SETUP AND REFERENCE BOOKLET

Setting Up Claude

A complete, follow-along guide for Mac and PC: Claude Code, Cowork, plugins, and connectors.

VERSION

v2 · June 2026

Welcome: how to use this booklet

Claude is an AI assistant made by Anthropic. You can chat with it, ask it questions, give it documents and data to work through, and hand it real tasks to carry out. This booklet is about setting Claude up properly on your own computer so it can do more than just talk: it can read and write your files, run jobs across several steps, and connect to the tools you already use.

This pack is a complete, follow-along guide to setting Claude up on a Mac or PC and using it well. It is written for two people at once: someone new who is being set up for the first time, and the person helping them do it. You do not need to be a developer to follow it, and nothing here is dumbed down.

By the end you will have three things:

1. The Claude desktop app, which gives you Chat (conversations and analysis) and Cowork (Claude carrying out multi-step tasks across your files).
2. Claude Code, which runs in your terminal and inside VS Code, for building things and any file-heavy work in a project folder.
3. Connectors to your own tools, such as Shopify, so Claude can work with your real accounts and data.

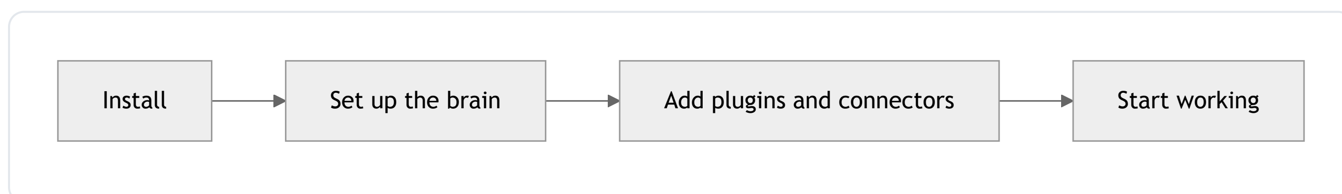
For a fresh setup, work through the chapters in order. If you already have part of this running, jump straight to the chapter you need.

You do not have to do all of this by hand. Chapter 7 gives you ready-made prompts you can paste into Claude so it sets things up with you and explains each step as it goes; the full set is in Appendix D, and there are downloadable starter files in the downloads folder (Appendix E). Many people learn the tool fastest this way: by building their own setup with Claude beside them.

At the back there is a one-page printable quick-start checklist (Appendix B), reference tables for commands and links (Appendix C), the prompt pack (Appendix D), and a list of the downloadable starter files (Appendix E).

How to read the diagrams

Boxes are steps or things. Diamonds are decisions, where the path splits depending on your answer. Arrows show the order you move through them. Read left to right, or top to bottom, following the arrows.



1. The big picture: Chat, Claude Code, and Cowork

Claude is not one product. It comes in three surfaces, and the first thing to get right is which one a task belongs to. Pick the wrong surface and the work feels harder than it should. Pick the right one and most of the friction disappears.

Here are the three, in plain language.

Claude (chat). The website at claude.ai and the desktop app. This is the conversation: you ask, it answers. Questions, writing, drafting, explaining, summarising, thinking something through. It does not touch the files on your computer; everything stays inside the chat window.

- Use it when: you want a quick answer, some writing, or help thinking through an idea.

Claude Code. An agentic coding tool that runs in your terminal and inside VS Code. It works inside a project folder on your computer, and it can read and write files, run commands, edit code, and use tools. It is built for building: software, websites, and any file-heavy work that lives in a project.

- Use it when: you want to build or edit code, a website, or work directly inside a folder of files.

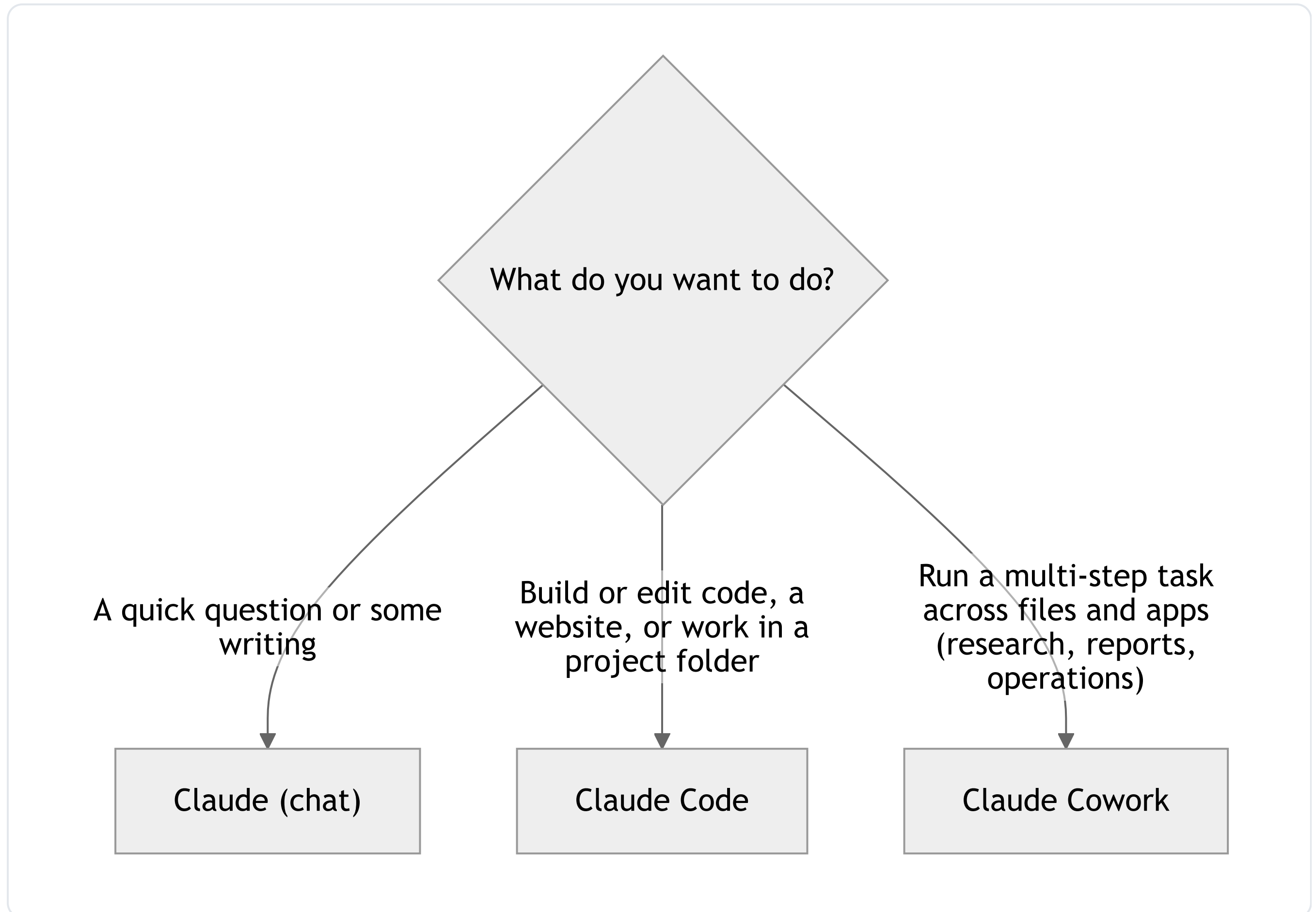
Claude Cowork. Anthropic's desktop agent for knowledge work, available to paid subscribers on macOS and Windows (Windows support arrived in February 2026). You describe a multi-step task; Claude plans it, executes it across the files and apps you have connected, and you steer as it goes. It can read, edit and create files in folders you grant it. It is built for non-developers (researchers, analysts, operations, finance, legal): people who work with documents, data and files rather than code.

- Use it when: you want a multi-step task run across your own files and apps, such as research, reports, or operations.

One sentence to hold onto: chat is for talking, Claude Code is for building inside a project, Cowork is for getting multi-step work done across your files.

