Working With AI Agents: A Guide for Non-Technical People

This guide is for people who aren't developers but want to understand — and use — AI agents to build things. No coding background required. By the end, you'll know what agents are, how they think (and don't), what tools exist, and how to direct an agent through a real project.

Here's what each chapter covers:

1. **What AI Agents Are (and Aren't)** — What AI agents are, how to direct them, and what engines (models) power them. Agents are eager, literal workers. Your job is to brief clearly and set boundaries.
2. **Memory and Context** — Agents have no long-term memory. This chapter explains context windows, what gets lost between sessions, and the three layers of external memory you'll need to understand.
3. **Tools of the Trade** — There are three ways to work with AI: chat with it directly, use an all-in-one platform, or use an agent-assisted tool. This chapter maps the landscape and helps you pick.
4. **Setting Up** — Installing Claude Code, configuring VS Code, and running your first command. Step by step.
5. **Configuring Your Agent** — The instruction file (CLAUDE.md), rules, memory, custom agents, skills, hooks, and MCP servers. How to set up your agent's working environment before you start building.
6. **Your First Project** — A hands-on walkthrough: from blank folder to working website.
7. **Git and GitHub** — Version control: saving your work, going back to previous versions, and backing up your project online.
8. **Servers, Hosting, and Deployment** — What happens after you build something. Where it lives, how people access it.

Each chapter ends with a **Practical tips** section — short, concrete advice you can apply immediately. You don't need to read those tips upfront. They're there when you need them.

Glossary — Every technical term introduced in this guide, collected in one place. Not a chapter — an addendum you can flip to anytime.

The learning curve

Working with an AI agent is a new skill. Like any skill, you'll get better with practice. Here's what the progression typically looks like:

Week 1: You ask for small things and review everything carefully. It feels slow — like the agent is doing work you could do yourself, just differently.

Week 2-3: You start trusting the agent with bigger tasks. You learn what kinds of instructions produce good results. Sessions get more productive.

Month 2+: You think in terms of outcomes, not steps. Instead of telling the agent exactly what to do, you describe what you want and let it figure out the path. You catch problems faster and give better feedback.

The key is to stay engaged. The agent is a powerful tool, but it works best with a human who understands the project, reviews the work, and makes the decisions. That's you.

Chapter 1: What AI Agents Are (and Aren't)

> **TL;DR:** An AI agent is a piece of software that can take actions on its own — write code, create files, run commands — not just answer questions. But agents are narrow: they do what you tell them, eagerly and literally. Your job isn't to do the work. It's to direct it clearly.

If you've seen Disney's *Fantasia*, you remember the scene. Mickey enchants a broom to carry water. It works perfectly — until it doesn't know when to stop and floods the room. That broom is an AI agent.

Mickey Mouse as the Sorcerer's Apprentice, from Disney's Fantasia (1940)

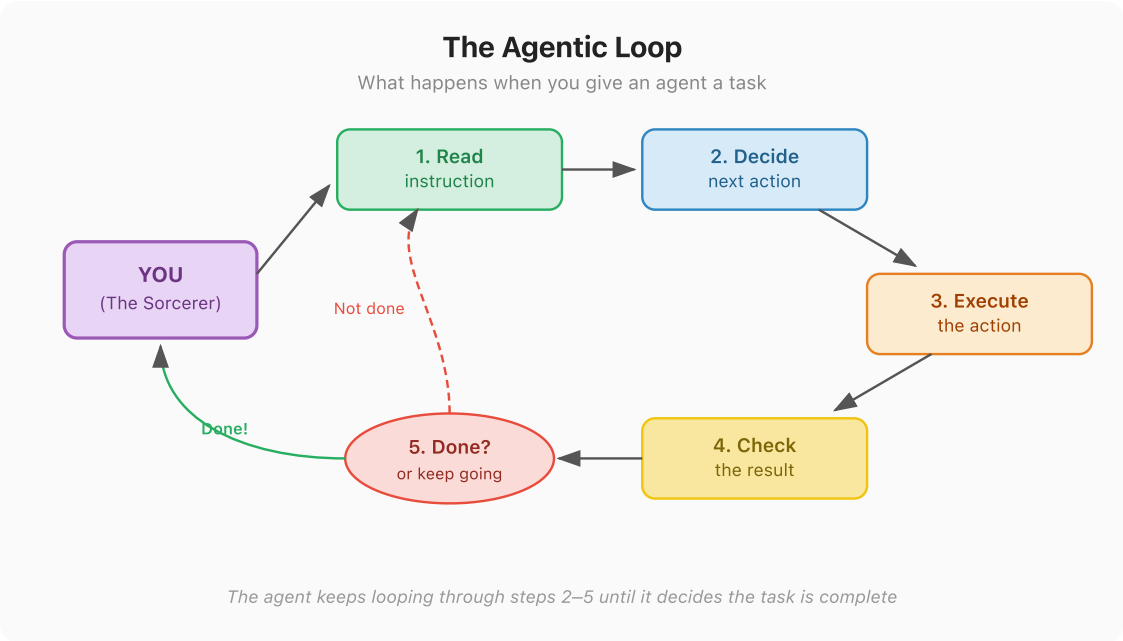
What agents are

Before we get into details, here's the shape of it:

1. **Agents act, not just talk.** Unlike a chatbot that answers questions, an agent creates files, writes code, runs commands, and takes a series of steps to complete a task.
2. **Agents are eager and literal.** They'll start immediately and keep going, making decisions along the way. If you're vague, they'll fill in the gaps with whatever seems reasonable — and you might not agree.
3. **Your job is direction, not execution.** You won't write code. You'll brief clearly, set boundaries, and course-correct.

The agentic loop. When you give an agent a task like "build me a website," it doesn't hand you a block of text and say "here, copy this." It actually creates the files, writes the code, installs software, and sets up the project. It takes a series of actions, one after another, to get from your request to a result.

The technical term for this is an **agentic loop** — the agent reads your instruction, decides what to do first, does it, checks the result, decides the next step, and keeps going until it thinks the job is done.



The Agentic Loop — how an agent processes a task

The brooms are eager. Agents will start working the moment you give them something to do, and they will keep going. This is a feature and a problem.

If you say "make me a website," the agent will make one — but it'll make a hundred decisions along the way that you might not agree with. What colors? What layout? What framework? It'll pick whatever seems reasonable and keep moving, because you didn't tell it to stop and ask.

How to direct well

You need to get good at directing the thing that does the work. That means three things: talking to it clearly, giving it tools for context, and setting guardrails.

Voice and briefing. You're not chatting — you're briefing. Plain, direct language. State what you want, then state what you don't want.

Vague (bad): > "Make me a nice website for my business"

Specific (better): > "Create a single-page website for a dog grooming business called 'Clean Paws.' Four sections: hero with the business name and a booking button, services list, pricing table, and contact form. Clean, minimal design. Light background, dark text. No animations."

A few rules of thumb:

- **Be concrete.** "A table with columns for name, email, and signup date" beats "some kind of user list."
- **Say what you don't want.** "Don't use any external libraries" or "Don't change the homepage" saves you from undoing work later.
- **Break it down, then sequence it.** Don't hand the agent a whole project at once. Split the work into steps and give them one at a time: "create the project structure and a homepage with placeholder content." Check it. Then give the next step. The agent won't do this for you — sequencing is your job.
- **When something's wrong, say what's wrong.** Not "this isn't right" but "the nav bar should be on the left side, not the top, and the links should be Home, Services, Pricing, Contact."

Tools that help

A spec or brief. A text file describing what you're building — who it's for, what it should do, what it should look like. Keep it as a file in the project that the agent reads automatically. More on this in Chapter 5.

Reference images. Most agents can look at images. A screenshot of a website you like, a rough sketch, a wireframe — these give the agent more to work with than words alone.

Checklists. A list of "the page should have X, Y, and Z" gives both you and the agent something to verify against.

Guardrails

Telling the agent what to do is only half the job. You also need to tell it what *not* to do. Without boundaries, agents will make "reasonable" choices that might be wrong — rewriting files you didn't want touched, picking technologies you don't want, or deleting things they think are unused.

- **"Don't modify any files in the /payments folder."**
- **"Use plain CSS, not Tailwind."**
- **"Ask me before deleting anything."**
- **"Keep the existing database structure — don't redesign it."**

One broom or many?

Sometimes you'll use a single agent. Other times, multiple agents on different parts of a project — one on the front end, another on back-end logic, another writing tests. Each is focused but unaware of what the others are doing. The tools handle this differently — Claude Code can spawn **sub-agents** to work on tasks in parallel, Antigravity has a dedicated Manager View for it, and the others are catching up. We'll cover the specifics in Chapter 3.

The engines underneath

Every agent runs on top of an AI model (sometimes called an **LLM**, which stands for Large Language Model). The model is the engine — the thing that actually reads your instructions, reasons about them, and generates code. The tool you use (Claude Code, Cursor, etc.) is just the interface. Understanding the engines matters because they have different strengths, different costs, and some tools let you switch between them.

Comparing the models

	Claude (Anthropic)	GPT / Codex (OpenAI)	Gemini (Google)
Main models	Sonnet 4.6 (fast), Opus 4.6 (powerful)	GPT-4o (fast), o1 (reasoning)	Gemini 2.0 Flash (fast), 2.5 Pro (powerful)
Subscription	$20/mo (Pro), $100–200/mo (Max)	$20/mo (Plus), $200/mo (Pro)	$20/mo (AI Pro), $250/mo (Ultra)
API cost (per 1M tokens)	Sonnet: $3 in / $15 out	GPT-4o: $2.50 in / $10 out	Flash: $0.10 in / $0.40 out
Strengths	Strong code generation, good at following detailed instructions, powers Claude Code	Large ecosystem, GPT-4o is price-competitive, o1 excels at complex reasoning	Flash is extremely cheap, good multimodal, integrates with Google Cloud
Weaknesses	Smaller ecosystem than OpenAI	o1 reasoning tokens inflate costs (hidden internal "thinking" tokens count toward your bill)	Lower code quality than Claude/GPT flagships, less mature tooling
Best for	Production coding agents, detailed project work	General coding tasks, complex reasoning (o1)	Cost-sensitive projects, quick prototyping

How you pay for the engine

There are two ways to access these models:

Subscription (through a product UI). You pay a monthly fee ($20/mo is the common entry point across all three) and get access through a chat interface or tool. Usage is included up to a limit. This is how most people start.

API (direct access). API stands for **Application Programming Interface** — a way for one piece of software to talk to another. You pay per use based on **tokens** sent and received (a token is roughly ¾ of a word). No monthly cap — you pay for what you use. More flexible, but requires setup: you need an **API key** (a password-like string that identifies your account to the service).

