aloa
EST 2018

Claude Code 101 Guide

Getting the best out of Claude Code comes down to smart setup, intentional context management, and knowing which levers to pull. Not just using the tool itself.

Audience: Anyone using Claude Code for development work
Date: March 2026

Table of Contents

1 THE FOUNDATION

How Claude Code Actually Works

Claude Code can read your files, run commands, and interact with external systems. But here is the thing: language models can only process text. So how does any of that actually happen?

The tool use loop:

- You send a request
- Claude decides it needs to read a file and responds with a formatted request
- Claude Code intercepts that and actually reads the file
- The contents come back to Claude as text
- Claude reads the result and decides its next action
- This loop continues until the task is done

Every action Claude takes goes through this loop. It never directly touches your system. It requests, the tool executes, the result comes back as text.

KEY TAKEAWAY

The quality of this loop is what separates a good AI coding assistant from a great one. Claude is particularly strong at chaining tools together for complex multi-step tasks, knowing which tool to use, in what order, and how to interpret the results.

2 SETUP

Your First Session

The first thing most people do when they open Claude Code is type a request. The first thing you should actually do is run one command:

```
/init
```

This analyzes your entire codebase and creates a CLAUDE.md file - a briefing document Claude reads at the start of every single session. Without it, Claude starts every conversation knowing nothing about your project. With it, Claude always knows your architecture, your conventions, your key files.

Real world example:

You open Claude Code on a project you have been building for six months. Without /init, Claude produces code that does not match your patterns and puts files in the wrong places. With /init, Claude already knows your stack, your folder structure, and your conventions. The first thing it builds fits right in.

KEY TAKEAWAY

Run /init once per project. Commit the resulting CLAUDE.md to source control so your whole team benefits from it automatically.

3 HIGHEST LEVERAGE SKILL

Context is Everything

This is the highest leverage skill in Claude Code and the one most people get wrong.

More context is not always better. Flooding Claude with irrelevant information actively hurts performance. The goal is always just enough relevant context - not everything, not nothing.

The CLAUDE.md system - three levels:

- **Project level** (committed to source control) - shared with your whole team. Architecture decisions, coding standards, which libraries you use. Every developer benefits automatically.
- **Local level** (not committed) - your personal layer. Preferences your teammates do not need to share. Always applied for you, invisible to everyone else.
- **Machine level** (global, all projects) - your universal preferences. Set it once, applied everywhere you work.

Real world example:

Your team CLAUDE.md says 'we use Prisma for database access and Vitest for testing.' Your local CLAUDE.md says 'always write tests alongside new functions.' Your machine CLAUDE.md says 'always use TypeScript strict mode.' Claude walks into every session knowing all three layers without you saying a word.

The @ symbol - surgical file context:

Use @ to point Claude at specific files for a task instead of letting it search your whole codebase.

- Adding a new API route: @src/routes/users.js so Claude matches your existing patterns
- Fixing a database bug: @src/db/queries.js so Claude sees how you actually query data
- Updating a component: @src/components/Button.tsx so Claude understands your design patterns

The # shortcut:

Type # followed by a natural language instruction to update your CLAUDE.md files on the fly. Use this mid-session the moment you notice something you want Claude to remember.

KEY TAKEAWAY

You are the context curator. Pointing Claude at the right files before writing your prompt takes five seconds and meaningfully changes what comes back.

4 WORKFLOW

Getting the Best Work Out of Claude

Screenshots with Ctrl+V:

Instead of describing UI problems in words, paste a screenshot directly into Claude Code with Ctrl+V. Note: on Mac this is Ctrl+V not Cmd+V - this catches people out.

This means non-technical teammates like designers and PMs can participate in Claude Code workflows without knowing anything about code. Show Claude exactly what you see instead of trying to describe it.

Plan Mode vs Thinking Mode:

These are two separate dials you can turn up when you need Claude to work harder.

