바이브 코딩을 이용한 DevOps

바이브 코딩이란?

- **AI 지원 개발** – 자연어 지시로 코드 구현하는 새로운 개발 스타일 (Karpathy, 2025)
- **LLM과 페어 프로그래밍** – 개발자-AI 대화형 소프트웨어 개발
- **"느낌대로 코딩"** – 코드보다 제품 전체 모습에 집중하는 즉흥적 개발 방식

왜 바이브 코딩인가?

- **고속 개발 사이클** – 아이디어 → 프로토타입 즉시 구현, MVP 개발 가속화
- **문제 해결 집중** – 구현보다 요구사항 정의와 솔루션 설계에 집중
- **빠른 반복 실험** – 저비용 아이디어 검증, 짧은 피드백 주기

바이브 코딩 현황과 영향

- **급속한 확산** – YC 2025 스타트업 25%가 코드베이스 95% AI 생성
- **업계별 도입률** – 디지털 에이전시 61%, 이커머스 57%
- **성과 지표** – 사용자 참여율 47%↑, 유저 유지율 32%↑

바이브 코딩과 엔지니어링 성숙도

- 성숙도가 바이브 코딩의 전제 조건 – 바이브 코딩은 단순한 자동 코드 생성이 아닙니다. 엔지니어링 성숙도가 높은 환경에서만 바이브 코딩은 제대로 힘을 발휘할 수 있습니다. 성숙한 개발 환경이란 충분한 문서화, 자동화된 테스트, 완전한 CI/CD 파이프라인, 그리고 코드형 인프라(IaC) 기반의 인프라를 갖춘 환경을 의미합니다. 이러한 기반이 없으면 AI가 생성한 코드의 품질을 검증하고 안전하게 배포하기 어렵습니다.
- 상호 보완적 발전 관계 – 흥미롭게도 바이브 코딩과 엔지니어링 성숙도는 상호 보완적인 관계를 가집니다. Claude Code 같은 도구는 코드 분석, 문서 작성, 구현, 테스트, 보안 스캔 등을 지원하여 엔지니어링 성숙도를 높이는 데 기여합니다. 동시에 바이브 코딩을 통해 문서화, 인프라, 리뷰 프로세스가 개선되면서 전체적인 엔지니어링 성숙도가 향상되는 선순환 구조가 형성됩니다.
- 도입 전 필수 준비사항 – 바이브 코딩을 효과적으로 도입하기 전에 다음 요소들을 먼저 구축해야 합니다: (1) 프로젝트 문서화 개선 - AI가 맥락을 이해할 수 있는 충분한 문서, (2) 테스트 자동화 구축 - AI 생성 코드의 품질 검증, (3) 완전 자동화된 CI/CD 파이프라인, (4) 인프라스트럭처 as 코드(IaC) 기반의 인프라. 이러한 기반이 갖춰진 후에야 AI와의 협업이 진정한 생산성 향상으로 이어집니다.

사례: 주말 프로젝트도 AI로 뚝딱

- 1시간만에 웹앱 골격 완성 – 한 개발자는 주말 사이 Claude Code를 활용해 Astro 기반 웹사이트를 몇 시간 만에 만들어냈습니다. 프로젝트 폴더에 `claude.md`로 기본 규칙(TypeScript 엄격 사용 등)을 적어주고, 원하는 앱의 기능과 디자인을 한 단락으로 설명했더니 약 한 시간 만에 기본 사이트가 실행되었습니다.
- 몇 주 걸릴 일이 하루에 – 추가 기능 구현과 UI 개선에 약 3시간이 더 소요되어 총 4시간여 만에 첫 버전이 완성되었습니다. Claude가 프로젝트 구조 전체를 인식하고 필요한 부분을 알아서 생성/수정해주었기 때문에, 개발자는 구현보다는 요구사항 조율과 간단한 오류 수정에 집중할 수 있었습니다. 보통 몇 주는 걸렸을 작업을 하루만에 끝낸 경험에 개발자는 놀라워했습니다.
- 자동 생성 코드의 검토 – 물론 AI가 생성한 코드를 개발자가 간간이 점검하고 수정하는 과정은 필요했습니다. Claude가 가끔 엉뚱한 방향으로 구현하려 할 때 이를 제지하고 원하는 방향으로 재프롬프트하는 식으로 조율했지만, 결국 대부분(70% 정도)의 코드를 AI가 알아서 작성했고 개발자는 마무리 다듬는 수준이었습니다. “상당히 잘 동작해서 무섭기까지 했다”는 소감이 나올 정도로 생산성이 높았습니다.

Claude Code: 바이브 코딩을 위한 AI 도구

- Anthropic의 Claude Code – Claude Code는 명령줄 기반 AI 코딩 도구로, 소스 코드 저장소 디렉토리에서 실행되며 프로젝트 전체 구조를 읽고 이해합니다. 개발자가 자연어로 변경 요청을 입력하면, Claude가 해당 내용을 반영하여 코드를 추가/수정하고 결과를 제시합니다. 이 모델은 코딩 작업에 특화된 LLM으로, 프로젝트 컨텍스트에 맞는 답변과 코드를 생성하도록 튜닝되어 있습니다.
- 코드 자동 적용 – Claude Code는 제안된 코드 변경을 실제 파일에 적용하기 전에 미리 보여주고 사용자에게 승인을 요청할 수 있으며, 신뢰도가 높으면 곧바로 파일을 수정하도록 자동 승인(autonomous mode) 설정도 가능합니다. 한 세션에서 프로젝트 내 여러 파일을 동시에 편집할 수 있고, AI가 제시한 변경이 마음에 들지 않으면 취소하거나 되돌린 후 다른 접근을 시도하게 할 수도 있습니다.
- 깊은 컨텍스트 이해 – 이 도구의 강점은 프로젝트 전역 컨텍스트를 인식한다는 것입니다. 현재 폴더의 코드베이스 전체를 읽어들여 각 파일 간 관계와 구조를 파악하므로, 부분적인 지시를 내려도 Claude가 관련되는 다른 모듈이나 함수까지 고려한 통합적인 수정을 수행합니다. 이로써 개발자는 세부 구현보다는 AI에게 목표를 설명하는 데 집중할 수 있습니다.

(Section) 엔지니어링 모범사례와 Claude Code

Claude Code 환경 세팅 팁

- 프로젝트 지침 `CLAUDE.md` 활용 – Claude Code는 시작 시 디렉토리의 `CLAUDE.md` 파일을 자동으로 읽어들여 대화 컨텍스트에 포함합니다. 따라서 이 파일에 프로젝트의 핵심 정보를 정리해두면 매우 유용합니다. 예를 들면 자주 쓰는 빌드/테스트 명령, 코드 스타일 가이드, 테스트 작성 지침, 브랜치 전략 등을 적어두면 Claude가 항상 이를 참고하여 일관성 있는 코드를 만들어줍니다. (`CLAUDE.md` 내용은 사람이 읽기 쉽고 간결하게 작성하는 것이 좋습니다.)
- 지속적인 프롬프트 튜닝 – 한 번 만든 `CLAUDE.md`를 그대로 두지 말고, AI의 응답 품질을 보면서 계속 개선하세요. 모델이 잘 따르지 않는 지침이 있다면 강조 표현(예: "IMPORTANT", "YOU MUST")을 추가해보고, 잘못된 출력이 나올 땐 그 원인을 분석해 내용을 조정합니다. Claude Code에서 `#` 명령을 쓰면 자동으로 CLAUDE.md에 내용을 추가해주기도 하니, 코딩 중간중간 규칙을 학습시키는 습관을 들이면 좋습니다.
- 도구 권한 미리 설정 – Claude Code는 기본적으로 파일 수정, shell 명령 실행 등 위험할 수 있는 액션에 대해 일일이 허가를 구합니다. 반복적인 액션의 경우 이를 자동화하면 편리합니다. `/permissions` 명령이나 설정 파일을 통해 허용 툴 목록(allowlist)을 관리할 수 있는데, 예를 들어 `Edit` (파일 편집)이나 `Bash(npm run test:*)` (테스트 실행) 등을 항상 허용해두면 작업 흐름이 중단되지 않습니다. 신뢰할 수 있는 명령은 미리 허용하고, 위험한 명령은 계속 확인하도록 설정을 조율하세요.

