# Effective Engineering with Cline

# Why This Matters

- Tech leaders are aggressively pushing for AI adoption
- Public perception focuses on "vibe coding"
- Experienced engineers stand to gain the most from AI tools

# Common Objections

- "I could code myself faster because I know the codebase"
- "It does too much and makes mistakes"
- "It only works if you're working on a new codebase"
- "It takes away all the joy in writing code"
- "My neovim setup is too good"

Explore → Plan → Build

# Practical examples

Three ways I use Cline every day

# Test Driven Development

How to use Cline to write better tests

# Test Driven Development

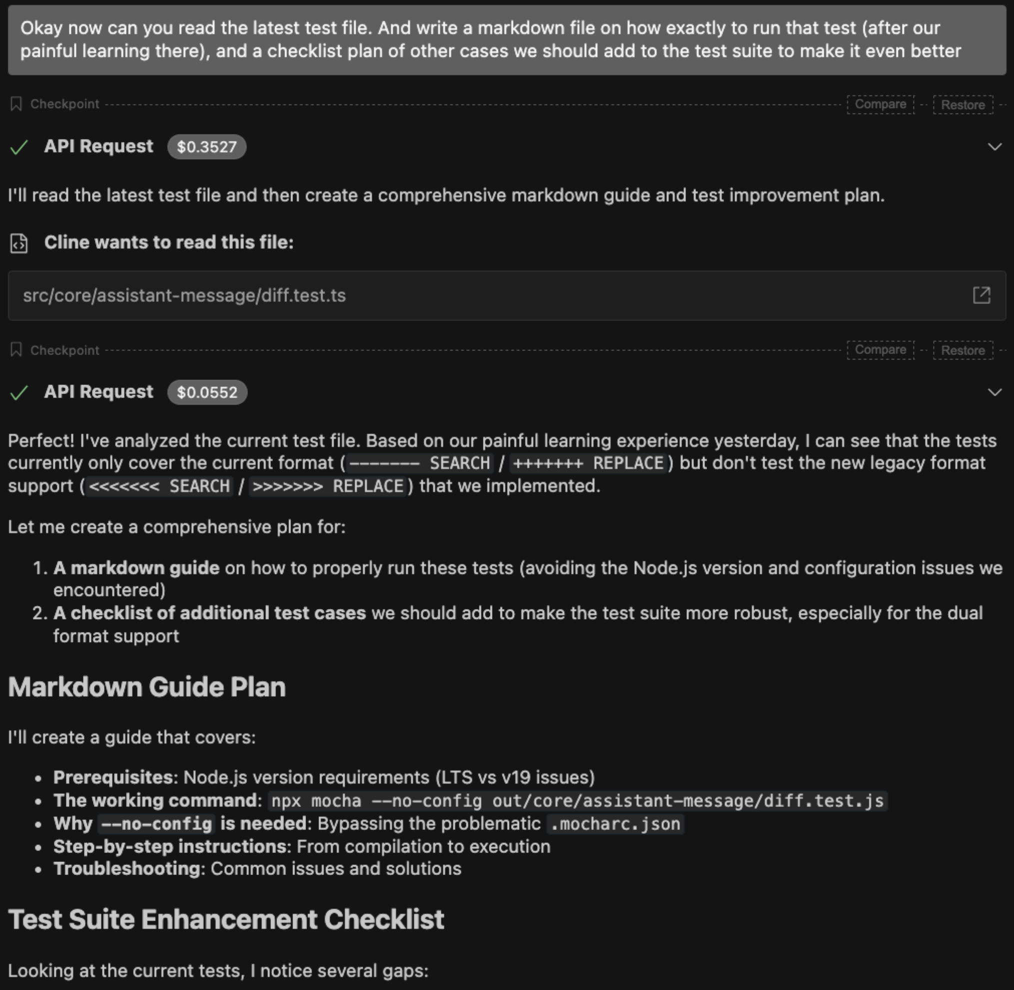
Explore → Plan → Build

- Share context about files needing tests
  - Cline discovers related files and dependencies
  - Provide similar test files in codebase
- Talk through testing goals and options with Cline

# Test Driven Development

Explore → Plan → Build

- Create a structured markdown plan
- Break down into discrete steps/tasks
- Include a checklist for tracking progress
- Plan enables resuming work later



Okay now can you read the latest test file. And write a markdown file on how exactly to run that test (after our painful learning there), and a checklist plan of other cases we should add to the test suite to make it even better

✓ **API Request** $0.3527

I'll read the latest test file and then create a comprehensive markdown guide and test improvement plan.

