I Discovered This Claude Code Framework (That Automates Your Boring Stuff)

Every day, I come across something that makes Claude Code better.

I recently discovered this framework, which provides structured commands to constrain Claude's output to specific, actionable results.

> *Claude Code CLI's responses are conversational and can become verbose or unfocused when handling routine development tasks.*

This framework attempts to give Claude Code prompts more structure and does so with premade commands, as you will see in my test later.

If you have not seen the other best Claude Code addons ***I recently shared, you can check the list here.***

But, for those new to Claude Code,

> *Start with this Claude Code cheat sheet I created for you here*— **we are continually updating as we discover new ideas, tips, and tricks.**

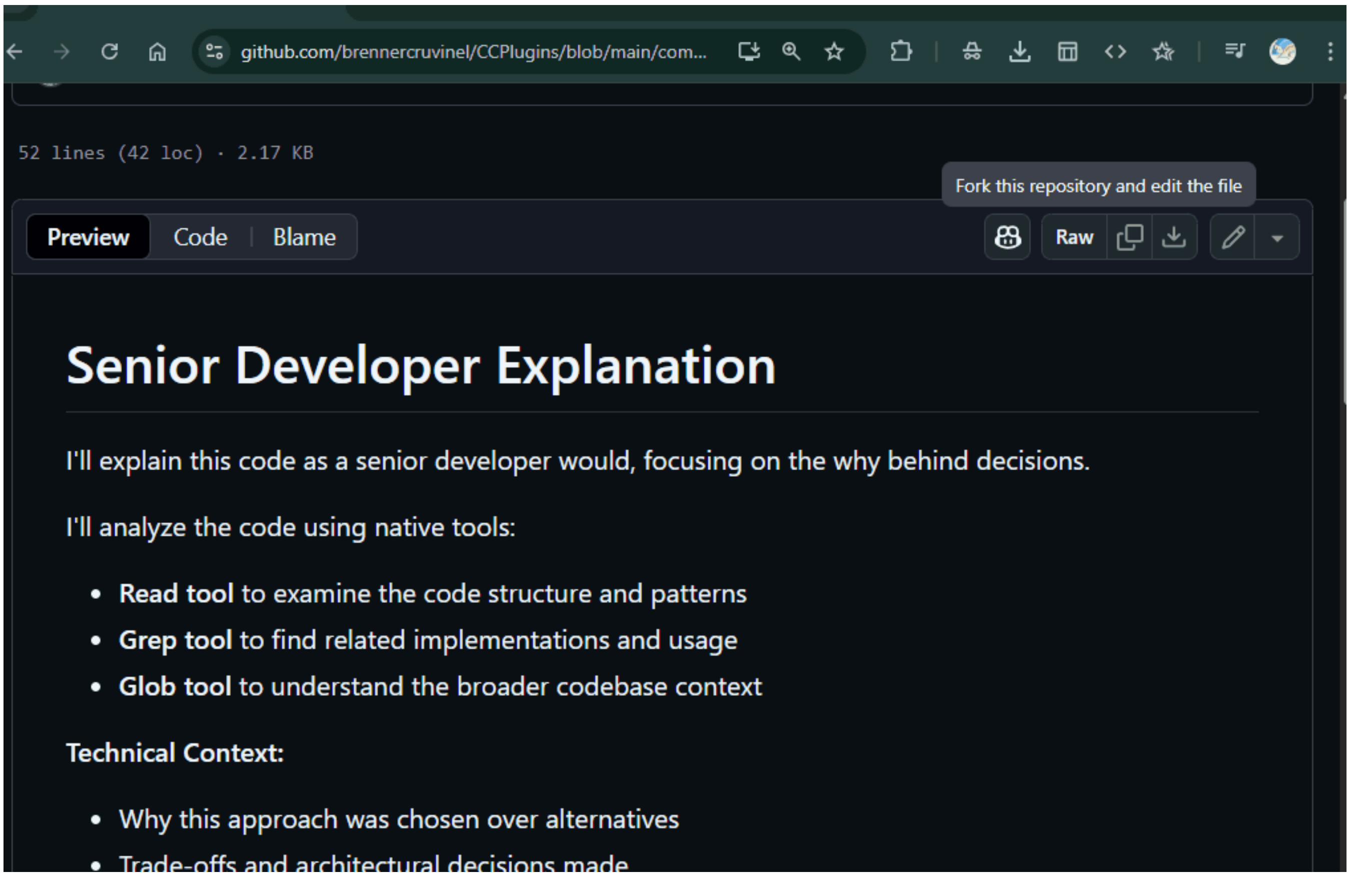
I tried out CCPlugins, and it surprised me with what's possible with Claude Code automation.

What Is Claude Code CCPlugins?

CCPlugins is a collection of 24 predefined commands that extend Claude Code CLI functionality.

> *Each command contains specific instructions that guide Claude to perform targeted development tasks with consistent output formats.*

I dug deep into the repo, and here's an example of the instructions file :


github.com/brennercruvinel/CCPlugins/blob/main/com...
52 lines (42 loc) · 2.17 KB
Fork this repository and edit the file
Preview
Code
Blame
Raw

Senior Developer Explanation

I'll explain this code as a senior developer would, focusing on the why behind decisions.

I'll analyze the code using native tools:

- **Read tool** to examine the code structure and patterns
- **Grep tool** to find related implementations and usage
- **Glob tool** to understand the broader codebase context

Technical Context:

- Why this approach was chosen over alternatives
- Trade-offs and architectural decisions made

The framework works by replacing open-ended conversations with structured command definitions.

And to use it, you execute a command that already contains the precise instructions, context requirements, and output specifications.

