

The hitch-hiker's guide to Claude Code



Beginner to advanced guide on
using Claude Code

By Sahil Chawla



The Hitchhiker's Guide to Claude Code

Don't Panic: A Zero-to-Hero Journey Through AI-Powered Development

Author: Sahil Chawla

Published: January 26, 2026

Version: 1.0

Foreword: Don't Panic!

If you're reading this, you've probably heard about Claude Code and wondered whether it's worth the hype. Maybe you're skeptical about AI coding assistants. Maybe you've tried one and found it wanting. Or perhaps you're just curious about what all the fuss is about.

Here's the truth: **Claude Code will change how you write software.** Not might. Will.

This guide is your towel—the most massively useful thing you can have when hitchhiking through the galaxy of AI-assisted development. It's practical, honest, and occasionally irreverent. We'll cover everything from installation to advanced techniques, with real examples and hard-won lessons.

Ready? **Don't Panic.** Let's begin.

Table of Contents

Part I: Getting Started (Don't Panic!)

1. Introduction: What is Claude Code? 1
2. Installation: Setting Up Your Towel 4
3. First Steps: Your First Conversation 9

Part II: The Guide to the Guide

4. CLAUDE.md: Your Project's Constitution 12
5. Commands: Slash Your Way to Productivity 19
6. Plan Mode: Think Before You Code 23

Part III: Advanced Techniques

7. Sub-agents: Delegating Like a Pro 27
8. Skills & Hooks: Customization Mastery 31
9. Context Engineering: Making Every Token Count 35

Part IV: Practical Matters

10. Pricing & Cost Management 39
11. Integrations: VS Code, Cursor & Beyond 43
12. Best Practices from Power Users 47

Part V: The Reality Check

13. The Open Source Experiment: What I Learned 51
 14. Quick Reference & Cheat Sheet 55
-

Part I: Getting Started

Chapter 1: Don't Panic - What is Claude Code?

The Elevator Pitch

Claude Code is Anthropic's terminal-native AI coding assistant. Unlike autocomplete tools that suggest snippets as you type, Claude Code operates as an **agentic system**—meaning it can:

- **Read and understand** your entire project structure
- **Search through files**, grep for patterns, analyze dependencies
- **Plan complex implementations** before writing any code
- **Execute multi-file changes** while respecting your conventions
- **Run shell commands**, tests, and build processes
- **Learn your project's rules** through configuration files

Think of it as having a senior developer in your terminal who never gets tired, never forgets your coding standards, and can hold 200,000 tokens of context in memory.

Why Should You Care?

Speed: Tasks that took hours now take minutes. I've scaffolded entire APIs, refactored legacy codebases, and debugged race conditions faster than ever before.

Quality: Claude doesn't just write code—it reasons about architecture, considers edge cases, and follows your project's patterns.

Learning: Every interaction is a learning opportunity. Claude explains its decisions, suggests improvements, and adapts to your style.

The Honest Comparison

Feature	Claude Code	GitHub Copilot	ChatGPT/Codex	Cursor
Interface	Terminal (CLI)	IDE Plugin	Web/API	Full IDE
Context Window	200,000 tokens	~8,000 tokens	128,000 tokens	Project-aware
Approach	Agentic reasoning	Autocomplete	Conversational	Multi-file refactoring
Best For	Architecture, debugging	Fast completions	Brainstorming	Large refactors
Learning Curve	Medium	Low	Low	Low
Pricing	\$20-200/month	\$0-39/month	\$20-200/month	\$20-40/month

My Take:

- **GitHub Copilot** excels at autocomplete and boilerplate
- **ChatGPT** is great for brainstorming and exploration
- **Cursor** shines for multi-file refactoring with IDE comfort
- **Claude Code** wins when you need deep reasoning and architectural understanding

 **Pro Tip:** You don't have to choose just one. I use all of them for different purposes. Claude Code is my primary tool for serious development work.

Chapter 2: Setting Up Your Towel (Installation)

Prerequisites

Before you start, you'll need:

- **A Claude subscription** (Pro at \$20/month minimum)
- **Node.js** (for npm installation method)
- **A terminal** (Command Prompt, PowerShell, or your IDE's integrated terminal)

Three Ways to Access Claude Code

Claude Code isn't just a CLI tool. You have three interfaces:

1. **CLI (Terminal)** - The original experience. Lightweight and fast.
2. **Desktop App** - Standalone application with dedicated UI
3. **Cloud (claude.ai)** - Browser-based, works anywhere including mobile

My Setup:

- **CLI** for intense coding sessions
- **Desktop** for general development (warning: memory intensive on long sessions)
- **Cloud/Mobile** for brainstorming away from my machine

They all share your Claude subscription—same account, same limits, different contexts.

Windows Installation

The official PowerShell script can be finicky. Here's the reliable approach:

Step 1: Install Node.js

```
bash

# Download from nodejs.org and run installer
# Verify installation:
node -v # Should show v22.x.x or newer
```

Step 2: Install Claude Code

```
bash

npm install -g @anthropic-ai/clause-code
```

Step 3: Launch

```
bash
```

On first run:

1. Choose your theme (dark mode recommended!)
2. Connect your Anthropic account
3. Trust the current directory

macOS / Linux Installation

Multiple options available:

```
bash

# Using Homebrew (recommended for macOS)
brew install --cask claude-code

# Or using the official installer
curl -fsSL https://claude.ai/install.sh | bash

# Then simply run:
claude
```

Troubleshooting Common Issues

Permission Errors on npm Install

Don't use `(sudo npm install -g)`. Instead:

```
bash

mkdir ~/.npm-global
npm config set prefix '~/.npm-global'
export PATH=~/.npm-global/bin:$PATH
```

Add that last line to your `(.bashrc)` or `(.zshrc)` for persistence.

Windows: Shift+Tab Not Working

Known issue where Plan Mode gets skipped. Try `(Alt + M)` instead.

Verification

Once installed, verify everything works:

```
bash  
  
claude --version  
claude --help
```

You should see version information and available commands.

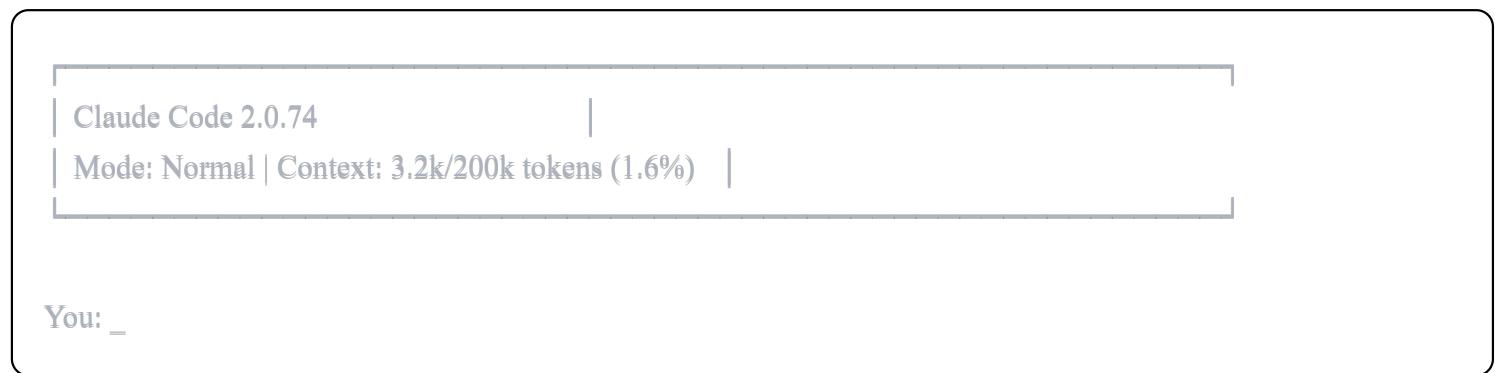
Chapter 3: Your First Conversation

Starting Claude Code

Navigate to your project directory and start Claude:

```
bash  
  
cd ~/my-project  
claude
```

The Interface Tour



Key Elements:

- **Mode indicator** (Normal, Plan, Auto-accept)
- **Context usage** (tracks your token budget)
- **Prompt area** (where you type)

Essential Keyboard Shortcuts

Shortcut	Action
Shift + Tab (2x)	Enter Plan Mode
Shift + Tab	Toggle Auto-accept Mode
Esc	Stop Claude mid-action
Esc (2x)	Open rewind/checkpoint menu
Ctrl/Cmd + R	Search prompt history
Tab	Toggle thinking on/off
#	Add to CLAUDE.md

Your First Task

Let's start simple. Ask Claude to analyze your project:

Analyze this codebase and create a CLAUDE.md file that covers:

1. Tech stack and dependencies
2. Project structure and folder purposes
3. Build/test/run commands
4. Coding conventions you observe
5. Workflow rules I should follow

Claude will scan your project and generate a comprehensive configuration file. This is your first step toward customization.

