

Coding with *four robots,* one terminal.

The plain-English guide to becoming productive with Claude Code, Gemini Code Assist, Gemini CLI, and Cursor — without drowning in documentation.

● Claude Code

● Gemini Code Assist

● Gemini CLI

● Cursor

What's inside

- 01 Meet the Cast
- 02 A Closer Look at Cursor
- 03 Writing Code
- 04 Understanding Code
- 05 Fixing Things
- 06 Testing & Docs
- 07 The Terminal
- 08 Giving Context
- 09 Plan Mode
- 10 Agent & Auto Mode
- 11 Reviewing Changes
- 12 Undo & Recovery
- 13 Commands Cheat Sheet
- 14 Subagents
- 15 Connecting Your Tools (MCP)
- 16 SKILL.md — Teaching the Robot

17 Spec-Driven Development

18 The Capstone

★ Glossary

The four robots

Before we open a terminal, let's get to know who we're working with. Four AI assistants — each with its own personality and home turf.

A LITTLE STORY

Imagine you're moving into a new house. You hire four helpers. The first lives in your **workshop** and helps you build things on the spot. The second is a **terminal generalist** you can chat with anywhere. The third is a **terminal specialist** who handles the hardest jobs. The fourth is a **full-time co-worker** who completely takes over your editor. Four robots, four roles.

- GEMINI CODE ASSIST

The **workshop helper**. Lives inside VS Code and IntelliJ as a plugin. Whispers code suggestions as you type, answers questions in a chat panel. Best for: Google Cloud teams who want AI added to their existing IDE.
- GEMINI CLI

The **terminal generalist**. Free to start, fast, runs in your shell. Comes from Google. Best for: scripting, quick experiments, and developers who already live in the terminal.
- CLAUDE CODE

The **terminal specialist**. Anthropic's terminal agent — known for thinking carefully before acting, planning, and handling messy multi-file work. Best for: complex changes you'd normally pair-program.
- CURSOR

The **full-time co-worker**. A complete IDE (VS Code fork) rebuilt around AI. Tab autocomplete, Composer for multi-file edits, Agent Mode for autonomous work — all first-class. Best for: developers who want AI baked into every keystroke.

THE BIG IDEA

"You don't pick the best tool. You pick the right tool for what you're doing right now."

One more distinction worth knowing: the four split into two camps.

CAMP	TOOLS	VIBE
IDE-based	Gemini Code Assist, Cursor	You stay in your visual editor. AI lives in panels, popups, and inline suggestions.
Terminal-based	Claude Code, Gemini CLI	You chat with the AI in a shell window. Fewer visuals, more conversation.



T OVERTHINK IT

Most developers end up using **one IDE-based tool + one terminal-based tool** together. The IDE tool handles inline work and visual debugging; the terminal tool handles bigger refactors and automation. Pick one of each, learn them well, then explore.

Quick check

You're refactoring a single React component and want suggestions while you type. Which robots are the right fit?

A. Claude Code or Gemini CLI

B. Gemini Code Assist or Cursor ✓

C. Any of them, doesn't matter

Right! Inline-while-you-type is the IDE-based tools' specialty. Gemini Code Assist and Cursor both shine here — Claude Code and Gemini CLI live in the terminal and aren't optimized for tab-completion-as-you-type.

A closer look at *Cursor*

Cursor deserves its own chapter because it's structurally different. The other tools add AI to what you're doing. Cursor rebuilds the editor around AI. That's a bigger shift than it sounds.

WHY IT'S DIFFERENT

Open Cursor and it looks almost exactly like VS Code — because it is, technically. It's a fork. But every surface of the editor has been re-engineered: the autocomplete, the chat, the diff viewer, the agent runner. You aren't bolting AI onto an IDE. You're using an IDE that **only exists because of AI**.

There are four ways Cursor expects you to work with it. Learning these four modes is essentially learning the tool.

MODE	SHORTCUT	WHAT IT'S FOR
Tab	Tab key	Predictive autocomplete. Cursor guesses your next several lines, sometimes across files.
Inline Edit	Cmd+K (Ctrl+K)	Select code, describe a change, get a focused edit in place. Like a surgical strike.
Composer	Cmd+I (Ctrl+I)	Multi-file edits. "Add dark mode across all components" → coordinated diff across many files.
Agent Mode	Toggle in Composer panel	Cursor runs autonomously — edits files, runs terminal commands, watches output, iterates.

