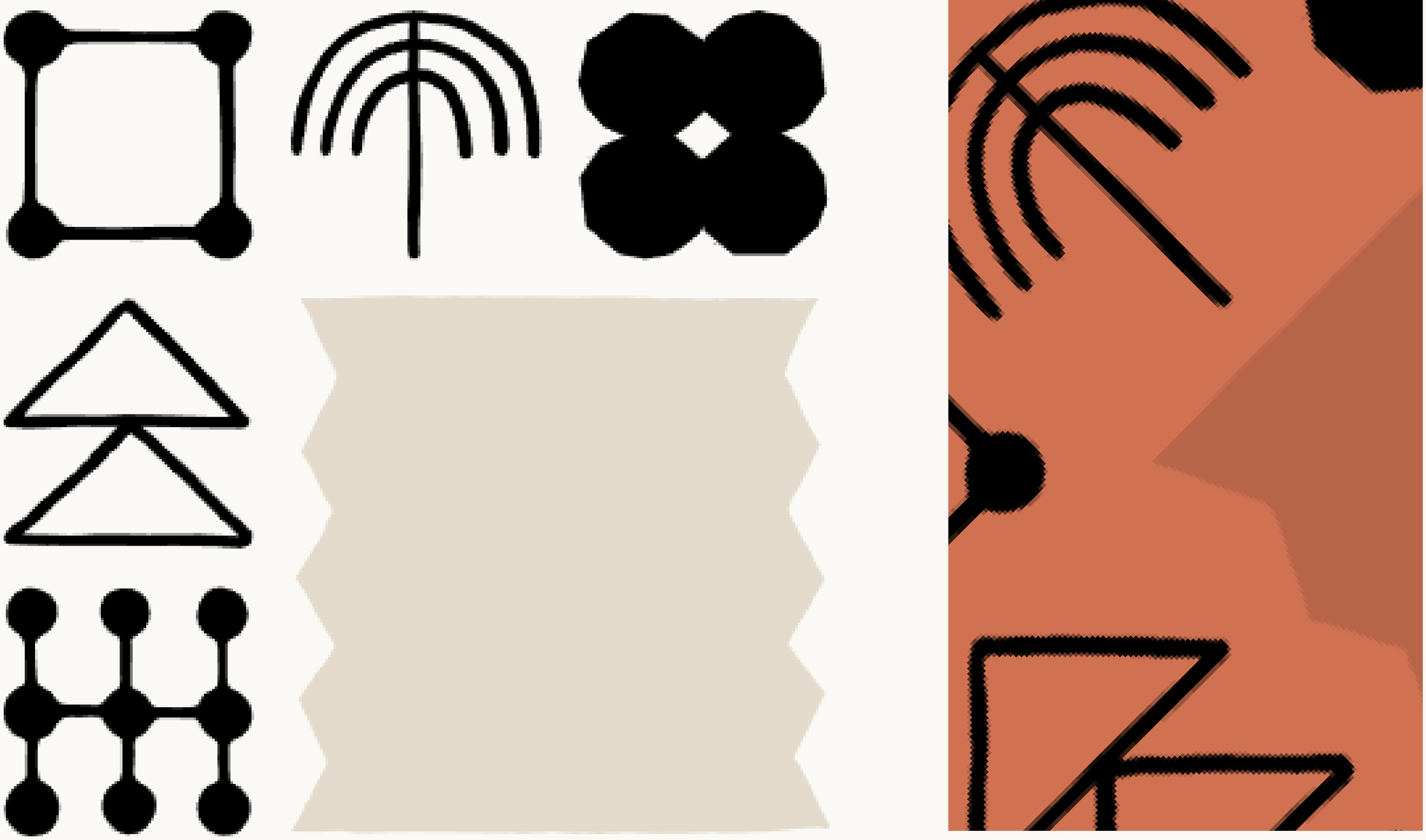
ANTHROP\C



Claude Code

Best practices for agentic coding

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.

ANTHROP\C ——— Research article available for access at: https://www.anthropic.com/engineering/claude-code-best-practices.

Table of Content

Preface

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 article 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
```

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

- Repo Root or Run Directory: Name it `CLAUDE.md` and check into git for sharing, or `CLAUDE.local.md` and `.gitignore` it.

