

# 로보코에서 최근 바이브 코딩으로 진행한 작업들

- **플랫폼 엔지니어링**

- 기존 인프라 환경을 테스트 및 검증이 포함되어 완전 자동화된 클라우드 플랫폼으로 이전

- **보안 리뷰**

- 아키텍처 문서화, 취약점 분석, 보완 계획 수립, 체크리스트 작성, 규정 매핑 테이블 작성

- **아키텍처 개선작업**

- 현황 분석, 계획 수립, 마이그레이션 수행, 파이프라인 구축, 문서화

- **마이크로서비스 마이그레이션**

- 기존 모노리식 시스템의 ERD와 API를 소스로 계획수립, 컨텍스트 바운더리 생성, 마이크로 서비스 후보군 탐색, API 및 데이터 스키마 자동 생성, 실제 마이그레이션 실행까지 수행

- **교육 관련**

- 바이브 코딩 교육을 위한 데모 프로젝트 6개, 교안작성, 서적 집필

- 모노리식 Realworld 앱을 서비스 마이크로 서비스 아키텍처로 마이그레이션



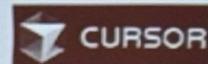
01

# 바이브 코딩이란 무엇인가?

# 왜 지금 바이브 코딩을 시작 해야 하는가?

- 압도적인 생산성
  - 심지어 빠르게 더 좋아지고 있음
- 미래의 보편적인 개발방식
- 소규모 인원으로 대규모 개발이 가능
  - 커뮤니케이션 비용 감소로 인한 생산성 증대
- 안정적인 품질 보장

## SMALL TEAMS ARE THE FUTURE:



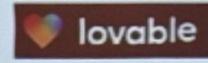
CURSOR

0 to \$100M ARR in 21 months w/ 20 people



bolt.  
new

0 to \$20M ARR in 2 months w/ 15 people



lovable

0 to \$10M ARR in 2 months w/ 15 people



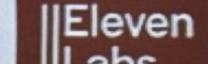
Midjourney

0 to \$200M ARR in 2 years w/ 10 people



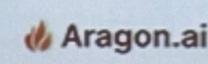
MERCOR

0 to \$50M ARR in 2 years w/ 30 people



Eleven  
Labs

0 to \$100M ARR in 2 years w/ 50 people



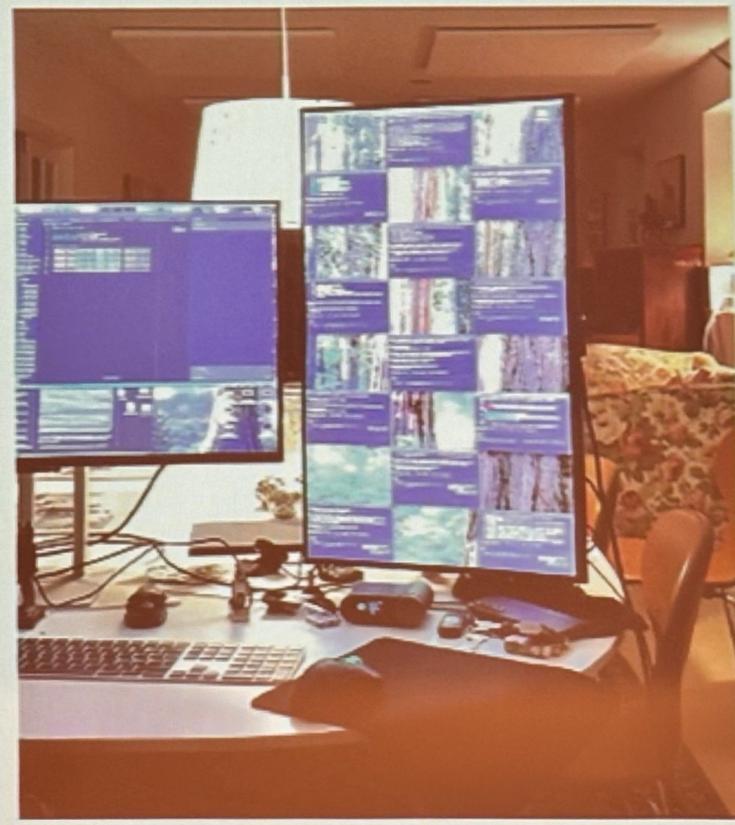
Aragon.ai

0 to \$10M ARR in 2 years w/ 9 people

curated by Ben Lang

# 그래서 얼마나 좋은데?

- 이론상 수십배의 생산성 향상이 가능
- 현 시점에서는 10배정도가 현실적인 목표치
- 품질은 왜 좋아지는가?



