리랜스 디자이너가 클라이언트별 청구 가능 시간을 기록하고 월간 CSV(콤마로 구분된 값—어떤 스프레드시트 앱으로도 열 수 있는 일반 텍스트 스프레드시트 파일) 로 내보내는 비공개 대시보드를 원했다. 화려할 것은 없지만, 계정 (그래서 그녀의 데이터가 오롯이 그녀 것이 되도록) 과 영속성이 필요했다.

스펙

Auth 와 데이터베이스는 판돈을 올리므로, 스펙은 경계에 대해 더 구체적이 됐다:

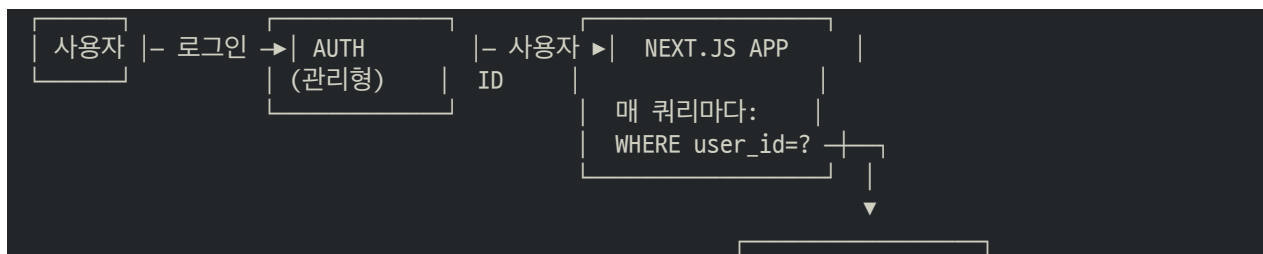
로그인한 웹 앱으로, 사용자가 다음을 할 수 있다: 이메일로 가입/로그인, 클라이언트 생성, 시간 항목 기록 (날짜, 클라이언트, 시간, 메모), 월별로 필터링된 항목 테이블 보기, 그 달을 CSV 로 다운로드. 각 사용자는 오직 자기 데이터만 본다. 모바일 친화적으로.

“각 사용자는 오직 자기 데이터만 본다”는 UX 줄처럼 보인다. 그것은 사실 앱 전체의 보안 모델을 아홉 단어로 압축한 것이다. 스펙에서 그것을 명명한 덕에, AI 가 표류할 때마다 우리는 거기로 되짚어 가리킬 수 있었다.

스택

1 인 빌더에게 이기는 수는 auth 와 데이터베이스가 당신이 작성하는 코드가 아니라 매니지드 서비스인 스택이다. 우리는 서버리스 호스트에 배포한 Next.js 앱에, 호스팅된 Postgres 데이터베이스와, 이메일 로그인·세션·비밀번호 재설정을 대신 처리해주는 드롭인 auth 공급자를 선택했다. 잘못될 코드가 적다는 것은 AI 가 우리를 대신해 잘못 만들 코드가 적다는 뜻이다. 특히 auth 는 AI 와 함께 직접 만들고 싶은 마음이 거의 들지 않는 부류다: 실패 양태가 조용하고, 폭발 반경이 모두의 계정이며, 매니지드 공급자는 수백만 번의 로그인으로 엣지 케이스를 두들겨 맞아왔다.

매니지드 조각들은 바깥에 있고, 당신이 소유하는 단 하나의 규칙은 가운데에 있다—모든 쿼리가 user_id 로 필터해서, 한 사용자가 다른 사용자의 행을 절대 읽을 수 없게:




```
| POSTGRES (managed) |  
| clients · entries |
```

↑ 관리형 = 틀릴 여지가 적음 ↑ 당신이 지키는 단 하나의 경계

핵심 프롬프트

auth 공급자 자체의 템플릿이 무거운 작업을 하게 두고, 그 위에 도메인 로직을 얹도록 AI 를 지시했다:

우리는 [auth 공급자] 의 Next.js 스타터를 쓰고 있다. 두 테이블을 가진 Postgres 스키마를 추가하라: clients (id, user_id, name) 와 time_entries (id, user_id, client_id, date, hours, note). 모든 쿼리는 반드시 세션의 로그인한 사용자 id 로 필터해야 한다. 마이그레이션과 타입이 지정된 데이터 접근 함수를 생성하라.

“반드시 user_id 로 필터하라”는 줄이 프로젝트 전체에서 가장 중요한 문장이었다. 우리는 데이터를 건드리는 거의 모든 프롬프트에서 그 제약을 반복했는데, 다중 사용자 앱에서 가장 무서운 단일 버그가 한 사용자가 다른 사용자의 행을 보는 것이기 때문이다. 타이핑할 때는 반복이 군더더기처럼 느껴진다; 바로 그 군더더기가 당신을 구하는데, 모델은 별개의 프롬프트들 사이에서 그 제약이 얼마나 중요한지에 대한 기억이 없기 때문이다.

내보내기에 대해서는:

한 달 (YYYY-MM) 을 받아, 그 달에 대한 로그인한 사용자의 time_entries 를 클라이언트 이름과 조인해 끌어오고, CSV 다운로드를 스트리밍하는 /api/export 라우트를 추가하라. 유효한 세션이 없으면 요청을 거부하라.

장애물

테스트하면서 우리는 계정 두 개를 만들었고, 계정 B 가 드롭다운에서 계정 A 의 클라이언트를 볼 수 있음을 발견했다. 이것이 바로 우리가 두려워한 버그다. AI 에게 “고쳐”라고 요청하기보다, 먼저 문제를 증명하게 만들었다:

계정 B 가 계정 A 의 클라이언트를 보고 있다. 코드베이스에서 clients 테이블을 읽는 모든 데이터베이스 쿼리를 보여주고, 각각에 대해 세션 user_id 로 필터하는지 알려달라. 아직 아무것도 고치지 말고 —감사만 하라.

감사 결과 쿼리 하나—드롭다운 로더—가 드러났는데, 그것은 우리가 제약을 추가하기 전에 작성되어 빠져나갔던 것이었다. 우리는 누락된 필터를 추가하게 한 다음, 가드를 요청했다:

모든 읽기가 거쳐 가는 단일 헬퍼를 추가하라. 세션을 받아 user_id 필터를 주입해서, 미래의 어떤 쿼리도 그것을 잊을 수 없게. 기존 쿼리들을 이걸 쓰도록 리팩터링하라.

그것이 일회성 수정을 구조적 보장으로 바꿨다. 그런 다음 우리는 그 보장을 테스트 가능하게 만들었는데, 검증할 수 없는 가드는 그저 희망에 불과하기 때문이다:

사용자 두 명을 만들고, 각자가 클라이언트를 하나씩 생성하게 한 뒤, 사용자 A 의 세션이 어떤 데이터 접근 함수를 통해서도 사용자 B 의 클라이언트를 절대 읽을 수 없음을 단언하는 테스트를 작성하라.

교훈: AI 가 보안 버그를 들여오면, 그 인스턴스만 패치하지 말라—그 실수의 부류를 제거하도록 지시한 다음, 누군가 그 것을 다시 열면 시끄럽게 실패하는 테스트로 그 부류를 잠가버려라.

출시

우리는 테스트용 한 달치 데이터를 시드하고, CSV 를 내보내고, 스프레드시트에서 열어 숫자와 인코딩이 맞는지 확인한 뒤, 강력한 데이터베이스 비밀번호를 설정하고 자격 증명을 어떤 로컬 파일에서도 빼내 교체했다. 서버리스 호스트에 배포하고, 그 대시보드에 프로덕션 환경 변수를 추가하고, 그녀에게 URL 을 줬다. 그녀는 진짜 가입으로 스스로 온보딩했다. 전체 빌드는 주말 하나였다.

사례 연구 3: 개인 자동화 유틸리티

아이디어

순전히 이기적인 것: 매일 아침 그날의 캘린더 일정과 날씨를 가져와, 한 줄 요약으로 포맷하고, 개인 텔레그램 채팅으로 보내는 스크립트. 핵심은 커피 전에 앱 세 개를 여는 것을 그만두는 것이었다.

스펙

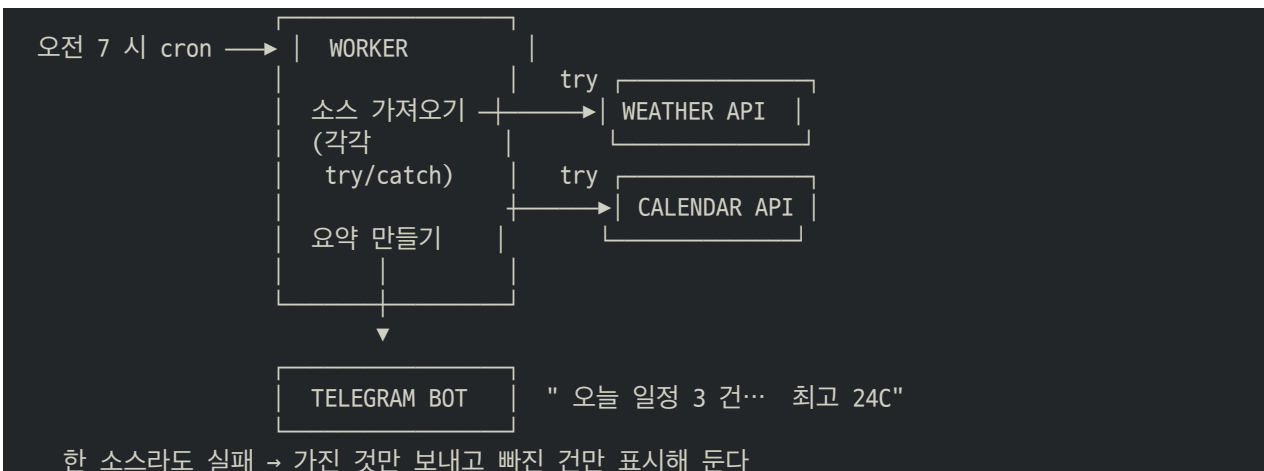
내 시간대로 오전 7 시에 돌아가는 예약 작업. 내 캘린더에서 오늘의 일정을 읽고, 내 도시의 예보를 가져오고, 이런 텔레그램 메시지 하나를 보낸다: " 오늘 일정 3 개, 첫 일정 9:30. 최고 24C, 오후 4 시 이후 비." 어떤 소스가 실패하면, 가진 것이라도 보내고 무엇이 빠졌는지 메모하라.

그 마지막 문장—충돌하는 대신 우아하게 성능 저하시키기—는 명시적으로 요청해야 하는 종류의 동작이다. AI 는 그것을 가정하지 않는다. 말하지 않으면, 모델이 기본으로 손을 뻗는 것은“에러 시 throw”이고, 오전 7 시 cron 작업에서 그것은 조용한 아침과 왜 그런지에 대한 단서 부재를 뜻한다.

스택

단일 예약 서버리스 함수 (Cron 으로 트리거되는 Worker) 라, 계속 돌려둘 머신이 없다. 전체가 파일 하나에 스케줄 하나다. 개인 유틸리티에서 이것은 들리는 것보다 더 중요하다: 보살펴야 할 서버가 있는 무엇이든 결국 당신이 결제를 잊어버릴 것이고, 그러면 프로젝트는 실패가 아니라 방치로 죽는다.