The big takeaway: Gemini Flash is dirt cheap, GPT-4o and Claude Sonnet are in the same ballpark, and the premium reasoning models (o1, Opus) cost 5–10x more. For beginners, start with a subscription. As you use agents more heavily, you may want to switch to API access for more headroom.

Cheap models vs. expensive models

Not every task needs the most powerful model. Here's how to think about it:

Model tier	Cost	Good for	Examples
Fast/cheap (Gemini Flash, Claude Haiku, GPT-4o mini)	~$0.10–0.50 per 1M tokens	Repetitive tasks, simple edits, reformatting, generating boilerplate code, summarizing text, quick Q&A	"Rename all variables from camelCase to snake_case," "Add comments to this file," "Convert this JSON to CSV"
Mid-range (Claude Sonnet, GPT-4o)	~$3–10 per 1M tokens	Most coding work, writing features, debugging, building pages, multi-step tasks that require understanding context	"Build a checkout page with form validation," "Find and fix the bug in the payment flow," "Refactor this component"
Premium (Claude Opus, o1, Gemini Pro)	~$15–60 per 1M tokens	Complex architecture decisions, multi-file refactoring, tricky debugging, tasks that need deep reasoning across a large codebase	"Redesign the authentication system to support SSO," "Why is the app crashing under load — investigate and fix," "Plan the migration from REST to GraphQL"

The practical rule: start with a mid-range model. Drop to cheap for grunt work. Escalate to premium when the agent is struggling or the task genuinely requires reasoning across a lot of context. If you're on a subscription, the platform handles this for you. On API, you choose — and your bill reflects it.

Practical tips

Say what, not how. You don't need to tell the agent *how* to do something — that's its job. Focus on *what* you want the result to be. "Create a centered section with a large heading and a paragraph of description text below it" beats "Create a div with flexbox, justify-content center, add an h1 with font-size 2rem." The agent knows CSS better than most humans. Let it pick the implementation.

Give context about why. When the agent understands *why* you want something, it makes better decisions about edge cases. "Add a character limit to the text input" is fine. "Add a character limit to the text input — this is for SMS messages, so it needs to be 160 characters max, with a counter so users know how many they have left" is better.

One task at a time. Don't dump five requests into one message. The agent handles one task well. Five tasks in one message means it might rush through some or miss details. "Add a contact form with name, email, and message fields" — then, after it's done — "Now fix the header spacing."

Ask the agent to explain before acting. When you're unsure about something, ask first: "Before making any changes — explain how you would add user authentication to this project. What technologies would you use, and what files would you need to create?" This gives you a plan to review before any code is written.

Read what the agent changed. The agent tells you what it did after each task. Read it. Sometimes it makes assumptions you didn't intend — changing a file you didn't ask about, using a library you don't want, or interpreting your request differently than you meant. Catching these early is easy. Catching them three tasks later is hard.

Watch what you approve. When the agent asks permission to run a command or make a change, read the request before approving. Pay special attention to file deletions, package installs, and configuration changes.

Chapter 2: Memory and Context

> **TL;DR:** AI agents have limited memory. Every time you start a new session, the agent starts fresh — it doesn't remember what you did before. To work well with agents, you need to think about how to give them context and how to get them to leave notes for next time.

In the movie *Memento*, the main character can't form new long-term memories. Every few minutes, he wakes up with no idea what just happened. To function, he builds a system of notes, photos, and tattoos — external memory for a brain that can't hold onto anything.

Still from Memento (2000)

Working with AI agents is the same problem. Here's the shape of it:

The memory problem at a glance

1. **Agents have a fixed-size short-term memory** (called a **context window**). It holds your conversation, files the agent has read, and instructions — but it has a hard limit.
2. **When it fills up, early information gets dropped.** The agent doesn't crash — it just quietly forgets things you said earlier in the session.
3. **New session = blank slate.** Close the session and open a new one, and the agent remembers nothing. Zero.
4. **You solve this with external memory** — files, project structure, and session briefings that give the agent context it can't hold on its own.

The context window

How it works

Everything the agent knows exists inside the **context window** — a fixed-size container that holds roughly 100,000–200,000 **tokens** (a token is about ¾ of a word). That sounds like a lot, but when the agent is reading files, tracking conversation, and working through code, it fills up.

When it's full, the earliest parts of the conversation drop out. Instructions you gave at the beginning might quietly disappear. And when you close a session and start a new one, it empties completely.

What this means in practice

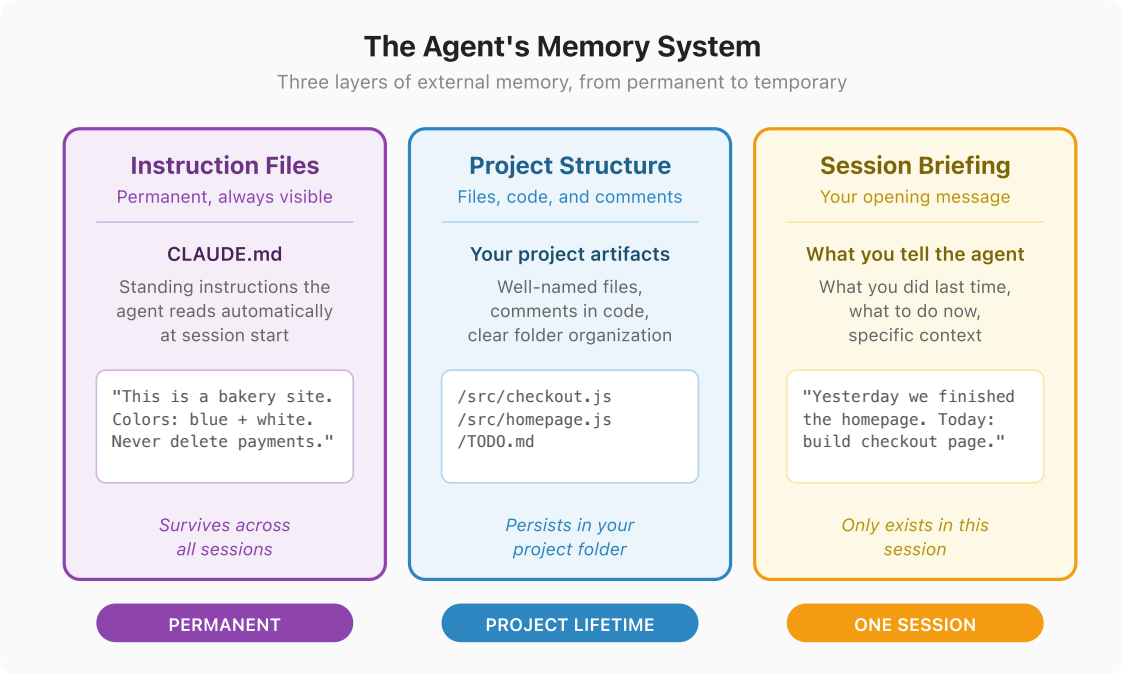
You work with an agent for an hour, make a dozen decisions, close the session. Next day: blank slate. The agent can scan your files and reconstruct *what* exists — but not *why* you chose it, or that you were mid-way through fixing a bug. That context is gone. This isn't a bug. It's how these systems work right now.

Building external memory

Since the agent can't remember, you build the memory system. There are three layers, and a strategy for getting the agent to help.

Three layers

From most permanent to most temporary:



The three layers of agent memory

Instruction files (permanent). A file like **CLAUDE.md** that sits in your project and contains standing instructions: what this project is, what technologies it uses, what the agent should never touch. The agent reads this automatically at the start of every session. You write it once, update it occasionally. (We'll build these files step by step in Chapter 5.)

The `.md` extension stands for **Markdown** — a simple way to write formatted text using plain characters. You don't need special software to write it — any text editor works. Here's what a markdown file looks like raw:

```
# Project: Clean Paws

A website for a dog grooming business.

## Rules
- Never delete the /assets folder
- Use **plain CSS**, not Tailwind
- Keep the design *minimal*
```

And here's how that same text renders when displayed by a tool like VS Code or GitHub:

> # Project: Clean Paws
>
> A website for a dog grooming business.
>
> ## Rules
>
> - Never delete the /assets folder
> - Use **plain CSS**, not Tailwind
> - Keep the design *minimal*

Same content, two views. Markdown files are everywhere in software projects because they work both ways — readable as plain text, and nicely formatted when rendered. CLAUDE.md is just a markdown file with a specific name that Claude Code looks for.

Project structure and comments (project lifetime). Your files, code, folder names, and the notes inside them. Clear file names, good comments, a well-organized folder structure. When the agent starts a fresh session and scans your project, these are the clues it uses to figure out what's going on.

Session briefing (one session). What you tell the agent at the start of each session: "Yesterday we finished the homepage. Today we're working on the checkout page. Here's what's left." This connects the permanent files and the project structure into a plan for right now.

Getting the agent to take its own notes

You can *tell* the agent to leave notes for its future self. Ask it to update the project documentation, write clear **commit messages** (a commit message is a short note that gets saved every time you save a version of your code — more on this in Chapter 7), or maintain a to-do list file that tracks what's done and what's left.

This turns the agent into an active participant in its own memory system. It still won't *remember* — but the next session's agent will find better clues.

The two timelines

Every time you work with an agent, keep two questions in mind:

1. **Right now:** Does the agent have enough context to do this task correctly?
2. **Next time:** When a fresh session starts, will it be able to pick up where this one left off?

If something "broke" between sessions, the agent didn't forget. It never knew. The knowledge needs to live outside the agent's head.

Practical tips

> **Start each session with context.** The agent starts fresh every session. Give it a quick briefing: "We're working on the Clean Paws website. Yesterday we finished the homepage and the contact form. Today I want to work on the About page. Check the CLAUDE.md for project details." Thirty seconds of context saves minutes of confusion.