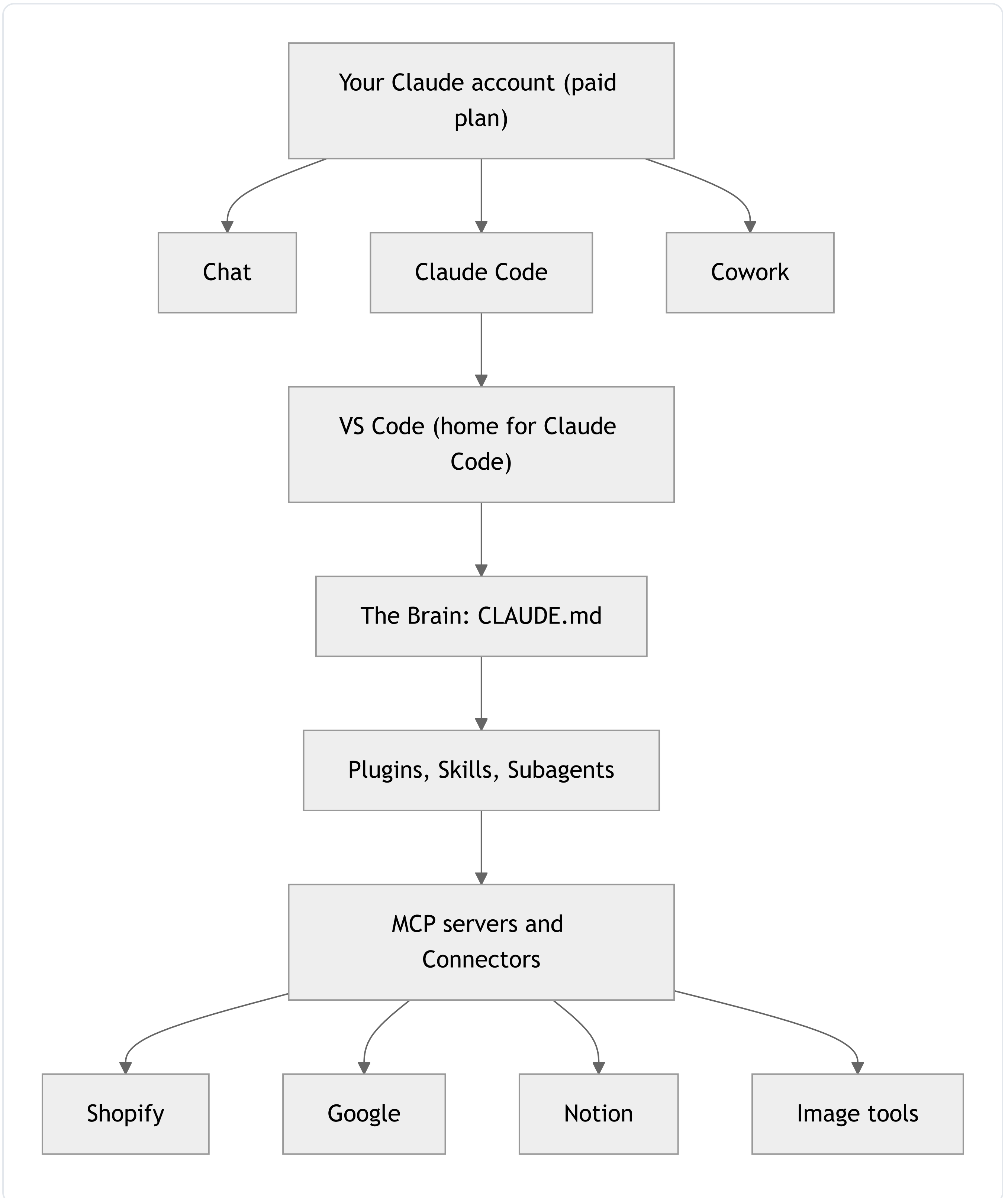
One practical note before you choose: Claude Code and Cowork both need a paid Claude plan (Pro, Max, Team, or Enterprise). The free tier covers basic Chat only. The install chapters say more about this.

Which surface for which job



The system map

The three surfaces are the starting point, not the whole picture. As you set Claude up properly, you add layers: a place to keep your project rules, extra abilities through plugins and skills, and connections out to the services you already use. The map below shows how it all fits together, top to bottom.



Read the map from the top. Everything starts with one paid Claude account. That account gives you the three surfaces. Claude Code has a comfortable home in VS Code. Inside a project sits CLAUDE.md, the brief and memory that tells Claude how you want it to work. Plugins, skills and subagents add specialist abilities on top of that. MCP servers and connectors then reach outward to the real services you use, such as Shopify, Google, Notion, and image tools.

Do not worry about building all of this at once. The rest of this booklet adds the map up one layer at a time, starting with accounts and installation and working outward to connectors. By the end you will have built the whole picture, in order, and understood each piece as you added it.

2. Before you start: accounts and what you need

Setting up Claude goes faster when you gather a few things first. This chapter is a short pre-flight check. Work through it before you install anything, so you do not stop halfway to sign up for an account or hunt for a password.

What to have ready

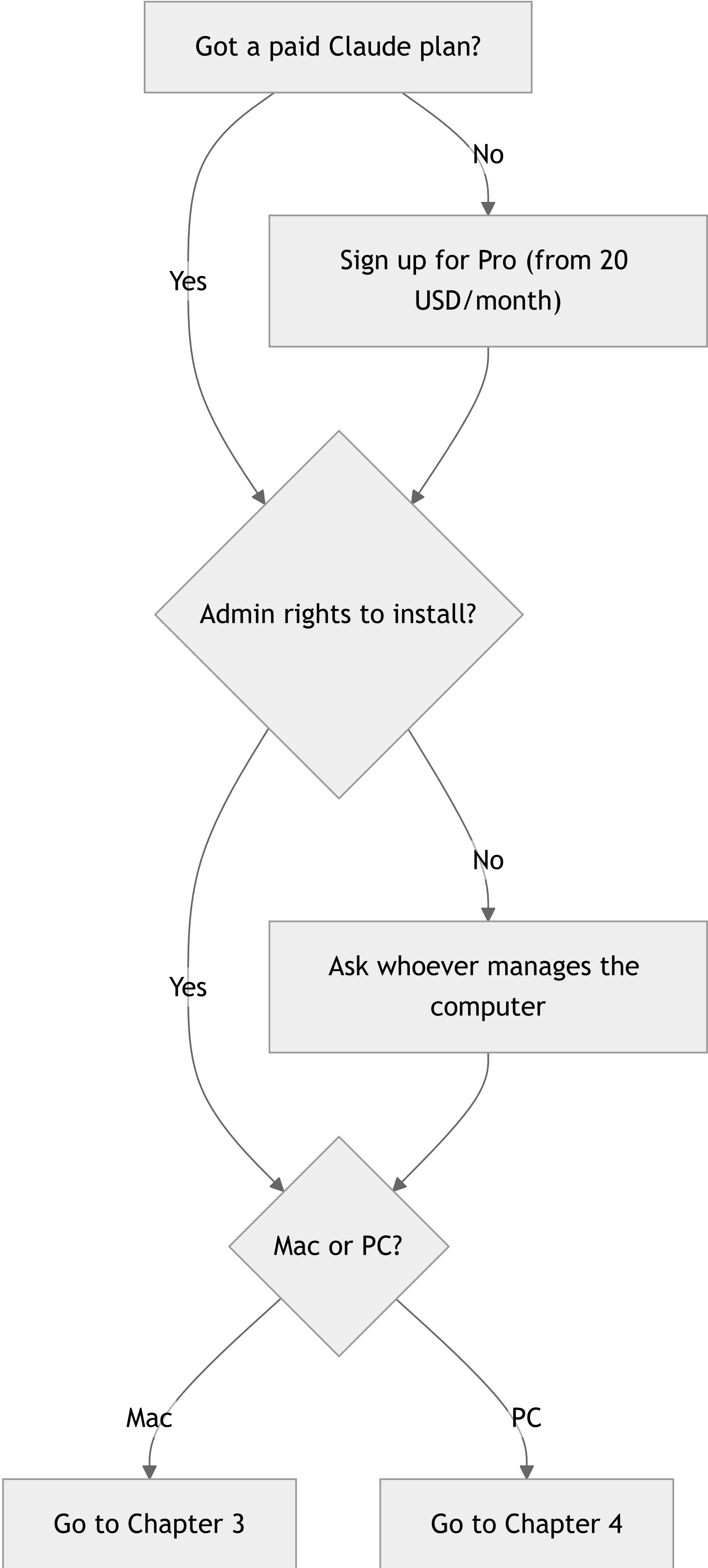
1. **A Claude account on a paid plan.** Claude Code and Claude Cowork are not part of the free Claude.ai plan. The entry point is **Claude Pro**, from 20 USD per month, which is enough for most people getting started. If you expect heavy daily use, **Claude Max** gives you more headroom. Team, Enterprise, and Claude Console (API) accounts also qualify. You will create or sign in to this account during setup.
2. **A computer that meets the minimum requirements.** A reasonably recent Mac or Windows PC is fine. See the table below.
3. **Admin rights on that computer.** Installing software is smoother when your account can approve installations. If this is a work machine you do not control, ask whoever manages it to grant admin rights or to install on your behalf.
4. **An email address and a payment card.** You need these to create the paid Claude account if you do not already have one.
5. **A free GitHub account (recommended).** GitHub is a place to store code online. You do not need it to chat with Claude, but it helps for code projects and is required by some plugins. Sign up free at github.com. Skip this if you are only doing document and file work.

You may also want accounts for services you plan to connect later, such as **Shopify**, a **Google** account (Gmail, Calendar, Drive), or **Notion**. These are optional now; you connect them when you reach the relevant chapter.

Minimum requirements

Item	Requirement
Mac	macOS 13.0 or later
Windows	Windows 10 (version 1809) or later, or Windows Server 2019+
Linux	Ubuntu 20.04+ / Debian 10+
Memory	4 GB RAM or more
Internet	A working internet connection

Pre-flight checklist


Got a paid Claude plan?
Yes
No
Sign up for Pro (from 20 USD/month)
Admin rights to install?
Yes
No
Ask whoever manages the computer
Mac or PC?
Mac
PC
Go to Chapter 3
Go to Chapter 4

Next step

Once you have a paid plan, admin rights, and a supported computer, you are ready to install. If you are on a Mac, go to **Chapter 3**. If you are on Windows, go to **Chapter 4**.

