COMPREHENSIVE GUIDE

The Complete Boris Cherny Claude Code Manual

Master Anthropic's AI-Powered Coding Tool Through the Methodologies of Its Creator and the Claude Code Team

259	497	40k+	100%
PRS IN 30 DAYS	COMMITS	LINES WRITTEN	AI-GENERATED

Synthesized from 60+ sources including Twitter threads, podcast transcripts, official documentation, and team methodologies

Version 1.0 · January 2026

Table of Contents

This manual distills Boris Cherny's complete methodology for using Claude Code at maximum effectiveness—from foundational principles to advanced orchestration patterns.

Synthesized from 60+ sources including Boris's Twitter thread (all replies), podcast transcripts (Latent Space, AI & I, Behind the Craft), official Anthropic documentation, Hacker News comments, and all referenced resources. Verified for accuracy January 2026.

Table of Contents

PART 1

The Boris Cherny Philosophy

Origins

Boris Cherny created Claude Code as a side project in September 2024. He was experimenting with Claude 3.6 and initially built a command-line tool that could tell him what music he was listening to at work via AppleScript. After Cat Wu (founding PM) suggested expanding filesystem access, Claude could autonomously explore codebases by following import chains.

> *"When I created Claude Code as a side project back in September 2024, I had no idea it would grow to be what it is today."*
>
> — Boris Cherny

The idea spread "like wildfire" at Anthropic after being given filesystem access. Within two months (November 2024), they had a dogfooding-ready version. On day one, 20% of Engineering used it. By day five, 50% of Engineering was using Claude Code.

The Numbers

In 30 days, Boris landed:

- **259 PRs**
- **497 commits**
- **40k lines added**
- **38k lines removed**
- **Every single line written by Claude Code + Opus 4.5**

His usage stats: ~~325 million tokens across 1.6k sessions, with the longest single session running nearly **2 days~~ (42 hours)**.

> *"In the last thirty days, 100% of my contributions to Claude Code were written by Claude Code."*
>
> — Boris Cherny

The Paired Approach (NOT "Vibe Coding")

Boris explicitly distinguishes his approach from pure "vibe coding":

> *"Probably near 80%, yeah it's very high. But so usually where we start is Claude writes the code, and then if it's not good, then maybe a human will dive in."*
>
> — Boris Cherny

When to vibe code:

> *"Vibe coding works well for throwaway code and prototypes, code that's not in the critical path. I do this all the time, but it's definitely not the thing you want to do all the time. You want maintainable code sometimes. You want to be very thoughtful about every line sometimes."*
>
> — Boris Cherny

When NOT to vibe code:

> *"If it's intricate data model refactoring or something, I won't leave it to Claude because I have really strong opinions."*
>
> — Boris Cherny

Build for the Model Six Months From Now

Boris's manager Ben pushed him to:

> *"Build for the model six months from now—not today's limitations."*
>
> — Boris Cherny

This forward-thinking approach proved crucial. Claude Code barely worked initially but became transformative once Sonnet and Opus 4 arrived. The team:

- Treats cost and latency gaps as temporary, not terminal
- Prototypes ahead of capability
- Deletes code with each model upgrade (recently removed 2,000 tokens from system prompts when Sonnet 4.5 eliminated the need for scaffolding that Opus 4.1 required)

Code is No Longer the Bottleneck

> *"Code is no longer the bottleneck. Execution and direction are."*
>
> — Boris Cherny

Human oversight and judgment remain essential:

> *"Models are still overall not great at coding. There's still so much room to improve, and this is the worst it's ever gonna be."*
>
> — Boris Cherny

The Karpathy Story (Mental Model Adjustment)

In response to Andrej Karpathy saying "I've never felt this much behind as a programmer":

> *"I feel this way most weeks tbh. Sometimes I start approaching a problem manually, and have to remind myself 'claude can probably do this'."*
>
> — Boris Cherny

The Memory Leak Example:

> *"Recently we were debugging a memory leak in Claude Code, and I started approaching it the old fashioned way: connecting a profiler, using the app, pausing the profiler, manually looking through heap allocations."*
>
> — Boris Cherny

His coworker was looking at the same issue and just asked Claude to make a heap dump, then read the dump to look for retained objects that probably shouldn't be there. **Claude 1-shotted it and put up a PR.**

Key insight on new vs. experienced engineers:

> *"Newer coworkers and even new grads that don't make all sorts of assumptions about what the model can and can't do—legacy memories formed when using old models—are able to use the model most effectively. It takes significant mental work to re-adjust to what the model can do every month or two, as models continue to become better and better at coding and engineering."*
>
> — Boris Cherny

PART 2

Boris's Personal Setup

The "Surprisingly Vanilla" Philosophy

> *"My setup might be surprisingly vanilla! Claude Code works great out of the box, so I personally don't customize it much. There is no one correct way to use Claude Code: we intentionally build it in a way that you can use it, customize it, and hack it however you like. Each person on the Claude Code team uses it very differently."*
>
> — Boris Cherny

Model Selection

Opus 4.5 with thinking for everything.

> *"It's the best coding model I've ever used. It's bigger and slower than Sonnet but requires less steering. Sonnet 3.7 is also good, but Opus 4.5 is better at tool use and reasoning."*
>
> — Boris Cherny

> *"Even though it's bigger & slower than Sonnet, since you have to steer it less and it's better at tool use, it is almost always faster than using a smaller model in the end."*
>
> — Boris Cherny

On model adherence:

> *"Adherence to instructions has improved significantly with newer models—Opus 4.5 listens to me the first time."*
>
> — Boris Cherny

Parallel Instance Strategy

Boris runs **15+ Claude instances** simultaneously:

Terminal (5 instances)

- Numbers tabs 1-5
- Uses iTerm2 system notifications to know when a Claude needs input
- **Setup**: iTerm2 → Preferences → Profiles → Terminal → Enable "Silence bell" and "Send escape sequence-generated alerts"

> *"I run 5 Claudes in parallel in my terminal. I number my tabs 1-5, and use system notifications to know when a Claude needs input."*
>
> — Boris Cherny

Web (5-10 instances)

- Runs on claude.ai/code in parallel with local instances
- Hands off local sessions to web using `&` operator
- Uses `--teleport` to move between terminal and web

> *"I also run 5-10 Claudes on claude.ai/code, in parallel with my local Claudes. As I code in my terminal, I will often hand off local sessions to web (using &), or manually kick off sessions in Chrome, and sometimes I will --teleport back and forth."*
>
> — Boris Cherny

Mobile

- Starts sessions from phone via Claude iOS app
- Checks status when reaching computer
- Power users start tasks during commute, "teleport" to desktop for deeper review

> *"I also start a few sessions from my phone (from the Claude iOS app) every morning and throughout the day, and check in on them later."*
>
> — Boris Cherny

Tab Management Method

The "cascading terminal tabs" approach:

1. Open 3-5 terminal tabs
2. Launch Claude in each with different tasks
3. Check in periodically, sweeping left-to-right
4. Approve/deny permissions as needed

His Daily Commands

Commands Boris uses **dozens of times daily**:

- `/commit-push-pr` - Commits, pushes, creates PR
- `/commit` - Runs linting and generates commit messages
- `/code-review` - Automated code review
- `/security-review` - Security audit (used for all PRs)
- `/feature-dev` - Complex feature development workflow

PART 3

The Three Core Principles

Boris's reply to a question prompted by Karpathy laid out his three core principles:

> *"1. Almost always use Plan mode 2. Give Claude a way to verify its output with unit tests, the Claude Chrome extension, or an iOS/Android sim 3. Hold the same bar for human and Claude code. Use /code-review to automate most of code review"*
>
> — Boris Cherny

Principle 1: Almost Always Use Plan Mode

How to Enter Plan Mode

- Press `Shift+Tab` twice
- Or start prompt with planning language

What Plan Mode Does

- Asks clarifying questions
- Creates todo lists
- Shows everything it will change before execution
- Uses extended thinking for comprehensive strategy

Thinking Intensifiers

Use progressively stronger prompts for complex problems:

Phrase	Thinking Budget
`"think"`	Basic extended thinking
`"think hard"`	More thinking budget
`"think harder"`	Even more
`"ultrathink"`	Maximum thinking allocation

> *"If you want Claude to think, just tell it to think. Make a plan. Think hard. Don't write any code yet."*
>
> — Boris Cherny

Boris's Workflow

> *"Most sessions start in Plan mode (shift+tab twice). For Pull Requests, I use Plan mode and go back and forth with Claude until I like its plan. From there, I switch into auto-accept edits mode where Claude can usually 1-shot it. A good plan is really important!"*
>
> — Boris Cherny

Key insight: Aligning on plans before implementation can "2-3x success rates pretty easily."

Plan Mode's Future

Boris on the temporary nature of features:

> *"Plan mode itself is an example—we'll probably unship it at some point when Claude can just figure out from the user's intent that they probably want to plan first."*
>
> — Boris Cherny

Principle 2: Give Claude Verification Feedback Loops

This is Boris's **most critical recommendation**:

> *"Give Claude a way to verify its output with unit tests, the Claude Chrome extension, or an iOS/Android sim. If Claude has that feedback loop, it will 2-3x the quality of the final output."*
>
> — Boris Cherny

From his Hacker News comment:

> *"Give the model a way to check its work. For svelte, consider using the Puppeteer MCP server and tell Claude to check its work in the browser. This is another 2-3x."*
>
> — Boris Cherny

Verification Methods

Method	Use Case	How
Unit Tests	Logic verification	Write tests first, Claude iterates until passing
Chrome Extension	UI/web verification	Real browser, console logs, DOM inspection
iOS/Android Simulator	Mobile app verification	Visual and functional testing
Puppeteer/Playwright MCP	Automated browser testing	Headless or headed browser control
Screenshot Comparison	Design matching	Compare output to design mocks
Linter/Type-checker	Code quality	Automated gates

The TDD Superpower

Test-driven development amplifies Claude's effectiveness:

1. Write tests from expected input/output pairs (be explicit about TDD to avoid mocks)
2. Run tests and confirm they fail
3. Commit the tests
4. Tell Claude to pass tests without modifying them
5. Use subagents to verify implementation isn't overfitting
6. Commit implementation

> *"I have not manually written a unit test in many months."*
>
> — Boris Cherny

Principle 3: Hold the Same Bar for Human and Claude Code

> *"Hold the same bar for human and Claude code. Use /code-review to automate most of code review."*
>
> — Boris Cherny

- Apply identical code review standards regardless of who wrote it
- Poor code gets rejected consistently
- Don't lower standards because AI wrote it
- Human code review remains essential despite high automation percentage

PART 4

Session Management & Continuous Operation

Stop Hooks: The Key to Multi-Hour/Multi-Day Sessions

Boris's record: **~42 hours straight** of uninterrupted Claude work.

> *"Claude consistently runs for minutes, hours, and days at a time (using Stop hooks)."*
>
> — Boris Cherny

How Stop Hooks Work

Configure hooks that run when Claude finishes responding:

- If tests don't pass, keep going
- If completion criteria not met, continue iterating
- Enables deterministic outcomes from stochastic processes

> *"Stop hooks fix Claude's tendency to 'stop' after a tool call by automatically resuming—turning short bursts into uninterrupted marathons."*
>
> — Boris Cherny

The Ralph Wiggum Plugin

The official Anthropic plugin Boris uses for continuous iteration:

Installation:

```
/plugin install ralph-wiggum@claude-plugins-official
```

Usage:

```
/ralph-loop "Your task description" --completion-promise "DONE" --max-iterations 50
```

How it works:

1. Feed a prompt to Claude
2. When Claude tries to exit, the Stop hook intercepts (exit code 2)
3. The same prompt is re-injected
4. Loop continues until completion criteria met or max iterations reached

> *"For very long-running tasks, I will either (a) prompt Claude to verify its work with a background agent when it's done, (b) use an agent Stop hook to do that more deterministically, or (c) use the ralph-wiggum plugin (originally dreamt up by Geoff Huntley)."*
>
> — Boris Cherny

Named after: Ralph Wiggum from The Simpsons—embodying persistent iteration despite setbacks.

Prompt Writing for Continuous Loops

Include clear completion criteria:

```
Build a REST API for todos.

When complete:
- All CRUD endpoints working
- Input validation in place
- Tests passing (coverage > 80%)
- README with API docs
- Output: <promise>COMPLETE</promise>
```

Include incremental goals:

```
Phase 1: User authentication (JWT, tests)
Phase 2: Product catalog (list/search, tests)
Phase 3: Shopping cart (add/remove, tests)

Output <promise>COMPLETE</promise> when all phases done.
```

Always include escape hatches:

```
After 15 iterations, if not complete:
- Document what's blocking progress
- List what was attempted
- Suggest alternative approaches
```

Permission Modes for Long Sessions

> *"I will also use either --permission-mode=dontAsk or --dangerously-skip-permissions in a sandbox to avoid permission prompts for the session, so Claude can cook without being blocked on me."*
>
> — Boris Cherny

Real-World Results

- YC hackathon teams shipped **6+ repos overnight** for ~$297 in API costs
- One developer built an entire programming language ("Cursed") over 3 months
- One $50k contract completed for $297 in API costs

Cost Considerations

> *"A typical 50-iteration loop on a medium-sized codebase might cost $50-100+ in API usage."*
>
> — Boris Cherny

Always use `--max-iterations` as cost control.