Understanding the Response

Claude will:

1. **Think** (you'll see the thinking process if enabled)
2. **Use tools** (Read, Glob, Grep to explore your codebase)
3. **Ask clarifying questions** (if needed)
4. **Generate the file** and ask for your approval

Review the generated CLAUDE.md. This file will guide all future interactions.

Part II: The Guide to the Guide

Chapter 4: CLAUDE.md - Your Project's Constitution

What is CLAUDE.md?

CLAUDE.md is your project's instruction manual for Claude Code. Every time you start a session, this file is automatically loaded into context. Think of it as onboarding documentation for a new developer—except this developer has perfect memory and follows instructions precisely.

The Memory Hierarchy

Claude Code discovers and loads memory files in this order (highest to lowest priority):

1. **Enterprise Policy** - `(/Library/Application Support/ClaudeCode/CLAUDE.md)` (macOS)
2. **Project Memory** - `(./CLAUDE.md)` or `(./claude/CLAUDE.md)`
3. **Project Rules** - `(./claude/rules/*.md)`
4. **User Memory** - `(~/.claude/CLAUDE.md)`
5. **User Rules** - `(~/.claude/rules/*.md)`
6. **Project Memory (Local)** - `(./CLAUDE.local.md)` (deprecated)

The Three-Layer Structure

Organize your CLAUDE.md around:

1. **The What** - Tech stack, project structure, key packages
2. **The Why** - Purpose of components, architectural decisions
3. **The How** - How you want Claude to work with your project

Example: .NET Web API Project

markdown

CLAUDE.md - MyApp API

Tech Stack

- .NET 9, ASP.NET Core Minimal APIs
- Entity Framework Core with PostgreSQL
- MediatR for CQRS
- FluentValidation for request validation

Project Structure

- `src/MyApp.Api/` - Entry point, endpoints, middleware
- `src/MyApp.Application/` - Use cases, handlers, DTOs
- `src/MyApp.Domain/` - Entities, value objects, domain events
- `src/MyApp.Infrastructure/` - EF Core, external services

Commands

- Build: `dotnet build`
- Test: `dotnet test`
- Run: `dotnet run --project src/MyApp.Api`

Coding Standards

- Use primary constructors for dependency injection
- Always pass CancellationToken to async methods
- Validation goes in FluentValidation validators, not handlers
- Never hardcode connection strings or secrets

Workflow Rules

- ALWAYS create a git branch before making changes
- Run `dotnet test` after every implementation
- Keep commits atomic and focused

The Import System

Use `@` to pull in other files:

markdown

```
# In your project's CLAUDE.md
@docs/architecture.md
@docs/testing.md
@~/claude/projects/my-app/CLAUDE.md
```

Why imports?

- Keeps CLAUDE.md concise
- Allows personal settings outside the repo
- Works seamlessly with git worktrees (unlike CLAUDE.local.md)

Modular Rules with .claude/rules/

For larger projects, organize instructions into focused files:

```
my-project/
└── .claude/
    ├── CLAUDE.md (main project instructions)
    └── rules/
        ├── code-style.md
        ├── testing.md
        └── security.md
```

All `.md` files in `.claude/rules/` are automatically loaded.

Path-Specific Rules

Scope rules to specific files using YAML frontmatter:

```
markdown

=====
paths:
- "src/api/**/*.ts"

# API Development Rules
- All API endpoints must include input validation
- Use the standard error response format
- Include OpenAPI documentation comments
```

This rule only applies when working on TypeScript files in `src/api/`.

Best Practices for CLAUDE.md

DO:

- Keep it concise (under 500 lines)

- Document non-obvious conventions
- Include common commands
- Specify what NOT to do
- Update it as your project evolves

✗ DON'T:

- Duplicate information Claude can infer
- Include every architectural detail
- Write essays—be direct
- Forget to commit it to version control

Real-World Example

Here's a snippet from my global `(~/.claude/CLAUDE.md)`:

markdown

Personal Preferences (All Projects)

Workflow

- Pay attention to branches when working on a project
- Don't speculate—if you have a theory, verify it
- No assumptions! Ask for more information if needed
- Always run unit tests after making changes

Communication

- Be direct and concise
- Don't apologize unnecessarily
- Explain your reasoning when making decisions

And from a work project:

markdown

Project-Specific Rules

Safety Rails

- Don't git commit/push without explicit approval
- Don't terraform apply without approval
- Don't helm upgrade without approval

Code Quality

- Always run mypy when modifying Python files
- PRs must be associated with a Linear issue
- PR description format: `ref <Linear issue number>`

Quick Setup with /init

Don't want to write CLAUDE.md from scratch? Use the `/init` command:

```
/init
```

Claude will analyze your project and generate a starter CLAUDE.md. You can then refine it based on your preferences.

Managing CLAUDE.md in Monorepos

For monorepos with multiple teams:

```
root/
  └── backend/
      └── my-project/
          ├── CLAUDE.md (or symlink)
          └── src/
  └── frontend/
      └── my-project/
          ├── CLAUDE.md (or symlink)
          └── src/
  └── infra/
      └── my-project/
          ├── CLAUDE.md (or symlink)
          └── src/
```

Create one master CLAUDE.md and symlink it in subdirectories to avoid duplication.

Chapter 5: Commands - Slash Your Way to Productivity

Built-in Slash Commands

Access commands by typing `/` in the prompt:

Command	Purpose
<code>/clear</code>	Reset conversation context
<code>/compact</code>	Compress conversation history
<code>/context</code>	View current token usage
<code>/usage</code>	See API usage statistics
<code>/stats</code>	Session statistics
<code>/resume</code>	Continue previous conversation
<code>/rewind</code>	Open checkpoint menu
<code>/init</code>	Generate CLAUDE.md
<code>/memory</code>	Edit memory files
<code>/plan</code>	View current plan
<code>/permissions</code>	Manage allowed tools
<code>/hooks</code>	Configure workflow hooks
<code>/plugins</code>	Browse plugin marketplace

Creating Custom Commands

Custom commands are reusable prompt templates stored in `.claude/commands/`:

Example: Create a PR Command

File: `.claude/commands/pr.md`

markdown

description: Create a pull request with structured description

Create a pull request for the current branch:

1. Review all uncommitted changes
2. Generate a descriptive title
3. Create a detailed PR description with:
 - Summary of changes
 - Testing performed
 - Breaking changes (if any)
4. Use '`gh pr create`' to submit

Format the description in Markdown.

Now you can type `(/pr)` to trigger this workflow.

Example: Test Command with Arguments

File: `.claude/commands/test.md`

markdown

description: Run tests and analyze failures

Run all tests for the solution and report any failures.

If tests fail, analyze the error and suggest fixes.

\$ARGUMENTS

Usage: `(/test --verbose)` (the `($ARGUMENTS)` token passes any additional text)

Useful Custom Commands

1. Code Review

markdown

```
# .claude/commands/review.md
```

description: Perform comprehensive code review

Review the staged changes for:

1. TypeScript/type safety issues
2. Potential bugs or edge cases
3. Performance concerns
4. Security vulnerabilities
5. Code style and readability

Provide specific line references and actionable feedback.

2. Migration Helper

markdown

```
# .claude/commands/migrate.md
```

description: Generate and apply EF Core migration

1. Generate an EF Core migration with name: \$ARGUMENTS
2. Review the generated migration
3. Apply it to the local database
4. Verify with a test query

3. Dependency Update

markdown

```
# .claude/commands/update-deps.md  
=====  
description: Update project dependencies safely  
=====
```

1. List all outdated packages
2. Check for breaking changes in release notes
3. Update non-breaking changes automatically
4. Flag breaking changes for manual review
5. Run tests after updates

Command Organization

Project-level commands: (.claude/commands/) (shared with team)

Personal commands: (~/.claude/commands/) (just for you)

Commit project commands to version control so your team benefits.