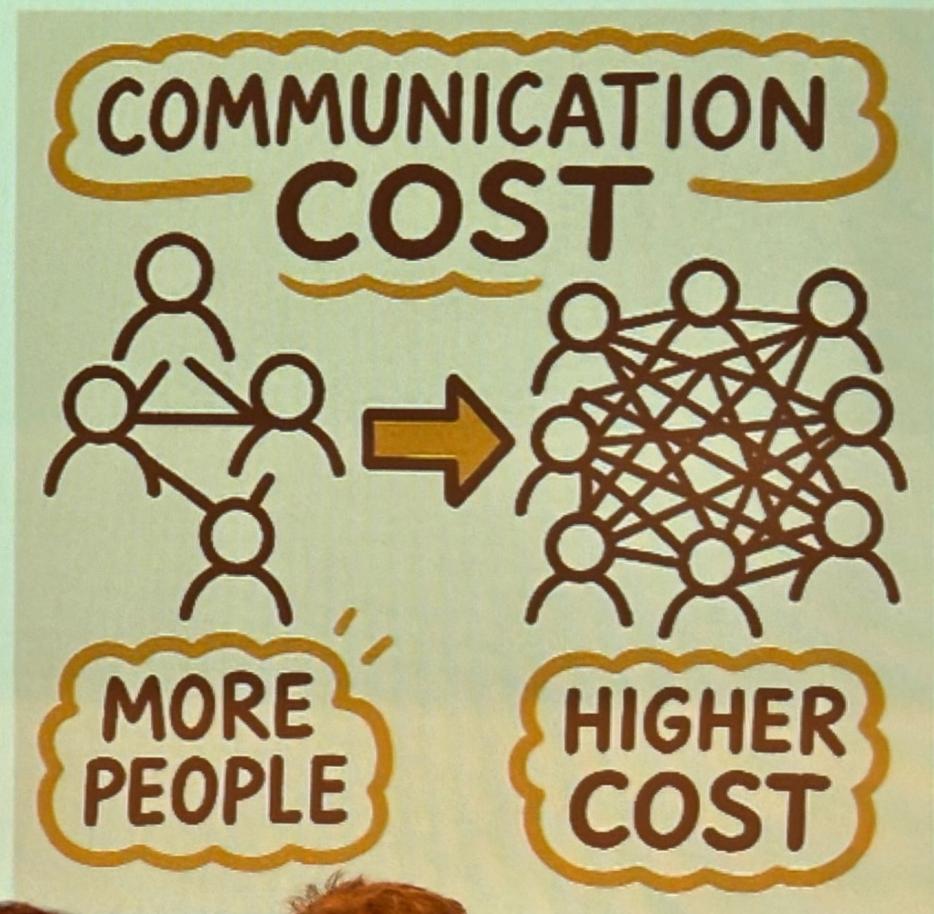
미겔이 미제다인

# 소인원 대규모 개발이 왜 좋은가?

- 커뮤니케이션 비용에 있어서 절대적인 우위
- 커뮤니케이션 비용 산출 공식(=모든 점을 잇는 선의 갯수)

$$\frac{x(x - 1)}{2}$$

기하급수적으로 증가!



# 바이브 코딩이 완성되면 - 특이점 발생

- 더이상 코드는 자산이 아니라 부채임
- 가장 중요한 것은 명확한 요건 정의
  - 소프트웨어는 그때그때 요청에 의해 빠르고 손쉽게 만들어지는 것
- 개발자가 없어질까? No
  - 하지만 코딩이라는 '행위'는 없어질 것이다
  - 캐슬린 부스가 어셈블리어를 만들자 존 폰 노이만이 노발대발하며 "이딴건 하드웨어 낭비야!"