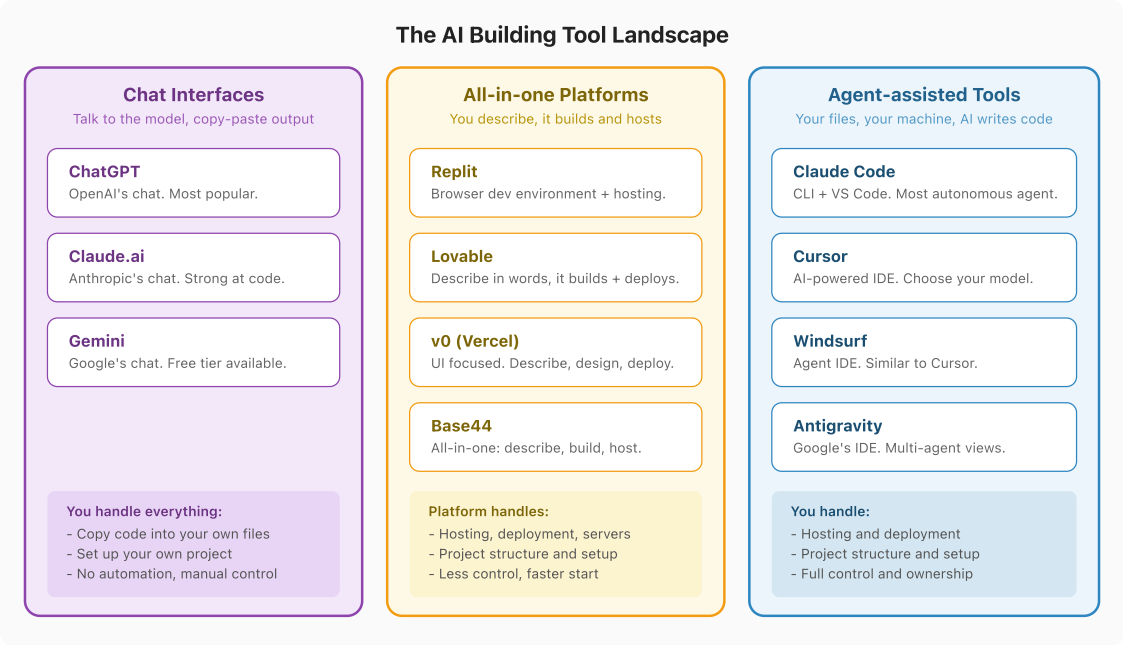
End each session with a save. Before closing: (1) commit your changes — “Commit everything with a message about what we did today,” (2) push to GitHub — “Push to GitHub,” (3) optionally, ask the agent to update notes — “Update the CLAUDE.md with what we accomplished and what’s left to do.” This takes a minute and means tomorrow’s session starts clean.

Know when to start fresh. If the agent seems confused, if the conversation is going in circles, or if you’ve been going back and forth on the same issue for too long — start a new session. A fresh agent with a clear instruction is often more productive than a confused agent with a long conversation history.

Chapter 3: Tools of the Trade

> **TL;DR:** There are three ways to work with AI: talk to it directly in a chat, use an all-in-one platform that handles everything, or use an agent-assisted tool where you control the files and decisions. This chapter maps the landscape — and explains why we recommend starting with one approach before graduating to another.

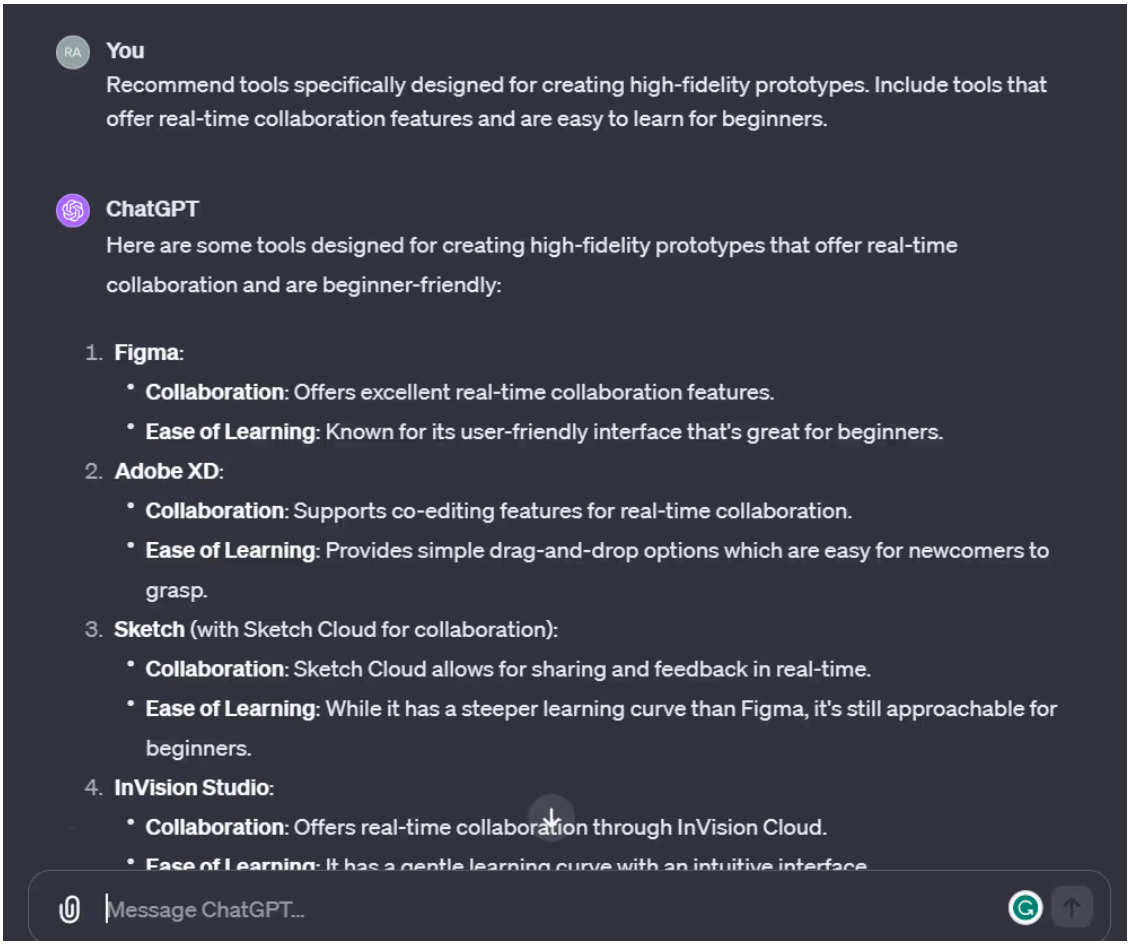
When people say "I built this with Claude" or "I used GPT to make my website," they're skipping a layer. You don't work with the AI model directly — or rather, you *can*, but it's just one of three approaches. Each puts a different amount of machinery between you and the model, and that machinery decides how much control you get.



The AI building tool landscape

Chat interfaces

This is where most people start. You open a website — ChatGPT, Claude.ai, Gemini — type what you want, and the model responds. That's it. No files, no project, no automation. Just a conversation.



A chat interface — you ask, it answers

Chat works well for small, self-contained tasks: “write me a function that validates email addresses,” “generate the HTML for a pricing table,” “explain what this error message means.” You copy the output, paste it wherever you need it, and move on.

And it’s not just code. People use chats for writing, research, brainstorming, analysis — anything where you want a quick answer or a draft. Need a marketing email? A summary of a long document? A meal plan? Chat handles that fine.

The limitation shows up when the task gets bigger. Say you’re writing a guide with ten chapters — each one a separate file, with cross-references between them, images, a glossary, and a consistent voice throughout. In a chat, you’d have to paste content in, get output back, paste it into the right file, remember what you changed in chapter 3 when you’re working on chapter 7. The model can’t see your files, can’t remember what you did earlier, and can’t work across multiple documents at once. You become the glue holding the project together.

That’s where the line between a chat and an agent-assisted tool becomes clear. It’s not about code vs. non-code — it’s about whether your task lives in a single conversation or spans multiple files and sessions. A chat is a conversation. An agent-assisted tool is a working environment.

Tool	Notes	Price
ChatGPT	OpenAI. Most widely used chat interface.	Free tier; Plus $20/mo
Claude.ai	Anthropic. Strong at code and longer conversations.	Free tier; Pro $20/mo
Gemini	Google. Integrated with Google services.	Free tier; Advanced $20/mo

All-in-one platforms

These tools run entirely in the cloud. You describe what you want — in plain language — and the platform builds it, hosts it, and deploys it for you. You might never touch a file directly. Just describe, preview, tweak, and publish.

The appeal is speed. You can go from idea to working prototype in minutes. The platform handles the technical decisions: what framework to use, how to structure the code, where to host it. You focus on *what* you want, not *how* to build it.

The tradeoff is control. You work within what the platform supports. If you need a specific database, a custom component, or a feature the platform doesn't offer — you're stuck. And your project lives on their servers. If they shut down, change pricing, or deprecate a feature, your project is affected. Moving your code elsewhere can range from inconvenient to impossible.

Platform	Specialty	Price	Notes
Replit	Full-stack apps	Free tier; paid from $25/mo	Complete dev environment in browser, built-in hosting, good for learning
Lovable	Web apps from descriptions	Free tier; Pro $25/mo, Business $50/mo	Describe what you want in plain language, it builds and deploys
v0 (Vercel)	UI and front-end design	Free tier; paid from $20/mo	Strong at generating interfaces and components, deploys through Vercel
Base44	All-in-one app building	Free tier; paid from $19/mo	Describe, build, host — lowest learning curve

> **If you want to start working on apps, try an all-in-one platform first.** Try Lovable, Replit, or one of the others. The value-for-money is excellent for straightforward projects. You'll have something working in minutes, not hours.
>
> But here's the real reason: you need to hit the wall. Every platform has limits — a component you can't customize, a database you can't connect, a feature that's just not possible within the platform's boundaries. When you hit that wall, you'll understand *why* agent-assisted tools exist and what they give you. That understanding makes the transition natural instead of forced.

Agent-assisted tools

This is where the rest of this guide lives.

Agent-assisted tools install on your computer. Your files live on your machine. The AI agent reads your files, takes instructions, writes code, runs commands — all locally. When you're done, the files are right there on your hard drive, in folders you control.

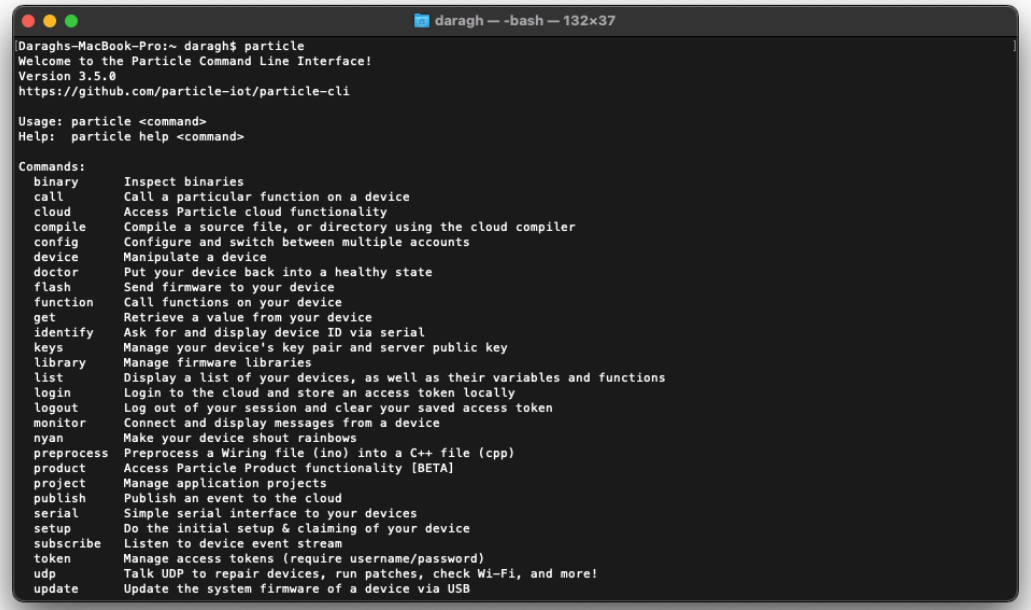
The platform layer here is thin. It connects you to the AI model and gets out of the way. You pick your own tools, your own framework, your own database. You decide where and how to deploy. The agent is powerful — it can make changes across multiple files, run your tests, browse the web for documentation — but *you* are the one making the decisions.