PART 5

CLAUDE.md: The Team Knowledge System

The Shared Knowledge File

> *"Our team shares a single CLAUDE.md for the Claude Code repo. We check it into git, and the whole team contributes multiple times a week. Anytime we see Claude do something incorrectly we add it to the CLAUDE.md, so Claude knows not to do it next time."*
>
> — Boris Cherny

Key Insight

This creates a **compounding knowledge system**:

- Mistakes are captured once
- Claude learns from them forever
- Team knowledge accumulates automatically

> *"CLAUDE.md is the simplest thing that could work."*
>
> — Boris Cherny

File Locations & Hierarchy

Location	Scope	Use Case
`~/.claude/CLAUDE.md`	All sessions	Personal preferences, global rules
`./CLAUDE.md`	Project root	Check into git for team sharing
`./CLAUDE.local.md`	Project root	Gitignore for personal use
`./subdirectory/CLAUDE.md`	Directory-specific	Pulled on demand when accessing
Parent directories	Monorepos	Shared across packages

Loading Hierarchy

1. Enterprise-level configuration
2. User home directory
3. Project root
4. Directory-specific files (loaded when accessing related files)

Essential Content Structure

```
# Project: [Name]

## Bash Commands
- npm run build: Build the project
- npm run typecheck: Run the typechecker
- npm test: Run test suite
- npm run lint: Run linter

## Code Style
- Use ES modules (import/export), not CommonJS
- Destructure imports when possible
- TypeScript strict mode always
- Prefer functional over class-based patterns

## Testing
- Run single tests, not entire suite
- Use vitest for unit tests
- Playwright for E2E

## Repository Conventions
- Feature branches from main
- Squash merge PRs
- Conventional commits

## Common Mistakes to Avoid
- [Document errors Claude has made]
- [Include corrections and patterns]
- [Add emphasis for critical items]

## Developer Environment
- Node 20+
- Required env vars: DATABASE_URL, API_KEY
```

Critical Guidelines

Keep It Under ~1,000 Tokens

From Boris's Hacker News comment:

> *"Keep CLAUDE.md under approximately 1,000 tokens to avoid performance issues. File size warnings will display if it becomes too large."*
>
> — Boris Cherny

The Team Removes Instructions With Each Model Release

> *"The team removes outdated instructions with each model release since smarter models need less guidance."*
>
> — Boris Cherny

Example: When Sonnet 4.5 shipped, they deleted ~2,000 tokens from system prompts that were only needed for Opus 4.1.

Treat It Like a Prompt

- Iterate on effectiveness, don't just accumulate
- Press `#` to have Claude automatically incorporate instructions
- Use the prompt improver tool on CLAUDE.md files

Add Emphasis for Critical Rules

Use **"IMPORTANT"** or **"YOU MUST"** to improve adherence:

```
IMPORTANT: Always run typecheck before committing
YOU MUST use the existing Button component, do not create new ones
```

Let Claude Edit It Directly

```
"At Claude, add this to CLAUDE.md so that the next time it just knows this
automatically."
```

GitHub Action for CLAUDE.md Updates

> *"During code review, I will often tag @.claude on my coworkers' PRs to add something to the CLAUDE.md as part of the PR. We use the Claude Code GitHub action (/install-github-action) for this. It's our version of Dan Shipper's Compounding Engineering."*
>
> — Boris Cherny

PART 6

Slash Commands

Boris's Approach

> *"I use slash commands for every 'inner loop' workflow that I end up doing many times a day. This saves me from repeated prompting, and makes it so Claude can use these workflows too. Commands are checked into git and live in .claude/commands/."*
>
> — Boris Cherny

Creating Slash Commands

Store prompt templates as Markdown files:

Project commands: `.claude/commands/` → accessed as `/project:command-name` **Personal commands:** `~/.claude/commands/` → accessed as `/user:command-name`

Boris's Key Commands

`/commit-push-pr`

Used dozens of times daily:

```
# .claude/commands/commit-push-pr.md

Create a commit, push, and open a PR with the following steps:

1. Run `git status` to see changes
2. Run `git diff --staged` to understand what's being committed
3. Generate a descriptive commit message based on changes
4. Run `git commit -m "<message>"`
5. Run `git push -u origin $(git branch --show-current)`
6. Run `gh pr create --fill`

If any step fails, stop and explain the error.
```

> *"The command uses inline bash to pre-compute git status and a few other pieces of info to make the command run quickly and avoid back-and-forth with the model."*
>
> — Boris Cherny

`/fix-github-issue`

```
# .claude/commands/fix-github-issue.md

Please analyze and fix the GitHub issue: $ARGUMENTS.

Follow these steps:
1. Use `gh issue view $ARGUMENTS` to get issue details
2. Understand the problem thoroughly
3. Search codebase for relevant files
4. Implement necessary changes
5. Write and run tests
6. Ensure code passes linting and type checking
7. Create descriptive commit message
8. Push and create PR

Remember to use GitHub CLI (`gh`) for all GitHub tasks.
```

Usage: `/project:fix-github-issue 1234`

`/feature-dev`

Created by Sid on the team—structured feature development:

> *"First ask me what exactly I want, build the specification, then build a detailed plan, then make a to-do list, walk through [that] step-by-step."*
>
> — Boris Cherny

`/commit`

> *"The most basic slash command is '/commit,' which handles the tedious work of committing code changes. Boris has configured his version to automatically run certain commands which save your code changes and push them to a shared repository without asking for permission each time."*
>
> — Boris Cherny

Advanced: Inline Bash Pre-computation

Commands can include bash scripts that pre-compute context:

````
---
allowed-tools:
  - Bash(git status)
  - Bash(git diff)
---

```bash
git status --short
git diff --staged | head -100
````

Based on the above changes, create a commit message following conventional commits.
````

````
## Parameterized Commands

Use `$ARGUMENTS`, `$1`, `$2` for parameters:

```markdown
# .claude/commands/test-file.md
Run tests for the file: $1

```bash
npm test -- $1
````

Report any failures and suggest fixes.

````
**Usage**: `/project:test-file src/utils/parser.ts`

---

# Part 7: Subagents

## Boris's Subagent Philosophy

> "I use a few subagents regularly: code-simplifier simplifies the code after Claude
is done working, verify-app has detailed instructions for testing Claude Code end to
end, and so on. Similar to slash commands, I think of subagents as automating the most
common workflows that I do for most PRs."

## Why Subagents?

> "Using multiple subagents creates separate context windows that don't know about
each other, producing better results than a single agent."

Key benefits:
- **Isolated contexts** prevent "context pollution"
- **Parallel execution** for concurrent work
- Keep main conversation focused on high-level planning
- Specialized expertise per agent

## Creating Subagents

**Via command**: `/agents`

**Manual creation**: `.claude/agents/agent-name.md`

```markdown
---
name: code-simplifier
description: Use this agent when you have functional code that needs refactoring to
improve readability, reduce complexity, or eliminate redundancy.
model: sonnet
tools:
  - Read
  - Edit
  - Grep
---

# Code Simplifier

You are a code simplification specialist. Your goal is to make code more readable and
maintainable without changing functionality.