# 바이브 코딩의 단점

- AI는 일반적으로 장황하고 복잡하게 코드를 짜려는 경향이 있음
  - 하드코딩, 예외처리를 통한 폴백등도 서슴없이 자행함
  - 처리 할 수 있는 컨텍스트 양의 제약
    - 빠르게 늘고 있지는 하나 아직 충분하지 않음
    - 컨텍스트 양에 비례하여 비용 발생
  - 최신 버전 사용 문제
    - 1~2년 전 코드로 학습이 되기 때문
    - 장기적인 사용이 인간에게 어떤 영향을 미칠까?



02

## 바이브 코딩이 실패하는 이유

# 이유1

- 레벨에 맞는 프로세스나 도구가 필요함
- 다양한 프로세스와 도구가 쏟아져 나오고 있으나 이를 적절히 활용할 수 있는 경험자가 절대 부족한 상황



# 바이브 코딩의 생산성 방정식

$$P = X * Y + Z$$

: 총 생산성

: 개발자가 낼 수 있는 생산성

※ 사람에 따라 소수점이나 심지어 음의 생산성도 존재함

툴과 프로세스가 낼 수 있는 생산성

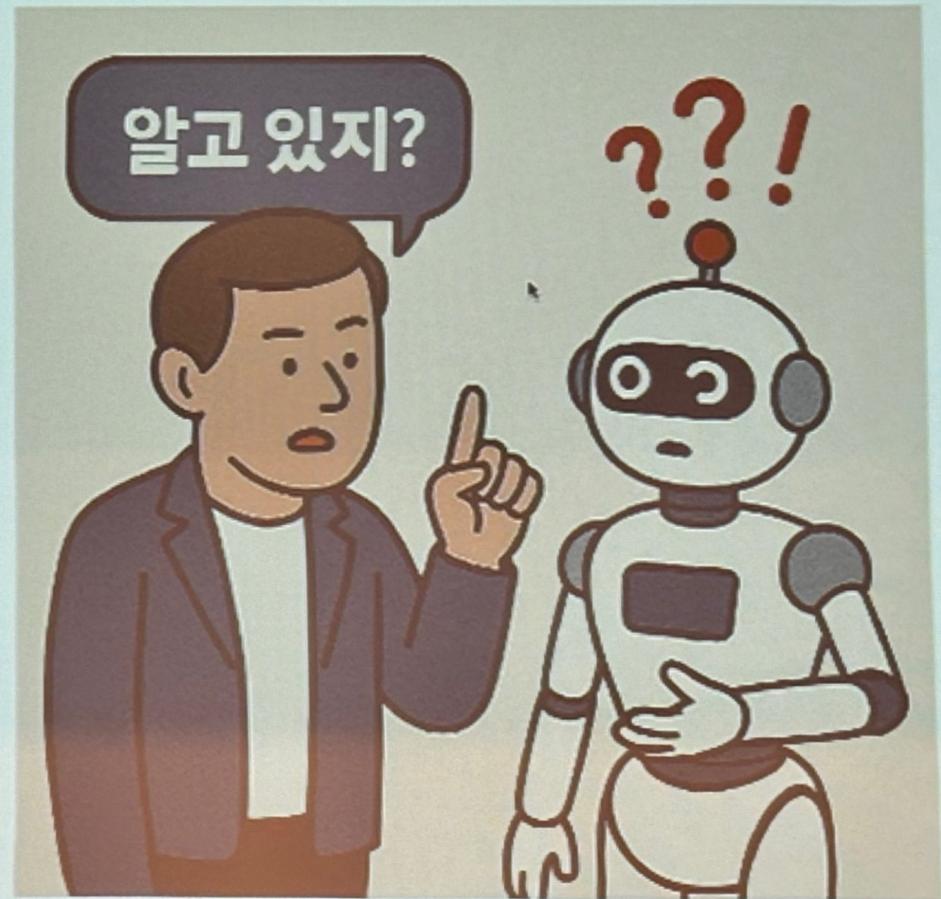
순수하게 AI가 낼 수 있는 생산성

# 다 같은 바이브 코딩이 아니다!

레벨	명칭	핵심 특징	개발자 역할 변화
L1	단순 보조자 수준 (Code Prediction)	함수 수준의 코드 자동완성 제공. 예: GitHub Copilot.	코드 작성의 주도권 유지. AI는 보조 역할.
L2	파일 단위 완성 수준 (Script Automation)	반복 작업이나 간단한 스크립트를 AI가 자동 생성.	코드 검토 및 소규모 수정 중심.
L3	모듈 수준 통합 (Modular Integration)	독립적인 기능 모듈을 AI가 설계 및 제안. 설계 원칙 일부 반영.	모듈 통합 및 관리. 설계 검토 역할 강화.
L4	프로젝트 수준 관리 (Project-Level Orchestration)	설계, 리팩터링, 테스트, 배포 등 프로젝트 전반을 AI가 지원.	요구사항 정의 및 결과물 검토 중심.
L5	비즈니스 목표 중심 자동화 (Business Goal-Driven Automation)	비즈니스 목표와 운영 환경까지 AI가 이해하고 자동화.	전략 수립 및 핵심 의사결정에 집중.

# 당신의 프로덕션 바이브 코딩이 실패하는 이유2

- 공부 부족