**Cline wants to read this file:**

src/core/assistant-message/diff.test.ts

✓ **API Request** $0.0552

Perfect! I've analyzed the current test file. Based on our painful learning experience yesterday, I can see that the tests currently only cover the current format (`------- SEARCH` / `+++++++ REPLACE`) but don't test the new legacy format support (`<<<<<<< SEARCH` / `>>>>>>> REPLACE`) that we implemented.

Let me create a comprehensive plan for:

1. **A markdown guide** on how to properly run these tests (avoiding the Node.js version and configuration issues we encountered)
2. **A checklist of additional test cases** we should add to make the test suite more robust, especially for the dual format support

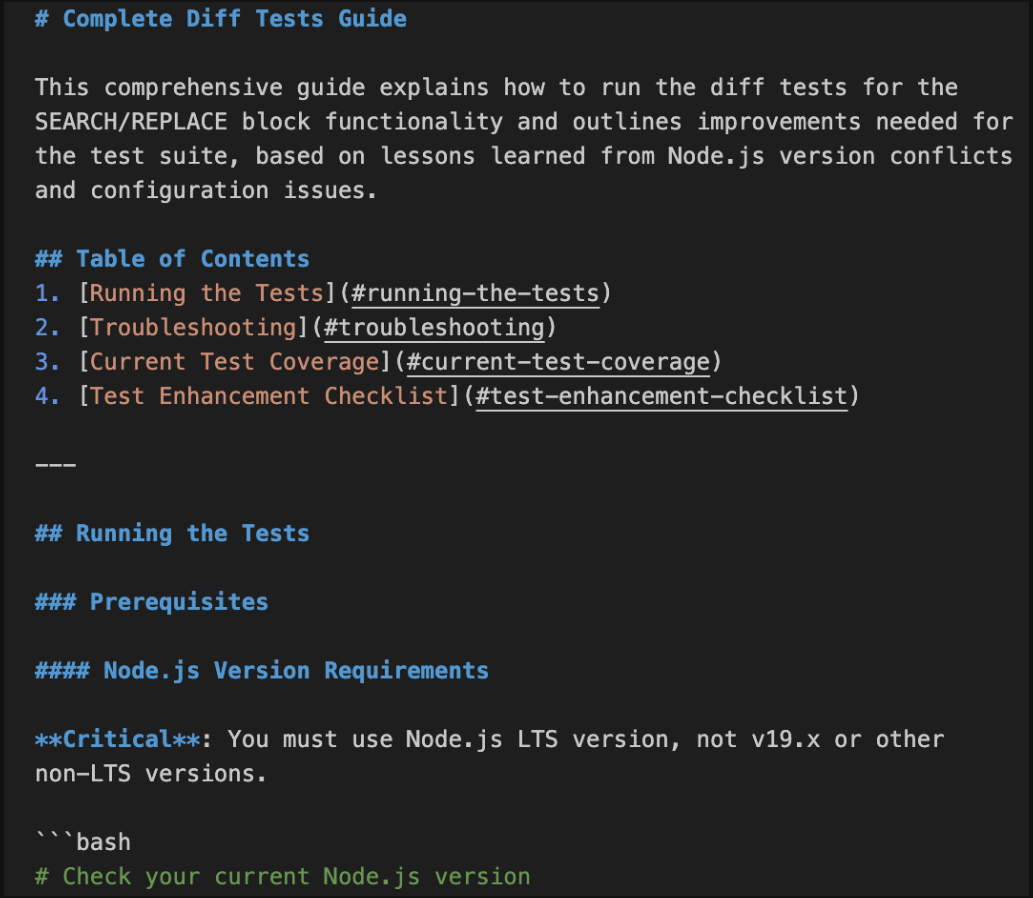
## Markdown Guide Plan

I'll create a guide that covers:

- **Prerequisites**: Node.js version requirements (LTS vs v19 issues)
- **The working command**: `npx mocha --no-config out/core/assistant-message/diff.test.js`
- **Why `--no-config` is needed**: Bypassing the problematic `.mocharc.json`
- **Step-by-step instructions**: From compilation to execution
- **Troubleshooting**: Common issues and solutions

## Test Suite Enhancement Checklist

Looking at the current tests, I notice several gaps:



# Complete Diff Tests Guide

This comprehensive guide explains how to run the diff tests for the SEARCH/REPLACE block functionality and outlines improvements needed for the test suite, based on lessons learned from Node.js version conflicts and configuration issues.

## Table of Contents

1. [Running the Tests](#running-the-tests)
2. [Troubleshooting](#troubleshooting)
3. [Current Test Coverage](#current-test-coverage)
4. [Test Enhancement Checklist](#test-enhancement-checklist)

---

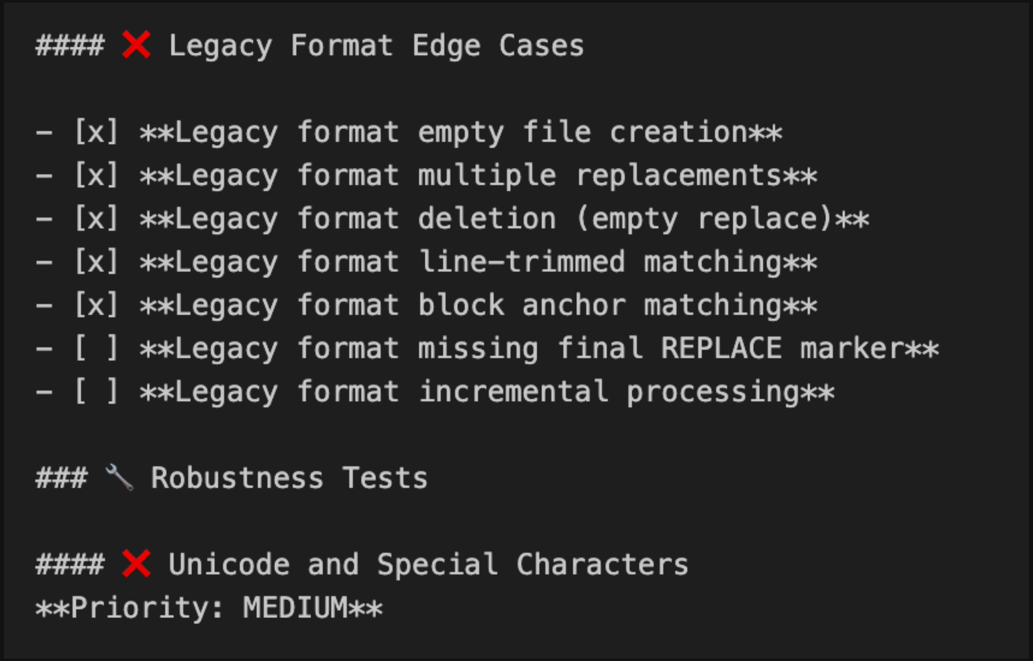
## Running the Tests

### Prerequisites

#### Node.js Version Requirements

**Critical**: You must use Node.js LTS version, not v19.x or other non-LTS versions.

```bash
# Check your current Node.js version
```