THE AUTONOMY SLIDER

Tab is "AI predicts the next keystroke." Agent Mode is "AI builds a feature while I get coffee." Everything in Cursor sits on that slider — you pick where on the slider you want to live for any given task.

Multi-model picker. Unlike the other tools, Cursor lets you swap between Claude Opus 4.7, GPT-5.5, Gemini 3.1 Pro, Grok 4, and Cursor's own Composer-2.5 model — all from a dropdown. You can use Claude for planning and GPT-5 for fast edits in the same session.

Background Agents (new in v3). Cursor can now run agents in the background while you keep working. Hand off "upgrade all dependencies and fix the breakages" and keep coding. You get a notification when it's done.

Mission Control. Cursor lets you run up to **eight agents in parallel** using Git worktrees. Useful for trying the same task with different models and comparing results, or working on three features at once.

BugBot. Cursor's GitHub-integrated bot that reviews pull requests automatically. Bring it into your repo and it leaves comments on PRs like a human reviewer would.

A typical Cursor day looks like this:

Morning: small bug fix

1. Open the file. Press **Cmd+K**.
2. Type "fix the off-by-one in this loop."
3. Accept the diff. Done in 30 seconds.

Afternoon: new feature

1. Press **Cmd+I** to open Composer.
2. "Add a settings page with theme toggle, language picker, and notification preferences."
3. Cursor opens 6 files in a coordinated diff. Review each.

Friday cleanup: hand off to background

1. Open Agent panel. Toggle "Background."
2. "Add proper TypeScript types everywhere we still have any."
3. Close the laptop. Come back to a PR.



THE HONEST TRADEOFFS

Cursor is paid: free tier exists, but real work needs **Pro at \$20/month** and metered usage on top. It's cloud-only by default (your code goes to model providers). And it works best on small-to-medium codebases — large monorepos (50k+ files) slow it down. If those tradeoffs don't fit, Claude Code or Gemini CLI might be better.

► TRY THIS

Install Cursor (free download from cursor.com). Open any project. Try one thing in each mode:

1. **Tab**: start typing a function — accept what Cursor predicts.
2. **Cmd+K**: select a function and type "add error handling."
3. **Cmd+I**: ask Composer to "add a dark mode toggle."
4. **Agent**: say "find and fix any TODOs in this project."

Twenty minutes total. You'll have a feel for whether Cursor fits your style.

Writing *code* with words

The most basic skill: describe what you want in plain English, get working code back. This is where you spend the first hour learning, and you'll use it forever.

All four tools do this. The differences are mostly about where you ask the question and how much they do on their own.



PRO TIP

Be specific. "Make a function" gets you generic. "Make a function that validates an email using a regex and returns a tuple of (is_valid, error_message)" gets you something usable. Specificity is the cheapest upgrade you'll ever buy.

In Gemini Code Assist: just start typing. Suggestions appear in grey. Press Tab to accept. Or open the chat panel and type a prompt like `"generate a Python function that..."`

In Cursor: three options. Tab (predictive autocomplete), Cmd+K (inline edit on selected code), or Cmd+I (Composer for bigger asks).

In Gemini CLI or Claude Code: type your request directly into the terminal:

TERMINAL

```
> Write a Python function that takes a CSV file path, returns  
  a dictionary mapping each column name to its data type.  
# The robot reads context, writes the function, asks if you want it saved.
```

► TRY THIS

In any of the four tools, ask: *"Write me a function that converts a temperature between Celsius, Fahrenheit, and Kelvin. Include input validation."* Watch what you get. Note: every robot interprets the same prompt slightly differently. That's normal.

Reading *unfamiliar* code

Inherited a project from someone who left the company? Spent an hour staring at a 400-line function? This is where AI earns its keep.

A SCENE FROM YOUR FUTURE

It's Tuesday. You open a file called `legacy_billing.py`. It has no comments. The variables are named `x1`, `x2`, and — for some reason — `banana`. You ask the robot: **"Explain what this file does in plain English, like I'm new to the project."** Two minutes later, you're caught up.