But, you will ask me,

“Joe, how about subagents? What’s the difference? “

I recently covered Claude Code subagents that work the same way.

> ***From my testing, ‘partially’,*** *these are well-packaged, tested subagents, and also have some additional features that go beyond the subagent functionality***.**
> ***This came before the official Claude Code subagents feature.***

The documentation makes this clear when it mentions

“Sub-agent orchestration for specialized analysis” describes commands that create ***“Security analysis agent,” “Performance optimization agent,”*** and ***“Architecture review agent.”***

So partly this framework functions like subagents,

But its clear commands give more value to specific use cases.

Instead of writing such a prompt

```
You are a senior security engineer. Analyze this codebase for vulnerabilities
Focus on:
- SQL injection risks
- XSS vulnerabilities
- Authentication flaws
- Dependency vulnerabilities
Use the grep tool to search for patterns, read suspicious files, and provide
a detailed report with remediation steps.
```

With this framework, you can use this command :

```
/security-scan
```

Both create the same type of specialized agent, But CCPlugins gives you a pre-written, tested version that includes specific instructions for Claude Code CLI’s tools, proper output formatting, session state management, and safety measures.

> ***The value is in the curation of tested ready to use tools, and subagents*** — *What do you think?*

So,

What are the other commands available?

CCPlugins Commands

Here’s what you get with CCPlugins — I’ve organized them by workflow to show you how they can fit into your development routine:

Development Workflow Commands

- `/cleanproject` - Removes debug artifacts with git safety (goodbye leftover console.logs)
- `/commit` - Smart conventional commits with analysis (never write commit messages again)
- `/format` - Auto-detects and applies your project's formatter
- `/scaffold feature-name` - Generates complete features from existing patterns
- `/test` - Runs tests with intelligent failure analysis
- `/implement url/path/feature` - Imports and adapts code from any source
- `/refactor` - Intelligent restructuring with validation phases

Code Quality & Security Commands

- `/review` - Multi-agent analysis covering security, performance, quality, and architecture
- `/security-scan` - Vulnerability analysis with remediation tracking
- `/predict-issues` - Proactive problem detection with timeline estimates

- `/remove-comments` - Cleans obvious comments while preserving valuable documentation
- `/fix-imports` - Repairs broken imports after refactoring
- `/find-todos` - Locates and organizes development tasks
- `/create-todos` - Adds contextual TODO comments based on analysis
- `/fix-todos` - Intelligently implements TODO fixes with full context

Advanced Analysis Commands

- `/understand` - Analyzes entire project architecture and patterns
- `/explain-like-senior` - Senior-level code explanations with context
- `/contributing` - Complete contribution readiness analysis
- `/make-it-pretty` - Improves readability without functional changes

Session & Project Management Commands

- `/session-start` - Begins documented sessions with CLAUDE.md integration
- `/session-end` - Summarizes and preserves session context
- `/docs` - Smart documentation management and updates
- `/todos-to-issues` - Converts code TODOs to GitHub issues
- `/undo` - Safe rollback with git checkpoint restore

Let’s Test It

Now, I’m going to walk you through installing and testing CCPlugins, so you can see how it works and decide if it’s worth adding to your toolkit.

Installation

The installation is straightforward. CCPlugins supports both quick install and manual installation:

Mac/Linux Quick Install:

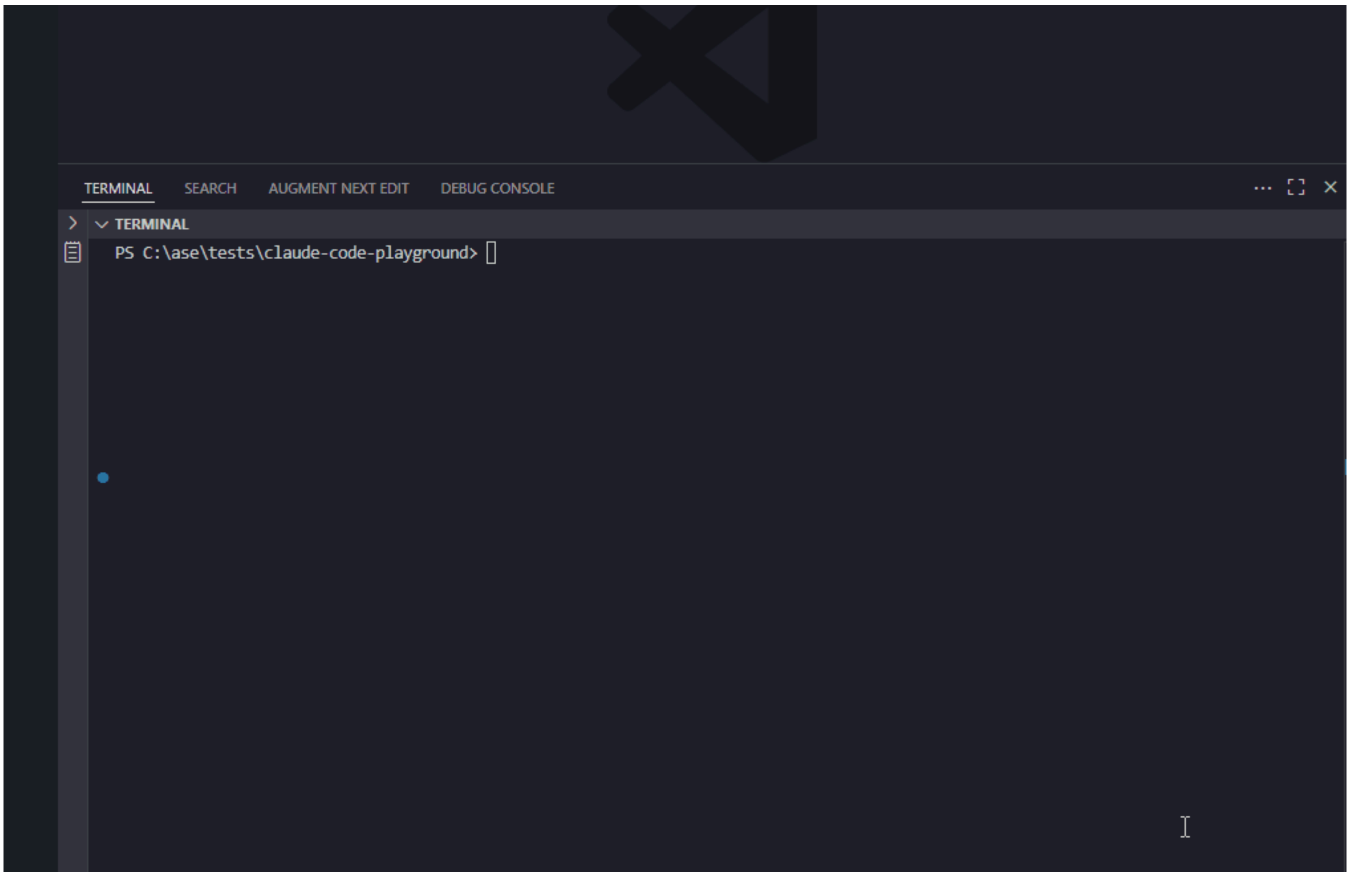
```
curl -sSL https://raw.githubusercontent.com/brennercruvinel/CCPlugins/main/ir
```

