MUDITEK

The Claude Code 45-Tip Power User Playbook

From basics to running 3 AI agents in parallel from your phone.

BY **GHILES MOUSSAOUI** — MUDITEK

Why This Playbook

Most people use Claude Code at 10% of its power. This playbook walks through all 46 tips (Tip 0 through Tip 45) in order, with full verbatim content and every code example ready to paste. Curated by someone who runs their entire business through Claude Code skills and subagents.

TIP 00

Customize Your Status Line

You can customize the status line at the bottom of Claude Code to show useful info. I set mine up to show the model, current directory, git branch (if any), uncommitted file count, sync status with origin, and a visual progress bar for token usage. It also shows a second line with my last message so I can see what the conversation was about:

status line preview

```
Opus 4.5 | my-project | main (scripts/context-bar.sh uncommitted, synced
12m ago) | ██░░░░░░░░ 18% of 200k tokens
Last message: This is good. I don't think we need to change the
documentation as long as we don't say that the default color is orange
el...
```

This is especially helpful for keeping an eye on your context usage and remembering what you were working on. The script also supports 10 color themes (orange, blue, teal, green, lavender, rose, gold, slate, cyan, or gray).

To set this up, you can use this sample script and check the setup instructions.

Why it matters: Knowing your token usage at a glance prevents surprise compaction. Knowing your branch prevents wrong-branch commits.

TIP 01

Learn a Few Essential Slash Commands

There are a bunch of built-in slash commands (type / to see them all). Here are a few worth knowing:

/usage

Check your rate limits:

/usage output

```
Current session
██████████                                          19% used
Resets 12:59am (America/Vancouver)

Current week (all models)
█████████████████████                               43% used
Resets Feb 3 at 1:59pm (America/Vancouver)

Current week (Sonnet only)
███████████████████                                 39% used
Resets 8:59am (America/Vancouver)
```

If you want to watch your usage closely, keep it open in a tab and use Tab then Shift+Tab or ← then → to refresh.

/chrome

Toggle Claude's native browser integration:

/chrome

```
> /chrome
Chrome integration enabled
```

/mcp

Manage MCP (Model Context Protocol) servers:

/mcp

```
Manage MCP servers
1 server

> 1. playwright  connected - Enter to view details

MCP Config locations (by scope):
 - User config (available in all your projects):
   - /Users/yk/.claude.json
```

/stats

View your usage statistics with a GitHub-style activity graph:

```
/stats output

    Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan
    ........................................... .  .
Mon ........................................... .  .
    ........................................... .  .
Wed .......................................... .  .
    .......................................... .  .
Fri .......................................... .  .
    .......................................... .  .

    Less . . . . More

Favorite model: Opus 4.5        Total tokens: 17.6m

Sessions: 4.1k                  Longest session: 20h 40m 45s
Active days: 79/80              Longest streak: 75 days
Most active day: Jan 26         Current streak: 74 days

You've used ~24x more tokens than War and Peace
```

/clear

Clear the conversation and start fresh.

Why it matters: These five commands give you visibility into usage, connected tools, session history, and a clean slate. Know your limits before you hit them.

TIP 02

Talk to Claude Code with Your Voice

I found that you can communicate much faster with your voice than typing with your hands. Using a voice transcription system on your local machine is really helpful for this.
On my Mac, I've tried a few different options:

- superwhisper
- MacWhisper
- Super Voice Assistant (open source, supports Parakeet v2/v3)

You can get more accuracy by using a hosted service, but I found that a local model is strong enough for this purpose. Even when there are mistakes or typos in the transcription, Claude is smart enough to understand what you're trying to say. Sometimes you need to say certain things extra clearly, but overall local models work well enough.

For example, in this screenshot you can see that Claude was able to interpret mistranscribed words like "ExcelElanishMark" and "advast" correctly as "exclamation mark" and "Advanced".

I think the best way to think about this is like you're trying to communicate with your friend. Of course, you can communicate through texts. That might be easier for some people, or emails, right? That's totally fine. That's what most people seem to do with Claude Code. But if you want to communicate faster, why wouldn't you get on a quick phone call? You can just send voice messages. You don't need to literally have a phone call with Claude Code. Just send a bunch of voice messages. It's faster, at least for me, as someone who's practiced the art of speaking a lot over the past number of years. But I think for a majority of people, it's going to be faster too.

A common objection is "what if you're in a room with other people?" I just whisper using earphones - I personally like Apple EarPods (not AirPods). They're affordable, high quality enough, and you just whisper into them quietly. I've done it in front of other people and it works well. In offices, people talk anyway - instead of talking to coworkers, you're talking quietly to your voice transcription system. I don't think there's any problem with that. This method works so well that it even works on a plane. It's loud enough that other people won't hear you, but if you speak close enough to the mic, your local model can still understand what you're saying. (In fact, I'm writing this very paragraph using that method on a flight.)

Update: Claude Code now has a built-in voice mode. I tested it and it works well, but I still personally use a local model because I find it faster.

Why it matters: Voice is 3-5x faster than typing for most people, and it lets you interact with Claude Code while walking, cooking, or commuting.

> **MY TAKE**
>
> *I run my entire business through voice to Claude Code. From my phone, walking, on planes. This single tip 10x'd my output.*

TIP 03

Break Down Large Problems into Smaller Ones

This is one of the most important concepts to master. It's exactly the same as traditional software engineering - the best software engineers already know how to do this, and it applies to Claude Code too.
If you find that Claude Code isn't able to one-shot a difficult problem or coding task, ask it to break it down into multiple smaller issues. See if it can solve an individual part of that problem. If it's still too hard, see if it can solve an even smaller sub-problem. Keep going until everything is solvable.

Essentially, instead of going from A to B, you can go from A to A1 to A2 to A3, then to B.

A good example of this is when I was building my own voice transcription system. I needed to build a system that could let the user select and download a model, take keyboard shortcuts, start transcribing, put the transcribed text at the user's cursor, and wrap all of this in a nice UI. That's a lot. So I broke it down into smaller tasks. First, I created an executable that would just download a model, nothing else. Then I created another one that would just record voice, nothing else. Then another one that would just transcribe pre-recorded audio. I completed them one by one like that before combining them at the end.

Highly related to this: your problem-solving skills and software engineering skills are still highly relevant in the world of agentic coding and Claude Code. It's able to solve a lot of problems on its own, but when you apply your general problem-solving and software engineering skills to it, it becomes a lot more powerful.

Why it matters: Decomposition is the master skill. A problem Claude can't solve in one shot often becomes trivial when broken into 3-4 sub-problems.

TIP 04

Using Git and GitHub CLI Like a Pro

Just ask Claude to handle your Git and GitHub CLI tasks. This includes committing (so you don't have to write commit messages manually), branching, pulling, and pushing.
I personally allow pull automatically but not push, because push is riskier - it doesn't contaminate the origin if something goes wrong with a pull.

For GitHub CLI (`gh`), there's a lot you can do. One thing I started doing more after using Claude Code is creating draft PRs. This lets Claude Code handle the PR creation process with low risk - you can review everything before marking it ready for review.

And it turns out, `gh` is pretty powerful. You can even send arbitrary GraphQL queries through it. For example, you can even find the exact times at which GitHub PR descriptions were edited:

gh GraphQL

```
Bash(gh api graphql -f query='
      query {
        repository(owner: "...", name: "...") {
          pullRequest(number: ...) {
            userContentEdits(first: 100) {
              nodes { editedAt editor { login } }
            }
          }
        }
      }')

Here's the full edit history for your PR description:

  | #  | Edited At (UTC)      | Editor |
  |----|---------------------|--------|
  | 1  | 2025-12-01 00:08:34 | ykdojo |
  | 2  | 2025-12-01 15:57:21 | ykdojo |
  | 3  | 2025-12-01 16:24:33 | ykdojo |
  | 4  | 2025-12-01 16:27:00 | ykdojo |
  | 5  | 2025-12-04 00:40:02 | ykdojo |
  ...
```

Disable commit/PR attribution

By default, Claude Code adds a `Co-Authored-By` trailer to commits and an attribution footer to PRs. You can disable both by adding this to `~/.claude/settings.json`:

~/.claude/settings.json

```json
{
  "attribution": {
    "commit": "",
    "pr": ""
  }
}
```

Setting both to empty strings removes the attribution entirely. This replaces the older `includeCoAuthoredBy` setting, which is now deprecated.