- 소프트웨어 개발에 대한 지식에 더해 바이브 코딩 자체에 대한 지식이 필요
  - 도구(IDE)
  - 모델
  - 프롬프트 엔지니어링
  - MCP



# 모델 선택의 중요성

- 사실상 가장 중요한 요소
- 벤치마크 점수와 실제 개발에서의 유용성은 다름
- 바이브 코딩에서 모델은 코딩만 하는 것이 아닌 기획, 설계, 문서작성, 일정 산출, 배포, 디버깅, 모니터링에 이르기까지 다양한 작업에 사용됨
- 2025년 6월 현재 Claude 4 계열 모델이 가장 우수함

# GPT vs Claude

- GPT계열
  - 1년에서 2년 전 사이의 Github의 소스 코드들로 학습
  - 다음 코드가 어떻게 작성될지를 예측하도록 훈련됨

- Claude계열
  - 선별된 소스코드 + 트레이너
  - 개발자가 다음에 어떤 행동을 하는지 예측하도록 훈련됨



03

## LEVEL4 VIBE CODING을 구현하기 위한 과제들

# 과제 1 – 너무 새롭다

- 정립된 프로세스가 아직 없음
- 윗분들이 주로 하는 말 - “**동종업계 국내 성공사례가 있나요?**”
  - 동종업계에서 국내 성공사례가 나오는 날이 우리 비즈니스 시한부 인생 선고일임
- 반대할 구실만 찾는 사람들이 얼마나 회사에 있어서 무서운 사람들인가?
- 최소한 무슨 일이 벌어지고 있는지는 알고 있어야 함

# 어떤 도구가 좋을까?

- 클로드 코드 - 현 시점 압도적 원탑
  - 엔트로픽에서 자신들이 쓰려고 만든 도구임
  - 엔트로픽 공식 홈페이지에 튜토리얼이 매우 잘 정리되어 있음
- 사람에게 좋은것은 AI에게도 좋다
  - 소프트웨어 공학과 모범사례들
  - 기술 스택은?
    - 아르민 로나허 - Agentic Coding Recommendations

# 클로드 코드의 유용한 기능들

- 규칙 자동 생성
- 커맨드(워크플로우)
- 헤드리스 모드
- 계획 모드

# MCP vs CLI

- 대부분의 경우 CLI로도 충분히 문제를 해결 할 수 있음
- CLI는 스크립팅을 통해 손쉽게 자동화가 가능함
- 로그를 사용해 이력에 대한 추적도 손쉬움
- Playwright나 Perplexity와 CLI도구가 제공되지 않거나 복잡한 연동이 필요한 경우는 MCP를 사용