More setup, more to learn, full control.

Two ways to talk to your computer

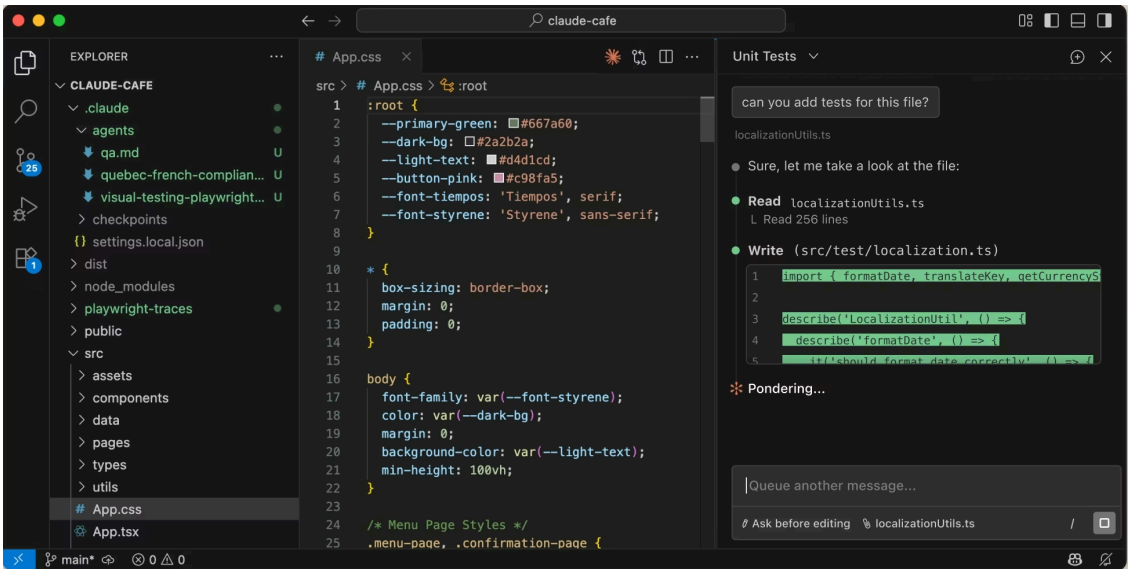
Before we look at specific tools, you need to understand two interfaces — because the tools below use one or both.

CLI (Command Line Interface) — A text-only way to talk to your computer. No buttons, no menus. You type a command, press Enter, get a result. If you've ever seen a black screen with green text in a movie — that's it (though in real life it's usually white text on a dark background). Think of it like the old DOS prompt: `C:\> dir` listed your files, `C:\> copy file.txt backup.txt` copied them. The Mac/Linux equivalent is called a **terminal**. Same idea, different name.



A terminal window — text in, text out

IDE (Integrated Development Environment) — A visual application where you write code. It looks like a souped-up text editor with extra powers: it can highlight syntax in color, catch errors as you type, manage your project files in a sidebar, and run your code with a button click. Think of it as the difference between writing in Notepad vs. writing in Word — same content, but one gives you a lot more help. Examples: VS Code, Cursor, Windsurf, Antigravity.



An IDE (VS Code) — files on the left, code in the middle, agent at the bottom

The key distinction: a CLI is text commands, an IDE is a visual application. Some tools work in both — Claude Code started as a pure CLI tool but now also runs as an extension inside VS Code (an IDE). You'll see this pattern: the boundary between CLI and IDE is blurring.

The tools

Tool	Interface	AI Model	Agent capabilities	Price
VS Code	IDE	Works with all AI agents via extensions — Claude Code, GitHub Copilot, and others.	Not an agent itself — it's the editor. Agents plug into it. The foundation that Cursor, Windsurf, and Antigravity are all built on.	Free
Claude Code	CLI + VS Code extension	Claude (Anthropic). Can also call other models via API for specific tasks.	Can spawn **sub-agents** — smaller agents that run tasks in parallel (e.g., one researches while another codes). Also supports tool use: reading files, running commands, browsing the web. The most autonomous of the group.	Included with Claude Pro ($20/mo) or Max ($100–200/mo); also API-based
Cursor	IDE (standalone app)	Claude, GPT, Gemini, and others — you switch models freely, even mid-conversation	Single agent, but you can choose the best model for each task. Strong at inline suggestions and code refactoring within the visual editor. Less autonomous than Claude Code — more of a copilot than an independent agent.	Free tier; Pro $20/mo
Windsurf	IDE (standalone app)	Multiple models	Single agent that can make changes across multiple files and understand broader project context. Similar to Cursor in capability.	Free tier; Pro $15/mo
Antigravity (Google)	IDE (standalone app)	Gemini 3, Claude, GPT (you choose)	**Manager View** lets you dispatch multiple agents to work on different tasks simultaneously — closest to true	Free during preview; expected ~$20/mo Pro

Tool	Interface	AI Model	Agent capabilities	Price
			orchestration in an IDE. Agents generate verifiable artifacts (plans, task lists) so you can review their logic. Still in public preview.	

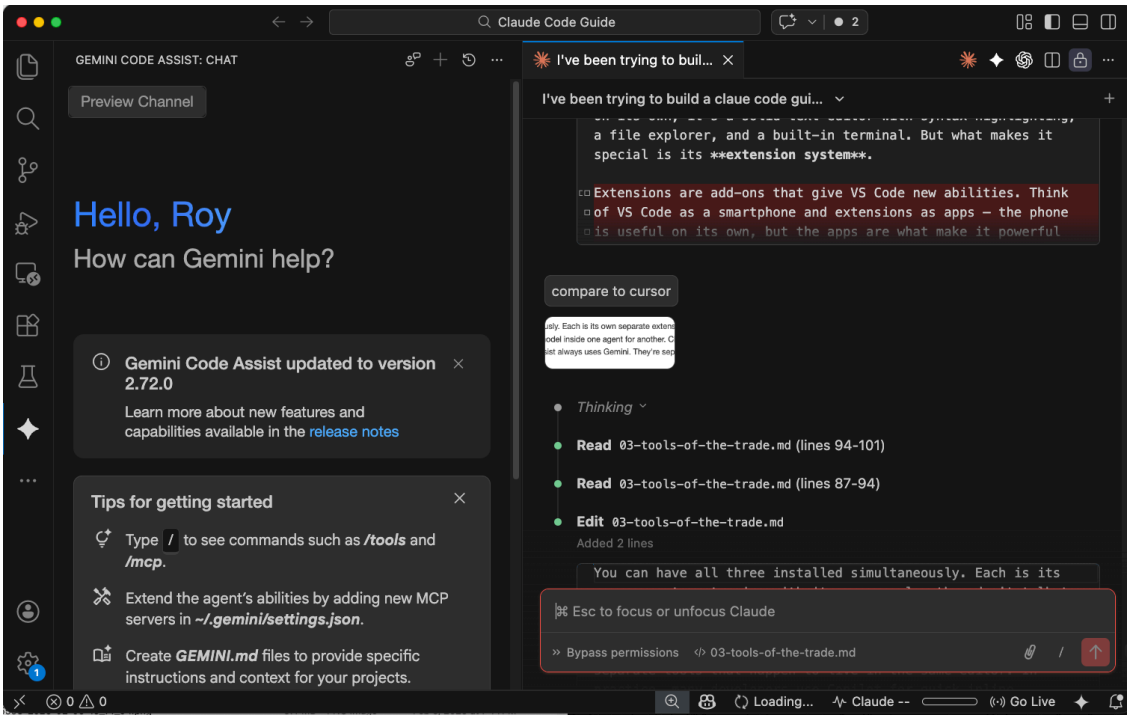
VS Code: the foundation

VS Code (Visual Studio Code) is a free IDE made by Microsoft. It's the most widely used code editor in the world. On its own, it's a solid text editor with syntax highlighting, a file explorer, and a built-in terminal. But what makes it special is its **extension system**.

Extensions are add-ons that give VS Code new abilities. Think of VS Code as a smartphone and extensions as apps — the phone is useful on its own, but the apps are what make it powerful for *your* specific needs. You install extensions with a few clicks inside VS Code — no terminal commands, no configuration files.

This matters because multiple AI agents can run inside VS Code as extensions, and you can install more than one at a time. Each brings a different model and different strengths:

- **Claude Code** — Anthropic's agent. The most autonomous of the group. It can read your entire project, make changes across multiple files, run commands, and spawn sub-agents to work in parallel. Uses Claude as its model.
- **GitHub Copilot** — Microsoft/OpenAI's agent. Built into VS Code natively. Strong at inline code suggestions — it predicts what you're about to type and offers completions as you go. Uses GPT and Codex models. Included free with VS Code, with a paid tier for more features.
- **Codex** — OpenAI's autonomous coding agent. Works like Claude Code — you give it a task and it works independently: reading files, writing code, running tests. Uses OpenAI's models. Currently in preview; requires an OpenAI API key.
- **Gemini Code Assist** — Google's agent. Connects to Gemini models. Good at explaining code and answering questions about your project. Free tier available.



Multiple AI agents running side by side in VS Code

You can have all of these installed simultaneously. Each is its own separate extension with its own panel — they don't talk to each other, and you can't swap the AI model inside one agent for another. Claude Code always uses Claude, Copilot always uses GPT, Codex always uses OpenAI's models, Gemini Code Assist always uses Gemini. They're separate tools that happen to live in the same editor. In practice, many developers use Copilot for quick inline suggestions while using Claude Code or Codex for larger tasks that require more autonomy.

This is the key difference from Cursor. Cursor is built on top of VS Code, but it takes the opposite approach: one agent, many models. Instead of installing separate extensions for each AI, Cursor lets you switch between Claude, GPT, Gemini, and others from a single dropdown — even mid-conversation. That's convenient if you want to try different models for different tasks without juggling multiple extensions. The tradeoff: Cursor costs $20/mo on top of whatever the AI model costs, its agent is less autonomous than Claude Code — more of a copilot than an independent agent — and it limits how much you can use each model. You get a monthly quota per model, and once you hit the cap, you either wait or switch to a cheaper model.