Three ways to use this:

- **"Explain this file."** Plain-English walkthrough of a single file.
- **"What does this function do? Walk me through line by line."** Deep dive on one tricky piece.
- **"How does authentication work in this project?"** The big one — the robot reads multiple files and traces connections.

In Gemini Code Assist, there's a special trick: **Outlines**. Open any file and you get a high-level summary right in the editor — no prompt needed.

MENTAL MODEL

Think of the AI as a smart colleague who reads ten times faster than you do. Your job is to ask the right question.

When something is *broken*

Bugs are where AI tools shine the brightest. They never get frustrated, they read carefully, and they don't get stuck on the obvious thing you've been staring at.

The pattern that works:

TERMINAL

> I'm getting this error when I run `npm test`:

```
TypeError: Cannot read property 'map' of undefined
    at UserList.render (UserList.jsx:14)
```

Look at `UserList.jsx` and figure out what's wrong. Don't fix it yet — just tell me what's happening.



THE "DIAGNOSE FIRST" RULE

Always ask the robot to **diagnose** before fixing. Two reasons: (1) you learn what went wrong, and (2) sometimes the AI's diagnosis is wrong, and you'd rather catch that before it rewrites 40 lines.

► TRY THIS

Find a project on your machine where the tests fail. Show the failure to the AI. Ask: *"Explain what's failing and why. Then suggest the smallest possible fix."* Then read the suggestion carefully before approving.

Quick check

Your test is failing. What's the better first prompt?

A. "Explain what's failing and why before fixing it." ✓

B. "Just fix it, I'm in a hurry."

Right. "Diagnose first" forces the AI to explain its thinking — and lets you catch a wrong diagnosis before it rewrites a bunch of code.

Tests and *documentation*

If there's one thing developers procrastinate on, it's tests and docs. Lucky for us, the robots actively enjoy this work.

Tests. Point at a file and ask:

TERMINAL

```
> Generate unit tests for every function in src/utils/validators.py.  
Use pytest. Include edge cases – empty strings, None, very long  
inputs, special characters.
```

Docs. The robots will happily write:

- Inline comments and docstrings
- README files based on what your code actually does
- API reference docs from your function signatures
- Architecture summaries (great for onboarding new teammates)



HONEST MOMENT

AI-generated tests are a starting point, not the finish line. Read them. Some will be excellent. Some will test the wrong thing. Treat them like tests written by a smart but distracted junior developer.

DAY TWO • THE TERMINAL AWAKENS

Living in the terminal

Gemini CLI and Claude Code both live in your terminal. If you've never spent serious time there, this is your moment.

The shape of a session is the same in both tools:

TERMINAL

```
# 1. Open your terminal, navigate to a project folder
$ cd ~/my-project

# 2. Start the assistant (one of these)
$ claude
$ gemini

# 3. Now you're chatting. Just type.
> What does this project do?
```

A few moves you'll do constantly:

YOU WANT TO...	TYPE
Point to a file	@filename.py
Run a shell command	Start with ! (eg. !ls -la)
Get help	/help
Clear the conversation	/clear
Quit	/exit or Ctrl+D

► TRY THIS

Open a terminal. Navigate to any project. Run `claude` or `gemini`. Type `/help`. Read what comes back. Then ask: *"What files are in this project, and what's the language breakdown?"*

Giving the robot *context*

This is the single most underrated thing in this whole guide. A robot with context is a senior engineer. A robot without context is a guessing game.

THE FUNDAMENTAL TRUTH

"The AI only knows what's in front of it. If it's making bad decisions, the fix is almost always more context."

Three ways to feed the robot context:

1. **The project memory file.** Create a file at the root of your project:

- For Claude Code → `CLAUDE.md`
- For Gemini CLI → `GEMINI.md`
- For Cursor → files in `.cursor/rules/` (or `agents.md` as a cross-IDE fallback)
- For Gemini Code Assist → context comes from your indexed repo (Enterprise)

The robot reads these every time it starts. What to put in them:

TERMINAL

```
# What I am
A Node.js + TypeScript API for a bookstore. Postgres for storage.

# Rules
- All async functions use try/catch with logging
- Tests live next to the file they test, with .test.ts suffix
- No 'any' types unless absolutely required
- Use kebab-case for filenames, camelCase for variables

# How to run things
- npm run dev (start the API)
- npm test (run tests)
- npm run lint (before committing)
```



CURSOR'S TWIST ON RULES

Instead of one big `.cursorrules` file, Cursor now uses a folder: `.cursor/rules/`. Each file in there is a focused rule that can be **scoped** — applied always, or only to certain file types or paths. So a rule about React components only fires when you're editing React files. Smaller context, smarter targeting.