- Parent Directories: Useful for monorepos, with `CLAUDE.md` files in parent directories.

- Child Directories: `CLAUDE.md` files in child directories are pulled on demand.

- Home Folder: Place in `~/.claude/CLAUDE.md` for all sessions.

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

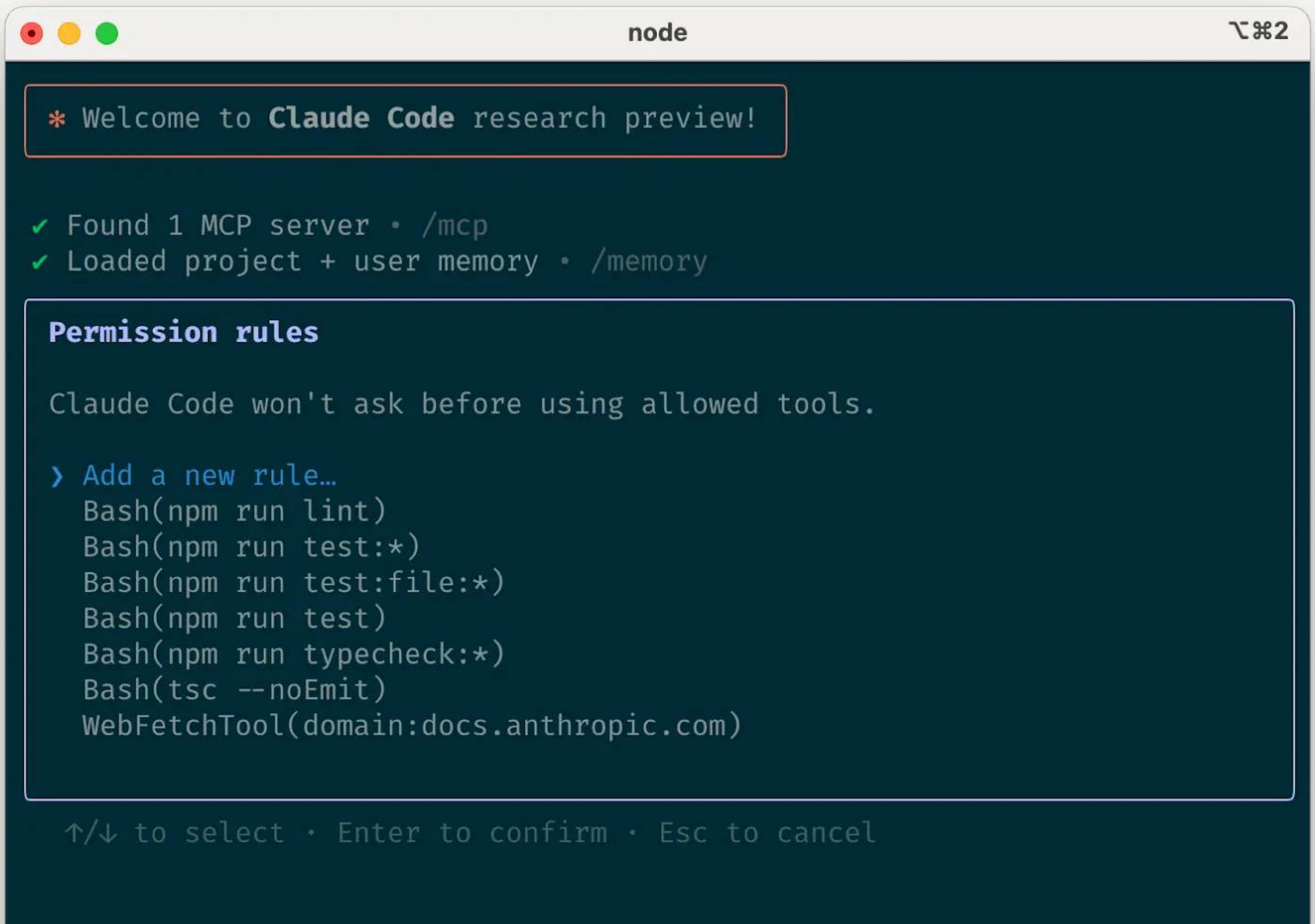
b. Tune your CLAUDE.md files

Your CLAUDE.md files become part of Claude's prompts.