자동 모드 vs. 동기식 협업

- Auto-accept 모드 (비동기) – Claude Code에는 AI가 변경을 제안했을 때 자동으로 수락하여 바로바로 다음 단계로 진행하게 하는 자동 수락 모드(auto-accept)가 있습니다. 이 모드를 활용하면 부수적인 기능이나 시범적인 프로토타입 작업을 AI에게 자율적으로 맡겨둘 수 있습니다. 예를 들어 UI 소소한 개선이나 성능 최적화처럼 핵심 비즈니스 로직이 아닌 작업은 자동 모드로 AI가 빠르게 처리하게 하고, 나중에 결과만 검토하는 식입니다.
- 실시간 동반 코딩 (동기) – 반면, 중요한 기능이나 핵심 비즈니스 로직 개발에는 자동보다는 사람이 실시간으로 보조를 맞추는 동기식 협업이 권장됩니다. 개발자가 Claude의 출력 한 단계 한 단계를 모니터링하면서 필요한 경우 즉각 수정 지시를 내리고 방향을 잡아주는 것입니다. Claude Code를 실시간으로 함께 코딩하는 페어 프로그래머처럼 대하면, AI가 엉뚱한 부분을 건드리거나 구조를 어지럽히는 것을 초기에 막을 수 있습니다. 중요한 부분일수록 상세한 지침을 주고, 결과를 확인하면서 진행하세요.

명확하고 구체적인 프롬프트 작성

- 모호성 최소화 – AI에게 지시할 때는 구체적인 설명과 요구를 제공해야 합니다. 예를 들어 "로그인 기능 개선해줘"보다는 "login 함수에서 비밀번호 시도 횟수를 5회로 제한하는 로직 추가"처럼 구체적으로 말하는 것이 좋습니다. 이렇게 하면 AI가 의도를 정확히 파악해 원치 않는 부분을 잘못 수정하는 실수를 줄일 수 있습니다.
- 맥락 정보 명시 – 프로젝트에 비슷한 이름의 함수나 모듈이 많은 경우, 대상 파일/함수 이름까지 지정하며 요청하세요. "UserService와 AuthService에 각각 이메일 검증 함수 추가"처럼 맥락을 명시하면 Claude가 잘못된 위치에 코드를 삽입하는 것을 방지할 수 있습니다. 또한 원하는 출력 형태(예: "결과를 JSON으로 알려줘")를 미리 알려주면 AI의 응답이 일관되고 다루기 쉽게 나옵니다.

Claude Code의 성과: Anthropic 사례

- 코드의 90%를 AI가 작성 – Anthropic 내부 개발팀은 Claude Code를 활용해 복잡한 기능을 빠르게 구현하고 있습니다. 예를 들어 Vim 모드 기능 개발 시, 전체 코드의 약 90%를 Claude Code가 자율 작성하여 완성했고 개발자는 나머지 10%만 수정/보완했습니다. 그 결과 구현 속도가 크게 향상되었습니다.
- 개발 속도 및 생산성 향상 – Claude Code 도입 후 팀의 개발 사이클이 빨라져 아이디어를 금세 프로토타입으로 만들고 여러 번의 iteration을 거칠 수 있게 되었습니다. AI가 구현 세부를 맡아주니, 개발자들은 큰 그림에서 아이디어를 실험하고 기능을 추가하는 데 집중하며 개발 효율을 높일 수 있었습니다.
- 자동화된 테스트로 품질 유지 – 또한 Claude는 기능을 구현하면서 테스트 코드도 포괄적으로 생성해주고, PR 리뷰에서 단순 버그를 직접 수정해주는 등 품질 관리 측면에서도 기여했습니다. "고품질의 코드를 유지하면서도 수작업은 줄었다"는 평가를 얻었고, 전반적인 코드 신뢰도와 유지보수성이 향상되었다고 합니다.

문서화 작업 자동화

- 문서 생성도 AI에게 – Claude Code는 코딩뿐만 아니라 문서 작성 지원도 뛰어납니다. 예를 들어 여러 소스의 정보를 모아 런북(runbook)이나 트러블슈팅 가이드 같은 마크다운 문서를 자동으로 만들어줄 수 있습니다. 보안팀 사례에서, Claude가 복잡한 인프라 관련 자료들을 요약해 일목요연한 안내서를 생성함으로써, 개발자가 일일이 문서를 찾지 않고도 문제를 해결할 수 있게 했습니다.
- 맥락 공유와 지식 축적 – Claude가 생성한 문서는 Slack이나 Confluence 등에 바로 활용될 수 있을 정도로 포맷이 깔끔하며, 팀원들 간 공유 지식 기반으로도 쓰이고 있습니다. AI를 통해 문서화 부담을 줄이면 개발자는 코드 구현과 설계에 더 집중할 수 있고, 문서의 최신 상태도 자동으로 유지되어 프로젝트 전반의 가시성 향상에 기여합니다.

테스트 코드 생성 자동화

- 다양한 테스트 생성 – Claude Code를 활용하면 단위 테스트(UT)는 물론, 통합 테스트(Integration Test)나 E2E(End-to-End) 테스트 시나리오까지 자동으로 작성할 수 있습니다. 개발자가 함수나 모듈의 의도만 설명해주면, AI가 해당 기능을 검증할 수 있는 테스트 케이스들을 알아서 만들어줍니다.
- 엣지 케이스 포착 – 특히 단위 테스트의 경우 Claude는 개발자가 놓치기 쉬운 경계 상황과 예외 케이스까지 포함한 포괄적인 테스트 코드를 생성해줍니다. Anthropic의 추론팀 사례에서, 핵심 기능 코드를 작성한 뒤 Claude에게 단위 테스트 작성을 요청하니 사람이라면 생각하기 번거로운 에지 케이스까지 모두 포함된 테스트 스위트를 몇 분 안에 얻을 수 있었다고 합니다. AI가 테스트 작성의 정신적 부담을 덜어주고, 코드 품질 보증을 자동화해주는 셈입니다.
- 테스트 주도 개발(TDD) 지원 – 나아가 Claude와 TDD 사이클을 구현할 수도 있습니다. 먼저 AI에게 요구사항에 대한 간단한 의사코드(pseudocode)와 테스트를 생성하게 하고, 그 테스트를 통과시키는 방향으로 실제 코드를 작성하도록 유도하는 것입니다. 이렇게 하면 AI도 명확한 목표를 가지고 구현하게 되어, 결과적으로 더 신뢰성 있는 코드를 얻게 됩니다.

인프라 코드(IaC)도 AI와 함께

- Infra as Code 이해 및 검토 – Claude Code는 Terraform과 같은 인프라스트럭처 코드도 충분히 이해합니다. 보안팀 사례에서, Claude에 Terraform 변경 내용을 보여주고 “이게 무슨 일을 하는지, 문제 없을지 검토해줘”라고 물어보니, Claude가 해당 인프라 변화가 가져올 영향을 설명해주었습니다. 이처럼 IaC 변경 사항에 대한 빠른 피드백을 AI로부터 얻어, 사람 리뷰 전에 미리 위험 요소를 발견할 수 있습니다.
- 인프라 구성 자동화 – 또한 Claude를 활용하면 새로운 인프라 설정 스크립트를 작성하는 데에도 도움을 받을 수 있습니다. 예를 들어 "AWS S3 버킷과 연동된 CloudFront 분산구성을 Terraform으로 작성해줘"라고 하면 Claude가 기본적인 Terraform 템플릿을 생성해줄 수 있습니다. 다만, 실제 적용 전에는 사람이 검토하여 보안 설정이나 비용 이슈가 없는지 확인해야 합니다. AI는 초안을 빠르게 만들어주고, 세부 튜닝은 엔지니어가 맡는 형태로 인프라 구축 속도를 높일 수 있습니다.