When to Create a Command

Create a custom command when you:

- Repeat the same prompt 3+ times
- Have a multi-step workflow
- Need consistent formatting
- Want to enforce team standards

Don't create commands for:

- One-off tasks
- Exploratory prompts
- Questions that vary significantly each time

Chapter 6: Plan Mode - Think Before You Code

The Problem with Direct Execution

Developers (and AIs) often jump straight to implementation without planning. This leads to:

- Solving the wrong problem
- Missing edge cases
- Inconsistent architecture
- Wasted effort refactoring later

Enter Plan Mode

Plan Mode forces Claude to **think before it acts**. In Plan Mode, Claude can only:

- **Read files**
- **Search and grep** through code
- **Browse the web** for documentation
- **Ask clarifying questions**

It **cannot** write, modify, or execute anything.

How to Enter Plan Mode

Press **Shift + Tab** twice (or **Alt + M** on some Windows setups). You'll see the mode indicator change:

Mode: Plan | Context: 5.2k/200k tokens (2.6%)

When to Use Plan Mode

Use Plan Mode for:

- Starting a new feature with multiple components
- Making architectural changes
- Working on an unfamiliar codebase
- Complex refactoring across many files

Skip Plan Mode for:

- Fixing typos
- Adding log lines
- Renaming variables
- Simple, well-defined tasks

The Plan Mode Workflow

1. Describe Your Goal

Build a chore management app with:

- Calendar view similar to Outlook
- Add/remove chores
- Recurring schedules
- Assign chores to team members
- Manage team members

Ask me questions to clarify requirements.

2. Claude Asks Questions

Claude: Great! A few questions to get started:

1. What framework should I use? React, Vue, or something else?
2. Local storage or a backend database?
3. Do you need user authentication?
4. Any specific UI library preferences?
5. Should this be a web app, desktop app, or both?

3. You Answer

1. React with TypeScript
2. Start with local storage, design for future backend
3. No auth for v1
4. Use Tailwind CSS + shadcn/ui
5. Web app (responsive)

4. Claude Generates a Plan

Claude: Excellent! Here's my implementation plan:

Phase 1: Project Setup

- Create React + TypeScript project with Vite
- Configure Tailwind CSS + shadcn/ui
- Set up folder structure:
 - /components (UI components)
 - /hooks (custom React hooks)
 - /store (state management)
 - /types (TypeScript definitions)

Phase 2: Data Models

- Team Member: { id, name, avatar? }
- Chore: { id, title, description, assigneeId, schedule: { type, interval, dates }, completed: boolean }
- RecurrenceRule: { frequency, interval, until }

Phase 3: Core Components

1. Calendar view (using react-big-calendar)
2. Chore form (add/edit)
3. Team member management
4. Recurring schedule builder

Phase 4: State Management

- Use Zustand for global state
- Local storage persistence
- Future: API abstraction layer

Phase 5: Testing & Polish

- Unit tests for critical logic
- Responsive design verification
- Accessibility audit

Should I proceed with this plan?

5. Approve or Adjust

You: Looks good, but use react-calendar instead of react-big-calendar. It's lighter weight.

Claude: Updated! I'll use react-calendar. Ready to implement?

You: Yes, proceed.

6. Exit Plan Mode and Execute

Press `Shift + Tab` again to exit Plan Mode. Claude now implements the plan you approved.

Advanced: Plan to Document

For very complex features, have Claude write the plan to a file:

In Plan Mode:

"Create the implementation plan and save it to PLAN.md"

Then /clear and start fresh:

"Read PLAN.md and implement Phase 1"

This keeps your context clean for execution.

Plan Mode Anti-Patterns

✗ Micro-planning simple tasks

Bad: "Plan how to add a console.log statement"

Good: Just do it directly

✗ Endless planning without execution

Bad: Spending 20 minutes refining a plan

Good: Get a solid plan, then iterate during implementation

✗ Ignoring Claude's questions

Bad: "Just use your best judgment"

Good: Answer specifically—your preferences matter

Part III: Advanced Techniques

Chapter 7: Sub-agents - Delegating Like a Pro

What Are Sub-agents?

Sub-agents are separate Claude instances spawned by the main agent for isolated tasks. They:

- Run in their own context window
- Have their own tool permissions
- Report back summaries to the main agent
- Prevent context bloat

Built-in Sub-agent Types

Claude Code comes with several pre-configured sub-agents:

1. Explore (Read-only file search)

- Tools: Glob, Grep, Read, limited Bash
- Best for: Searching large codebases
- Context: Fresh (doesn't inherit conversation)

2. Plan (Software architect)

- Tools: All tools
- Best for: Designing implementation strategies
- Context: Inherits full context

3. General-purpose

- Tools: All tools
- Best for: Complex research, multi-step tasks
- Context: Inherits full context

4. claude-code-guide (Documentation lookup)

- Tools: Glob, Grep, Read, WebFetch, WebSearch
- Best for: Questions about Claude Code features

How Sub-agents Work

Sub-agents are spawned via the `Task` tool:

```
typescript

{
  description: "Search for authentication code",
  prompt: "Find all files related to user authentication",
  subagent_type: "Explore",
  model: "haiku", // Optional: sonnet, opus, haiku
  run_in_background: false
}
```

Using Sub-agents Effectively

Manual Invocation:

You: Use an Explore sub-agent with Sonnet to find all API endpoints in this codebase.

Claude will spawn the sub-agent, which explores independently and returns a summary.

Automatic Invocation:

Claude often spawns sub-agents automatically when:

- You ask to search for something specific
- A task requires reading many files
- Investigation would bloat the main context

The Context Isolation Benefit

Problem: Searching a large codebase reads hundreds of files, filling your context window with irrelevant details.

Solution: Sub-agents do the searching in their own context, return only the relevant summary.

Main Agent: 3.5k tokens used

↓ spawns

Explore Sub-agent: 45k tokens used (searching 200 files)

↓ returns

Main Agent: 4.2k tokens used (summary added)

Background Agents for Monitoring

Launch long-running processes in the background:

You: Launch a background agent to monitor the build logs
and alert me if any errors occur.

The agent runs asynchronously, notifying you only on failures.

Creating Custom Sub-agents

Define custom sub-agents in `(.claude/agents/)`:

Example: Security Reviewer

File: `(.claude/agents/security-reviewer.md)`

markdown

`name: security-reviewer`

`description: Reviews code for security vulnerabilities`

`tools: Read, Grep, Glob, Bash`

`model: opus`

You are a senior security engineer. Review code for:

- Injection vulnerabilities (SQL, XSS, command injection)
- Authentication and authorization flaws
- Secrets or credentials in code
- Insecure data handling

Provide specific line references and suggested fixes.

Usage:

You: Use a sub-agent to review this code for security issues.

Sub-agent Best Practices

DO:

- Use Explore for broad codebase searches
- Use background agents for long-running tasks
- Specify model (haiku for speed, opus for quality)
- Let sub-agents work in parallel when possible

DON'T:

- Spawn sub-agents for simple file reads
- Use sub-agents when context isn't an issue
- Chain too many sub-agent levels (max 2-3 deep)

Chapter 8: Skills & Hooks - Customization Mastery

Skills: On-Demand Expertise

Skills are folders containing instructions, scripts, and resources that Claude loads dynamically when relevant.

Structure:

```
.claude/skills/my-skill/
├── SKILL.md (required - instructions)
├── resources/ (optional - supporting files)
├── scripts/ (optional - executable code)
└── templates/ (optional - template files)
```

Example: API Conventions Skill

File: `.claude/skills/api-conventions/SKILL.md`

markdown

```
---  
name: api-conventions  
description: REST API design conventions for our services  
---
```

API Conventions

- Use kebab-case for URL paths
- Use camelCase for JSON properties
- Always include pagination for list endpoints
- Version APIs in the URL path (/v1/, /v2/)
- Standard error response format:

```
```json  
{
 "error": {
 "code": "ERROR_CODE",
 "message": "Human-readable message",
 "details": {}
 }
}
```
```

How It Works:

1. Skill metadata is loaded into system prompt
2. When relevant, Claude reads the full SKILL.md
3. Domain expertise is available without bloating context
4. Claude can invoke scripts defined in the skill

Skills vs. CLAUDE.md

Use CLAUDE.md for

Use Skills for

Always-on context

On-demand expertise

Project essentials

Domain-specific knowledge

Simple instructions

Complex workflows

Universal rules

Conditional patterns

Hooks: Workflow Automation

Hooks run custom actions at specific points in Claude's workflow.