Why it matters: Never write a commit message manually again. Draft PRs give you a review checkpoint before anything goes public.

TIP 05

AI Context is Like Milk; It's Best Served Fresh and Condensed!

When you start a new conversation with Claude Code, it performs the best because it doesn't have all the added complexity of having to process the previous context from earlier parts of the conversation. But as you talk to it longer and longer, the context gets longer and the performance tends to go down.
So it's best to start a new conversation for every new topic, or if the performance starts to go down.

Why it matters: Stale, bloated context is the single biggest reason Claude gives bad or confused answers. Fresh sessions fix this instantly.

> **MY TAKE**
>
> *Every new task gets a new session. No exceptions. Stale context is the #1 reason Claude gives bad answers.*

TIP 06

Getting Output Out of Your Terminal

Sometimes you want to copy and paste Claude Code's output, but copying directly from the terminal isn't always clean. Here are a few ways to get content out more easily:

- **/copy command**: The simplest option - just type `/copy` to copy Claude's last response to your clipboard as markdown
- **Clipboard directly**: On Mac or Linux, ask Claude to use `pbcopy` to send output straight to your clipboard
- **Write to a file**: Have Claude put the content in a file, then ask it to open it in VS Code (or your favorite editor) so you can copy from there. You can also specify a line number, so you can ask Claude to open the specific line it just edited. For markdown files, once it's open in VS Code, you can use Cmd+Shift+P (or Ctrl+Shift+P on Linux/Windows) and select "Markdown: Open Preview" to see the rendered version
- **Opening URLs**: If there's a URL you want to examine yourself, ask Claude to open it in your browser. On Mac, you can ask it to use the `open` command, but in general asking to open in your favorite browser should work on any platform
- **GitHub Desktop**: You can ask Claude to open the current repo in GitHub Desktop. This is particularly useful when it's working in a non-root directory - for example, if you asked it to create a git worktree in a different directory and you haven't opened Claude Code from there yet

You can combine some of these together too. For example, if you want to edit a GitHub PR description, instead of having Claude edit it directly (which it might mess up), you can have it copy the content into a local file first. Let it edit that, check the result yourself, and once it looks good, have it copy and paste it back into the GitHub PR. That works really well. Or if you want to do that yourself, you can just ask it to open it in VS Code or give it to you via pbcopy so you can copy and paste it manually.

Of course, you can run these commands yourself, but if you find yourself doing it repetitively, it's helpful to let Claude run them for you.

Why it matters: Getting output out of the terminal cleanly is the bottleneck between Claude's work and your other tools. These methods eliminate that friction.

TIP 07

Set Up Terminal Aliases for Quick Access

Since I use the terminal more because of Claude Code, I found it helpful to set up short aliases so I can launch things quickly. Here are the ones I use:

- `c` for Claude Code (this is the one I use the most)
- `ch` for Claude Code with Chrome integration
- `gb` for GitHub Desktop
- `co` for VS Code
- `q` for going to the project directory where I have most projects. From there I can manually cd into an individual folder to work on that project, or I can just launch Claude Code with `c` to let it basically have access to any project it needs to access.

To set these up, add lines like this to your shell config file (`~/.zshrc` or `~/.bashrc`):

```
~/.zshrc

alias c='claude'
alias ch='claude --chrome'
alias gb='github'
alias co='code'
alias q='cd ~/Desktop/projects'
```

Once you have these aliases, you can combine them with flags: `c -c` continues your last conversation, and `c -r` shows a list of recent conversations to resume. These work with `ch` too (`ch -c`, `ch -r`) for Chrome sessions.

Why it matters: Typing `c` instead of `claude` saves keystrokes, but more importantly it lowers the activation energy to start a session.

TIP 08

Proactively Compact Your Context

There's a `/compact` command in Claude Code that summarizes your conversation to free up context space. Automatic compaction also happens when the full available context is filled. The total available context window for Opus 4.5 is currently 200k, and 45k of that is reserved for automatic compaction. About 10% of the total 200k is automatically filled with the system prompt, tools, memory, and dynamic context. But I found that it's better to proactively do it and manually tune it. I turned off auto-compact with `/config` so I have more context available for the main conversation and more control over when and how compaction happens.
The way I do this is to ask Claude to write a handoff document before starting fresh. Something like:

```
handoff pattern
```

```
> Put the rest of the plan in the system-prompt-extraction folder as
HANDOFF.md.
  Explain what you have tried, what worked, what didn't work, so that the
next
  agent with fresh context is able to just load that file and nothing else
to
  get started on this task and finish it up.
```

Claude will create a file summarizing the current state of work:

```
handoff output
```

```
Write(experiments/system-prompt-extraction/HANDOFF.md)
  Wrote 129 lines to experiments/system-prompt-extraction/HANDOFF.md
     # System Prompt Slimming - Handoff Document
     ## Goal
     Reduce Claude Code's system prompt by ~45% (currently at 11%, need
~34% more).
     ## Current Progress
     ### What's Been Done
     - **Backup/restore system**: backup-cli.sh and restore-cli.sh with
SHA256 verification
     - **Patch system**: patch-cli.js that restores from backup then
applies patches
     ...
```

After Claude writes it, review it quickly. If something's missing, ask for edits:

```
review
```

```
> Did you add a note about iteratively testing instead of trying to do
  everything all at once?
```

Then start a fresh conversation. For the fresh agent, you can just give the path of the file and nothing else like this, and it should work just fine:

```
fresh session
```

```
> experiments/system-prompt-extraction/HANDOFF.md
```

In subsequent conversations, you can ask the agent to update the document for the next agent.

I've also created a `/handoff` slash command that automates this - it checks for an existing HANDOFF.md, reads it if present, then creates or updates it with the goal, progress, what worked, what didn't, and next steps. You can find it in the skills folder, or install it via the dx plugin.

Alternative: Use plan mode

Another option is to use plan mode. Enter it with `/plan` or Shift+Tab. Ask Claude to gather all the relevant context and create a comprehensive plan for the next agent:

plan mode

```
> I just enabled plan mode. Bring over all of the context that you need for
  the next agent. The next agent will not have any other context, so you'll
  need to be pretty comprehensive.

Would you like to proceed?

> 1. Yes, clear context and auto-accept edits (shift+tab)
  2. Yes, auto-accept edits
  3. Yes, manually approve edits
  4. Type here to tell Claude what to change
```

Option 1 clears the previous context and starts fresh with the plan. The new Claude instance sees only the plan, so it can focus without the baggage of the old conversation. It also gets a link to the old transcript file in case it needs to look up specific details.

Why it matters: Manual handoffs preserve the important context while dumping the noise. The next agent starts sharp instead of wading through a 200k history of dead ends.

> **MY TAKE**
>
> *I never let context fill up on its own. HANDOFF.md is my most-used pattern. Write it, review it, start fresh. Clean handoffs are the difference between a productive agent and a confused one.*

TIP 09

Complete the Write-Test Cycle for Autonomous Tasks

If you want Claude Code to run something autonomously, like `git bisect`, you need to give it a way to verify results. The key is completing the write-test cycle: write code, run it, check the output, and repeat.

For example, let's say you're working on Claude Code itself and you notice `/compact` stopped working and started throwing a 400 error. A classic tool to find the exact commit that caused this is `git bisect`. The nice thing is you can let Claude Code run bisect on itself, but it needs a way to test each commit.

For tasks that involve interactive terminals like Claude Code, you can use tmux. The pattern is:

1. Start a tmux session
2. Send commands to it
3. Capture the output
4. Verify it's what you expect

Here's a simple example of testing if `/context` works:

tmux test pattern

```
tmux kill-session -t test-session 2>/dev/null
tmux new-session -d -s test-session
tmux send-keys -t test-session 'claude' Enter
sleep 2
tmux send-keys -t test-session '/context' Enter
sleep 1
tmux capture-pane -t test-session -p
```

Once you have a test like this, Claude Code can run `git bisect` and automatically test each commit until it finds the one that broke things.

This is also an example of why your software engineering skills still matter. If you're a software engineer, you probably know about tools like `git bisect`. That knowledge is still really valuable when working with AI - you just apply it in new ways.

Another example is simply writing tests. After you let Claude Code write some code, if you want to test it, you can just let it write tests for itself too. And let it run on its own and fix things if it can. Of course, it doesn't always go in the right direction and you need to supervise it sometimes, but it's able to do a surprising amount of coding tasks on its own.