CI/CD 파이프라인 구성 자동화

- 반복 작업 AI에게 일임 – 프로젝트마다 비슷하게 설정해야 하는 CI/CD 파이프라인 (예: 빌드 → 테스트 → 배포 단계)이 있다면, Claude에게 그 패턴을 가르쳐서 자동 구성하게 할 수 있습니다. Anthropic 데이터 인프라팀은 `CLAUDE.md`에 파이프라인 스크립트 패턴을 정리해두고, Claude Code로 새로운 데이터 파이프라인 구축 시 해당 패턴에 맞춰 스크립트를 작성하게 했더니 매번 수작업하던 것을 AI가 척척 처리해주었다고 합니다.
- 예시: CI YAML 생성 – 예를 들어 GitHub Actions 설정 파일(`.github/workflows/ci.yml`)을 만들어야 한다면, 현재 프로젝트에 맞는 Job 구성(테스트, 린트, 빌드, 배포 등)을 AI에게 자연어로 설명하고 생성하도록 할 수 있습니다. Claude는 이미 학습한 다양한 CI 설정 예제를 바탕으로, 요구사항에 부합하는 YAML을 생성해줄 것입니다. 생성된 파이프라인은 곧바로 저장소에 커밋해 실험해보고, 실패 시 Claude에게 오류 로그를 보여주며 수정하게 함으로써 파이프라인 설정 역시 대화형으로 완성할 수 있습니다.

🛑 엔지니어링 원칙 없이 위험!

- AI 코드 = 버그 양산기? – 엔지니어링 규칙이 뒷받침되지 않은 채 AI에만 의존하면 잘못된 코드와 버그가 빠르게 양산될 수 있습니다. LLM은 맥락이 부족하면 사실과 다른 코드(환각)를 만들어내거나 일관성 없는 결과를 낼 수 있기 때문에, 검증 없는 채로 받아들였다간 심각한 오류가 누적될 위험이 있습니다. 소규모 프로토타입에서는 큰 문제 아니더라도, 복잡한 실제 서비스에 vibe 코딩을 무턱대고 적용하면 유지보수 악몽이 될 수 있습니다.
- 코드 품질 저하 우려 – AI가 생성한 코드는 그럴듯해 보여도 잘못된 로직이나 비효율이 숨어있을 수 있습니다. 특히 개발자가 코드를 전혀 리뷰하지 않고 "느낌 좋다"고 넘어가면, 버그 뿐 아니라 보안 취약점이나 성능 문제까지 놓칠 수 있습니다. 결과적으로 제품 품질이 떨어지고 디버깅에 더 많은 시간이 들어갈 수 있습니다. (바이브 코딩이 재미있다고 테스트를 소홀히 하면 안 되는 이유입니다!)
- Scale의 함정 – AI를 통해 생산성 향상이 눈에 보이니, 관리자가 단기간에 너무 많은 기능 개발을 요구할 수도 있습니다. 그러나 엔지니어링 성숙도 없이 양만 추구하면 나중에 감당 못할 기술 부채로 돌아옵니다. AI와 함께 폭주하는 대신, 견고한 아키텍처와 클린 코드 원칙을 지키며 속도와 품질의 균형을 잡는 것이 중요합니다.

엔지니어링 모범 사례와 결합 필요

- 기본기 준수 – AI와 개발할 때일수록 소프트웨어 엔지니어링의 기본 원칙을 철저히 지켜야 합니다. 예컨대 TDD(테스트 주도 개발) 사이클을 유지하고, 작은 변경마다 리팩토링하며, 일관된 코드 스타일과 아키텍처 원칙을 고수해야 AI가 코드를 망쳐놓는 것을 방지할 수 있습니다. Claude Code의 성능이 뛰어나도 사람의 관리와 설계가 없으면 품질을 담보하기 어렵습니다. ("AI가 코딩해주니 오히려 개발자가 PM처럼 원칙을 더 신경쓰게 된다"는 경험도 있습니다.)
- AI + TDD: Anthropic 팀 사례 – Anthropic 개발팀도 처음엔 AI가 생성한 코드에 테스트를 깜빡하고 넘어가 "디자인 문서 → 조악한 코드 → 리팩터 → 테스트 포기"의 악순환을 겪었다고 합니다. 하지만 이제는 Claude에게 먼저 간단한 코드 설계와 테스트 작성을 시키고, 그 테스트를 통과하도록 실제 코드를 작성하는 방식으로 전환했습니다. 이 Augmented TDD 접근으로 더 안정적이고 테스트 가능한 코드를 얻고 있으며, 개발자는 중간중간 AI가 헤맬 때만 개입하면 되어 효율도 높였습니다.
- 코드 리뷰 문화 – AI의 도움으로 개발 속도가 빨라졌다고 해서 코드 리뷰를 생략해선 안 됩니다. 오히려 동료 리뷰와 AI 리뷰를 모두 활용해 이중으로 코드 품질을 점검하세요. AI가 지나친 코드를 짰을 수 있으니 사람이 걸러주고, 반대로 사람이 놓친 부분은 AI가 잡아줄 수 있습니다. 또한 이슈 트래커와 CI를 연계한 자동 알림/체크리스트로 모두가 모범 사례를 따르고 있는지 지속적으로 관리해야 합니다.

Kent Beck의 Tidy First 방법론

- Augmented Coding의 제안 – 익스트림 프로그래밍의 대가 Kent Beck은 AI 시대의 개발 방법으로 Augmented Coding을 자신의 블로그에서 제안했습니다. 핵심 아이디어는 코드 변경을 두 종류로 명확히 구분하는 것입니다:
  - 구조적 변화 – 코드의 동작을 바꾸지 않고 구조를 정리하는 변경 (예: 코드 리포맷, 함수명 변경, 메서드 추출, 파일 이동 등 리팩터링).
  - 행동적 변화 – 프로그램의 행동(기능)을 추가/변경하는 실제 기능 구현 변경.
- 구조 먼저, 기능 나중 – Beck은 이 두 가지 변화를 절대 한 커밋에 섞지 말라 강조합니다. 항상 구조적 리팩터링을 선행해서 코드 복잡도를 낮추고 깨끗한 상태에서, 테스트가 통과하는 것을 확인한 후에 행동적 기능을 추가해야 한다고 합니다. 이렇게 하면 변경의 논리 단위가 분명해지고, 문제 발생 시 원인을 쉽게 파악할 수 있습니다.
- "Tidy First" – Beck의 규칙을 한국의 한 컨설턴트가 “Tidy First 방법론”이라 명명했습니다. 의미 그대로 “먼저 정리부터 하라”는 원칙으로, AI와 코딩할 때도 이 방식을 따르면 개발자가 코드에 대한 주도권과 명확성을 유지할 수 있다고 강조합니다. AI가 종종 필요 없는 기능을 덧붙이거나 복잡도를 올릴 때, 이를 방지하려면 항상 구조를 정돈하고 검증한 뒤 다음 기능으로 넘어가라는 철학입니다.

Tidy First 핵심 규칙 (1)

1. TDD 사이클을 엄격히 준수할 것 (레드 → 그린 → 리팩터링).
2. 가장 간단한 실패하는 테스트를 먼저 작성할 것.
3. 최소한의 코드로 테스트를 통과시키고, 필요 이상의 코드는 작성하지 말 것.
4. 테스트가 통과된 후에만 리팩터링할 것 (테스트 실패 상태에서 구조 변경 금지).