2. **Point to specific files** with `@` when you ask a question. (Cursor also supports `@web`, `@docs`, and `@codebase` for broader scopes.)

3. **Drop in images.** Screenshot of a UI bug? A diagram? Paste it directly into the chat — all four tools accept images.



FIVE-MINUTE INVESTMENT, LIFETIME PAYOFF

Writing a good context file takes five minutes. It pays off every single session forever. Don't skip it — regardless of which tool you're using.

Plan Mode

The most important habit you'll learn this week. Plan Mode says: don't change anything yet — just tell me what you'd do.

WHY IT MATTERS

You ask the robot to "refactor the auth module to use JWT tokens." Without Plan Mode, it dives in, modifies eleven files, and now you have to read all eleven changes to know if it's right. With Plan Mode, it writes you **a numbered list of what it intends to do** — and waits. You read the plan in two minutes, fix the one assumption that was wrong, then say go.

Available in: Claude Code (built in), Gemini CLI (added Feb 2026), Gemini Code Assist (Agent Mode always shows a plan first), and Cursor (Composer and Agent Mode both show a multi-file plan before applying).

TERMINAL

```
> /plan Refactor the authentication module to use JWT tokens
instead of session cookies.
```

```
# The robot now responds with a numbered plan.
```

```
# You can edit it, ask for changes, or approve and let it run.
```

Quick check

When should you skip Plan Mode?

A. Never — always plan first.

B. For tiny, low-risk changes ("rename this variable"). ✓

C. When you're tired of approving things.

Right. Plan Mode shines on multi-file, high-risk work. For "rename this one variable" or "add this missing import", planning would slow you down.

Agent Mode & Auto Mode

There's a point where babysitting every approval becomes the bottleneck. That's where these modes come in. They're powerful, and they require a different kind of attention.

MODE	WHAT IT DOES	FOUND IN
Agent Mode	Give the robot a goal, it executes the steps across multiple files. Plan shown by default.	All four
Auto Mode	Safe actions run without asking. Risky actions still pause. Built-in safety checker watches in the background.	Claude Code
Auto Approve	The agent acts on your behalf — no pause at each step. Use with caution.	Gemini Code Assist
Background Agents	The agent works in a background thread while you keep coding. You get notified when it's done.	Cursor
Mission Control	Run up to 8 agents in parallel using Git worktrees. Try the same task with different models, or work on different tasks at once.	Cursor



A SAFETY HABIT WORTH KEEPING

Always run hands-free modes on a **feature branch**, not main. That way, if the robot does something weird, `git checkout main` undoes everything.

WHEN THIS IS MAGICAL

You're cleaning up tech debt on a Friday afternoon. You ask: **"Find every place we still use the old requests library and migrate them to httpx with proper async support."** You flip on Auto Mode, go get coffee, come back fifteen minutes later. The work is done across nine files, and there's a summary of every change.

Reviewing *every change*

The robots will show you what they changed. Your job is to actually look.

Every tool shows changes as a **diff**: green for added lines, red for removed. You can approve, reject, or edit before the change lands.

THE DISCIPLINE

Rubber-stamping AI changes is how bugs ship. Spend 30 seconds reading the diff before you press accept.

Look out for these patterns:

- Changes to files you didn't expect
- "Helpful" additions like new dependencies or imports you don't recognize
- Deleted code that you actually wanted to keep
- Subtle logic changes hidden inside what looks like a rename

► TRY THIS

Next time the robot proposes a change, open the diff and look at **one thing**: did it modify any file you didn't expect? If yes, ask "why did you touch that file?" before approving.

Undo, recover, resume

Things will go sideways. Here's how to recover.

Gemini CLI's checkpointing — every time the AI modifies a file, it makes a hidden snapshot. To roll back:

TERMINAL

```
> /restore
```

```
# Shows you a list of recent snapshots. Pick one, your project  
# goes back to that exact state. Magic.
```

Claude Code's session recap — close your laptop, come back tomorrow, type `--continue` or `/recap`, and the robot catches you up.