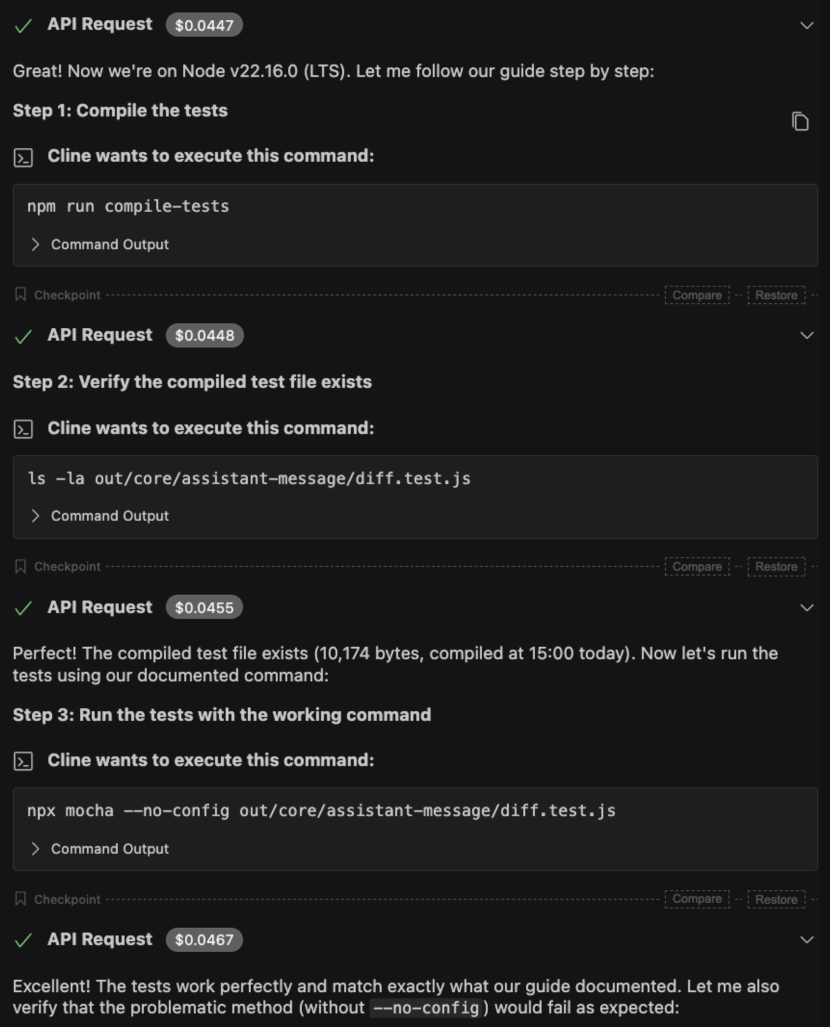


```
#### ❌ Legacy Format Edge Cases

- [x] **Legacy format empty file creation**
- [x] **Legacy format multiple replacements**
- [x] **Legacy format deletion (empty replace)**
- [x] **Legacy format line-trimmed matching**
- [x] **Legacy format block anchor matching**
- [ ] **Legacy format missing final REPLACE marker**
- [ ] **Legacy format incremental processing**

### 🔧 Robustness Tests

#### ❌ Unicode and Special Characters
**Priority: MEDIUM**
```

# Test Driven Development

Explore → Plan → Build

- Apply Steps one by one
- Make adjustments as needed
- Commit to git and update the plan file



Great! Now we're on Node v22.16.0 (LTS). Let me follow our guide step by step:

**Step 1: Compile the tests**

**Cline wants to execute this command:**

```
npm run compile-tests
```

**Step 2: Verify the compiled test file exists**

**Cline wants to execute this command:**

```
ls -la out/core/assistant-message/diff.test.js
```

Perfect! The compiled test file exists (10,174 bytes, compiled at 15:00 today). Now let's run the tests using our documented command:

**Step 3: Run the tests with the working command**

**Cline wants to execute this command:**

```
npx mocha --no-config out/core/assistant-message/diff.test.js
```

Excellent! The tests work perfectly and match exactly what our guide documented. Let me also verify that the problematic method (without `--no-config`) would fail as expected:

# Test Driven Development

**Takeaways**

- Tests actually get written
- Easy to resume after interruptions
- I no longer feel like an asshole for asking contributors to write tests

# PR Comments

How to use Cline to handle feedback faster

# PR Comments

Explore → Plan → Build

- Ask Cline to use `gh` to gather the comments on my PR into context, as well as the full diff of the files changed
- Cline uses commands like `gh pr view` and `gh pr diff` to gather the context
- To get a better understanding of the landscape, Cline explores the surrounding files and reads them into context as well