트리거 하나가 두 소스로 펼쳐졌다가 메시지 하나로 모인다. 각 소스를 감싼 try/catch 가, 한 번의 실패가 전체 실행을 죽이는 대신 우아하게 성능 저하시키게 한다:



핵심 프롬프트

Asia/Seoul 오전 7 시에 cron 으로 트리거되는 Cloudflare Worker 를 작성하라. 날씨 API 와 캘린더 API 를 호출하고 (두 토큰 다 비밀로

제공하겠다), 한 줄 요약을 만들어, Telegram Bot API 로 POST 한다. 각 외부 호출을 try/catch 로 감싸서 한 번의 실패가 메시지를 죽이지 않게. 모든 토큰을 env 에서 읽어라.

오전 7 시를 기다리지 않고 문구를 반복 개선하고 싶었을 때:

수동 트리거를 추가하라: Worker 가 GET 요청과 ?test=1 쿼리 파라미터로 호출되면, 같은 로직을 즉시 돌리고 메시지를 보내는 대신 그 텍스트를 응답으로 반환하라. 나만 트리거할 수 있도록 이걸 비밀 토큰 뒤에 두어라.

그 테스트 트리거가 우리를 잔혹한 피드백 루프에서 구해줬다—하루에 한 번이 아니라 원할 때 출력을 볼 수 있었다. 이것은 훔칠 가치가 있는 일반적인 수다: 코드가 스케줄로 돌아갈 때면 언제든, 시계를 기다리기로 약속하기 전에 지금 온 디맨드로 돌릴 방법을 스스로 만들어 두어라.

장애물

텔레그램 메시지는 도착했지만, 시간이 틀렸다: 일정이 한국 시간이 아니라 UTC 로 표시됐다. 날씨는 괜찮았다. 우리는 그것을 분리했다:

캘린더 시간이 9 시간 어긋난다 — Asia/Seoul 이 아니라 UTC 를 표시한다. 날씨 시간 (" 오후 4 시 이후 비") 은 정확하다. 둘 다 이렇게 포맷한다: [붙여넣음]. 캘린더 포맷팅만 고치고 날씨 경로는 왜 이미 맞았는지 알려달라.

AI 는 날씨 API(애플리케이션 프로그래밍 인터페이스—한 서비스가 다른 프로그램이 자기 데이터를 가져갈 수 있도록 열어두는 출입구) 가 이미 현지화된 문자열을 반환하는 반면, 캘린더 API 는 우리가 그대로 출력하던 UTC 타임스탬프를 반환한다고 설명했다. 캘린더 포맷팅 단계에 단일 시간대 변환을 추가했다. 수정을“캘린더 경로만”으로 범위 좁힌 것이 AI 가 멀쩡히 동작하던 날씨 코드를 다시 쓰는 것을 막았다—AI 수정이 멀쩡하던 것을 깨뜨리는 흔한 방식이다. 그 프롬프트의 후반부—“날씨 경로는 왜 이미 맞았는지 알려달라”—는 두 가지 일을 했다: 추측이 아니라 실제 근본 원인을 알려줬고, AI 가 그 차이를 분명히 표현하게 강제해서 그 수정이 두 경로를 하나의 깨진 것으로 실수로“통합”하지 않게 했다.

출시

여기서“출시”는 그저 Worker 를 배포하고, cron 스케줄을 설정하고, API 토큰을 암호화된 비밀로 추가하는 것을 뜻했다. 우리는 ?test=1 엔드포인트를 몇 번 쳐서 메시지가 잘 읽히는지 확인한 뒤, 그대로 두었다. 그 이후로 매일 아침 돌아가고 있다—그리고 우아한 성능 저하 줄은 한 달 뒤 날씨 API 에 장애가 났을 때 그 값어치를 증명했는데, 메시지는 여전히 도착했고 단지 예보만 빠졌을 뿐이었다.

사례 연구 4: 손 갈 필요 없는 콘텐츠 사이트

아이디어

한 취미가가 마크다운 노트 폴더—수년간 여행하며 쓴 식당 리뷰—를 검색 가능한 공개 사이트로 바꾸고 싶어 했다. 로그인할 CMS 도 없고 월 청구서도 없이. 노트를 추가하고, 푸시하고, 끝.

스펙

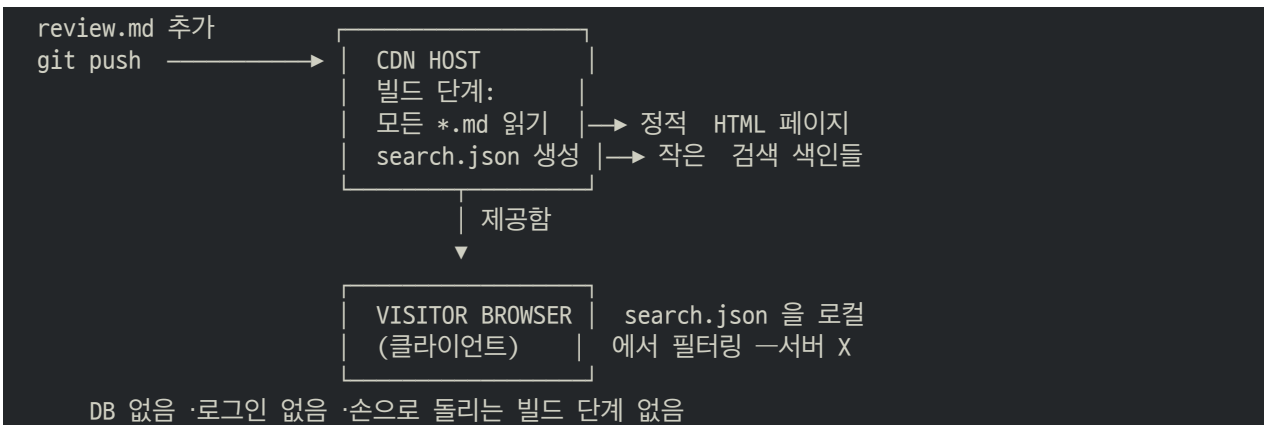
마크다운 파일 폴더에서 생성되는 정적 사이트, 리뷰당 파일 하나 (프론트매터에 title, city, rating, body). 리뷰를 최신순으로 나열하는 홈페이지, 리뷰당 페이지 하나, 그리고 도시나 이름으로 하는 클라이언트 사이드 검색을 만들어라. 데이터베이스 없음, 로그인 없음, 내가 손으로 돌려야 하는 빌드 단계 없음. 새 .md 파일을 푸시하면 그게 게시되어야 한다.

숨은 요구사항은 마지막 문장에 있다. “내가 손으로 돌려야 하는 빌드 단계 없음”은 스펙이 사실 출력만이 아니라 워크플로에 관한 것임을 뜻한다—그리고 워크플로에 요구사항은 명명하지 않으면 AI가 건너뛰는 부류인데, 돌아가는 앱에서는 드러나지 않기 때문이다.

스택

빌드 타임에 마크다운을 읽는 정적 사이트 생성기를, Git 트리거 빌드를 갖춘 CDN 호스트에 배포했다. 저장소에 푸시하면 호스트가 다시 빌드하고, CDN이 서빙한다. 검색은 빌드 시 생성된 작은 JSON 인덱스 위에서 클라이언트 사이드로 하므로, 서버도 없고 쿼리 비용도 없다—리뷰 수백 개에서는 괜찮은 트레이드오프이자, 발견한 게 아니라 우리가 명명한 의도적 선택이다.

전체 워크플로는 화살표 하나다: 푸시가 게시하는 빌드를 트리거한다. 방문자 자신의 브라우저가 인덱스를 필터하는 것 말고는 요청 시점에 아무것도 돌지 않는다:



핵심 프롬프트

우리는 어떤 UI 보다 먼저 AI를 데이터 형태에 맞 내렸다:

```
/reviews 안의 모든 .md 파일을 읽는 정적 사이트를 세팅하라. 각  
파일의 프론트매터에 title, city, rating(1-5), date 가 있다. 빌드  
타임에 그것들을 전부 파싱해 정렬된 목록 (최신순) 으로 만들고  
생성하라: 그것들을 나열하는 홈페이지, 그리고 /reviews/<slug> 에  
리뷰당 페이지 하나. 어떤 스타일링보다 먼저 데이터 로딩 코드를  
보여달라.
```

“어떤 스타일링보다 먼저”는 의도적인 레버다. 한꺼번에 다 요청하면 신뢰할 수 없는 데이터에 연결된 아름다운 페이지를 얻는다; 데이터 레이어를 먼저 요청하면 단 한 픽셀이 주의를 흐뜨리기 전에 토대를 검증할 수 있다. 그런 다음 싸게 유지하도록 범위를 좁힌 검색:

```
빌드 타임에 모든 리뷰에 대한 {title, city, slug} 로 search.json 을  
생성하라. 홈페이지에서 검색 박스를 추가해 보이는 목록을  
클라이언트 사이드로 city 나 title 매칭으로 필터하라 —네트워크  
호출 없이, 이미 로드된 인덱스를 그냥 필터하라. JS 50 줄 아래로  
유지하라.
```

장애물

빌드는 로컬에서 통과했지만 배포된 사이트는 개별 리뷰 페이지마다 404를 냈다. 홈페이지는 동작했고; 리뷰별 페이지는 아니었다. 우리는 이론화하기보다 증상과 설정을 붙여 넣었다:

```
홈페이지는 라이브에서 동작하는데, /reviews/<slug> 페이지들은  
로컬 dev에서는 동작하면서 프로덕션에서는 전부 404가 난다. 여기
```

내 빌드 출력 디렉터리 목록과 호스트의 배포 설정이다: [붙여넣음].
왜 로컬 dev 는 이 페이지들을 서빙하는데 프로덕션은 안 하나?

AI 는 그것을 dev 서버 라우팅 (경로를 동적으로 해석) 과 정적 호스팅 (물리적으로 존재하는 파일만 서빙) 의 차이로 추적했다. 빌드가 리뷰 페이지들을 잘못된 출력 폴더로 생성하고 있어서, 그것들은 절대 업로드되지 않았다 — dev 가 프로덕션이 못 하는 라우트를 라우터가 가짜로 만들어줘서 버그를 숨겼던 것이다.

우리는 출력 디렉터리를 호스트가 실제로 배포하는 것에 맞췄고, 페이지들이 나타났으며, 출시 체크리스트에 한 줄을 추가했다: 홈페이지뿐 아니라 라이브 사이트에서 딥 링크를 클릭하라. 더 넓은 교훈은 “dev 에서 동작함”과 “배포되어 동작함”은 다른 주장이고, 정적 익스포트가 바로 그 둘이 갈리는 지점이라는 것이다—dev 서버는 실제 파일 기반 호스팅이 결코 그렇지 않은 방식으로 라우트에 너그럽다.