Git is still your best friend. Before any big session, commit your current work. If things go wrong, `git reset` beats everything.



THE "ALWAYS COMMIT FIRST" RULE

Before you ask an AI to do anything substantial, run `git status` and commit anything you've been working on. That five-second habit will save you a panicked hour someday.

DAY THREE • GOING FURTHER

The *commands* you'll actually use

Bookmark this section. These are the commands and shortcuts you'll reach for daily.

Slash commands (terminal tools and IDE chat):

COMMAND	WHAT IT DOES	WORKS IN
/help	Show all available commands	All four
/clear	Reset the conversation, keep project context	Claude Code, Gemini CLI
/plan	Switch to planning mode	Claude Code, Gemini CLI
/restore	Roll back to a snapshot	Gemini CLI
/cost	Show how much this session has cost	Claude Code
/model	Switch between Opus, Sonnet, Haiku, etc.	Claude Code
/usage	See what's pushing you toward limits	Claude Code
/effort	Dial reasoning depth (low → xhigh)	Claude Code
/recap	Get a summary when returning to a session	Claude Code
/memory	Manage saved context	Gemini CLI
/fix	Quick fix on the selected code	Gemini Code Assist
/explain	Explain the selected code	Gemini Code Assist
/tests	Generate tests for selected code	Gemini Code Assist
/doc	Generate documentation	Gemini Code Assist

Cursor keyboard shortcuts (memorize these four):

SHORTCUT	WHAT IT DOES
Tab	Accept predicted autocomplete
Cmd+K / Ctrl+K	Inline Edit — make a focused change on selected code
Cmd+L / Ctrl+L	Open the Chat panel
Cmd+I / Ctrl+I	Open Composer for multi-file edits and Agent Mode
@web / @docs / @codebase	Pull in extra context — web search, docs, or your whole repo

Subagents

Sometimes the main robot delegates. Subagents are mini-helpers that take on a specific job — researching a codebase, writing tests, reviewing security — and report back.

WHY THIS MATTERS

Picture this: you ask the main robot to "implement a new payment feature." Behind the scenes, it might send a **code investigator** to map out where the payment logic lives, a **test writer** to handle test coverage, and a **security reviewer** to check for obvious holes — all working in parallel. You get the result, without the main conversation getting cluttered.

Built-in specialists you can call directly:

TERMINAL

```
# Gemini CLI – explicit delegation with @ syntax
> @codebase_investigator How does authentication flow through
  this app, end to end?

# Claude Code – manage sub-agents
> /agents
```

You can also write your own custom subagents — Markdown files describing their purpose and the tools they're allowed to use. Both Claude Code and Gemini CLI support this.



WHEN TO USE THEM

You don't need subagents on day one. Reach for them when (a) your main conversation is getting cluttered, or (b) you have repetitive specialist work — code review, test generation, security checks — that your whole team would benefit from.

Connecting your *real tools* (MCP)

Your robot is useful by itself. It becomes a different beast when it can talk to GitHub, Jira, Slack, your database, and your internal systems. The plumbing is called MCP — Model Context Protocol.

What it unlocks: Instead of "let me copy this Jira ticket text into the chat," you say "look at JIRA-1234 and start implementing." Instead of "here's what my database schema looks like," you say "query the users table and tell me how many signed up last week."

Setup pattern (Claude Code shown; Gemini CLI, Cursor, and Gemini Code Assist are similar — all four support MCP):

TERMINAL

```
# Add a connector
$ claude mcp add github

# Follow the OAuth prompt in your browser. Done.
# Now in your session:

> List my open GitHub issues with the "bug" label,
  sorted by oldest first.
```

THE SHIFT

Without MCP: the AI is a coding helper. With MCP: the AI is a teammate who can see your work alongside you.

► TRY THIS

Connect **one** tool. Maybe GitHub if you use it daily, or Google Drive if your team lives in docs. Then ask the robot to summarize your last week of activity in that tool. Watch what happens.

SKILL.md — your custom playbooks

Subagents are who does the work. Skills are how the work should be done. A SKILL.md file is a little instruction sheet you give the robot — and it shows up exactly when needed.