## Simplification Principles
- Reduce complexity and nesting
- Extract repeated logic into reusable functions
- Use meaningful variable names
- Remove dead code
````
````

```
- Simplify conditional logic
- Follow DRY principles
```

Boris's Key Subagents

code-simplifier

Purpose: Simplify code after Claude is done working

When to invoke:

- After each feature implementation
- Upon TODO completion
- Following major code additions
- Before code reviews

Trade-off: Development speed halves, but long-term maintainability improves dramatically.

verify-app

Purpose: Detailed end-to-end testing instructions for Claude Code

code-reviewer (The 3+5 Pattern)

Purpose: Multi-perspective code review

Boris's code review spawns **several subagents at once**:

First pass (3 agents):

- One checks style guidelines
- Another combs through the project's history to understand what's already been built
- Another flags obvious bugs

Second pass (5 agents):

> *"The first pass catches real problems but also false alarms, so he uses five more subagents specifically tasked with poking holes in the original findings."*
>
> — Boris Cherny

> *"In the end, the result is awesome—it finds all the real issues without the false [ones]."*
>
> — Boris Cherny

Parallel Migration Pattern (Map-Reduce)

For large-scale operations:

> *"The main agent makes a big to-do list for everything and map reduces over a bunch of subagents. You can instruct Claude to start 10 agents and just go 10 at a time and migrate all the stuff over."*
>
> — Boris Cherny

Example workflow:

1. Main agent creates task list (e.g., 2,000 files needing migration)
2. Spawn 10 subagents in parallel
3. Each handles a batch of files
4. Main agent aggregates results

Real-world usage: Organizations spending $1,000+/month run these swarm migrations for framework updates, lint rule rollouts, and API migrations—achieving **10x+ speedup** vs sequential.

> *"We support parallel workflows—fixing 1,000 lint violations simultaneously."*
>
> — Boris Cherny

Specialized Subagent Examples

Frontend Testing (by Cat Wu)

Built with Playwright for automated visual testing

Opponent Processing (by Dan)

- "Pro-Dan subagent" argues one side
- "Auditor subagent" debates the opposite
- Used for finance audits and expense reviews

Code Explorer

Explores different aspects of codebase:

- Similar features and patterns
- Architecture and abstractions
- Related feature implementations

Code Architect

Multiple agents with different focuses:

- Minimal changes approach
- Clean architecture approach
- Pragmatic balance

PART 8

Hooks

Boris's Hook Usage

> *"We use a PostToolUse hook to format Claude's code. Claude usually generates well-formatted code out of the box, and the hook handles the last 10% to avoid formatting errors in CI later."*
>
> — Boris Cherny

Eight Hook Types

Hook	Trigger	Use Case
PreToolUse	Before tool executes	Validate inputs, block dangerous commands
PermissionRequest	Before permission dialog	Auto-approve safe operations
PostToolUse	After tool completes	Run formatters, linters, log changes
PreCompact	Before context compaction	Back up transcripts
SessionStart	Session begins/resumes	Inject git status, load TODOs
Stop	Claude finishes responding	Verify completion, run tests
SubagentStop	Subagent completes	Validate output
UserPromptSubmit	User submits prompt	Inject context, validate requests

Why Hooks Matter

> *"Hooks are 'huge' for Claude Code. While not necessary for hobby projects, they are critical for steering Claude in a complex enterprise repo. They provide deterministic 'must-do' rules that complement the 'should-do' suggestions in CLAUDE.md."*
>
> — Boris Cherny

Configuration

Hooks live in JSON settings files:

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "prettier --write \"$CLAUDE_TOOL_INPUT_FILE_PATH\""
          }
        ]
      }
    ]
  }
}
```

Boris's PostToolUse Hook (Prettier)

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "prettier --write \"$CLAUDE_TOOL_INPUT_FILE_PATH\""
          }
        ]
      }
    ]
  }
}
```

Common Hook Patterns

Auto-approve test commands

```
{
  "hooks": {
    "PermissionRequest": [
      {
        "matcher": "Bash(npm test*)",
        "hooks": [
          {
            "type": "command",
            "command": "echo '{\"decision\": \"approve\"}'"
          }
        ]
      }
    ]
  }
}
```

Inject context on session start

```
{
  "hooks": {
    "SessionStart": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "git status --short && echo '---' && cat TODO.md"
          }
        ]
      }
    ]
  }
}
```

Stop hook for continuous iteration

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "if ! npm test; then echo '{\"continue\": true}'; fi"
          }
        ]
      }
    ]
  }
}
```

Matcher Syntax

- Simple: `"Write"` matches only Write tool
- Multiple: `"Write|Edit"` matches either
- Wildcard: `"*"` matches all tools
- Arguments: `"Bash(npm test*)"` matches specific patterns
- MCP: `"mcp__memory__.*"` for MCP tools

PART 9

MCP Integrations

Boris's MCP Setup

> *"Claude Code uses all my tools for me. It often searches and posts to Slack (via the MCP server), runs BigQuery queries to answer analytics questions (using bq CLI), grabs error logs from Sentry, etc. The Slack MCP configuration is checked into our .mcp.json and shared with the team."*
>
> — Boris Cherny

Recommended MCPs

MCP	Use Case	Boris's Notes
Puppeteer	Browser automation	"High utility"
Playwright	Browser testing	"High utility"
Sentry	Error log retrieval	Used daily
Slack	Team communication	Checked into git
BigQuery	Analytics queries	Using bq CLI
Asana	Project management	
GitHub	Repository operations	Auto-commits via GitHub MCP

Configuration

Project-level (shared via git)

```
// .mcp.json
{
  "mcpServers": {
    "slack": {
      "command": "node",
      "args": ["/path/to/slack-mcp/index.js"],
      "env": {
        "SLACK_TOKEN": "${SLACK_TOKEN}"
      }
    },
    "sentry": {
      "command": "npx",
      "args": ["@sentry/mcp-server"],
      "env": {
        "SENTRY_AUTH_TOKEN": "${SENTRY_AUTH_TOKEN}"
      }
    }
  }
}
```

Adding MCP servers