3. Install on a Mac

This chapter sets up a Mac for Claude Code, the desktop app, and the supporting tools you will use later. Do the steps in order. You do not need to be a programmer to follow them.

Before you start

Check your Mac meets the requirements: macOS 13.0 or later, at least 4 GB of RAM, and an internet connection. Admin rights on the machine help when installing software. You will also need a paid Claude plan (Pro, Max, Team, Enterprise, or a Console account); the free plan does not include Claude Code or Cowork.

Most steps run in Terminal, which is the Mac app for typing commands. To open it:

1. Press Cmd and Space together to open Spotlight.
2. Type `Terminal`.
3. Press Return.

A small window opens with a text prompt. You type a command, press Return, and wait for it to finish before typing the next one.

Step 1: Install Homebrew

Homebrew is the package manager for macOS. A package manager installs and updates software for you from one place, so you do not have to hunt for downloads. Many later steps rely on it.

Paste this line into Terminal and press Return.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
```

It may ask for your Mac password (the cursor will not move as you type; that is normal) and take a few minutes. More about Homebrew is at brew.sh.

Step 2: Install Claude Code

Claude Code is the tool that runs in your terminal and works inside a project folder on your computer. The recommended way to install it is the native installer, which keeps itself up to date automatically.

```
curl -fsSL https://claude.ai/install.sh | bash
```

If you would rather install it through Homebrew, use the line below instead. Note that the Homebrew version does not auto-update; you upgrade it yourself with `brew upgrade claude-code`.

```
brew install --cask claude-code
```

Pick one method, not both.

Step 3 (optional): Install Node.js

Node.js lets some optional tools run. You will need it for certain MCP tools that use `npx`, and for the Shopify CLI covered later in this guide. If you know you will not touch those, you can skip this step and return to it later.

```
brew install node
```

Step 4: Install Git and the GitHub CLI

Git tracks changes in code projects, and the GitHub CLI (`gh`) connects you to GitHub, which some plugins use. Install both together.

```
brew install git gh
```

Then sign in to GitHub. This opens a short browser sign-in (a free GitHub account is fine).

```
gh auth login
```

Follow the prompts. If you are not working on code projects yet, you can still install these now and sign in later.

Step 5: Install the Claude desktop app

The desktop app is where Chat and Cowork live. Download it from claude.ai/download, open the downloaded file to install it, then open the app and sign in with your Claude account. Cowork runs inside this app, and the app needs to stay open while a Cowork task is working; closing it ends the task.

Step 6: First run and verify

Now confirm Claude Code works.

1. In Terminal, move into a project folder (any folder you keep work in).
2. Run the command below and follow the browser prompt to log in.

```
claude
```

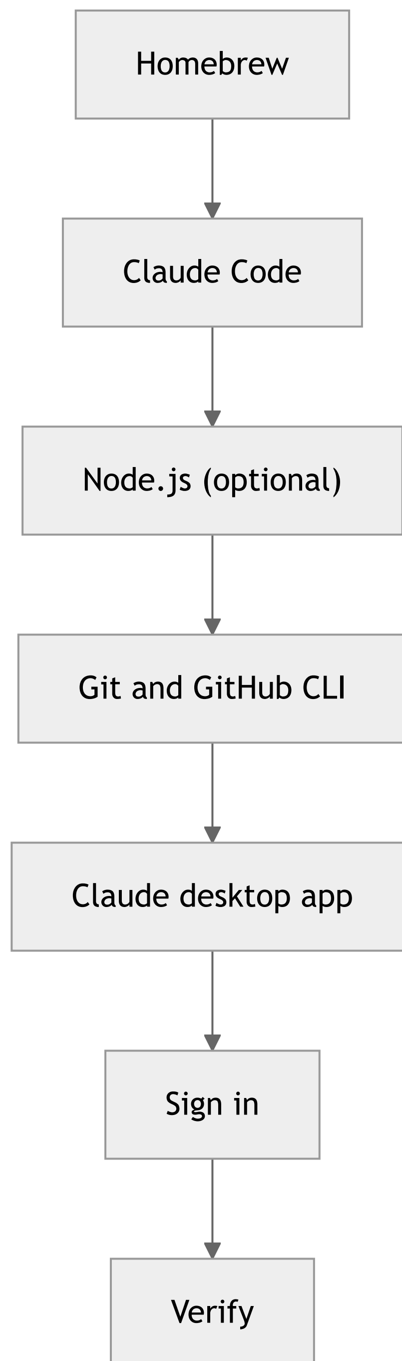
Once you are signed in, check the install with these two commands. The first prints the version number; the second runs a deeper health check.

```
claude --version
```

```
claude doctor
```

If both run without errors, your Mac is set up.

Install order at a glance



Tips

- If Terminal says `claude` is "not found" straight after installing, close the Terminal window and open a new one, then try again. A fresh window picks up the newly installed command.
- Copy each command exactly, including the symbols. A single missing character will stop it working.

- You do not need to understand what each command does internally to run it safely. These are the standard, official install commands.

4. Install on a PC (Windows)

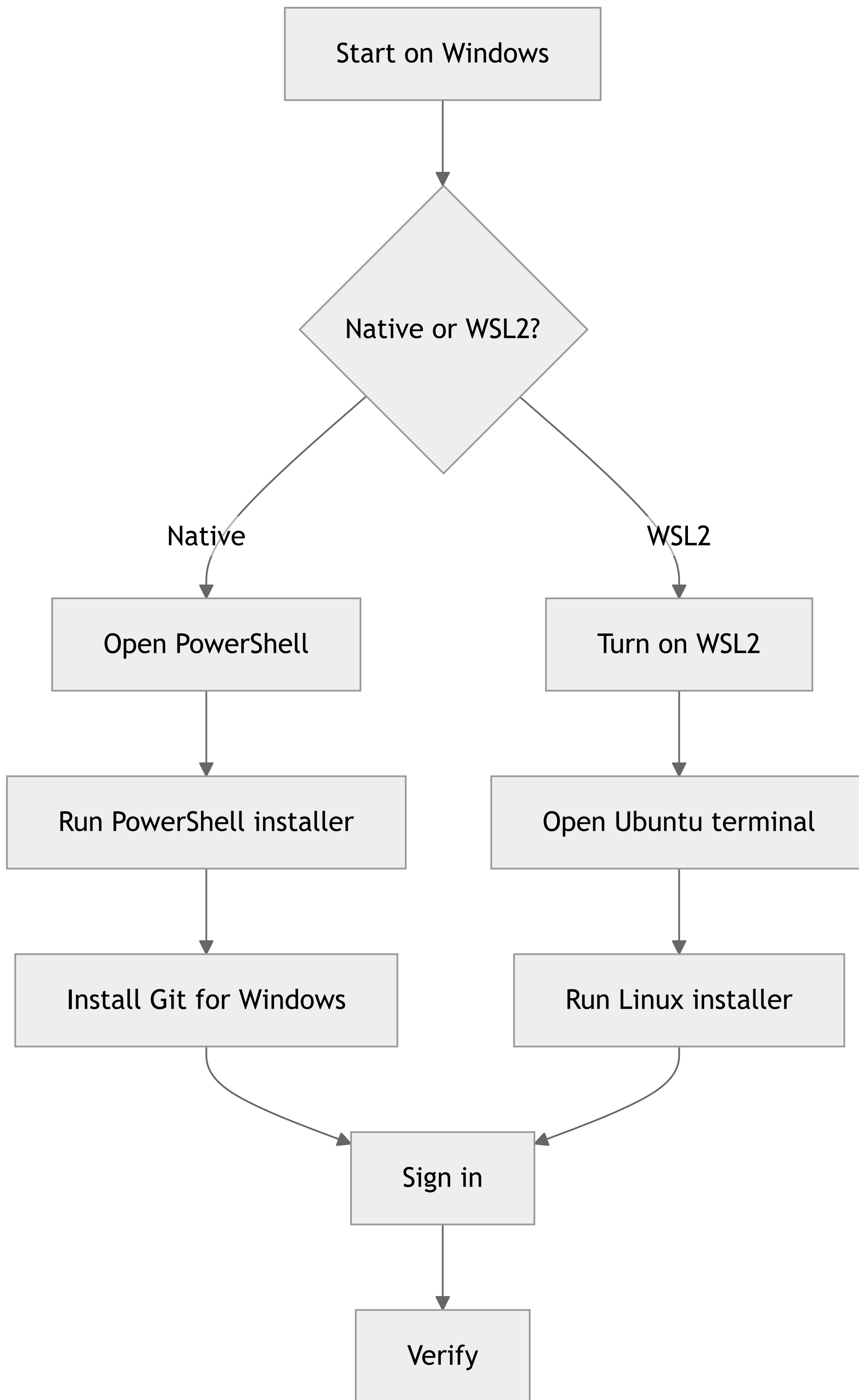
This chapter sets up Claude on a Windows computer. There are two routes. Most people should use Route 1. Only choose Route 2 if you already do serious development with Linux tools.

Before you start, you need a paid Claude plan (Pro, Max, Team, Enterprise, or a Console account). The free plan does not include Claude Code or Cowork. You will also want a stable internet connection. Windows 10 (version 1809 or later) or Windows Server 2019 or later, with 4 GB of RAM or more, covers Claude Code.

One thing to check if you want Cowork: on Windows, Cowork needs the Windows Pro, Enterprise, or Education edition, because it relies on a Windows feature called Hyper-V. Windows Home does not support Cowork. Claude Code itself runs fine on Windows Home; only Cowork has this limit. To see which edition you have, open Settings, then System, then About.

Two routes, and how to choose

- **Route 1, Native Windows.** Claude Code installs and runs directly on Windows. Simplest path, good for most non-developers. Choose this unless you have a specific reason not to.
- **Route 2, WSL2.** WSL2 (Windows Subsystem for Linux) runs a full Ubuntu Linux system inside Windows. Choose this only if you work with Linux command-line tools day to day.


Start on Windows
Native or WSL2?
Native
WSL2
Open PowerShell
Run PowerShell installer
Install Git for Windows
Turn on WSL2
Open Ubuntu terminal
Run Linux installer
Sign in
Verify

Opening PowerShell, and telling it apart from CMD

Click the Start menu, type `PowerShell`, and open it. Windows has two command tools that look similar. The difference matters because the install commands are written for PowerShell.

- **PowerShell:** the prompt begins with `PS C:\`.
- **Command Prompt (CMD):** the prompt begins with just `C:\`, without the `PS`.

If you do not see `PS` at the start of the line, close the window and open PowerShell from the Start menu.

Route 1, Native Windows

1. Open PowerShell (Start menu, type `PowerShell`).
2. Install Claude Code with the native installer:

```
irm https://claude.ai/install.ps1 | iex
```

If you prefer to use the Windows package manager, you can instead run:

```
winget install Anthropic.ClaudeCode
```

3. Install Git for Windows so Claude Code can use its Bash tool. Download it from git-scm.com/downloads/win, run the installer, and accept the default options. This is recommended on native Windows.

That is the Windows-specific part of Route 1. Now skip to **Shared steps** below.

Route 2, WSL2

This route is for development work with Linux tools.

1. Turn on WSL2. Open PowerShell as Administrator (right-click the Start button and choose Terminal (Admin), or PowerShell (Admin)) and run:

```
wsl --install
```

Restart when Windows prompts you; this installs Ubuntu. Route 2 suits people who already work with Linux tools, so if that is not you, use Route 1.

2. Open the Ubuntu terminal from the Start menu.
3. Inside that Ubuntu window, run the macOS and Linux installer (not the PowerShell one):

```
curl -fsSL https://claude.ai/install.sh | bash
```

Run all later Claude Code commands inside the Ubuntu terminal, not in PowerShell. Then continue to **Shared steps**.

Shared steps (both routes)

1. **Install Node.js if needed.** Some tools expect Node.js. If you do not have it, download the installer from nodejs.org and run it. (The native installers above do not require Node.js by themselves, but it is useful to have.)
2. **Install the Claude desktop app.** This is for Chat and for Cowork. Download it from claude.ai/download, install it, open it, and sign in with your Claude account.
3. **Sign in to Claude Code.** Open a terminal in your project folder (PowerShell for Route 1, Ubuntu for Route 2) and run:

```
claude
```

A browser window opens. Follow the prompt to log in. After that, Claude Code remembers you.

4. **Verify the install.** Run:

```
claude --version
```

If that prints a version number, the install worked. For a deeper check that looks for common problems, run:

```
claude doctor
```

Tips

- **PowerShell versus CMD.** The two windows look alike. PowerShell shows `PS C:\` at the start of the line; CMD shows only `C:\`. The install command `irm ... | iex` is written for PowerShell and will not work in CMD. If a command fails straight away, check which window you are in first.
- **You do not need admin rights** for the native Windows install. If your work computer is locked down, the native installer should still run under your own user account. Admin rights only help when you are installing other software that asks for them.
- **One window per route.** On Route 1 do everything in PowerShell. On Route 2 do everything in the Ubuntu terminal. Mixing the two is the most common source of confusion.

Once `claude --version` reports a version and the desktop app is signed in, your PC is ready. Chapter 7 shows how to set up your workspace by talking to Claude: it can build your folders and your CLAUDE.md with you, explaining each step.

5. VS Code: a home for Claude Code

Claude Code can run two ways: in a plain terminal window, or inside VS Code, a free code editor made by Microsoft. The terminal works perfectly well, but VS Code gives Claude Code a visual interface that most people find more comfortable, especially when you want to see exactly what Claude is about to change before it changes it.

You do not have to choose one forever. The terminal `claude` command and the VS Code extension share one engine, plus your sign-in, settings, and history. One thing to know when installing: the extension brings its own copy of Claude Code, so the panel works even if you never ran the terminal installer. It does not work the other way round, though; installing the extension does not add the `claude` command to your terminal, so if you also want to type `claude` in VS Code's built-in terminal, install it from the Mac or Windows chapter too.

Installing the extension

1. Install VS Code from code.visualstudio.com. Download the version for your operating system, then open the installer and follow the prompts. The Claude Code extension needs VS Code version 1.98.0 or later; a fresh install will be fine, but if you already had VS Code, update it first (Help, then Check for Updates; on Mac, Code, then Check for Updates).
2. Open the Extensions view. On Mac press Cmd+Shift+X; on Windows press Ctrl+Shift+X.
3. In the search box, type "Claude Code". Find the official extension (its marketplace id is `anthropic.claude-code`) and click Install.
4. Open your project folder so Claude has something to work on. Go to File, then Open Folder, and choose the folder for the work you want to do.
5. Open the Claude Code panel from the sidebar. If it asks you to sign in, do so with your Claude account and follow the prompts.

The extension needs a paid Claude subscription (Pro, Max, Team, Enterprise, or a Console account). You do not need an API key.

What the extension adds over the terminal

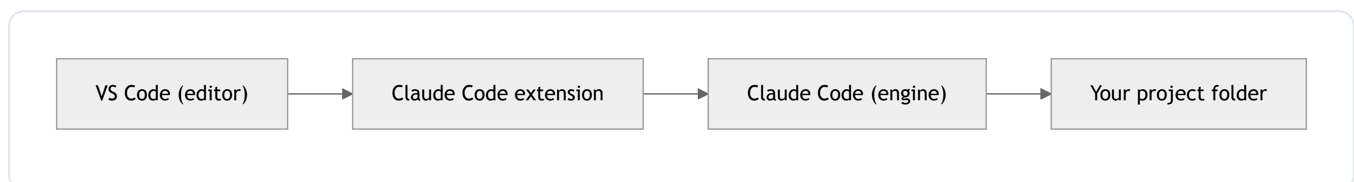
In the plain terminal, Claude Code reads and writes files and runs commands as text. The VS Code extension wraps the same engine in a visual layer:

- A sidebar panel that lives alongside your files, so Claude is always one click away.
- Inline diffs: before any edit is saved, you see the proposed change shown old-line against new-line, and you accept or reject it. You can also tweak the proposed change directly in the diff before accepting; Claude is told you edited it and carries on from your version.

- Plan mode: one of three modes (default, Plan, auto-accept) you pick in the prompt box. In Plan mode Claude writes its plan out as a document you can comment on, and waits for your approval before it changes anything.
- @-mentioning: type @ to point Claude at a specific file, or even a specific range of lines, instead of describing it in words.
- Conversation history, so you can scroll back through what was said and done.
- Multiple chats in tabs, so you can keep separate jobs running side by side without them tangling.

A few shortcuts make these fast to reach: Cmd/Ctrl+Esc toggles between the editor and the Claude panel, Option/Alt+K drops an @-mention to the file or lines you have selected, and Cmd/Ctrl+Shift+Esc opens a new conversation in a tab. You will not need them on day one, but they are worth knowing.

How the pieces fit together



VS Code hosts the extension. The extension is a front end to the Claude Code engine. The engine does the work inside the project folder you opened. The terminal `claude` command plugs into that same engine; the two front ends share one tool underneath.

> **Tip:** Keep one project folder open at a time. Claude takes its context from the folder you open, including the CLAUDE.md file at its root. One folder open means clear, focused context; several folders or a parent folder full of unrelated projects makes Claude's job harder and its answers vaguer.

6. The brain: CLAUDE.md and project memory

This is the single most useful habit in this whole booklet. Out of the box, Claude is clever but forgetful. It does not know who you are, what you are working on, or the rules you care about. So at the start of every conversation you would be explaining yourself from scratch. The fix is to give Claude a brain: a small set of notes it reads automatically, every time, before you say a word.

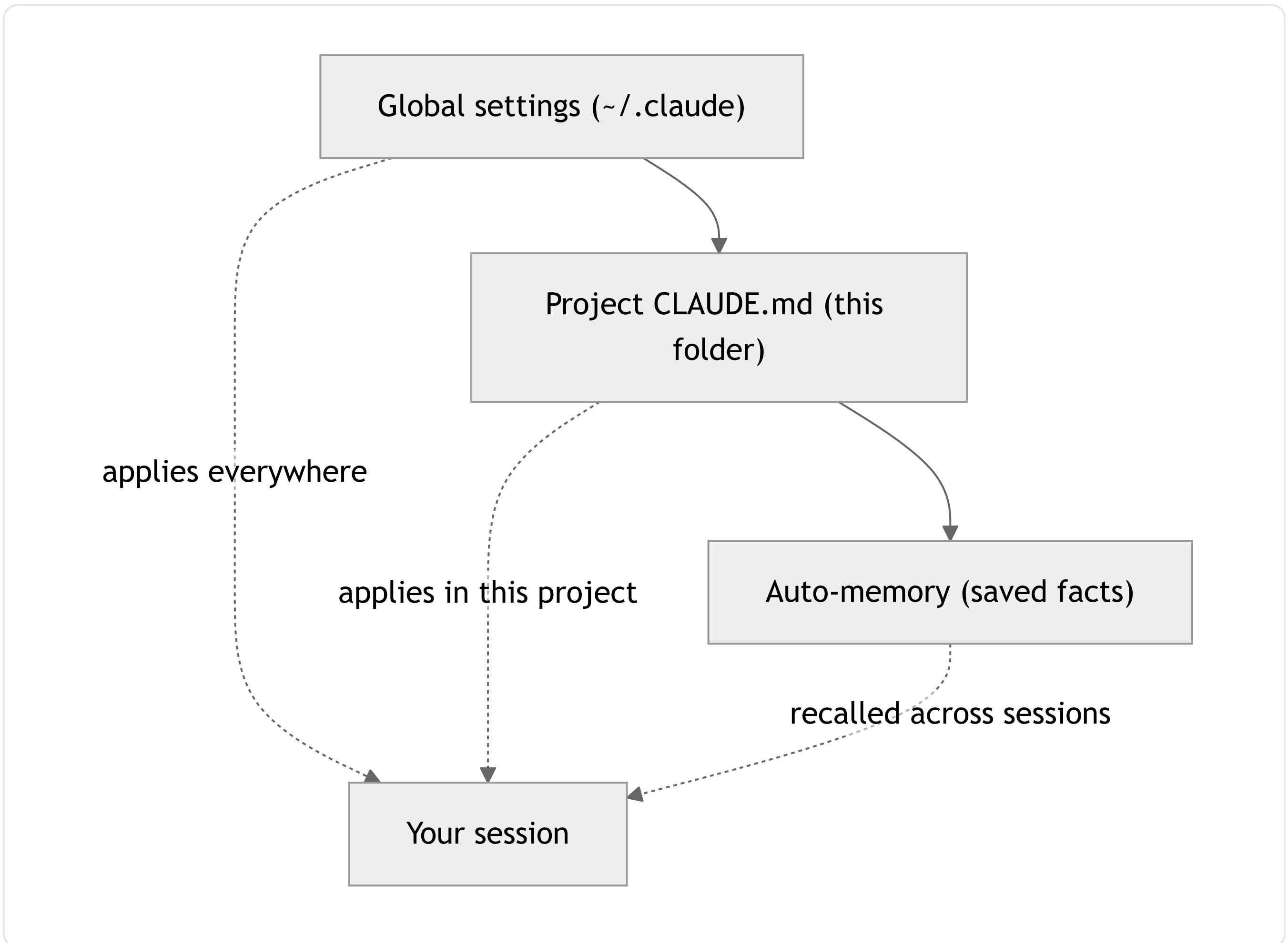
What CLAUDE.md is

`CLAUDE.md` is a plain markdown file (just text with simple formatting) that lives at the root of your project folder. Claude Code loads it automatically at the start of every session in that folder. You do not have to open it or paste it in; it is simply there.

Think of it as the project's brief and house rules in one place: who you are, what the project is, how you want Claude to work, the things it must never do, and where the important files live. Write it once, refine it over time, and every future conversation starts already informed.

The three layers of memory

Claude's memory works on three layers, from broadest to most specific.



1. Global user settings live in a folder called `~/.claude` on your computer. These apply everywhere, in every project. Use them for preferences you always want.
2. The project `CLAUDE.md` applies only inside that one project folder. Use it for anything specific to this piece of work.
3. Auto-memory is made up of facts Claude saves as you work and recalls in later sessions, so useful details are not lost when a conversation ends.

You do not have to manage all three by hand. The one you will write yourself, and the one that matters most, is the project `CLAUDE.md`.

A starter CLAUDE.md

Copy this skeleton into a file named `CLAUDE.md` at the top of your project, then fill in the blanks. Keep it short and specific to you.