THE LIGHTBULB MOMENT

Your team has a strict process for code reviews: check security first, then performance, then style. You explain this every single time. With a SKILL.md, you write it once. The next time anyone says **"review this PR"**, the robot follows your team's exact process — no reminder needed. The skill activates itself.

THE BIG IDEA

CLAUDE.md is loaded every session — it's always there. Skills are loaded only when relevant. Use skills for specialist knowledge you don't need 90% of the time.

Anatomy of a skill. It's just a folder with a SKILL.md file inside:

TERMINAL

```
.agents/skills/code-reviewer/
├── SKILL.md           # the instructions
├── scripts/           # optional: scripts the robot can run
│   └── lint.sh
└── references/        # optional: example files, templates
    └── good-pr.md
```

What goes in SKILL.md. Two parts: the frontmatter at the top, instructions below.

TERMINAL

```
---
name: code-reviewer
description: Reviews code changes for security, performance, and style.
  Use when the user asks to review a PR or diff.
---

# When activated, follow this process:

1. Run git diff to see the changes
2. Check for security issues first (input validation, secrets)
3. Then performance (N+1 queries, unbounded loops)
4. Finally style (naming, comments, formatting)
5. Write a summary with one section per category
```



THE MAGIC WORD: "DESCRIPTION"

The description field is the trigger. It tells the robot when to activate this skill. Be specific. *"Used for things"* is bad. *"Use when the user asks to review a pull request, diff, or recent changes"* is good. The robot reads descriptions of all your skills at session start and picks the right one when your prompt matches.

Where to put skills. Same idea across the agentic tools:

LOCATION	SCOPE	WHEN TO USE
.agents/skills/	Project (portable across tools)	Recommended. Works in Claude Code and Gemini CLI from the same folder.
.claude/skills/	Project, Claude only	Claude-specific features you don't want in Gemini CLI.
.gemini/skills/	Project, Gemini only	Gemini-specific features.
.cursor/skills/	Project, Cursor only	Cursor's skills format — accessible from Composer and Agent Mode.
~/ .agents/skills/	Personal (every project)	Your own habits — commit messages style, etc.

THE PORTABILITY MOVE

Use .agents/skills/. Your team's skill folder works in both Claude Code and Gemini CLI — no copy-pasting between tools.

The lifecycle. Here's what actually happens behind the scenes:

1. **Discovery.** When you start a session, the robot scans your skill folders and reads just the name and description of each — not the whole file. Saves on context.
2. **Activation.** When your prompt matches a skill's description, the robot asks: *"Can I load the code-reviewer skill?"*
3. **You consent.** Type y to approve. The full SKILL.md content gets loaded.
4. **Execution.** The robot follows the skill's instructions until the job is done.

► TRY THIS

Create your first skill in five minutes:

1. In any project, make a folder: `mkdir -p .agents/skills/commit-helper`
2. Create SKILL.md inside it with frontmatter (name + description) and instructions: *"When the user asks for a commit message, write a conventional commit (feat:, fix:, refactor:) under 72 characters, with a one-paragraph body explaining the why."*
3. Start Claude Code or Gemini CLI, say "help me write a commit message for these changes." Watch the skill activate.



SHORTCUT FOR THE LAZY

Don't want to write SKILL.md by hand? Both tools have a built-in **skill-creator**. Just ask: *"Create a new skill called 'pr-reviewer' that does X, Y, Z."* It generates the whole folder structure for you.

WHY THIS SCALES

When you're solo, skills feel like a productivity tweak. When you're on a team of ten, they're transformative. Commit one shared `.agents/skills/` folder to your repo — and every developer's AI assistant suddenly follows the same review process, generates the same commit format, scaffolds the same component structure. Your team's institutional knowledge becomes **executable**, not buried in a wiki.

Quick check

What's the difference between CLAUDE.md and a SKILL.md file?

- A. There isn't one — they're the same thing.
- B. CLAUDE.md is always loaded; skills load only when their description matches the prompt. ✓
- C. CLAUDE.md is for Claude; SKILL.md is for Gemini.

Right. CLAUDE.md is your project's always-on background. Skills are on-demand specialists. You'd use CLAUDE.md for "we use TypeScript" and a skill for "here's how we do code reviews" — because you don't need review instructions when you're writing a quick prototype.