Creative testing strategies

Sometimes you need to be creative with how you complete the write-test cycle. For example, if you're building a web app, you could use Playwright MCP, Chrome DevTools MCP, or Claude's native browser integration (through `/chrome`). I haven't tried Chrome DevTools yet, but I've tried Playwright and Claude's native integration. Overall, Playwright generally works better. It does use a lot of context, but the 200k context window is normally enough for a single task or a few smaller tasks.

The main difference between these two seems to be that Playwright focuses on the accessibility tree (structured data about page elements) rather than taking screenshots. It does have the ability to take screenshots, but it doesn't normally use them to take actions. On the other hand, Claude's native browser integration focuses more on taking screenshots and clicking on elements by specific coordinates. It can click on random things sometimes, and the whole process can be slow.

This might improve over time, but by default I would go with Playwright for most tasks that aren't visually intensive. I'd only use Claude's native browser integration if I need to use a logged-in state without having to provide credentials (since it runs in your own browser profile), or if it specifically needs to click on things visually using their coordinates.

This is why I disable Claude's native browser integration by default and use it through the `ch` shortcut I defined previously. That way Playwright handles most browser tasks, and I only enable Claude's native integration when I specifically need it.

Additionally, you can ask it to use accessibility tree refs instead of coordinates. Here's what I put in my CLAUDE.md for this:

CLAUDE.md for Chrome

```
# Claude for Chrome

- Use `read_page` to get element refs from the accessibility tree
- Use `find` to locate elements by description
- Click/interact using `ref`, not coordinates
- NEVER take screenshots unless explicitly requested by the user
```

In my personal experience, I've also had a situation where I was working on a Python library (Daft) and needed to test a version I built locally on Google Colab. The trouble is it's hard to build a Python library with a Rust backend on Google Colab - it doesn't seem to work that well. So I needed to actually build a wheel locally and then upload it manually so that I could run it on Google Colab. I also tried monkey patching, which worked well in the short term before I had to wait for the whole wheel to build locally. I came up with these testing strategies and executed them by going back and forth with Claude Code.

Another situation I encountered is I needed to test something on Windows but I'm not running a Windows machine. My CI tests on the same repo were failing because we had some issues with Rust on Windows, and I had no way of testing locally. So I needed to create a draft PR with all the changes, and another draft PR with the same changes plus enabling Windows CI runs on non-main branches. I instructed Claude Code to do all of that, and then I tested the CI directly in that new branch.

Why it matters: Without a test loop, autonomous tasks are blind. With one, Claude can run git bisect, fix CI, or iterate on code without you approving every step.

MY TAKE

The tmux pattern is what separates vibe-coding from real agentic workflows. Once Claude can test its own output, you can walk away and come back to results.

TIP 10

Cmd+A and Ctrl+A Are Your Friends

I've been saying this for a few years now: Cmd+A and Ctrl+A are friends in the world of AI. This applies to Claude Code too.
Sometimes you want to give Claude Code a URL, but it can't access it directly. Maybe it's a private page (not sensitive data, just not publicly accessible), or something like a Reddit post that Claude Code has trouble fetching. In those cases, you can just select all the content you see (Cmd+A on Mac, Ctrl+A on other platforms), copy it, and paste it directly into Claude Code. It's a pretty powerful method.

This works great for terminal output too. When I have output from Claude Code itself or any other CLI application, I can use the same trick: select all, copy, and paste it back to CC. Pretty helpful.

Some pages don't lend themselves well to select all by default - but there are tricks to get them into a better state first. For example, with Gmail threads, click Print All to get the print preview (but cancel the actual print). That page shows all emails in the thread expanded, so you can Cmd+A the entire conversation cleanly. For asking questions about a YouTube video or summarizing it, you can click "Show transcript" on a YouTube video and then do Cmd+A or Ctrl+A.

This applies to any AI, not just Claude Code.

Why it matters: When Claude can't access a URL, Cmd+A is the fastest workaround. No MCP setup, no browser integration, just copy-paste.

TIP 11

Use Gemini CLI as a Fallback for Blocked Sites

Claude Code's WebFetch tool can't access certain sites, like Reddit. But you can work around this by creating a skill that tells Claude to use Gemini CLI as a fallback. Gemini has web access and can fetch content from sites that Claude can't reach directly.

This uses the same tmux pattern from Tip 9 - start a session, send commands, capture output. The skill file goes in `~/.claude/skills/reddit-fetch/SKILL.md`. See skills/reddit-fetch/SKILL.md for the full content.

Skills are more token-efficient because Claude Code only loads them when needed. If you want something simpler, you can put a condensed version in `~/.claude/CLAUDE.md` instead, but that gets loaded into every conversation whether you need it or not.

I tested this by asking Claude Code to check how Claude Code skills are regarded on Reddit - a bit meta. It goes back and forth with Gemini for a while, so it's not fast, but the report quality was surprisingly good. Obviously, you'll need to have Gemini CLI installed for this to work. You can also install this skill via the dx plugin.

Why it matters: Reddit, paywalled sites, and other blocked URLs become accessible through Gemini as a proxy. One skill unlocks a whole category of previously inaccessible content.

TIP 12

Invest in Your Own Workflow

Personally, I've created my own voice transcription app from scratch with Swift. I created my own custom status line from scratch using Claude Code, this one with bash. And I created my own system for simplifying the system prompt in Claude Code's minified JavaScript file.
But you don't have to go overboard like that. Just taking care of your own CLAUDE.md, making sure it's as concise as possible while being able to help you achieve your goals - stuff like that is helpful. And of course, learning these tips, learning these tools, and some of the most important features.

All of these are investments in the tools you use to build whatever you want to build. I think it's important to spend at least a little bit of time on that.

Why it matters: Small workflow investments compound daily. A 30-minute setup saves 5 minutes per session across hundreds of sessions.

TIP 13

Search Through Your Conversation History

You can ask Claude Code about your past conversations, and it'll help you find and search through them. Your conversation history is stored locally in `~/.claude/projects/`, with folder names based on the project path (slashes become dashes).

For example, conversations for a project at `/Users/yk/Desktop/projects/claude-code-tips` would be stored in:

```
conversation history path

~/.claude/projects/-Users-yk-Desktop-projects-claude-code-tips/
```

Each conversation is a `.jsonl` file. You can search through them with basic bash commands:

```
search history

# Find all conversations mentioning "Reddit"
grep -l -i "reddit" ~/.claude/projects/-Users-yk-Desktop-projects-*/*.jsonl

# Find today's conversations about a topic
find ~/.claude/projects/-Users-yk-Desktop-projects-*/*.jsonl -mtime 0 -exec
grep -l -i "keyword" {} \;

# Extract just the user messages from a conversation (requires jq)
cat ~/.claude/projects/.../conversation-id.jsonl | jq -r
'select(.type=="user") | .message.content'
```

Or just ask Claude Code directly: "What did we talk about regarding X today?" and it'll search through the history for you.

Why it matters: Past conversations contain solutions you've already found. Searching history is faster than re-solving the same problem.

TIP 14

Multitasking with Terminal Tabs

When running multiple Claude Code instances, staying organized is more important than any specific technical setup like Git worktrees. I recommend focusing on at most three or four tasks at a time.

My personal method is what I would call a "cascade" - whenever I start a new task, I just open a new tab on the right. Then I sweep left to right, left to right, going from oldest tasks to newest. The general direction stays consistent, except when I need to check on certain tasks, get notifications, etc.

Here's what my setup typically looks like:

In this example:

1. **Leftmost tab** - A persistent tab running my voice transcription system (always stays here)
2. **Second tab** - Setting up a Docker container
3. **Third tab** - Checking disk usage on my local machine
4. **Fourth tab** - Working on an engineering project
5. **Fifth tab (current)** - Writing this very tip

Why it matters: When you can see all your active tasks as tabs, you know your capacity at a glance. The cascade pattern prevents tab chaos.

TIP 15

Slim Down the System Prompt

Claude Code's system prompt and tool definitions take up about 19k tokens (~10% of your 200k context) before you even start working. I created a patch system that reduces this to about 9k tokens - saving around 10,000 tokens (~50% of the overhead).

Component	Before	After	Savings
System prompt	3.0k	1.8k	1,200 tokens
System tools	15.6k	7.4k	8,200 tokens
Total	**~19k**	**~9k**	**~10k tokens (~50%)**

The patches work by trimming verbose examples and redundant text from the minified CLI bundle while keeping all the essential instructions.