A common mistake is adding extensive content without iterating on its effectiveness.

Add content to `CLAUDE.md` manually or press `#` to auto-incorporate instructions. Engineers use `#` to document commands and guidelines, committing changes to `CLAUDE.md` for team use.

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.



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 /allowed-tools 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 those 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. For example, Add Puppeteer and Sentry servers to a checked-in `.mcp.json` file for all engineers to use.

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.

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 Wrokflows

Claude Code offers flexible usage without imposing a specific workflow. Successful usage patterns have emerged within the user community.

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.
    a. In this part, use subagents for complex problems to verify details and preserve context, especially early in a conversation or task, without losing 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.
    a. If the results seem reasonable, have Claude create a document or GitHub issue with its plan. This allows you to reset if the implementation 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 may jump straight to coding. Researching and planning first improves performance for complex problems.

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:

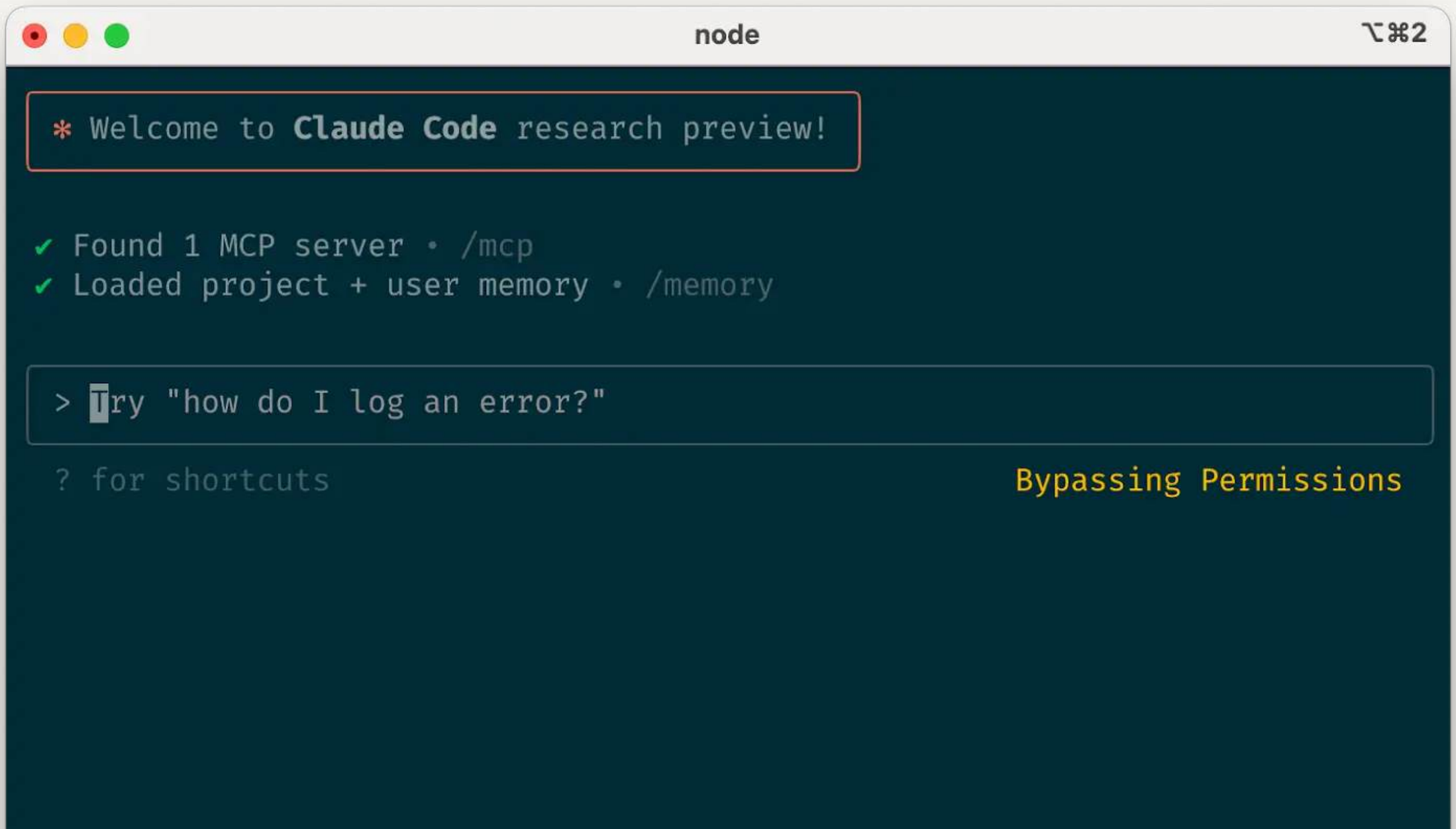
1. Ask Claude to write tests based on expected input/output pairs. Specify that you're doing test-driven development to avoid mock implementations.