- **Plan Mode** (Shift+Tab twice) - tells Claude to explore widely before acting. Maps your codebase, understands dependencies, creates a detailed plan before writing any code. Use for multi-step tasks touching many files.
- **Thinking Mode** (language triggered) - tells Claude to reason deeply about a specific problem. The intensity scales with your words:
  - "think" - a step up from default
  - "think hard" - more deliberate
  - "think harder" - deeper still
  - "ultra think" - maximum reasoning for genuinely hard problems
- **Plan Mode = breadth.** Refactor our authentication system across the whole codebase.
- **Thinking Mode = depth.** Ultra think through why this race condition only appears under load.
- **Both together** = complex tasks needing wide understanding and deep reasoning.

KEY TAKEAWAY

Both modes use more tokens. Use them intentionally, not by default. The words you use literally change how much cognitive budget Claude allocates to your problem.

5 CONTEXT CONTROL

Managing Claude Over Time

Most people treat each Claude session like a fresh conversation. The real skill is managing Claude across time - teaching it your preferences, recovering from wrong turns, and keeping sessions clean as they get longer.

- **Escape** - stop Claude mid-execution the moment it goes in the wrong direction. Do not wait for it to finish.

- **Escape + #** - the most underrated combination. Stop Claude, immediately teach it something. Instead of correcting the same mistake repeatedly, you burn the correction into memory once.

Real world example:

Claude keeps rewriting your utility functions instead of importing them. Hit Escape the moment you see it, then # 'never rewrite existing utility functions, always import from /utils.' It will not happen again.

- **Double Escape** - rewind to a previous point in the conversation. Not just undo - actual navigation back to a known good state. If a debugging session went wrong at message 15, jump back there and go a different direction.

- **/compact** - summarizes conversation history while preserving everything Claude has learned. Use when a session is long and cluttered but Claude has built up genuine expertise you want to keep.

- **/clear** - full reset. Use when switching to a completely unrelated task.

KEY TAKEAWAY

The mindset shift: instead of 'Claude got it wrong again' think 'I have not taught Claude my preferences yet.' The # shortcut is how you teach it. Do it once, never repeat it.

6 SECURITY - DO THIS FIRST

Protect Your Secrets Before You Do Anything Else

Before pointing Claude at a real project, do one thing: protect your .env file.

Claude is thorough. If reading your database credentials helps it understand a connection issue, it will read your .env file. Not maliciously - it is just doing its job. But thorough plus secrets is a problem.

The fix - add to .claude/settings.local.json:

```
{ "hooks": { "PreToolUse": [{ "matcher": "Read|Grep", "hooks": [{ "type": "command",
"command": "node ./hooks/env_protect.js" }] }] } }
```

Create ./hooks/env_protect.js:

```
if (input.tool_input?.path?.includes('.env')) { console.error('Access to .env files
is blocked.'); process.exit(2); // exit code 2 = block } process.exit(0); // exit code
0 = allow
```

Restart Claude after adding this. You do not need to understand the full hooks system to do this one thing. Just set it up.

KEY TAKEAWAY

Exit code 2 blocks the operation and sends your error message back to Claude. Exit code 0 lets it proceed. This is beginner-friendly despite using hooks - set it up before you work on any real project.

7 AUTOMATION

Custom Commands

If you are explaining the same multi-step process to Claude more than once, you are doing extra work. Custom commands encode any workflow into a single slash command.

How it works:

- Create a folder at .claude/commands/ in your project
- Any markdown file you put there becomes a command
- audit.md creates /audit, write_tests.md creates /write_tests
- Restart Claude after creating new commands or they will not appear

The $ARGUMENTS placeholder:

Whatever you type after the command name gets inserted wherever $ARGUMENTS appears, making commands flexible and reusable.

Real world example - /write_tests command:

```
Write comprehensive tests for: $ARGUMENTS Testing conventions: - Use Vitest with React
Testing Library - Place test files in __tests__ directory - Test happy paths, edge
cases, and error states
```

Run with: /write_tests the useAuth hook in src/hooks

KEY TAKEAWAY

A good custom command is institutional knowledge turned into a repeatable workflow. Your conventions, file structure, testing patterns - encoded once, available to everyone on the team forever.

8 GAME CHANGER

Extending Claude with MCP Servers

Claude Code's default tools cover the basics. MCP servers extend what Claude can actually do in the world. The default toolset is a starting point, not a limit.

