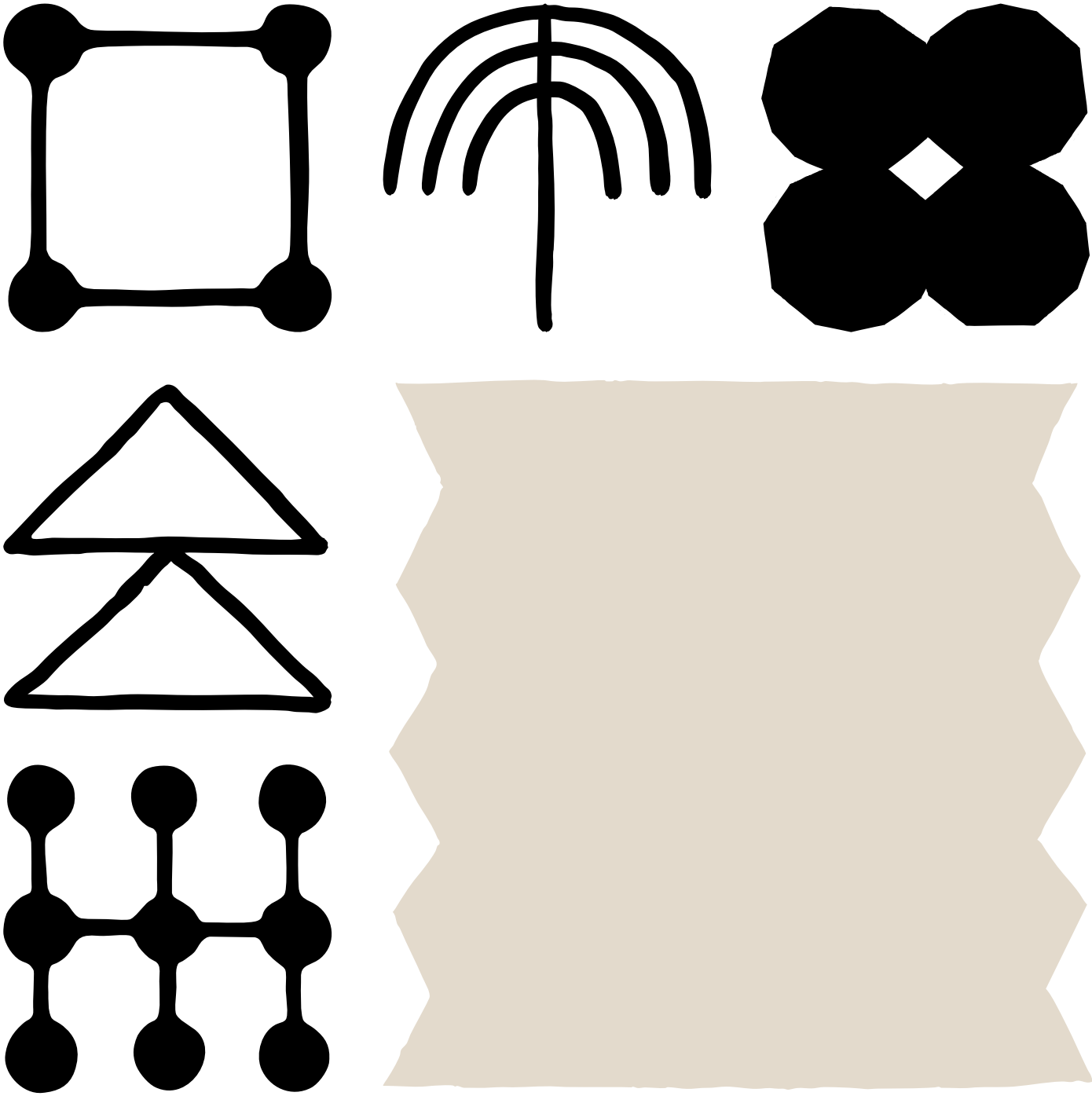


Claude Code: 에이전틱 코딩을 위한 모범 사례

출간됨2025년 4월 18일

Claude Code는 에이전틱 코딩을 위한 명령줄 도구입니다. 이 글에서는 다양한 코드베이스, 언어, 환경에서 Claude Code를 사용할 때 효과적인 것으로 입증된 팁과 요령들을 다룹니다.

최근 저희는Claude Code를 출시했습니다, 에이전틱 코딩을 위한 명령줄 도구입니다. 연구 프로젝트로 개발된 Claude Code는 Anthropic 엔지니어와 연구자들이 Claude를 자신들의 코딩 워크플로우에 보다 자연스럽게 통합할 수 있는 방법을 제공합니다.

Claude Code는 의도적으로 저수준이고 특정 관점을 강요하지 않으며, 특정 워크플로우를 강제하지 않으면서 원시 모델에 가까운 접근을 제공합니다. 이러한 설계 철학은 유연하고 사용자 정의 가능하며 스크립트 작성이 가능하고 안전한 파워 툴을 만들어냅니다. 강력하지만, 이러한 유연성은 에이전틱 코딩 도구를 처음 접하는 엔지니어들에게 학습 곡선을 제시합니다—적어도 그들이 자신만의 모범 사례를 개발할 때까지는 말입니다.

이 게시물은 Anthropic의 내부 팀과 다양한 코드베이스, 언어, 환경에서 Claude Code를 사용하는 외부 엔지니어들 모두에게 효과적임이 입증된 일반적인 패턴들을 설명합니다. 이 목록의 내용은 확정된 것도 아니고 보편적으로 적용되는 것도 아닙니다. 이러한 제안들을 시작점으로 여기시기 바랍니다. 실험해보시고 여러분에게 가장 적합한 방법을 찾아보시기를 권장합니다!

더 자세한 정보를 찾고 계신가요? 다음에서 포괄적인 문서를 확인하세요claude.ai/code 이 게시물에서 언급된 모든 기능을 다루며 추가 예제, 구현 세부사항, 고급 기법을 제공합니다.

1. 설정 사용자 지정

Claude Code는 자동으로 컨텍스트를 프롬프트에 가져오는 에이전트 코딩 어시스턴트입니다. 이러한 컨텍스트 수집은 시간과 토큰을 소모하지만, 환경 튜닝을 통해 최적화할 수 있습니다.

a. 생성 `CLAUDE.md` 파일

`CLAUDE.md` 는 Claude가 대화를 시작할 때 자동으로 컨텍스트에 가져오는 특별한 파일입니다. 이는 다음과 같은 내용을 문서화하기에 이상적인 장소입니다:

- 일반적인 bash 명령어
- 핵심 파일 및 유틸리티 함수
- 코드 스타일 가이드라인
- 테스트 지침
- 저장소 에티켓 (예: 브랜치 명명, 병합 vs 리베이스 등)
- 개발자 환경 설정 (예: pyenv 사용, 작동하는 컴파일러)
- 프로젝트에 특정한 예상치 못한 동작이나 경고
- Claude가 기억했으면 하는 기타 정보

필수 형식은 없습니다 `CLAUDE.md` 파일의 경우. 간결하고 사람이 읽기 쉽게 유지하는 것을 권장합니다. 예를 들어:

```
# Bash 명령어
- npm run build: 프로젝트 빌드
- npm run typecheck: 타입 체커 실행

# 코드 스타일
- CommonJS(require)가 아닌 ES 모듈(import/export) 구문 사용
- 가능한 경우 import를 구조 분해할당으로 사용 (예: import { foo } from 'bar')

# 워크플로우
- 일련의 코드 변경을 완료한 후에는 반드시 타입 체크 실행
- 성능을 위해 전체 테스트 스위트가 아닌 단일 테스트 실행을 선호
```

복사

다음 위치에 `CLAUDE.md` 파일을 배치할 수 있습니다:

- **저장소의 루트**, 또는 `claude` 을 실행하는 곳 어디든지 (가장 일반적인 사용법). 이름을 `CLAUDE.md` 로 지정하고 git에 체크인하여 세션 간에 그리고 팀과 공유할 수 있도록 하거나 (권장), 이름을 `CLAUDE.local.md` 와 `.gitignore` it
- **디렉토리의 모든 상위 디렉토리**실행하는 곳 `claude` . 이는 모노레포에서 가장 유용하며, `claude` 에서 `root/foo` 를 실행하고, `CLAUDE.md` 파일들이 둘 다 `root/CLAUDE.md` 와 `root/foo/CLAUDE.md` 에 있습니다. 이 둘 모두 자동으로 컨텍스트에 포함될 것입니다
- **디렉토리의 모든 하위 항목**실행하는 곳 `claude` . 이는 위의 반대 경우이며, 이 경우 Claude는 하위 디렉토리의 파일들로 작업할 때 필요에 따라 `CLAUDE.md` 파일들을 가져올 것입니다
- **홈 폴더** (`~/.claude/CLAUDE.md`), 이는 모든 *claude* 세션에 적용됩니다

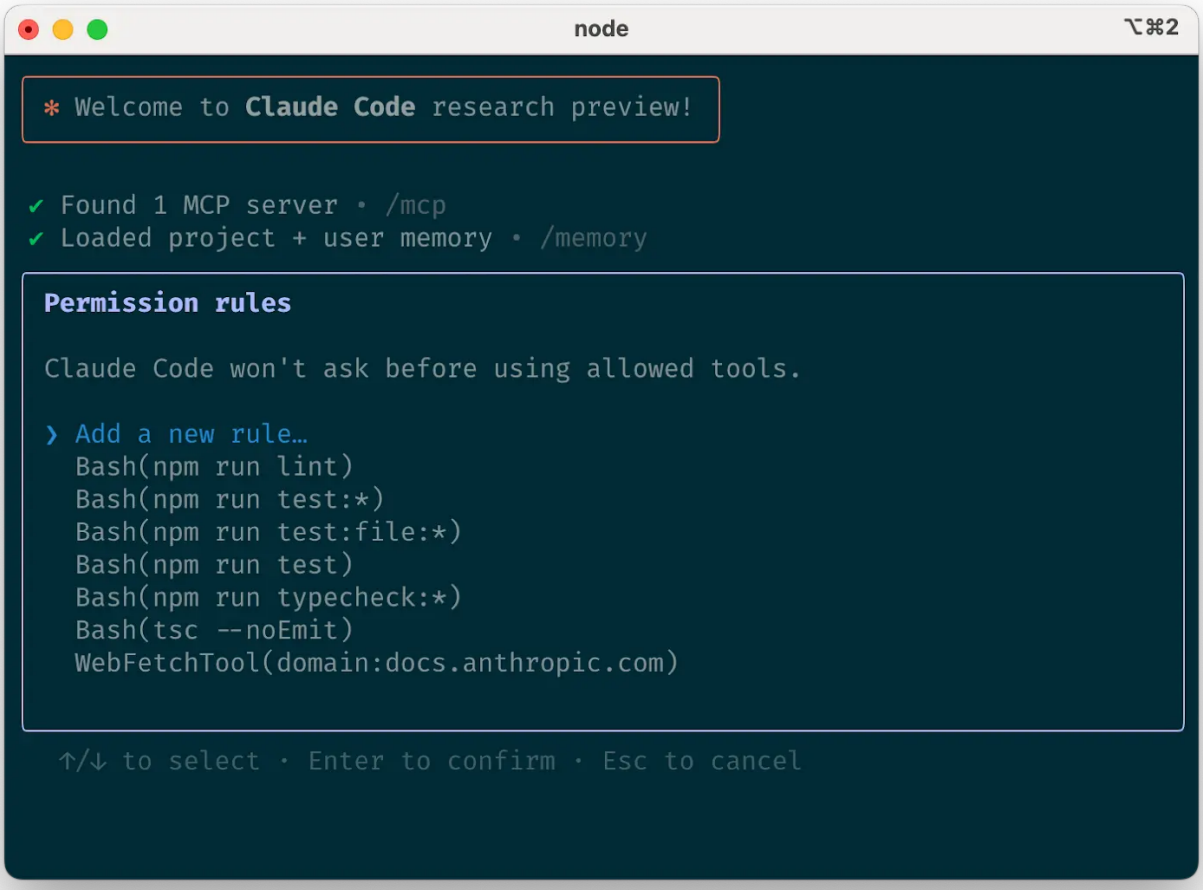
명령을 실행하면 `/init` Claude가 자동으로 `CLAUDE.md` 을 생성해 드립니다.

b. 당신의 조정 `CLAUDE.md` 파일

당신의 `CLAUDE.md` 파일들은 Claude의 프롬프트의 일부가 되므로, 자주 사용되는 프롬프트처럼 정제되어야 합니다. 흔한 실수는 효과를 반복 검증하지 않고 광범위한 내용을 추가하는 것입니다. 시간을 들여 실험하고 모델로부터 최상의 지시 수행을 이끌어내는 것이 무엇인지 파악하세요.

다음에 콘텐츠를 추가할 수 있습니다 `CLAUDE.md` 수동으로 또는 `#` 클로드가 관련된 내용에 자동으로 통합할 지침을 제공하는 것이 핵심입니다 `CLAUDE.md` 많은 엔지니어들이 사용합니다 `#` 코딩하는 동안 명령어, 파일, 스타일 가이드라인을 문서화하기 위해 자주 사용하며, 그런 다음 포함 `CLAUDE.md` 커밋의 변경사항으로 팀 멤버들도 혜택을 받습니다.

Anthropic에서는 때때로 `CLAUDE.md` 파일들을 통해프롬프트 개선기그리고 종종 지시사항을 조정하여 (예: "중요" 또는 "반드시"와 같은 강조 표현 추가) 준수도를 향상시킵니다.



c. Claude의 허용된 도구 목록 관리

기본적으로 Claude Code는 시스템을 수정할 수 있는 모든 작업에 대해 권한을 요청합니다: 파일 쓰기, 많은 bash 명령어, MCP 도구 등. 우리는 안전성을 우선시하기 위해 의도적으로 보수적인 접근 방식으로 Claude Code를 설계했습니다. 안전하다고 알고 있는 추가 도구를 허용하거나, 쉽게 되돌릴 수 있는 잠재적으로 안전하지 않은 도구(예: 파일 편집, `git commit`).