I've tested this extensively and it works well. It feels more raw - more powerful, but maybe a little less regulated, which makes sense because the system instruction is shorter. It feels more like a pro tool when you use it this way. I really enjoy starting with lower context because you have more room before it fills up, which gives you the option to continue conversations a bit longer. That's definitely the best part of this strategy.

Check out the system-prompt folder for the patch scripts and full details on what gets trimmed.

Why patching? Claude Code has flags that let you provide a simplified system prompt from a file (`--system-prompt` or `--system-prompt-file`), so that's another way to go about it. But for the tool descriptions, there's no official option to customize them. Patching the CLI bundle is the only way. Since my patch system handles everything in one unified approach, I'm keeping it this way for now. I might re-implement the system prompt portion using the flag in the future.

Supported installations: npm and native binary (macOS and Linux).

Important: If you want to keep your patched system prompt, disable auto-updates by adding this to `~/.claude/settings.json`:

```
~/.claude/settings.json

{
  "env": {
    "DISABLE_AUTOUPDATER": "1"
  }
}
```

This applies to all Claude Code sessions regardless of shell type (interactive, non-interactive, tmux). You can manually update later when you're ready to re-apply patches to a new version.

Lazy-load MCP tools

If you use MCP servers, their tool definitions are loaded into every conversation by default - even if you don't use them. This can add significant overhead, especially with multiple servers configured.

Enable lazy-loading so MCP tools are only loaded when needed:

```
lazy-load MCP

{
  "env": {
    "ENABLE_TOOL_SEARCH": "true"
  }
}
```

Add this to `~/.claude/settings.json`. Claude will search for and load MCP tools on-demand rather than having them all present from the start. As of version 2.1.7, this happens automatically when MCP tool descriptions exceed 10% of the context window.

Why it matters: 10k saved tokens means longer conversations before compaction kicks in. More room = more useful work per session.

> **MY TAKE**
>
> *Starting at 5% instead of 10% system overhead sounds small, but it compounds across every session. You get noticeably more productive turns before context starts degrading.*

TIP 16

Git Worktrees for Parallel Branch Work

If you're working on multiple things at the same time in the same project and you don't want them to get conflicted, Git worktrees are a great way to do that. You can just ask Claude Code to create a git worktree and start working on it there - you don't have to worry about the specific syntax.
The basic idea is that you can work on a different branch in a different directory. It's essentially a branch + a directory.

You can add this layer of Git worktrees on top of the cascade method I discussed in the multitasking tip.

What are git worktrees?

A git worktree is just like any other git branch, but with a new directory specifically assigned to it.

So if you're working on, let's say, the main branch and feature-branch-1, then without git worktrees, you can only work on them one at a time because your project folder can only be set to one branch at a time.

However, with a git worktree, you can keep working on the main branch (or any other branch for that matter) in the original project folder, and at the same time work on feature-branch-1 in a new folder.

worktrees

```
# Create a worktree for a feature branch
git worktree add ../my-project-feature-1 feature-1

# Each worktree gets its own Claude Code session
cd ../my-project-feature-1 && claude

# List all active worktrees
git worktree list

# Clean up when done
git worktree remove ../my-project-feature-1
```

Why it matters: 3 worktrees = 3 independent Claude Code agents = 3 features shipping simultaneously from a single repo.

MY TAKE

3 worktrees running in parallel = 3 features shipping simultaneously. This is how I run agents from my phone.

TIP 17

Manual Exponential Backoff for Long-Running Jobs

When waiting on long-running jobs like Docker builds or GitHub CI, you can ask Claude Code to do manual exponential backoff. Exponential backoff is a common technique in software engineering, but you can apply it here too. Ask Claude Code to check the status with increasing sleep intervals - one minute, then two minutes, then four minutes, and so on. It's not programmatically doing it in the traditional sense - the AI is doing it manually - but it works pretty well.
This way the agent can continuously check the status and let you know once it's done.

(For GitHub CI specifically, `gh run watch` exists but outputs many lines continuously, which wastes tokens. Manual exponential backoff with `gh run view <run-id> | grep <job-name>` is actually more token-efficient. This is also a general technique that works well even when you don't have a dedicated wait command handy.)

For example, if you have a Docker build running in the background, it keeps going until the job completes.

Why it matters: Token-efficient monitoring means Claude can watch a 30-minute CI run without burning through your context window.

TIP 18

Claude Code as a Writing Assistant

Claude Code is an excellent writing assistant and partner. The way I use it for writing is I first give it all the context about what I'm trying to write, and then I give it detailed instructions by speaking to it using my voice. That gives me the first draft. If it's not good enough, I try a few times.
Then I go through it line by line, pretty much. I say okay, let's take a look at it together. I like this line for these reasons. I feel like this line needs to move over there. This line needs to change in this particular way. I might ask about reference materials as well.

So it's this sort of back-and-forth process, maybe with the terminal on the left and your code editor on the right.

That tends to work really well.

Why it matters: Voice input + line-by-line review produces better writing than any one-shot prompt. The iterative loop is the key.

TIP 19

Markdown is the S**t

Typically when people write a new document, they might use something like Google Docs or maybe Notion. But now I honestly think the most efficient way to go about it is markdown. Markdown was already pretty good even before AI, but with Claude Code in particular, because it's so efficient as I mentioned with regards to writing, it makes the value of markdown higher in my opinion. Whenever you want to write a blog post or even a LinkedIn post, you can just talk to Claude Code, have it be saved as markdown, and then go from there.

A quick tip for this one: if you want to copy and paste markdown content into a platform that doesn't accept it easily, you can paste it into a fresh Notion file first, then copy from Notion into the other platform. Notion converts it to a format that other platforms can accept. If regular pasting doesn't work, try Command + Shift + V to paste without formatting.

Why it matters: Markdown is the native format of Claude Code. Writing in markdown means zero conversion friction between you, Claude, and your publishing platform.

TIP 20

Use Notion to Preserve Links When Pasting

It turns out the reverse also works. If you have text with links from other places, let's say from Slack, you can copy it. If you paste it directly into Claude Code, it doesn't show the links. But if you put it in a Notion document first, then copy from there, you get it in markdown, which of course Claude Code can read.

Why it matters: Links carry context. Losing them when pasting into Claude means it can't follow references. Notion as a middleman preserves that context.

TIP 21

Containers for Long-Running Risky Tasks

Regular sessions are more for methodical work where you control the permissions you give and review output more carefully. Containerized environments are great for `--dangerously-skip-permissions` sessions where you don't have to give permission for each little thing. You can just let it run on its own for a while.
This is useful for research or experimentation, things that take a long time and maybe could be risky. A good example is the Reddit research workflow from Tip 11, where the reddit-fetch skill goes back and forth with Gemini CLI through tmux. Running that unsupervised is risky on your main system, but in a container, if something goes wrong, it's contained.

Another example is how I created the system prompt patching scripts in this repo. When a new version of Claude Code comes out, I need to update the patches for the minified CLI bundle. Instead of running Claude Code with `--dangerously-skip-permissions` on my host machine (where it has access to everything), I run it in a container. Claude Code can explore the minified JavaScript, find the variable mappings, and create new patch files without me approving every little thing that way.

In fact, it was able to complete the migration pretty much on its own. It tried applying the patches, found that some didn't work with the new version, iterated to fix them, and even improved the instruction document for future instances based on what it learned.

I even created SafeClaw to make running containerized Claude Code sessions easy. It lets you spin up multiple isolated sessions, each with a web terminal, and manage them all from a dashboard. It uses several customizations from this repo, including the optimized system prompt, dx plugin, and status line.

Advanced: Orchestrating a worker Claude Code in a container

You can take this further by having your local Claude Code control another Claude Code instance running inside a container. The trick is using tmux as the control layer:

1. Your local Claude Code starts a tmux session
2. In that tmux session, it runs or connects to the container
3. Inside the container, Claude Code runs with `--dangerously-skip-permissions`
4. Your outer Claude Code uses `tmux send-keys` to send prompts and `capture-pane` to read output

This gives you a fully autonomous "worker" Claude Code that can run experimental or long-running tasks without you approving every action. When it's done, your local Claude Code can pull the results back. If something goes wrong, it's all sandboxed in the container.