```
claude mcp add puppeteer npx @playwright/mcp@latest
claude mcp add slack -e SLACK_TOKEN=xxx -- node /path/to/server
```

Debugging

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

Key Insight

> *"MCPs suit complex tools requiring multiple calls (Puppeteer) and cross-system integration. Slash commands work as simple saved prompts for local workflows."*
>
> — Boris Cherny

Agentic Search vs. RAG

> *"We tried RAG, attempted various search tools, landed on agentic search. It outperformed alternatives while avoiding indexing complexity."*
>
> — Boris Cherny

The team uses grep and glob for agentic search rather than vector stores.

PART 10

The Chrome Extension & Visual Verification

Boris's Verification Workflow

> *"Give Claude a way to verify its output with unit tests, the Claude Chrome extension, or an iOS/Android sim."*
>
> — Boris Cherny

> *"Claude tests every single change I land to claude.ai/code using the Claude Chrome extension. It opens a browser, tests the UI, and iterates until the code works and the UX feels good."*
>
> — Boris Cherny

Chrome Extension Capabilities

Capability	Description
Live debugging	Reads console errors, DOM state, then fixes code
Design verification	Build from mockups, verify matches in browser
Web app testing	Test forms, validation, visual regressions
Authenticated apps	Interact with Gmail, Notion, Google Docs
Data extraction	Pull structured info from pages
Session recording	Record interactions as GIFs
Workflow learning	Record a workflow, Claude learns to repeat it

> *"The most intriguing feature of Claude in Chrome is recording a workflow to teach the AI how to do something on your behalf—you click the record button, go about your business, and Claude watches and remembers."*
>
> — Boris Cherny

Setup

```
# Update Claude Code
claude update

# Start with Chrome enabled
claude --chrome

# Or enable by default
/chrome
# Select "Enabled by default"
```

Example Workflows

Test a local web app

```
I just updated the login form validation. Can you open localhost:3000,
try submitting with invalid data, and check if error messages appear?
```

Debug with console logs

```
Open the dashboard page and check the console for any errors when loading.
```

> *"Claude can read browser console output, including errors, network requests, and DOM state, helping developers identify and debug issues without leaving the browser."*
>
> — Boris Cherny

Build from design mock

```
Here's the Figma mock [paste image]. Implement this design.
After each change, take a screenshot and compare.
Iterate until it matches.
```

Record a demo GIF

```
Record a GIF showing the checkout flow from cart to confirmation.
```

Visual Development Workflow

1. Give Claude screenshot capabilities
2. Provide visual mock (paste image or file path)
3. Tell Claude to implement, screenshot, and iterate
4. After 2-3 iterations, outputs typically improve significantly

> *"Builds UIs by providing screenshots and iterating with Puppeteer until matching mockups."*
>
> — Boris Cherny

PART 11

Advanced Workflows

Workflow A: Explore → Plan → Code → Commit

This is the official Anthropic-recommended workflow:

Step 1: Explore

```
Read relevant files, images, or URLs... but don't write code yet.
```

- Use subagents to verify details and investigate questions
- Preserve main context for high-level planning

Step 2: Plan

```
Make a plan for how to approach this problem.
Think hard about how to implement rate limiting.
Consider existing patterns and create a detailed plan.
Document it in a GitHub issue.
```

- Use thinking intensifiers: "think" → "think hard" → "ultrathink"
- Iterate until satisfied
- Document plan as GitHub issue to reset if implementation fails

Step 3: Code

- Switch to auto-accept mode (`Shift+Tab`) for autonomous execution
- Or stay in approval mode for active collaboration
- Ask Claude to verify solution reasonableness as pieces are built

Step 4: Commit

```
Commit result and create pull request.
Update READMEs/changelogs with explanations.
```

Key insight: Steps 1-2 are crucial—without them, Claude tends to jump straight to coding.

Workflow B: Throwaway Development

Boris's technique for exploring unclear features:

> *"One unintuitive step of the plan development process is even if you don't exactly know what the thing that needs to be built is, you just have a little sentence in mind like 'I want feature X' and have Claude just implement it without giving it anything else. You see what it does and that helps understand 'here's actually what I mean' because it made all these different mistakes or did something unexpected that might be good. Then you use that learning from a sort of 'throwaway development'—clear it out and it helps write a better plan spec for the actual feature development."*
>
> — Boris Cherny

The workflow:

1. Give Claude a vague requirement
2. Let it attempt implementation
3. Observe what works and what fails
4. Press **Escape twice** to revert

5. Use learnings to write better spec
6. Implement properly

Workflow C: Test-Driven Development

Boris's preferred quality approach:

1. **Write tests from expected I/O pairs**

```
Write tests for user registration. Expected:
- Valid email + password → success
- Invalid email → error "Invalid email format"
Be explicit: use TDD, no mocks for core logic.
```

1. **Confirm tests fail**

```
Run tests and confirm they fail. Do NOT write implementation yet.
```

1. **Commit tests**
2. **Implement to pass**

```
Write code to pass these tests. Do NOT modify the tests.
```

1. **Verify with subagent** Use code-simplifier or reviewer to check implementation isn't overfitting
2. **Commit implementation**

Workflow D: Visual Iteration

1. Give Claude screenshot capabilities (Chrome extension, Puppeteer, simulator)
2. Provide visual mock
3. Tell Claude to implement, screenshot, compare, iterate
4. Typically 2-3 iterations yields significant improvement

Workflow E: Prototyping Multiple Versions

Instead of design documents:

> *"I'll just ask Claude to prototype three versions. Try the feature. See which I like better."*
>
> — Boris Cherny

> *"Investigate three separate ideas for refactoring. Do it in parallel. Use three agents."*
>
> — Boris Cherny

Workflow F: Codebase Q&A

When onboarding or exploring:

```
How does logging work in this codebase?
How do I make a new API endpoint?
What does `async move { ... }` do on line 134 of foo.rs?
What edge cases does CustomerOnboardingFlowImpl handle?
Why call `foo()` instead of `bar()`?
What's the Java equivalent of this Python code?
```

> *"No special prompting needed. Claude searches agentically. This has become Anthropic's core onboarding workflow."*
>
> — Boris Cherny

Workflow G: Git History Analysis

Claude effectively handles:

- Searching git history for release contents
- Understanding feature ownership
- Analyzing API design decisions
- Writing commit messages
- Reverting files, resolving rebases

> *"Many Anthropic engineers use Claude for 90%+ of git interactions."*
>
> — Boris Cherny

> *"Explicitly ask Claude to search git history for architectural decisions."*
>
> — Boris Cherny

PART 12

The Feature Development Plugin

Overview

Created by Sid Bidasaria on the Claude Code team. A 7-phase structured workflow:

```
/feature-dev Add user authentication with OAuth
```

> *"For more complex work, Boris uses '/feature-dev,' which walks him through building something step-by-step."*
>
> — Boris Cherny

The 7 Phases

Phase 1: Discovery

- Clarifies the feature request
- Asks what problem you're solving
- Identifies constraints and requirements

Phase 2: Codebase Exploration

- Launches 2-3 `code-explorer` agents in parallel
- Each explores different aspects (similar features, architecture, UI patterns)

- Presents comprehensive summary with key files

Phase 3: Clarifying Questions

- Identifies underspecified aspects
- Edge cases, error handling, integration points
- **Waits for your answers before proceeding**

Phase 4: Architecture Design

- Launches 2-3 `code-architect` agents with different focuses: - Minimal changes approach - Clean architecture approach - Pragmatic balance
- Presents comparison with trade-offs
- **Asks which approach you prefer**

Phase 5: Implementation

- **Waits for explicit approval**
- Implements following chosen architecture
- Updates todos as progress is made

Phase 6: Quality Review

- Launches 3 `code-reviewer` agents: - Simplicity/DRY/Elegance - Bugs/Correctness - Conventions/Abstractions
- **Presents findings and asks what to fix**

Phase 7: Summary

- Documents what was built
- Key decisions made
- Files modified
- Suggested next steps

Installation