허용된 도구를 관리하는 네 가지 방법이 있습니다:

- **"항상 허용"을 선택하세요**세션 중에 메시지가 표시될 때.
- **사용하세요** `/permissions` **명령어를**Claude Code를 시작한 후 허용 목록에서 도구를 추가하거나 제거하기 위해. 예를 들어, 다음을 추가할 수 있습니다 `편집` 항상 파일 편집을 허용하거나, `Bash(git commit:*)` git 커밋을 허용하거나, `mcp__puppeteer__puppeteer_navigate` Puppeteer MCP 서버로 탐색을 허용합니다.
- **수동으로 편집** your `.claude/settings.json` 또는 `~/.claude.json` (팀과 공유하기 위해 전자를 소스 제어에 체크인하는 것을 권장합니다) .
- **사용하세요** `--allowedTools` **CLI 플래그** 세션별 권한을 위해.

d. GitHub를 사용하는 경우, gh CLI를 설치하세요

Claude는 `gh` CLI를 사용하여 GitHub와 상호작용하는 방법을 알고 있습니다. 이슈 생성, 풀 리퀘스트 열기, 댓글 읽기 등을 할 수 있습니다. `gh` 가 설치되지 않은 경우에도 Claude는 GitHub API나 MCP 서버(설치되어 있다면)를 사용할 수 있습니다.

2. Claude에게 더 많은 도구 제공하기

Claude는 당신의 셸 환경에 접근할 수 있어서, 당신이 직접 하는 것처럼 편의 스크립트와 함수 세트를 구축할 수 있습니다. 또한 MCP와 REST API를 통해 더 복잡한 도구들을 활용할 수도 있습니다.

a. bash 도구와 함께 Claude 사용하기

Claude Code는 당신의 bash 환경을 상속받아 모든 도구에 접근할 수 있습니다. Claude는 unix 도구와 같은 일반적인 유틸리티들을 알고 있지만 `gh` , 지시사항 없이는 사용자 정의 bash 도구에 대해 알지 못합니다:

1. 사용 예시와 함께 Claude에게 도구 이름을 알려주세요
2. Claude에게 실행하라고 말하세요 `--help` 도구 문서를 보기 위해
3. 자주 사용하는 도구들을 문서화하세요 `CLAUDE.md`

b. MCP와 함께 Claude 사용하기

Claude Code는 MCP 서버와 클라이언트 역할을 모두 수행합니다. 클라이언트로서 여러 MCP 서버에 연결하여 세 가지 방법으로 도구에 액세스할 수 있습니다:

- **프로젝트 설정에서** (해당 디렉토리에서 Claude Code를 실행할 때 사용 가능)
- **전역 설정에서** (모든 프로젝트에서 사용 가능)
- **체크인된 `.mcp.json` 파일에서** (코드베이스에서 작업하는 모든 사람이 사용 가능). 예를 들어, Puppeteer와 Sentry 서버를 추가할 수 있습니다 `.mcp.json` , 따라서 당신의 저장소에서 작업하는 모든 엔지니어가 이를 즉시 사용할 수 있습니다.

MCP로 작업할 때, 구성 문제를 식별하는 데 도움이 되도록 Claude를 `--mcp-debug` 플래그와 함께 실행하는 것도 유용할 수 있습니다.

c. 사용자 정의 슬래시 명령어 사용

반복적인 워크플로우(디버깅 루프, 로그 분석 등)의 경우, 프롬프트 템플릿을 다음 위치의 Markdown 파일에 저장하세요: `.claude/commands` 폴더에 저장됩니다. 이들은 `/` 를 입력할 때 슬래시 명령어 메뉴를 통해 사용할 수 있게 됩니다. 이러한 명령어들을 git에 커밋하여 팀의 나머지 구성원들이 사용할 수 있도록 만들 수 있습니다.

사용자 정의 슬래시 명령어는 특별한 키워드를 포함하여 `$ARGUMENTS` 명령어 호출 시 매개변수를 전달할 수 있습니다.

예를 들어, Github 이슈를 자동으로 가져와서 수정하는 데 사용할 수 있는 슬래시 명령어는 다음과 같습니다:

```
GitHub 이슈를 분석하고 수정하세요: $ARGUMENTS.

다음 단계를 따르세요:

1. `gh issue view`를 사용하여 이슈 세부사항을 확인하세요
2. 이슈에서 설명된 문제를 이해하세요
3. 관련 파일을 찾기 위해 코드베이스를 검색하세요
4. 이슈를 수정하기 위해 필요한 변경사항을 구현하세요
5. 수정사항을 검증하기 위해 테스트를 작성하고 실행하세요
6. 코드가 린팅과 타입 검사를 통과하는지 확인하세요
7. 설명적인 커밋 메시지를 작성하세요
8. 푸시하고 PR을 생성하세요

모든 GitHub 관련 작업에는 GitHub CLI(`gh`)를 사용하는 것을 기억하세요.
```

복사

위의 내용을 `.claude/commands/fix-github-issue.md`에 넣으면 Claude Code에서 `/project:fix-github-issue` 명령어로 사용할 수 있습니다. 예를 들어 `/project:fix-github-issue 1234`를 사용하여 Claude가 이슈 #1234를 수정하도록 할 수 있습니다. 마찬가지로, 모든 세션에서 사용하고 싶은 명령어를 `~/.claude/commands` 폴더에 개인 명령어로 추가할 수 있습니다.

3. 일반적인 워크플로우 시도하기

Claude Code는 특정 워크플로우를 강요하지 않으므로, 원하는 방식으로 사용할 수 있는 유연성을 제공합니다. 이러한 유연성이 제공하는 공간 내에서, Claude Code를 효과적으로 사용하기 위한 몇 가지 성공적인 패턴이 사용자 커뮤니티에서 나타났습니다:

a. 탐색, 계획, 코딩, 커밋

이 다재다능한 워크플로우는 많은 문제에 적합합니다:

1. **Claude에게 관련 파일, 이미지 또는 URL을 읽도록 요청하세요**일반적인 지침("로깅을 처리하는 파일을 읽어주세요") 또는 구체적인 파일명("logging.py를 읽어주세요")을 제공하되, 아직은 코드를 작성하지 말라고 명시적으로 말하세요.
    1. 이는 특히 복잡한 문제에 대해 하위 에이전트를 적극적으로 활용해야 하는 워크플로우 부분입니다. Claude에게 하위 에이전트를 사용하여 세부 사항을 검증하거나 특정 질문을 조사하도록 지시하는 것은, 특히 대화나 작업 초기에, 효율성 손실 측면에서 큰 단점 없이 컨텍스트 가용성을 보존하는 경향이 있습니다.
2. **Claude에게 특정 문제에 접근하는 방법에 대한 계획을 세우도록 요청하세요**확장된 사고 모드를 활성화하기 위해 "think"라는 단어를 사용하는 것을 권장합니다. 이는 Claude가 대안들을 더 철저히 평가할 수 있도록 추가적인 계산 시간을 제공합니다. 다음과 같은 특정 구문들은 시스템에서 사고 예산 수준을 직접적으로 증가시키는 것과 연결됩니다: "think" < "think hard" < "think harder" < "ultrathink." 각 수준은 Claude가 사용할 수 있는 사고 예산을 점진

적으로 더 많이 할당합니다.

    1. 이 단계의 결과가 합리적으로 보인다면, Claude에게 계획이 담긴 문서나 GitHub 이슈를 생성하도록 요청할 수 있습니다. 그러면 구현(3단계)이 원하는 대로 되지 않을 경우 이 지점으로 되돌아갈 수 있습니다.

3. **Claude에게 해결책을 코드로 구현하도록 요청하세요**. 이는 또한 솔루션의 각 부분을 구현하면서 해결책의 합리성을 명시적으로 검증하도록 요청하기에 좋은 시점입니다.
4. **Claude에게 결과를 커밋하고 풀 리퀘스트를 생성하도록 요청하세요**. 관련이 있다면, 이때 Claude가 방금 수행한 작업에 대한 설명과 함께 README나 변경 로그를 업데이트하도록 하는 것도 좋습니다.

1-2단계는 매우 중요합니다. 이 단계들이 없으면 Claude는 바로 솔루션 코딩으로 넘어가는 경향이 있습니다. 때로는 그것이 원하는 것일 수도 있지만, Claude에게 먼저 조사하고 계획하도록 요청하면 사전에 깊은 사고가 필요한 문제들에 대한 성능이 크게 향상됩니다.

b. 테스트 작성, 커밋; 코딩, 반복, 커밋

이는 단위 테스트, 통합 테스트 또는 엔드투엔드 테스트로 쉽게 검증할 수 있는 변경 사항에 대한 Anthropic이 선호하는 워크플로우입니다. 테스트 주도 개발(TDD)은 에이전틱 코딩과 함께 더욱 강력해집니다:

1. **Claude에게 예상되는 입력/출력 쌍을 기반으로 테스트를 작성하도록 요청하세요**. 테스트 주도 개발을 하고 있다는 사실을 명시적으로 알려주어 코드베이스에 아직 존재하지 않는 기능에 대해서도 모의 구현을 만들지 않도록 하세요.
2. **Claude에게 테스트를 실행하고 실패하는지 확인하도록 지시하세요**. 이 단계에서는 구현 코드를 작성하지 말라고 명시적으로 말하는 것이 종종 도움이 됩니다.
3. **Claude에게 테스트를 커밋하도록 요청하세요**만족스러울 때까지 작업하세요.
4. **Claude에게 테스트를 통과하는 코드를 작성하도록 요청하세요**, 테스트를 수정하지 말라고 지시하면서요. Claude에게 모든 테스트가 통과할 때까지 계속하라고 말하세요. 보통 Claude가 코드를 작성하고, 테스트를 실행하고, 코드를 조정하고, 다시 테스트를 실행하는 과정을 몇 번 반복해야 합니다.
    1. 이 단계에서는 독립적인 하위 에이전트들과 함께 구현이 테스트에 과적합되지 않았는지 검증하도록 요청하는 것이 도움이 될 수 있습니다
5. **Claude에게 코드를 커밋하도록 요청하세요**변경사항에 만족하면 말이죠.

Claude는 반복 작업을 수행할 명확한 목표가 있을 때 최고의 성능을 발휘합니다—시각적 목업, 테스트 케이스, 또는 다른 종류의 출력물 같은 것들 말이죠. 테스트와 같은 예상 출력을 제공함으로써 Claude는 변경사항을 만들고, 결과를 평가하며, 성공할 때까지 점진적으로 개선할 수 있습니다.

c. 코드 작성, 결과 스크린샷, 반복

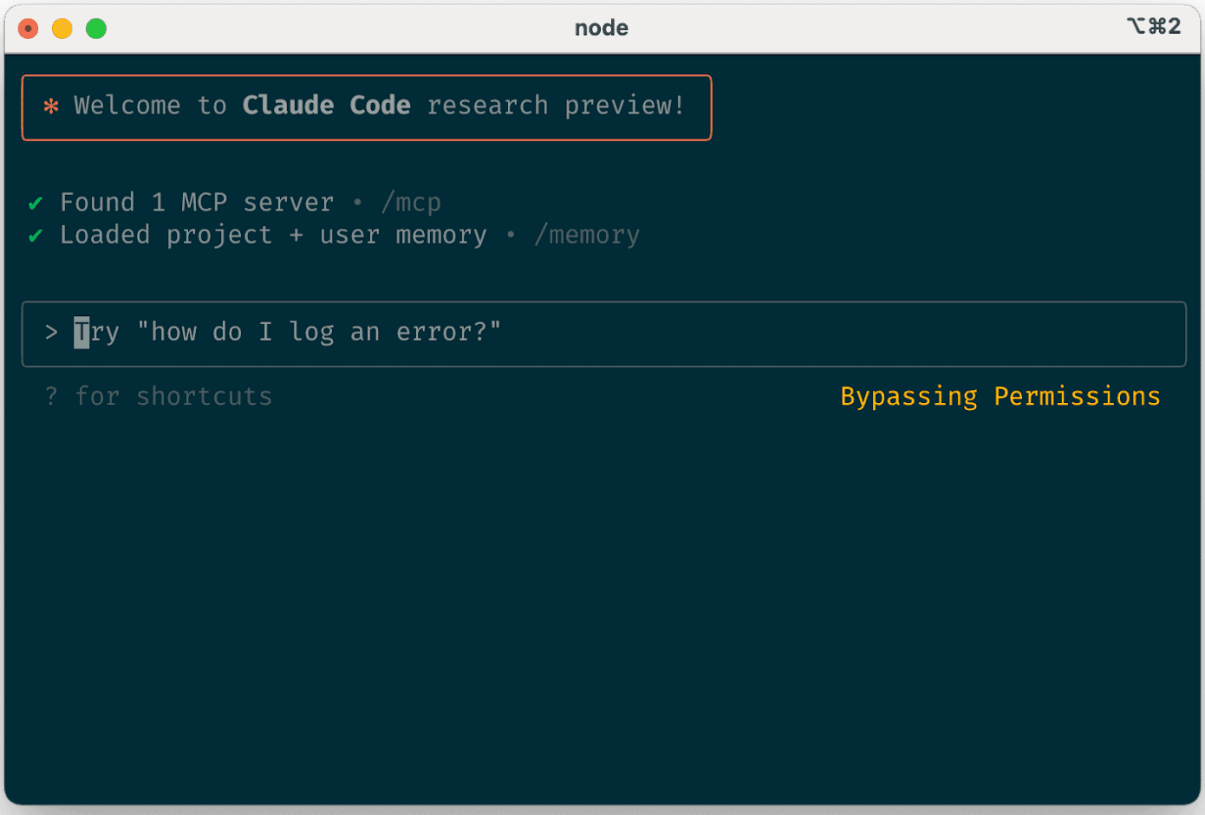
테스팅 워크플로우와 유사하게, Claude에게 시각적 목표를 제공할 수 있습니다:

1. **Claude에게 브라우저 스크린샷을 찍을 수 있는 방법을 제공하세요** (예: Puppeteer MCP 서버, iOS 시뮬레이터 MCP 서버, 또는 스크린샷을 Claude에 수동으로 복사/붙여넣기).
2. **시각적 목업을 Claude에게 제공** 이미지를 복사/붙여넣기하거나 드래그 앤 드롭하거나, Claude에게 이미지 파일 경

로를 제공하여.