Tidy First 핵심 규칙 (2)

5. 구조적 변화와 행동적 변화를 분리하고, 커밋을 명확히 구분할 것 (리팩터링 커밋 vs 기능 커밋을 따로).
6. 모든 테스트가 통과하고, 경고가 없으며, 작업의 논리적 단위가 명확할 때에만 커밋할 것.
7. 중복 코드를 제거하고, 의도가 드러나도록 명확한 이름과 구조로 개선할 것.
8. 함수/메서드는 작게 유지하며, 한 가지 책임만 수행하도록 할 것 (단일 책임 원칙).

Tidy First의 효과

- 복잡도 제어 – Claude Code와 함께 Tidy First를 적용해보니, 프로젝트 대부분에서 매우 효과적이었습니다. 항상 구조 정리부터 시작해 코드베이스를 깔끔하게 유지한 후 기능을 추가하니, 코드가 복잡해져서 중간에 길을 잃는 일이 크게 줄어들었습니다. AI도 깨끗한 구조 위에서 작동하니 불필요한 혼선을 빚을 확률이 낮아졌습니다.
- AI 출력 통제 – Kent Beck이 강조했듯, Tidy First를 따르면 개발자가 코드에 대한 주도권을 놓지 않게 됩니다. AI가 설령 쓸데없는 코드를 만들려 해도, 구조적 리팩터링 단계를 통해 미리 정돈하고 넘어가기 때문에 AI의 과잉 구현을 억제할 수 있습니다. 결과적으로 코드 품질과 복잡도를 개발자가 계속 관리 가능한 범위에 둘 수 있습니다.
- 버그 예방 및 생산성 향상 – 작은 단위로 변화하고 테스트하므로 버그를 조기에 발견하고 수정할 수 있어 안정성이 높아집니다. 또한 기능 추가 전에 리팩터링이 선행되므로 새 기능 구현도 더 수월해집니다. 이러한 사이클이 자리 잡으면 AI와 함께 하면서도 품질과 속도 모두 잡는 개발이 가능해집니다. Kent Beck도 “이 방법론을 여러분 프로젝트에도 강력히 권장한다”고 할 만큼 자신감을 보였습니다.

Tidy First 실전 적용 가이드

- 단계별 구현 원칙 – 실제 프로젝트에서 Tidy First를 적용할 때는 단계적 접근이 중요합니다. 첫째, 가장 간단한 실패하는 테스트를 작성하고, 둘째, 최소한의 코드로만 해당 테스트를 통과시킵니다. 이때 불필요한 기능 추가나 과도한 추상화는 피해야 합니다. 셋째, 모든 테스트가 통과하면 리팩터링 단계에서 코드 구조를 개선하되, 이때도 기능 변경은 하지 않습니다.
- 효과적인 커밋 관리 – 구조적 변화와 행동적 변화를 별도 커밋으로 분리하는 것이 핵심입니다. 리팩터링 커밋은 "Refactor: 중복 코드 제거" 형태로, 기능 추가 커밋은 "Feature: 사용자 인증 로직 추가" 형태로 구분하여 작성합니다. 이렇게 하면 코드 리뷰 시 변경 의도가 명확해지고, 문제 발생 시 원인 추적이 용이해집니다.
- AI와의 협업 최적화 – Claude Code와 함께 작업할 때는 명확한 지시가 중요합니다. "이 함수를 리팩터링해줘"보다는 "Login 함수에서 중복된 검증 로직을 별도 함수로 추출하되, 기능 변경은 하지 말아줘"라고 구체적으로 요청하세요. AI가 불필요한 기능을 추가하려 할 때는 즉시 제지하고 구조 정리에만 집중하도록 유도합니다.

이슈 트래커 통합 활용

- AI의 이슈 관리 능력 – 대부분의 이슈 트래커 (GitHub Issues, Jira 등)는 CLI 도구나 API를 제공합니다. Claude Code는 이러한 CLI를 활용해 이슈 생성부터 PR 생성까지 자동화할 수 있습니다. 예를 들어 GitHub CLI(`gh`)를 설치해두면, Claude가 명령줄에서 `gh issue create` 등을 실행해 새 이슈를 등록하거나, `gh pr create` 로 PR을 열고 설명 작성까지 해줄 수 있습니다. AI가 이슈 트래커와 직접 상호작용함으로써, 개발자는 터미널 안에서 이슈 관리까지 일원화하여 다룰 수 있습니다.
- 원활한 작업 흐름 – 이러한 통합 덕분에 작업 단위를 명확히 유지할 수 있습니다. Claude와 코딩할 때도 현재 다루는 이슈의 ID나 내용을 기억시켜 주면 AI가 해당 맥락에 집중하여 작업하고, 완료 후 자동으로 이슈를 닫거나 커밋 메시지에 이슈 번호를 언급하는 등 일관된 워크플로우를 이어갈 수 있습니다. Jira의 경우도 REST API나 CLI를 통해 유사한 통합이 가능하며, Claude Code에 필요한 스크립트를 가르치면 티켓 상태 변경, 코멘트 등록 등을 AI가 대행할 수 있습니다.
- 예: "이슈 #42 버그 수정" 작업을 Claude에게 시키면, Claude가 `gh` CLI로 이슈 #42 내용을 읽어오고, 거기에 맞게 코드를 수정한 후 `gh pr create` 로 "Fix #42" PR까지 생성해주는 흐름을 구현할 수 있습니다. 개발자는 승인만 해주면 되니 반복 작업 감소와 추적 자동화라는 두 마리 토끼를 잡는 셈입니다.

한 번에 하나의 이슈 – 작은 반복 사이클

- 작게 쪼개서 빠르게 – 바이브 코딩을 할 때도 멀티태스킹은 금물입니다. 한 번에 하나의 이슈에 집중하여 완료하는 습관을 들이세요. 작업을 최대한 작은 기능/버그 단위로 쪼개면 Claude에게 명확한 목표를 제시하기 쉬워지고, 구현→테스트→배포의 사이클을 짧게 돌릴 수 있습니다. 이렇게 하면 맥락 전환 부담이 줄어들고, 문제 발생 시 어느 변경에서 비롯됐는지 파악하기 수월합니다.
- Tidy First 적용 – 하나의 이슈를 처리할 때도 Tidy First 원칙을 적용합니다. 먼저 해당 부분의 코드를 리팩터링(구조적 변화)하여 깨끗한 상태를 만들고, 그 다음 이슈의 요구 기능을 추가/수정(행동적 변화)합니다. 예를 들어 "회원 가입 오류 메시지 수정" 이슈라면, 우선 관련 함수들의 중복 코드나 잘못된 구조를 정돈하고, 테스트를 통과시키고, 그 다음 요구사항대로 메시지를 변경합니다. 이렇게 하면 이슈 해결과 동시에 코드베이스 품질도 한 단계 향상됩니다.
- CI로 즉각 검증 – 각 이슈 구현을 끝내고 코드를 커밋/PR 하면, CI 파이프라인이 자동으로 테스트와 빌드를 수행하여 문제가 없는지 바로 알려줍니다. 만약 테스트에서 실패나 경고가 발생하면 Claude에게 그 내용을 전해주어 즉각 수정하게 하고(AI 핫픽스), 통과될 때까지 반복하세요. 이러한 작은 이슈 단위의 반복 사이클은 버그를 키우지 않고 제때 잡아주며, 개발자는 항상 다음 작은 목표로 빠르게 나아갈 수 있어 효율성과 안정성을 모두 잡게 됩니다.