Windows/Cross-platform:

```
python install.py
```

Manual Installation:

```
git clone https://github.com/brennercruvinel/CCPlugins.git
cd CCPlugins
python install.py
```



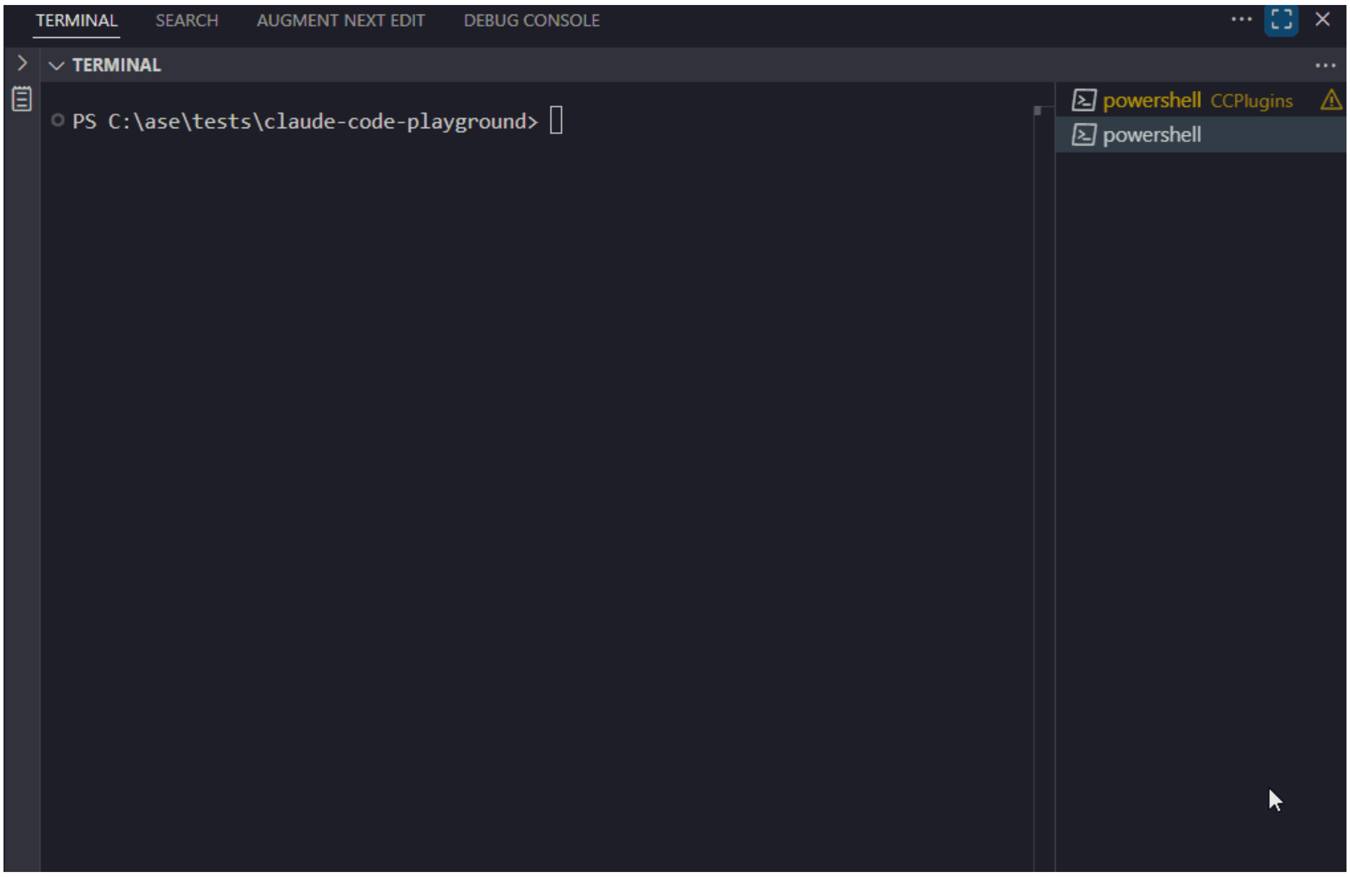
```
PS C:\ase\tests\claude-code-playground>
```

The installer places command definitions in `~/.claude/commands/` where Claude Code CLI can access them.