Available Hook Points:

- `PreToolUse` - Before Claude uses a tool
- `PostToolUse` - After a tool completes
- `Stop` - When Claude finishes a task
- `UserPromptSubmit` - When you submit a prompt

Example: Auto-format on File Edit

File: `.claude/hooks/post-edit-format.sh`

```
bash

#!/bin/bash

# PostToolUse hook: Run Prettier after file edits

if [[ "$TOOL_NAME" == "Edit" ]]; then
    FILE="$TOOL_ARG_PATH"

    if [[ "$FILE" =~ \.(ts|tsx|js|jsx)$ ]]; then
        npx prettier --write "$FILE"
        echo "✓ Formatted $FILE"
    fi
fi
```

Configure in `.claude/settings.json`:

```
json

{
  "hooks": {
    "PostToolUse": ".claude/hooks/post-edit-format.sh"
  }
}
```

Example: Test Before Commit

```
bash
```

```
#!/bin/bash
```

```
# PreToolUse hook: Block commits if tests fail
```

```
if [[ "$TOOL_NAME" == "Bash" ]] && [[ "$TOOL_ARGS" =~ "git commit" ]]; then
    if [ ! -f /tmp/tests-passed ]; then
        echo "✗ Tests have not passed. Run tests first."
        exit 1
    fi
    rm /tmp/tests-passed
fi
```

Now Claude cannot commit until tests pass.

Combining Skills, Hooks, and Reminders

Scenario: You want Claude to always consider security when modifying authentication code.

1. Create a skill:

```
markdown
```

```
# .claude/skills/security/SKILL.md
```

```
---
```

```
name: security-best-practices
```

```
description: Security guidelines for authentication code
```

```
paths:
```

```
- "src/auth/**/*.ts"
```

```
---
```

Security Guidelines

- Never log passwords or tokens
- Always hash passwords with bcrypt
- Use secure session cookies (HttpOnly, Secure, SameSite)
- Implement rate limiting on login endpoints

2. Add a hook:

```
bash
```

```
# .claude/hooks/security-reminder.sh
# PreToolUse hook

if [[ "$FILE_PATH" =~ src/auth/ ]]; then
    echo "⚠️ You're modifying authentication code."
    echo "Security skill has been loaded."
fi
```

3. Result:

When Claude works on `src/auth/`, the skill auto-loads and the hook reminds you. Security guidelines are automatically considered.

Chapter 9: Context Engineering - Making Every Token Count

Understanding the Context Window

Claude Code has a **200,000 token context window**. For perspective:

- 200k tokens ≈ 150,000 words
- The entire Harry Potter series ≈ 1.08 million words (would need ~7 context windows)

But here's the catch: Performance degrades as context fills. At 60%+ usage, Claude starts "forgetting" earlier instructions.

What Consumes Context?

Every session starts with:

- **System prompt:** ~3.1k tokens
- **CLAUDE.md files:** varies (keep under 500 tokens)
- **Tool definitions:** ~29k tokens
- **MCP tools:** varies (can be significant)

Then during conversation:

- **Your messages:** varies
- **Claude's responses:** varies
- **Tool call results:** can be large (file contents, search results)

Monitoring Context Usage

Check anytime with `/context`:

Context Usage: 45k/200k tokens (22.5%)

- └─ System prompt: 3.0k tokens (1.5%)
- └─ System tools: 29.0k tokens (14.5%)
- └─ MCP tools: 8.1k tokens (4.1%)
- └─ Custom agents: 2.0k tokens (1.0%)
- └─ Messages: 2.9k tokens (1.5%)

Memory files:

- └ CLAUDE.md: 388 tokens

Context Management Strategies

1. Clear Regularly

Use `/clear` between unrelated tasks:

```
/clear
```

Your CLAUDE.md stays loaded, but conversation resets.

2. Compact Instead of Clear

For complex ongoing work, use `/compact`:

```
/compact
```

Claude summarizes the conversation, preserving key decisions and code state.

3. Handoff Pattern for Long Tasks

When approaching 60% context:

You: Before we continue, document our progress in PROGRESS.md:

- What we've completed
- Current issues
- Next steps

Claude: [Creates PROGRESS.md]

You: /clear

You: Read PROGRESS.md and continue from where we left off.

4. Use Sub-agents for Investigation

Instead of cluttering main context with exploration:

You: Use an Explore sub-agent to find all database migrations
and summarize the schema changes.

The sub-agent reads dozens of files, but your main context only receives the summary.

5. Strategic File Reading

Don't read entire files when you only need snippets:

✗ Wasteful:

You: Read all files in src/services/ and explain the architecture.

(Loads 50 files × 500 lines = 25k tokens)

✓ Efficient:

You: Use Glob to list files in src/services/, then read only
the interfaces and main class definitions.

(Loads interfaces only = 3k tokens)

6. Keep CLAUDE.md Lean

Every token in CLAUDE.md counts against every session.

Bad CLAUDE.md (bloated):

markdown

```
# CLAUDE.md - 2000 lines of documentation  
# (loads 1500 tokens every session)
```

Good CLAUDE.md (lean with imports):

markdown

```
# CLAUDE.md - 50 lines of essentials  
# Reference detailed docs:  
# See docs/architecture.md for system design  
# See docs/api-conventions.md for API patterns  
# (loads 388 tokens, pulls docs on-demand)
```

Context Optimization Checklist

- Monitor with `/context` regularly
- Clear between unrelated tasks
- Use sub-agents for exploration
- Compact at 60% usage
- Keep CLAUDE.md under 500 tokens
- Use handoff pattern for long sessions
- Remove unused MCP servers

Part IV: Practical Matters

Chapter 10: Pricing & Cost Management

The Brutal Truth About Costs

API credits burn faster than you think. I learned this the hard way: **\$20 disappeared in 20 minutes** during a serious refactoring session.

Here's what happened:

- Complex codebase (200+ files)
- Multiple sub-agents spawned for exploration
- Heavy use of extended thinking
- Iterative debugging with full context

Each cycle consumed massive tokens. The result? API credits evaporated.

Claude Subscription Plans (2026)

| Plan | Monthly Cost | Claude Code Access | Usage Limit | Best For |
|---------|--------------|---|---------------------------------|------------------------------------|
| Free | \$0 | <input checked="" type="checkbox"/> No | --- | Trying claude.ai only |
| Pro | \$20 | <input checked="" type="checkbox"/> Yes | ~45 messages per 5-hour window | Individual developers, light usage |
| Max 5x | \$100 | <input checked="" type="checkbox"/> Yes | ~225 messages per 5-hour window | Heavy individual usage |
| Max 20x | \$200 | <input checked="" type="checkbox"/> Yes | ~900 messages per 5-hour window | All-day intensive coding |
| Teams | Variable | <input checked="" type="checkbox"/> Yes | Centralized billing | Organizations |

Important: These are message limits, not token limits. Complex messages with large codebases consume your allocation faster.

What Happens When You Hit the Limit?

1. **Warning:** Claude alerts you as you approach the limit
2. **Throttling:** Requests slow down or pause
3. **Wait:** Your allocation refreshes every 5 hours
4. **No overage fees:** Fixed allocation, no surprise bills

API Usage for Advanced Users

If you're using Claude Code via API (with your own API key):

Pricing (as of January 2026):

- **Input tokens:** \$3 per million tokens
- **Output tokens:** \$15 per million tokens
- **Cached input:** \$0.30 per million tokens (90% discount!)