Windsurf and Antigravity also built on VS Code's open-source foundation, each baking their own AI features directly into the editor. That's why they all look similar (same sidebar, same file explorer, same layout) but each has its own approach to AI built in rather than added as extensions.

So when you learn VS Code, you're not learning one tool — you're learning the interface that nearly every agent-assisted tool uses.

Claude Code: terminal or VS Code?

Claude Code is the only tool here that works in two very different interfaces. Which one you use depends on your comfort level:

	Terminal (CLI)	VS Code extension
Looks like	Text on a dark screen. You type commands, agent responds in text.	A familiar visual editor with files in a sidebar, syntax highlighting, and the agent in a panel.
Strengths	Full power, nothing hidden. Faster for experienced users. Easier to script and automate.	Less intimidating for beginners. You can see your files and code visually while the agent works.
Best for	People comfortable with a terminal, or those willing to learn. Power users.	People who want a visual safety net. If the terminal feels alien, start here.

Same agent, same capabilities, same model underneath. The difference is purely how you interact with it.

What all agent-assisted tools share

- Your files stay on your computer
- You can use **Git** (a version control system — think of it as an infinite undo button that also lets multiple people work on the same project) to track changes (we'll cover Git and GitHub in Chapter 7)
- You choose where and how to deploy your project when it's ready
- You need to understand at least the basics of your project's structure

What this guide recommends

For the agent-assisted path, this guide uses **VS Code + the Claude Code extension**. Here's why:

VS Code is free and has a massive ecosystem of extensions. The Claude Code extension gives you the full power of Claude's agent — sub-agents, tool use, web browsing, autonomous execution — inside a visual editor where you can see your files and code. You get the best of both worlds: a proper IDE for when you want to look at things, and a powerful agent for when you want things done.

What about Cursor? Cursor is a strong alternative. Since it's built on VS Code, it looks and feels similar — but it ships with its own AI agent baked in rather than using an extension. Two advantages: it lets you switch between AI models mid-conversation (Claude today, GPT tomorrow, Gemini for a quick cheap task), and its inline code suggestions are polished. The tradeoff: it costs $20/mo on top of whatever you pay for the AI model, and its agent is less autonomous than Claude Code — it's more of a copilot that works alongside you than an independent agent you dispatch.

Both are solid choices. This guide uses Claude Code because it teaches you the most about how agents actually work — and because you can start with a tool you already have access to through a Claude subscription.

Comparison at a glance

	Chat interfaces	All-in-one platforms	Agent-assisted tools
Tools	ChatGPT, Claude.ai, Gemini	Replit, Lovable, v0, Base44	Claude Code, Cursor, Windsurf, Antigravity
How it works	You ask, it answers. You copy the output into your project.	You describe, it builds and hosts.	An agent reads your files, writes code, runs commands on your machine.
Where files live	Nowhere — output is in the chat. You manage files yourself.	On their servers — you access them through the browser.	On your computer, in folders you control.
Setup required	Sign up, start talking.	Sign up with email, start building.	Install the tool, configure subscription or API key.
Hosting	Not included — you figure that out.	Built in — click "publish" and it's live.	You set it up yourself — deploy to Vercel, Railway, AWS, etc.
Customization	Unlimited in theory — you get raw code to put anywhere. But no automation.	Constrained — you work within what the platform supports.	Total — pick any framework, database, service. Modify any line of code.
Learning curve	Lowest — if you can chat, you can use it.	Gentle — describe what you want in plain language.	Steeper — need to understand terminal, project structure, deployment basics.
Best for	Quick questions, small snippets, learning, one-off tasks.	Quick prototypes, MVPs, landing pages.	Projects you want to own, customize, and maintain long-term.
Limitations	No automation — you copy-paste everything manually. No file access, no deployment, no memory between sessions.	Platform lock-in. Limited customization. If they shut down or change pricing, your project is affected. Hard to move your code elsewhere.	Steeper learning curve. You handle hosting, deployment, and project structure yourself. Things can break and you need to troubleshoot.

Chapter 4: Setting Up

> **TL;DR:** You need three things: VS Code (free), the Claude Code extension (free), and a Claude subscription ($20/mo). This chapter walks through every step — downloading, installing, and running your first command.

Setting up takes about ten minutes. No terminal commands, no technical knowledge required. Just downloading, clicking, and signing in.

Step 1: Download VS Code

VS Code is the free editor we recommended in Chapter 3. It's where you'll do all your work.

1. Go to code.visualstudio.com
2. Click the big download button — it detects your operating system automatically
3. Open the downloaded file and follow the installer

Once it's installed, open VS Code. You'll see an editor with a sidebar on the left and a welcome tab. That's your workspace.

Step 2: Install the Claude Code extension

Extensions are add-ons that give VS Code new abilities. The Claude Code extension is what connects you to the AI agent.

1. In VS Code, click the **Extensions** icon in the left sidebar (it looks like four small squares)
2. In the search box at the top, type **Claude Code**
3. Find the one by **Anthropic** (look for the verified checkmark)
4. Click **Install**

That's it. Once installed, you'll see a small Claude icon in the sidebar. Click it to open the Claude Code panel.

Step 3: Get a Claude subscription

Claude Code needs a Claude account to work. If you don't have one yet:

1. Go to claude.ai and create an account
2. Subscribe to **Claude Pro** ($20/mo) or **Claude Max** ($100–200/mo) — Claude Code is included with both

The difference: Pro gives you a generous amount of usage for everyday work. Max gives you significantly more — useful if you're working on bigger projects or using the agent heavily throughout the day.

Step 4: Sign in

1. Click the Claude icon in the VS Code sidebar to open the Claude Code panel
2. It will prompt you to sign in — click the link and log in with your Claude account
3. Once connected, you'll see a text input where you can type instructions

Getting to know VS Code

When you first open VS Code, it can look overwhelming — buttons everywhere, panels, icons. Here's what you're actually looking at. There are only four areas that matter:

The four main areas of VS Code

Activity Bar (far left, the thin strip of icons). This is your navigation. Each icon opens a different panel in the sidebar. The main ones: - **File explorer** (top icon) — shows your project's files and folders - **Search** — find text across your entire project - **Extensions** (four squares icon) — where you installed Claude Code - **Claude Code icon** (at the bottom) — opens the agent panel

File Explorer (the sidebar next to the activity bar). This shows every file and folder in your project. Think of it like Finder on Mac or File Explorer on Windows — but built into the editor. Click any file to open it.

Editor Area (the center). This is where files open when you click them. You can have multiple files open at once — each one gets a tab at the top, just like browser tabs. This is where you read and edit code (or any text file).

Agent Panel (the right side, next to the editor). This is Claude Code. You type instructions here, and the agent responds. It can read your files, create new ones, and make changes — all from this panel. You see the agent's work on the right while your files are open on the left.

Terminal (the bottom panel, below the editor). A text command line built into VS Code. You won't need it much at first — Claude Code runs commands for you — but it's there when you need it. More on this in the next section.

The basic workflow: Click a file in the sidebar to see it in the editor. Type instructions in the agent panel on the right. The agent edits the file, and you see the changes appear in the editor on the left. That's the loop.

Step 5: Your first command

Let's make sure everything works. First, create a project folder:

1. Create a new folder on your computer (anywhere — Desktop is fine). Call it something like **my-first-project**
2. In VS Code, go to **File → Open Folder** and select that folder

Now, in the Claude Code panel, type:

```
Create a file called hello.txt with the text "Hello from Claude Code"
```

The agent will create the file. You'll see it appear in the VS Code file explorer on the left. Click it — the text should be there.

That's it. You have a working agent.

Working with the terminal

VS Code has a built-in terminal — that text-based command line we described in Chapter 3. You don't need to open a separate app. It lives right inside VS Code, in the same bottom panel area as Claude Code.

To open it: go to **Terminal → New Terminal** in the menu bar, or press `Ctrl+`` (that's the backtick key, usually above Tab).

You won't need the terminal much at the start. Claude Code handles most commands for you — when it needs to run something, it does it automatically. But as your projects grow, you'll occasionally want to run commands yourself: installing a package, starting a local server, or checking the status of your files in Git.

The terminal is always there when you need it. For now, just know where it is.

Recommended extensions

While you're in the Extensions panel, there are a few other extensions worth installing. None of these are required, but they make the experience better:

Extension	What it does	Why you'd want it
Markdown Preview Enhanced	Shows a live preview of markdown files side by side with the raw text	Useful for reading and editing CLAUDE.md and other documentation files. You'll be writing these in Chapter 5.
GitLens	Shows who changed what and when in your code	Helpful once you start using Git (Chapter 7) — makes version history visual instead of text-based.
Error Lens	Shows error messages directly in your code, right next to the line that caused them	Catches problems as you type instead of making you hunt through a separate panel.

To install any of these: search for the name in the Extensions panel, find the right one, click Install. Same process as Claude Code.

If something goes wrong

Problem	Fix
Can’t find the Claude Code extension	Make sure VS Code is up to date: go to **Help → Check for Updates**. Claude Code requires a recent version.
Can’t sign in	Make sure you have a Claude account at claude.ai. Claude Code requires at least a Pro subscription ($20/mo).
Agent responds but can’t create files	Check that you’ve opened a folder in VS Code (File → Open Folder), not just a single file. The agent needs a project folder to work in.
Extension installed but no icon in sidebar	Try restarting VS Code (close it completely and reopen). Sometimes extensions need a restart to appear.

Chapter 5: Configuring Your Agent

> **TL;DR:** Claude Code reads configuration files to know how to behave. The instruction file (CLAUDE.md) is where you start. Rules, custom agents, skills, hooks, memory, and MCP servers add more power as you need it. The best part: you don't have to create any of these files by hand — just ask Claude to set them up for you.

Chapter 4 got Claude Code running. This chapter is about making it work *well* — giving it the right context, the right tools, and the right guardrails before you hand it a real task.

Where everything lives: the .claude folder

All of Claude Code's configuration lives in a folder called `.claude`. The dot at the beginning means it's a **hidden folder** — it won't show up in Finder or File Explorer by default, but VS Code shows it in your file explorer.

There are two levels:

Level	Location	Who it affects	What goes here
Project	`my-project/.claude/`	Just this project	Project-specific rules, agents, skills, settings
Global	`~/.claude/` (your home folder)	Every project on your machine	Personal preferences, global rules, your memory file

The ~ symbol means your home folder — `/Users/yourname` on Mac, `C:\Users\yourname` on Windows. You don't need to create these folders yourself. Claude Code creates them when needed.