```
npx claude-plugins install @anthropics/claude-code-plugins/feature-dev
```

PART 13

Multi-Claude Orchestration

Boris's Parallel Strategy

> *"I run 5 Claudes in parallel in my terminal... also run 5-10 Claudes on claude.ai/code in parallel with local Claudes."*
>
> — Boris Cherny

Pattern A: One Writes, Another Verifies

1. Use Claude to write code
2. Run `/clear` or start second Claude in another terminal
3. Second Claude reviews first Claude's work
4. Start third Claude to read both code and review
5. Third Claude edits based on feedback

> *"Separation often yields better results than single-agent workflows."*
>
> — Boris Cherny

Pattern B: Multiple Git Checkouts

1. Create 3-4 git checkouts in separate folders
2. Open each folder in separate terminal tabs

3. Start Claude in each with different tasks
4. Cycle through to check progress

Pattern C: Git Worktrees (Recommended)

Lighter-weight alternative with shared Git history:

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

# Launch Claude in each
cd ../project-feature-a && claude
cd ../project-feature-b && claude

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

Tips:

- Use consistent naming conventions
- One terminal tab per worktree
- Set up iTerm2 notifications
- Use separate IDE windows

Pattern D: Map-Reduce Swarm

For large migrations:

```
# Main agent creates task list
# Then spawns 10+ subagents in parallel
/ralph-loop "Migrate all 2000 files from React to Vue.
Use 10 subagents in parallel.
Output COMPLETE when done." --max-iterations 100
```

Pattern E: Headless Automation (Non-Interactive Mode)

```
# Fan out for batch processing
claude -p "migrate foo.py from React to Vue. Return OK or FAIL." \
  --allowedTools Edit "Bash(git commit:*)"
```

Use `--output-format stream-json` for structured output.

Scaling approach:

> *"Start small. Test on one test. Scale to 10. Gradually increase."*
>
> — Boris Cherny

1. Have Claude write a script generating 2k tasks
2. Loop through tasks calling Claude programmatically
3. Refine prompts iteratively
4. Scale up gradually

PART 14

Context Engineering

The Core Problem

> *"Context refers to the set of tokens included when sampling from an LLM; the problem is optimizing token utility against constraints."*
>
> — Boris Cherny

Boris's Context Management

Keep Context Fresh

- Use `/clear` frequently between tasks
- Start fresh sessions for new work
- Single task per session when possible

> *"Use `/clear` frequently between tasks to reset context window and maintain focus."*
>
> — Boris Cherny

Monitor Context Health

- Run `/context` to identify bloat sources
- Compact at ~60% for complex tasks
- Watch for degradation in long sessions

Use Checklists for Complex Workflows

> *"Have Claude use Markdown as a working scratchpad for large migrations, lint fixes, or build scripts. Tell Claude to generate task list. Address each issue sequentially, checking off before moving on."*
>
> — Boris Cherny

The Escape Twice / Rewind Feature

Double-tap Escape to jump back in history:

- Edit previous prompts
- Explore alternatives
- Undo destructive actions

> *"Press Esc twice or run `/rewind` to restore code, conversation, or both."*
>
> — Boris Cherny

The rewind feature:

1. Shows your conversation history with file diffs
2. Pick which state to revert to
3. Claude Code restores code state (undoing file edits, writes, or deletions)
4. Restores conversation context
5. Preserves earlier state

The Handoff Pattern

When context window fills or Claude struggles:

```
# /handoff command creates HANDOFF.md with:
- Goal of the session
- Progress made
- What worked
- What didn't work
- Next steps

Then /clear and continue from HANDOFF.md
```

Compounding Engineering

Each feature should make the next easier:

> *"Codify learnings about planning modifications, testing discoveries, and missed items back into prompts, subagents, and slash commands."*
>
> — Boris Cherny

PART 15

Permissions & Safety

Boris's Approach

> *"I don't use --dangerously-skip-permissions. Instead, I use /permissions to pre-allow common bash commands that I know are safe in my environment, to avoid unnecessary permission prompts. Most of these are checked into .claude/settings.json and shared with the team."*
>
> — Boris Cherny

Smart Permissions Configuration

```
// .claude/settings.json
{
  "permissions": {
    "allow": [
      "Bash(git *)",
      "Bash(npm test *)",
      "Bash(npm run lint)",
      "Read(*)",
      "Write(src/**)",
      "Edit(src/**)"
    ],
    "deny": [
      "Read(.env*)",
      "Bash(rm -rf *)",
      "Bash(sudo *)",
      "Write(.env*)"
    ]
  }
}
```

Methods to Set Permissions

1. **During sessions**: Select "Always allow" when prompted
2. **Persistent configuration**: Use `/permissions` command
3. **Manual configuration**: Edit `.claude/settings.json` or `~/.claude.json`
4. **Session-specific**: Use `--allowedTools` CLI flag

When to Use Dangerous Mode

Only use `--dangerously-skip-permissions` in:

- Containers without internet access
- Docker Dev Containers with proper isolation
- For lint fixes and boilerplate generation

> *"Safe YOLO mode: Run `claude --dangerously-skip-permissions` for uninterrupted work (use only in containerized environments without internet access)."*
>
> — Boris Cherny

Pre-commit Hook Performance

> *"Pre-commit hooks should run in five seconds—just types and lint. Expensive stuff goes in GitHub Actions."*
>
> — Boris Cherny

> *"If you're using Husky or Git pre-commit hooks, just add: `claude -p` with instructions."*
>
> — Boris Cherny

PART 16

Mobile & Cross-Device Workflows

Boris's Mobile Workflow

> *"I start sessions from my phone via the Claude iOS app, and check in on them when I reach my computer."*
>
> — Boris Cherny

> *"I also start a few sessions from my phone (from the Claude iOS app) every morning and throughout the day, and check in on them later."*
>
> — Boris Cherny

The Teleport Feature

Move seamlessly between web and CLI:

- Copies chat transcript and edited files
- Start on phone during commute
- "Teleport" to desktop for deeper work

> *"This design underscores Anthropic's hybrid philosophy: expanding local workflows into managed, sandboxed environments that can be paused, audited, or resumed anywhere."*
>
> — Boris Cherny

Tools for Remote Access

Claude Code Companion (Telegram)