Advanced: Multi-model orchestration

Beyond just Claude Code, you can run different AI CLIs in containers - Codex, Gemini CLI, or others. I tried OpenAI Codex for code review, and it works well. The point isn't that you can't run these CLIs directly on your host machine - you obviously can. The value is that Claude Code's UI/UX is smooth enough that you can just talk to it and let it handle the orchestration: spinning up different models, sending data between containers and your host. Instead of manually switching between terminals and copy-pasting, Claude Code becomes the central interface that coordinates everything.

Why it matters: Containers let you give Claude full autonomy without risking your system. The worker pattern is how you get true unattended agentic workflows.

TIP 22

The Best Way to Get Better at Using Claude Code Is by Using It

Recently I saw a world-class rock climber being interviewed by another rock climber. She was asked, "How do you get better at rock climbing?" She simply said, "By rock climbing."
That's how I feel about this too. Of course, there are supplementary things you can do, like watching videos, reading books, learning about tips. But using Claude Code is the best way to learn how to use it. Using AI in general is the best way to learn how to use AI.

I like to think of it like a billion token rule instead of the 10,000 hour rule. If you want to get better at AI and truly get a good intuition about how it works, the best way is to consume a lot of tokens. And nowadays it's possible. I found that especially with Opus 4.5, it's powerful enough but affordable enough that you can run multiple sessions at the same time. You don't have to worry as much about token usage, which frees you up a lot.

Why it matters: Intuition for AI comes from token-hours, not reading hours. The billion token rule: use it until the patterns become instinctive.

TIP 23

Clone/Fork and Half-Clone Conversations

Sometimes you want to try a different approach from a specific point in a conversation without losing your original thread. The clone-conversation script lets you duplicate a conversation with new UUIDs so you can branch off.
Built-in alternatives (recent versions): Claude Code now has native forking:

- `/fork` - forks the current session from within a conversation
- `--fork-session` - use with `--resume` or `--continue` (e.g., `claude -c --fork-session`)

Since `--fork-session` has no short form, you can add this function to your `~/.zshrc` or `~/.bashrc` to use `--fs` as a shortcut:

fork shortcuts

```
claude() {
  local args=()
  for arg in "$@"; do
    if [[ "$arg" == "--fs" ]]; then
      args+=("--fork-session")
    else
      args+=("$arg")
    fi
  done
  command claude "${args[@]}"
}
```

This intercepts all `claude` commands, expands `--fs` to `--fork-session`, and passes everything else through unchanged. Works with aliases too (see Tip 7): `c -c --fs`, `ch -c --fs`, etc.

The clone script below predates these built-in options, but the half-clone script below that remains unique for reducing context.

The first message is tagged with `[CLONED <timestamp>]` (e.g., `[CLONED Jan 7 14:30]`), which shows up both in the `claude -r` list and inside the conversation.

To set it up manually, symlink both files:

clone setup

```
ln -s /path/to/this/repo/scripts/clone-conversation.sh
~/.claude/scripts/clone-conversation.sh
ln -s /path/to/this/repo/skills/clone ~/.claude/skills/clone
```

Or install via the dx plugin - no symlinks needed.

Then just type `/clone` (or `/dx:clone` if using the plugin) in any conversation and Claude will handle finding the session ID and running the script.

I've tested this extensively and the cloning works really well.

Half-clone to reduce context

When a conversation gets too long, the half-clone-conversation script keeps only the later half. This reduces token usage while preserving your recent work. The first message is tagged with `[HALF-CLONE <timestamp>]` (e.g., `[HALF-CLONE Jan 7 14:30]`).

To set it up manually, symlink both files:

half-clone setup

```
ln -s /path/to/this/repo/scripts/half-clone-conversation.sh
~/.claude/scripts/half-clone-conversation.sh
ln -s /path/to/this/repo/skills/half-clone ~/.claude/skills/half-clone
```

Or install via the dx plugin - no symlinks needed.

Auto-suggest half-clone with a hook

Optionally, you can use a hook to automatically trigger `/half-clone` when your context gets too long. The check-context script runs after every Claude response and checks context usage. If it's over 85%, it tells Claude to run `/half-clone`, which creates a new conversation with only the later half so a new agent can continue there.

To set it up, first copy the script:

hook setup

```
cp /path/to/this/repo/scripts/check-context.sh ~/.claude/scripts/check-
context.sh
chmod +x ~/.claude/scripts/check-context.sh
```

Then add the hook to your `~/.claude/settings.json`:

~/.claude/settings.json

```json
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "~/.claude/scripts/check-context.sh"
          }
        ]
      }
    ]
  }
}
```

This requires auto-compact to be disabled (`/config` > Auto-compact > false), otherwise Claude Code may compact the context before the hook gets a chance to fire. When triggered, the hook blocks Claude from stopping and tells it to run `/half-clone`. The advantage over auto-compact is that half-clone is deterministic and fast - it keeps your actual messages intact instead of summarizing them.

Recommended permission for clone scripts

Both clone scripts need to read `~/.claude` (for conversation files and history). To avoid permission prompts from any project, add this to your global settings (`~/.claude/settings.json`):

permissions

```json
{
  "permissions": {
    "allow": ["Read(~/.claude)"]
  }
}
```

Why it matters: Forking lets you explore different solutions without losing progress. Half-cloning keeps conversations lean without losing recent context.

TIP 24

Use realpath to Get Absolute Paths

When you need to tell Claude Code about files in a different folder, use `realpath` to get the full absolute path:

```
absolute paths

realpath some/relative/path
```

Why it matters: Relative paths fail when Claude's working directory isn't what you expect. Absolute paths are unambiguous.

TIP 25

Understanding CLAUDE.md vs Skills vs Slash Commands vs Plugins

These are somewhat similar features and I initially found them pretty confusing. I've been unpacking them and trying my best to wrap my head around them, so I wanted to share what I learned.
CLAUDE.md is the simplest one. It's a bunch of files that get treated as the default prompt, loaded into the beginning of every conversation no matter what. The nice thing about it is the simplicity. You can explain what the project is about in a particular project (`./CLAUDE.md`) or globally (`~/.claude/CLAUDE.md`).

Skills are like better-structured CLAUDE.md files. They can be invoked by Claude automatically when relevant, or manually by the user with a slash (e.g., `/my-skill`). For example, you could have a skill that opens a Google Translate link with proper formatting when you ask how to pronounce a word in a certain language. If those instructions are in a skill, they only load when needed. If they were in CLAUDE.md, they'd already be there taking up space. So skills are more token-efficient in theory.

Slash Commands are similar to skills in that they're ways of packaging instructions separately. They can be invoked manually by the user, or by Claude itself. If you need something more precise, to invoke at the right time at your own pace, slash commands are the tool to use.

Skills and slash commands are pretty similar in the way they function. The difference is the intention of the design - skills are primarily designed for Claude to use, and slash commands are primarily designed for the user to use. However, they have ended up merging them, as I had suggested this change.

Plugins are a way to package skills, slash commands, agents, hooks, and MCP servers together. But a plugin doesn't have to use all of them. Anthropic's official `frontend-design` plugin is essentially just a skill and nothing else. It could be distributed as a standalone skill, but the plugin format makes it easier to install.

For example, I built a plugin called `dx` that bundles slash commands and a skill from this repo together. You can see how it works in the Install the dx plugin section (Tip 44).

Why it matters: Understanding this hierarchy lets you put instructions in the right place. CLAUDE.md for always-on defaults, skills for on-demand expertise, plugins for distribution.

> MY TAKE
>
> *I keep CLAUDE.md minimal and move specialized knowledge into skills. Every token in CLAUDE.md is a tax on every conversation. Skills only load when needed.*

TIP 26

Interactive PR Reviews

Claude Code is great for PR reviews. The procedure is pretty simple: you ask it to retrieve PR information using the `gh` command, and then you can go through the review however you want. You can do a general review, or go file by file, step by step. You control the pace. You control how much detail you want to look into and the level of complexity you want to work at. Maybe you just want to understand the general structure, or maybe you want to have it run tests too.

The key difference is that Claude Code acts as an interactive PR reviewer, not just a one-shot machine. Some AI tools are good at one-shot reviews (including the latest GPT models), but with Claude Code you can have a conversation.

Why it matters: Interactive review means you can go as deep or as shallow as you want. One-shot tools give you one perspective; Claude gives you a conversation partner.