CI/CD 파이프라인에서 자동 검증

- AI self-check – Claude Code에게 자기 검증 루프를 설정해줄 수 있습니다. 예를 들어 코드를 생성할 때마다 자동으로 빌드, 테스트, 린트를 돌리게 함으로써, Claude가 스스로 자신의 출력물을 검증하고 오류를 수정하도록 할 수 있습니다. Anthropic 팀은 특히 "코드 작성 전에 테스트를 먼저 생성하게 하면 Claude가 그 테스트를 통과하려고 알아서 품질을 맞춘다"고 조언합니다. 이러한 자동 검증 루프로 AI의 일탈을 조기에 교정할 수 있습니다.
- 지속적 통합(CI) 활용 – 개발 과정에서 CI를 적극 활용하세요. GitHub에 코드를 푸시하거나 PR을 열 때마다 유닛 테스트, 통합 테스트, 정적 분석 등이 자동 실행되도록 설정합니다. 이는 AI가 만든 코드라 해도 사람과 동일하게 품질 게이트를 통과해야 병합되도록 보장합니다. 즉각적인 피드백은 AI에게 추가 지시를 내릴 때도 활용됩니다 – 예를 들어 "CI 테스트 결과를 보니 A 함수에서 null 처리 오류가 있다"고 Claude에게 알려주면, AI가 그 부분만 고쳐주는 식입니다. CI를 통한 빠른 피드백이 AI와 인간의 협업 루프를 촉진시켜 줍니다.
- 배포 파이프라인 – 나아가 AI가 특정 브랜치에 merge되면 자동 배포까지 이어지는 CD(Continuous Deployment) 파이프라인도 구축할 수 있습니다. 이는 결국 작은 이슈 단위로 시작된 변경이 검증을 거쳐 곧바로 사용자에게 전달되는 완전한 DevOps 사이클을 완성합니다. AI가 속도를 높여준 만큼, 파이프라인은 안전장치 역할을 톡톡히 해서 이상 징후를 차단하고, 이상 없을 땐 신속 배포하는 역할을 합니다.

Claude Code GitHub Actions 통합

- GitHub에서 Claude 부르기 – Anthropic은 Claude Code를 GitHub에 통합하는 공식 GitHub Action을 제공하고 있습니다. 이 액션을 저장소에 설치하면, 이슈나 PR 코멘트에서 `@claude` 멘션만으로 Claude Code를 호출할 수 있습니다. 예를 들어, 이슈에서 "@claude 이 기능을 이슈 설명대로 구현해줘"라고 댓글을 달면 Claude가 해당 이슈를 분석해 새로운 브랜치에 커밋과 PR을 생성해주는 식입니다.
- 간편한 자동화 워크플로우 – Claude Code GitHub Action을 활용하면 복잡한 API 연동 없이도 댓글 지시를 자동 처리하게 만드는 다양한 워크플로우를 구축할 수 있습니다.
  - 이슈 -> PR: 이슈 코멘트에 "@claude implement"라고 쓰면 이슈 내용 기반 코드 작성 + PR 생성까지 한 번에 이뤄집니다.
  - 구현 조언: PR 코멘트에 "@claude how to implement X?"라고 물으면 PR 코드 분석 후 구체적 구현 가이드를 답해줍니다.
  - 버그 핫픽스: 이슈에 "@claude fix the TypeError in Y"라고 하면 Claude가 버그 위치 파악 -> 수정 -> PR 작성까지 해줍니다.
- 보안 및 설정 – 이 GitHub Action은 Anthropic API 키를 저장소 시크릿에 저장하고, Anthropic에서 제공하는 GitHub 앱을 설치하여 동작합니다. 작업 자체는 GitHub의 러너 상에서 이뤄지므로 코드 유출 없이 안전하게 진행되

(Section) 컨텍스트 엔지니어링: 대규모 AI 코딩 기법

컨텍스트 엔지니어링이란?

- “프롬프트” 이상으로 – 컨텍스트 엔지니어링은 단순히 한두 문장의 프롬프트로 AI를 부리는 것을 넘어, AI가 일할 전체 환경(Context)을 설계하는 접근법입니다. 마치 시니어 개발자가 주니어에게 가이드 문서, 코딩 규칙, 예시 코드 등을 다 제공해주고 작업을 맡기듯, AI에게도 충분한 정보 구조를 제공하여 혼돈 없이 일관된 결과를 얻고자 합니다.
- Vibe 코딩의 한계 보완 – vibe 코딩이 개발자 직관에 의존해 AI에게 최소한의 힌트만 주는 반면, 컨텍스트 엔지니어링은 AI가 실수하거나 헛다리를 짚는 것을 줄이기 위해 의도적으로 상세한 맥락과 제약을 설정합니다. 이 방법을 쓰면 AI 출력의 일관성, 정확성이 높아지고, 대규모/고복잡도 프로젝트에서도 AI를 활용할 수 있게 됩니다. 실제로 vibe 코딩만으로는 어려웠던 보안-critical한 작업들도 컨텍스트 엔지니어링으로 안정성을 확보했다는 보고가 있습니다.
- 환경 설계의 예 – 컨텍스트 엔지니어링의 일환으로, AI에게 출력 형식(예: JSON 스키마)을 미리 알려주거나, 관련 라이브러리 문서와 코드 예시를 함께 제공할 수 있습니다. 이렇게 AI가 추측해야 할 부분을 줄여주면, 잘못된 상상(“환각”)을 할 여지가 줄고 원하는 결과물을 얻기가 쉬워집니다. 결국 개발자는 AI의 입력 지문 전체를 정교하게 프로그래밍하는 셈입니다.

컨텍스트 엔지니어링의 핵심 요소

- 명확한 규칙과 예제 제공 – AI에게 미리 코딩 규칙, 스타일 가이드, 금지 사항 등을 알려줍니다. 예를 들어 "모든 함수에 JSDoc 주석을 달아" 같은 규칙이나, "이전에 만든 함수 X의 구현을 참고해 비슷하게 만들어" 같은 예시 코드를 제공하면 AI가 그 틀을 벗어나지 않습니다.
- 구조화된 출력 설계 – AI가 낼 답변의 포맷을 정의합니다. 예컨대 "API 명세서를 Markdown 표 형식으로 작성해줘" 또는 "JSON으로 결과를 응답해"처럼 지시하면 AI 출력이 일관된 형식을 가지게 되어 후속 처리가 용이하고 오류가 줄어듭니다. 이러한 출력 템플릿을 활용하면 AI가 답을 못 찾아도 엉뚱한 말 대신 최소한 빈 구조라도 내놓아 개발자가 바로잡기 수월합니다.
- 상태 메모리와 장기 맥락 – 큰 프로젝트에서는 AI에게 작업 내역과 결정 사항을 기억시켜야 합니다. 컨텍스트 엔지니어링 기법으로, 대화 히스토리나 요약본을 유지해 AI의 단기 기억 한계를 보완합니다. 예를 들어 이전 대화 내용 중 중요한 부분(디자인 결정 등)을 요약해 system 프롬프트에 넣어두거나, 외부 벡터 DB에 저장해두고 필요시 불러오는 RAG(Retrieval-Augmented Generation) 기법을 씁니다.
- 외부 지식 통합 – AI 모델이 프로젝트 도메인 지식을 다 알고 있지는 않습니다. 따라서 문서/데이터베이스 연동을 통해 필요한 정보를 실시간 제공하는 것도 컨텍스트 엔지니어링의 일부입니다. 예를 들어, Claude Code의 MCP 서버 기능을 이용해 프로젝트 위키나 API 문서에 질의하게 하거나, 관련 코드베이스를 통째로 읽어들이게 함으로써 현실 세계 지식과 프로젝트 맥락을 AI에게 장착시킵니다.