출시

우리는 진짜 리뷰를 하나 푸시하고, 호스트가 다시 빌드하는 것을 지켜보고, 그 특정 리뷰로 가는 딥 링크를 라이브 도메인에서 로드했다—dev 가 거짓말했던 바로 그 경우. 그런 다음 인덱스가 그것을 집어 들었는지 확인하려 그 도시를 검색했다. 다음 리뷰를 추가하는 것은 한 줄짜리 커밋이었는데, 그게 핵심 전부였다: 출시는 한 순간이 아니라, 우리 없이도 계속 동작하는 워크플로였다.

넷의 공통점

- **스펙이 먼저였고, 짧았다.** 문서가 아니라 한 문단. 그러나 중요한 제약을 명명했다—언어, “자기 데이터만 본다”, “가진 것이라도 보내라”, “파일을 푸시하면 게시되어야 한다”—돌아가는 앱에서는 결코 드러나지 않는 워크플로 제약까지 포함해서.
- **비밀은 저장소를 절대 건드리지 않았다.** 모든 프로젝트가 코드가 아니라 대시보드에 설정된 환경 변수나 암호화된 비밀에서 키를 읽었다. 우리는 “절대 하드코딩하지 말라”를 소리 내어 말했는데, 기본값이 그 반대이기 때문이다.
- **스택은 잘못될 게 더 적도록 골랐다.** 매니지드 auth, 매니지드 데이터베이스, 정적 호스팅, 서버리스 함수—모든 선택이 AI 가 우리를 대신해 깨뜨릴 수 있는 코드의 한 부류와, 우리가 유지보수해야 할 것의 한 부류를 제거했다.
- **장애물은 AI 에게 정확한 증거를 먹임으로써 해결됐다**—문자 그대로의 에러, 실제 코드, 배포 설정, 진짜 증상—그리고 동작하는 코드가 계속 동작하도록 수정의 범위를 좁게 잡음으로써. 두 번이나 증상은 코드에 있었지만 원인은 인프라에 있었다.
- **출시는 현실에 대한 진짜 테스트를 뜻했다:** 라이브 구매 (그리고 거절된 구매), 두 번째 사용자 계정, 온디맨드 트리거, 배포된 사이트의 딥 링크. “컴파일됐다”가 아니라.

시작하는 데 거창한 아이디어는 필요 없다. 작은 아이디어 하나, 한 문단짜리 스펙, 그리고 타이핑하기보다 지시하려는 의지가 필요하다—그리고 끝났다고 부르기 전에 그것을 현실에 대고 시험할 인내가.

정리와 연습

핵심 요점

- 스펙이 먼저였고 짧았다—실행 중인 앱에는 결코 드러나지 않는 워크플로 제약을 포함해, 중요한 제약을 짚은 한 단락이었다.
- 비밀값은 저장소에 닿지 않았다. 모든 프로젝트가 환경 변수나 대시보드에 설정한 암호화된 설정에서 키를 읽었다.
- 스택은 잘못될 여지가 적도록 선택됐다—관리형 인증, 관리형 데이터베이스, 정적 호스팅, 서버리스 함수.
- 장애물은 AI 에게 정확한 증거를 먹여 해결됐다—실제 에러, 실제 코드, 배포 설정—그리고 수정 범위를 좁게 잡았다.
- 출시는 “컴파일됨”이 아니라 현실에 대한 진짜 테스트를 뜻했다: 실제 결제, 두 번째 계정, 배포된 사이트의 딥 링크.

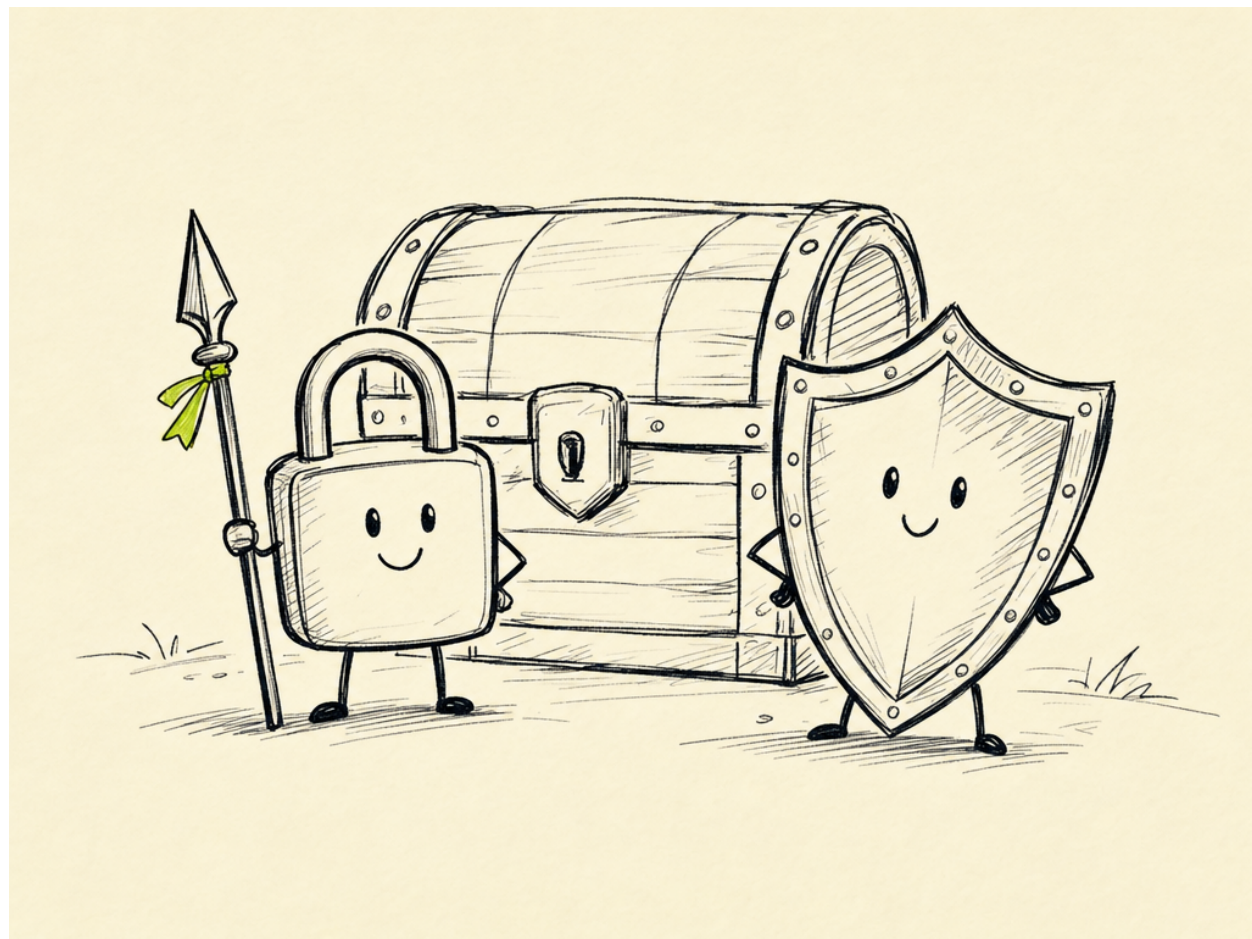
직접 해보기

자신의 아이디어 하나를 골라, 이 네 사례처럼 한 단락짜리 스펙을 써라: 핵심 동작, 각 사용자가 볼 수 있는 데이터, 그리고 최소한 하나의 워크플로 제약 (콘텐츠가 어떻게 게시되는지, 누가 무엇을 할 수 있는지) 을 명시하라. 그런 다음 망가질 커스텀 코드를 줄이기 위해 기댓 관리형 빌딩 블록을 나열하라. 한 단락에 짧은 글머리표 목록 정도로 유지하라—그보다 커지면 과하게 명세하고 있는 것이다.

이 장의 프롬프트

```
I want to build: [one-paragraph description of the app].
Before any code, help me tighten this into a buildable spec.
1. Restate the single core action a user performs.
2. List the data each user can see — and what they must NOT see.
3. Name the workflow constraints that won't show up in the UI
   (e.g. "pushing a file should publish it", "only their own data").
4. Recommend a stack that MINIMIZES custom code: managed auth,
   managed database, static or serverless hosting where possible.
5. Flag the one part most likely to go wrong, and how we'll test it
   against reality (a real purchase, a second account, a deep link).
Keep the spec to a paragraph plus bullets. Push back if it's too big.
```

바이브 코더를 위한 보안



보안은 바이트 코딩에서 조용히 물어뜯는 부분이다. 망가진 기능은 즉시 드러난다—버튼이 작동하지 않으면 고치면 된다. 보안 구멍은 결코 드러나지 않다가, 누군가 그것을 발견해서 당신 사용자들의 데이터를 어느 포럼에 뿌리는 날이 오면 그제야 드러난다. 그때쯤이면 그 버그를 쓴 AI 는 이미 사라진 지 오래고, 천 명에게 왜 자기 비밀번호가 새어나갔는지 설명하는 사람은 당신이다.

이 장은 공포에 관한 것이 아니다. 안전한 작은 앱을 출시하려고 침투 테스터가 될 필요도, OWASP Top 10 을 외울 필요도 없다. 소프트웨어가 뚫리는 몇 가지 방식을 이해하고, AI 가 계속 만들어내는 패턴을 알아보고, 출시 전에 리뷰 게이트 하나를 추가하면 된다. 철학은 이 책의 나머지와 같다: **이해하지 못한 것은 절대 출시하지 말라**—그리고 보안은 이해하지 못하는 것의 대가가 가장 큰 곳이다.

AI 가 생성한 코드가 기본적으로 안전하지 않은 이유

AI 는 당신이 준 프롬프트에 최적화하는데, 당신의 프롬프트는 거의 항상 “작동하게 만들어”라고 말하지 “안전하게 만들어”라고는 절대 말하지 않는다. 그래서 AI 는 작동하는 기능에 이르는 가장 짧은 경로를 만들어낸다. 가장 짧은 경로는 대개 안전하지 않은 쪽이다: 문자열로 이어붙인 쿼리, 권한 점검이 없는 엔드포인트 (앱으로 들어가는 주소가 매겨진 문 가운데 하나—앱이 요청에 응답하는 단일 URL), 학습한 예제가 그랬다는 이유로 인라인에 붙여넣은 비밀번호.

더 나쁜 건, AI 가 인터넷 전체로부터 배웠다는 점이다—그 자체로 안전하지 않았던 10 년 치 튜토리얼과 버려진 저장소들을 포함해서. 안전하지 않은 코드가 학습 데이터에서 안전한 코드보다 더 흔한데, 안전한 코드는 쓰기가 더 어렵고 올리기가 더 드물기 때문이다. “로그인 폼” 하나를 요청하면, 통계적으로 인터넷의 평균적인 로그인 폼을 받을 가능성이 높고, 인터넷의 평균적인 로그인 폼에는 문제가 있다.

AI 에게는 위협 감각도 없다. 파일 업로드를 만드는 인간은 잠깐 걱정의 떨림을 느낀다—누가 고약한 걸 올리면 어찌지? AI 는 아무것도 느끼지 않는다. 빈틈없든 활짝 열려있든 똑같은 자신감으로 핸들러를 작성한다. 망설임도, “음, 이 부분이 좀 불안한데”같은 것도 없다. 그 불안함은 당신이 직접 공급해야 한다.