3. **Claude에게 디자인을 코드로 구현하도록 요청** 하고, 결과의 스크린샷을 찍어서 목업과 일치할 때까지 반복.
4. **Claude에게 커밋하도록 요청하세요**만족스러울 때.

인간과 마찬가지로 Claude의 출력물은 반복을 통해 크게 개선되는 경향이 있습니다. 첫 번째 버전이 좋을 수 있지만, 2-3번의 반복 후에는 일반적으로 훨씬 더 나아 보일 것입니다. 최상의 결과를 위해 Claude가 자신의 출력물을 볼 수 있는 도구를 제공하세요.



d. 안전한 YOLO 모드

Claude를 감독하는 대신 `claude --dangerously-skip-permissions` 을 사용하여 모든 권한 검사를 우회하고 Claude가 완료될 때까지 중단 없이 작업할 수 있습니다. 이는 린트 오류 수정이나 보일러플레이트 코드 생성과 같은 워크플로우에 잘 작동합니다.

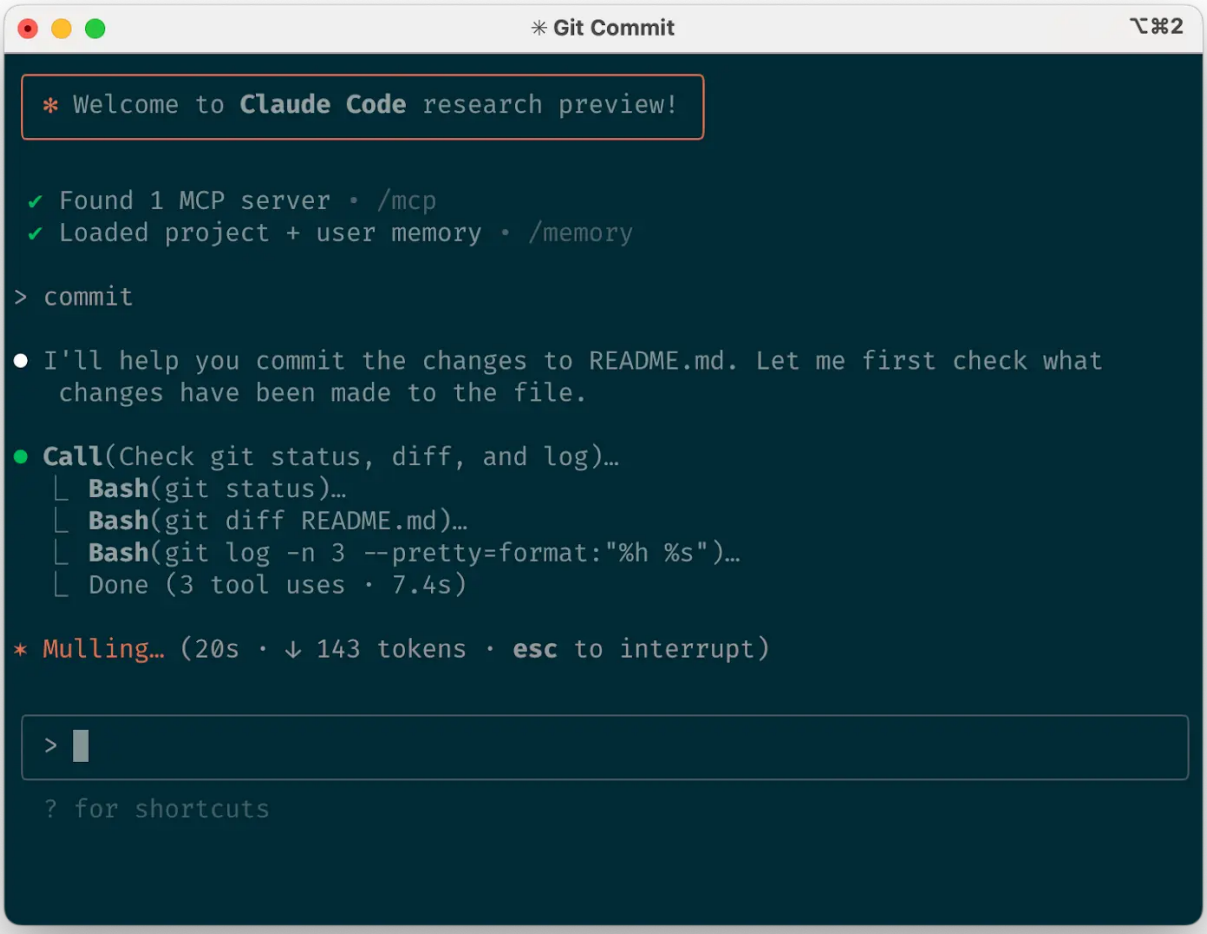
Claude가 임의의 명령을 실행하도록 하는 것은 위험하며 데이터 손실, 시스템 손상, 심지어 데이터 유출(예: 프롬프트 인젝션 공격을 통한)을 초래할 수 있습니다. 이러한 위험을 최소화하려면 `--dangerously-skip-permissions` 인터넷 접근이 없는 컨테이너에서. Docker Dev Containers를 사용하여 이참조 구현을 따를 수 있습니다.

e. 코드베이스 Q&A

새로운 코드베이스에 온보딩할 때, 학습과 탐색을 위해 Claude Code를 사용하세요. 페어 프로그래밍을 할 때 프로젝트의 다른 엔지니어에게 물어볼 법한 질문들을 Claude에게 할 수 있습니다. Claude는 다음과 같은 일반적인 질문들에 답하기 위해 코드베이스를 능동적으로 검색할 수 있습니다:

- 로깅은 어떻게 작동하나요?
- 새로운 API 엔드포인트는 어떻게 만드나요?
- 이것은 무엇을 `async move { ... }` 134번째 줄에서 수행하는 `foo.rs` ?
- 어떤 예외 상황들을 `CustomerOnboardingFlowImpl` 처리하나요?
- 왜 333번째 줄에서 `foo()` 대신에 `bar()` 를 호출하고 있나요?
- Java에서 `baz.py` 의 334번째 줄에 해당하는 것은 무엇인가요?

Anthropic에서는 이런 방식으로 Claude Code를 사용하는 것이 핵심 온보딩 워크플로우가 되었으며, 적응 시간을 크게 단축하고 다른 엔지니어들의 부담을 줄였습니다. 특별한 프롬프팅이 필요하지 않습니다! 단순히 질문하면 Claude가 코드를 탐색하여 답을 찾아줍니다.



f. Claude를 사용하여 git과 상호작용하기

Claude는 많은 git 작업을 효과적으로 처리할 수 있습니다. 많은 Anthropic 엔지니어들이 90% 이상의 *git* 상호작용에 Claude를 사용합니다:

- **검색*git* history** "v1.2.3에는 어떤 변경사항이 포함되었나요?", "이 특정 기능의 담당자는 누구인가요?", 또는 "이 API가 왜 이런 방식으로 설계되었나요?"와 같은 질문에 답하기 위해서입니다. 이런 질문들에 답하기 위해 Claude에게 git history를 살펴보도록 명시적으로 요청하는 것이 도움이 됩니다.
- **커밋 메시지 작성**. Claude는 자동으로 변경 사항과 최근 기록을 살펴보고 모든 관련 컨텍스트를 고려하여 메시지를 작성합니다
- **복잡한 git 작업 처리**파일 되돌리기, 리베이스 충돌 해결, 패치 비교 및 접목과 같은 작업

g. Claude를 사용하여 GitHub와 상호작용

Claude Code는 많은 GitHub 상호작용을 관리할 수 있습니다:

- **풀 리퀘스트 생성**: Claude는 "pr"이라는 줄임말을 이해하고 diff와 주변 컨텍스트를 기반으로 적절한 커밋 메시지를 생성합니다.
- **간단한 코드 리뷰 댓글에 대한 원샷 해결책 구현**: PR의 댓글을 수정하라고 말하기만 하면 됩니다(선택적으로 더 구체적인 지침을 제공할 수 있음). 완료되면 PR 브랜치로 다시 푸시합니다.
- **실패한 빌드 수정**또는 린터 경고
- **열린 이슈들을 분류하고 우선순위 매기기**Claude가 열린 GitHub 이슈들을 반복 처리하도록 요청하여

기억해야 할 필요성을 제거합니다 `gh` 일상적인 작업을 자동화하면서 명령줄 구문을 사용합니다.

h. Claude를 사용하여 Jupyter 노트북 작업하기

Anthropic의 연구원과 데이터 사이언티스트들은 Claude Code를 사용하여 Jupyter 노트북을 읽고 작성합니다. Claude는 이미지를 포함한 출력을 해석할 수 있어 데이터를 탐색하고 상호작용하는 빠른 방법을 제공합니다. 필수 프롬프트나 워크플로우는 없지만, 저희가 권장하는 워크플로우는 VS Code에서 Claude Code와 `.ipynb` 파일을 나란히 열어두는 것입니다.

동료들에게 보여주기 전에 Claude에게 Jupyter 노트북을 정리하거나 미적으로 개선해달라고 요청할 수도 있습니다. 노트북이나 데이터 시각화를 "미적으로 보기 좋게" 만들어달라고 구체적으로 말하면, 인간의 시청 경험을 최적화하고 있다는 것을 상기시키는 데 도움이 됩니다.

4. 워크플로우 최적화

아래 제안사항들은 모든 워크플로우에 적용됩니다:

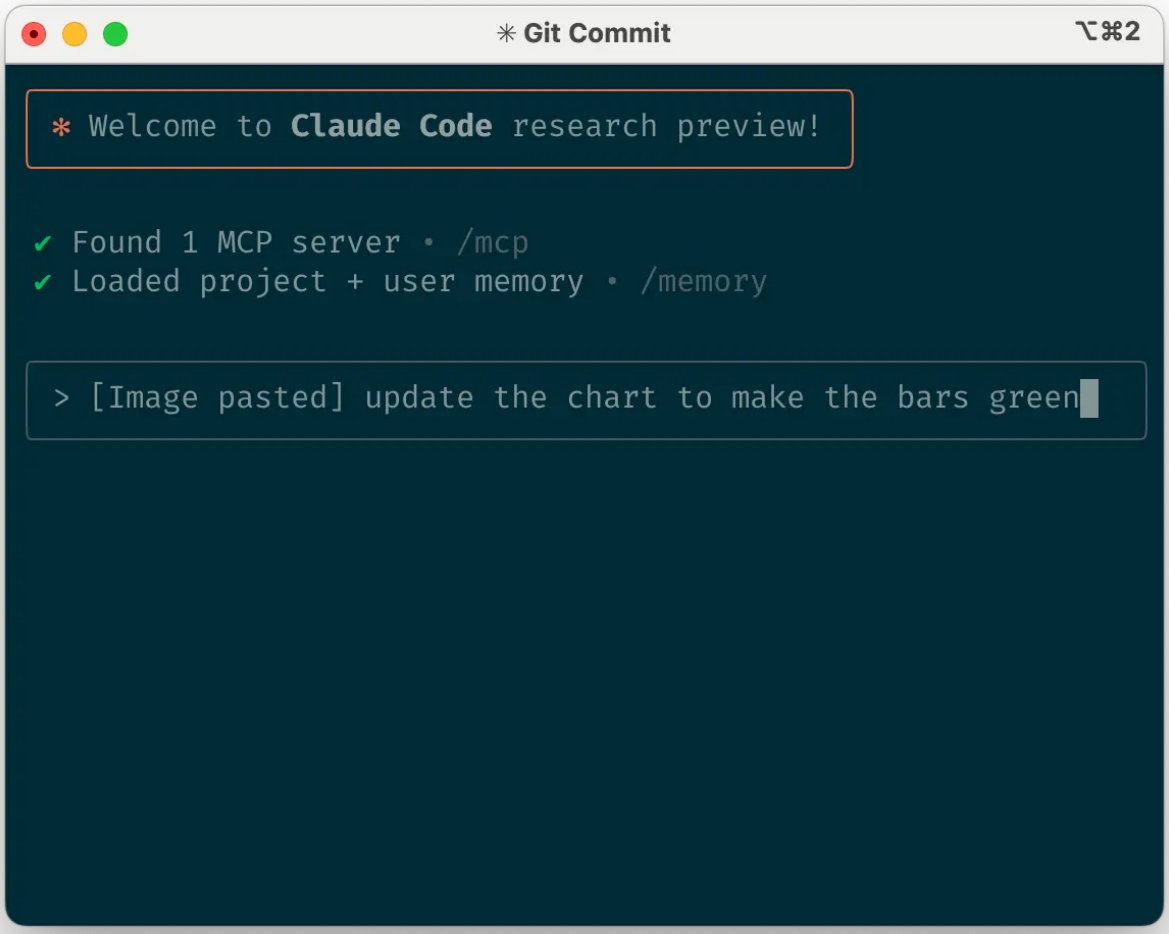
a. 지시사항을 구체적으로 작성하세요

Claude Code의 성공률은 더 구체적인 지시사항을 제공할 때, 특히 첫 번째 시도에서 크게 향상됩니다. 처음부터 명확한 방향을 제시하면 나중에 수정할 필요성을 줄일 수 있습니다.