2. Tell Claude to run the tests and confirm they fail. Explicitly telling it not to write any implementation code at this stage is often helpful.
3. Ask Claude to commit the tests when you're satisfied with them.
4. Ask Claude to write code that passes the tests without modifying them. Instruct it to keep going until all tests pass, which may take a few iterations.
    a. At this stage, it can help to ask it to verify with independent subagents that the implementation isn't overfitting to the tests

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.

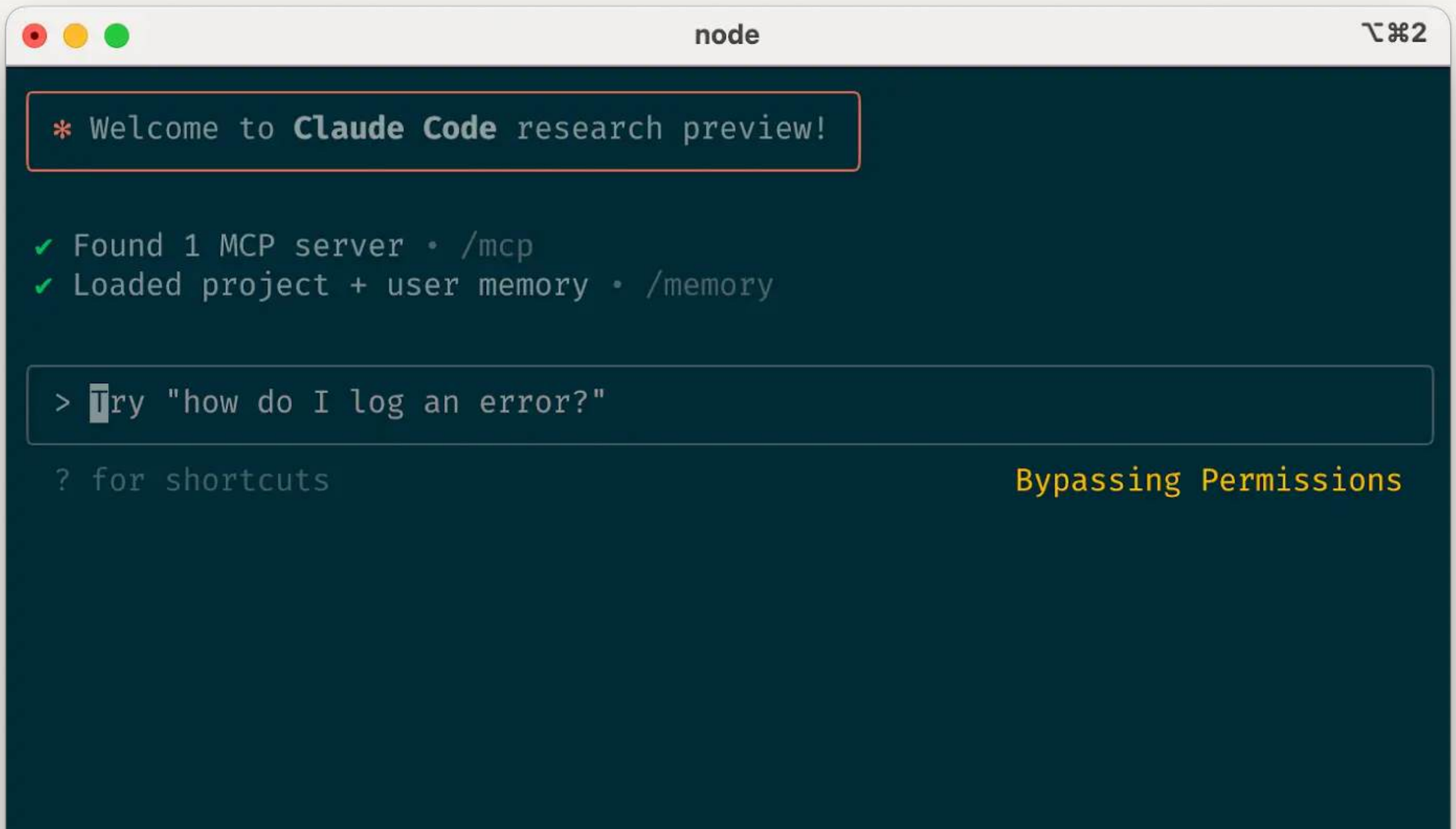


c. Write code, screenshot result, iterate

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.