The key idea: Project-level configuration is shared with your team (it lives in the project). Global configuration is personal to you.

You don't have to create files by hand

This is important: **you can ask Claude to set up any of this configuration for you.** You don't need to know the exact folder structure or file format. Just tell the agent what you want:

- "Create a CLAUDE.md for this project"
- "Add a rule that all code should use TypeScript"
- "Set up a custom agent for code review"
- "Create a skill for deploying to production"

Claude knows where these files go and how to format them. As you get more comfortable, you might want to edit them directly — but you never have to.

The configuration layers

There are seven layers, from most essential to most advanced. You only need the first one to start:

1. **The instruction file (CLAUDE.md)** — Standing orders for every session
2. **Rules** — Organized instructions by topic
3. **Memory** — Notes the agent keeps across sessions
4. **Custom agents** — Specialized agents for specific tasks
5. **Skills** — Reusable commands you can invoke
6. **Hooks** — Automated actions at specific moments
7. **MCP servers** — Connections to external tools

1. The instruction file (CLAUDE.md)

This is the most important configuration you'll make. CLAUDE.md is a text file that Claude Code reads automatically at the start of every session — before you say anything. It's your agent's standing orders.

What goes in it

Think of CLAUDE.md as a brief for a new team member who joins the project every morning with no memory of yesterday. What would you tell them?

- What this project is and who it's for
- What technologies it uses
- What commands to run (build, test, deploy)
- What files or folders to never touch
- Coding style and naming conventions
- Known issues or quirks

A real example

```
# Project: Clean Paws Website

A single-page website for a dog grooming business.

## Tech stack
- HTML, CSS, vanilla JavaScript
- No frameworks, no build tools
- Hosted on Vercel

## Commands
- Preview locally: open index.html in browser
- Deploy: push to main branch (Vercel auto-deploys)

## Rules
- Never delete or modify /assets/images — these are client-provided
- Use plain CSS, not Tailwind or Bootstrap
- All text content is in English
- Keep the design minimal: white background, dark text, blue accents (#2563EB)

## File structure
- index.html — main page
- styles.css — all styles
- script.js — form validation and booking logic
- /assets/images — client photos (do not modify)
```

How to create it

The fastest way — just ask Claude:

```
Create a CLAUDE.md file for this project
```

Claude will scan your project and generate one based on what it finds — detected languages, file structure, build tools. Edit it from there.

You can also run the shortcut command `/init` in the Claude Code panel, which does the same thing.

Where it lives

Location	Scope
Your project root (`/my-project/CLAUDE.md`)	Read for this project only
Inside the .claude folder (`/my-project/.claude/CLAUDE.md`)	Same — project only
Your home folder (`~/.claude/CLAUDE.md`)	Read for every project on your machine

Most people use a project-level CLAUDE.md. If you have rules that apply everywhere ("never commit .env files"), put those in the global one.

There's also a **local** variant — `.claude/CLAUDE.local.md` — for personal preferences you don't want to share with your team. This file is ignored by Git, so it stays on your machine.

2. Rules

As your CLAUDE.md grows, you might want to organize it. Instead of one giant file, you can split instructions into **rules** — separate files organized by topic.

```
my-project/.claude/rules/
├── code-style.md      — naming conventions, formatting
├── testing.md         — how to write and run tests
├── security.md        — input validation, secret handling
└── deployment.md      — how to deploy, what to check
```

Each rule is a plain markdown file. Claude reads all of them automatically, just like CLAUDE.md. The advantage is organization — when you want to update your testing rules, you edit one focused file instead of hunting through a giant instruction document.

Rules can also live globally (`~/.claude/rules/`) for instructions you want across all projects.

To create rules: Just ask Claude — "Create a rule for code style in this project" — and it will create the file in the right place.

3. Memory

Remember Chapter 2 — the memory problem? Memory is one of the solutions.

Claude Code has an **auto memory** system. As you work, it can save notes to a file called `MEMORY.md` that persists across sessions. Next time you start a new session, Claude reads this file and remembers what it learned.

What goes in memory: - Patterns it discovered about your project ("this codebase uses Tailwind for styling") - Debugging insights ("the auth system requires a restart after config changes") - Your preferences ("user prefers concise commit messages")

Memory lives at `~/.claude/projects/<your-project>/memory/MEMORY.md`. You can also ask Claude to remember things explicitly:

```
Remember that I always want you to run tests before committing
```

And it will save that to its memory file. You can also tell it to forget things:

```
Stop remembering the thing about Tailwind — we switched to plain CSS
```

4. Custom agents

A **custom agent** is a specialized version of Claude that you define for a specific type of task. Think of it as creating a team member with a specific role: one agent for code review, another for writing tests, another for planning.

Each agent is a markdown file with a name, description, and instructions:

```
---
name: code-reviewer
description: Reviews code for quality and security issues
---

You are a code review specialist. When reviewing code:
1. Check for security issues first
2. Check for logic errors
3. Verify naming conventions match the project style
4. Summarize findings in a clear table
```

Custom agents live in `.claude/agents/` (project) or `~/.claude/agents/` (global). When Claude Code needs to handle a task that matches an agent's description, it can dispatch that agent — or you can invoke one directly.

To create one: Ask Claude — "Create a custom agent for code review" — and it will set it up.

Why use custom agents?

The main agent can do everything. But custom agents are useful when: - You want consistent behavior for a specific task (always review code the same way) - You want to run tasks in parallel (one agent researches while another writes code) - You want a restricted agent that can only read files, not modify them

5. Skills

A **skill** is a reusable command that teaches the agent how to do a specific type of task. Think of it as a recipe card you can invoke by name.

Skills are markdown files that live in `.claude/skills/<skill-name>/SKILL.md`:

```
---
name: deploy
description: Deploy the project to production
---

# Deployment Steps

1. Run all tests — stop if any fail
2. Build the project
3. Push to the main branch
4. Verify the deployment URL responds
5. Report the result
```

Once created, you invoke a skill by typing its name with a slash: `/deploy`. The agent reads the instructions and follows them.

Built-in skills come with Claude Code. You can also install community skills or create your own. Creating your own becomes useful when you find yourself giving the agent the same instructions repeatedly.

6. Hooks

Hooks are automated actions that run at specific moments in the agent's workflow — like tripwires. You set them up once, and they fire every time that moment occurs.

Hook event	When it fires	Example use
`PreToolUse`	Before the agent runs a command	Block dangerous commands
`PostToolUse`	After the agent edits a file	Auto-format the code
`Stop`	When the agent finishes a task	Run tests automatically
`SessionStart`	When a new session begins	Check prerequisites

Say you want the agent to automatically format code after every edit. You'd create a hook that fires after each file change (`PostToolUse`) and runs your code formatter.

You can set up hooks by typing `/hooks` inside Claude Code — it opens an interactive menu.

Who needs hooks? Not beginners. Hooks are a power feature for when you're running agents more autonomously and want automated guardrails. Come back to this when you need it.

7. MCP servers

MCP stands for **Model Context Protocol**. An MCP server is a bridge that connects Claude Code to an external tool or service — like GitHub, Slack, a database, or a project management tool.

Without MCP, the agent can only work with files on your computer. With MCP servers, it can:

- Read and create GitHub issues and pull requests
- Query your database directly
- Pull designs from Figma
- Access your project management tools (like Linear or Jira)

To add one, you can ask Claude:

```
Connect to GitHub using MCP
```

Or use the command directly:

```
claude mcp add --transport http github https://mcp.github.com
```

Who needs MCP servers? Anyone working on a project that connects to external services. If your project uses GitHub for code or a database for data, MCP servers let the agent interact with those tools directly.

The configuration stack at a glance

Layer	What it does	When to set it up	How to create it
CLAUDE.md	Standing instructions for every session	Day one — before your first real task	Ask Claude or run `/init`
Rules	Organized instructions by topic	When your CLAUDE.md gets too long	Ask Claude to create a rule
Memory	Notes that persist across sessions	Automatic — Claude learns as you work	Ask Claude to remember something
Custom agents	Specialized agents for specific tasks	When you want consistent behavior for a task type	Ask Claude to create an agent
Skills	Reusable commands you invoke by name	When you repeat the same instructions	Ask Claude to create a skill
Hooks	Automated checks and actions	When you want automated guardrails	Type `/hooks` in Claude Code
MCP servers	Connections to external tools	When your project uses external services	Ask Claude to connect a service

Global vs. project: what goes where

Put it in project (`.claude/`) when…	Put it in global (`~/.claude/`) when…
It's specific to this project	It applies to all your projects
Your team should see it	It's a personal preference
It references project files or structure	It's about how *you* work, not the project

Examples: - "Use Tailwind for styling" → project rule (specific to this project) - "Always explain your changes before making them" → global rule (your preference everywhere) - A deploy skill for this project → project skill - A code review agent you use everywhere → global agent

---

Practical tips

> **Don't skip the CLAUDE.md.** Starting without a CLAUDE.md is like hiring someone and not telling them what the project is. The agent will work — but it'll make assumptions about your tech stack, style, and preferences that might not match yours. Five minutes writing a CLAUDE.md saves hours of corrections.

Chapter 6: Your First Project

> **TL;DR:** This chapter walks you through building a real project from a blank folder to a working website. You'll create a business landing page step by step — giving instructions to your agent, reviewing its work, and learning the rhythm of working with AI.

Everything so far has been setup and theory. This chapter is where you actually build something.

We're going to create a landing page for a fictional dog grooming business called **Clean Paws**. It's simple enough to finish in one session, but real enough to teach you the workflow you'll use on every project going forward.

Start with a folder

1. Create a new folder on your computer. Call it `clean-paws`.
2. Open it in VS Code: **File → Open Folder** → select the folder.

You should see an empty file explorer on the left. That's your blank canvas.

Create the instruction file

Before building anything, give your agent context. In the Claude Code panel, type:

```
Create a CLAUDE.md for this project. It's a single-page landing page for
a dog grooming business called Clean Paws. Tech stack: plain HTML, CSS,
and JavaScript — no frameworks. The design should be clean and minimal:
white background, dark text, blue accents.
```

The agent will create a CLAUDE.md file. Click it in the file explorer to see what it wrote. Read through it — does it capture what you want? If something is off, tell the agent:

```
Update the CLAUDE.md — add a rule that all images go in an /assets folder
```

This is your project's foundation. Every future session starts by reading this file.

Build the page structure

Now, build the skeleton:

```
Create the landing page with these sections:
- A header with the business name and a navigation bar
- A hero section with a large heading, a tagline, and a "Book Now" button
- A services section showing three services (Bath & Brush, Full Groom, Puppy's First)
- A contact section with the address, phone number, and hours
- A footer with a copyright notice
```

The agent will create your HTML file. Once it's done, let's see it in a browser.