예를 들어:

나쁨	좋음
foo.py에 대한 테스트 추가	foo.py에 대한 새로운 테스트 케이스를 작성하되, 사용자가 로그아웃된 엣지 케이스를 다루세요. 목(mock)은 사용하지 마세요
ExecutionFactory가 왜 이렇게 이상한 API를 가지고 있나요?	ExecutionFactory의 git 히스토리를 살펴보고 해당 API가 어떻게 만들어졌는지 요약해주세요
캘린더 위젯 추가	홈페이지에서 기존 위젯들이 어떻게 구현되어 있는지 살펴보고 패턴을 이해하세요. 특히 코드와 인터페이스가 어떻게 분리되어 있는지 확인하세요. HotDogWidget.php가 시작하기 좋은 예시입니다. 그런 다음 패턴을 따라 사용자가 월을 선택하고 앞뒤로 페이지를 넘겨 연도를 선택할 수 있는 새로운 캘린더 위젯을 구현하세요. 코드베이스의 나머지 부분에서 이미 사용된 라이브러리 외에는 다른 라이브러리 없이 처음부터 구축하세요.

Claude는 의도를 추론할 수 있지만, 마음을 읽을 수는 없습니다. 구체적으로 명시할수록 기대에 더 잘 부합하는 결과를 얻을 수 있습니다.

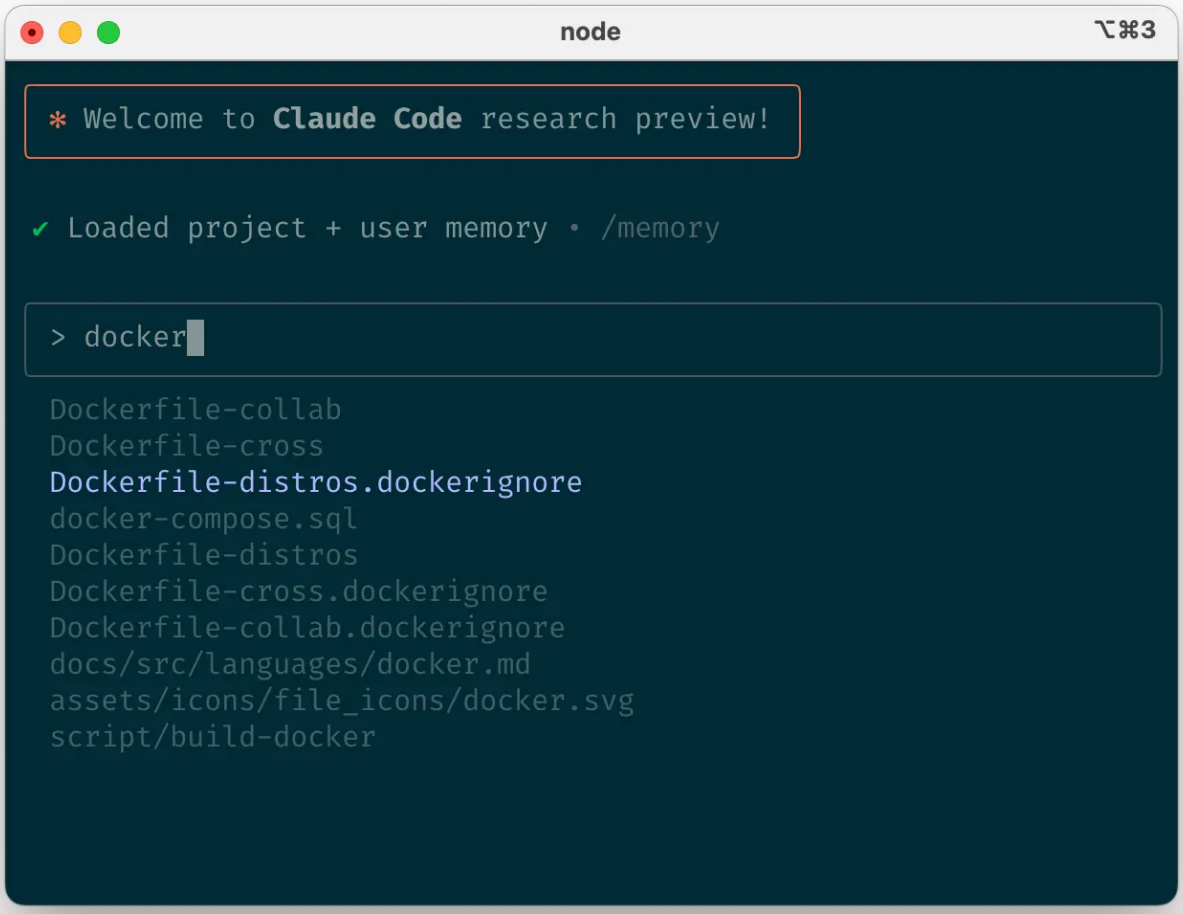


b. Claude에게 이미지 제공하기

Claude는 여러 방법을 통해 이미지와 다이어그램을 잘 처리합니다:

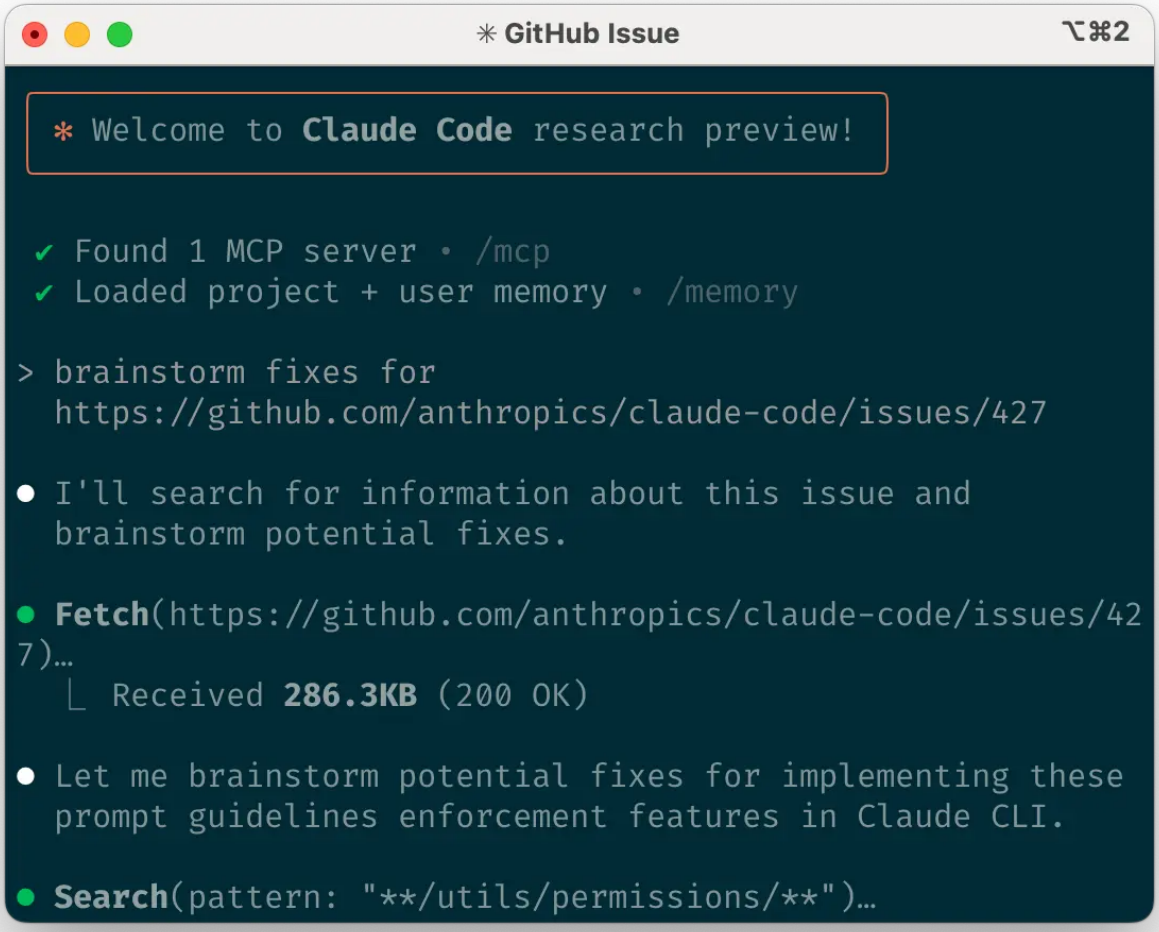
- **스크린샷 붙여넣기** (팁: macOS에서 *cmd+ctrl+shift+4* 를 눌러 클립보드에 스크린샷을 찍고 *ctrl+v* 를 눌러 붙여넣기하세요. 이것은 맥에서 일반적으로 사용하는 cmd+v와는 다르며 원격으로는 작동하지 않습니다.)
- **드래그 앤 드롭**이미지를 프롬프트 입력에 직접
- **이미지의 파일 경로 제공**이미지용

이는 UI 개발의 참조점으로 디자인 목업을 사용하거나 분석 및 디버깅을 위한 시각적 차트를 다룰 때 특히 유용합니다. 컨텍스트에 시각적 요소를 추가하지 않더라도, 결과가 시각적으로 매력적이어야 하는 것이 얼마나 중요한지 Claude에게 명확히 전달하는 것이 도움이 될 수 있습니다.



c. Claude가 살펴보거나 작업하기를 원하는 파일을 언급하세요

탭 완성 기능을 사용하여 저장소 내 어디에 있는 파일이나 폴더든 빠르게 참조할 수 있으며, 이를 통해 Claude가 올바른 리소스를 찾거나 업데이트할 수 있도록 도울 수 있습니다.



d. Claude에게 URL 제공하기

Claude가 가져와서 읽을 수 있도록 프롬프트와 함께 특정 URL을 붙여넣으세요. 동일한 도메인(예: docs.foo.com)에 대한 권한 프롬프트를 피하려면 `/permissions` 을 사용하여 허용 목록에 도메인을 추가하세요.

e. 조기에 그리고 자주 방향을 수정하세요

자동 수락 모드(shift+tab으로 전환)를 사용하면 Claude가 자율적으로 작업할 수 있지만, 일반적으로 적극적인 협력자가 되어 Claude의 접근 방식을 안내할 때 더 나은 결과를 얻을 수 있습니다. 처음에 Claude에게 작업을 철저히 설명하면 최상의 결과를 얻을 수 있지만, 언제든지 Claude의 방향을 수정할 수도 있습니다.

다음 네 가지 도구가 방향 수정에 도움이 됩니다:

- **Claude에게 계획을 세우도록 요청하세요**코딩하기 전에. 계획이 좋아 보인다고 확인하기 전까지는 코딩하지 말라고 명시적으로 지시하세요.
- **Escape를 눌러 중단하기**Claude를 어떤 단계에서든(사고, 도구 호출, 파일 편집) 중단할 수 있으며, 컨텍스트를 보존하여 지시사항을 재지정하거나 확장할 수 있습니다.

- **Escape를 두 번 눌러 기록으로 돌아가기**, 이전 프롬프트를 편집하고 다른 방향을 탐색할 수 있습니다. 원하는 결과를 얻을 때까지 프롬프트를 편집하고 반복할 수 있습니다.
- **Claude에게 변경사항 취소 요청하기**, 종종 다른 접근 방식을 취하기 위해 옵션 #2와 함께 사용됩니다.

Claude Code가 때때로 첫 번째 시도에서 문제를 완벽하게 해결하기도 하지만, 이러한 수정 도구들을 사용하면 일반적으로 더 나은 솔루션을 더 빠르게 얻을 수 있습니다.

f. 컨텍스트 집중을 위해 사용하기 `긴 세션 동안 Claude의 컨텍스트 창은 관련 없는 대화, 파일 내용, 명령어들로 가득 찰 수 있습니다. 이는 성능을 저하시키고 때로는 Claude의 주의를 분산시킬 수 있습니다. 작업 간에` /clear

/clear `긴 세션 동안 Claude의 컨텍스트 창은 관련 없는 대화, 파일 내용, 명령어들로 가득 찰 수 있습니다. 이는 성능을 저하시키고 때로는 Claude의 주의를 분산시킬 수 있습니다. 작업 간에` 명령어를 자주 사용하여 컨텍스트 창을 재설정하세요.

g. 복잡한 워크플로우에는 체크리스트와 스크래치패드 사용

코드 마이그레이션, 수많은 린트 오류 수정, 복잡한 빌드 스크립트 실행과 같이 여러 단계가 필요하거나 철저한 해결책이 필요한 대규모 작업의 경우, Claude가 Markdown 파일(또는 GitHub 이슈!)을 체크리스트와 작업 스크래치패드로 사용하도록 하여 성능을 향상시키세요:

예를 들어, 많은 수의 린트 문제를 수정하려면 다음과 같이 할 수 있습니다:

1. **Claude에게 린트 명령을 실행하도록 지시**모든 결과 오류를 (파일명과 줄 번호와 함께) 마크다운 체크리스트로 작성하세요
2. **Claude에게 각 문제를 하나씩 해결하도록 지시하세요**, 체크 표시를 하고 다음으로 넘어가기 전에 수정하고 검증하세요

h. Claude에 데이터 전달하기

Claude에 데이터를 제공하는 여러 방법이 있습니다:

- **복사 및 붙여넣기**프롬프트에 직접 입력 (가장 일반적인 방법)
- **Claude Code로 파이프**(예: `cat foo.txt | claude`), 특히 로그, CSV, 대용량 데이터에 유용함
- **Claude에게 데이터를 가져오도록 지시하세요**bash 명령어, MCP 도구 또는 사용자 정의 슬래시 명령어를 통해
- **Claude에게 파일 읽기 요청**또는 URL 가져오기 (이미지도 가능)

대부분의 세션은 이러한 접근 방식들의 조합을 포함합니다. 예를 들어, 로그 파일을 파이프로 입력한 다음 Claude에게 도구를 사용하여 로그를 디버깅하기 위한 추가 컨텍스트를 가져오도록 지시할 수 있습니다.