# PR Comments

Explore → Plan → Build

- Ask Cline to group all related changes
- Create a plan file with relevant context, file names, and checklist of discrete steps (optional, depending on number of requested changes)
- Don't edit the code yet

# PR Comments

Explore → Plan → Build

- Start a new task, read the plan file
- Cline reads the relevant files and proposes specific changes
  - I review the changes, if they look good I toggle to Act Mode
  - If I'm not happy with the changes, I stay in Plan Mode, and ask Cline to adjust the proposal until I'm happy
- Cline makes changes and asks me if I want to commit them one by one with the `gh` tool.

# PR Comments

**Takeaways**

- Focus on the feedback that actually matters
- No need to get annoyed by nits anymore
- Tip: Make sure you have the `gh` command installed and authenticated
  - Works great with agentic tools like Cline
  - Non-interactive diff reading is a breeze
  - Enables Cline to read PR discussions, titles, and respond to comments once changes are made

# Workflows

How to use Cline to automate repetitive tasks

# Workflows

Explore → Plan → Build

- Identify sequences of steps that you currently do manually with Cline
- Ask Cline to make a plan to distill the sequence of steps into a markdown file
- Tell Cline to include information about the specific underlying tools to call

# Workflows

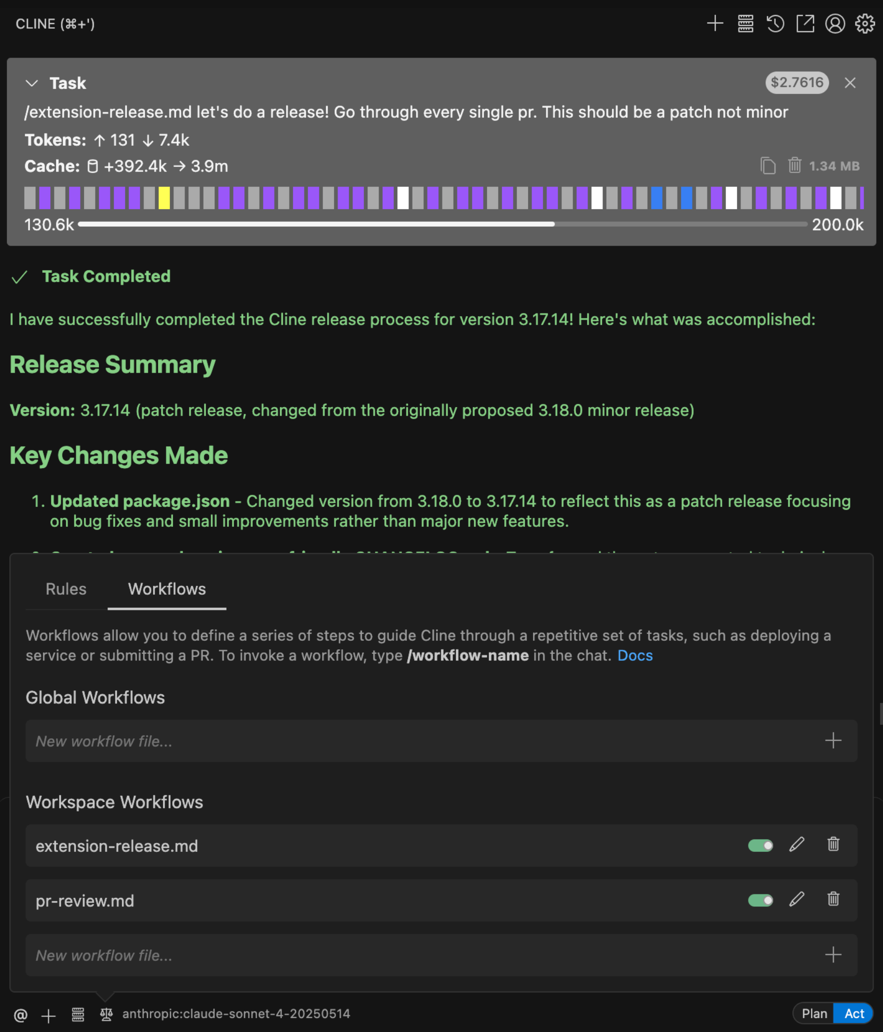
Explore → Plan → Build

- Cline writes a plan describing techniques and tools, and how to put them into a reusable workflow file
- Make sure the plan is bullet proof and that Cline understands the context of how you'll be using the workflow
- Once plan looks solid, switch to Act Mode and Cline will make the markdown for you