Preview your work

You need to see what the page actually looks like. Ask the agent:

```
How do I preview this page in my browser?
```

For a simple HTML page, the answer is usually: right-click `index.html` in the file explorer and select **"Open with Live Server"** (if you have that extension) or just open the file directly in your browser.

You should see an unstyled page — just text, no colors, no layout. That's expected. The structure is there; the design comes next.

Add styling

```
Style the page. Use the blue accent color from the CLAUDE.md. Make the
header stick to the top. The hero section should be tall with centered
text. The services section should show the three services in a row on
desktop and stacked on mobile. Keep it clean and professional.
```

Refresh your browser. Now you should see something that looks like a real website. If something doesn't look right, be specific:

```
The services section looks cramped — add more spacing between the three
cards. And make the "Book Now" button bigger.
```

This is the core loop: **instruct → review → adjust**. You'll do this dozens of times per session.

Add interactivity

A static page is fine, but let's add some life:

```
When someone clicks the "Book Now" button, scroll smoothly down to the
contact section. Also add a simple mobile menu — on small screens, the
navigation should collapse into a hamburger menu that opens on tap.
```

Check it in your browser. Resize the window to see if the mobile menu works. Click the button to test the scroll.

What just happened

Let's pause and look at what you have. Open your file explorer:

```
clean-paws/
├── CLAUDE.md        — your agent's instructions
├── index.html       — the page structure
├── styles.css       — how it looks
└── script.js        — the interactive parts
```

Four files. A complete landing page. You didn't write any code — you described what you wanted, and the agent built it. But the files are right there on your computer, in a folder you control. You can open them, read them, edit them, or move them anywhere.

Save your work

Now is the time to set up Git (Chapter 7) and save everything. But for now, a quick save:

```
Initialize a Git repository and commit everything with the message
"Initial landing page for Clean Paws"
```

Your project is now tracked. You can always come back to this point.

Make it yours

Here's where it gets fun. The landing page works, but it's generic. Start customizing:

```
Change the hero heading to "Your Dog Deserves the Best" and the tagline
to "Professional grooming in downtown Portland since 2019"

Add a testimonials section between services and contact. Show three
customer quotes with names and star ratings.

The color scheme feels too corporate. Make it warmer — try a soft cream
background instead of pure white, and use a teal accent instead of blue.
```

Each instruction changes the page. Each change is immediately visible in your browser. This is how real projects evolve — not in one giant leap, but in dozens of small, reviewable steps.

Try breaking something (on purpose)

This is a safe space to learn. Try asking for something ambitious:

```
Add a photo gallery with a lightbox effect — clicking a photo opens it
full-screen with previous/next navigation
```

It might work perfectly. It might not. If something breaks:

```
The gallery isn't working — the images don't open when I click them.
Check the console for errors and fix it.
```

This is normal. Building software is iterative. Things break, you fix them, you move on. The agent handles the debugging — you just need to describe what's wrong.

Deploy it

When you're happy with the page, put it on the internet. This is a preview of Chapter 8:

```
Deploy this to Vercel so I can share the link
```

The agent will walk you through connecting Vercel (free account, one-time setup). When it's done, you'll have a live URL — a real website that anyone can visit.

The workflow you just learned

This is the workflow for every project, not just this one:

1. **Create a folder** and open it in VS Code
2. **Set up CLAUDE.md** — give the agent context about what you're building
3. **Build the structure first** — get the skeleton right before styling
4. **Style and refine** — iterate visually, checking the browser after each change
5. **Add features** — one at a time, testing each one
6. **Commit often** — save your progress at meaningful milestones
7. **Deploy** — put it online when it's ready

The project is different every time. The workflow stays the same.

What to build next

Now that you've done it once, try something on your own:

- **A personal portfolio** — a page about you, your work, and how to reach you
- **A restaurant menu** — sections for appetizers, mains, desserts, with prices
- **An event page** — date, location, schedule, and an RSVP button
- **A simple blog** — a few posts with a clean reading layout

---

Practical tips

Start with the structure, then fill in details. Don't try to build a complete page in one shot. Build the skeleton first: "Create the basic page structure with a header, main content area, and footer. No styling yet." Then add content, then style it. Each step is reviewable. If the structure is wrong, you fix it before investing time in styling.

Use a reference. If you know what you want it to look like, say so: "Make the pricing section look similar to Stripe's pricing page — three columns, each with a plan name, price, feature list, and a button." The agent has seen thousands of websites. A reference gives it a target instead of guessing.

Review in the browser, not just in code. After the agent makes changes, open your project in a browser and look at it. The code might be perfect, but the visual result might not be what you expected. Fonts, spacing, colors, layout — these are easier to judge visually than by reading code.

Let the agent manage files, but review what it changed. As projects grow, the agent works across many files. You don't need to specify which files to create or modify — the agent knows. But after a task, ask: "What files did you modify?" Spot-check the important ones. If a file is doing too many things, ask the agent to split it up.

Undo mistakes quickly. If the agent made a change you didn't want, ask it to undo: "Undo the last change you made to index.html." Or, if you committed before the change: "Revert to the last commit."

Be specific about what's wrong. "This doesn't look right" gives the agent nothing to work with. "The header text is too small — make it bigger. And the button is overlapping the paragraph below it — add some spacing" tells it exactly what to fix.

Ask the agent to investigate. When something breaks and you don't know why: "Something broke — the page shows a blank white screen. Check the console for errors and figure out what went wrong." The agent can read error messages, trace the problem, and fix it. But it needs to know something is broken.

Describe the problem, not the solution. When you're stuck: "I want to add a way for visitors to book appointments on the website. I'm not sure how to approach this. What are my options?" The agent will suggest approaches, explain the tradeoffs, and help you pick one. You don't have to know the technical solution.

Pick one. Create a folder. Write a CLAUDE.md. Start building. You know the workflow now.

Chapter 7: Git and GitHub

> **TL;DR:** Git is an infinite undo button for your project. GitHub is where your project lives online. Together, they let you save your work, go back to any previous version, and collaborate with others. Your agent handles most of the mechanics — but you need to understand what's happening.

Why version control matters

Imagine writing a 20-page document with no undo button. Every change is permanent. Delete a paragraph by accident? Gone. Rewrite a section and realize the original was better? Too late.

That's what building software is like without **version control** — a system that tracks every change you make, lets you undo any of them, and keeps a complete history of your project.

Git is the version control system used by almost every software project in the world. It's not a product or a service — it's a tool that runs on your computer.

GitHub is a website where you store your Git projects online. Think of Git as the engine and GitHub as the garage — Git tracks changes locally, GitHub stores them in the cloud where they're safe and shareable.

The core concepts

There are only four concepts you need to understand. Everything else builds on these.

1. Repository (repo)

A **repository** is a project tracked by Git. When you tell Git to start tracking a folder, that folder becomes a repository. It looks exactly the same — same files, same folders — but now Git is watching every change.

Your agent can create a repository for you:

```
Initialize a Git repository for this project
```

2. Commits

A **commit** is a saved snapshot of your project at a specific moment. Think of it as a save point in a video game. You can always go back to any commit.

Each commit has: - A **message** describing what changed ("Added contact form to homepage") - A **timestamp** (when it was saved) - A **unique ID** (a long string of letters and numbers)

You don't make commits constantly — you make them at meaningful moments. Finished the homepage? Commit. Fixed a bug? Commit. About to try something risky? Commit first, so you can go back if it breaks.

Your agent creates commits too. When it finishes a task, you can ask:

```
Commit these changes with a message describing what you did
```

3. Branches

A **branch** is a parallel copy of your project where you can make changes without affecting the original. The main version of your project is usually called `main`.

Say you want to add a new feature but you're not sure it'll work. You create a branch, make your changes there, and if it works — you merge it back into `main`. If it doesn't — you delete the branch and nothing was harmed.

```
Create a new branch called "add-contact-form"
```

Branches are how teams work together without stepping on each other's changes. One person works on the login page in their branch. Another works on the homepage in theirs. When both are done, they merge their branches into `main`.

4. Push and pull

Push sends your local commits to GitHub (your computer → the cloud). **Pull** downloads the latest changes from GitHub (the cloud → your computer).

When you commit changes, they're saved locally on your machine. Pushing uploads them to GitHub where they're backed up and visible to others.

```
Push my changes to GitHub
```

Setting up Git and GitHub

Git

Git comes pre-installed on Mac. On Windows, your agent can help you install it, or download it from git-scm.com.

To check if you have it, ask Claude:

```
Check if Git is installed
```

GitHub

1. Create a free account at github.com
2. Ask Claude to connect your project to GitHub:

```
Create a GitHub repository for this project and push the code
```

Claude will walk you through connecting your account (usually by opening a browser window for authentication) and then push your code.

The everyday workflow

Once set up, the daily workflow is simple:

1. **Start working** — make changes to your project (or ask Claude to)
2. **Commit** — save a snapshot when you reach a good stopping point
3. **Push** — upload to GitHub so it's backed up

That's it. Your agent handles the Git commands. You just tell it when to save and what to call the save point.

What your agent does with Git

Claude Code uses Git automatically for several things:

- **Tracking what changed** — before making edits, it can show you what files were modified
- **Creating commits** — you tell it to commit, it writes a descriptive message and saves
- **Working with branches** — it can create, switch, and merge branches
- **Pushing to GitHub** — sends your changes to the cloud
- **Reverting changes** — if something goes wrong, it can undo recent changes

You don't need to memorize Git commands. The agent handles the mechanics. But understanding *what's happening* — that you're saving snapshots, that branches are parallel copies, that pushing backs up your work — makes you a better collaborator with the agent.

Pull requests: proposing changes

A **pull request** (PR) is how you propose changes on GitHub. Instead of pushing directly to `main`, you push to a branch and then open a pull request that says "here's what I changed, please review it."

This is especially useful when: - Working with a team (someone reviews your changes before they go live) - Working with an agent (you review the agent's changes before merging them)

```
Create a pull request for the contact form feature
```

Claude will push your branch to GitHub and open a PR with a description of what changed.

Common situations

Situation	What to do
"I want to save my progress"	Ask Claude to commit your changes
"I want to try something risky"	Ask Claude to create a branch first
"Something broke and I want to go back"	Ask Claude to revert to the last commit
"I want to back up my work"	Ask Claude to push to GitHub
"I want someone to review my changes"	Ask Claude to create a pull request
"I accidentally deleted a file"	Ask Claude to restore it from Git history

What not to worry about

Git has hundreds of commands and options. You don't need most of them. Here's what you can safely ignore for now:

- **Rebasing** — an advanced way to organize commit history. Merging works fine.
- **Cherry-picking** — copying specific commits between branches. You won't need this.
- **Stashing** — temporarily hiding changes. Just commit instead.
- **Submodules** — projects inside projects. Avoid unless absolutely necessary.