5. 헤드리스 모드를 사용하여 인프라를 자동화하세요

Claude Code는 다음을 포함합니다**헤드리스 모드**CI, pre-commit 훅, 빌드 스크립트, 자동화와 같은 비대화형 컨텍스트를 위한 기능입니다. 프롬프트와 함께 `-p` 플래그를 사용하여 헤드리스 모드를 활성화하고, `--output-format stream-json` 스트리밍 JSON 출력용.

헤드리스 모드는 세션 간에 지속되지 않습니다. 각 세션마다 이를 활성화해야 합니다.

a. 이슈 분류에 Claude 사용하기

헤드리스 모드는 저장소에 새로운 이슈가 생성되는 것과 같은 GitHub 이벤트에 의해 트리거되는 자동화를 지원할 수 있습니다. 예를 들어, 공개 **Claude Code 저장소**는 Claude를 사용하여 새로 들어오는 이슈를 검사하고 적절한 라벨을 할당합니다.

b. Claude를 린터로 사용하기

Claude Code는 제공할 수 있습니다**주관적인 코드 리뷰를**기존 린팅 도구가 감지하는 것 이상으로, 오타, 오래된 주석, 오해를 불러일으키는 함수나 변수명 등의 문제를 식별합니다.

6. 멀티-Claude 워크플로우로 업레벨하기

독립적인 사용을 넘어서, 가장 강력한 애플리케이션 중 일부는 여러 Claude 인스턴스를 병렬로 실행하는 것을 포함합니다:

a. 한 Claude가 코드를 작성하게 하고, 다른 Claude를 사용해 검증하기

간단하지만 효과적인 접근법은 한 Claude가 코드를 작성하는 동안 다른 Claude가 이를 검토하거나 테스트하게 하는 것입니다. 여러 엔지니어와 작업하는 것과 유사하게, 때로는 별도의 컨텍스트를 갖는 것이 유익합니다:

1. Claude를 사용해 코드 작성
2. 실행 `긴 세션 동안 Claude의 컨텍스트 창은 관련 없는 대화, 파일 내용, 명령어들로 가득 찰 수 있습니다. 이는 성능을 저하시키고 때로는 Claude의 주의를 분산시킬 수 있습니다. 작업 간에` 또는 다른 터미널에서 두 번째 Claude 시작
3. 두 번째 Claude가 첫 번째 Claude의 작업을 검토하도록 하세요
4. 다른 Claude를 시작하여 (또는 `긴 세션 동안 Claude의 컨텍스트 창은 관련 없는 대화, 파일 내용, 명령어들로 가득 찰 수 있습니다. 이는 성능을 저하시키고 때로는 Claude의 주의를 분산시킬 수 있습니다. 작업 간에` 다시) 코드와 검토 피드백을 모두 읽도록 하세요
5. 피드백을 바탕으로 이 Claude가 코드를 편집하도록 하세요

테스트에서도 비슷한 방법을 사용할 수 있습니다: 한 Claude가 테스트를 작성하고, 다른 Claude가 테스트를 통과하는 코드를 작성하도록 하는 것입니다. 각각에게 별도의 작업용 스크래치패드를 제공하고 어느 것에 쓰고 어느 것에서 읽을지 알려주면 Claude 인스턴스들이 서로 소통하도록 할 수도 있습니다.

이러한 분리는 단일 Claude가 모든 것을 처리하는 것보다 종종 더 나은 결과를 가져다줍니다.

b. 저장소의 여러 체크아웃을 보유하세요

클로드가 각 단계를 완료할 때까지 기다리는 대신, Anthropic의 많은 엔지니어들이 하는 방법은:

1. **3-4개의 git 체크아웃을 생성하는 것입니다**별도의 폴더에
2. **각 폴더를 열어주세요**별도의 터미널 탭에서
3. **각 폴더에서 Claude를 시작하세요**다른 작업들과 함께
4. **순환하세요**진행 상황을 확인하고 권한 요청을 승인/거부하기 위해

c. git worktree 사용하기

이 접근 방식은 여러 독립적인 작업에 유용하며, 여러 체크아웃에 대한 더 가벼운 대안을 제공합니다. Git worktree를 사용하면 동일한 저장소에서 여러 브랜치를 별도의 디렉토리로 체크아웃할 수 있습니다. 각 worktree는 격리된 파일을 가진 자체 작업 디렉토리를 가지면서, 동일한 Git 히스토리와 reflog를 공유합니다.

git worktree를 사용하면 프로젝트의 서로 다른 부분에서 여러 Claude 세션을 동시에 실행할 수 있으며, 각각은 독립적인 작업에 집중할 수 있습니다. 예를 들어, 한 Claude는 인증 시스템을 리팩토링하고 다른 Claude는 완전히 관련 없는 데이터 시각화 컴포넌트를 구축할 수 있습니다. 작업이 겹치지 않기 때문에 각 Claude는 다른 Claude의 변경 사항을 기다리거나 병합 충돌을 처리할 필요 없이 최대 속도로 작업할 수 있습니다:

1. **워크트리 생성**: `git worktree add ../project-feature-a feature-a`
2. **각 워크트리에서 Claude 실행**: `cd ../project-feature-a && claude`
3. **추가 워크트리 생성**필요에 따라 (새 터미널 탭에서 1-2단계 반복)

몇 가지 팁:

- 일관된 명명 규칙을 사용하세요
- 워크트리당 하나의 터미널 탭을 유지하세요
- Mac에서 iTerm2를 사용하고 있다면,**알림을 설정하세요**클로드가 관심을 필요로 할 때를 위해
- 다른 워크트리에 대해 별도의 IDE 창 사용
- 완료 후 정리: `git worktree remove ../project-feature-a`

d. 사용자 정의 하네스와 함께 헤드리스 모드 사용

`claude -p` (헤드리스 모드)는 내장 도구와 시스템 프롬프트를 활용하면서 Claude Code를 더 큰 워크플로우에 프로그래밍 방식으로 통합합니다. 헤드리스 모드를 사용하는 두 가지 주요 패턴이 있습니다:

1. **팬아웃** 대규모 마이그레이션이나 분석을 처리합니다 (예: 수백 개의 로그에서 감정 분석 또는 수천 개의 CSV 분석):
2. Claude에게 작업 목록을 생성하는 스크립트를 작성하도록 합니다. 예를 들어, 프레임워크 A에서 프레임워크 B로 마

이그레이션해야 하는 2천 개 파일의 목록을 생성합니다.

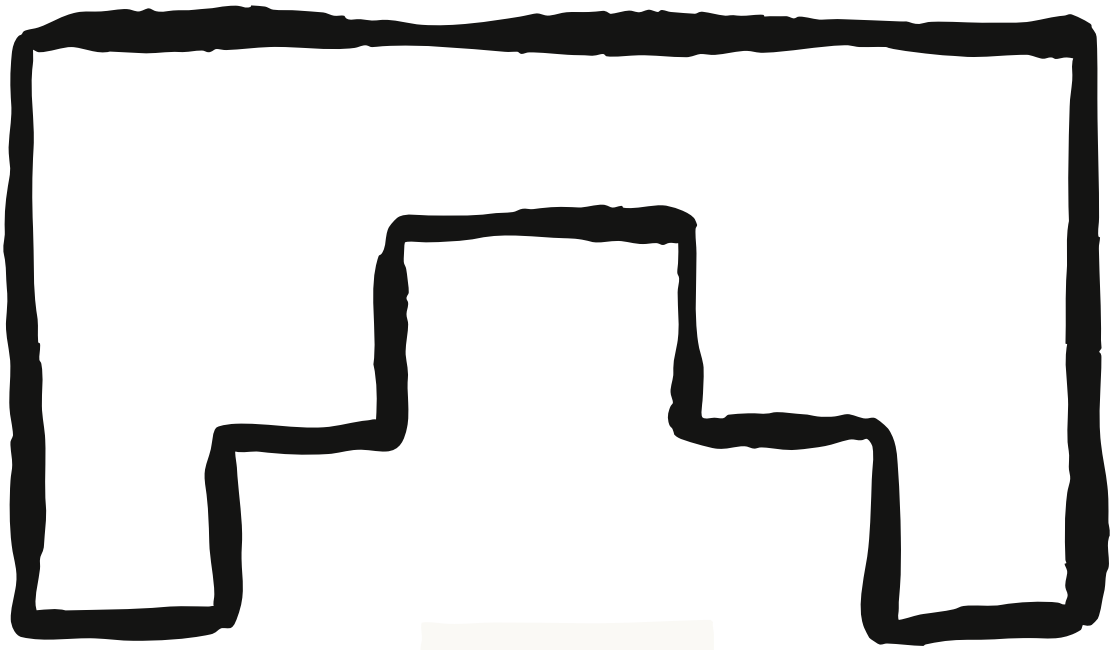
3. 작업을 반복하면서 각각에 대해 Claude를 프로그래밍 방식으로 호출하고, 작업과 사용할 수 있는 도구 세트를 제공합니다. 예를 들어: `claude -p "foo.py를 React에서 Vue로 마이그레이션하세요. 완료되면 성공한 경우 OK 문자열을, 작업이 실패한 경우 FAIL을 반드시 반환해야 합니다." --allowedTools Edit Bash(git commit:*)`
4. 스크립트를 여러 번 실행하고 프롬프트를 개선하여 원하는 결과를 얻으세요.
5. **파이프라이닝** 기존 데이터/처리 파이프라인에 Claude를 통합합니다:
6. 호출 `claude -p "" --json | your_command`, 여기서 `your_command`는 처리 파이프라인의 다음 단계입니다
7. 이게 전부입니다! JSON 출력(선택사항)은 더 쉬운 자동화 처리를 위한 구조를 제공하는 데 도움이 될 수 있습니다.

이 두 사용 사례 모두에서 Claude 호출을 디버깅하기 위해 `--verbose` 플래그를 사용하는 것이 도움이 될 수 있습니다. 일반적으로 더 깔끔한 출력을 위해 프로덕션 환경에서는 verbose 모드를 끄는 것을 권장합니다.

Claude Code 작업에 대한 팁과 모범 사례가 있으신가요? @AnthropicAI를 태그해 주시면 여러분이 구축하고 있는 것을 확인할 수 있습니다!

감사의 말

Boris Cherny가 작성했습니다. 이 작업은 광범위한 Claude Code 사용자 커뮤니티의 모범 사례를 바탕으로 하며, 이들의 창의적인 접근법과 워크플로우는 계속해서 우리에게 영감을 주고 있습니다. 또한 Daisy Hollman, Ashwin Bhat, Cat Wu, Sid Bidasaria, Cal Rueb, Nodir Turakulov, Barry Zhang, Drew Hodun 그리고 Claude Code에 대한 귀중한 통찰력과 실무 경험으로 이러한 권장사항을 형성하는 데 도움을 준 많은 다른 Anthropic 엔지니어들에게 특별한 감사를 표합니다.



더 자세히 알아보고 싶으신가요?

Anthropic Academy의 강좌를 통해 API 개발, 모델 컨텍스트 프로토콜, Claude Code를 마스터하세요. 완료 시 수료증을 받을 수 있습니다. **코스 탐색**

Claude Code: Best practices for agentic coding

Published Apr 18, 2025

Claude Code is a command line tool for agentic coding. This post covers tips and tricks that have proven effective for using Claude Code across various codebases, languages, and environments.

We recently released Claude Code, a command line tool for agentic coding. Developed as a research project, Claude Code gives Anthropic engineers and researchers a more native way to integrate Claude into their coding workflows.

Claude Code is intentionally low-level and unopinionated, providing close to raw model access without forcing specific workflows. This design philosophy creates a flexible, customizable, scriptable, and safe power tool. While powerful, this flexibility presents a learning curve for engineers new to agentic coding tools—at least until they develop their own best practices.

This post outlines general patterns that have proven effective, both for Anthropic's internal teams and for external engineers using Claude Code across various codebases, languages, and environments. Nothing in this list is set in stone nor universally applicable; consider these suggestions as starting points. We encourage you to experiment and find what works best for you!

Looking for more detailed information? Our comprehensive documentation at claude.ai/code covers all the features mentioned in this post and provides additional examples, implementation details, and advanced techniques.