핵심은 “AI 코드는 위험하니 쓰지 말라”가 아니다. **“작동한다”와 “안전하다”는 별개의 두 질문**이라는 것, 그리고 두 번째에 답하도록 강제하지 않는 한 AI 는 첫 번째에만 답한다는 것이다.

인젝션: 입력이 코드가 될 때

인젝션은 가장 오래됐으면서 여전히 가장 흔한 심각한 버그다. 사용자로부터 온 데이터가 데이터가 아니라 명령으로 취급될 때 발생한다. 대부분의 앱에서는 두 가지 종류가 중요하다.

SQL injection —SQL(스트럭처드 쿼리 랭귀지) 은 앱이 데이터베이스에 데이터를 요청할 때 쓰는 언어다—은 사용자 입력이 데이터베이스 쿼리 안으로 미끄러져 들어갈 때 일어난다. 전형적인 취약 패턴:

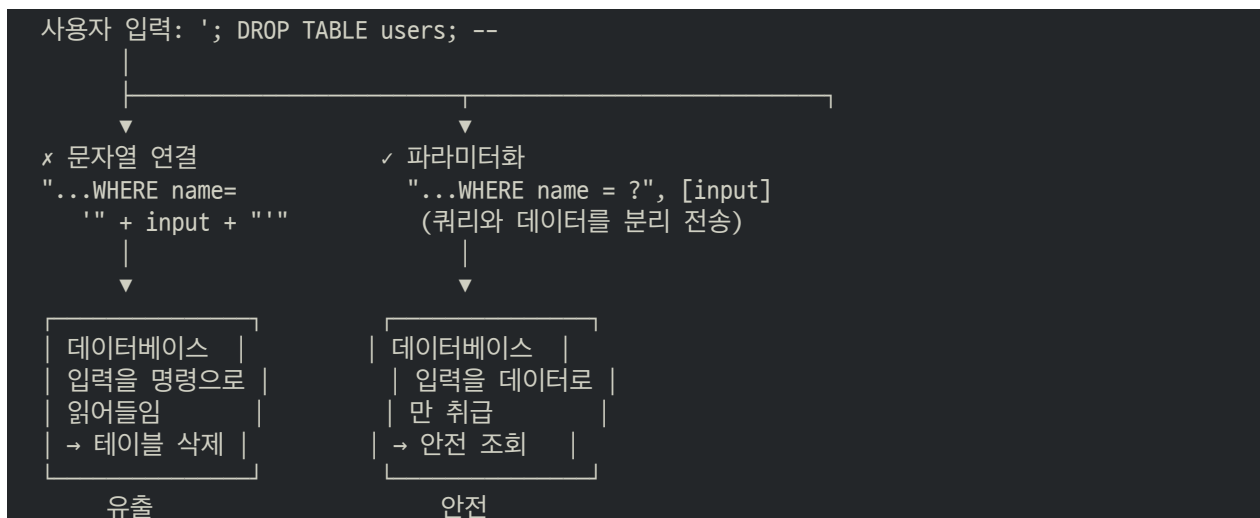
```
// VULNERABLE: 사용자의 입력이 쿼리에 그대로 붙어버린다
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// 누가 name = '; DROP TABLE users; -- 를 넘기면
// 당신의 쿼리는 테이블을 삭제하라는 명령이 된다.
```

해결책은 **parameterized query**(prepared statement 라고도 한다) 다. 쿼리와 데이터를 따로 보내서 데이터베이스가 둘을 결코 혼동하지 않게 한다:

```
// SAFE: 값은 파라미터로 전달되고, 절대 코드로 취급되지 않는다
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

규칙: **문자열 연결로 쿼리를 만들지 말라**. diff 에서 변수 주위로 SQL 을 조립하는 백틱이나 + 가 보이면, 멈추고 parameterized 버전을 요청하라.

두 경로는 같은 사용자 입력을 데이터베이스에 넣지만, 데이터베이스가 그 입력을 명령으로 착각하지 않게 하는 것은 둘 중 하나뿐이다:



XSS (cross-site scripting) 는 브라우저에서 일어나는 같은 발상이다. 사용자 입력을 받아 페이지에 raw HTML(하이퍼텍스트 마크업 언어, 브라우저에 무엇을 보여줄지 알려주는 코드)로 떨어뜨리면, 공격자는 당신의 다른 사용자들의 브라우저에서 실행되는 `<script>` 태그를 주입할 수 있다—예를 들어 그들의 세션을 훔쳐서. 해결책은 이스케이핑이다: 사용자 콘텐츠를 HTML 이 아니라 텍스트로 렌더링하라. React 같은 현대 프레임워크는 기본적으로 이스케이프해 주는데, 이건 훌륭하다—AI 가 “작동하게 만들려고” `dangerouslySetInnerHTML` 이나 `innerHTML` 같은 탈출구에 손을 뻗기 전까지는. 그것들은 보호를 우회한다. diff 에서 그런 호출이 보이면 의심해야 할 대상으로 취급하라.

모든 인젝션 버그 뒤에 있는 통합 원리: **데이터와 코드를 분리하라**. 사용자 입력이 쿼리, 명령, 템플릿, 혹은 HTML 로 넘어갈 때마다, 무언가가 그것을 이스케이프하거나 `parameterize` 해야 한다.

비밀값과 API 키: 노출된 키의 함정

비밀값은 접근을 허용하는 모든 것이다: API(애플리케이션 프로그래밍 인터페이스) 키, 데이터베이스 비밀번호, 결제 제공자 토큰, 서명 키. 가장 흔하면서—또 가장 비싼—바이브 코딩 실수는 그중 하나를 새어나가게 하는 것이다.

사람들이 끊임없이 걸러드는 두 가지 함정:

- **클라이언트 코드 안의 비밀값**. 당신의 프론트엔드 (브라우저에서 실행되는 JavaScript) 에 있는 것은 무엇이든 공개다. 사용자는 개발자 도구를 열어 그것을 읽을 수 있다. AI 에게 React 컴포넌트에서 API 를 호출하라고 하면, 기꺼이 당신의 비밀 키를 거기 바로 붙여넣는다—그러면 이제 당신 사이트를 방문하는 누구나 그걸 복사해 당신의 청구서를 불러놓을 수 있다. 비밀 키는 서버에 있어야지, 브라우저로 배포되는 코드에는 절대 안 된다.
- **저장소 안의 비밀값**. 파일에 하드코딩된 키는 git 에 커밋된다. 나중에 지워도 git 히스토리에 영원히 남고, 봇들은 푸시 후 몇 분 안에 정확히 이걸 노리고 공개 저장소를 스캔한다.

비밀값의 올바른 거처는 **환경 변수 (environment variable)** 다—런타임에 앱에 공급되는 값으로, 코드에서는 완전히 빠져 있다:

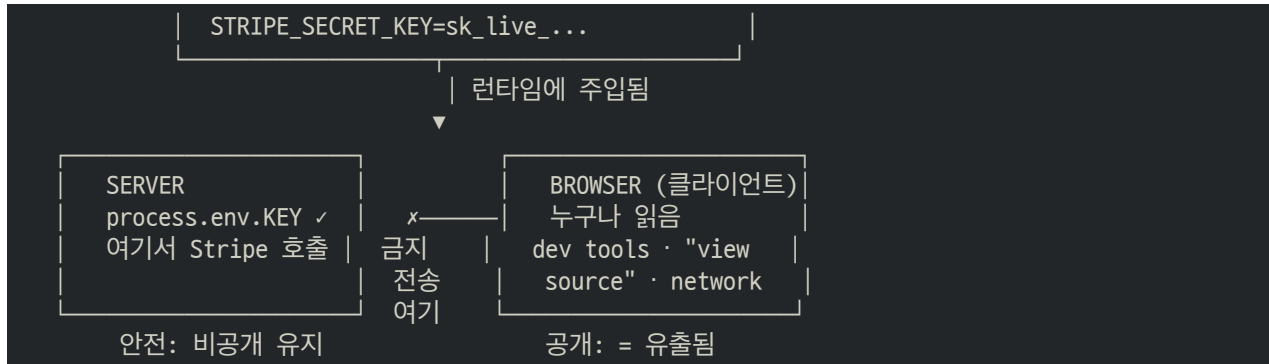
```

// VULNERABLE: 키가 소스에 들어 있고, git 에 커밋될 것이다
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SAFE: 키는 환경에서 읽어오고, 코드에는 절대 쓰이지 않는다
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
  
```

비밀값에는 안전한 거처가 정확히 하나 있다—서버 측에서, 환경에서 읽어오는 것. 브라우저는 공개된 장소다: 거기서 배포된 것은 무엇이든 개발자 도구를 여는 누구라도 읽을 수 있다.

`.env` (gitignore 됨, 커밋 안 함)



비밀값을 한 줄이라도 쓰기 전에 .env(그리고 .env.local 등) 를 .gitignore 에 추가하라. 키가 한 번이라도 당신의 코드나 히스토리에 들어갔다면, 타버린 것으로 취급하라: 로테이션하라 (새 것을 생성하고 옛것을 폐기하라) —줄을 지우는 것만으로는 부족한데, 옛 키는 여전히 유효하고 여전히 바깥에 떠돌기 때문이다.

인증 vs 인가: 누구나 호출할 수 있는 엔드포인트

이 두 단어는 비슷하게 들리고, 진짜 침해는 그 차이 속에 산다.

- **인증 (Authentication)** 은 당신은 누구인가? 다—로그인, 신원 증명.
- **인가 (Authorization)** 는 당신은 무엇을 할 수 있는가? 다—이 로그인한 사용자가 이 데이터에 이 동작을 수행해도 되는지.

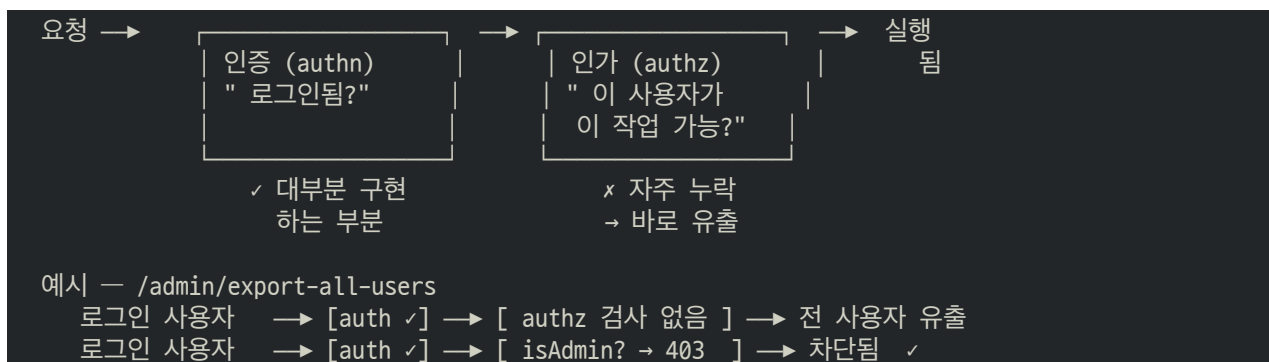
AI 는 인증은 꽤 잘한다; 라이브러리가 대부분을 처리한다. 인가는 AI 가 일상적으로 실패하는 곳인데, 인가는 당신 앱의 규칙에 특화되어 있고 AI 는 그 규칙을 모르기 때문이다. 교과서적인 재앙:

```
// VULNERABLE: 로그인했는지는 점검하지만, 당신이 누구인지는 점검하지 않는다
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // 로그인한 사용자라면 누구나 이걸 칠 수 있다
});
```

그 엔드포인트는“보호”되어 있다—로그인해야만 한다. 하지만 30 초 전에 가입한 사람을 포함해 어떤 로그인 사용자든 그것을 호출해 모든 사용자의 데이터를 내려받을 수 있다. 인증은 있고, 인가는 없다. 해결책은 동작 그 자체에 권한을 점검하는 것이다:

```
// SAFE: 이 사용자가 실제로 admin 인지 확인한다
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

요청이 통과해야 하는 두 개의 게이트를 생각해 보라. 인증은“로그인했나?”를 묻고, 인가는“당신이 이것을 해도 되나?”를 묻는다. 침해는 앱이 첫 번째 게이트를 만들고 두 번째를 잊을 때 일어난다:



같은 함정이 도처에서 축소판으로 나타난다: 주문 #1234 를 반환하면서 그 주문이 요청한 사용자의 것인지 점검하지 않는 엔드포인트. 누구나 URL 의 숫자를 바꿔서 남의 주문을 읽을 수 있다. 규칙은 화려하지 않고 절대적이다: **데이터를 다루는 모든 엔드포인트에서 인가를 점검하고, 클라이언트가 보낸 ID 를 신뢰하지 말라.** 숨겨진 URL 이 숨겨져 있다는 이유로 안전하다고 가정하지 말라—“아무도 이게 존재하는 줄 모른다”는 보안 통제가 아니다.

파일 처리와 업로드

사용자가 파일을 올리게 하는 것은 조용한 지뢰밭이고, AI 의 첫 초안은 거의 그 구멍들을 닫지 않는다:

- **타입과 크기를 검증하라.** 제한이 없으면 누가 10 GB 파일이나 실행 파일을 올린다. 파일명 확장자가 아니라 실제 내용을 점검하라—확장자는 누구나 바꿔 달 수 있다.
- **파일명을 절대 신뢰하지 말라.** ../../etc/passwd 같은 파일명은 순진한 핸들러가 의도한 폴더 밖에 쓰게 만들 수 있다 (이것을 path traversal 이라 한다). 제공된 이름을 쓰는 대신 당신만의 안전한 이름을 생성하라.
- **업로드를 당신 앱과 같은 origin 에서 서빙하지 말고,** 업로드된 파일이 코드로 실행되게 두지 말라. 당신 서버가 실행할 폴더에 떨어진 사용자 제공 파일은 장악으로 가는 직통로다.

이걸 전부 직접 만들 필요는 없다—스토리지 서비스가 대부분을 처리한다—하지만 요청은 해야 한다, AI 는 자진해서 하지 않으니까. “업로드 기능을 추가해”라고 하면 정상 경로를 얻는다. “파일 타입과 크기를 검증하고, 이미지가 아니면 거부하고, 생성된 이름으로 파일을 저장하는 업로드 기능을 추가해”라고 하면 안전에 더 가까운 무언가를 얻는다.

의존성 위험: AI 가 지어낸 패키지

현대 앱은 수십 개의 서드파티 패키지에 기댄다, 그리고 AI 는 그것들을 거리낌 없이 제안한다. 두 가지 위험이 따라온다.

첫째, **타이포스쿼팅과 환각 패키지.** 공격자들은 진짜와 머리카락 한 올 차이 나는 이름의 악성 패키지를 게시해서 (requests 대신 requests) 오타에 베팅한다. AI 는 때때로 존재하지도 않는 패키지를 자신만만하게 import 하는데—그 습관을 눈치챈 공격자는 그 정확한 이름을 멀웨어를 넣어 등록할 수 있다. AI 가 제안하는 무언가를 설치하기 전에 한번 살펴보자: 실제로 존재하는가, 진짜 다운로드 수와 저장소가 있는가, 이름은 당신이 예상하는 철자대로인가?

둘째, **검증되지 않은 의존성 일반.** 당신이 추가하는 모든 패키지는 당신 앱의 모든 접근 권한으로 실행되는 코드다. 의존성이 많다는 건 버그와 공급망 공격 양쪽 모두에 대한 표면이 더 넓어진다는 뜻이다. 무명의 긴 꼬리보다 적고 잘 알려진 라이브러리를 선호하고, 하나를 추가하기 전에 “이것에 정말 패키지가 필요한가, 아니면 열 줄짜리인가?”를 물어라.

당신 생태계의 audit 도구를 주기적으로 돌려라 (npm audit, pip-audit 등등) —이미 설치한 것들에서 알려진 취약점을 표시해주고, AI 는 그것이 보고한 것을 잘 고친다.

보안 리뷰 게이트

위의 모든 것을 걱정거리에서 프로세스로 바꾸는 습관 하나가 여기 있다: 출시 전에, **AI 가 자기 코드를 공격하게 만들라.** 기능을 쓴 모델은 보통 그 안의 구멍을 찾아낼 수 있다—다만 시키지 않으면 안 할 뿐이다. 빌더에서 공격자로 뒤집어라:

너는 이 엔드포인트를 작성했다. 이제 그것을 깨뜨리려는 공격자처럼 행동하라.

악의적인 사용자가 다음을 할 수 있는 모든 방법을 나열하라:

- 자기가 봐서는 안 되는 데이터를 읽거나 수정 (인가 구멍)
- 입력을 통해 코드 주입 (SQL injection, XSS, command injection)
- 빠진 검증이나 rate limit을 악용

각각에 대해 그것을 익스플로잇하는 정확한 요청을 보여주고, 그다음 해결책을 보여라.

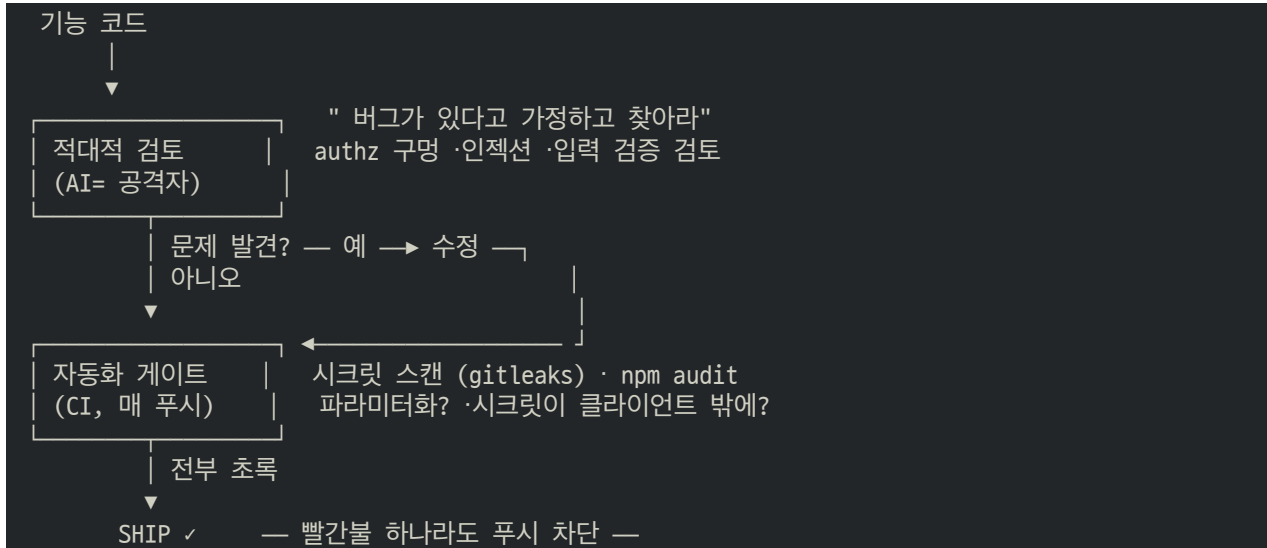
나를 안심시키지 말라 — 취약점이 있다고 가정하고 그것을 찾아라.

그 마지막 줄이 중요하다: 중립으로 두면 AI 는 “안전해 보인다!”라고 말하는 경향이 있다. 결함이 존재한다고 가정하라고 시키면, 실제로 찾으려 나선다. 그 적대적 패스를, 사용자에게 노출되는 무엇에든 돌리는 짧은 출시 전 체크리스트와 짝지어라:

- ☐ 모든 엔드포인트가 사용자가 로그인했는지뿐 아니라 인가를 점검한다
- ☐ 모든 데이터베이스 쿼리가 parameterized 다—문자열로 만든 SQL 없음
- ☐ 페이지에 렌더링되는 사용자 입력이 이스케이프된다 (raw HTML 주입 없음)
- ☐ 클라이언트 코드에 비밀값 없음, 저장소에 커밋된 것도 없음

- .env 가 gitignore 되어 있음; 새어나간 키는 모두 로테이션됨
- 파일 업로드가 타입과 크기를 검증하고 생성된 이름을 사용한다
- 새 의존성이 실제 존재 여부와 평판을 눈으로 확인받았다

통과하지 않으면 아무것도 출시되지 않는 시퀀스로 게이트를 그려보라. AI 가 빌더에서 공격자로 모자를 바꿔 쓰고, 그 다음 자동 스캔이 사람의 패스가 놓칠 수 있는 것을 받쳐준다:



그리고 푸시하기 전에 **secret scanner** 를 돌려라—gitleaks 같은 도구 (혹은 당신 플랫폼의 내장 스캐닝) 는 당신의 코드와 히스토리에서 키처럼 생긴 것들을 grep 한다. 이 목록에서 가장 비싼 실수에 대한 명령어 하나짜리 안전망이고, AI 를 시켜 CI(컨티뉴어스 인테그레이션—당신이 코드를 푸시할 때마다 자동으로 점검을 돌려주는 로봇) 에 엮어 넣으면 매번 푸시할 때마다 돌게 할 수 있다.

자동화 안일, 보안 편

가장 위험한 순간은 첫째 날이 아니다—아흔째 날이다. 첫째 날에는 신중하다; 모든 diff 를 읽고, 공격자 프롬프트를 돌리고, 업로드 핸들러를 점검한다. 그러다 백 개의 변경이 출시되고 나쁜 일이 아무것도 일어나지 않으면, 점검이 의례처럼 느껴지기 시작한다. 작은 변경에 “이번 한 번만” 리뷰 게이트를 건너뛴다. 그 작은 변경이 인가 점검 없는 엔드포인트를 추가한다. 한동안은 아무 일도 일어나지 않는데, 아무 일도 일어나지 않는 것이야말로 보안 구멍이 안에서 보이는 모습 그 자체이기 때문이다.