```
# CLAUDE.md , project memory

## Who I am / what this project is
One or two lines: your name or business, and what this
project is for.

## How I want you to work
How you like answers: direct, options when there is a real
choice, British English, no marketing tone. Anything about
pace or format.

## Hard rules and never-do
The things that must always hold, and the things to never do.
Be blunt. These are not suggestions.

## Where the important files live
Point to the source-of-truth files (a brand guide, a price
list, a spec) rather than pasting their contents here.

## Current focus
What we are working on right now. Update this as it changes.
```

You do not have to write this from a blank page. In a new project the `/init` command generates a starter CLAUDE.md for you to edit, and Chapter 7 shows how to have Claude interview you and draft one with you. The skeleton above is simply what a good one looks like once it is filled in.

A reusable folder template

A brain works best when the project around it is tidy. The simplest way to keep work findable, and to make it clear what belongs where, is to use numbered top-level folders. Numbers force an order and stop the list rearranging itself.

```
my-project/
├── CLAUDE.md
├── 00 brand
├── 01 [flagship project]
├── 02 products
├── 03 web
├── 04 email
├── 05 applications
├── 06 correspondence
├── 07 print
├── 08 social
├── 09 strategy
└── 10 finance
```

Adapt the names to your own business; a builder, a charity and a shop would each label these differently. Give every folder a short `README` file saying what belongs in it, and use clear, dated or versioned file names (for example `2026-06-pricing-v2`). The payoff is that work is easy to find, responsibility for each area is obvious, and Claude can be pointed at exactly the right place.

Five tips for a good CLAUDE.md

1. Keep it short. A page that gets read beats five pages that get skimmed.
2. State the goal plainly. If Claude knows what you are trying to achieve, it makes better calls on everything else.
3. List your never-do items explicitly. The hard rules are the highest-value lines in the file.
4. Link to source-of-truth files instead of pasting them in. Point to the brand guide; do not copy it. One source, no drift.
5. Update it when things change. A stale brain quietly steers you wrong. When the project moves on, edit the file the same day.

Start small. Even three honest lines about who you are and what you want will change how every conversation goes.

7. Set it up by talking to Claude

The chapters around this one explain what each piece is: the brain, plugins, connectors. This chapter is different. It hands you the exact words to paste into Claude so that Claude sets things up with you, and explains what it is doing while it does it. It is the fastest way to learn, because you end up with a working setup and an understanding of how it was built at the same time.

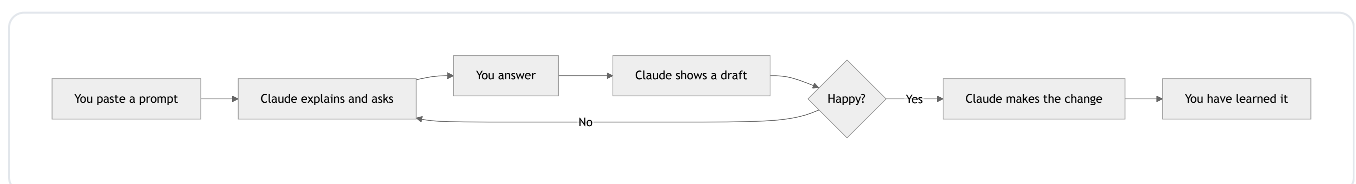
You do not have to do it this way. You can follow the step-by-step chapters by hand, or run the ready-made scripts in Appendix E. But if you are new, this is the route worth trying first. Paste a prompt, read what Claude tells you, answer its questions, and watch your workspace take shape.

How to use these prompts

1. Open Claude in the place that fits the job. For setting up files and folders, that is Claude Code in your project folder, or a Cowork session connected to that folder. For a quick explanation, plain Chat is fine.
2. Copy one of the prompts below. On screen, use the Copy button on the block; on paper, type it out.
3. Paste it in and send it. Then follow along: answer the questions, read the explanations, and approve each step before Claude makes a change.

You do not even have to type these. A dictation tool lets you speak a prompt and have the words land in the box, tidied up with punctuation as you go, which is far quicker for anything this long. See Chapter 14 for two that are worth having.

Every prompt below tells Claude to go slowly, explain as it goes, and check with you before it creates or changes anything. You are always the one who says go ahead. If you do not like what it proposes, say so, and it will adjust.



Prompt 1: set me up from scratch (and teach me as you go)

The big one. Use this the first time you sit down with Claude Code in a new, empty project folder. It walks the whole setup with you.

```
I have just installed Claude Code and I am brand new to it. I want
you to help me set up this project, and teach me what you are doing
as we go, so I actually understand it and do not just end up with
files I did not write.

Work in small steps and stop for me after each one. Start by telling
me, in plain English and a few sentences, what we are going to set up
and why it is worth it. Then take the first step only: explain what a
CLAUDE.md file is, ask me a few short questions about me and my work,
and from my answers draft one for me to read. Do not create any files
until I have seen the draft and said go ahead.
```

Prompt 2: build my folder structure

Gets Claude to create a tidy, numbered set of folders with a short README in each, once you have agreed on the names.

```
Help me set up a tidy folder structure for this project. I want
numbered top-level folders so they always stay in the same order, and
a short README in each one saying what belongs there.