1. Customize your setup

Claude Code is an agentic coding assistant that automatically pulls context into prompts. This context gathering consumes time and tokens, but you can optimize it through environment tuning.

a. Create `CLAUDE.md` files

`CLAUDE.md` is a special file that Claude automatically pulls into context when starting a conversation. This makes it an ideal place for documenting:

- Common bash commands
- Core files and utility functions
- Code style guidelines
- Testing instructions
- Repository etiquette (e.g., branch naming, merge vs. rebase, etc.)
- Developer environment setup (e.g., pyenv use, which compilers work)
- Any unexpected behaviors or warnings particular to the project

- Other information you want Claude to remember

There's no required format for `CLAUDE.md` files. We recommend keeping them concise and human-readable. For example:

```
# Bash commands
- npm run build: Build the project
- npm run typecheck: Run the typechecker

# Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

# Workflow
- Be sure to typecheck when you're done making a series of code changes
- Prefer running single tests, and not the whole test suite, for performance
```

Copy

You can place `CLAUDE.md` files in several locations:

- **The root of your repo**, or wherever you run `claude` from (the most common usage). Name it `CLAUDE.md` and check it into git so that you can share it across sessions and with your team (recommended), or name it `CLAUDE.local.md` and `.gitignore` it
- **Any parent of the directory** where you run `claude`. This is most useful for monorepos, where you might run `claude` from `root/foo`, and have `CLAUDE.md` files in both `root/CLAUDE.md` and `root/foo/CLAUDE.md`. Both of these will be pulled into context automatically
- **Any child of the directory** where you run `claude`. This is the inverse of the above, and in this case, Claude will pull in `CLAUDE.md` files on demand when you work with files in child directories
- **Your home folder** (`~/.claude/CLAUDE.md`), which applies it to all your *claude* sessions

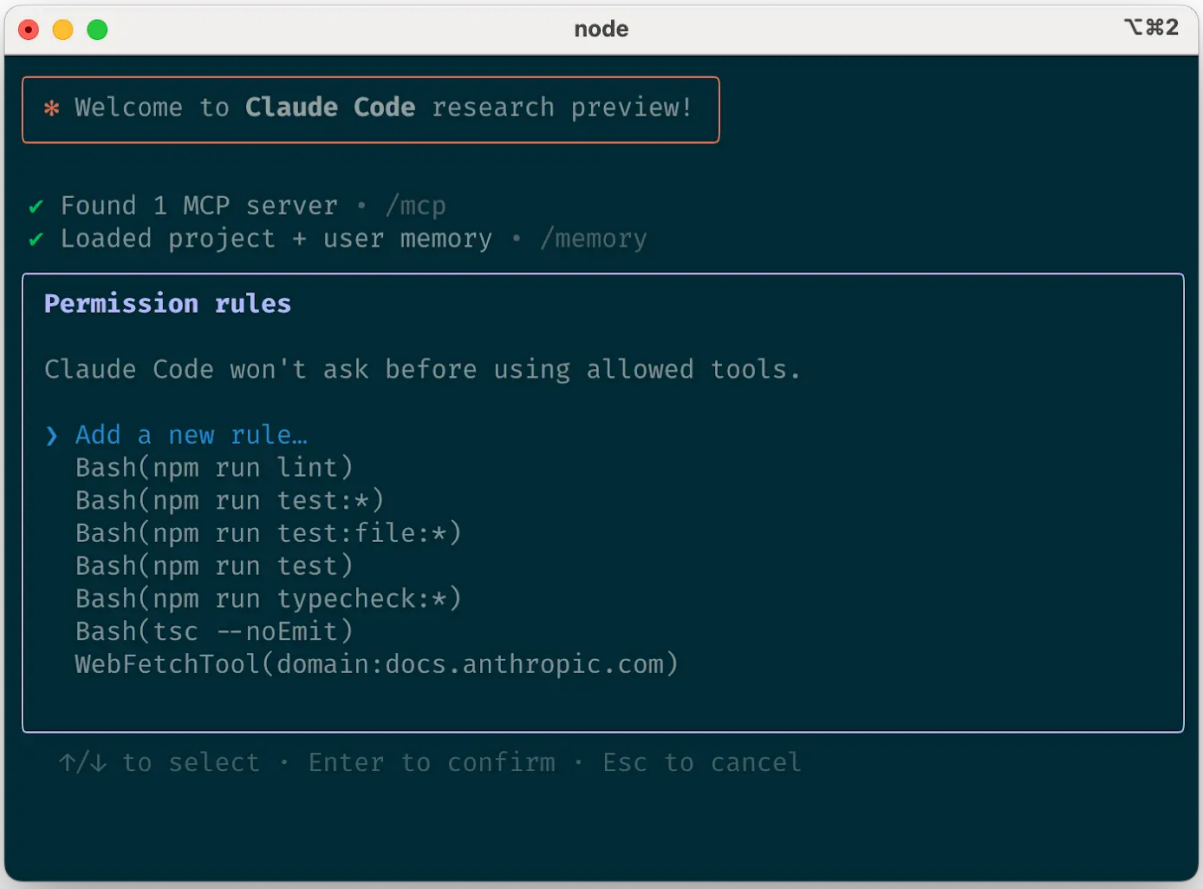
When you run the `/init` command, Claude will automatically generate a `CLAUDE.md` for you.

b. Tune your `CLAUDE.md` files

Your `CLAUDE.md` files become part of Claude's prompts, so they should be refined like any frequently used prompt. A common mistake is adding extensive content without iterating on its effectiveness. Take time to experiment and determine what produces the best instruction following from the model.

You can add content to your `CLAUDE.md` manually or press the `#` key to give Claude an instruction that it will automatically incorporate into the relevant `CLAUDE.md`. Many engineers use `#` frequently to document commands, files, and style guidelines while coding, then include `CLAUDE.md` changes in commits so team members benefit as well.

At Anthropic, we occasionally run `CLAUDE.md` files through the prompt improver and often tune instructions (e.g. adding emphasis with "IMPORTANT" or "YOU MUST") to improve adherence.



c. Curate Claude's list of allowed tools

By default, Claude Code requests permission for any action that might modify your system: file writes, many bash commands, MCP tools, etc. We designed Claude Code with this deliberately conservative approach to prioritize safety. You can customize the allowlist to permit additional tools that you know are safe, or to allow potentially unsafe tools that are easy to undo (e.g., file editing, `git commit`).

There are four ways to manage allowed tools:

- **Select "Always allow"** when prompted during a session.
- **Use the `/permissions` command** after starting Claude Code to add or remove tools from the allowlist. For example, you can add `Edit` to always allow file edits, `Bash(git commit:*)` to allow git commits, or `mcp__puppeteer__puppeteer_navigate` to allow navigating with the Puppeteer MCP server.
- **Manually edit** your `.claude/settings.json` or `~/.claude.json` (we recommend checking the former into source control to share with your team).

- Use the `--allowedTools` CLI flag for session-specific permissions.

d. If using GitHub, install the gh CLI

Claude knows how to use the `gh` CLI to interact with GitHub for creating issues, opening pull requests, reading comments, and more. Without `gh` installed, Claude can still use the GitHub API or MCP server (if you have it installed).

2. Give Claude more tools

Claude has access to your shell environment, where you can build up sets of convenience scripts and functions for it just like you would for yourself. It can also leverage more complex tools through MCP and REST APIs.

a. Use Claude with bash tools

Claude Code inherits your bash environment, giving it access to all your tools. While Claude knows common utilities like unix tools and `gh`, it won't know about your custom bash tools without instructions:

1. Tell Claude the tool name with usage examples
2. Tell Claude to run `--help` to see tool documentation
3. Document frequently used tools in `CLAUDE.md`

b. Use Claude with MCP

Claude Code functions as both an MCP server and client. As a client, it can connect to any number of MCP servers to access their tools in three ways:

- **In project config** (available when running Claude Code in that directory)
- **In global config** (available in all projects)
- **In a checked-in `.mcp.json` file** (available to anyone working in your codebase). For example, you can add Puppeteer and Sentry servers to your `.mcp.json`, so that every engineer working on your repo can use these out of the box.

When working with MCP, it can also be helpful to launch Claude with the `--mcp-debug` flag to help identify configuration issues.

c. Use custom slash commands

For repeated workflows—debugging loops, log analysis, etc.—store prompt templates in Markdown files within the `.claude/commands` folder. These become available through the slash commands menu when you type `/`. You can check these commands into git to make them available for the rest of your team.

Custom slash commands can include the special keyword `$ARGUMENTS` to pass parameters from command invocation.

For example, here's a slash command that you could use to automatically pull and fix a Github issue:

```
Please analyze and fix the GitHub issue: $ARGUMENTS.

Follow these steps:

1. Use `gh issue view` to get the issue details
2. Understand the problem described in the issue
3. Search the codebase for relevant files
4. Implement the necessary changes to fix the issue
5. Write and run tests to verify the fix
6. Ensure code passes linting and type checking
7. Create a descriptive commit message
8. Push and create a PR

Remember to use the GitHub CLI (`gh`) for all GitHub-related tasks.
```

Copy

Putting the above content into `.claude/commands/fix-github-issue.md` makes it available as the `/project:fix-github-issue` command in Claude Code. You could then for example use `/project:fix-github-issue 1234` to have Claude fix issue #1234. Similarly, you can add your own personal commands to the `~/.claude/commands` folder for commands you want available in all of your sessions.

3. Try common workflows

Claude Code doesn't impose a specific workflow, giving you the flexibility to use it how you want. Within the space this flexibility affords, several successful patterns for effectively using Claude Code have emerged across our community of users:

a. Explore, plan, code, commit

This versatile workflow suits many problems:

1. **Ask Claude to read relevant files, images, or URLs**, providing either general pointers ("read the file

that handles logging") or specific filenames ("read logging.py"), but explicitly tell it not to write any code just yet.

    1. This is the part of the workflow where you should consider strong use of subagents, especially for complex problems. Telling Claude to use subagents to verify details or investigate particular questions it might have, especially early on in a conversation or task, tends to preserve context availability without much downside in terms of lost efficiency.

2. **Ask Claude to make a plan for how to approach a specific problem**. We recommend using the word "think" to trigger extended thinking mode, which gives Claude additional computation time to evaluate alternatives more thoroughly. These specific phrases are mapped directly to increasing levels of thinking budget in the system: "think" < "think hard" < "think harder" < "ultrathink." Each level allocates progressively more thinking budget for Claude to use.

    1. If the results of this step seem reasonable, you can have Claude create a document or a GitHub issue with its plan so that you can reset to this spot if the implementation (step 3) isn't what you want.

3. **Ask Claude to implement its solution in code**. This is also a good place to ask it to explicitly verify the reasonableness of its solution as it implements pieces of the solution.
4. **Ask Claude to commit the result and create a pull request**. If relevant, this is also a good time to have Claude update any READMEs or changelogs with an explanation of what it just did.

Steps #1-#2 are crucial—without them, Claude tends to jump straight to coding a solution. While sometimes that's what you want, asking Claude to research and plan first significantly improves performance for problems requiring deeper thinking upfront.

b. Write tests, commit; code, iterate, commit

This is an Anthropic-favorite workflow for changes that are easily verifiable with unit, integration, or end-to-end tests. Test-driven development (TDD) becomes even more powerful with agentic coding:

1. **Ask Claude to write tests based on expected input/output pairs**. Be explicit about the fact that you're doing test-driven development so that it avoids creating mock implementations, even for functionality that doesn't exist yet in the codebase.
2. **Tell Claude to run the tests and confirm they fail**. Explicitly telling it not to write any implementation code at this stage is often helpful.
3. **Ask Claude to commit the tests** when you're satisfied with them.
4. **Ask Claude to write code that passes the tests**, instructing it not to modify the tests. Tell Claude to keep going until all tests pass. It will usually take a few iterations for Claude to write code, run the tests, adjust the code, and run the tests again.

    1. At this stage, it can help to ask it to verify with independent subagents that the

implementation isn't overfitting to the tests

5. **Ask Claude to commit the code** once you're satisfied with the changes.

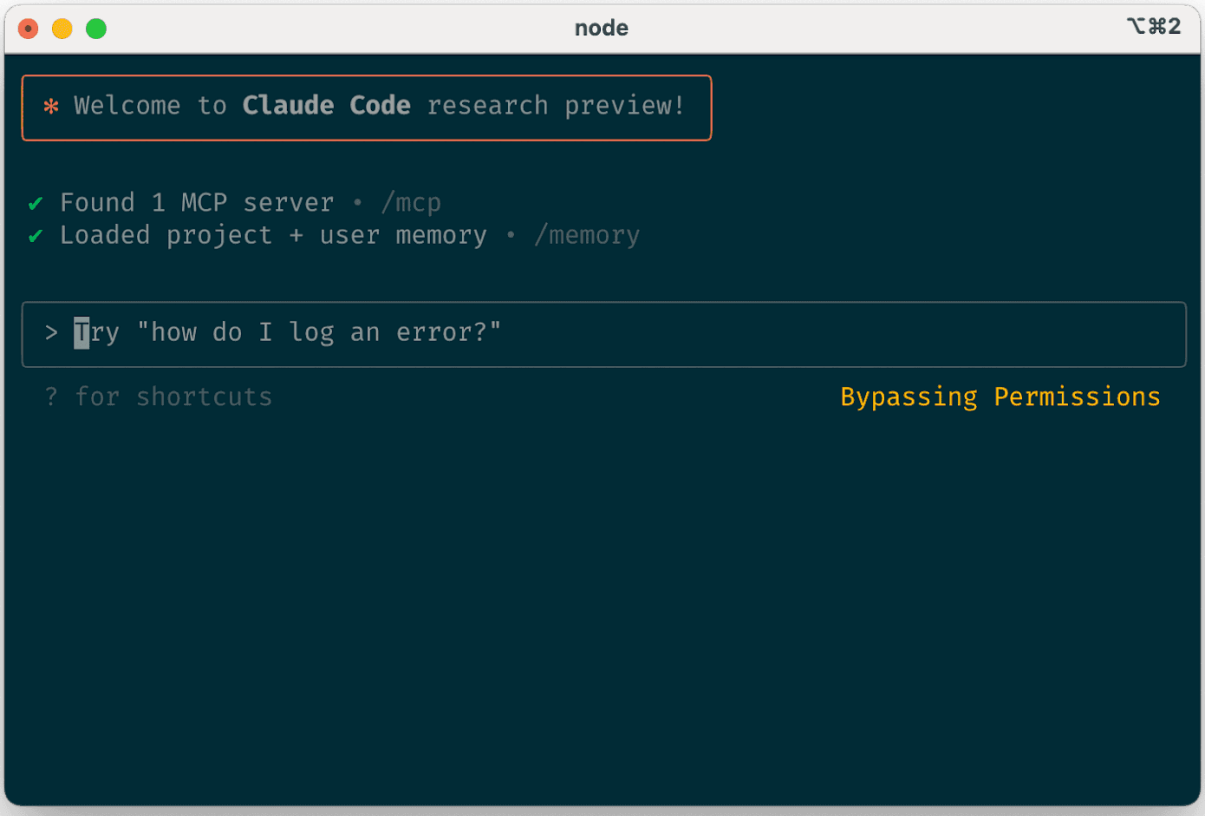
Claude performs best when it has a clear target to iterate against—a visual mock, a test case, or another kind of output. By providing expected outputs like tests, Claude can make changes, evaluate results, and incrementally improve until it succeeds.

c. Write code, screenshot result, iterate

Similar to the testing workflow, you can provide Claude with visual targets:

1. **Give Claude a way to take browser screenshots** (e.g., with the Puppeteer MCP server, an iOS simulator MCP server, or manually copy / paste screenshots into Claude).
2. **Give Claude a visual mock** by copying / pasting or drag-dropping an image, or giving Claude the image file path.
3. **Ask Claude to implement the design** in code, take screenshots of the result, and iterate until its result matches the mock.
4. **Ask Claude to commit** when you're satisfied.