If you understand commits, branches, push, and pull — you know enough to work effectively with your agent and keep your project safe.

Practical tips

> **Commit after every completed task.** Every time you reach a working state — commit. Think of commits as save points in a game. If the agent breaks something in the next task, you can go back. A good rhythm: one commit per completed task. Finished the contact form? Commit. Fixed the header? Commit. About to redesign the homepage? Commit first.

Chapter 8: Servers, Hosting, and Deployment

> **TL;DR:** Building something is half the job. The other half is putting it somewhere people can access it. This chapter explains what servers are, where your project can live, and how to get it from your computer to the internet.

What is a server?

Your computer runs your project while you're working on it. But when you close your laptop, nobody else can see it. A **server** is a computer that's always on, always connected to the internet, always ready to serve your project to anyone who visits.

When you type a website address into your browser, your browser sends a request to a server somewhere in the world. That server sends back the website — the HTML, the images, the code. Every website you've ever visited is running on a server.

Two kinds of projects

How you deploy depends on what you built:

Static sites — Files that don't change. HTML, CSS, JavaScript, images. A blog, a landing page, a portfolio. The server just hands files to the browser. Simple, cheap, fast.

Dynamic apps — Code that runs on the server. A login system, a database, a payment form. The server does work — checking passwords, saving data, processing orders — before sending a response. More complex, more expensive, more powerful.

Most beginner projects are static or lightly dynamic. If your project is a website with no user accounts and no database — it's static.

Where to host

For static sites

These platforms host static files for free or near-free. You push your code, they handle the rest.

Platform	What it does	Price	How to deploy
Vercel	Hosts static sites and simple apps. Automatic deploys when you push to GitHub.	Free tier (generous)	Connect your GitHub repo, push code, it deploys automatically
Netlify	Similar to Vercel. Hosts static sites with automatic deploys.	Free tier	Connect GitHub repo, it deploys on push
GitHub Pages	Hosts static sites directly from a GitHub repository.	Free	Enable in your repo settings

Recommended for beginners: Vercel. It's free, fast, and integrates with GitHub. Push your code, it builds and deploys automatically. You get a URL like `your-project.vercel.app` that anyone can visit.

Self-hosting: your own mini server

You don't have to use a cloud platform. A small computer like a **Raspberry Pi** or a **Mac Mini** sitting in your home can serve a website to the internet. It's always on, always connected, and you own it completely.

This is a great option for small personal sites, home dashboards, or projects where you want total control without monthly fees. The tradeoff: you're responsible for keeping it running, connected, and secure. If your internet goes down, your site goes down.

Your agent can help you set this up — configuring a web server, opening the right ports, and getting a domain pointed to your home IP address. It's more hands-on than cloud hosting, but for a small site it works surprisingly well.

For dynamic apps

These platforms run your server-side code — Node.js, Python, databases, background jobs.

Platform	What it does	Price	Notes
Railway	Runs apps with databases. Simple setup.	Free trial; from $5/mo	Good balance of simplicity and power
Render	Hosts web services, databases, cron jobs.	Free tier (limited); from $7/mo	Similar to Railway, slightly more manual
Fly.io	Runs apps close to your users globally.	Free tier; from $5/mo	More control, slightly more complex
Supabase	Database and authentication as a service.	Free tier; from $25/mo	Use alongside a static host for your frontend

Deploying with your agent

You don't need to figure out deployment commands yourself. Claude can handle it:

```
Deploy this project to Vercel
```

For a first deployment, Claude will walk you through: 1. Connecting your account (usually opening a browser to sign in) 2. Linking your project 3. Pushing the code 4. Verifying the deployment

After the first time, deployments happen automatically when you push to GitHub — or you can ask Claude to deploy manually.

The deployment workflow

Once set up, here's the typical flow:

1. **Build locally** — work on your project with Claude
2. **Test** — make sure it works on your machine
3. **Commit and push** — save your changes and push to GitHub (Chapter 7)
4. **Auto-deploy** — your hosting platform detects the push and deploys automatically
5. **Verify** — check the live URL to make sure it looks right

Most platforms give you a **preview URL** for every push — so you can check your changes before they go live to the main URL.

Domains

When you first deploy, you get a free URL from the platform — something like `my-project.vercel.app` or `my-project.netlify.app`. That works fine for testing and sharing.

When you're ready for a real domain (like `www.cleanpaws.com`), you:

1. Buy a domain from a **registrar** (Namecheap, Google Domains, Cloudflare — around $10-15/year)
2. Point it to your hosting platform (the platform's docs explain how)
3. The platform handles the SSL certificate (the padlock icon in browsers) automatically

Your agent can help with the configuration:

```
Help me connect the domain cleanpaws.com to my Vercel deployment
```

Environment variables

Some projects need secrets — API keys, database passwords, service credentials. You never put these in your code (anyone who sees your code would see your secrets). Instead, you store them as **environment variables** on the hosting platform.

Every hosting platform has a settings page where you add environment variables — key-value pairs like:

```
DATABASE_URL = postgres://user:password@host:5432/mydb
API_KEY = sk-abc123...
```

Your code reads these at runtime. The values never appear in your code or on GitHub.

```
Add the environment variables from my .env file to Vercel
```

What can go wrong

Problem	What happened	Fix
Site works locally but not when deployed	Missing environment variables, or the build command is different	Check the platform's deploy logs; add missing env vars
"Page not found" on refresh	Single-page apps need a redirect rule	Ask Claude to add the redirect configuration for your platform
Site is slow	Large images, no caching, server far from users	Ask Claude to optimize images and add caching headers
Deploy fails	Build error — usually a dependency issue	Check the build logs; ask Claude to fix the error

Static vs. dynamic: how to choose

If your project…	You need…	Start with…
Has no database, no user accounts, no server logic	Static hosting	Vercel or Netlify (free)
Has a database or user authentication	Dynamic hosting + database	Railway or Render
Is a simple API or backend service	Dynamic hosting	Railway or Fly.io
Is a frontend that talks to a separate API	Static hosting for frontend + dynamic for API	Vercel (frontend) + Railway (API)

If you’re not sure, start static. You can always add a backend later.

Glossary

Quick reference for terms used in this guide, listed alphabetically.

Agent — A piece of software that can take actions autonomously: creating files, writing code, running commands, and making decisions to complete a task. Unlike a chatbot, an agent doesn't just respond — it acts.

Agentic loop — The cycle an agent follows: read instruction → decide what to do → do it → check the result → decide next step → repeat until done.

API (Application Programming Interface) — A way for one piece of software to talk to another. In this guide, it usually means the direct connection to an AI model that you pay for per use, as opposed to a subscription.

API key — A password-like string that identifies your account to a service. You need one to use AI models via API access. Keep it secret — anyone with your key can use your account and run up your bill.

CI/CD (Continuous Integration / Continuous Deployment) — Automated systems that test your code and deploy it whenever you make changes. Advanced concept — not needed for beginners, but mentioned in comparison tables.

CLAUDE.md — A markdown file in your project that Claude Code reads automatically at the start of every session. Contains standing instructions: what the project is, what rules to follow, what to avoid.

CLI (Command Line Interface) — A text-only way to interact with your computer. You type commands, press Enter, get results. No buttons or menus.

Commit — A saved snapshot of your project at a point in time, with a short message describing what changed. Created using Git.

Commit message — The short note attached to a commit describing what changed and why. Good commit messages help the next session's agent understand the project's history.

Context window — The fixed-size memory an agent has during a session. It holds your conversation, files the agent has read, and instructions. Typical size: 100,000–200,000 tokens. When full, early information gets dropped.

Deploy / Deployment — The process of making your project accessible on the internet. Moving it from your computer to a server where people can use it.

Framework — A pre-built structure of code that provides common functionality so you don't start from scratch. React, Next.js, and Vue are JavaScript frameworks. Django and Flask are Python frameworks.

Git — A version control system that tracks changes to your files over time. Think of it as an infinite undo button that also lets multiple people work on the same project. Every saved checkpoint is called a commit.

Hook — An automated action that fires at a specific moment in the agent's workflow (e.g., before executing a command, after finishing a task). Used for automated guardrails and quality checks.

Hosting — Keeping your project running on a server so people can access it via the internet. Can be free (Vercel free tier) or paid, depending on the service and your project's needs.

IDE (Integrated Development Environment) — A visual application for writing code, with features like syntax highlighting, file management, error detection, and code execution. VS Code, Cursor, and Windsurf are IDEs.

LLM (Large Language Model) — The AI model that powers an agent. It reads your instructions, reasons about them, and generates responses. Claude, GPT, and Gemini are all LLMs.

Markdown (.md) — A simple text format that uses plain characters for formatting: `#` for headings, `**bold**` for bold, `-` for bullet points. Easy to write in any text editor, and renders with nice formatting in tools like VS Code and GitHub. Software projects use markdown files everywhere — READMEs, documentation, instruction files.

MCP (Model Context Protocol) — A standard protocol that connects Claude Code to external tools and services. An MCP server is the bridge — one for GitHub, one for Slack, one for your database, etc.

MCP server — A specific integration that implements MCP for one tool or service. Adding an MCP server lets the agent interact with that tool directly.

Node.js — A program that lets you run JavaScript outside a web browser. Some older Claude Code installation methods required it. The current native installer does not.

Platform — The tool layer between you and the AI model. It handles the interface, file management, and deployment. Claude Code, Cursor, and Replit are all platforms with different tradeoffs.

Railway — A cloud platform for hosting back-end services, databases, and full-stack applications. Alternative to Vercel for projects that need server-side infrastructure.

Skill — A reusable bundle of instructions (stored as a SKILL.md file) that teaches the agent how to handle a specific type of task. Like a recipe card the agent can reference.

Sub-agent — A smaller agent spawned by a main agent to handle a specific sub-task in parallel. Claude Code uses sub-agents to work on multiple things simultaneously.

Subscription — A monthly fee ($20/mo is the common entry point) that gives you access to an AI model through a product interface. Usage is included up to a limit. The alternative to API access.

Terminal — The application that provides a CLI. On Mac: Terminal or iTerm. On Windows: PowerShell or Command Prompt. On Linux: any terminal emulator. Same concept as CLI, different word.

Token — The unit models use to measure text. One token is roughly 3/4 of a word. A 1,000-word document is about 1,333 tokens. Models charge per token and have token limits on how much they can process at once.

Vercel — A cloud platform for hosting websites and web applications. Popular with front-end developers. Offers free tier for small projects.

VS Code (Visual Studio Code) — A free, widely-used IDE made by Microsoft. Claude Code runs inside it as an extension.