Spec-driven *development*

The most reliable way to use AI for real work. Instead of "go build me a feature," you write a spec — a clear description of what should exist — and the robot builds from that. Boring? Maybe. Reliable? Absolutely.

A TALE OF TWO DEVELOPERS

Developer A says: *"Build me a user authentication system."* Three hours later, they have something — but it's not quite what they wanted, uses a library they hate, and missed a key requirement. Developer B writes a one-page spec: **endpoints, auth method, error responses, edge cases.** Then says: *"Implement this."* Three hours later, they have exactly what they specified. Same time, vastly different outcomes.

THE SHIFT

"Stop treating the AI like a magic wand. Treat it like a contractor. Contractors need blueprints."

The three-step rhythm:

STEP	WHAT YOU DO	WHAT THE AI DOES
1. Spec	Describe what should exist — features, contracts, edge cases.	Helps you draft and challenge the spec.
2. Plan	Approve the implementation plan derived from the spec.	Breaks the spec into a sequence of concrete tasks.
3. Build	Review and ship.	Implements task by task, checking against the spec.

What a spec looks like. Not a 50-page document. A focused markdown file:

TERMINAL

```
# Spec: User Profile API

## What this does
REST API for users to view and update their profile.

## Endpoints
- GET /api/profile          → returns current user's profile
- PATCH /api/profile        → updates fields the user owns
- POST /api/profile/avatar  → uploads a profile picture

## Rules
- Auth required on all endpoints (JWT bearer token)
- Users can only edit their own profile
- Avatars: max 5MB, must be PNG or JPEG
- Email changes require re-verification

## Error cases to handle
- 401 if no token → returns { error: "auth_required" }
- 403 if editing someone else's profile
- 413 if avatar too large
- 415 if avatar wrong format
```

The tools that help. Several frameworks have emerged in 2026 — you don't need them to start, but they're worth knowing:

● OPENSPEC

Lightweight, works with both Claude Code and Gemini CLI. Run `openspec init` and you get a `/openspec:propose` command for spec-first work.

● SPEC KIT (GITHUB)

GitHub's official toolkit. More structured, with explicit phase gates: `/speckit.specify`, `/speckit.plan`, `/speckit.tasks`.

● PLAIN MARKDOWN

Don't want extra tools? A `spec.md` file in your project works fine. Reference it with `@spec.md` when prompting.



START SIMPLE

You don't need OpenSpec, Spec Kit, or any framework to do spec-driven development. Start with a plain `spec.md` file. If you outgrow it, the tools are right there.

The full session in practice:

TERMINAL

```
# Step 1: draft the spec with help
> Help me write a spec for a user profile API.
  Ask me clarifying questions before writing anything.

# The AI asks: auth method? avatar size limits? rate limits?
# You answer. The AI writes spec.md.

# Step 2: review and refine
> Read spec.md. What edge cases are missing? What would
  break in production?

# The AI suggests: concurrent updates, deleted users, etc.
# You add the missing cases.

# Step 3: plan the build
> /plan Implement spec.md. Show me the task breakdown
  before writing any code.

# Step 4: build
> Looks good. Proceed task by task. Stop after each task
  so I can review.
```

WHY THIS WORKS SO WELL

A clear spec gives the AI a "north star." When it gets confused mid-build, it re-reads the spec instead of making something up. Bugs shrink. Rework drops. You catch misunderstandings in the spec phase — when they cost minutes — instead of after the code is written.

► TRY THIS

Pick a small feature you've been putting off. **Don't** ask the AI to build it directly. First, ask: "Help me write a spec for [feature]. Ask me clarifying questions first." Then refine the spec. Only after that, say: "Now implement the spec." Compare the result to your usual workflow.

THE SENIOR ENGINEER'S SECRET

The developers who get the most out of AI tools aren't the ones with the cleverest prompts. They're the ones who slow down at the start. They write the spec. They argue with the AI about edge cases. They make sure the plan is right **before** a single line of code gets written. Five minutes of spec discipline saves an hour of debugging.

Quick check

In a spec-driven workflow, when should you ask the AI to start writing code?

- A. Right away — the AI is fast, who needs a spec?
- B. After you've described the feature in one sentence.
- C. After the spec is clear, edge cases are listed, and the plan is reviewed. ✓

Yes. The whole point of SDD is to front-load the thinking. By the time code starts being written, the hard decisions are already made — the AI just executes a clear plan.