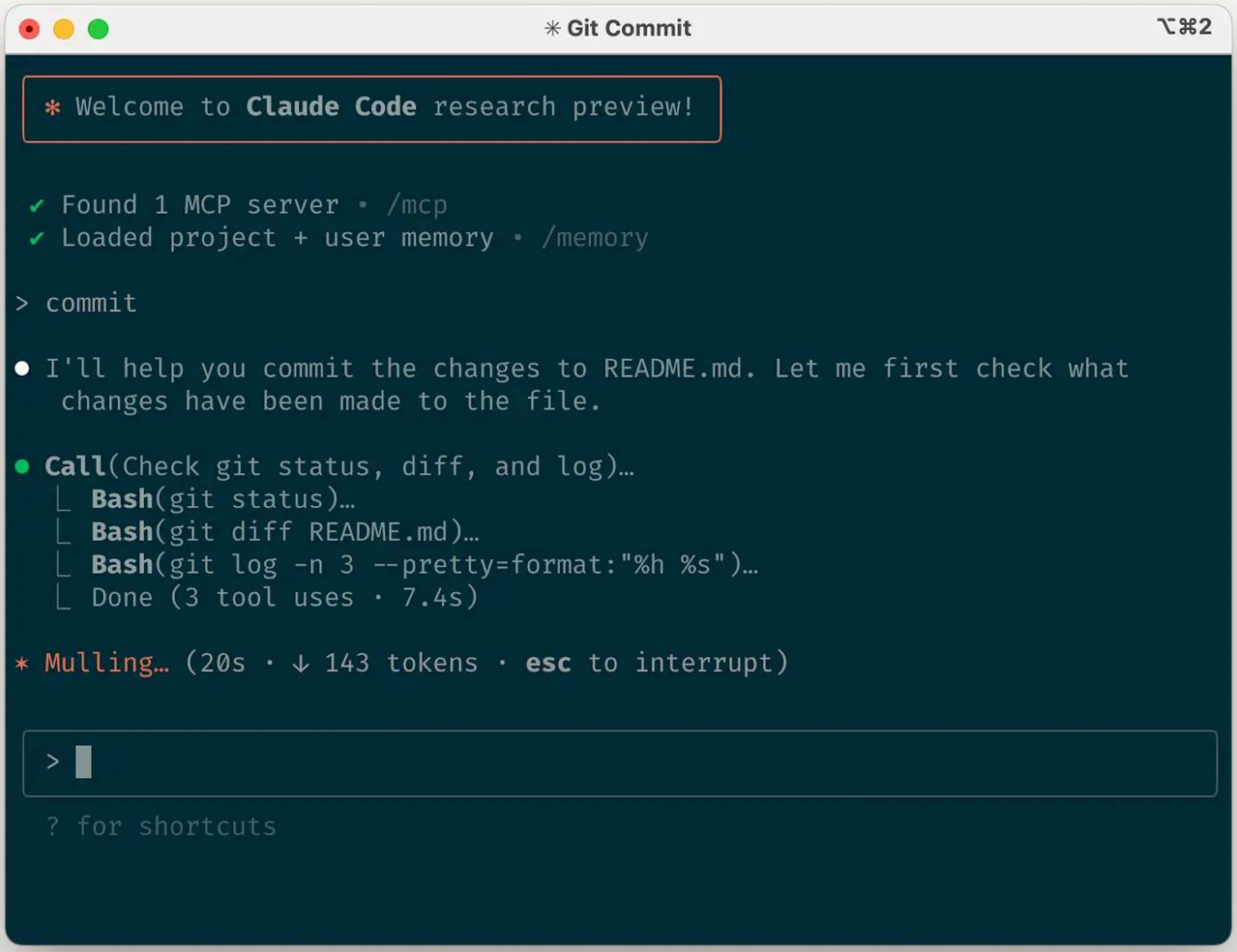
e. Codebase Q&A

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.