# 바이브 코딩을 위한 최고의 언어 - Golang

- Go언어를 추천하는 이유
  - context 전달이 명확
  - 테스트 캐싱이 잘 되어 반복 작업이 빠름
  - 구조적 인터페이스 덕분에 LLM이 타입을 쉽게 이해할 수 있음
  - 생태계 변화가 적어, 에이전트가 예측 가능한 코드를 생성
  - Python/Rust는 빌드/실행 지연, 숨겨진 동작이 많아 에이전트가 실수하기 쉬움

## 과제2 - 보안

- 컴플라이언스
- AI 주권 문제
- AI가 생성한 코드를 100% 믿을 수 있는가?



# L4 프로세스

국내 컴플라이언스를 충족하는 프로덕션 레벨 바이브 코딩 툴 셋 예시

Process

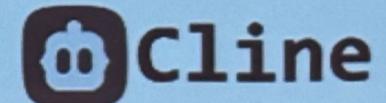


Tool



*upstage*  
Solar LLM

TaskMaster



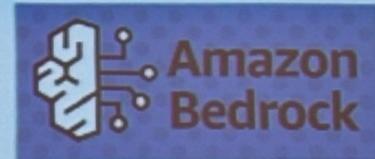
Cline  
Claude Code

Model



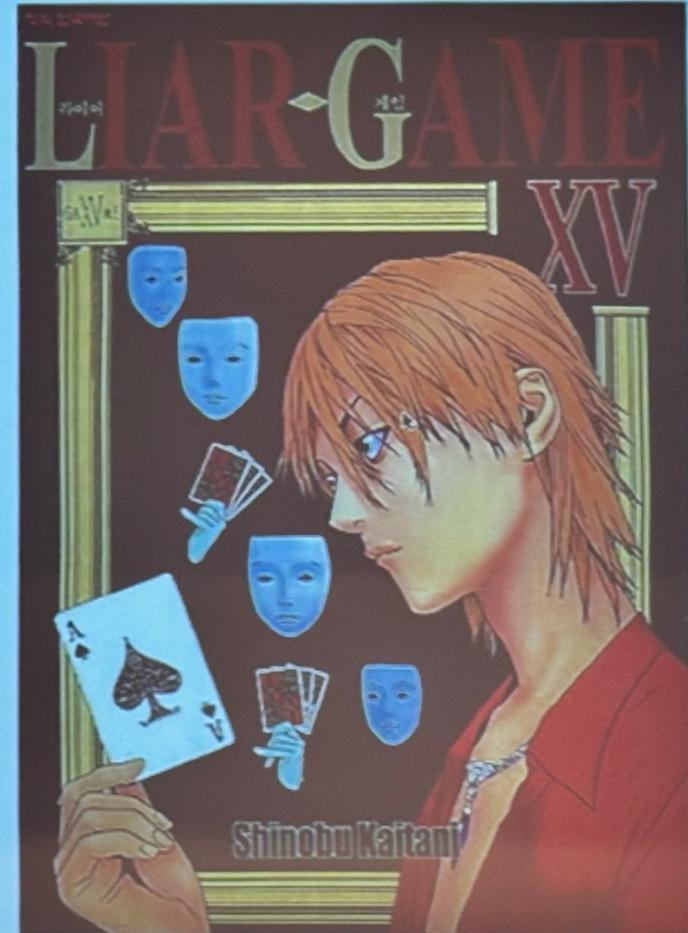
*upstage*  
Solar LLM

Claude



Amazon  
Bedrock

AI가 만든  
코드를 100%  
믿어도 될까?



# AI가 만든 코드를 100% 믿어도 될까?

“**코드**는 의심하고 봐야해.  
많은 사람들이 오해하고 있지만  
**코드**를 의심한다는 것은 그 **코드**를 알려는 행위라고.