Like humans, Claude's outputs tend to improve significantly with iteration. While the first version might be good, after 2-3 iterations it will typically look much better. Give Claude the tools to see its outputs for best results.



d. Safe YOLO mode

Instead of supervising Claude, you can use `claude --dangerously-skip-permissions` to bypass all permission checks and let Claude work uninterrupted until completion. This works well for workflows like fixing lint errors or generating boilerplate code.

Letting Claude run arbitrary commands is risky and can result in data loss, system corruption, or even data exfiltration (e.g., via prompt injection attacks). To minimize these risks, use `--dangerously-skip-permissions` in a container without internet access. You can follow this reference implementation using Docker Dev Containers.

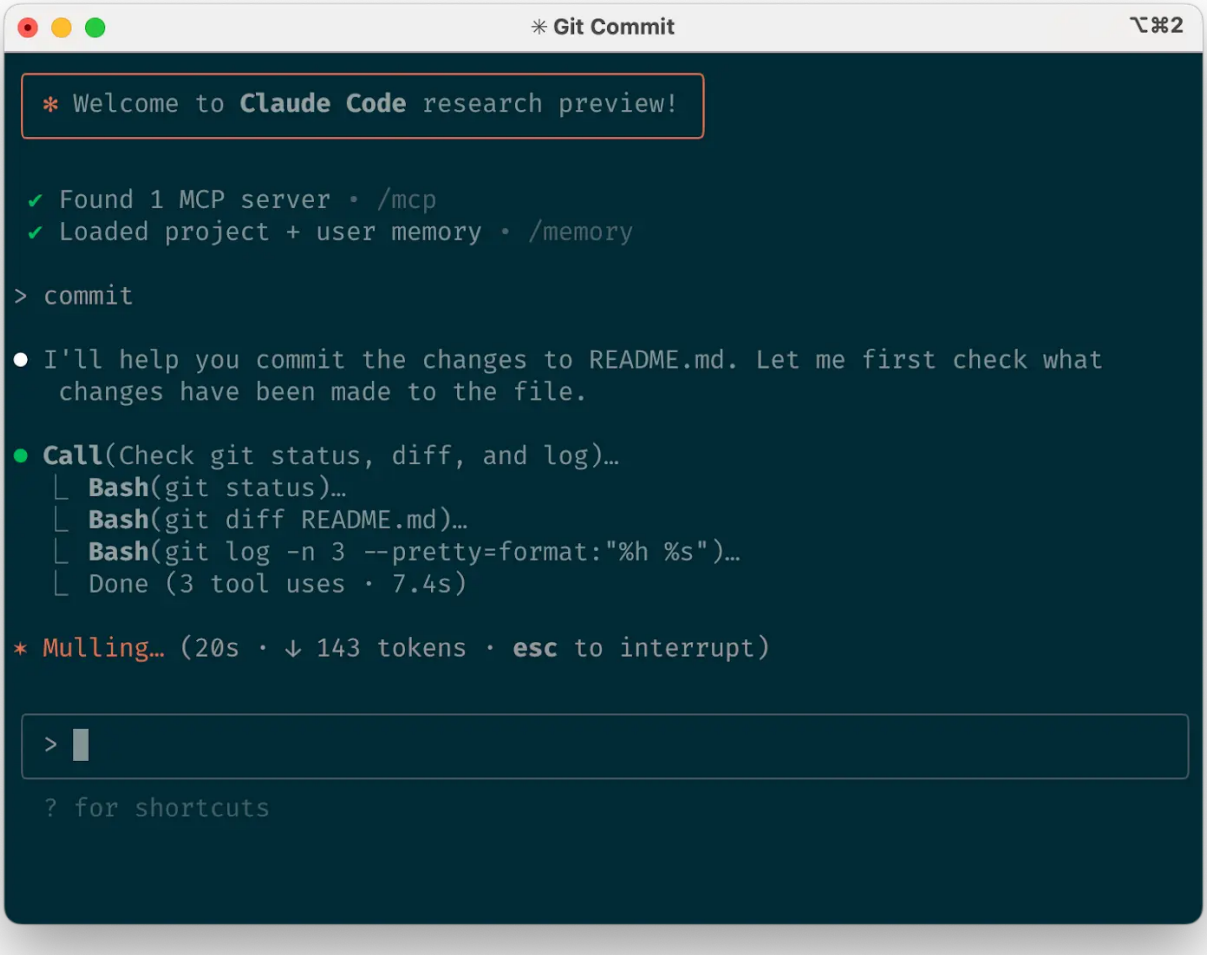
e. Codebase Q&A

When onboarding to a new codebase, use Claude Code for learning and exploration. You can ask Claude the same sorts of questions you would ask another engineer on the project when pair programming. Claude can agentically search the codebase to answer general questions like:

- How does logging work?
- How do I make a new API endpoint?
- What does `async move { ... }` do on line 134 of `foo.rs`?

- What edge cases does `CustomerOnboardingFlowImpl` handle?
- Why are we calling `foo()` instead of `bar()` on line 333?
- What's the equivalent of line 334 of `baz.py` in Java?

At Anthropic, using Claude Code in this way has become our core onboarding workflow, significantly improving ramp-up time and reducing load on other engineers. No special prompting is required! Simply ask questions, and Claude will explore the code to find answers.



f. Use Claude to interact with git

Claude can effectively handle many git operations. Many Anthropic engineers use Claude for 90%+ of our *git* interactions:

- **Searching *git* history** to answer questions like "What changes made it into v1.2.3?", "Who owns this particular feature?", or "Why was this API designed this way?" It helps to explicitly prompt Claude to look through git history to answer queries like these.
- **Writing commit messages**. Claude will look at your changes and recent history automatically to compose a message taking all the relevant context into account

- **Handling complex git operations** like reverting files, resolving rebase conflicts, and comparing and grafting patches

g. Use Claude to interact with GitHub

Claude Code can manage many GitHub interactions:

- **Creating pull requests**: Claude understands the shorthand "pr" and will generate appropriate commit messages based on the diff and surrounding context.
- **Implementing one-shot resolutions** for simple code review comments: just tell it to fix comments on your PR (optionally, give it more specific instructions) and push back to the PR branch when it's done.
- **Fixing failing builds** or linter warnings
- **Categorizing and triaging open issues** by asking Claude to loop over open GitHub issues

This eliminates the need to remember `gh` command line syntax while automating routine tasks.

h. Use Claude to work with Jupyter notebooks

Researchers and data scientists at Anthropic use Claude Code to read and write Jupyter notebooks. Claude can interpret outputs, including images, providing a fast way to explore and interact with data. There are no required prompts or workflows, but a workflow we recommend is to have Claude Code and a `.ipynb` file open side-by-side in VS Code.

You can also ask Claude to clean up or make aesthetic improvements to your Jupyter notebook before you show it to colleagues. Specifically telling it to make the notebook or its data visualizations “aesthetically pleasing” tends to help remind it that it’s optimizing for a human viewing experience.

4. Optimize your workflow

The suggestions below apply across all workflows:

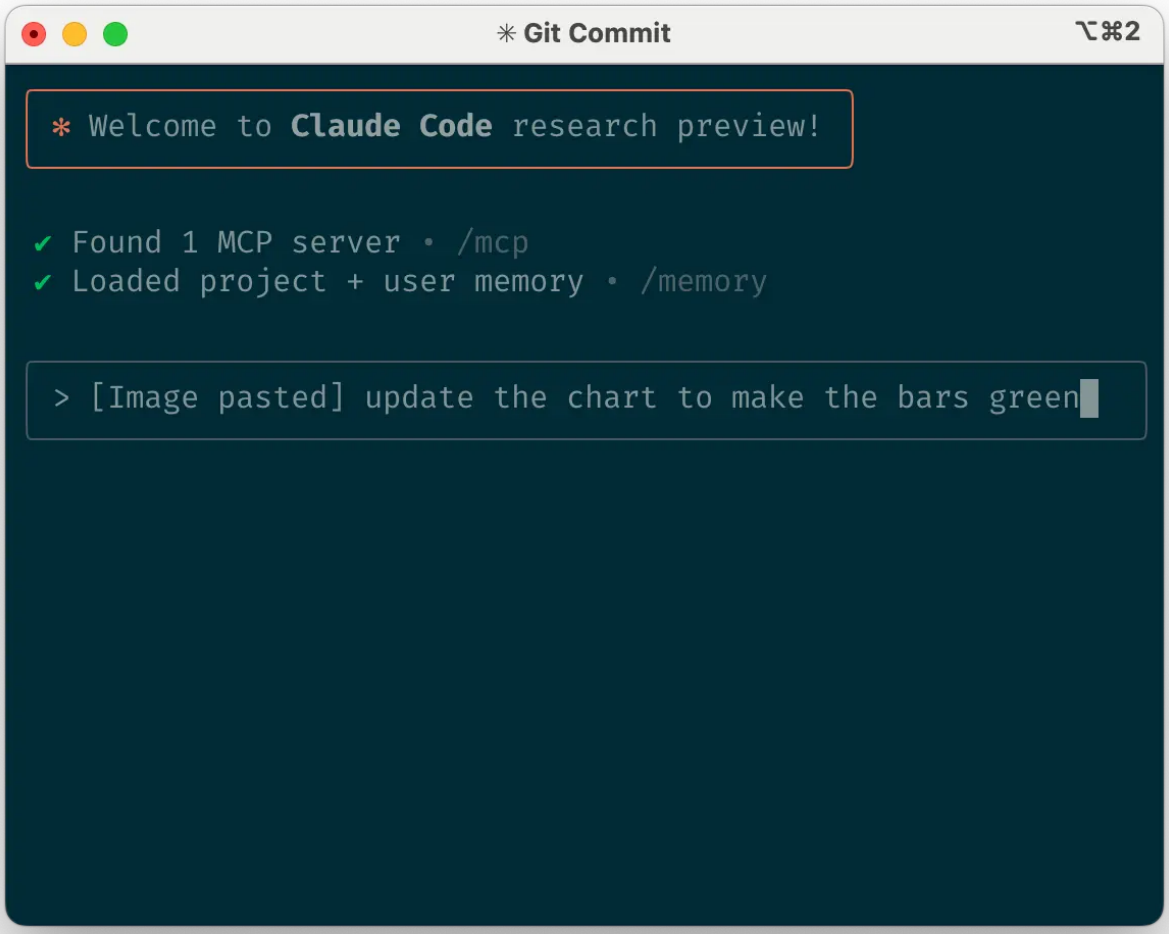
a. Be specific in your instructions

Claude Code’s success rate improves significantly with more specific instructions, especially on first attempts. Giving clear directions upfront reduces the need for course corrections later.

For example:

Poor	Good
add tests for foo.py	write a new test case for foo.py, covering the edge case where the user is logged out. avoid mocks
why does ExecutionFactory have such a weird api?	look through ExecutionFactory's git history and summarize how its api came to be
add a calendar widget	look at how existing widgets are implemented on the home page to understand the patterns and specifically how code and interfaces are separated out. HotDogWidget.php is a good example to start with. then, follow the pattern to implement a new calendar widget that lets the user select a month and paginate forwards/backwards to pick a year. Build from scratch without libraries other than the ones already used in the rest of the codebase.

Claude can infer intent, but it can't read minds. Specificity leads to better alignment with expectations.