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:

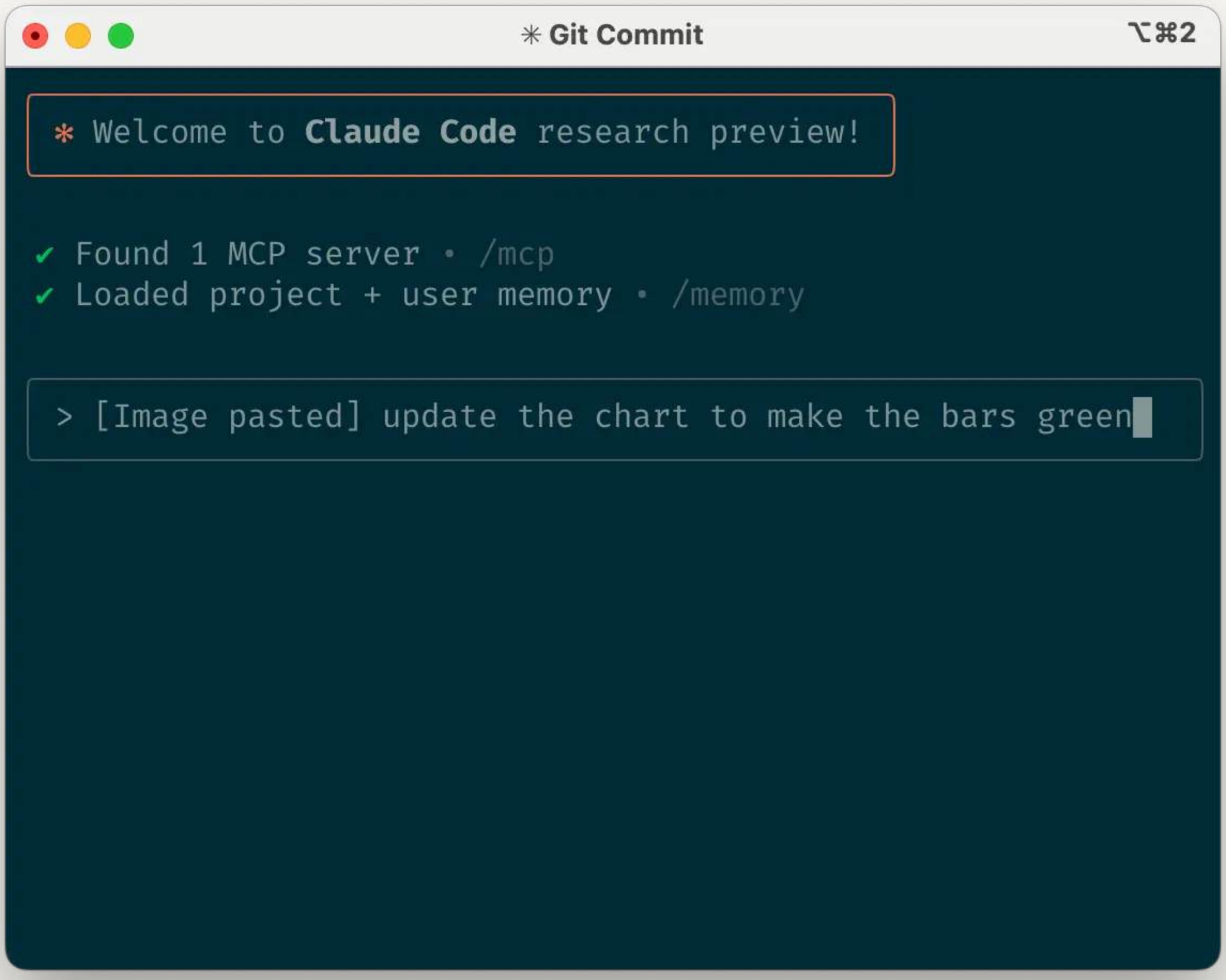
a. Be specific in your instructions

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.