Claude Code Sub-Agents 기능

- 특화된 하위 에이전트 – Claude Code는 서브 에이전트(Sub-Agent)라는 개념을 도입했습니다. 이는 특정 작업 유형에 맞게 미리 구성된 작은 AI 에이전트로, Claude가 작업 중 해당 분야에 이르면 그 서브에이전트에게 일을 위임합니다. 각각의 서브에이전트는 독자적인 시스템 프롬프트(역할 지침)와 별도의 컨텍스트 창을 가지며, 필요한 전용 도구 권한도 개별 설정됩니다.
- 컨텍스트 분리의 이점 – Sub-Agent마다 자기만의 맥락에서 움직이기 때문에, 주 대화 컨텍스트가 혼란스러워지지 않습니다. 예를 들어 "코드 리뷰" 서브에이전트가 있다면, 메인 Claude는 코딩에 집중하면서도 코드 리뷰 단계에서는 이 서브에이전트를 불러 그 작업만 전담시킵니다. 이렇게 하면 여러 역할을 병렬로 처리하거나, 복잡한 문제를 전문가 몇 명이 협업하듯 AI 내부에서 처리하게 할 수 있습니다.
- 재사용성과 안전성 – 한 번 정의한 서브에이전트는 프로젝트 전체에서 재사용 가능하며, 팀과 공유할 수도 있습니다. 또한 서브에이전트별로 허용 툴을 제한함으로써, 예컨대 테스트 실행 에이전트에게는 파일 쓰기 권한을 안 준다든지 하는 세밀한 권한 관리가 됩니다. 이를 통해 AI의 행동 반경을 역할별로 통제하여 안전성을 높일 수 있습니다.
- 예시 – 코드 리뷰 봇 서브에이전트를 만들어 두면, Claude가 `/review` 명령 시 해당 에이전트로 하여금 코드 변경분을 검사하고 피드백 생성을 맡게 합니다. 이 에이전트는 사전에 "코드 리뷰어"로서의 시스템 프롬프트(성능, 보안 체크리스트 등)가 주어져 있고, 필요 도구(예: `npm run lint`)만 허용된 상태로 동작합니다. 그 결과 전문화된 리뷰 출력을 얻을 수 있습니다 (마치 AI 안에 전문 리뷰어를 한 명 넣은 효과).

SuperClaude 프레임워크

- 컨텍스트 엔지니어링 프레임워크 – SuperClaude는 오픈소스 프로젝트로, Claude Code를 위한 경량 설정 프레임워크입니다. 추가 코딩 없이 구성 파일과 템플릿만으로 Claude Code를 특정 맥락에 최적화된 개발 파트너로 바꿔줍니다. 즉, 개발원칙을 코드로 자동 적용해주는 일종의 AI 강화 패키지입니다.
- 19개의 특화 명령어 제공 – SuperClaude는 개발 생애주기를 아우르는 여러 slash 명령을 추가합니다. 예를 들어 `/build`, `/test`, `/review`, `/deploy` 같은 명령으로 특정 작업을 수행하면, 그 내부에서 적절한 프롬프트와 서브에이전트 호출 등이 일어나 일련의 작업이 자동화됩니다. 덕분에 사용자는 고수준의 명령만 내리면 세부 작업은 AI가 알아서 실행합니다.
- 9가지 인지 페르소나 – SuperClaude는 아키텍트, 백엔드 전문가, 보안 전문가, QA 등 9개의 AI 페르소나를 내장하고 있습니다. 명령 실행 시 `--persona-architect`와 같이 옵션을 주면, 해당 분야에 맞는 시각으로 문제를 검토하고 해결책을 제시합니다. 이는 Claude Code의 Sub-Agent와 유사한 개념으로, 역할별 사고방식을 미리 설계해 둔 것입니다.
- GitHub 워크플로우 개선 – SuperClaude는 Git과 통합되어 커밋 메시지 자동 생성, 체인지로그 자동 작성, PR 코드 리뷰 자동 수행 등을 지원합니다. 예를 들어 `/commit` 명령을 쓰면 Claude가 변경된 코드 맥락을 파악해 의미 있는 커밋 메시지를 제안하고, `/changelog`를 하면 여러 커밋을 모아 요약된 변경 기록을 만들어줍니다. 또한 `/review` 명령으로 보안/성능 관점까지 고려한 리뷰 보고서를 얻을 수 있어, 리뷰 품질이 향상됩니다.

Claude Code Hooks로 자동화 강화

- 훅(Hook)의 개념 – Claude Code는 훅 기능을 통해, AI 동작의 특정 지점에 사용자 정의 스크립트를 실행할 수 있습니다. 훅은 정해진 이벤트(예: 툴 실행 전후, 노티fication 발생 시, 응답 완료 시 등)에 자동으로 실행되며, 이를 활용해 일정한 동작을 강제하거나 환경을 제어합니다. LLM에게 프롬프트로 매번 지시하는 대신, 훅에 스크립트를 등록해두면 해당 이벤트마다 항상 코드가 실행됩니다.
- 활용 예시 – Hooks로 구현할 수 있는 자동화는 다양합니다:
  - 작업 중 Claude가 파일을 수정하면 직후에 코드 포매터(`prettier`, `gofmt` 등)를 자동 실행하여 일관된 스타일을 유지.
  - Claude가 실행한 bash 명령을 로깅하여, 어떤 명령이 수행됐는지 `~/.claude/commands.log`에 누적 (감사 및 디버깅 용도).
  - Claude가 생성한 코드에 금지된 패턴이 있으면(예: `console.log` 남용) 자동 피드백을 주고 커밋을 막음.
  - 알림 커스터마이즈: Claude가 입력 대기 상태일 때 슬랙이나 데스크톱 알림을 보내기.
- 훅 구현 방법 – Claude Code에서 `/hooks` 명령으로 인터랙티브 설정을 열고, 이벤트 종류별로 훅을 추가할 수 있습니다. 예를 들어 PreToolUse 이벤트에 `matcher: Bash`를 추가하고 `jq` 커맨드를 등록하면, 모든 Bash 명령 실행 전에 해당 jq 스크립트가 실행되어 명령을 로깅하거나 차단할 수 있습니다. 훅은 프로젝트 단위 또는 전체 사용자 단위

(Section) AI를 활용한 코드 리뷰

폭증하는 코드에 대한 리뷰 부담

- 양이 많다! – 바이브 코딩은 짧은 시간에 대량의 코드와 문서를 생성하는 경향이 있습니다. AI가 개발을 가속한 만큼, 한 번의 기능 추가나 리팩터링에서 변경되는 코드 라인이 수천 줄에 달할 수 있습니다. 이처럼 변경 규모가 클 때 사람이 모든 Diff를 꼼꼼히 리뷰하기란 현실적으로 어렵습니다. 특히 주니어 개발자 입장에서는 어디부터 봐야 할지 막막할 정도의 패치도 AI는 순식간에 만들어냅니다.
- 휴먼 리뷰 한계 – 리뷰어는 보통 로직 버그, 코드 스타일, 성능, 보안 등 여러 측면을 검토해야 하지만, 변경이 방대하면 주의력이 분산되어 일부를 놓치기 쉽습니다. 또한 반복적이거나 보일러플레이트성 코드 변경에 시간을 쓰느라 정작 중요한 설계상의 문제를 못 발견할 수도 있습니다. 결국 버그가 리뷰를 통과해 배포되는 상황이 발생할 우려가 있습니다.
- 심리적 압박 – AI가 생성한 코드는 "기계가 했으니 맞겠지"라는 잘못된 신뢰를 주거나, 반대로 "혹시 엉망이면 어쩌지?" 하는 불안을 주기도 합니다. 전자는 리뷰 소홀로, 후자는 과도한 검토로 이어져 리뷰 효율 저하를 초래합니다. 또한 경험상 AI 코드에는 주석이나 설명이 부족한 경우가 많아, 리뷰어가 맥락을 이해하는 데 더 애를 먹습니다.