Example calculation:

Session with 50k input tokens + 10k output tokens:

$$= (50k \times \$3/1M) + (10k \times \$15/1M)$$

$$= \$0.15 + \$0.15$$

$$= \$0.30 \text{ per session}$$

Heavy day (30 sessions): \$9

Month (20 work days): \$180

With prompt caching enabled:

First session: \$0.30

Subsequent sessions (80% cached):

$$= (10k \times \$3/1M) + (40k \times \$0.30/1M) + (10k \times \$15/1M)$$

$$= \$0.03 + \$0.012 + \$0.15$$

$$= \$0.192 \text{ per session}$$

Heavy day: \$5.86

Month: \$117

Prompt caching saves ~35% on repeated context.

Alternative: Z.ai for Open Models

Z.ai provides access to models like GLM-4.7 at significantly lower costs:

| Provider | Model | Input Cost | Output Cost | Quality |
|-----------|-------------|------------|-------------|-----------|
| Anthropic | Opus 4.5 | \$3/1M | \$15/1M | Excellent |
| Z.ai | GLM-4.7 | \$0.50/1M | \$1.50/1M | Good |
| Z.ai | DeepSeek V3 | \$0.27/1M | \$1.10/1M | Good |

Trade-offs:

- 80-90% cost savings
- Good for simple tasks
- Lower reasoning quality

- ✗ Weaker at complex architecture
- ✗ May require more iterations

My recommendation: Use cheaper models for:

- Boilerplate generation
- Simple refactoring
- Documentation writing
- Test generation

Use Opus 4.5 for:

- Complex debugging
- Architectural decisions
- Multi-file reasoning
- Critical implementations

Cost Optimization Strategies

1. Start Small

Begin with Claude Pro (\$20/month). Upgrade only when you consistently hit limits.

2. Batch Your Work

Group tasks into focused sessions rather than spreading them throughout the day.

3. Use Plan Mode First

Planning is cheaper than executing wrong implementations and refactoring.

4. Monitor Context Usage

Run `/context` regularly. Clear when approaching 60%.

5. Leverage Prompt Caching

If using API directly, enable prompt caching in your configuration.

6. Strategic Model Selection

Use the right model for the task:

- **Haiku** for simple tasks (faster, cheaper)

- **Sonnet** for balanced performance
- **Opus** for complex reasoning

7. Avoid Repeated Explorations

Document findings in CLAUDE.md so Claude doesn't re-explore the same code paths.

8. Use Sub-agents Wisely

Sub-agents are great for context isolation, but they still consume tokens. Don't spawn unnecessarily.

Cost Monitoring Tools

Built-in Commands:

```
/usage # See your current usage
/stats # Session statistics
/context # Token consumption breakdown
```

External Tools:

For API usage, use services like:

- **Helicone** (API observability)
- **LangSmith** (debugging and monitoring)
- **Custom dashboards** with Anthropic's API usage endpoints

The Enterprise Equation

For teams, calculate the real cost:

Developer hourly rate: \$100

Time saved per day: 2 hours

Value created: \$200/day

Claude Max 20x cost: \$200/month

Work days: 20

Daily cost: \$10

ROI: $(\$200 - \$10) \times 20 = \$3,800/\text{month value}$

Even at \$200/month, the productivity gains vastly exceed the cost.

My Personal Cost Management

Here's what I do:

1. **Claude Max 5x subscription** (\$100/month)
2. **Monitor with /usage daily**
3. **Clear context between tasks** (saves tokens)
4. **Use Plan Mode for complex features** (avoid wasted execution)
5. **Keep CLAUDE.md lean** (reduces baseline token usage)
6. **Batch serious coding** into focused sessions
7. **Use ChatGPT for brainstorming** (different budget)

Result: Rarely hit my limits, massive productivity boost.

Chapter 11: Integrations & Ecosystem

The Three Interfaces

Claude Code isn't locked to one interface. You have flexibility:

1. CLI (Command Line Interface)

Best for: Developers who live in the terminal

Pros:

- Lightweight and fast
- Direct integration with your workflow
- Scriptable and automatable
- Works over SSH

Cons:

- Steeper learning curve
- Less visual feedback
- Terminal-only interface

Installation:

```
bash
```

```
npm install -g @anthropic-ai/clause-code  
clause
```

2. Desktop App

Best for: Visual preference, window management

Pros:

- Dedicated window for Claude
- Better syntax highlighting
- Easier file attachments
- Native OS integration

Cons:

- Higher memory usage
- Can get sluggish in long sessions
- Separate from your IDE

Installation:

```
bash
```

```
# macOS
```

```
brew install --cask clause-code
```

```
# Windows
```

```
# Download from clause.ai/download
```

3. Cloud/Mobile (clause.ai)

Best for: Brainstorming, on-the-go access

Pros:

- Access from anywhere
- Mobile app available
- No installation required

- Sync across devices

Cons:

- No direct file system access
- Limited to web interface
- Requires internet connection

Access: Simply visit claude.ai

IDE Integrations

VS Code Integration

While Claude Code doesn't have an official VS Code extension, you can integrate it seamlessly:

Method 1: Integrated Terminal

1. Open VS Code
2. Open integrated terminal (Ctrl + `)
3. Run: claude

You now have Claude Code running alongside your code editor.

Method 2: Split View

1. Open a separate terminal window
2. Run: claude
3. Arrange windows side-by-side with VS Code

Method 3: Custom VS Code Task

Create `.vscode/tasks.json`:

json

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Launch Claude Code",
      "type": "shell",
      "command": "claude",
      "problemMatcher": [],
      "presentation": {
        "reveal": "always",
        "panel": "dedicated"
      }
    }
  ]
}
```

Now run with: Terminal > Run Task > Launch Claude Code

Cursor Integration

Cursor and Claude Code can work together:

Complementary Usage:

- Use **Cursor** for inline completions and quick edits
- Use **Claude Code** for architectural planning and complex refactoring

Workflow Example:

1. Use Claude Code in terminal for planning:

"Plan the authentication system"

2. Switch to Cursor for implementation:

(Cursor handles inline suggestions)

3. Return to Claude Code for review:

"Review the authentication implementation"

Both tools read from the same codebase, but serve different purposes.

JetBrains IDEs (IntelliJ, PyCharm, etc.)

Similar to VS Code, use the integrated terminal:

1. View > Tool Windows > Terminal
2. Run: claude

MCP Server Ecosystem

Model Context Protocol (MCP) lets Claude Code connect to external services.

Popular MCP Servers:

1. Filesystem MCP

```
bash
claude mcp add filesystem -- npx @anthropic-ai/mcp-server-filesystem /path/to/project
```

Grants Claude controlled file system access.

2. GitHub MCP

```
bash
claude mcp add github -- npx @modelcontextprotocol/server-github
```

Access repositories, issues, PRs directly.

3. Playwright MCP

```
bash
claude mcp add playwright -- npx @playwright/mcp@latest
npx playwright install chromium
```

Browser automation for testing UIs.

4. Google Drive MCP

```
bash
claude mcp add gdrive -- npx @modelcontextprotocol/server-gdrive
```

Access documents, sheets, and files.

5. Notion MCP

```
bash
```

```
claude mcp add notion -- npx @modelcontextprotocol/server-notion
```

Read and write Notion databases.

Managing MCP Servers:

List connected servers:

```
bash
```

```
claude mcp list
```

Remove a server:

```
bash
```

```
claude mcp remove github
```

Debug MCP connections:

```
bash
```

```
claude --mcp-debug
```

Git Worktrees with Claude Code

Problem: Working on multiple features simultaneously in separate contexts.

Solution: Git worktrees + multiple Claude Code instances.

Setup:

```
bash
```

```

# Main branch in primary directory
cd ~/my-project

# Create worktree for feature-A
git worktree add ../my-project-feature-a feature-a

# Create worktree for feature-B
git worktree add ../my-project-feature-b feature-b

# Open separate terminal tabs
# Tab 1: cd ~/my-project && claude
# Tab 2: cd ~/my-project-feature-a && claude
# Tab 3: cd ~/my-project-feature-b && claude

```

Each worktree has isolated:

- Working directory
- Branch
- Claude Code session

But they share:

- Git history
- CLAUME.md (if using imports from home directory)

CI/CD Integration

GitHub Actions with Claude Code:

yaml

```

# .github/workflows/clause-review.yml
name: Claude Code Review

on:
  pull_request:
    types: [opened, synchronize]

jobs:
  clause-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install Claude Code
        run: npm install -g @anthropic-ai/clause-code