```
/new myproject → Creates topic + session
# Claude works on tasks
# Attach from PC terminal to continue
```

tmux + SSH

- Persistent terminal environment
- Access from phone via Termius
- Push notifications when AI needs attention

PART 17

Team Practices at Anthropic

"Antfooding" (Internal Dogfooding)

> *"70-80% of technical Anthropic employees use Claude Code every day. Feedback appears every 5 minutes in internal channels."*
>
> — Boris Cherny

> *"Anthropic's technical employees are affectionately known as 'ants,' and this is their version of dogfooding."*
>
> — Boris Cherny

Adoption Timeline

- Day 1: ~20% of Engineering
- Day 5: ~50% of Engineering
- Today: 70-80% daily active usage

> *"The Claude Code team has an unfair advantage: They get to watch hundreds of smart engineers use their product every single day, and all it takes is a stroll around their office."*
>
> — Boris Cherny

Productivity Impact

> *"Even though Anthropic has tripled since the start of the year, productivity per engineer has grown almost 70% per engineer because of Claude Code."*
>
> — Boris Cherny

Boris on personal productivity:

> *"For me, it's probably 2x my productivity. I think there's some engineers at Anthropic where it's probably 10x their productivity. And then there are some people that haven't really figured out how to use it yet."*
>
> — Boris Cherny

Code Generation Stats

- **80-90% of Claude Code** was written by Claude Code
- Some teams report **90% of code** written using Claude
- The team literally deletes code with each model upgrade

> *"Probably near 80%, yeah it's very high."*
>
> — Boris Cherny

Development Velocity

- 60-100 internal releases daily
- ~5 PRs per engineer daily
- 1 external release daily

- 20+ prototypes of todo-list feature in 2 days

Cost & ROI

> *"It's an ROI question, not a cost question. Engineers are expensive. And if you can make an engineer 50, 70% more productive, that's worth a lot."*
>
> — Boris Cherny

> *"~$6/day per active user; some internal users spend thousands daily."*
>
> — Boris Cherny

> *"Cost of writing code is going down and productivity is going up. The industry hasn't internalized the implications yet."*
>
> — Boris Cherny

Cat Wu's PM Philosophy

> *"I think I PM with a pretty light touch. I think Boris and the team are, like, extremely strong product thinkers. And for the vast majority of the features on our roadmap, it's actually just, like, people who have a lot of experience. People building the thing that they wish that the product had. So very little actually is tops down."*
>
> — Boris Cherny

Feature Development Approach

> *"We build most things that we think would improve Claude Code's capabilities, even if that means we'll have to get rid of it in three months. If anything, we hope that we will get rid of it in three months."*
>
> — Boris Cherny

PART 18

The Bitter Lesson Approach

Boris's Philosophy

> *"I think our approach to product is similar to our approach to the model, which is Bitter Lesson. So just freeform. Keep it really simple. Keep it close to the metal."*
>
> — Boris Cherny

What This Means in Practice

Architectural Simplicity

- One main loop, not multi-agent architecture
- Simple search, simple todolist
- Flat list of messages (apart from summarization)

> *"Claude Code is not a product as much as it's a Unix utility. We think of it as like a Unix utility. The same way that you would compose grep or cat."*
>
> — Boris Cherny

> *"All secret sauce is in the model. This is the thinnest possible wrapper over the model."*
>
> — Boris Cherny

"Unshipping" Tools

> *"We want to unship tools and kind of keep it simple for the model. Last week or two weeks ago, we unshipped the LS tool."*
>
> — Boris Cherny

Why: They built permission enforcement into bash itself, making the separate tool unnecessary.

> *"The model kind of tends to subsume everything over time."*
>
> — Boris Cherny

Resist Over-Engineering

> *"Resist the urge to over-engineer. Build good harness for the model, let it cook!"*
>
> — Boris Cherny

Design for Future Models

> *"Build for the model six months from now—not today's limitations."*
>
> — Boris Cherny

The team:

- Treats cost/latency gaps as temporary
- Prototypes ahead of capability
- Deletes scaffolding when models improve

Continuous Simplification

> *"We've rewritten it from scratch. Probably every three weeks, four weeks or something."*
>
> — Boris Cherny

> *"It's got simpler. It doesn't go more complex."*
>
> — Boris Cherny

PART 19

Power User Techniques

From Boris's Direct Tips

Terminal Mastery

- Number tabs 1-5 for parallel sessions
- Use iTerm2 notifications for alerts
- Set up aliases: `c` for claude, `c -c` for continue

Bash Mode

> *"The `!` prefix runs commands directly without Claude filtering."*
>
> — Boris Cherny

Context Efficiency

- Start fresh conversations for optimal performance
- Create HANDOFF.md before /clear
- Search history in `~/.claude/` using grep

Course Correction Tools

1. **Ask Claude to make a plan** before coding; confirm before proceeding
2. **Press Escape** to interrupt any phase (thinking, tool calls, edits) while preserving context

3. **Double-tap Escape** to jump back in history, edit previous prompts, explore alternatives
4. **Ask Claude to undo changes** to take a different approach

> *"Using these tools generally produces better solutions faster than perfect first attempts."*
>
> — Boris Cherny

Voice Input

Use voice transcription for faster communication—Claude understands context despite transcription errors

Brainstorming Mode

Instruct Claude to ask clarifying questions during planning, simulating a thought partner

Be Specific in Instructions

Poor: "add tests for foo.py" **Good**: "write a new test case for foo.py, covering the edge case where the user is logged out. avoid mocks"

Poor: "why does ExecutionFactory have such a weird api?" **Good**: "look through ExecutionFactory's git history and summarize how its api came to be"

> *"Specificity reduces course corrections and improves first-attempt success."*
>
> — Boris Cherny

Provide Rich Context

- **Images**: Paste screenshots (macOS: Cmd+Ctrl+Shift+4, paste with Ctrl+V)
- **Files**: Use tab-completion to reference files/folders
- **URLs**: Paste alongside prompts; use `/permissions` to allowlist domains
- **Data**: Pipe into Claude (`cat foo.txt | claude`)

Manual Exponential Backoff

For long-running tasks, check at 1, 2, 4+ minute intervals

The Billion Token Rule

> *"Consuming tokens across multiple sessions builds intuition faster than theory."*
>
> — Boris Cherny

Automation of Automation

> *"Progress toward automation of automation—automate repetitive tasks through scripts, skills, CLAUDE.md updates."*
>
> — Boris Cherny

PART 20