AI 코드 리뷰 보조

- AI에게 리뷰 맡기기? – 해결책 중 하나는 AI를 코드 리뷰어로 활용하는 것입니다. Claude 같은 모델에게 PR의 Diff를 입력으로 주고 "코드 리뷰를 해줘"라고 하면, 변경 요약, 개선 사항, 잠재 버그 지적 등을 척척 내놓습니다. 실제 SuperClaude 등의 도구는 보안/성능 관점까지 포괄하는 상세 리뷰 리포트를 자동 생성해줍니다. 사람 리뷰 전에 AI가 1차로 훑어서 중대한 이슈를 선별해주는 것이죠.
- 요약 및 체크리스트 제공 – AI 리뷰의 큰 장점은 한눈에 보는 요약을 제공한다는 점입니다. PR에 포함된 커밋들의 의도와 변경된 주요 모듈 간 관계를 서머리해서, 리뷰어에게 전반적 그림을 줍니다. 또한 "이 부분은 스레드 세이프티 검토 필요", "입력 유효성 검사 추가 고려" 등 체크리스트 형태로 검토 포인트를 나열해줄 수 있습니다. 리뷰어는 이를 참고해 빼먹기 쉬운 항목까지 놓치지 않고 확인하게 됩니다.
- 잠재 문제 경고 – AI는 학습된 지식을 바탕으로, 코드에서 향후 문제가 될 만한 부분을 경고해줄 수도 있습니다. 예컨대 "이 패턴은 메모리 누수 위험 있음", "이 API 사용은 Deprecated" 등 사람 리뷰어가 놓칠 수 있는 점도 짚어냅니다. 특히 보안 취약점이나 성능 병목 같은 부분은 AI가 관련 데이터셋을 학습해서 알고 있는 경우가 많아, 의외로 유용한 지적을 합니다.
- 리뷰 시간 절약 – 결과적으로 리뷰어는 AI가 준 요약과 지적 사항을 중심으로 짧은 시간 내 핵심을 파악하고, 중요한 문제 위주로 깊게 검토할 수 있게 됩니다. 반복적 스타일 문제나 자잘한 버그 지적은 AI가 처리하니, 사람은 설계적 타당성 등 고차원적인 리뷰에 집중 가능합니다. 한 연구에 따르면 AI 보조를 활용한 코드 리뷰는 평균 30% 이상 시간이 단축되었다는 보고도 있습니다.

PR 리뷰 자동화 사례

- PR 열리면 AI 리뷰 시작 – 예를 들어 GitHub Actions로 PR 생성 이벤트가 발생하면 Claude Code를 호출해 자동으로 리뷰를 수행하게 할 수 있습니다. Anthropic 액션을 이용하면, 특정 PR에 `@claude review` 라는 댓글을 다는 것만으로 Claude가 해당 PR의 코드를 분석해 리뷰 코멘트를 작성합니다. 이러한 자동 리뷰에는 앞서 언급한 요약, 체크리스트, 발견된 문제점 등이 포함됩니다.
- 포맷팅/마이너 수정 자동 처리 – AI 리뷰는 단순 지적에서 그치지 않고, 자동 수정을 제안하거나 바로 적용할 수도 있습니다. Anthropic 팀 사례로, PR 코멘트 중 코드 포매팅 요청이나 간단한 변수명 변경은 Claude가 감지하여 자동 커밋으로 반영하도록 했습니다. 즉, 리뷰어가 "이 변수명은 의미가 모호하니 `userCount` 로 변경 바랍니다"라고 댓글을 남기면 Claude가 곧바로 해당 변경을 커밋해주는 식입니다. 이로써 리뷰어는 큰 문제에 집중하고, 자잘한 수정은 AI가 책임지게 되어 리뷰 효율이 극대화됩니다.
- 지속 피드백 루프 – 리뷰어(인간)이 남긴 코멘트에 대해서도 AI가 즉각 반응하게 할 수 있습니다. 예컨대 리뷰어가 "이 로직의 복잡도가 걱정됩니다"라고 적으면, Claude가 해당 부분을 리팩터링하여 새 커밋을 올리거나, 리뷰어에게 "이 부분은 X 이유로 이렇게 구현되었으며, 대안으로 Y를 고려해볼 수 있습니다"라고 답변하는 등 대화형 코드 리뷰가 펼쳐집니다. 이러한 실시간 피드백 루프를 통해, 리뷰 과정에서 발견된 문제가 신속히 수정되고 재검토되어 PR 승인까지 걸리는 시간을 단축시킵니다.
- 품질 게이트 – 마지막으로, AI를 리뷰 품질 감시자로 삼는 방법도 있습니다. 사람 리뷰가 모두 끝나고 PR을 머지하려 할 때, AI가 변화된 코드와 리뷰 코멘트를 다시 한번 훑어서 "혹시 아직 논의 안 된 잠재 이슈는 없는지" 체크하게 할 수

실습 시나리오: ToDo List 앱 (Vibe Coding)

- 시나리오 소개 – 이제 간단한 웹 애플리케이션을 만들어보는 실습을 가정해보겠습니다. ToDo List 애플리케이션을 바이브 코딩 방식으로 구현할 예정입니다. 이 앱은 할 일을 추가하고, 체크하고, 삭제하는 기본 기능을 갖춘 작은 프로젝트입니다. 주니어 개발자분들이 AI와 함께 개발하는 흐름을 체험할 수 있도록 구성했습니다.
- 목표 – Claude Code를 사용해 프런트엔드와 백엔드 코드를 생성하고, 문서와 테스트까지 자동화하는 DevOps 파이프라인을 경험해봅니다. 이 과제를 통해 앞서 배운 원칙들(Tidy First, 작은 이슈 단위, CI 활용 등)을 실제 적용하는 법을 익힐 것입니다. 또한 실습 과정에서 AI의 한계나 에러 상황을 다뤄보며, 언제 사람의 개입이 필요한지도 확인해볼 것입니다.
- 예제 리포지토리 – Hands-On-Vibe-Coding/ToDoList (GitHub) 라는 저장소에 단계별 구현 예제가 준비되어 있습니다. 각 브랜치에는 특정 단계(예: UI 스켈레톤 생성, API 구현, 테스트 추가 등)의 결과물이 담겨 있고, 관련 `CLAUDE.md` 와 프롬프트 대화 내용도 기록되어 있습니다. 이를 참고하면서 진행할 예정입니다.

실습 단계 1: 요구 사항 도출 및 이슈 작성

- 기능 목록 정리 – 먼저 ToDo List 앱의 요구사항을 정의합니다. 예를 들어 "아이템 추가", "아이템 체크 (완료 표시)", "아이템 삭제", "필터 (전체/미완료/완료)" 등의 기능을 리스트업합니다. 이러한 요구사항을 각각 GitHub 이슈로 만듭니다 (예: Issue #1: "할 일 추가 기능", #2: "완료 상태 토글", #3: "삭제 기능", #4: "필터 UI 구현").
- Claude에게 맥락 제공 – `CLAUDE.md` 파일에 프로젝트 개요와 주요 기술 스택(예: React 프론트엔드, Node/Express 백엔드, SQLite DB 등)을 적어둡니다. 또한 각 이슈의 내용을 Claude가 볼 수 있도록, 이슈 설명에 구체적인 요구 (화면 예시, API 스펙 등)를 작성해둡니다. Claude Code는 추후 `gh` CLI를 통해 이슈 내용을 불러와 이해할 것이므로, 이슈 자체가 AI에 대한 설계서 역할을 하도록 충실히 작성하는 것이 포인트입니다.
- 환경 설정 확인 – 실습용 저장소를 로컬에 클론하고, Claude Code를 해당 디렉토리에서 실행할 준비를 합니다. `npm init` 등으로 기본 프로젝트 구조를 만들어두고, `gh` CLI 로그인이 되어있는지, 필요한 MCP 서버나 API 키 설정이 되어있는지 확인합니다. 이제 본격적인 vibe coding을 시작할 준비가 되었습니다.