이것이 보안을 거는 자동화 안일이고, 다른 어디보다 여기서 더 나쁜데, 보안 실패는 조용하고 지연되기 때문이다. 망가진 기능은 몇 분 안에 당신을 벌한다. 망가진 권한 점검은 침해가 일어나기 직전까지 걸보기 성공으로 당신을 보상한다. 재앙의 부재는 안전의 증거가 아니다—그저 아직 발견되지 않았을 뿐인 취약한 앱의 정상 상태다.

방어책은 게이트를 충분히 값싸게 만들어 건너뛰지 않게 하고, 충분히 자동으로 만들어 건너뛴 수 없게 하는 것이다. secret scan 과 audit 을 CI 에 넣어 당신의 기억 없이도 돌게 하고, auth, 데이터, 업로드를 건드리는 무엇에든 공격자 프롬프트를 가까이 두라. 한 줄을 지켜라: 사용자 데이터나 권한을 다루는 변경은 누군가가—당신이, AI 의 정직한 도움을 받아—그것을 적극적으로 깨뜨려보기 전까지는 출시되지 않는다. 바이트 코딩에서의 속도는 좋은 게이트에서 나오지, 조용함이 모든 게 괜찮다는 뜻이라고 믿는 데서 나오지 않는다.

정리와 연습

핵심 요점

- AI 가 생성한 코드는 기본적으로 안전하지 않다. 깨끗해 보이는 첫인상을 믿지 말고, 취약점이 있다고 가정하고 찾아 나서라.
- 큰 범주는 반복된다: 인젝션, 노출된 비밀값, 누락된 인가, 안전하지 않은 파일 업로드, 그리고 지어낸 의존성.

- 인가는 로그인 단위가 아니라 요청 단위다—모든 엔드포인트는 단지 로그인했는지가 아니라 이 사용자가 이 동작을 해도 되는지 확인해야 한다.
- 푸시할 때마다 gitLeaks 같은 비밀값 스캐너를 돌리고, 한 번이라도 유출된 키는 교체하라. 건너뛴 수 없도록 스캔을 CI에 엮어라.
- 90 일째가 첫날보다 위험하다—보안 게이트를 건너뛰지 않을 만큼 싸게, 건너뛴 수 없을 만큼 자동으로 만들어라.

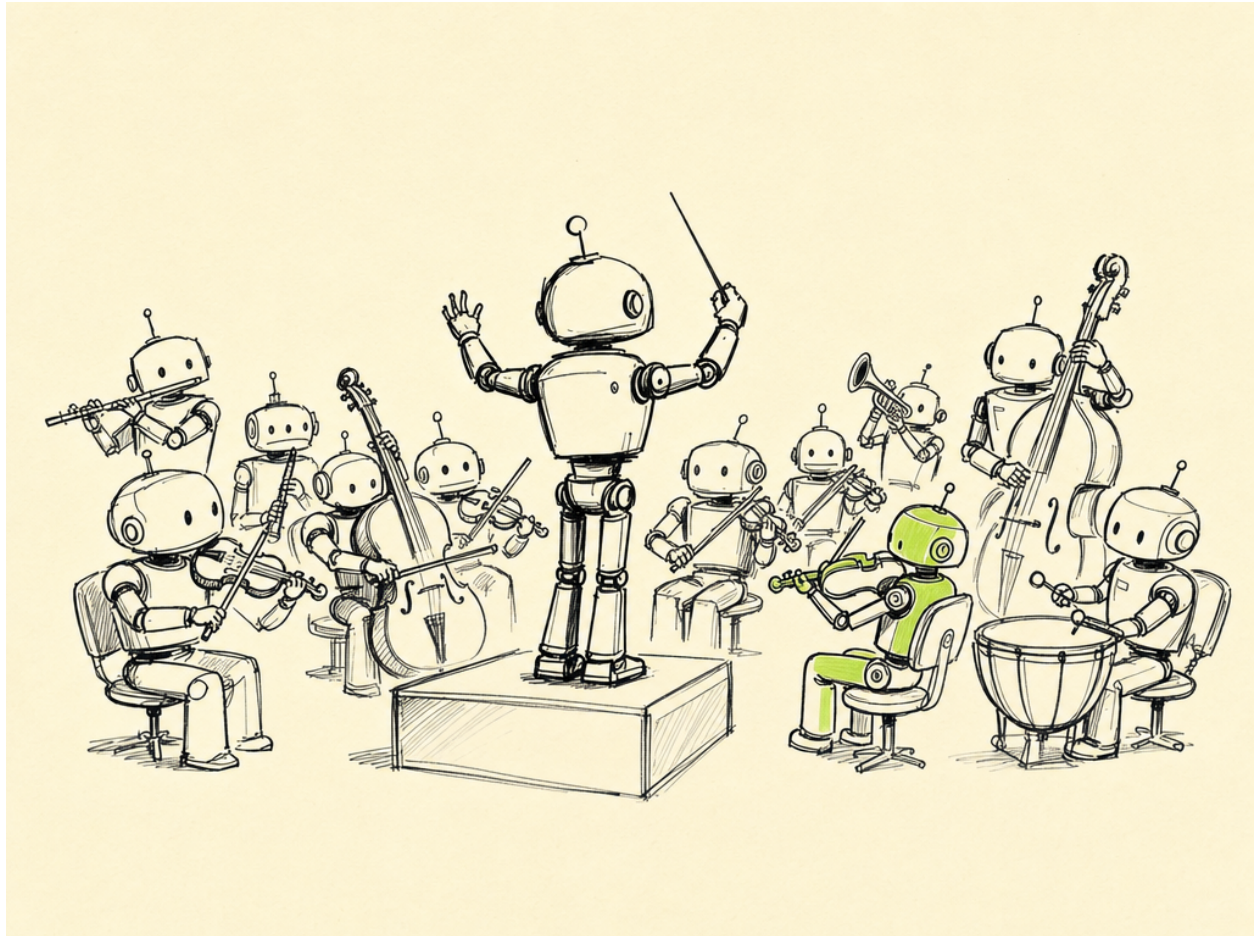
직접 해보기

인증, 사용자 데이터, 업로드를 다루는 자신의 기능 하나를 골라 아래의 적대적 리뷰 프롬프트를 돌려라. AI가 보고하는 취약점마다 그것을 악용하는 정확한 요청과 최소 수정을 요청하고, 수정을 적용한 뒤 프롬프트를 다시 돌려라. 마지막으로 저장소와 히스토리 전체에 gitLeaks(또는 플랫폼의 비밀값 스캔)를 실행하며 마무리하라.

이 장의 프롬프트

```
Here is the code for a feature that handles [auth / user data / uploads]:
[paste the code].
Act as an attacker reviewing this for security holes. Specifically check:
- broken authorization (can a user act on another user's data?)
- injection via input (SQL injection, XSS, command injection)
- exposed secrets or keys in code or responses
- unsafe file handling (type, size, path, generated names)
- missing validation or rate limits
For EACH issue, show the exact request that exploits it, then the fix.
Do NOT reassure me — assume there IS a vulnerability and find it.
```


에이전트 오케스트레이션



이 책의 대부분에서 AI 는 당신이 손에 쥐고 있는 도구였다. 당신이 묻고, AI 가 답하고, 당신이 결과를 확인한다. 대부분의 경우 이것이 여전히 올바른 사고 모델이다. 하지만 도구들은 두 번째 모드를 갖게 되었다—단계 대신 목표를 넘겨주면, AI 가 당신에게 돌아오기 전까지 몇 분 (때로는 더 길게) 동안 일하는 모드다.

이 장은 그 전환에 관한 것이고, 그 전환이 일어날 때 핵심을 놓치지 않는 법에 관한 것이다. “에이전트”를 “이제 신경 끄어도 된다”로 읽고 싶은 유혹이 있다. 그게 바로 사람들을 곤경에 빠뜨리는 해석이다. 에이전트는 당신이 이미 이해하고 있는 일에 대한 지렛대일 뿐—당신이 더 이상 들여다보지 않게 된 코드를 출시해도 좋다는 허가증이 아니다.

세 가지 계층

AI 보조를 세 가지 계층으로 보면 도움이 된다. 각 계층은 당신에게 조금씩 더 많은 고삐를 넘겨준다.

- **에디터 내 자동완성.** 당신이 타이핑하는 동안 AI 가 다음 줄이나 블록을 제안한다. 당신이 완전히 통제한다. 각 제안을 받아들이거나 거부한다. 작업 단위는 몇 줄이고, 커서 바로 아래에 나타나므로 기본적으로 전부 검토한다.
- **레포 단위 보조.** 프로젝트 전체를 볼 수 있고, 여러 파일을 한 번에 편집하고, 명령을 실행하고, 출력을 읽는 보조와 대화한다. 이 책의 작업 대부분이 여기서 일어난다. 작업 단위는 한 단계나 작은 기능이며, 매 턴마다 디프를 검토한다.
- **자율 에이전트.** 목표를 준다—“테스트 스위트를 통과시켜라”, “이 모듈을 새 API(애플리케이션 프로그래밍 인터페이스, 한 소프트웨어가 다른 것들이 쓸 수 있도록 노출하는 명령들의 집합)로 마이그레이션하라”—그러면 에이전트가 계획하고, 편집하고, 실행하고, 오류를 읽고, 여러 단계에 걸쳐 다시 시도한다. 당신은 개별 행동 하나하나가 아니라 결과를 검토한다. 작업 단위는 과업 전체다.

각 계층은 당신에게 더 많은 고삐를 넘겨준다: 더 큰 작업 단위, 더 적은 연속적 조종, 그리고 검토하는 시점이 더 늦어진

다:

계층	작업 단위	검토 대상	제어
자동완성	몇 줄	제안 하나하나	 뽀뽀
▼			
레포 보조	단계/기능	디프/턴	
▼			
자율 에이전트	과업 전체	그 결과	 느슨

더 많은 고삐 ▼ — 계속 조종 → 풀어놓고 결과 확인

중요한 도약은 마지막 것이다. 앞의 두 계층에서는 끊임없이 조종한다. 자율 에이전트에서는 풀어놓고 결과를 확인한다. 그건 다른 기술이고, 이 장의 나머지가 그것에 관한 내용이다.

언제 풀어놓고 언제 직접 조종할 것인가

에이전트를 무인 상태로 돌릴지 결정하는 것은 세 가지 질문으로 귀결된다—자리를 뜨기 전에 돌리는 체크리스트라고 생각하라.

- **폭발 반경.** 이게 잘못되면 얼마나 많은 게 망가지는가? 코드베이스 전반의 포매팅 작업은 폭발 반경이 작다. 데이터베이스 마이그레이션이나 인증 로직 변경은 크다. 반경이 클수록 직접 손으로 조종하라.
- **되돌릴 수 있는가.** 값싸게 되돌릴 수 있는가? 깨끗한 작업 트리에서 새 git 브랜치 위의 작업은 되돌릴 수 있다—git restore . 하면 사라진다. 데이터를 삭제하거나, 이메일을 보내거나, 카드를 청구하거나, 프로덕션에 푸시하는 작업은 그렇지 않다. 되돌릴 수 없는 것에는 에이전트를 무인으로 돌리지 마라.
- **검증할 수 있는가.** 결과가 맞는지 알 수 있는가? “완료”가 초록색 테스트 스위트나 눈에 보이게 동작하는 페이지를 뜻한다면, 몇 초 안에 검증할 수 있고 에이전트가 잘 맞는다. “완료”가 미묘한 판단—돈 계산, 보안 규칙, 경쟁 조건—에 달려 있다면, 빠른 확인으로 게이트를 걸 수 없으니 가까이 머물러라.