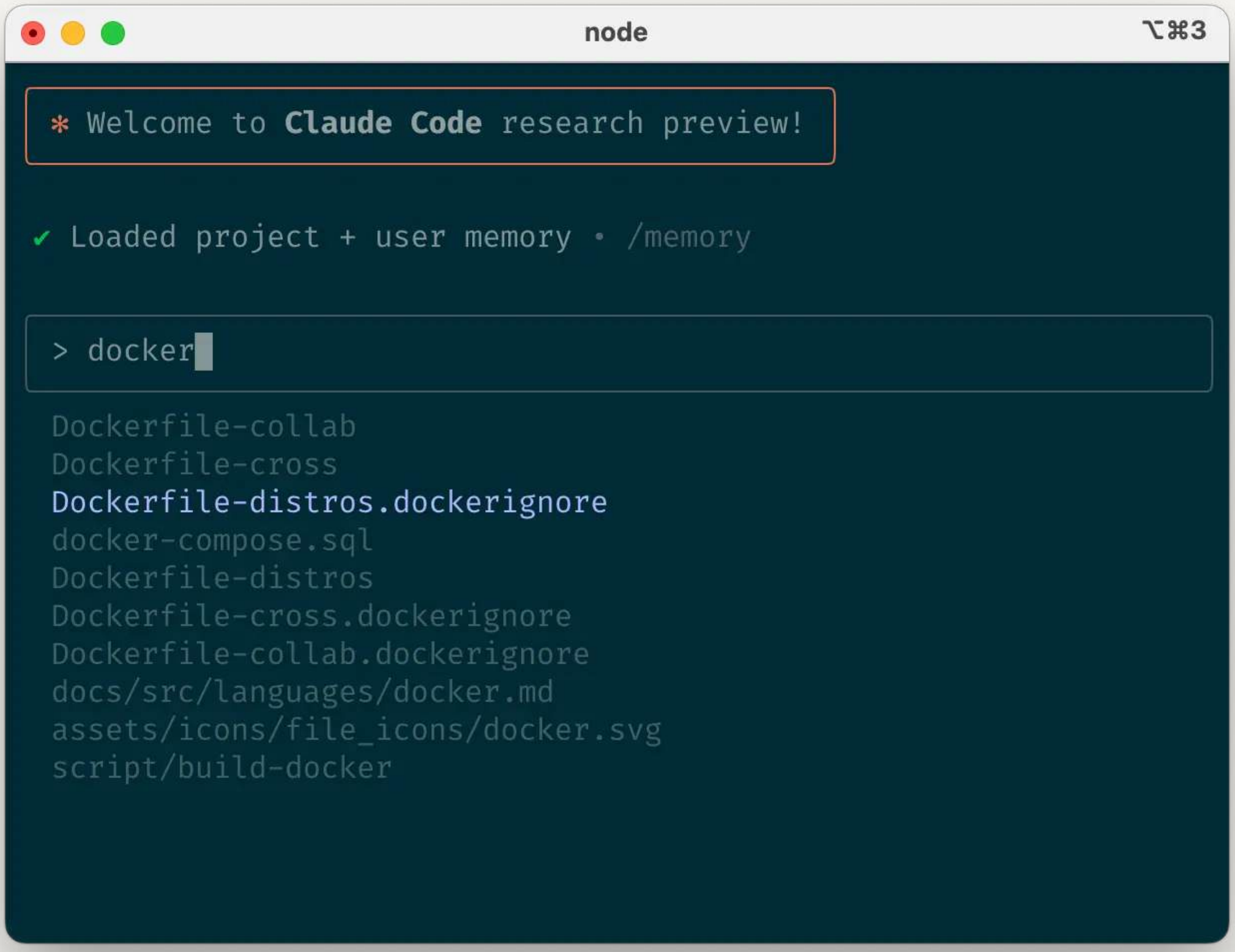
Claude can infer intent, but it can't read minds. Specificity leads to better alignment with expectations.