Quick Reference

Essential Commands

Command	Action
`Shift+Tab`	Toggle modes (default → auto → plan)
`Shift+Tab` twice	Enter Plan mode
`Esc`	Interrupt Claude
`Esc Esc`	Jump back in history / Rewind
`/clear`	Reset context
`/context`	Check context usage
`/compact`	Compress context
`/rewind`	Restore previous state
`/permissions`	Manage permissions
`/agents`	Create/manage subagents
`/hooks`	Configure hooks
`/chrome`	Chrome integration
`/mcp`	MCP server management
`!`	Enter bash mode
`@file`	Reference file in prompt
`#`	Add to CLAUDE.md
`claude -c`	Continue last session

Command	Action
`claude -p "prompt"`	Headless mode
`--teleport`	Move between web and CLI

Boris's Daily Commands

Command	Frequency
`/commit-push-pr`	Dozens/day
`/commit`	Multiple/day
`/code-review`	Every PR
`/security-review`	Every PR
`/feature-dev`	Complex features

Thinking Intensifiers

Phrase	Thinking Budget
"think"	Basic
"think hard"	More
"think harder"	Even more
"ultrathink"	Maximum

Model Selection

Model	Use Case
Opus 4.5	Boris's choice for everything
Sonnet 4.5	Fast, less steering needed
Haiku	Cost-efficient for simple tasks

The 3 Core Principles (Quick)

1. **Almost always use Plan mode** (Shift+Tab twice)
2. **Give Claude verification feedback loops** (tests, Chrome extension, simulators)
3. **Hold the same bar for human and Claude code** (use /code-review)

Key Resources

Official Documentation

- [Claude Code Best Practices](https://www.anthropic.com/engineering/claude-code-best-practices) - Anthropic's official guide
- [Hooks Reference](https://docs.anthropic.com/en/docs/claude-code/hooks) - Complete hooks documentation
- [SDK Overview](https://docs.anthropic.com/en/docs/claude-code/sdk) - Agent SDK documentation
- [IDE Integrations](https://docs.anthropic.com/en/docs/claude-code/ide-integrations) - VS Code, JetBrains setup
- [Subagents Documentation](https://code.claude.com/docs/en/sub-agents)

- [Chrome Extension](https://code.claude.com/docs/en/chrome)
- [Hooks Guide](https://code.claude.com/docs/en/hooks-guide)
- [Slash Commands](https://code.claude.com/docs/en/slash-commands)
- [Settings](https://code.claude.com/docs/en/settings)
- [Terminal Config](https://code.claude.com/docs/en/terminal-config)
- [GitHub Actions](https://code.claude.com/docs/en/github-actions)
- [Plugins Announcement](https://www.anthropic.com/news/claude-code-plugins)
- [Claude Code Settings](https://claude.ai/settings/claude-code) - Web settings panel

GitHub Repositories

- [Claude Code Main Repo](https://github.com/anthropics/claude-code) - Official source
- [**Plugins Directory**](https://github.com/anthropics/claude-code/tree/main/plugins) - All 13 official plugins
- [Plugin Marketplace JSON](https://github.com/anthropics/claude-code/blob/main/.claude-plugin/marketplace.json)
- [Plugins README](https://github.com/anthropics/claude-code/blob/main/plugins/README.md)
- [GitHub MCP Server](https://github.com/github/github-mcp-server) - Official GitHub MCP

Official Plugins (All 13)

Plugin	Link	Description
agent-sdk-dev	GitHub	Agent SDK development kit
claude-opus-4-5-migration	GitHub	Model migration tool
code-review	GitHub	5 parallel Sonnet reviewers
commit-commands	GitHub	Git workflow automation
explanatory-output-style	GitHub	Educational insights
feature-dev	GitHub	7-phase development workflow
frontend-design	GitHub	Production-grade UI guidance
hookify	GitHub	Custom hook creation
learning-output-style	GitHub	Interactive learning mode
plugin-dev	GitHub	Plugin development toolkit
pr-review-toolkit	GitHub	Comprehensive PR review
ralph-wiggum	GitHub	Autonomous iteration loops
security-guidance	GitHub	Security pattern monitoring

Boris's Content

- [Main Setup Thread](https://x.com/bcherny/status/2007179832300581177) - The original tweetstorm
- [259 PRs Thread](https://x.com/bcherny/status/2004887829252317325) - 30-day stats

- [Tips Reply to Karpathy](https://x.com/bcherny/status/2004711722926616680) - Mental model advice
- [Memory Leak Story](https://x.com/bcherny/status/2004626064187031831) - Debugging anecdote
- [Thread Reader Version](https://twitter-thread.com/t/2007179832300581177) - Easier to read format

Podcasts & Interviews

- [AI & I: How to Use Claude Code Like the People Who Built It] (https://every.to/podcast/how-to-use-claude-code-like-the-people-who-built-it)
- [AI & I Transcript](https://every.to/podcast/transcript-how-to-use-claude-code-like-the-people-who-built-it)
- [Latent Space: Claude Code](https://www.latent.space/p/claude-code)
- [Inside Claude Code From the Engineers Who Built It] (https://podcasts.apple.com/us/podcast/inside-claude-code-from-the-engineers-who-built-it/id1719789201?i=1000734060623)
- [Behind the Craft: Cat Wu Interview](https://creatoreconomy.so/p/inside-claude-code-how-an-ai-native-actually-works-cat-wu)

Community Resources

- [Awesome Claude Code Plugins](https://github.com/ccplugins/awesome-claude-code-plugins) - Curated plugin list
- [40+ Claude Code Tips](https://github.com/ykdojo/claude-code-tips) - Community tips collection
- [Awesome Claude Code](https://github.com/hesreallyhim/awesome-claude-code) - Resources & integrations
- [Awesome Claude Code (jmanhype)](https://github.com/jmanhype/awesome-claude-code) - Plugins, MCP servers, editors

Community Guides

- [Latent Space Podcast Summary](https://vlad.build/cc-pod/) - Visual summary
- [McKay Johns: Get the Most Out of Claude Code](https://mckayjohns.substack.com/p/get-the-most-out-of-claude-code)
- [Storm in the Castle: Power Tips](https://www.storminthecastle.com/posts/claude_tips/)
- [Pragmatic Engineer: How Claude Code is Built](https://newsletter.pragmaticengineer.com/p/how-claude-code-is-built)
- [Shipyard: CLI Cheatsheet](https://shipyard.build/blog/claude-code-cheat-sheet/)
- [Coder: Inside Anthropic's AI-First Development](https://coder.com/blog/inside-anthropics-ai-first-development)

The Bottom Line

Boris Cherny's approach combines:

- **Rigorous quality standards** (same bar for human and AI code)
- **Aggressive automation** (15+ parallel instances, Ralph Wiggum loops)
- **Verification-first methodology** (tests, visual verification, subagent review)
- **The Bitter Lesson** (simplicity over scaffolding, build for future models)
- **Compounding knowledge** (CLAUDE.md captures mistakes, slash commands automate patterns)
- **Throwaway development** (prototype first to learn, then build properly)
- **Continuous simplification** (unship tools, delete code as models improve)

The result: 259 PRs in 30 days, ~42-hour continuous sessions, 2x personal productivity (10x for some engineers), and 80-90% of code written by Claude Code itself.

Manual compiled from 60+ sources on January 4, 2026 Includes all Boris Cherny tweets, replies, podcast appearances, Hacker News comments, and official documentation