이것들을 합치는 간단한 방법: **작업이 되돌릴 수 있고 결과를 값싸게 검증할 수 있을 때 풀어놓고, 그렇지 않을 때 직접 손으로 조종하라.** 기능 만들기 장의 규칙을 가져오라—폭발 반경이 클수록 단계는 작게—그리고 그 형제 규칙을 더하라: 폭발 반경이 클수록 고삐는 짧게.

헤매지 못하도록 과업의 범위를 정하기

자율 에이전트는 특정한 방식으로 실패한다. 표류한다. 한 가지를 요청했는데 “친절하게” 다른 셋을 리팩터링하고, 언급한 적 없는 파일을 건드리고, 읽을 수 없는 디프를 들고 돌아온다. 해법은 더 똑똑한 에이전트가 아니라—더 뽀뽀한 브리프다.

좋은 자율 과업은 네 부분으로 이루어진다: 명확한 **목표**, 명시적인 **제약**, 구체적인 **완료 기준**, 그리고 에이전트가 스스로 돌릴 수 있는 **가드레일**. 가드레일이 가장 중요한 부분인데, 그게 에이전트가 당신 모르게 넘을 수 없는 울타리이기 때문이다.

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

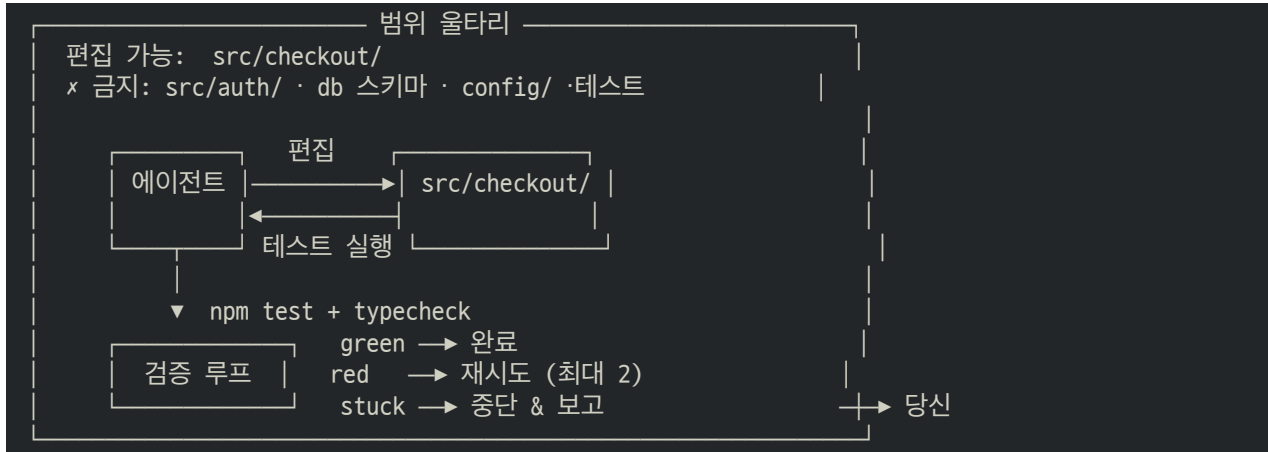
- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.
- Do not change the tests themselves — the tests define correct behavior.

Done when:

- `npm test tests/checkout/` is fully green.
- `npm run typecheck` passes.
- `git diff --stat` shows changes only under src/checkout/.

If you get stuck after two real attempts, stop and report what you tried and what's still failing. Don't paper over a failure to make the suite go green.

이 브리프가 효과적인 것은 에이전트를 가둬놓기 때문이다: 제약이 건드려도 되는 것에 벽을 치고, 완료 기준은 에이전트가 돌릴 수 있는 명령이며, 검증 루프가 그것을 정직하게 유지한다:



세 가지가 이 브리프에서 헤매기 어렵게 만든다. 제약은 에이전트가 건드려도 되는 파일 주위에 벽을 친다. 완료 기준은 분위기가 아니라 누구나 돌릴 수 있는 명령이다. 그리고 마지막 단락은 허우적대는 대신 멈추고 묻도록 허가를 준다—에이전트가 20 분 동안 계속 “고치다가” 트리를 처음보다 더 나쁘게 남겨두는 슬로모션 재앙을 막아주는 것이 바로 이것이다.

진짜 올타리는 테스트다. 자기 테스트를 돌릴 수 있는 에이전트는 뽀뽀하고 정직한 피드백 루프를 가지며 대체로 그 안에 머문다. 자기 일을 확인할 방법이 없는 에이전트는 망가진 것을 자신만만하게 건넨다. 이 장에서 한 가지 습관만 가져간다면: **에이전트에게 스스로 검증할 방법을 주고, 그것을 통과하는 것을 완료의 정의로 삼아라.**

백그라운드 및 병렬 에이전트

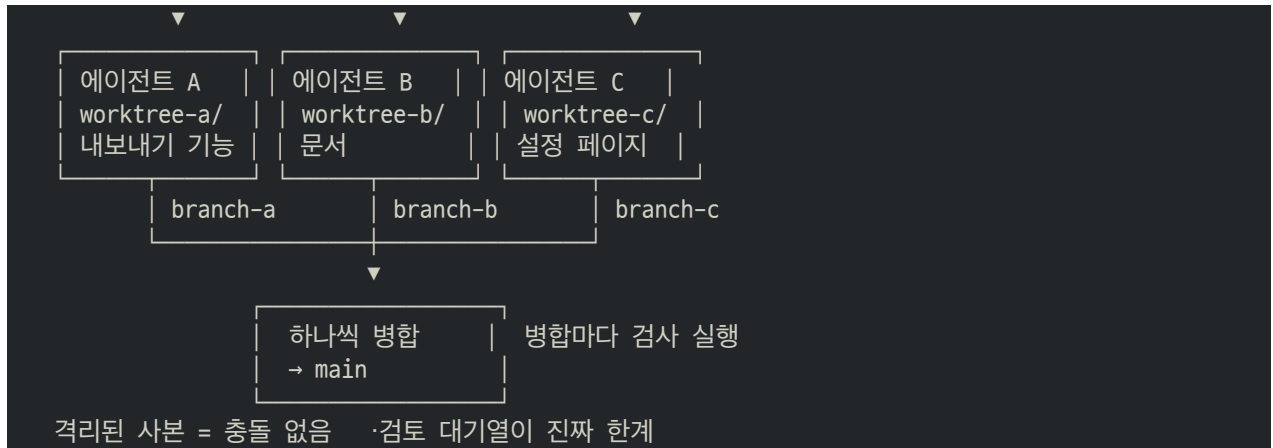
범위가 정해진 과업에서 에이전트를 신뢰하게 되면, 다음 수는 뻥하다: 여러 개를 한 번에 돌리는 것. 현대의 도구들은 백그라운드 에이전트를—각각 자기 브랜치나 워크트리 (프로젝트의 별도 작업 사본으로, 한 에이전트의 편집이 다른 에이전트의 것과 결코 충돌하지 않게 해준다) 에서—띄울 수 있게 해주어, 당신이 다른 일을 하는 동안 서너 개의 범위 정해진 과업이 진척된다.

이것은 진짜 속도 향상이지만, 오직 독립적인 일에 한해서다. 당신이 맞바꾸는 비용은 조율이다. 같은 파일을 편집하는 두 에이전트는 충돌하는 디프를 만들어내고, 그것을 손으로 병합하는 데 절약한 것보다 더 많은 비용이 들 수 있다.

- **줄 수가 아니라 경계로 나뉘라.** 각 에이전트에게 다른 것들과 겹치지 않는 덩어리를 쥐라—별개의 모듈, 별개의 기능, 별개의 파일. “에이전트 A 는 내보내기 기능, 에이전트 B 는 문서 작성”은 깔끔하다. “두 에이전트가 모두 app.js 를 편집”은 병합 골칫거리를 예약하는 셈이다.
- **작업 공간을 격리하라.** 각 에이전트를 자기 git 브랜치나 워크트리에서 돌려 서로의 커밋되지 않은 변경을 밟지 못하게 하라. 한 번에 하나씩 다시 병합하고, 병합할 때마다 검사를 돌려라.
- **개수를 정직하게 유지하라.** 여전히 모든 결과를 검토해야 한다. 네 개의 에이전트가 돌고 있다는 건 읽어야 할 네 개의 디프와 신뢰해야 할 네 세트의 테스트다. 한 줌을 넘어가면, 에이전트가 아니라 검토 대기열이 당신의 병목이 된다.

각 에이전트는 자기만의 격리된 워크트리에서 작업해 그들의 편집이 결코 충돌하지 않는다; 당신은 한 번에 하나씩 다시 병합하며, 각 게이트에서 검사를 돌린다:





실수는 병렬성을 공짜로 여기는 것이다. 그렇지 않다—당신은 일을 작성에서 검토와 통합으로 옮긴 것이다. 종종 훌륭한 거래지만, 오직 당신이 실제로 검토를 할 때에만 그렇다.

멀티 에이전트: 팬아웃, 적대자, 종합

별개의 과업들에 걸쳐서가 아니라 하나의 과업 안에서 돌아가는 “여러 에이전트”의 두 번째 갈래가 있다. 한 에이전트가 모든 것을 하는 대신, 역할을 나눈다:

- **너비를 위한 팬아웃.** 여러 에이전트가 같은 문제를 다른 각도에서 공략하거나—큰 코드베이스의 다른 부분을 탐색한 다음—보고한다. 커버리지를 원할 때 유용하다: “이 폐기 예정 함수를 호출하는 모든 곳을 찾아라”는 잘 병렬화된다.
- **적대적 리뷰어.** 한 에이전트가 작성한다. 두 번째 에이전트의 유일한 임무는 첫 번째의 출력을 공격하는 것이다—버그, 놓친 엣지 케이스, 처리되지 않은 오류를 찾는다. “이것이 틀렸다고 가정하고 왜 그런지 알아내라”는 지시로 디프를 향한 리뷰어는 작성자 에이전트가 스스로 납득해버린 것들을 잡아낸다.
- **종합.** 마지막 패스가 발견 사항을 모아 하나의 일관된 결과를 만들어낸다—동작하는 코드와, 리뷰어가 무엇을 지적했고 어떻게 해결했는지에 대한 짧은 메모.