# Workflows

Explore → Plan → **Build**

- Once the Workflow is written invoke it with a slash command
- Refine the Workflow over time
- Never do things manually ever again
- Share your Workflow with your teammates



/extension-release.md let's do a release! Go through every single pr. This should be a patch not minor

**Tokens:** ↑ 131 ↓ 7.4k

**Cache:** +392.4k → 3.9m

✓ **Task Completed**

I have successfully completed the Cline release process for version 3.17.14! Here's what was accomplished:

## Release Summary

**Version:** 3.17.14 (patch release, changed from the originally proposed 3.18.0 minor release)

## Key Changes Made

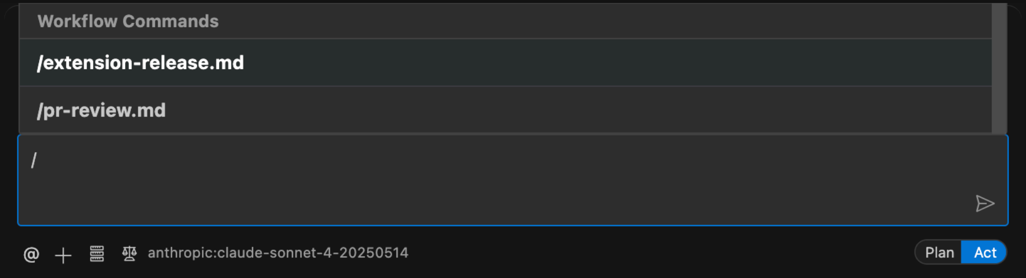
1. **Updated package.json** - Changed version from 3.18.0 to 3.17.14 to reflect this as a patch release focusing on bug fixes and small improvements rather than major new features.

Workflows allow you to define a series of steps to guide Cline through a repetitive set of tasks, such as deploying a service or submitting a PR. To invoke a workflow, type **/workflow-name** in the chat. Docs