실습 단계 2: AI와 함께 기능 구현

- Issue #1 – "할 일 추가 기능": 첫 이슈부터 Claude와 작업을 시작합니다. Claude Code 터미널에서 "`gh issue view 1`" 등의 명령을 통해 이슈 상세를 읽게 하고, `claude` 프롬프트로 "이 이슈를 해결하는 코드 작성 시작해 줘"라고 입력합니다. Claude는 현재 프로젝트 구조에 맞춰 (예: React 컴포넌트 생성, Express 엔드포인트 추가 등) 코드 변경을 제안할 것입니다.
    - Claude가 제안한 코드(예: `TodoList.jsx` 컴포넌트와 `POST /todos` API 등)를 검토합니다. Tidy First 원칙에 따라, 만약 기존 구조를 개선해야 한다면 (예: "UI에 상태 관리 로직 분리 필요") Claude에게 먼저 리팩터링을 지시합니다.
    - Claude의 변경이 마음에 들면 "승인"하여 실제 파일에 적용합니다. Claude Code가 자동으로 파일을 생성/수정하고, `git diff`로 결과를 보여줄 것입니다.
- Issue #2 – "완료 상태 토글": 다음 이슈로 넘어가, "완료/미완료 표시를 토글하는 기능을 추가해줘"라고 Claude에게 요청합니다. Claude는 이전 단계에서 만든 코드를 인식하고 있으므로, 해당 부분에 체크박스 onChange 핸들러와 `/todos/:id/toggle` API 등을 생성해줄 것입니다. 이때도 하나의 이슈에 집중하여, Claude가 산으로 가지 않도록 필요한 경우 "이 이슈 범위를 넘지 말고 진행해"라고 추가 지시를 합니다.
- AI와 대화하며 개발 – 구현 도중 Claude가 질문을 할 수도 있습니다 (예: "완료 토글 시 DB 필드 어떤 걸 쓰나요?"). 그러면 적절히 답변을 주면 페어 프로그래밍하듯 진행합니다. 또한 Claude가 잘못된 방향으로 코드를 변경하려 하면

실습 단계 3: 자동 테스트 및 CI 검증

- 테스트 코드 생성 – 기능 구현이 완료될 때마다, Claude에게 해당 부분의 테스트 작성을 요청합니다. 예를 들어 Issue #1 완료 후 "이 기능에 대한 유닛 테스트와 통합 테스트 작성해줘"라고 지시하면, Claude가 Jest 테스트 파일이나 API 통합 테스트 코드를 생성해줄 것입니다. 생성된 테스트를 로컬에서 실행해보고, 통과하지 못하면 Claude와 함께 디버깅/수정합니다. Claude Code는 `npm run test` 명령을 직접 실행해볼 수도 있으므로, AI에게 "테스트 돌려봐"라고 해도 됩니다.
- CI 설정 확인 – `.github/workflows/ci.yml` 파일이 있다면, PR 시 자동으로 테스트가 돌도록 설정되어 있을 것입니다. (앞서 Claude에게 파이프라인 설정을 맡겼다면 이 단계는 완료되어 있겠죠.) 로컬에서 모든 테스트가 통과하면 코드를 커밋하고 원격 브랜치에 푸시합니다.
- GitHub Actions 확인 – PR을 열면 GitHub Actions가 실행되어 CI 파이프라인이 구동됩니다. 빌드, 린트, 테스트가 자동으로 수행되고 그 결과가 PR에 표시됩니다. 만약 실패하거나 경고가 있으면, Claude Code GitHub Action을 통해 AI가 자동 수정하거나, 직접 로그를 보고 Claude에게 해당 이슈를 고치도록 지시합니다. 예를 들어 "CI에서 로그인 테스트 실패 – 원인 분석해줘"라고 Claude에게 요청하면, Claude가 로그를 분석해 "아, API 경로 오타가 있네요"와 같이 알려줄 수 있습니다.
- 반복 개선 – CI가 성공 (녹색)할 때까지, AI와 함께 코드를 수정하고 push하는 사이클을 반복합니다. 작은 이슈 단위로 작업했기에, 문제가 생겨도 디버깅 범위가 좁아 빠르게 원인을 찾고 해결할 수 있습니다. 결국 모든 체크를 통과하면 해당 PR을 머지하고 다음 단계로 넘어갑니다. 이 과정에서 AI가 테스트를 모두 작성해주고 실행까지 해주니, 개발자는 품

실습 단계 4: 배포 및 반복

- 코드 머지 및 배포 – 모든 기능이 구현되어 메인 브랜치에 머지되면, CD(배포) 단계가 진행됩니다 (만약 설정되어 있다면). 예를 들어 Heroku나 Vercel에 자동 배포되도록 했으면, 최신 코드가 프로덕션에 반영되고 실제 앱에서 동작을 테스트해볼 수 있습니다. 이때도 Claude Code를 활용해 "배포 후 간단한 건강 체크 테스트"를 수행할 수도 있습니다.
- 리뷰 및 회고 – 구현된 ToDo List 앱을 동료들과 함께 리뷰합니다. Claude가 생성한 코드 중 개선할 점은 없는지(과도한 중첩, 변수 네이밍 등) 사람 리뷰어가 확인하고, 필요하면 리팩터링 이슈를 새로 만들어 Claude와 처리합니다. AI 코드 리뷰 결과도 참고하여, 잠재적 문제를 선제적으로 수정합니다. 예컨대 성능상 O(n^2) 알고리즘이 발견되면 이슈로 등록해 다음 스프린트에 다루기로 합니다.
- 다음 이터레이션 준비 – 이번 실습에서는 기본 기능만 구현했지만, 추가로 "마감기한 추가", "다중 사용자 지원" 같은 기능을 확장해나갈 수 있습니다. 새로운 이슈를 작성하고 컨텍스트 엔지니어링을 한 단계 업그레이드해서 Claude에게 더 큰 역할을 맡겨볼 수 있습니다. 예를 들어, "프로젝트 README 업데이트"를 Claude에게 시켜본다든지, "Docker 컨테이너 설정"을 요청해볼 수도 있습니다. 이런 식으로 프로젝트 규모를 키워가며 AI 도우미와 함께 성장해 나가는 것이 바이브 코딩의 묘미입니다.
- 정리 – Hands-On-Vibe-Coding/ToDoList 저장소에 최종 코드와 Claude와 주고받은 주요 프롬프트 대화를 정리해 커밋합니다. 이를 통해 다른 팀원들도 AI 협업 과정을 이해할 수 있고, 이후 비슷한 프로젝트에 참고 사례로 활용할 수 있습니다.

결론: AI + 엔지니어링 = 🚀

- 학습 곡선 – 주니어 개발자에게 바이브 코딩은 처음엔 생소하지만, 올바른 방식으로 접근하면 개발 실력을 키워주는 촉진제가 됩니다. AI의 제안을 분석하고 함께 디버깅하는 과정에서 다양한 코드 패턴을 접하고, 모범 사례를 몸에 익힐 수 있습니다. 물론 AI가 실수도 하므로, 비판적 시각으로 결과물을 검증하는 자세를 잃지 말아야 합니다.
- 생산성과 품질 – 바이브 코딩을 통해 생산성의 비약적 향상을 누릴 수 있었지만, 동시에 강조된 것은 소프트웨어 공학적 원칙의 중요성입니다. TDD, 리팩터링, 작은 커밋, 코드 리뷰 등 기본기를 지켜준 덕분에 AI의 출력물도 의미있는 소프트웨어로 거듭날 수 있었습니다. 이 균형을 유지한다면, 적은 인원으로도 대규모 프로젝트를 충분히 감당할 수 있을 것입니다.
- 미래 개발자의 역할 – AI 도구가 발전할수록 개발자의 역할은 단순 코딩에서 설계자, 검증자, 윤리적 판단자로 변화할 것입니다. 여러분은 AI라는 강력한 엔진을 단 자동차의 운전자입니다. 컨텍스트 엔지니어링과 엔지니어링 원칙이라는 두 핸들을 잘 잡고 운전한다면, 어떤 어려운 소프트웨어 도전도 거뜬히 해낼 수 있을 것입니다. 빠르게 변하는 시대에 AI와 협업하는 개발 역량을 갖추는 여러분이 되길 바랍니다!

디스코드 커뮤니티 안내

강의에 대한 질문이 있으시면 디스코드를 통해서 자유롭게 질문해 주세요