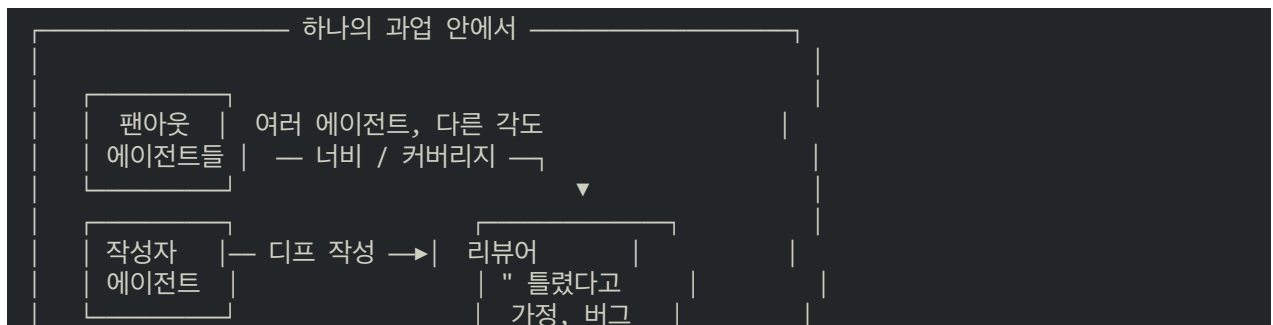
Reviewer agent brief:

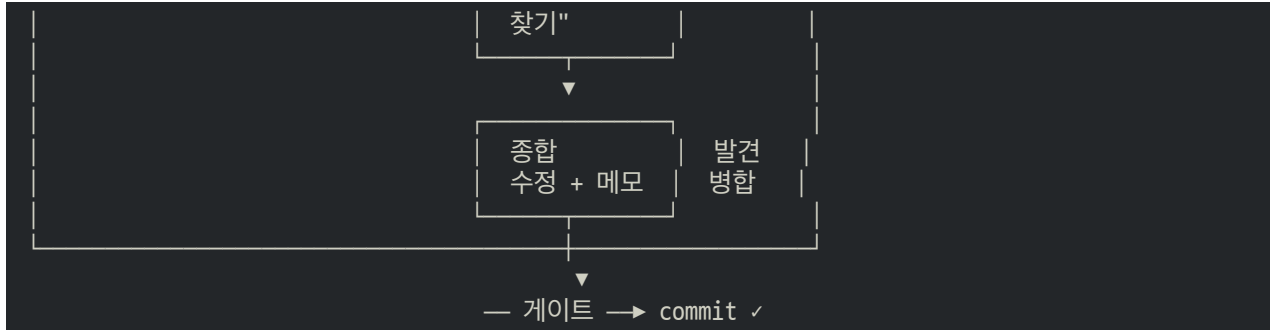
You are reviewing a diff written by another agent. Assume it contains at least one bug and find it. Specifically check:

- Does it actually satisfy the original goal, or just pass the tests?
- Edge cases: empty input, nulls, the largest realistic input.
- Anything deleted or changed that the task didn't ask for.

Report concrete problems with file and line. Do not rewrite the code — just report. If you find nothing after a genuine look, say so plainly.

세 역할은 끝에 게이트가 있는 파이프라인을 이룬다: 적대적 리뷰어가 패스를 마치고 종합이 발견한 것을 해결하기 전까지는 아무것도 커밋에 닿지 않는다.





적대적 리뷰어는 에이전트 함대가 없는 혼자인 경우에도 갖춰줄 가치가 있다—어떤 변경의 작성자는 사람이든 AI 든 그 결함을 찾기에 가장 부적합한 사람이기 때문이다. 적대적인 프롬프트로 한 번 더 돌아보는 것은 값싼 보험이다.

규모에 맞춰 출력 검토하기

여기 정직한 문제가 있다. 에이전트들이 여러 과업에 걸쳐 수백 줄을 작성할 때, 당신은 열 줄짜리 디프를 검토하듯 모든 줄을 읽을 수 없다. 읽을 수 있는 척하면 고무도장 찍기 (보지도 않고 승인) 나 마비 (아무것도 신뢰하지 않음) 중 하나로 이어진다. 둘 다 출시하지 못한다.

뚫고 나가는 길은 전부 읽는 대신 몇 개의 층에서 게이트를 거는 것이다:

- **테스트는 바닥선이다.** 관련 테스트가 초록색이고 타입 체커가 깨끗하지 않으면 어떤 에이전트 출력도 병합되지 않는다. 이건 타협 불가이며 자동이다—이 부분은 기계가 한다.
- **위험한 부분을 스폿 리뷰하라.** 전부 읽을 수는 없으니, 폭발 반경이 큰 부분을 읽어라: 인증, 돈, 데이터 삭제, 외부 호출을 건드리는 것 무엇이든. 나머지는 기능 만들기 장의 위험 신호를 훑어라—건드리지 말았어야 할 파일, 조용히 삭제된 것, 작은 요청이 만들어낸 거대한 디프.
- **코드베이스가 아니라 디프를 읽어라.** `git diff --stat` 은 무엇이 얼마나 바뀌었는지 몇 초 만에 알려준다. 한 모듈로 범위를 정한 과업이 여섯 모듈에서 변경을 보인다면, 한 줄을 읽기 전에 더 자세히 들여다보라는 신호다.
- **실행하라.** 초록색 테스트 스위트가 동작하는 앱은 아니다. 그것을 클릭하라. 마지막 확인은 언제나 다른 모든 장과 같은 것이다: 당신 눈앞에서 요청한 대로 실제로 동작하는가?

여기서 범위 정하기는 두 번 보답한다. 뻑뻑하게 범위를 정한 과업은 실제로 검토할 수 있는 디프를 만든다. 풀어놓은 과업은 그럴 수 없는 디프를 만든다. 검토 문제는 대부분 상류에서, 브리프에서 해결된다.

비용과 폭주 루프

자율 에이전트는 루프를 돌며 일한다: 시도하고, 확인하고, 다시 시도한다. 그 루프가 핵심 전부다—그리고 그것은 또한 조용히 청구서를 불러거나 영원히 돌 수 있는 것이기도 하다.

이것은 앞에서 나온 비용 안전 개념과 연결된다: **정액 구독 도구는 멈춤으로써 실패하고, 종량제 유료 API 에이전트는 청구함으로써 실패한다.** 구독 중인 코딩 보조가 한도에 다다르면, 창이 리셋될 때까지 그냥 더 하지 않을 뿐이다. 당신이 유료 API 에 대고 작성한 스크립트가 재시도 루프에 갇히면, 계속 호출한다—그리고 매 호출마다 돈이 든다—당신이 알아차리거나 카드 한도가 찰 때까지.

그래서 규칙은 단도직입적이다:

- **유료 API 에 대고 제한 없는 에이전트 루프를 절대 돌리지 마라.** LLM(라지 랭귀지 모델—이 코딩 도구들 뒤에 있는 종류의 AI) API 를 호출하는 자기 루프를 작성했다면, 반드시 하드 캡이 있어야 한다—최대 반복 횟수, 지출 한도, 타임아웃. 예외 없음, “내가 지켜볼게”도 없음. 안 지켜볼 거니까.
- **모든 루프에 출구를 줘라.** “N 번 시도 후 멈춰라”, “테스트가 두 번 또 실패하면 멈춰라”, “10 분 동안 돌았으면 멈춰라”. 정지 조건이 없는 에이전트는 기능이 아니라 버그다.
- **첫 실행들을 지켜봐라.** 에이전트를 백그라운드에서 돌리도록 신뢰하기 전에, 몇 번의 포그라운드 실행을 처음부터 끝까지 지켜봐라. 당신은 그것이 수렴하는지 확인하는 것이다—같은 수정을 맴돌며 막히지 않고, 완료에 가까워지는지.

- **가드레일이 있는 도구를 선호하라.** 내장 한도가 있는 관리형 보조가 한밤중에 작성한 날것의 API 루프보다 돌려두기에 안전하다. 닫힌 채로 실패하는 구독은 기능이고, 열린 채로 실패하는 API 는 위험요소다.

정직한 한계

에이전트는 이 책에서 가장 강력한 형태의 지렛대이며, 바로 그렇기 때문에 그것이 무엇이고 무엇이 아닌지에 대해 명확한 눈을 갖는 것이 가치 있다.

그것은 당신이 이해하기를 멈출 수 있는 자율성이 아니다. “이 코드가 무엇을 해야 하고, 틀렸다면 내가 어떻게 알 수 있는가”에 더 이상 답할 수 없게 되는 순간, 당신은 지렛대를 쓰는 데서 운을 바라는 데로 넘어간 것이다. 이 장의 모든 총범위 정하기, 가드레일, 테스트, 스폿 리뷰, 비용 상한—은 에이전트가 타이핑을 훨씬 더 많이 하게 두면서도 당신을 그 선의 이해하는 쪽에 머물게 하기 위해 존재한다.

기술의 천장은 이 책의 나머지와 같다: 원하는 것을 정확히 말하고 그것을 얻었을 때 알아보는 당신의 능력. 에이전트는 그 능력을 공급한다. 대체하지는 않는다. 화를 입는 사람은 에이전트를 쓴 사람이 아니라—에이전트를 써서 생각하기를 멈춘 사람이다. 그렇게 되지 마라. 일은 넘겨주되, 판단은 쥐고 있어야.

정리와 연습

핵심 요약

- 에이전트는 원하는 바를 정확히 말하고 그것을 얻었는지 알아보는 능력을 배가시킬 뿐, 대체하지 않는다.
- 범위 설정은 두 번 보상한다: 좁게 잡은 작업은 실제로 리뷰할 수 있는 diff 를 내고, 느슨한 작업은 리뷰할 수 없는 diff 를 낸다.
- 코드베이스가 아니라 diff 를 리뷰하라—`git diff --stat` 은 몇 초 만에 범위 이탈을 짚어내고, 통과한 테스트 스위트가 직접 실행을 대신할 수는 없다.
- 유료 API 를 상대로 스로틀 없는 에이전트 루프를 절대 돌리지 마라. 모든 루프에 하드 캡을 뒤라—최대 반복 횟수, 지출 한도, 또는 타임아웃.
- 백그라운드 실행을 믿기 전에 첫 실행들을 포그라운드에서 지켜보며 에이전트가 수렴하는지 확인하라.

직접 해보기

작고 경계가 분명한 작업 하나를 골라, 명시적 종료 조건 (“3 번 시도 후 중단”또는“테스트가 두 번 더 실패하면 중단”) 과 함께 에이전트에 넘겨라. 포그라운드에서 실행을 끝까지 지켜본 뒤, `git diff --stat` 에 이어 `git diff` 로 diff 만 리뷰하라. 변경이 범위로 정한 것보다 많은 파일을 건드렸다면, 그것이 브리프를 조이고 다시 돌리라는 신호다.

이 장의 프롬프트

```
I'm giving you an autonomous task. Obey these guardrails exactly:
- SCOPE: only touch [list the files/modules]. If the fix needs more,
  STOP and tell me instead of expanding scope.
- GOAL: [one concrete, checkable outcome].
- STOP CONDITIONS: stop after 3 attempts, OR if tests fail twice in a
  row, OR if you've made changes outside the scope above.
- COST: do not call any paid API in a loop without a hard cap.
- REPORT: when you stop, show me `git diff --stat`, the test result,
  and one sentence on whether you converged or got stuck.
Begin, and narrate each attempt so I can watch you converge.
```