b. Give Claude images

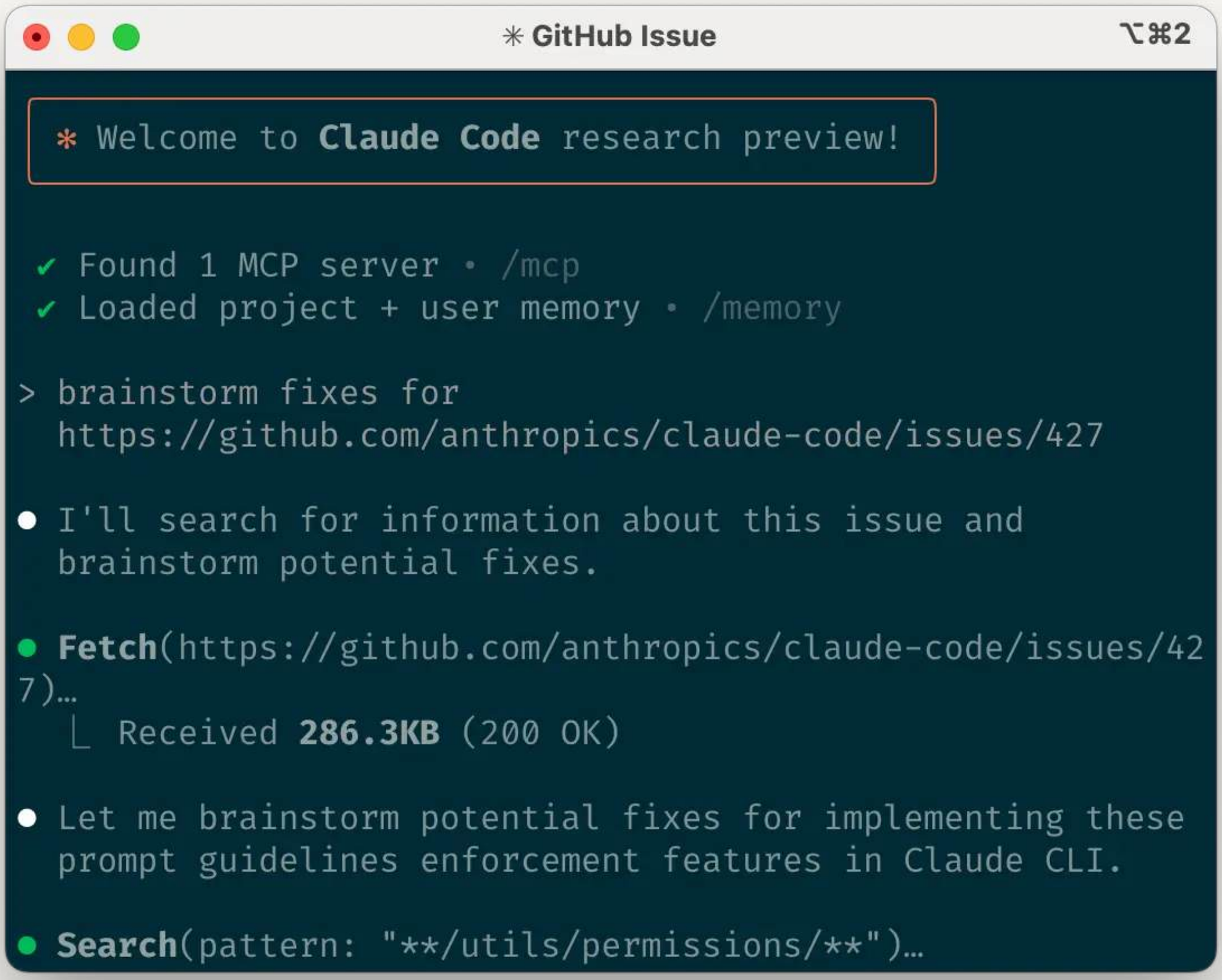
Claude excels with images and diagrams through several methods:

- 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.



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 /allowed-tools 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.

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

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.

2. 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.

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.

From **Anthropic**