TIP 27

Claude Code as a Research Tool

Claude Code is amazing for any sort of research. It's essentially a Google replacement or deep research replacement, but more advanced in a few different ways. Whether you're researching why certain GitHub Actions failed (which I've been doing a lot recently), doing sentiment or market analysis on Reddit, exploring your codebase, or exploring public information to find something - it's able to do that.

The key is giving it the right pieces of information and instructions about how to access those pieces of information. It might be `gh` terminal command access, or the container approach (Tip 21), or Reddit through Gemini CLI (Tip 11), or private information through an MCP like Slack MCP, or the Cmd+A / Ctrl+A method (Tip 10) - whatever it is. Additionally, if Claude Code has trouble loading certain URLs, you can try using Playwright MCP or Claude's native browser integration (see Tip 9). For scientific research, I created a paper-search plugin for searching academic papers.

In fact, I was even able to save $10,000 by using Claude Code for research.

Why it matters: Research is about connecting information from multiple sources. Claude Code can access GitHub, Reddit, Slack, the web, and your codebase in a single session.

TIP 28

Mastering Different Ways of Verifying Its Output

One way to verify its output if it's code is to have it write tests and make sure the tests look good in general. That's one way, but you can of course check the code it generates as it goes, just on the Claude Code UI. Another thing is you can use a visual Git client like GitHub Desktop for example. I personally use it. It's not a perfect product, but it's good enough for checking changes quickly. And having it generate a PR as I probably mentioned earlier in this post is a great way as well. Have it create a draft PR, check the content before turning it into a real PR.

Another one is letting it check itself, its own work. If it gives you some sort of output, let's say from some research, you can say "are you sure about this? Can you double check?" One of my favorite prompts is to say "double check everything, every single claim in what you produced and at the end make a table of what you were able to verify" - and that seems to work really well.

Why it matters: Trust but verify. Multiple verification methods catch different types of errors. The self-check prompt is surprisingly effective at catching hallucinations.

TIP 29

Claude Code as a DevOps Engineer

I wanted to specifically create a separate tip for this because it's been really amazing for me. Whenever there are GitHub Actions CI failures, I just give it to Claude Code and say "dig into this issue, try to find the root cause." Sometimes it gives you surface level answers, but if you just keep asking - was it caused by a particular commit, a particular PR, or is it a flaky issue? - it really helps you dig into these nasty issues that are hard to dig into by hand. You would need to wade through a bunch of logs and that would be super painful to do manually, but Claude Code is able to handle a lot of that.
I've packaged this workflow as a `/gha` slash command - just run `/gha <url>` with any GitHub Actions URL and it will automatically investigate the failure, check for flakiness, identify breaking commits, and suggest fixes. You can find it in the skills folder, or install it via the dx plugin.

Once you identify what the particular problem was, you can just create a draft PR and go through some of the tips I mentioned earlier - check the output, make sure it looks good, let it verify its own outputs, and then turn it into a real PR to actually fix the issue. It's been working really well for me personally.

Why it matters: CI failure investigation is tedious, log-heavy work that Claude excels at. The /gha command turns a 30-minute debugging session into a 5-minute review.

TIP 30

Keep CLAUDE.md Simple and Review It Periodically

It's important to keep CLAUDE.md simple and as concise as possible. You can just start with no CLAUDE.md at all. And if you find that you keep telling Claude Code the same thing over and over again, then you can just add it to CLAUDE.md. I know there is an option to do that through the # symbol, but I prefer to just ask Claude Code to either add it to the project level CLAUDE.md or the global CLAUDE.md and it'll know what to edit exactly.
It's also important to periodically review your CLAUDE.md files because they can get outdated over time. Instructions that made sense some time ago might no longer be relevant, or you might have new patterns that should be documented. I created a skill for this called `review-claudemd` that analyzes your recent conversations and suggests improvements for your CLAUDE.md files.

Why it matters: A bloated CLAUDE.md wastes tokens and confuses Claude with contradictory instructions. Lean and current beats comprehensive and stale.

MY TAKE

My CLAUDE.md has been rewritten 4 times. Each time it gets shorter and sharper. The best version is the one that says only what Claude keeps getting wrong without it.

TIP 31

Claude Code as the Universal Interface

I used to think with Claude Code, CLI is like the new IDE, and it's still true in a way. I think it's a great first place to open your project whenever you want to make quick edits and stuff like that. But depending on the severity of your project, you want to be more careful about the outputs than just staying at the vibe coding level.
But what's also true, the more general case of that, is that Claude Code is really the universal interface to your computer, the digital world, any sort of digital problem that you have. You can let it figure it out in many cases. For example, if you need to do a quick edit of your video, you can just ask it to do that - it'll probably figure out how to do that through ffmpeg or something similar. If you want to transcribe a bunch of audio files or video files that you have locally, you can just ask it to do that - it might suggest to use Whisper through Python. If you want to analyze some data that you have in a CSV file, it might suggest to use Python or JavaScript to visualize that. And of course with internet access - Reddit, GitHub, MCPs - the possibilities are endless.

It's also great for any operations you want to perform on your local computer. For example, if you're running out of storage, you can just ask it to give you some advice on how to clean that up. It'll look through your local folders and files, try to find what's taking up a lot of space, and then give you advice on how to clean them up - maybe delete particularly large files. In my case, I had some Final Cut Pro files that were really large that I should have cleaned up. Claude Code told me about it. Maybe it'll tell you to clean up unused Docker images and containers using `docker system prune`. Or maybe it'll tell you to clean up some cache that you never realized was still there. No matter what you want to do on your computer, Claude Code is the first place I go to now.

I think it's kind of interesting because the computer started with a text interface. And we're, in a way, coming back to this text interface that you can spin up three or four tabs at a time, as I mentioned earlier. To me, that's really exciting. It feels like you have a second brain, in a way. But because of the way it's structured, because it's just a terminal tab, you can open up a third brain, a fourth brain, a fifth brain, a sixth brain. And as the models become more powerful, the proportion of the thinking that you can delegate to these things - not the important things, but things that you don't want to do or that you find boring or too tedious - you can just let them take care of it. As I mentioned, a good example of that is looking into GitHub Actions. Who wants to do that? But it turns out these agents are really good at those boring tasks.

Why it matters: Claude Code isn't just for coding. It's the first tool to try for any digital task. Video editing, data analysis, system cleanup, research -- all from one interface.

TIP 32

It's All About Choosing the Right Level of Abstraction

As I mentioned earlier, sometimes it's okay to stay at the vibe coding level. You don't necessarily have to worry about every single line of code if you're working on one-time projects or non-critical parts of the codebase. But other times, you want to dig in a little deeper - look at the file structure and functions, individual lines of code, even checking dependencies.
The key is that it's not binary. Some people say vibe coding is bad because you don't know what you're doing, but sometimes it's totally fine. But other times, it is helpful to dig deeper, use your software engineering skills, understand code at a granular level, or copy and paste parts of the codebase or specific error logs to ask Claude Code specific questions about them.

It's sort of like you're exploring a giant iceberg. If you want to stay at the vibe coding level, you can just fly over the top and check it from far away. Then you can go a little bit closer. You can go into diving mode. You can go deeper and deeper, with Claude Code as your guide.

Why it matters: Knowing when to vibe-code and when to deep-dive is the meta-skill. Both have their place. The best engineers use the full spectrum.

TIP 33

Audit Your Approved Commands

I recently saw this post where someone's Claude Code ran `rm -rf tests/ patches/ plan/ ~/` and wiped their home directory. It's easy to dismiss as a vibe coder mistake, but this kind of mistake could happen to anyone. So it's important to audit your approved commands from time to time. To make it easier, I built **cc-safe** - a CLI that scans your `.claude/settings.json` files for risky approved commands.
It detects patterns like:

- `sudo`, `rm -rf`, `Bash`, `chmod 777`, `curl | sh`
- `git reset --hard`, `npm publish`, `docker run --privileged`
- And more - it's container-aware so `docker exec` commands are skipped

It recursively scans all subdirectories, so you can point it at your projects folder to check everything at once. You can run it manually or ask Claude Code to run it for you:

security audit

```
npm install -g cc-safe
cc-safe ~/projects
```

Or just run it directly with npx:

npx

```
npx cc-safe .
```

GitHub: cc-safe

Why it matters: **One bad approved command can wipe your home directory. A 10-second audit prevents catastrophic data loss.**

TIP 34

Write Lots of Tests (and Use TDD)

As you write more code with Claude Code, it becomes easier to make mistakes. PR reviews and visual Git clients help catch issues (as I mentioned earlier), but writing tests is crucial as your codebase grows larger.
You can have Claude Code write tests for its own code. Some people say AI can't test its own work, but it turns out it can - similar to how the human brain works. When you write tests, you're thinking about the same problem in a different way. The same applies to AI.