After installation, commands appear with a `(user)` tag to distinguish them from built-in commands.

After installation, these commands can now be accessed from Claude Code CLI.

Here is the demo :



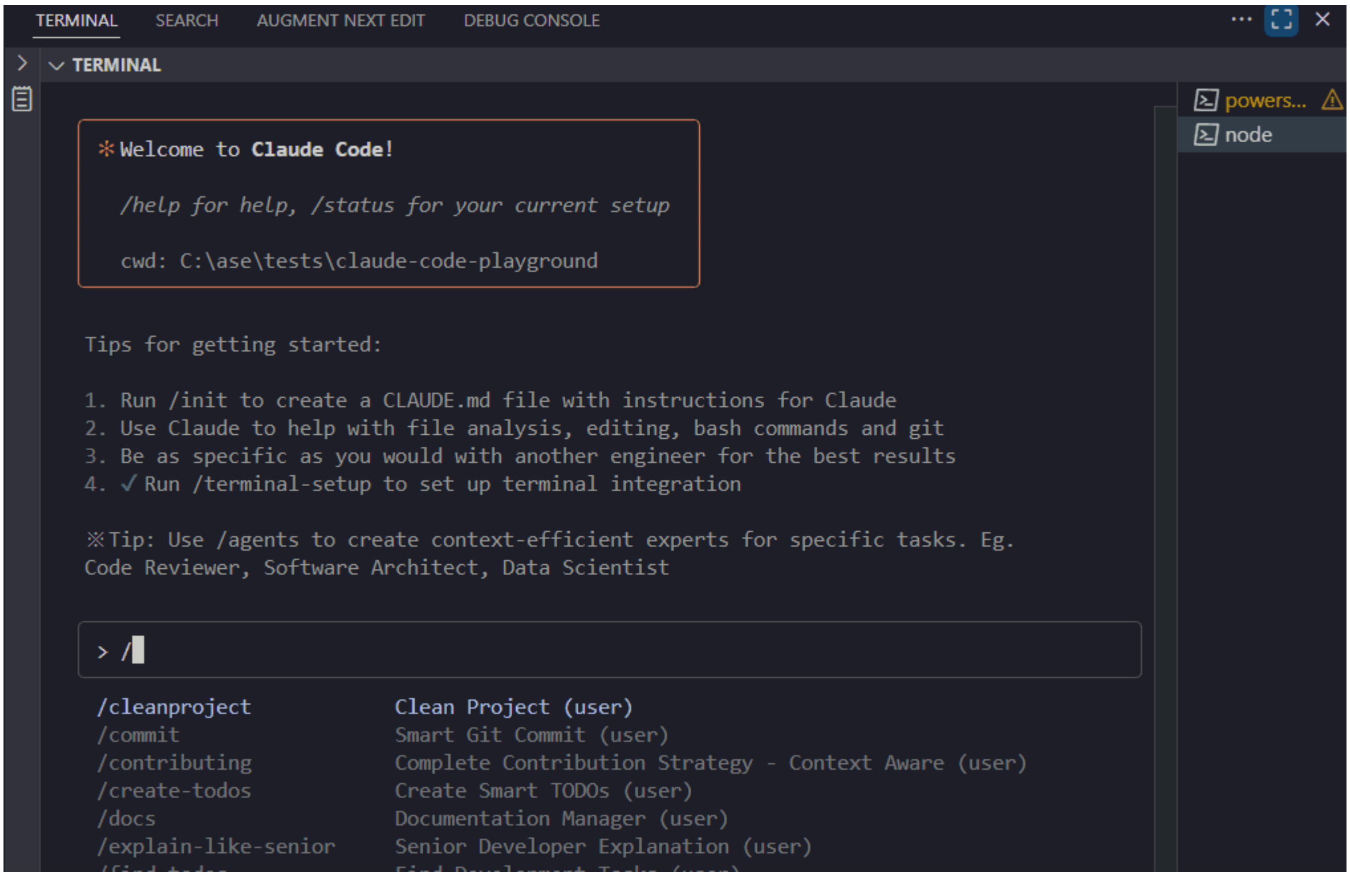
```
PS C:\ase\tests\claude-code-playground>
```

```
Usage:
 1. Open Claude Code CLI
 2. Type / to see available commands
 3. Use /cleanproject, /commit, /refactor, etc.
```

Testing Key Commands

Now let’s test some of the most valuable commands to see how they perform in real scenarios.

`/cleanproject` - Removing Development Artifacts



```
✻ Welcome to Claude Code!

  /help for help, /status for your current setup

  cwd: C:\ase\tests\claude-code-playground


Tips for getting started:

1. Run /init to create a CLAUDE.md file with instructions for Claude
2. Use Claude to help with file analysis, editing, bash commands and git
3. Be as specific as you would with another engineer for the best results
4. ✓ Run /terminal-setup to set up terminal integration

※Tip: Use /agents to create context-efficient experts for specific tasks. Eg.
Code Reviewer, Software Architect, Data Scientist

> /

 /cleanproject             Clean Project (user)
 /commit                   Smart Git Commit (user)
 /contributing             Complete Contribution Strategy - Context Aware (user)
 /create-todos             Create Smart TODOs (user)
 /docs                     Documentation Manager (user)
 /explain-like-senior      Senior Developer Explanation (user)
```

This command removes debug files, temporary artifacts, and backup files while preserving actual project code. Here's what it does:

Before cleanup:

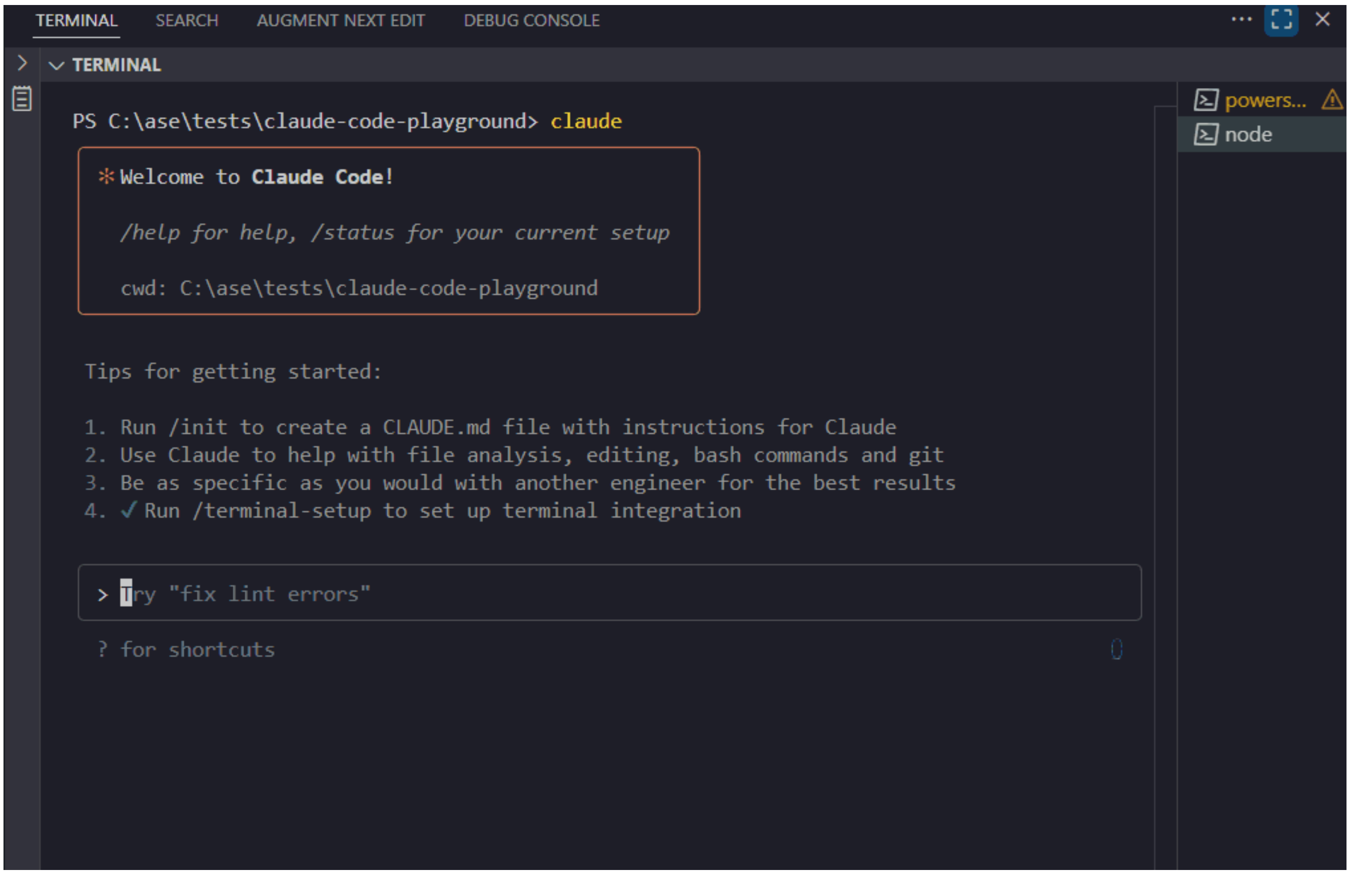
```
src/
├── UserService.js
├── UserService.test.js
├── UserService_backup.js     # Old version
├── debug.log                 # Debug output
├── test_temp.js              # Temporary test
└── notes.txt                 # Dev notes
```

After `/cleanproject`:

```
src/
├── UserService.js          # Clean production code
```

```
    └── UserService.test.js      # Actual tests preserved
```