Installing Playwright (the most valuable one to start with):

```
claude mcp add playwright npx @playwright/mcp@latest
```

Run this in your terminal, not inside Claude Code.

Pre-approving permissions in .claude/settings.local.json:

```
{ "permissions": { "allow": ["mcp__playwright"], "deny": [] } }
```

Note the double underscores in mcp__playwright. This pre-approves Playwright tools so Claude does not ask permission every time. Permission friction kills automated workflows.

Why Playwright is the biggest unlock:

Before Playwright: Claude writes code and assumes it works. With Playwright: Claude can actually verify it works.

Claude builds a feature, opens the browser, walks through it as a user would, finds what is broken, fixes it, verifies the fix. You receive something that actually works rather than something that needs another review round.

KEY TAKEAWAY

Playwright gives Claude the full developer loop: write, test, see what breaks, fix, verify. This is the difference between Claude as a code writer and Claude as a developer.

GitHub Integration

Everything up to this point makes Claude more powerful for individual developers. GitHub integration makes Claude a member of your team - always on, embedded in your existing workflow.

Setup:

```
Run /install-github-app inside Claude Code
```

This installs the Claude Code app on GitHub, adds your API key, and generates a PR with two GitHub Actions.

The two default actions:

- **Mention support** - tag @claude in any GitHub issue or PR. Claude reads the issue, explores your codebase, implements the fix, and opens a PR. No developer needs to relay the request.
- **Automatic PR reviews** - every new PR gets reviewed automatically. Claude traces the impact of changes and posts a detailed report, catching things humans miss.

Real world example:

A developer adds user email to a Lambda function output without realizing that function feeds data to an external partner. Claude traces the entire infrastructure flow, identifies that PII is now being shared externally, and flags it before the PR merges.

The YML file - where the real power lives:

```
custom_instructions: | Server is running at localhost:3000. Logs are in logs.txt.
allowed_tools: "Bash(npm:*),mcp__playwright__browser_snapshot"
```

Every tool must be explicitly listed. No wildcards, no shortcuts. This is intentional - you define exactly what Claude is allowed to do inside your CI pipeline.

KEY TAKEAWAY

GitHub integration turns Claude from a local tool into team infrastructure. Tag @claude in an issue and the work gets done without anyone context switching into Claude Code.

10 ADVANCED

Hooks and the SDK

You do not need this to get value from Claude Code. But when you are ready to go further, this is where significant capability lives.

Hooks: Making Claude Self-Correcting

Hooks are scripts that run automatically before or after Claude executes any tool. Set them up once, they run forever.

- **PreToolUse hooks** - run before a tool executes. Can block the operation with exit code 2 or allow with exit code 0.
- **PostToolUse hooks** - run after a tool executes. Cannot block but can run follow-up operations and feed results back to Claude.

The TypeScript type checker hook:

Claude changes a function signature but misses the call sites. A PostToolUse hook runs tsc --no-emit after every TypeScript file edit. Type errors get fed back to Claude automatically. Claude fixes the call sites. You never had to get involved.

The duplicate code prevention hook:

Claude creates a new database query that already exists elsewhere. A PostToolUse hook launches a separate Claude instance to compare new code against existing code. If a duplicate is found, exit code 2 blocks it and Claude gets told what already exists and where.

The Claude Code SDK:

The SDK gives you programmatic access to Claude Code via TypeScript or Python. Use it to embed Claude intelligence into automated pipelines.

```
import { query } from '@anthropic-ai/claude-code'; for await (const message of query({
prompt: 'Find duplicate queries in ./src/queries', options: { allowedTools: ['Edit']
} })) { console.log(message); }
```

By default the SDK is read-only. Specify allowedTools to enable write access. This is where Claude stops being a tool you use and becomes infrastructure that runs.

KEY TAKEAWAY

Hooks turn Claude's known weaknesses into self-correcting loops. Identify a recurring problem, encode the check once, Claude course-corrects automatically every time.

Ready to level up your development workflow?

Start with the highest-impact items: run /init on your first project, protect your .env file, and add the Playwright MCP server. These three steps alone will dramatically change how you work with Claude Code.

Learn more at **aloa.co**

aloa

EST 2018