I've found that TDD (Test-Driven Development) works really well with Claude Code:

1. Write tests first
2. Make sure they fail
3. Commit the tests
4. Write the code to make them pass

This is actually how I built cc-safe. By writing failing tests first and committing them before implementation, you create a clear contract for what the code should do. Claude Code then has a concrete target to hit, and you can verify the implementation is correct by running the tests.

If you want to be extra sure, review the tests yourself to make sure they don't do anything stupid like just returning true.

Why it matters: Tests give Claude a concrete target. Instead of generating "looks right" code, it generates "passes the tests" code. That's a measurable difference.

> **MY TAKE**
>
> *TDD + Claude Code is the most underrated combo. Writing failing tests first creates a contract. Claude hits the target instead of drifting.*

TIP 35

Be Braver in the Unknown; Iterative Problem Solving

Since I started using Claude Code more intensely, I've noticed that I became more and more brave in the unknown.
For example, when I started working at Daft, I noticed a problem with our frontend code. I'm not an expert in React, but I decided to dig into it anyway. I just started asking questions about the codebase and about the problem. Eventually I was able to solve it because I knew how to iteratively solve problems with Claude Code.

A similar thing happened recently. I was building a guide for users of Daft and ran into some very specific issues: cloudpickle not working with Google Colab with Pydantic, and a separate issue with Python and a bit of Rust where things weren't printing correctly in JupyterLab even though they worked fine in the terminal. I had never worked with Rust before.

I could have just created an issue and let other engineers handle it. But I thought, let me dig into the codebase. Claude Code came up with an initial solution, but it wasn't that good. So I slowed down. A colleague suggested we just disable that part, but I didn't want any regression. Can we find a better solution?

What followed was a collaborative and iterative process. Claude Code suggested potential root causes and solutions. I experimented with those. Some turned out to be dead ends, so we went in a different direction. Throughout this, I controlled my pace. Sometimes I went faster, like when letting it explore different solution spaces or parts of the codebase. Sometimes I went slower, asking "what does this line mean exactly?" Controlling the level of abstraction, controlling the speed.

Eventually I found a pretty elegant solution. The lesson: even in the world of the unknown, you can do a lot more with Claude Code than you might think.

Why it matters: Claude Code expands your capability frontier. Problems in unfamiliar languages or frameworks become solvable through iterative collaboration.

TIP 36

Running Bash Commands and Subagents in the Background

When you have a long-running bash command in Claude Code, you can press Ctrl+B to move it to run in the background. Claude Code knows how to manage background processes - it can check on them later using the BashOutput tool.
This is useful when you realize a command is taking longer than expected and you want Claude to do something else in the meantime. You can then either have it use the exponential backoff method I mentioned in Tip 17 to check on progress, or just let it work on something else entirely while the process runs.

Claude Code also has the ability to run subagents in the background. If you need to do long-running research or have an agent check on something periodically, you don't have to keep it running in the foreground. Just ask Claude Code to run an agent or task in the background, and it'll handle it while you continue with other work.

Using subagents strategically

Beyond just running things in the background, subagents are useful when you have a large task to break down. For example, if you have a huge codebase that you need to analyze, you can have subagents analyze it in different ways or look at different parts of the codebase in parallel. Just ask Claude to spawn multiple subagents to handle different pieces.

You can customize subagents by just asking:

- **How many** - ask Claude to spawn the number you want
- **Background vs foreground** - ask to run them in the background, or press Ctrl+B
- **Which model** - ask for Opus, Sonnet, or Haiku depending on the complexity of each task (subagents default to Sonnet)

Why it matters: Background subagents let you run 2-3 analysis or coding tasks simultaneously from a single terminal. Parallel execution, not sequential waiting.

> **MY TAKE**
>
> *Combined with worktrees and voice, this is the "run 3 agents from your phone" workflow.*

TIP 37

The Era of Personalized Software Is Here

We're entering an era of personalized, custom software. Since AI came out - ChatGPT in general, but especially Claude Code - I've noticed that I'm able to create a lot more software, sometimes just for myself, sometimes for small projects.
As I mentioned earlier in this document, I've created a custom transcription tool that I use every day to talk to Claude Code. I've created ways to customize Claude Code itself. I've also done a bunch of data visualization and data analysis tasks using Python much faster than I could otherwise.

Here's another example: korotovsky/slack-mcp-server, a popular Slack MCP with almost 1,000 stars, is designed to run as a Docker container. I had trouble using it smoothly inside my own Docker container (Docker-in-Docker complications). Instead of fighting with that setup, I just asked Claude Code to write a CLI using Slack's Node SDK directly. It worked really well.

This is an exciting time. Whatever you want to get done, you can ask Claude Code to do it. If it's small enough, you can build it in an hour or two. I even created a slide deck template - a single HTML file with CSS and JavaScript that lets you embed an interactive, persistent terminal process inside.

Why it matters: The cost of custom software dropped to near-zero. Instead of adapting to tools, build tools that adapt to you.

TIP 38

Navigating and Editing Your Input Box

Claude Code's input box is designed to emulate common terminal/readline shortcuts, which makes it feel natural if you're used to working in the terminal. Here are some useful ones:
Navigation:

- `Ctrl+A` - Jump to the beginning of the line
- `Ctrl+E` - Jump to the end of the line
- `Option+Left/Right` (Mac) or `Alt+Left/Right` - Jump backward/forward by word

Editing:

- `Ctrl+W` - Delete the previous word
- `Ctrl+U` - Delete from cursor to beginning of line
- `Ctrl+K` - Delete from cursor to end of line
- `Ctrl+C` / `Ctrl+L` - Clear the current input
- `Ctrl+G` - Open your prompt in an external editor (useful for pasting long text, since pasting directly into the terminal can be slow)

If you're familiar with bash, zsh, or other shells, you'll feel right at home.

For `Ctrl+G`, the editor is determined by your `EDITOR` environment variable. You can set it in your shell config (`~/.zshrc` or `~/.bashrc`):

editor setup

```
export EDITOR=vim       # or nano, code, nvim, etc.
```

Or in `~/.claude/settings.json` (requires restart):

~/.claude/settings.json

```
{
  "env": {
    "EDITOR": "vim"
  }
}
```

Entering newlines (multi-line input):

The quickest method works everywhere without any setup: type \ followed by Enter to create a newline. For keyboard shortcuts, run `/terminal-setup` in Claude Code. On Mac Terminal.app, I use Option+Enter.

Pasting images:

- `Ctrl+V` (Mac/Linux) or `Alt+V` (Windows) - Paste an image from your clipboard

Note: On Mac, it's `Ctrl+V`, not `Cmd+V`.

Why it matters: These shortcuts eliminate the friction of editing long prompts. Ctrl+G alone is worth learning for complex multi-line inputs.

TIP 39

Spend Some Time Planning, but Also Prototype Quickly

You want to spend enough time planning so that Claude Code knows what to build and how to build it. This means making high-level decisions early: what technology to use, how the project should be structured, where each functionality should live, which files things should go in. It's important to make good decisions as early as you can.
Sometimes prototyping helps with that. Just by making a simple prototype quickly, you might be able to say "okay, this technology works for this particular purpose" or "this other technology works better."

For example, I was recently experimenting with creating a diff viewer. I first tried a simple bash prototype with tmux and lazygit, then tried making my own git viewer with Ink and Node. I had a lot of trouble with different things and ended up not publishing any of these results. But what I got reminded of through this project is the importance of planning and prototyping. I found that just by planning a little bit better at the beginning before you let it write code, you're able to guide it better. You still need to guide it throughout the process of coding, but letting it plan a little first is really helpful.

You can use plan mode for this by pressing Shift+Tab to switch to it. Or you can just ask Claude Code to make a plan before writing any code.

Why it matters: 10 minutes of planning saves hours of wrong-direction coding. Prototypes validate tech choices before you commit to a full implementation.