많은 개발자들이 테스트를 작성하지 않는 행위는 사실  
**코드**를 알려는 노력의 포기야

그것은 결코 ‘믿는것’이 아니라 무관심이지  
무관심이 의심보다 더욱 비열한 행위라는 것을 많은  
사람들은 모르고 있어.”

정개발 로보코



# AI가 만든 코드를 100% 믿어도 될까?

- AI가 제안하는 코드를 모두 의심하는 프로세스. 자동화된 테스트·리뷰 프로세스가 필수.
- 사람의 개입을 줄이고 구현과 검수를 완전 자동화 하는 품질 관리 및 테스트 체계를 갖춘 다음, 이를 빠르게 개선해 나가는 방식으로 접근 해야 함.
- AI코드리뷰가 포함된 TDD와 BDD에 기반한 완전 자동화된 CI/CD가 좋은 해결책이 될 수 있음.
- 전통적인 개발에 있어서 "그거 좋은거 아는데, 우리는 할 사람이 없어" 와 같은 변명이 더 이상 통하지 않는 시대가 오고 있음.



# 일관성 문제 해결책

- 문서화와 규칙을 AI가 자동적으로 업데이트하도록 한다
- 코드나 문서의 예시나 템플릿을 자동으로 생성하고 이를 참조
- 자동화된 코드 리뷰에서 일관성에 대한 규칙을 강조
- 처음이 중요
  - 깨진 창문 이론 – AI는 기준에 작성된 코딩 스타일을 참조해서 작업 하므로 처음에 각을 잘 잡아놔야함



# 컨텍스트 양의 제약

- 기억할 수 있는 컨텍스트에 제약이 있어 적절한 컨텍스트 크기를 유지할 필요가 있음
  - 마이크로 서비스 아키텍처와의 상성이 좋음
- 문서화를 자동으로 수행하고, 구현 전에 문서를 참고하도록 규칙을 설정
  - 작은 기억 용량을 문서/규칙으로 보완
- 가능한 작은 단위로 나누어 구현과 검증을 빠르게 반복

# 바이브 코딩 실무를 위한 조언들

- 요구사항 외에 의도를 함께 전달한다
- 먼저 물어보자 – “너에게 이러이러한 작업을 지시하려해. 내가 뭘 알려줘야해?”
- 정기적으로 여러 도구들을 사용해서 벤치마크를 돌려보자
  - PRD에서 바이브 코딩으로 원하는 결과물이 얼마나 빠르고 정확하게 나오는지 체크



# 바이브 코딩에 있어서 모범 사례의 중요성

- 사람에게 유용한 것은 AI에게도 유용하다.
- 지금까지 소프트웨어 개발에서 권장되어 왔던 여러 모범 사례들은 바이브 코딩에 있어서도 유용하다.
- 적은 비용으로 기계적으로 수행 가능한 정적 코드체커, 린트, 타입 체커, 테스트 등을 최대한 자주 수행 될 수 있도록 CI/CD 파이프라인을 구성한다.



# 유용한 모범사례들(1)

- XY Problem
  - 모델에게 의도를 명확하게 전달할 것
  - 요청 전에 어떤 것을 알려줘야 할지 물어본다 – 요청을 위한 질문
- 마이크로서비스
  - 빠르고 안정적인 개발/확장에 효율적
  - 컨텍스 범위 한정에 매우 효과적
- SOLID 원칙
  - 수정에는 닫혀있고 확장에는 열려있는 구조
  - 결합도는 낮고 응집도는 높은 코드 작성
- 클린아키텍쳐/DDD
  - 코어 비즈니스 로직 격리