`/security-scan` **- Vulnerability Analysis**



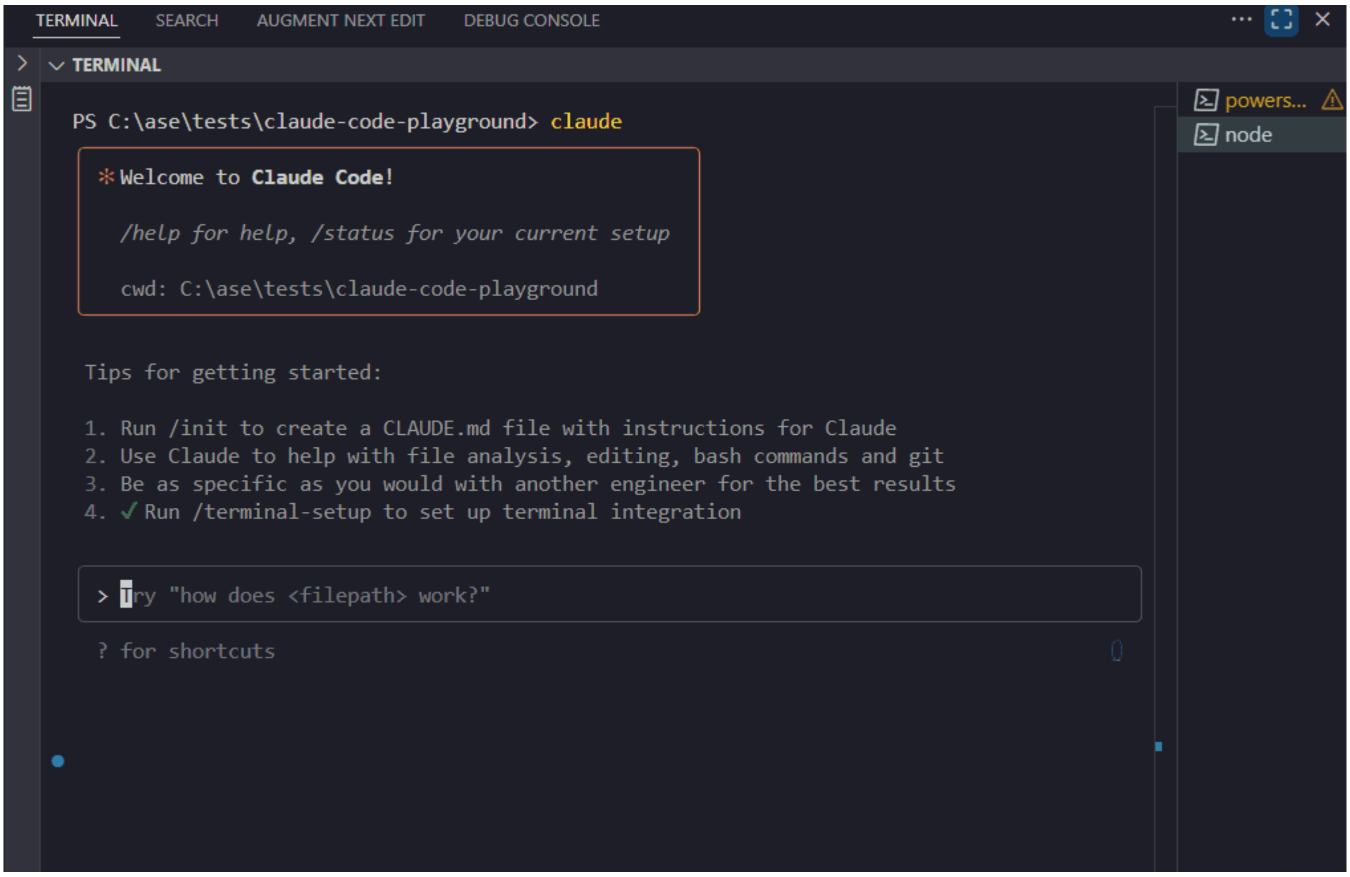
```
PS C:\ase\tests\claude-code-playground> claude
╭──────────────────────────────────────────────────╮
│ ✻ Welcome to Claude Code!                        │
│                                                  │
│   /help for help, /status for your current setup │
│                                                  │
│   cwd: C:\ase\tests\claude-code-playground       │
╰──────────────────────────────────────────────────╯

 Tips for getting started:

 1. Run /init to create a CLAUDE.md file with instructions for Claude
 2. Use Claude to help with file analysis, editing, bash commands and git
 3. Be as specific as you would with another engineer for the best results
 4. ✓ Run /terminal-setup to set up terminal integration

╭──────────────────────────────────────────────────────────────────────────╮
│ > Try "fix lint errors"                                                  │
╰──────────────────────────────────────────────────────────────────────────╯
  ? for shortcuts
```

This command creates a specialized security analysis agent that examines your codebase for common vulnerabilities. It performs:

- ***SQL injection pattern detection***
- ***XSS vulnerability identification***
- ***Authentication flaw analysis***
- ***Dependency security assessment***
- ***Configuration security review***

The scan creates detailed reports with specific remediation steps and maintains session state for tracking fixes.