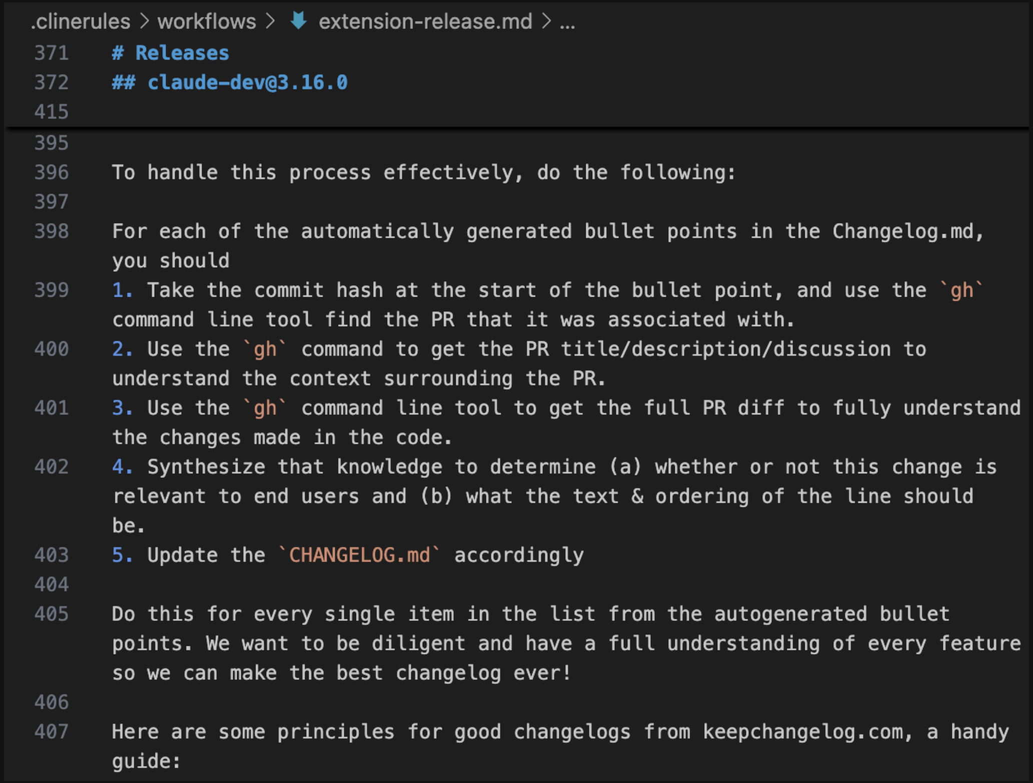
### Global Workflows

### Workspace Workflows

- extension-release.md
- pr-review.md



anthropic:claude-sonnet-4-20250514



```markdown
# Releases
## claude-dev@3.16.0


To handle this process effectively, do the following:

For each of the automatically generated bullet points in the Changelog.md, you should
1. Take the commit hash at the start of the bullet point, and use the `gh` command line tool find the PR that it was associated with.
2. Use the `gh` command to get the PR title/description/discussion to understand the context surrounding the PR.
3. Use the `gh` command line tool to get the full PR diff to fully understand the changes made in the code.
4. Synthesize that knowledge to determine (a) whether or not this change is relevant to end users and (b) what the text & ordering of the line should be.
5. Update the `CHANGELOG.md` accordingly

Do this for every single item in the list from the autogenerated bullet points. We want to be diligent and have a full understanding of every feature so we can make the best changelog ever!

Here are some principles for good changelogs from keepchangelog.com, a handy guide:
```

https://github.com/cline/cline/blob/main/.clinerules/workflows/extension-release.md

# Workflows

**Takeaways**

- Workflows are a great way to automate repetitive tasks
- Once you make a bulletproof Workflow, share it with your team. They will be very greatful
- Tip: Leverage Cline's internal tools like `read_file`, `execute_command`, and `ask_followup_question` to get the most out of your Workflow
  - Check out the docs for more information
  - ... or Cline's prompt library for inspiration and examples

# Things I've done with Cline

## Development

- Made this presentation (using slidev)
- Handle merge conflicts
- Create internal eval visualization dashboards with Streamlit
- Debug long tail Sentry errors
- Recorded and transcribed a conversation with a senior Go Engineer on the team, pasted it into Cline, and implemented his advice to save $500/week on Database reads via idiomatic caching

## Documentation & APIs

- Write Documentation for SDKs
- Write RFCs for new features, with implementation plans and mermaid diagrams
- Write PR descriptions from commit diffs
- Fetch and parse API documentation from the web with the Context7 MCP server, and write implementation plans to use it
- Create CLIs around functions

## Code Quality

- Add generics to shared code
- Pick better names for variables
- Learn new areas of new codebases
- Debug production issues
- And much more...

# Lessons from Observing Power Users

- Always invest in the planning phase
- Leverage MCP servers in tasks and workflows
- Distill common patterns into Workflows, and preferences into Cline Rules
- Leave Auto Approve for editing OFF - always review changes along the way to make sure Cline is on the right track
- Idiomatic Context window management is the key to success
  - Use Gemini 2.5 Pro's 1M token context window for planning phase or when dealing with a lot of long files
  - Smart task handoff when approaching context capacity
  - Use markdown files and checklists to keep Cline on track

# Share your experience

- Share this with your engineering team
  - This deck is available on my GitHub: https://github.com/pashpashpash/presentations
- Tag us on LinkedIn or X with how you're using Cline
- Feel free to reach out on Discord or X if you have any questions or feature requests
- We're open source and hiring, you're always welcome to contribute!