      - name: Run Claude Review
        env:
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
        run:
          claude -p "Review this PR for:
          1. Code quality issues
          2. Potential bugs
          3. Security vulnerabilities
          4. Performance concerns

          Output findings in markdown." > review.md

      - name: Comment PR
        uses: actions/github-script@v6
        with:
          script:
            const fs = require('fs');
            const review = fs.readFileSync('review.md', 'utf8');
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,

```

```
repo: context.repo.repo,  
body: review  
});
```

Obsidian Integration

For knowledge management, integrate Claude Code with Obsidian:

Setup:

1. Create Obsidian vault for project notes
2. Add CLAUDE.md to vault root:

markdown

Project Knowledge Base

This Obsidian vault contains:

- Research notes in /research
- Design decisions in /decisions
- Architecture docs in /architecture

For Claude Code

When working on this project:

- Check /decisions before making architectural changes
- Update /research with findings
- Document patterns in /patterns

3. Run Claude Code from vault directory:

bash

```
cd ~/ObsidianVault/MyProject  
claude
```

Claude can now:

- Read your research notes
- Update documentation
- Create new notes
- Link related concepts

Workflow:

You: Analyze the competitor research in /research and create a feature comparison table in /analysis/competitors.md

Claude: [Reads all research notes, creates structured comparison]

You: Based on this analysis, what features should we prioritize?

Claude: [Provides recommendations based on the research]

Docker & Containers

Run Claude Code in isolated environments:

Dockerfile:

```
dockerfile
FROM node:18

# Install Claude Code
RUN npm install -g @anthropic-ai/clause-code

# Set up project
WORKDIR /app
COPY . .

# Set API key (pass at runtime)
ENV ANTHROPIC_API_KEY=""

CMD ["clause"]
```

Usage:

```
bash
```

```
docker build -t my-project-claude .
docker run -it \
-e ANTHROPIC_API_KEY=$ANTHROPIC_API_KEY \
-v $(pwd):/app \
my-project-claude
```

Why use Docker?

- Consistent environment across team
 - Isolated dependencies
 - Safe experimentation
 - Easy CI/CD integration
-

Chapter 12: Best Practices from Power Users

After months of intensive use and learning from the community, here are the patterns that separate effective Claude Code users from beginners.

1. Treat Claude Like a Senior Engineer

Bad:

```
You: Can you maybe help me with adding validation  
if that's okay?
```

Good:

```
You: Add FluentValidation to CreateUserCommand.  
Validate: Email (required, valid format),  
Name (required, max 100 chars).
```

Claude responds to clear, specific instructions. Be direct.

2. Commit Early, Commit Often

The Pattern:

```
bash
```

```
# After each successful change:  
git add .  
git commit -m "feat: implement user authentication"  
  
# If Claude breaks something:  
git diff HEAD~1 # See what changed  
git checkout HEAD~1 -- src/file.cs # Restore specific file
```

AI can introduce regressions. Commits are your safety net.

3. Always Review the Code

Don't blindly accept suggestions.

I've seen Claude:

- Add dependencies I didn't ask for
- Use patterns inconsistent with my architecture
- Write code that works but isn't idiomatic

Review like you would any pull request:

1. Read the diff
2. Understand the changes
3. Test the functionality
4. Push back if needed

4. Use Plan Mode for Non-Trivial Tasks

If a task touches 3+ files or involves architectural decisions, start with Plan Mode.

Bad workflow:

```
You: Build an authentication system  
Claude: [Jumps straight to code, makes assumptions]  
You: No, that's not what I wanted...
```

Good workflow:

[Enter Plan Mode]

You: Build an authentication system

Claude: [Asks clarifying questions, proposes plan]

You: [Reviews plan, provides feedback]

[Exit Plan Mode]

Claude: [Implements approved plan]

5. Invest in Your CLAUDE.md

Spend time crafting a good CLAUDE.md. Every minute invested saves hours of repetition.

Example refinement process:

Week 1:

markdown

CLAUDE.md

- Use .NET 9
- Write tests

Week 2: (After noticing Claude uses wrong patterns)

markdown

CLAUDE.md

- Use .NET 9, ASP.NET Core Minimal APIs
- Write tests with xUnit, not NUnit
- Use FluentAssertions for assertions

Week 4: (After more refinements)

markdown

CLAUDE.md

Stack

- .NET 9, ASP.NET Core Minimal APIs
- Entity Framework Core with PostgreSQL
- xUnit + FluentAssertions for testing

Patterns We Use

- MediatR for CQRS
- Result<T> pattern for error handling

Patterns We DON'T Use

- Repository pattern (use EF Core directly)
- AutoMapper (write explicit mappings)

Workflow

- Create git branch before changes
- Run tests after every implementation

The file evolves with your project.

6. Don't Be Afraid to Say "No"

Claude will suggest approaches. Sometimes they're wrong.

Example:

Claude: I'll implement this using the repository pattern.

You: No, we don't use repository pattern. Use EF Core directly as specified in CLAUDE.md.

Claude: You're right, my apologies. I'll use EF Core directly with DbContext.

Push back confidently. Claude adapts.

7. Strategic Context Management

The handoff pattern for complex features:

Phase 1: Setup (Context: 15%)

You: Set up the project structure

Claude: [Creates folders, config files]

You: /clear

Phase 2: Models (Context: 20%)

You: Read CLAUDE.md, then create domain models

Claude: [Implements models]

You: git commit && /clear

Phase 3: Business Logic (Context: 25%)

You: Read CLAUDE.md, then implement handlers

Claude: [Implements handlers]

You: git commit && /clear

Phase 4: API Endpoints (Context: 20%)

You: Read CLAUDE.md, then create API endpoints

Claude: [Implements endpoints]

Each phase starts fresh, preventing context rot.

8. Use Background Agents for Monitoring

Long-running tasks benefit from background monitoring:

You: Launch a background agent to monitor the test suite and alert me if any test fails.

[Continue working on other features]

Claude (background): ⚠️ Test failure detected in
UserServiceTests.CreateUser_ShouldValidateEmail

9. Learn from Mistakes

When Claude makes a mistake, ask why:

You: You used AutoMapper, but CLAUDE.md says not to.

Why did this happen?

Claude: You're correct. I overlooked the "Patterns We DON'T Use" section in CLAUDE.md. This happened because I defaulted to common .NET patterns without checking your specific constraints. I'll be more careful to reference CLAUDE.md before making architectural decisions.

Understanding failure modes improves future interactions.

10. The 80/20 Rule

80% of value comes from 20% of features:

Essential 20%:

- CLAUDE.md configuration
- Plan Mode for complex tasks
- /clear between unrelated work
- Review all generated code
- Commit frequently

Nice-to-have 80%:

- Custom commands
- Hooks
- MCP servers
- Sub-agents
- Advanced workflows

Master the essentials first.

11. Multi-Tool Strategy

Don't rely on Claude Code alone:

My toolkit:

- **Claude Code:** Complex implementations, debugging, architecture

- **GitHub Copilot:** Quick autocomplete, boilerplate
- **ChatGPT:** Brainstorming, learning new concepts
- **Cursor:** Large refactorings, IDE comfort

Each tool excels at different tasks.

12. The "Throwaway First Draft" Technique

For complex features with uncertain requirements:

Step 1: Create feature branch

You: Build the entire feature end-to-end

Step 2: Review Claude's interpretation

[Identify what it got right/wrong]

Step 3: Create new branch

You: Now implement it correctly, incorporating
what we learned

Step 4: Compare approaches

git diff feature-draft feature-final

[Understand Claude's biases and assumptions]

This builds intuition about Claude's decision-making.

Part V: The Reality Check

Chapter 13: The Open Source Experiment - What I Learned

The Setup

This morning, I ran an experiment: **Use Claude Code with an open-source model instead of Claude's API.**

The hypothesis: Can the Claude Code framework work with cheaper, local models?

The test:

- **Model:** qwen2.5-coder:14b via Ollama

- **Task:** Generate a CLAUDE.md file using `/init`
- **Goal:** See if open-source models can power agentic workflows

The Fighter Jet Metaphor

Claude Code is like a sophisticated fighter jet—precise controls, powerful capabilities, everything optimized for performance.

The model you use? That's your pilot.

You can have the most advanced aircraft in the world, but if your pilot doesn't have the training, experience, and reflexes to operate it effectively, you're not going anywhere.

What Happened

✓ What Worked:

- The model understood the task conceptually
- It outlined a reasonable 4-step plan
- It called the right tools (Glob, Read, Grep)

✗ What Didn't:

- Execution spun for **1 minute 14 seconds** and produced nothing
- When I asked "what's going on?", generic responses
- When I asked "did you create CLAUDE.md?", it said no
- Basic commands like "analyze @my-project/" just hung

The Technical Reality

Claude Code infrastructure provides:

- ✓ Tool integrations
- ✓ Workflow orchestration
- ✓ Command parsing
- ✓ Context management

But the reasoning layer—the model—determines whether:

- Tools actually get used correctly
- Context is maintained across turns

- Tasks are completed vs. abandoned
- Plans translate to execution

Why Open-Source Models Struggle

1. Reasoning Gaps

Opus 4.5 thinks in multiple dimensions:

- What tools to call
- What order to call them
- When to stop
- How to handle failures
- How to maintain context across turns

qwen2.5-coder:14b tried, but:

- Called tools then forgot why
- Lost track of the task mid-execution
- Couldn't recover from errors

2. Context Retention

In agentic workflows, the model must remember:

- The original goal
- What it's already tried
- What worked/didn't work
- Where it is in the plan

Open-source models struggle to maintain this state across 10+ tool calls.

3. Task Completion

Frontier models (Opus, GPT-4) have been trained on:

- Following instructions to completion
- Recognizing when a task is done
- Asking for help when stuck

Open-source models often:

- Abandon tasks mid-flight
- Loop without progress
- Give up silently

The Cost-Performance Trade-off

Anthropic Opus 4.5:

- Excellent reasoning
- Reliable task completion
- Strong context retention
- \$3-15 per million tokens

qwen2.5-coder:14b via Ollama:

- Free (local)
- Good for simple completions
- Weak agentic capabilities
- Unreliable execution

The verdict: For simple, one-shot tasks, open-source works. For complex, multi-step agentic workflows, you need frontier models.

Testing with OpenRouter (OpenAI OSS)

I also tried Claude Code with **OpenRouter**, routing to various open-source models:

Models tested:

- Meta-Llama-3.1-70B-Instruct
- Mistral-7B-Instruct
- CodeLlama-34B

Results:

- Better than Ollama (cloud-hosted, optimized)
- Handled simple tasks (file reading, basic refactoring)
- Still struggled with multi-step workflows

- ✗ Context degradation after 5-7 turns
- ✗ Unreliable tool calling

Example failure:

Task: "Refactor this API to use async/await"

Model behavior:

1. Read the file ✓
2. Propose changes ✓
3. Started writing changes ✓
4. Got confused mid-edit ✗
5. Produced incomplete code ✗
6. Didn't recognize it failed ✗

What This Means for AI Development

1. Frameworks ≠ Intelligence

Great tooling doesn't compensate for model limitations. Claude Code is brilliantly designed, but it can't make a weak model strong.

2. Agentic Workflows Amplify Gaps

Simple prompt-response tasks might work fine with smaller models. Multi-step agentic workflows expose every weakness in:

- Reasoning ability
- Context retention
- Task completion
- Error recovery

3. The Open-Source Path is Still Valuable

This isn't a criticism of open models—it's recognition that cutting-edge agentic tools are designed for cutting-edge models.

As open models improve, they'll unlock these frameworks' potential.

Current trajectory:

- 2024: Open models lag 12-18 months behind frontier

- 2025: Gap narrowing to 6-12 months
- 2026: Some open models approaching frontier (e.g., DeepSeek R1)

4. Tool Design Matters

Claude Code works beautifully with its intended "pilot." That's not a bug—it's impressive product-model co-design.

The lesson for builders: **your UX is only as good as your model's ability to execute it.**

My Recommendations

For serious development work:

- Use Opus 4.5 or GPT-4 Turbo
- The cost is worth the reliability
- Your time is more valuable than API credits

For learning/experimentation:

- Try open-source models
- Understand their limitations
- Build intuition about capabilities

For production:

- Don't cheap out on the model
- Unreliable AI is worse than no AI
- Failed executions waste more money than expensive models

The Takeaway

Choose your pilot wisely. ✈️

The Claude Code framework is exceptional. But without a capable model driving it, you're grounded.

Chapter 14: Quick Reference & Cheat Sheet

Essential Commands

bash

Installation

`npm install -g @anthropic-ai/clause-code`

`clause --version`

Launch

`clause` *# Start in current directory*

`clause --resume` *# Resume previous conversation*

`clause --continue` *# Continue most recent conversation*

Session management

`/clear` *# Reset conversation context*

`/compact` *# Compress conversation history*

`/context` *# View token usage*

`/usage` *# View API usage*

`/stats` *# Session statistics*

Configuration

`/init` *# Generate CLAUDE.md*

`/memory` *# Edit memory files*

`/permissions` *# Manage allowed tools*

`/hooks` *# Configure hooks*

`/plugins` *# Browse plugins*

Navigation

`/rewind` *# Open checkpoint menu*

`Esc` *# Stop Claude mid-action*

`Esc + Esc` *# Rewind to previous checkpoint*

`Ctrl + R` *# Search prompt history*

Modes

`Shift + Tab (2x)` *# Enter Plan Mode*

`Shift + Tab` *# Toggle Auto-accept*

`Tab` *# Toggle thinking display*

CLAUDE.md Template

markdown

PROJECT_NAME

Tech Stack

- Language/Framework version
- Key dependencies
- Database/storage

Project Structure

- '/src' - Description
- '/tests' - Description
- '/docs' - Description

Commands

- Build: 'command'
- Test: 'command'
- Run: 'command'
- Deploy: 'command'

Coding Standards

- Pattern to follow
- Pattern to avoid
- Style preferences

Workflow Rules

- Git branch strategy
- Testing requirements
- Review process

References

- @docs/architecture.md
- @docs/api-conventions.md

Common Patterns

Planning a Feature:

[Enter Plan Mode: Shift+Tab 2x]

Design an authentication system with:

- Email/password login
- JWT tokens
- Password reset flow
- Rate limiting

Ask clarifying questions about:

- Tech stack preferences
- Security requirements
- UI framework
- Database choice

Debugging:

There's a bug where [describe symptom].

Steps to reproduce:

1. [Step 1]
2. [Step 2]
3. [Expected vs Actual]

Relevant files:

- `@src/service.ts`
- `@src/handler.ts`

Find the root cause and fix it.

Refactoring:

Refactor `@src/legacy-code.ts` to:

1. Use TypeScript strict mode
2. Replace callbacks with `async/await`
3. Add proper error handling
4. Write unit tests

Maintain existing functionality.

Code Review:

Review the staged changes for:

- Type safety issues
- Potential bugs
- Security vulnerabilities
- Performance concerns
- Code style

Provide specific line references.

Keyboard Shortcuts

| Shortcut | Action |
|--------------|------------------------------|
| # | Add instruction to CLAUDE.md |
| ! | Force command execution |
| @ | Reference file/URL |
| Tab | Autocomplete file paths |
| Ctrl/Cmd + C | Cancel input |
| Ctrl/Cmd + R | Search history |

Cost Optimization Checklist

- Monitor with `/context` regularly
- Use Plan Mode before execution
- Clear between unrelated tasks
- Keep CLAUDE.md under 500 tokens
- Use sub-agents for exploration
- Batch sessions strategically
- Start with Pro plan, upgrade as needed
- Enable prompt caching (API users)

Troubleshooting Guide

"Permission denied" errors:

bash

```
# Don't use sudo, fix npm permissions
mkdir ~/.npm-global
npm config set prefix '~/.npm-global'
export PATH=~/.npm-global/bin:$PATH
```

"Claude doesn't understand my project":

1. Create/update CLAUDE.md
2. Tell Claude to explore: "Read the project structure"
3. Point to key files: "Look at @src/core/app.ts"

"Claude forgot earlier instructions":

1. Check context: [/context]
2. If > 60%, use [/compact] or [/clear]
3. For complex tasks, document progress before clearing

"Plan Mode not working (Windows)":

- Try [Alt + M] instead of [Shift + Tab]

"Rate limit hit":

- Wait for 5-hour window reset
- Consider upgrading plan
- Batch work more efficiently

"Open-source model doesn't work":

- Expected behavior—use frontier models
- See Chapter 13 for explanation

File Organization Best Practices