The capstone

You've made it through the guide. Now we test what stuck.

YOUR MISSION

Pick a real task from your team's backlog — something you'd normally spend half a day on. Complete it with AI help, using everything you've learned.

The workflow (memorize this — it's the rhythm of professional AI coding):

1. **Explore** — ask the AI to read relevant files and explain the current state.
2. **Plan** — use Plan Mode to get a written plan. Edit it.
3. **Code** — let the AI execute the plan. Review diffs as they come.
4. **Test** — generate and run tests.
5. **Commit** — have the AI write a clean commit message. You review and ship.

This rhythm — *Explore* → *Plan* → *Code* → *Test* → *Commit* — is what separates "I tried AI coding once" from "AI is part of how I work now."

Final check

A teammate says: "AI coding tools are unreliable — I asked Claude to refactor my code and it broke everything." What's the most likely reason?

A. Claude is a bad tool.

B. The robots are inherently unreliable for refactoring.

C. They probably skipped Plan Mode, didn't review diffs, and gave the AI no project context. ✓

Yes! The three habits — plan first, review diffs, give context — are what separate good outcomes from broken ones. Tools are only as reliable as the workflow around them.

ONE LAST THING

The robots will get better, faster, and stranger over the next year. The specific commands in this guide will shift. **The habits won't.** Give context. Plan before acting. Review every change. Keep humans in the loop. Master those, and the tools become interchangeable — you'll pick up the next one in an afternoon.

The glossary

Words you'll keep running into. Demystified.

CLAUDE.md / GEMINI.md

A file at the root of your project that the AI reads every session. Tells it your stack, conventions, and rules.

Agent Mode

The AI does multi-step work on its own, calling tools and editing files across your project to reach a goal.

MCP

Model Context Protocol. The plumbing that lets AI tools connect to GitHub, Jira, your database, etc.

Tab autocomplete

Cursor's predictive code completion. Suggests your next several lines based on context. Press Tab to accept.

Composer (Cmd+I)

Cursor's multi-file editor. Describe a change, see a coordinated diff across many files, approve per file.

BugBot

Cursor's GitHub-integrated bot that automatically reviews pull requests and leaves comments like a human reviewer.

agents.md

Cross-IDE fallback context file. Tools that support it (including Cursor) read this when their own rules folder isn't found.

SKILL.md

The instruction file at the heart of a skill. Has YAML frontmatter (name, description) and Markdown instructions. The description tells the AI when to activate the skill.

Spec-Driven Development

A workflow where you write a clear specification first, then ask the AI to implement against it. More reliable than free-form prompting.

Plan Mode

"Tell me what you'd do before doing it." The robot writes a plan, you approve or edit, then it executes.

Auto Mode

Claude Code's middle ground — safe actions run automatically, risky ones still pause for approval.

Subagent

A mini-AI that handles a specific job in its own context window and reports back to the main session.

Inline Edit (Cmd+K)

Cursor shortcut for focused edits. Select code, describe a change, get a surgical diff in place.

Mission Control

Cursor's grid view for managing up to 8 parallel agents. Run the same task with different models or different tasks at once.

.cursor/rules/

Cursor's project context folder. Each file is a scoped rule — applied always, or only to certain file types or paths.

Skill

A reusable bundle of instructions and assets — like a recipe the AI can follow for a specific kind of task.

.agents/skills/

The portable, cross-tool skills folder. Works in both Claude Code and Gemini CLI — write a skill once, both robots can use it.

OpenSpec / Spec Kit

Frameworks that turn spec-driven development into structured commands (`/openspec:propose`,

	<code>/speckit.plan</code>). Optional — plain markdown specs work too.
Checkpoint A snapshot of your project before a change. Gemini CLI's safety net for "undo whatever just happened."	Context window How much the AI can "see" at once. Bigger window = more files and conversation history fit at the same time.
Diff A view of changes: green for added lines, red for removed. The thing you should always read before approving.	Hooks Custom scripts that run at moments in the AI's workflow — before a file edit, after a tool call, etc. Power-user territory.
Routine Claude Code's scheduled task — recurring work that runs in the cloud (review PRs every morning, audit deps weekly).	

You made it. Now go build something.

A BEGINNER'S STUDY GUIDE — MAY 2026