# 유용한 모범사례들(2)

- 단위 테스트/ E2E 테스트 작성
  - 테스트 코드는 LLM에게 매우 좋은 '요청 언어(명세)'가 될 수 있음. 자연어로 요구사항을 설명하는 것보다, 테스트 코드는 코드가 어떤 동작을 해야 하는지 명확히 정의함. 많은 사람들이 LLM이 만들어낸 중간 정도 수준의 코드를 검토해야 하고 비용도 든다고 불평하지만, 테스트 코드를 작성해두면 테스트 실행만으로 LLM이 작성한 코드를 손쉽게 검증할 수 있음.
- 버전 관리 사용
  - LLM은 비동기로 작동하므로, 사람처럼 서서히 코드를 작성하는 것이 아니라 단시간에 많은 코드를 생성할 수 있음. 이때 버전 관리를 사용하면 실수를 되돌리거나 수정하기가 훨씬 쉬움. 또한 LLM과 협업할 때, 사람은 diff를 검토하여 필요한 부분만 선택적으로 머지할 수 있어 작업을 효율적으로 관리할 수 있음.
- 린터와 포매터 사용
  - LLM이 어느 정도 포매팅과 린팅된 코드를 생성할 수 있지만, 자동화 룰이 훨씬 빠르고 정확하게 버그나 문제를 발견할 수 있음. 린터와 포매터 설정을 사용하면, LLM도 해당 코딩 표준을 학습해 더 일관된 코드를 생성할 수 있음.



# 유용한 모범사례들(3)

- README 문서 유지하기
  - LLM은 별도의 모델로 재학습하지 않는 이상 아무것도 기억하지 못한다. 주어진 컨텍스트만을 사용하므로, 종종 처음부터 새로 시작하는 편이 좋다.
- 마일스톤, 단계, 진행 상황 문서화하기
  - LLM 역시 방대한 맥락 속에서 막연한 요청을 받는 것보다, 작은 단위의 작업을 더 효과적으로 처리할 수 있음. 또한 인간이 LLM의 진행 상황을 이해하기도 더 쉬워지고, 복잡한 문제를 작게 쪼개어 처리함으로써 LLM이 더 어려운 작업도 수행할 수 있게 됨.

# 바이브 코딩 관련 최신 정보들

- 아르민 로나허(@ArminRonacher)
  - 플라스크 개발자로 유명
  - 유튜브 채널과 블로그를 통해 바이브 코딩 노하우 공유
- 바이브 코딩 뉴스 단톡방(<https://open.kakao.com/o/gy12RTBh>)
  - 바이브 코딩 관련 최신 정보를 공유하는 단톡방

# 규칙 생성

- 매번 프롬프트, 히스토리와 함께 LLM에 입력으로 사용됨
- 강력하게 AI의 행동을 제약함
- 모델에 따라 규칙 준수 정도가 다름
- 요건정의로부터 자동 생성이 가능
- 템플릿을 사용할 수 있음
- 툴에 따라 복잡한 규칙 적용이 가능

1



# 요건정의

- 바이브 코딩에 있어서 가장 중요
- 육하원칙
  - 누가 만드는가(Who)
  - 언제까지 만들 것인가(When)
  - 어디에(플랫폼) 만들 것인가(Where)
  - 무엇을 만들 것인가(What)
  - 어떻게 만들 것인가(How)
  - 왜 만드는가(Why)



# 설계 문서 작성

- 요건 정의 내용을 구체화
- 하나의 파일에 콘텍스트가 너무 많아지지 않도록 적당히 파일을 분리하는 것이 필요
- 템플릿이나 샘플을 제공하면 원하는 결과물을 얻는데 도움이 됨
- mermaid와 svg를 적극 활용



# 테스크 작성

- 설계 문서에 기반해서 작업을 분할
- 체크리스트 형태로 작성
- 한 번 분할한 후에 분할이 충분한지 검토
- 확인 가능한 지점을 커밋 지점으로 설정하고 인수 조건 (AC)을 명시하도록 요청
- 테스크 상태를 자동적으로 업데이트 하도록 규칙 설정



# 구현작업

- 부트스트랩 작업은 튜토리얼을 함께 제공하는 것을 권장
- TDD로 구현 작업 진행
- husky등을 사용해 Git Hook으로 커밋시에 Lint, Build, Test 가 수행될 수 있도록 설정
- 변경사항이 있을 경우 커밋전에 문서 업데이트 수행



# 디버깅 작업

- 디버깅에 가장 많은 토큰이 소모됨
- 로그 필터링을 스크립트로 작성해 두면 토큰을 아낄 수 있음