`/refactor` - Intelligent Code Restructuring

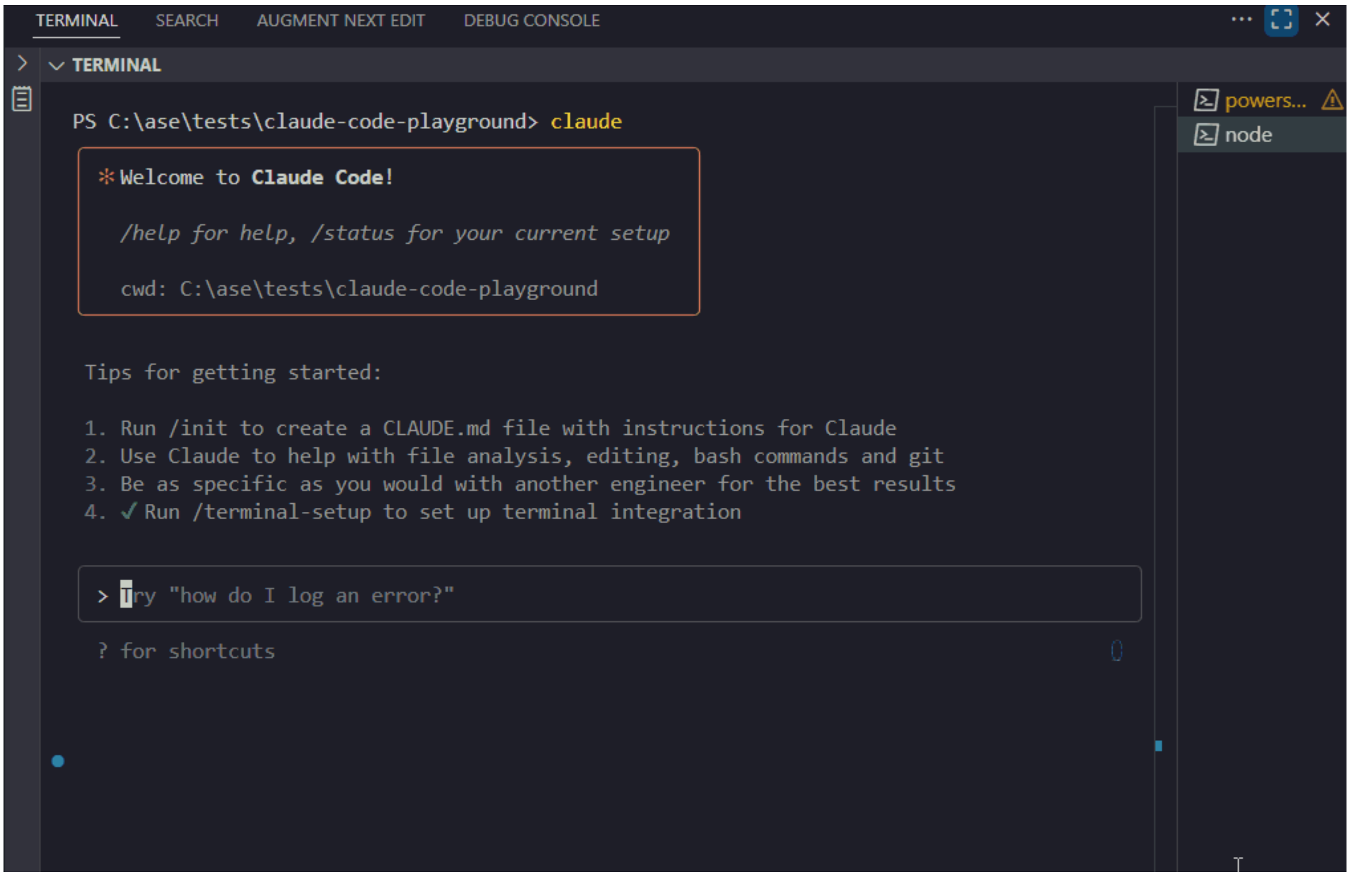


```
PS C:\ase\tests\claude-code-playground> claude

 * Welcome to Claude Code!

   /help for help, /status for your current setup

   cwd: C:\ase\tests\claude-code-playground

 Tips for getting started:

 1. Run /init to create a CLAUDE.md file with instructions for Claude
 2. Use Claude to help with file analysis, editing, bash commands and git
 3. Be as specific as you would with another engineer for the best results
 4. ✓ Run /terminal-setup to set up terminal integration

 > Try "how does <filepath> work?"

  ? for shortcuts
```

One of the most powerful commands, `/refactor` performs architectural analysis and code restructuring with validation phases. It:

- Analyzes current code patterns
- Plans refactoring strategy
- Implements changes incrementally
- Validates completeness with `/refactor validate`
- Maintains state across sessions

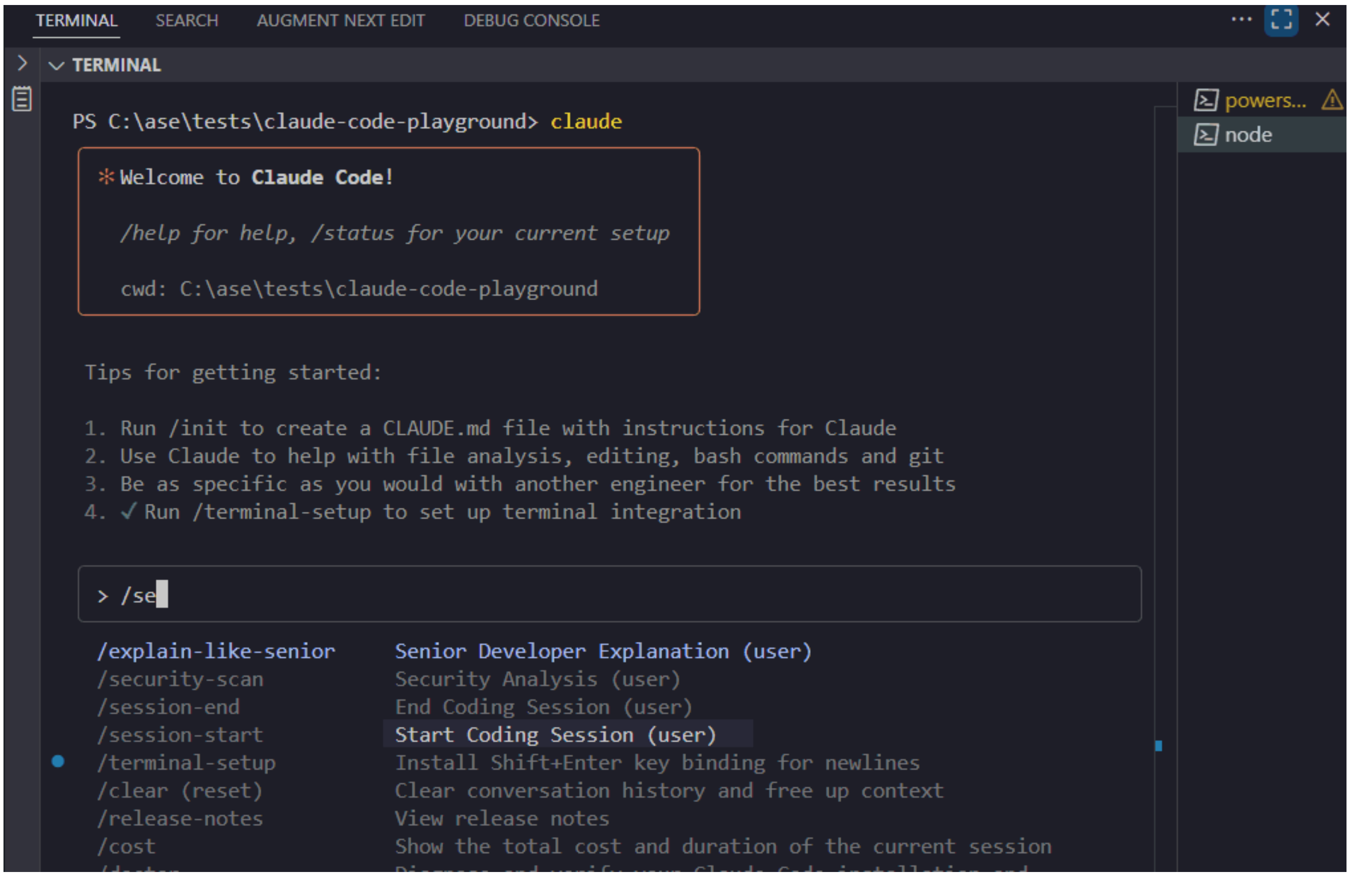
`/commit` - Smart Git Commits



```
PS C:\ase\tests\claude-code-playground> claude
 ✻ Welcome to Claude Code!

   /help for help, /status for your current setup

   cwd: C:\ase\tests\claude-code-playground

 Tips for getting started:

 1. Run /init to create a CLAUDE.md file with instructions for Claude
 2. Use Claude to help with file analysis, editing, bash commands and git
 3. Be as specific as you would with another engineer for the best results
 4. ✓ Run /terminal-setup to set up terminal integration

 > Try "how do I log an error?"

  ? for shortcuts
```