TIP 40

Simplify Overcomplicated Code

I've found that Claude Code sometimes overcomplicates things and writes too much code. It makes changes you didn't ask for. It just seems to have a bias for writing more code. The code might work correctly if you've followed the other tips in this guide, but it's going to be hard to maintain and hard to check. It can be kind of a nightmare if you don't review it enough.
So sometimes you want to check the code and ask it to simplify things. You could fix things yourself, but you could also just ask it to simplify. You can ask questions like "why did you make this particular change?" or "why did you add this line?"

Some people say if you write code only through AI, you'll never understand it. But that's only true if you don't ask enough questions. If you make sure you understand every single thing, you can actually understand code faster than otherwise because you can ask AI about it. Especially when you're working on a large project.

Note that this applies to prose as well. Claude Code often tries to summarize previous paragraphs in the last paragraph, or previous sentences in the last sentence. It can get pretty repetitive. Sometimes it's helpful, but most of the time you'll need to ask it to remove or simplify it.

Why it matters: Code you don't understand is a liability. Asking "why?" forces Claude to justify every line and surfaces unnecessary complexity.

TIP 41

Automation of Automation

At the end of the day, it's all about automation of automation. What I mean by that is I've found it's the best way to not just become more productive, but also make the process more fun. At least to me, this whole process of automation of automation is really fun.
I personally started with ChatGPT and wanted to automate the process of copy-pasting and running commands that ChatGPT gave me in the terminal. I automated that whole process by building a ChatGPT plugin called Kaguya. I've consistently worked towards more and more automation since then.

Nowadays, luckily, we don't even have to build a tool like that because tools like Claude Code exist and they work really well. And as I've used it more and more, I found myself thinking, well, what if I could automate the process of typing? So I used Claude Code itself to build my voice transcription app, as I mentioned earlier.

Then I started to think, I find myself repeating myself sometimes. So I would put those things in CLAUDE.md. Then I would think, okay, sometimes I go through running the same command over and over again. How can I automate that? Maybe I can ask Claude Code to do it. Or maybe I can put them in skills. Or maybe I can even have it create a script so I don't have to repeat the same process over and over again.

I think ultimately that's where we're heading. Whenever you find yourself repeating the same task or the same command over and over again, a couple of times is okay, but if you repeat it over and over again, then think about a way to automate that whole process.

Why it matters: Each layer of automation compounds. Voice replaces typing. CLAUDE.md replaces repeated instructions. Skills replace repeated workflows. The end state is near-zero friction.

TIP 42

Share Your Knowledge and Contribute Where You Can

This tip is a bit different from the others. I found that by learning as much as you can, you're able to share your knowledge with people around you. Maybe through posts like these, maybe even books, courses, videos. I also recently had an internal session for my colleagues at Daft. It's been very rewarding.
And whenever I share tips, I often get information back. For example, when I shared my trick for shortening the system prompt and tool descriptions (Tip 15), some people told me about the `--system-prompt` flag that you can use as an alternative. Another time, I shared about the difference between slash commands and skills (Tip 25), and I learned new things from comments on that Reddit post.

So sharing your knowledge isn't just about establishing your brand or solidifying your learning. It's also about learning new things through that process. It's not always a one-way street.

When it comes to contributing, I've been sending issues to the Claude Code repo. I thought, okay, if they listen, cool. If they don't, that's totally fine. I didn't have any expectations. But in version 2.0.67, I noticed they took multiple suggestions from reports I made:

- Fixed scroll position resetting after deleting a permission rule in `/permissions`
- Added search functionality to `/permissions` command

It's kind of amazing how fast the team can react to feature requests and bug reports. But it makes sense because they're using Claude Code to build Claude Code itself.

Why it matters: Teaching is the fastest way to learn. Contributing is how tools get better. Both create a positive feedback loop.

TIP 43

Keep Learning!

There are several effective ways to keep learning about Claude Code:
Ask Claude Code itself - If you have a question about Claude Code, just ask it. Claude Code has a specialized subagent for answering questions about its own features, slash commands, settings, hooks, MCP servers, and more.

Check the release notes - Type `/release-notes` to see what's new in your current version. This is the best way to learn about the latest features.

Learn from the community - The r/ClaudeAI subreddit is a great place to learn from other users and see what workflows people are using.

Follow Ado for tips - Ado (@adocomplete) is a DevRel at Anthropic who posted daily Claude Code tips throughout December 2025 in his "Advent of Claude" series. While this particular series has already ended, he continues to share useful tips on X.

- Twitter/X: Advent of Claude posts
- LinkedIn: Advent of Claude posts

Why it matters: Claude Code evolves weekly. Features from a month ago may already be outdated. Staying current is a competitive advantage.

TIP 44

Install the dx Plugin

This repo is also a Claude Code plugin called `dx` (developer experience). It bundles several tools from the tips above into a single install:

Skill	Description
`/dx:gha <url>`	Analyze GitHub Actions failures (Tip 29)
`/dx:handoff`	Create handoff documents for context continuity (Tip 8)
`/dx:clone`	Clone conversations to branch off (Tip 23)
`/dx:half-clone`	Half-clone to reduce context (Tip 23)
`/dx:reddit-fetch`	Fetch Reddit content via Gemini CLI (Tip 11)
`/dx:review-claudemd`	Review conversations to improve CLAUDE.md files (Tip 30)

Install with two commands:

install

```
claude plugin marketplace add ykdojo/claude-code-tips
claude plugin install dx@ykdojo
```

After installing, the commands are available as `/dx:clone`, `/dx:half-clone`, `/dx:handoff`, and `/dx:gha`. The `reddit-fetch` skill is invoked automatically when you ask about Reddit URLs. The `review-claudemd` skill analyzes your recent conversations and suggests improvements for your CLAUDE.md files. For the clone commands, see the recommended permission.

Recommended companion: Playwright MCP for browser automation - add with `claude mcp add -s user playwright npx @playwright/mcp@latest`

Why it matters: One install gives you 6 tools that would otherwise require manual setup. Plugins are the distribution mechanism for Claude Code workflows.

TIP 45

Quick Setup Script

If you want to set up multiple recommendations from this repo at once, there's a setup script that handles many of them:

one-liner setup

```
bash <(curl -s https://raw.githubusercontent.com/ykdojo/claude-code-tips/main/scripts/setup.sh)
```

The script shows you everything it will configure and lets you skip any items:

setup options

```
INSTALLS:
  1. DX plugin - slash commands (/dx:gha, /dx:clone, /dx:handoff) and
skills (reddit-fetch)
  2. cc-safe - scans your settings for risky approved commands like 'rm -
rf' or 'sudo'

SETTINGS (~/.claude/settings.json):
  3. Status line - shows model, git branch, uncommitted files, token usage
at bottom of screen
  4. Disable auto-updates - prevents Claude Code from auto-updating (useful
for system prompt patches)
  5. Lazy-load MCP tools - only loads MCP tool definitions when needed,
saves context
  6. Read(~/.claude) permission - allows clone/half-clone commands to read
conversation history
  7. Read(//tmp/**) permission - allows reading temporary files without
prompts
  8. Disable attribution - removes Co-Authored-By from commits and
attribution from PRs

SHELL CONFIG (~/.zshrc or ~/.bashrc):
  9. Aliases: c=claude, ch=claude --chrome, cs=claude --dangerously-skip-
permissions
 10. Fork shortcut: --fs expands to --fork-session (e.g., claude -c --fs)

Skip any? [e.g., 1 4 7 or Enter for all]:
```

Why it matters: One command configures 10 optimizations. Skip what you don't want, keep what you do. Zero manual editing of config files.

Quick Reference Cheat Sheet

claude-code-cheatsheet

`alias c='claude'`	Quick launch from any terminal
`claude -c`	Continue last conversation
`claude -r`	Resume a recent conversation
`/compact`	Summarize context to free tokens

`/fork`	Fork conversation from current point
`/copy`	Copy last response to clipboard as markdown
`/usage`	Check rate limits and token usage
`/chrome`	Toggle native browser integration
`Ctrl+B`	Move running command to background
`Shift+Tab`	Toggle plan mode
`Ctrl+G`	Open prompt in external editor
`git worktree add ../feat feature`	Create parallel branch workspace
`npx cc-safe ~/projects`	Audit approved commands for danger
`claude plugin install dx@ykdojo`	Install the dx plugin bundle
`ENABLE_TOOL_SEARCH: "true"`	Lazy-load MCP tools (settings.json)

Curated by Ghiles Moussaoui — Muditek

AI RevOps — I find your revenue leaks, build the fix, and run it without you.

Source: github.com/ykdojo/claude-code-tips