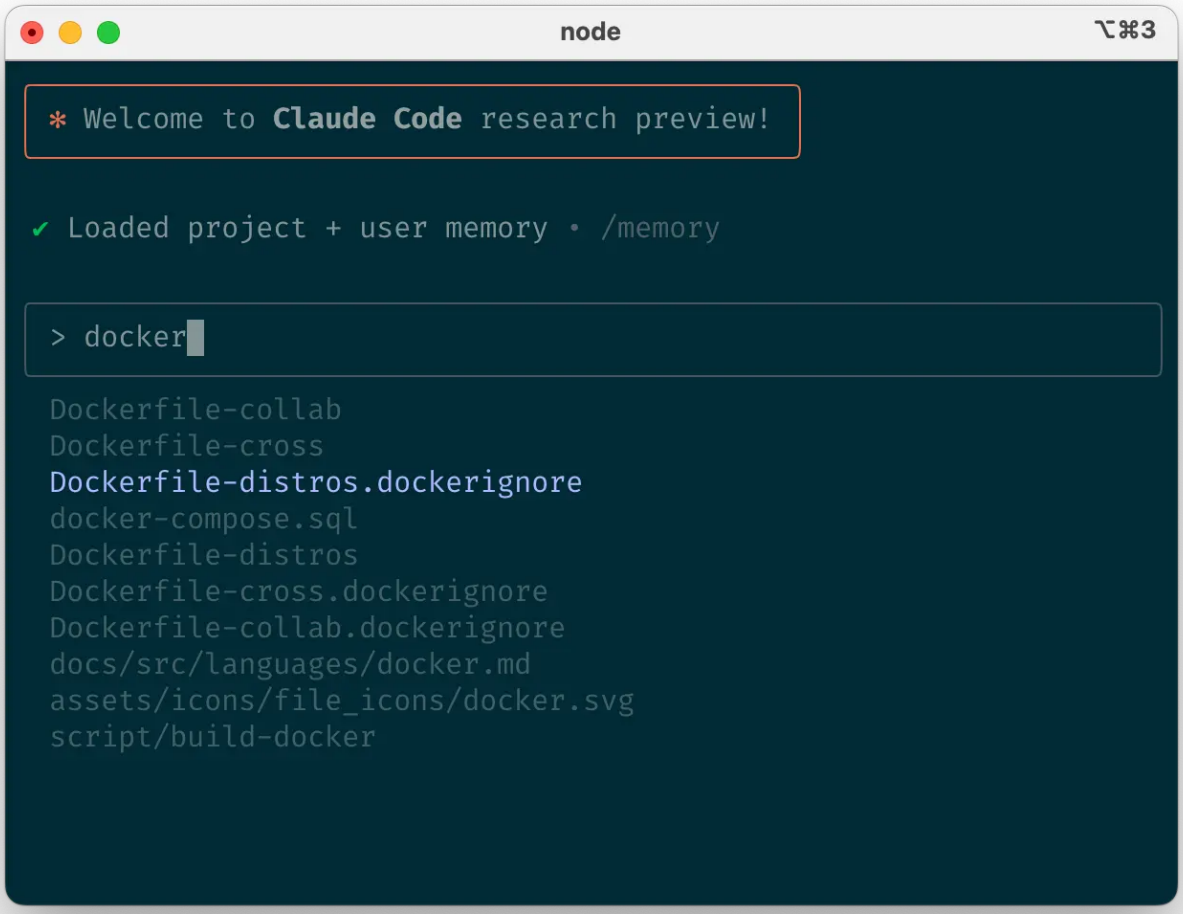


b. Give Claude images

Claude excels with images and diagrams through several methods:

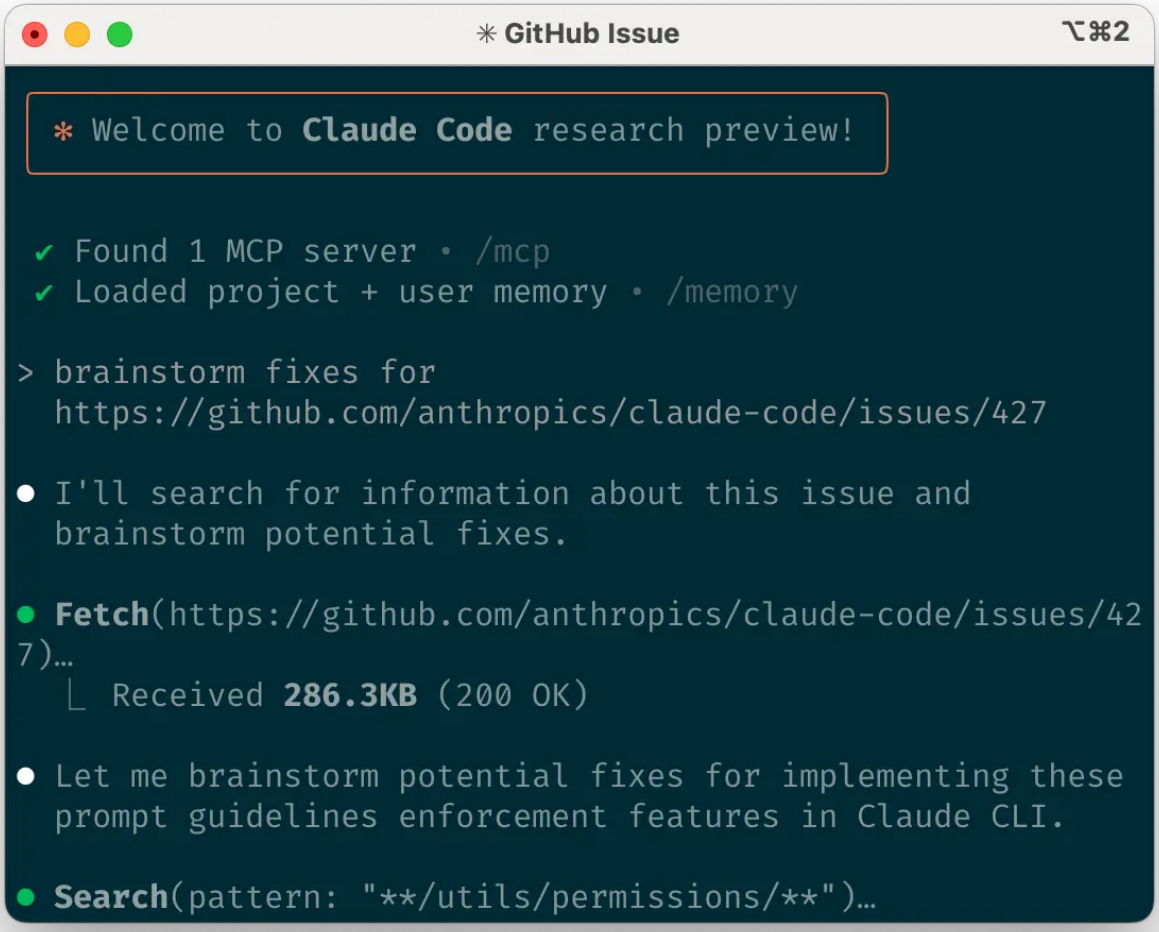
- **Paste screenshots** (pro tip: hit *cmd+ctrl+shift+4* in macOS to screenshot to clipboard and *ctrl+v* to paste. Note that this is not cmd+v like you would usually use to paste on mac and does not work remotely.)
- **Drag and drop** images directly into the prompt input
- **Provide file paths** for images

This is particularly useful when working with design mocks as reference points for UI development, and visual charts for analysis and debugging. If you are not adding visuals to context, it can still be helpful to be clear with Claude about how important it is for the result to be visually appealing.



c. Mention files you want Claude to look at or work on

Use tab-completion to quickly reference files or folders anywhere in your repository, helping Claude find or update the right resources.



d. Give Claude URLs

Paste specific URLs alongside your prompts for Claude to fetch and read. To avoid permission prompts for the same domains (e.g., docs.foo.com), use `/permissions` to add domains to your allowlist.

e. Course correct early and often

While auto-accept mode (shift+tab to toggle) lets Claude work autonomously, you'll typically get better results by being an active collaborator and guiding Claude's approach. You can get the best results by thoroughly explaining the task to Claude at the beginning, but you can also course correct Claude at any time.

These four tools help with course correction:

- **Ask Claude to make a plan** before coding. Explicitly tell it not to code until you've confirmed its plan looks good.
- **Press Escape to interrupt** Claude during any phase (thinking, tool calls, file edits), preserving context

so you can redirect or expand instructions.

- **Double-tap Escape to jump back in history**, edit a previous prompt, and explore a different direction. You can edit the prompt and repeat until you get the result you're looking for.
- **Ask Claude to undo changes**, often in conjunction with option #2 to take a different approach.

Though Claude Code occasionally solves problems perfectly on the first attempt, using these correction tools generally produces better solutions faster.

f. Use `/clear` to keep context focused

During long sessions, Claude's context window can fill with irrelevant conversation, file contents, and commands. This can reduce performance and sometimes distract Claude. Use the `/clear` command frequently between tasks to reset the context window.

g. Use checklists and scratchpads for complex workflows

For large tasks with multiple steps or requiring exhaustive solutions—like code migrations, fixing numerous lint errors, or running complex build scripts—improve performance by having Claude use a Markdown file (or even a GitHub issue!) as a checklist and working scratchpad:

For example, to fix a large number of lint issues, you can do the following:

1. **Tell Claude to run the lint command** and write all resulting errors (with filenames and line numbers) to a Markdown checklist
2. **Instruct Claude to address each issue one by one**, fixing and verifying before checking it off and moving to the next

h. Pass data into Claude

Several methods exist for providing data to Claude:

- **Copy and paste** directly into your prompt (most common approach)
- **Pipe into Claude Code** (e.g., `cat foo.txt | claude`), particularly useful for logs, CSVs, and large data
- **Tell Claude to pull data** via bash commands, MCP tools, or custom slash commands
- **Ask Claude to read files** or fetch URLs (works for images too)

Most sessions involve a combination of these approaches. For example, you can pipe in a log file, then tell Claude to use a tool to pull in additional context to debug the logs.

5. Use headless mode to automate your infra

Claude Code includes headless mode for non-interactive contexts like CI, pre-commit hooks, build scripts, and automation. Use the `-p` flag with a prompt to enable headless mode, and `--output-format stream-json` for streaming JSON output.

Note that headless mode does not persist between sessions. You have to trigger it each session.

a. Use Claude for issue triage

Headless mode can power automations triggered by GitHub events, such as when a new issue is created in your repository. For example, the public Claude Code repository uses Claude to inspect new issues as they come in and assign appropriate labels.

b. Use Claude as a linter

Claude Code can provide subjective code reviews beyond what traditional linting tools detect, identifying issues like typos, stale comments, misleading function or variable names, and more.

6. Uplevel with multi-Claude workflows

Beyond standalone usage, some of the most powerful applications involve running multiple Claude instances in parallel:

a. Have one Claude write code; use another Claude to verify

A simple but effective approach is to have one Claude write code while another reviews or tests it. Similar to working with multiple engineers, sometimes having separate context is beneficial:

1. Use Claude to write code
2. Run `/clear` or start a second Claude in another terminal
3. Have the second Claude review the first Claude's work
4. Start another Claude (or `/clear` again) to read both the code and review feedback
5. Have this Claude edit the code based on the feedback

You can do something similar with tests: have one Claude write tests, then have another Claude write code to make the tests pass. You can even have your Claude instances communicate with each other by giving them separate working scratchpads and telling them which one to write to and which one to read from.

This separation often yields better results than having a single Claude handle everything.

b. Have multiple checkouts of your repo

Rather than waiting for Claude to complete each step, something many engineers at Anthropic do is:

1. **Create 3-4 git checkouts** in separate folders
2. **Open each folder** in separate terminal tabs
3. **Start Claude in each folder** with different tasks
4. **Cycle through** to check progress and approve/deny permission requests

c. Use git worktrees

This approach shines for multiple independent tasks, offering a lighter-weight alternative to multiple checkouts. Git worktrees allow you to check out multiple branches from the same repository into separate directories. Each worktree has its own working directory with isolated files, while sharing the same Git history and reflog.

Using git worktrees enables you to run multiple Claude sessions simultaneously on different parts of your project, each focused on its own independent task. For instance, you might have one Claude refactoring your authentication system while another builds a completely unrelated data visualization component. Since the tasks don't overlap, each Claude can work at full speed without waiting for the other's changes or dealing with merge conflicts:

1. **Create worktrees**: `git worktree add ../project-feature-a feature-a`
2. **Launch Claude in each worktree**: `cd ../project-feature-a && claude`
3. **Create additional worktrees** as needed (repeat steps 1-2 in new terminal tabs)

Some tips:

- Use consistent naming conventions
- Maintain one terminal tab per worktree
- If you're using iTerm2 on Mac, set up notifications for when Claude needs attention
- Use separate IDE windows for different worktrees
- Clean up when finished: `git worktree remove ../project-feature-a`

d. Use headless mode with a custom harness

`claude -p` (headless mode) integrates Claude Code programmatically into larger workflows while leveraging its built-in tools and system prompt. There are two primary patterns for using headless mode:

1. **Fanning out** handles large migrations or analyses (e.g., analyzing sentiment in hundreds of logs or analyzing thousands of CSVs):

1. Have Claude write a script to generate a task list. For example, generate a list of 2k files that need to be migrated from framework A to framework B.
2. Loop through tasks, calling Claude programmatically for each and giving it a task and a set of tools it

can use. For example: `claude -p "migrate foo.py from React to Vue. When you are done, you MUST return the string OK if you succeeded, or FAIL if the task failed." --allowedTools Edit Bash(git commit:*)`

3. Run the script several times and refine your prompt to get the desired outcome.
4. **Pipelining** integrates Claude into existing data/processing pipelines:

1. Call `claude -p "<your prompt>" --json | your_command`, where `your_command` is the next step of your processing pipeline
2. That's it! JSON output (optional) can help provide structure for easier automated processing.

For both of these use cases, it can be helpful to use the `--verbose` flag for debugging the Claude invocation. We generally recommend turning verbose mode off in production for cleaner output.

What are your tips and best practices for working with Claude Code? Tag @AnthropicAI so we can see what you're building!

Acknowledgements

Written by Boris Cherny. This work draws upon best practices from across the broader Claude Code user community, whose creative approaches and workflows continue to inspire us. Special thanks also to Daisy Hollman, Ashwin Bhat, Cat Wu, Sid Bidasaria, Cal Rueb, Nodir Turakulov, Barry Zhang, Drew Hodun and many other Anthropic engineers whose valuable insights and practical experience with Claude Code helped shape these recommendations.