Instead of writing commit messages manually, `/commit` analyzes your changes and generates conventional commit messages with proper formatting and scope identification.

Session Management and State Persistence



```
PS C:\ase\tests\claude-code-playground> claude
╭──────────────────────────────────────────────────────╮
│ ✻ Welcome to Claude Code!                            │
│                                                      │
│   /help for help, /status for your current setup     │
│                                                      │
│   cwd: C:\ase\tests\claude-code-playground           │
╰──────────────────────────────────────────────────────╯

 Tips for getting started:

 1. Run /init to create a CLAUDE.md file with instructions for Claude
 2. Use Claude to help with file analysis, editing, bash commands and git
 3. Be as specific as you would with another engineer for the best results
 4. ✓ Run /terminal-setup to set up terminal integration

> /se

  /explain-like-senior     Senior Developer Explanation (user)
  /security-scan           Security Analysis (user)
  /session-end             End Coding Session (user)
  /session-start           Start Coding Session (user)
  /terminal-setup          Install Shift+Enter key binding for newlines
  /clear (reset)           Clear conversation history and free up context
  /release-notes           View release notes
  /cost                    Show the total cost and duration of the current session
```

A notable feature is how CCPlugins handles session continuity.

Complex commands like `/implement`, `/refactor`, and `/security-scan` Create folders in your project root to maintain state:

```
project/
├── refactor/
│   ├── plan.md            # Refactoring roadmap
│   └── state.json         # Completed transformations
├── security-scan/
│   ├── plan.md            # Vulnerabilities and fixes
│   └── state.json         # Remediation progress
└── implement/
    ├── plan.md            # Implementation progress
    └── state.json         # Session decisions
```

This allows you to resume complex operations across multiple Claude Code sessions without losing context.

Performance Impact

According to the documentation, CCPlugins provides significant time savings:

- ***Security analysis: 45–60 minutes → 3–5 minutes***
- ***Architecture review: 30–45 minutes → 5–8 minutes***
- ***Feature scaffolding: 25–40 minutes → 2–3 minutes***
- ***Code cleanup: 20–30 minutes → 1 minute***
- ***Import fixing: 15–25 minutes → 1–2 minutes***

The total time savings amounts to 4–5 hours per week with professional-grade analysis quality.

Is CCPlugins Worth Using?

CCPlugins solves the friction of repeatedly writing effective prompts for common development tasks.

Instead of crafting detailed subagent instructions each time, you get optimized commands that produce consistent results.

The framework is valuable for:

- ***Developers who often perform security scans, code reviews, and refactoring***
- ***Teams that want standardized development workflows***
- ***Projects requiring consistent code quality and documentation standards***

For developers already using Claude Code, CCPlugins is a good addition to their toolkit.

Final Thoughts

CCPlugins has perfected reusable, professional-grade subagent definitions and improved Claude Code workflow tools that are pretty useful.

The session state persistence, operations like, `/refactor` and `/security-scan` enable enterprise-grade workflow.

The Claude Code ecosystem is growing very fast, which is good for the community, as we get more tools and frameworks that solve real-world coding problems.

> ***Have you tried CCplugins? What was your experience?***

Claude Course Course

> ***I am currently working on an advanced Claude Code course to demonstrate how to build workflows that coordinate multiple agents for complex development tasks.***

It will take what you have learned from this article to the next level of complete automation.

The Claude Code course explores subagents, hooks, advanced workflows, and productivity techniques that many developers may not discover.

This course will cover:

- *Advanced subagent patterns and workflows*
- *Production-ready hook configurations*
- *MCP server integrations for external tools*
- *Team collaboration strategies*
- *Enterprise deployment patterns*
- *Real-world case studies from my consulting work*

If you're interested in getting notified when the Claude Code course launches, **click here to join the early access list →**

> I’ll share exclusive previews, early access pricing, and bonus materials with people on the list.