```
my-project/
└── .claude/
    ├── CLAUDE.md      # Main instructions
    └── rules/
        └── code-style.md # Code patterns
```

```
|- testing.md      # Test guidelines
|- security.md    # Security rules
|- commands/
|   |- pr.md        # Create PR
|   |- review.md    # Code review
|   |- test.md      # Run tests
|- agents/
|   |- reviewer.md  # Code reviewer
|   |- security.md  # Security auditor
|- hooks/
|   |- pre-commit.sh # Pre-commit checks
|   |- post-edit.sh  # Post-edit formatting
|- docs/
|   |- architecture.md # System design
|   |- api-conventions.md # API patterns
|   |- deployment.md   # Deploy guide
|- src/
  |- [your code]
```

MCP Quick Setup

```
bash

# Essential MCP servers
claude mcp add github -- npx @modelcontextprotocol/server-github
claude mcp add playwright -- npx @playwright/mcp@latest
claude mcp add filesystem -- npx @anthropic-ai/mcp-server-filesystem

# List connected
claude mcp list

# Remove server
claude mcp remove <server-name>
```

Git Worktree Pattern

```
bash
```

```
# Create worktrees for parallel work
git worktree add ..../project-feature-a feature-a
git worktree add ..../project-feature-b feature-b

# Open separate Claude sessions
# Terminal 1: cd ..../project-feature-a && claude
# Terminal 2: cd ..../project-feature-b && claude

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

Prompt Templates

Feature Implementation:

Implement [feature name] with these requirements:

1. [Requirement 1]
2. [Requirement 2]
3. [Requirement 3]

Technical constraints:

- [Constraint 1]
- [Constraint 2]

Ask clarifying questions before starting.

Bug Fix:

Fix the bug where [description].

Reproduction steps:

1. [Step]
2. [Step]
3. [Expected vs Actual]

Relevant context:

- [Context 1]
- [Context 2]

Root cause analysis first, then implement fix with tests.

Documentation:

Create comprehensive documentation for `@src/module.ts`:

1. Module purpose and architecture
2. Public API reference
3. Usage examples
4. Common pitfalls

Format: Markdown with code examples.

Appendix A: Resources & Further Reading

Official Documentation

- **Claude Code Docs:** <https://code.claude.com/docs>
- **Anthropic API Docs:** <https://docs.anthropic.com>
- **MCP Documentation:** <https://modelcontextprotocol.io>

Community Resources

- **Claude Code GitHub:** <https://github.com/anthropics/clause-code>
- **Discord Community:** [Join for discussions]
- **Reddit:** r/ClaudeAI, r/ClaudeCode

Alternative Tools

- **GitHub Copilot:** <https://github.com/features/copilot>
- **Cursor:** <https://cursor.com>
- **OpenAI Codex:** <https://openai.com/codex>
- **Gemini Code:** <https://gemini.google.com>

Learning More About AI Development

- **Anthropic Research:** <https://www.anthropic.com/research>
- **Building Effective Agents:** <https://www.anthropic.com/engineering/building-effective-agents>
- **Prompt Engineering Guide:** <https://docs.anthropic.com/claude/docs/prompt-engineering>

Model Providers

- **Anthropic (Claude):** <https://claude.ai>
 - **OpenRouter:** <https://openrouter.ai>
 - **Z.ai:** <https://z.ai>
 - **Ollama:** <https://ollama.ai>
-

Appendix B: Pricing Comparison Tables

Subscription Plans Comparison

| Plan | Claude Pro | Claude Max 5x | Claude Max 20x | GitHub Copilot | Cursor Pro |
|-------------|-------------|---------------|----------------|----------------|---------------------|
| Cost | \$20/mo | \$100/mo | \$200/mo | \$10-39/mo | \$20/mo |
| Messages | ~45/5hr | ~225/5hr | ~900/5hr | Unlimited* | Unlimited* |
| Context | 200k | 200k | 200k | ~8k | Variable |
| Claude Code | ✓ | ✓ | ✓ | ✗ | ✗ |
| IDE Plugin | ✗ | ✗ | ✗ | ✓ | ✓ |
| Best For | Individuals | Heavy users | All-day coding | Autocomplete | Full IDE experience |

*Subject to rate limiting

API Pricing (January 2026)

| Provider | Model | Input (per 1M tokens) | Output (per 1M tokens) | Quality Tier |
|-----------|-------------|-----------------------|------------------------|--------------|
| Anthropic | Opus 4.5 | \$3.00 | \$15.00 | Excellent |
| Anthropic | Sonnet 4.5 | \$1.00 | \$5.00 | Very Good |
| Anthropic | Haiku 4.5 | \$0.25 | \$1.25 | Good |
| Z.ai | GLM-4.7 | \$0.50 | \$1.50 | Good |
| Z.ai | DeepSeek V3 | \$0.27 | \$1.10 | Good |

| Provider | Model | Input (per 1M tokens) | Output (per 1M tokens) | Quality Tier |
|----------|---------------|-----------------------|------------------------|--------------|
| OpenAI | GPT-4 Turbo | \$2.50 | \$10.00 | Excellent |
| OpenAI | GPT-3.5 Turbo | \$0.50 | \$1.50 | Good |

Appendix C: Glossary

Agent: An AI system that can proactively use tools to achieve goals autonomously.

Agentic: Referring to the autonomous, tool-using capabilities of an AI model.

API: Application Programming Interface - how programs communicate with services.

CLAUDE.md: Configuration file that Claude Code reads automatically to understand project context.

Context Window: The amount of text (measured in tokens) an AI model can process at once.

Hook: A script that runs automatically at specific points in Claude's workflow.

MCP (Model Context Protocol): Standard for connecting AI models to external tools and services.

Plan Mode: A mode where Claude can only read and analyze, not write or execute.

Prompt Caching: A technique to reuse previously processed context, reducing costs.

Skill: Modular instructions and scripts that Claude loads on-demand.

Sub-agent: A separate Claude instance spawned to handle isolated tasks.

Token: Basic unit of text processing (roughly 0.75 words in English).

Tool Calling: The ability of an AI to invoke specific functions or commands.

Conclusion: Don't Panic, Keep Building

We've covered a lot of ground in this guide—from installation to advanced workflows, from cost optimization to the realities of open-source models.

Here's what matters most:

1. **Start Simple:** Install Claude Code, create a basic CLAUDE.md, and use it for real work.

2. **Build Intuition:** The more you use it, the better you'll understand when to plan, when to execute, and when to course-correct.
3. **Invest in Configuration:** Time spent on CLAUDE.md, commands, and hooks pays dividends across every session.
4. **Review Everything:** Claude is a powerful assistant, but you're still the senior engineer. Review, challenge, and refine.
5. **Use the Right Tool:** Claude Code excels at reasoning and architecture. Use other tools for their strengths too.
6. **Manage Costs:** Monitor usage, optimize context, and choose the right plan for your needs.
7. **Stay Honest:** Understand the limitations—both of the tool and of alternative models.

The Future is Collaborative

AI coding assistants aren't replacing developers. They're making good developers more productive and great developers unstoppable.

The developers who thrive in 2026 and beyond aren't the ones who resist AI or expect it to do everything. They're the ones who:

- Understand the strengths and weaknesses of these tools
- Know when to use which tool
- Invest in configuration and workflows
- Maintain high standards for code quality
- Keep learning and adapting

Your Journey Starts Now

You have everything you need to become proficient with Claude Code:

- The technical knowledge (this guide)
- The tools (Claude Code itself)
- The community (forums, Discord, GitHub)

What's missing is practice. So close this guide, open your terminal, and start building.

And remember the most important rule:

Don't Panic. 🎒

Feedback & Contact

Found an error? Have a suggestion? Want to share your own Claude Code tips?

Reach out on LinkedIn or contribute to the community knowledge base.

Acknowledgments

Special thanks to:

- The Anthropic team for building Claude and Claude Code
 - The broader AI coding community for sharing workflows and best practices
 - The developers who experimented with open-source models and shared results
-

The Hitchhiker's Guide to Claude Code

End of Document

"So long, and thanks for all the fish!" - Douglas Adams

 Don't forget your towel!