First ask me what kind of work this project covers, then suggest a set
of folders based on my answer. Let me adjust the names. Once I confirm,
create the folders and the README files, and as you go, explain the
naming convention so I can keep it up myself. Do not create anything
until I have approved the list.
```

Prompt 3: interview me to write my CLAUDE.md

This is the memory prompt. Claude asks you questions, one at a time, and turns your answers into a brief it reads at the start of every future session. If you already ran Prompt 1 you will have a CLAUDE.md draft; reach for this one when you skipped the full setup and just want the memory file, or when you want to redo it more thoroughly.

```
Interview me so you can write a good CLAUDE.md for this project. Ask
me one question at a time and keep each one short. Cover, in order:
who I am and what this project is; how I like you to work; my hard
rules and the things you must never do; where my important files live;
and what I am focused on right now.

When you have my answers, show me a draft CLAUDE.md and explain why
each part is there. Let me edit it. Only save the file once I am
happy with it.
```

Prompt 4: teach me as we work

Not a setup prompt, a way of working. Paste it at the start of any task while you are still learning, and Claude treats the job as a lesson as well as a task.

```
For the task I am about to give you, briefly explain what you are doing
and why as you go, especially anything I could reuse another time. If
there is a simpler or more standard way to do what I ask, tell me before
you start. Treat this as me learning the tool, not just getting the job
done.
```

Prompt 5: check my setup is healthy

Once things are installed, use this to have Claude confirm everything is working and explain what it finds.

```
Walk me through checking my Claude Code setup is healthy. Run, or tell
me to run, /doctor and /status, and explain the output in plain English.
Then show me how to check my connected tools (/mcp), my plugins
(/plugins), and my plan usage (/usage). For each one, say what a healthy
result looks like and what to do if something is missing or signed out.
```

Why learning this way works

You remember what you take part in. Reading a manual end to end is slow and easy to forget; building your own setup with a guide beside you is quick and sticks. You also finish the session knowing why each piece is there, which is what lets you change it later with confidence.

Keep the prompts you like. The fuller set, grouped by when you would reach for each, is in Appendix D, and you can download the whole pack as a file from the downloads folder (Appendix E) to keep open beside Claude.

A last word on safety. These prompts ask Claude to confirm before it writes or changes anything, but the responsibility is still yours. Read what it proposes, and only approve what you understand. That habit, more than any single setting, is what keeps you in control.

8. Claude Cowork: your desktop agent

Claude Cowork is Anthropic's desktop agent for knowledge work. It is available to anyone on a paid plan (Pro, Max, Team, Enterprise, or a Console account), on macOS and Windows (Windows support arrived in February 2026). On Windows it needs the Pro, Enterprise, or Education edition, because it relies on a feature called Hyper-V; Windows Home cannot run it. It is built for people who work with documents, data and files rather than code: researchers, analysts, and people in operations, finance and legal roles.

The difference from Claude Chat (Chapter 1) is short. Chat answers your questions; Cowork does the task. You describe a multi-step job, Claude plans it, executes it across your files and connected apps, and you steer as it goes.

Getting started

Cowork lives inside the Claude desktop app you installed in Chapter 4. To begin:

1. Open the Claude desktop app and sign in.
2. Start a Cowork session.
3. Connect a folder that Claude is allowed to read and write in.

On that last point, take care. Do not connect your whole drive. Create a dedicated working folder first (for example, a single folder for the project you want help with) and connect only that. Claude can only see and touch what you give it, so a narrow grant keeps everything else off limits.

Adding tools

To let Cowork reach beyond your files and into apps like email, calendars or your store, add a connector:

1. Click the "+" button at the bottom of the chat box.
2. Choose "Connectors".
3. Sign in to the service when prompted.

Connectors are covered in more depth in Chapter 11. For now, the rule is the same as for folders: only connect what the task needs.

Beyond files and connectors: computer use

Cowork can also drive applications on your computer directly, not just your files and connectors. This is more powerful and carries more risk, so Claude asks before it opens each app, and some

sensitive categories (banking, healthcare, government, trading and crypto apps) are blocked or best avoided. There are two modes: "Ask before acting", which pauses for your approval at each step and is the one to use, and "Act without asking", which is faster but only worth it for short, supervised jobs on apps you trust.

The permission model

Every tool, whether it comes from a connector or an MCP server, has one of three permission stances. You set these so Claude knows what it may do on its own and what it must check first.

Stance	What it means	When to use it
Allow	Runs without asking	Safe, read-only actions you trust, such as searching or reading
Ask	Asks you each time before running	Anything that changes, sends or deletes data; the safe default when unsure
Blocked	Cannot run at all	Tools you do not want used in this session

In the Cowork interface the first stance may be labelled "Always Allow". Tools are grouped into read-only and write-or-delete sets. On Team and Enterprise plans these stances are set for the whole organisation, so you cannot change them yourself; on a personal plan you control them.

The guiding rule: anything that changes or deletes data should usually be set to Ask, so you see the action before it happens.

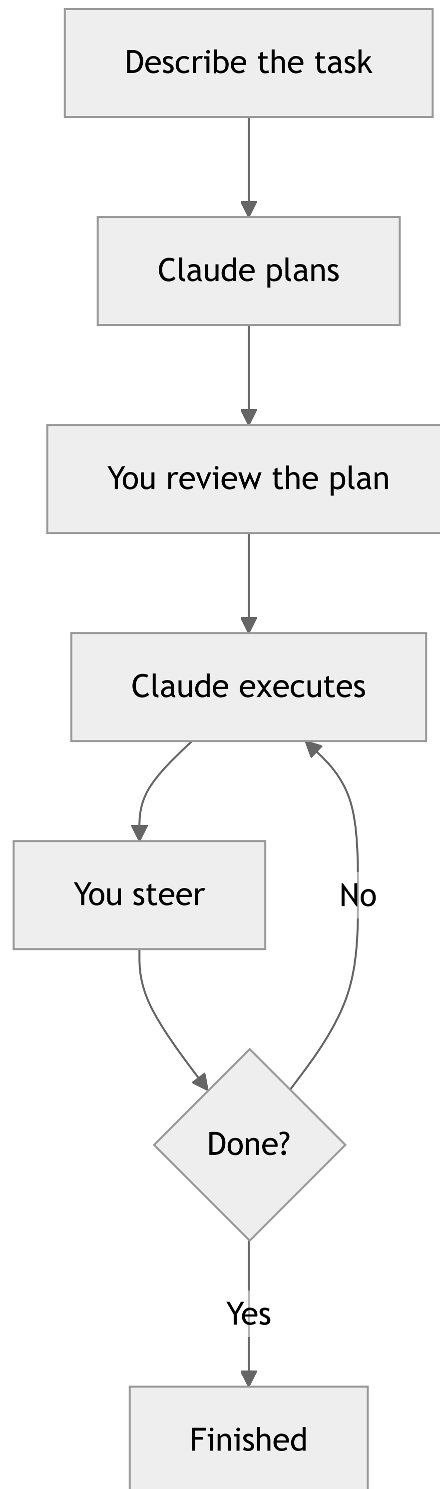
Working safely

Claude can read, write and permanently delete files in the folders you connect to it. That power is the point, but it means you should:

1. Keep a dedicated working folder and connect only that, not your whole drive.
2. Review Claude's planned actions before allowing them, especially for sensitive or hard-to-replace files.
3. Connect only the accounts and tools you actually need for the task.

How a session flows

A Cowork session is a loop. You stay in control at the review and steer points, so nothing important happens without you seeing it.



Example tasks

A few jobs that fit Cowork well:

1. Tidy and rename a folder of documents so the names are consistent and the contents are easy to find.
2. Pull figures from several spreadsheets into one summary.

3. Research a topic and draft a report from what it finds.

In each case you describe the outcome, check the plan, and let Claude do the legwork while you steer.

Learn more

Anthropic runs a free Cowork course at https://anthropic.skilljar.com. It is the fastest way to get comfortable with the loop above. For the safe-use guidance in full, see the Help Center at https://support.claude.com, which covers getting started and using Cowork safely.

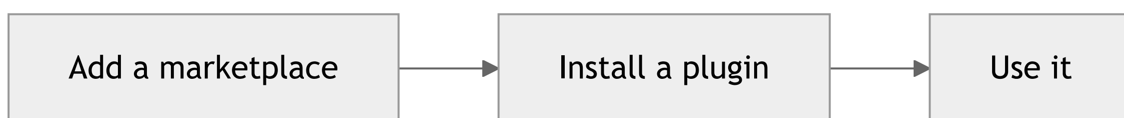
9. Plugins and marketplaces

A plugin is a bundle of extras for Claude Code, installed in one command. A single plugin can carry skills, subagents, slash commands, hooks, and sometimes MCP servers (the small servers that let Claude talk to outside tools, covered in chapter 11). Rather than wiring those up one at a time, you install the bundle and everything inside it becomes available at once.

A marketplace is a catalogue of plugins. Most marketplaces are simply a GitHub repository that lists what is on offer. One marketplace, the official `claude-plugins-official`, is built in and available immediately, so you can install from it without adding anything first.

The two-step flow

Installing a plugin is at most two steps. If the plugin lives in the built-in official marketplace, you skip straight to the install step. If it lives in another marketplace, you add that marketplace first, then install.



The commands, typed inside Claude Code:

1. Browse what is available. Type `/plugin`, then open the Discover view.
2. Add a marketplace (only needed for non-official ones). The pattern is the GitHub owner and repo name:

```
/plugin marketplace add owner/repo
```

3. Install a plugin. The pattern is the plugin name, then `@`, then the marketplace it comes from:

```
/plugin install name@marketplace
```

Once installed, you use a plugin's pieces like any other. A skill from a plugin is invoked by typing its name as a slash command, or simply by asking for the thing it does.

The four plugins in this reference setup

These four are the plugins set up as a reference. Use the install commands exactly as written.

Plugin	What it does	Where to get it	Install command
superpowers	A software-development methodology: skills for brainstorming, writing plans, test-driven development, systematic debugging, code review, subagent-driven development, and git worktrees.	Official marketplace (project: github.com/obra/superpowers)	`/plugin install superpowers@claude-plugins-official`
ecc (Everything Claude Code)	A large toolkit: 60+ specialist subagents (language reviewers, planners, build-fixers), 200+ skills across languages, frameworks and operations, hooks, and bundled MCP servers (GitHub, Context7 docs, Exa search, memory, Playwright, sequential-thinking).	GitHub marketplace github.com/affaan-m/everything-claude-code (site: ecc.tools)	`/plugin marketplace add affaan-m/everything-claude-code` then `/plugin install ecc@ecc`
imessage	macOS only. Lets Claude read your iMessage history and send replies through Messages.app.	Official marketplace	`/plugin install imessage@claude-plugins-official`

Plugin	What it does	Where to get it	Install command
	Needs macOS Full Disk Access.		
swift-lsp	The Swift language server (SourceKit-LSP) for Swift code intelligence.	Official marketplace	`/plugin install swift-lsp@claude-plugins-official`

Note that `ecc` is the only one of the four that needs a marketplace added first; the other three come from the built-in official marketplace.

A note on Adobe: "Adobe for Creativity" is not a Claude Code plugin. It is a connector, switched on inside the app rather than installed from a marketplace. Connectors are covered in chapter 11.

Which of these should a client install

For most people, two of the four carry their weight straight away:

- **superpowers** and **ecc** are the high-value general ones. superpowers gives a sensible way of working through any build (plan, test, debug, review). ecc adds a deep bench of specialist subagents and skills. Between them they cover the bulk of day-to-day work.
- **imessage** and **swift-lsp** are situational. Install imessage only if you want Claude working with your Mac messages, and only on a Mac. Install swift-lsp only if you are writing Swift code.

A tip on getting started

Install a couple to begin with, not everything at once. Each plugin adds new skills, commands and behaviour, and it is easier to learn what they do when you add them in small numbers. Start with superpowers and ecc, get comfortable, then add the situational ones if and when you actually need them.

10. Skills and subagents

Skills and subagents are two ways to package up work so Claude does it the same way every time and keeps your main conversation tidy.

A **skill** is a set of packaged instructions Claude follows for a specific task. Think of it as a recipe card: when you reach for the skill, Claude does that job the way the card says, every time.

A **subagent** is a separate Claude that runs one focused job and reports back. The main Claude hands the job over, the subagent goes off and does it, then returns a short answer. The benefit is that all the noise of the side job (the searching, the reading, the trial and error) stays out of your main conversation, so your main chat stays clear and on-topic.

Where skills come from

There are two main sources, plus a third route covered later in this chapter.

1. **Plugins you install.** When you install a plugin (see Chapter 9), it brings its skills with it. For example, the `superpowers` plugin adds skills like `superpowers:brainstorming`, and the `ecc` plugin adds skills like `ecc:deep-research`.
2. **Your own project.** Inside a project you can write custom skills in a `.claude/skills/` folder. These belong to that project and do exactly what you need, in your own words.

How to use a skill

Two ways, and both work:

1. Type `/` followed by the skill name, for example `/brainstorming`.
2. Just ask for the thing the skill does, in plain English. Claude recognises the request and reaches for the right skill on its own.

You do not have to memorise skill names. Asking plainly is usually enough.

The Printing Press family

Printing Press is a separate system of global skills and command-line tools. It is installed through its own system, not the plugin marketplace. The skills are named `printing-press` and `pp-something`, for example `pp-shopify`, `pp-klaviyo`, `pp-stripe` and `pp-figma`. Each one generates and runs a small command-line tool for an outside service.

One thing to know: each Printing Press CLI needs its own API key before it works. The key is how the tool signs in to that service (your Shopify store, your Klaviyo account, and so on). Without it, the tool is installed but cannot do anything.

Adding a skill from outside the marketplaces: HyperFrames

Skills do not only come from plugins. Some are published on their own and added with a single terminal command, using the pattern `npx skills add owner/repo`. A good example is **HyperFrames**, an open-source tool from HeyGen that turns HTML into animated video. It is what made the short explainer animations in this booklet. With it, you describe a motion graphic in plain language, preview it in a browser, and render it to a video file, all driven from Claude Code.

Add its skills with one command:

```
npx skills add heygen-com/hyperframes --all
```

Rendering needs Node.js (version 22 or newer) and FFmpeg installed on your machine. Once the skills are added, Claude Code picks them up automatically, and you drive the tool with `npx hyperframes` commands such as `init`, `preview` and `render`. It is an advanced, optional tool: reach for it when you actually want video, not as part of a basic setup. The project lives at https://github.com/heygen-com/hyperframes.

Representative skills and where they come from

Skill	Source	What it does
`superpowers:brainstorming`	superpowers plugin	Explores an idea with you before any work starts
`ecc:deep-research`	ecc plugin	Runs a multi-source research job and writes it up
`ecc:security-scan`	ecc plugin	Checks code for common security problems
`pp-shopify`	Printing Press	Runs Shopify store operations from the command line
`pp-klaviyo`	Printing Press	Works with a Klaviyo email account from the command line
`jcd-imagery`	Custom project skill	A house image-generation prompt scaffold
`icons`	Custom project skill	Generates a matching icon set

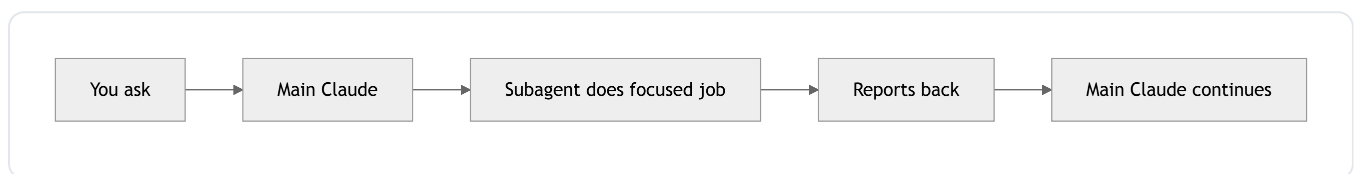
The last two are examples of skills written inside one project for that project's own needs. Yours would be different and named for your own work.

Example custom subagents

Subagents are just as buildable. These are examples of ones written for a single project, to show the shape of the idea:

Subagent	Purpose
`shopify-expert`	Answers store and theme questions, then reports back
`voice-reviewer`	Checks a piece of writing against your house tone
`research-aggregator`	Gathers and condenses findings from several sources

How a subagent fits in



The handover is invisible to you. You ask once, and you get one clean answer back.

Tip

If you find yourself asking Claude for the same kind of task again and again, ask it to write you a custom skill for it. Once that skill exists in your project, the task is one short command away and is done the same way every time.

11. MCP servers and connectors

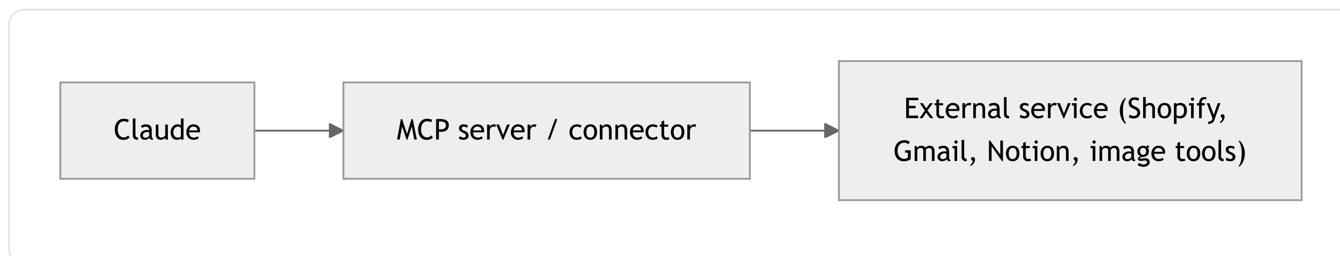
Claude is good at reasoning and writing on its own, but most real work involves outside tools: your shop, your inbox, your calendar, your files, your design software. MCP is how Claude reaches those tools safely. This chapter explains what MCP is, the two ways you add it, and the rules that keep your accounts safe.

What MCP is, in plain English

MCP stands for Model Context Protocol. It is an open standard that lets Claude talk to outside tools and services through small programs called "servers". Each server is a translator: it speaks Claude's language on one side and a specific service's language on the other. When you ask Claude to "list my recent orders" or "search my Drive", an MCP server is what actually does the fetching.

You do not need to understand the standard itself to use it. The full specification lives at https://modelcontextprotocol.io if you ever want the detail.

A connector is the easy version of the same thing: a ready-made MCP integration that you switch on inside the Claude app with a sign-in. No setup, no command line.



Two ways to add tools

There are two routes, and which one you use depends on whether you are working in the desktop or web app, or in Claude Code.

Route A: Connectors in the app. In the desktop or web app, go to Settings, then Connectors. In Cowork, you can also use the "+" button at the bottom of the chat box, then choose Connectors. You pick a service, sign in once with OAuth (the standard "Sign in with..." flow), and it is connected. No code, no keys to copy. Examples available include Shopify, Gmail, Google Calendar, Google Drive, Microsoft 365, Notion, and Adobe for Creativity, with more in the directory.

Route B: CLI-added MCP servers for Claude Code. In the terminal you add servers with the `claude mcp add` command. There are two kinds:

- **stdio**: a local command that runs on your own computer, often started through `npx`. Good for developer tools.
- **http**: a server reached at a web address (a URL).

The reference setup uses a few of each. Playwright is a stdio server for browser automation (it lets Claude open and drive a web browser). Flora is an http server for image generation, at the address `https://agents.flora.ai/mcp`.

Here is one concrete add command, the Shopify developer tools:

```
claude mcp add --transport stdio shopify-dev-mcp -- npx -y @shopify/dev-mcp@latest
```

The part after `--` is the actual command Claude Code runs to start the server. You do not need to memorise it; copy it as shown.

Which route is for you

	Connector route	**CLI route**
Who it is for	Everyone, especially non-developers	Claude Code users and developers
Where you add it	App: Settings, then Connectors (or "+" in Cowork)	Terminal: `claude mcp add ...`
Sign-in type	One-click OAuth ("Sign in with...")	Depends on the server; often an API key you provide

If a service offers both a connector and a CLI server, the connector is almost always the simpler choice. Reach for the CLI route only when you are working inside Claude Code or the tool you need is not available as a connector.

Safety: these act on your real accounts

> **Read this before connecting anything.**
>
> MCP servers and connectors can act on real accounts and real data. A connected inbox can send mail. A connected shop can change prices. Treat them with the same care you would a member of staff with the keys.
>
> - Only connect what you actually need. Do not switch on a connector "just in case".
> - Keep API keys private. A key is like a password for a service.
> - Never paste a key, password, or secret into a shared document, a chat with other people, or anywhere it might be saved or seen.
> - Prefer the "Ask" permission stance for anything that changes or deletes data, so Claude checks with you before acting. Save "Allow" for read-only, low-risk tools.
> - Review Claude's planned actions before approving them, especially anything that writes, sends, or removes.

You set these permission stances per tool. Chapter 8 covers how Allow, Ask, and Blocked work in practice.

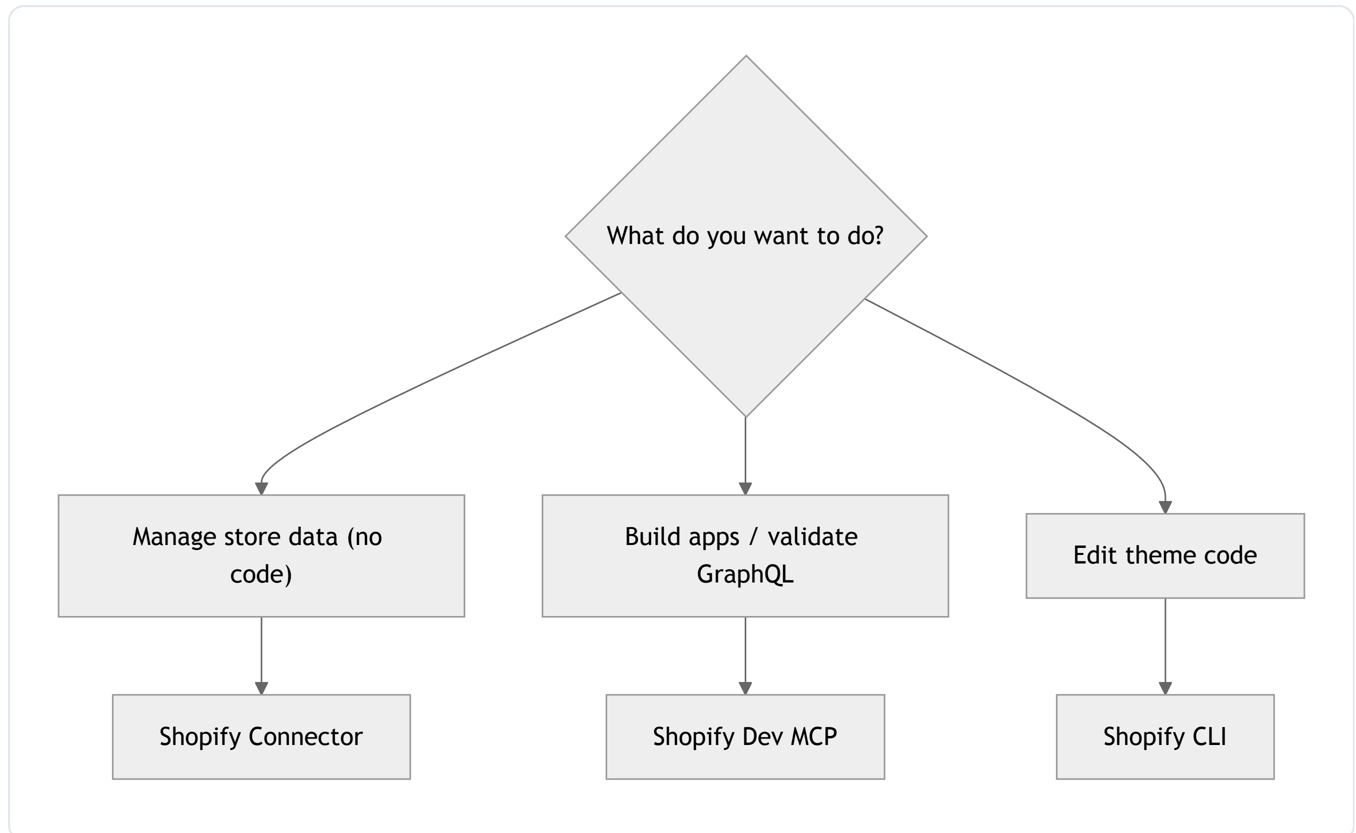
Where to go next

Shopify is the most common first connection for shop owners, and it has both a no-code connector and a developer route. Chapter 12 walks through it step by step, including which route to choose and what you can ask for in plain English.

12. Connect Claude to Shopify

If you run a Shopify store, Claude can help you manage it in two quite different ways. One way needs no code at all and covers everyday shopkeeping: orders, products, collections, inventory, reports and discounts. The other way is for developers and agencies who build apps or edit the store's theme code. Most people only ever need the first.

Start by deciding which job you are doing.



Route A: the Shopify connector (no code)

This is the fastest path and what most people want. A connector is a ready-made integration you switch on inside the Claude app, with a one-click, secure sign-in.

1. Open the Claude desktop or web app.
2. Go to Settings, then Connectors.
3. Find Shopify and add it.
4. Sign in to your store when prompted (this uses OAuth, so you authorise Claude without handing over any passwords or keys).

Once it is connected, you ask for what you want in plain English. There is no syntax to learn. For example:

- "List the last 7 days of orders."
- "Search products for the cropped hoodie."
- "Update the summer collection to include these three items."
- "Check inventory for the navy cap."
- "Run a sales report for last month."
- "Create a discount code for 10 percent off."

Claude reads the request, talks to your store through the connector, and reports back. For anything that changes or deletes data, review what it proposes before you let it run.

Route B: the Shopify Dev MCP for Claude Code

If you build apps or write store integrations, add the Shopify Dev MCP (part of Shopify's AI Toolkit) to Claude Code. Run this in your terminal:

```
claude mcp add --transport stdio shopify-dev-mcp -- npx -y @shopify/dev-mcp@latest
```

This gives Claude Code documentation search, GraphQL validation (so it can check queries before you run them), and store operations. It sits alongside Route A rather than replacing it: the connector is for plain-English store management, the Dev MCP is for development work.

Editing theme code with the Shopify CLI

To change your store's look, you edit Liquid templates, and that work uses the Shopify command-line tool rather than a connector.

1. Install the CLI (this needs Node.js 18 or newer):

```
npm install -g @shopify/cli@latest
```

2. Sign in to your store:

```
shopify auth login
```

3. Preview your theme locally as you work:

```
shopify theme dev
```

4. Download the current theme files:

```
shopify theme pull
```

5. Upload your changes:

```
shopify theme push
```

Three flags are worth knowing on `theme push`:

- `--theme <id>` targets a specific theme by its id, so you push to the right one.
- `--only <files>` limits the upload to named files instead of the whole theme.
- `--allow-live` is required to push to the published, live theme. Without it, the live store is protected.

Strongly recommended: do all your work on an unpublished development theme. Preview it, satisfy yourself it is right, and only push to live when you are ready. This keeps a broken edit away from real customers.

A small worked example

Using the connector (Route A), you could ask:

```
Show me last week's orders and top products.
```

Claude returns the orders from the past seven days and which products sold best, with no commands typed by you.

Separately, using the CLI, you push one section to a development theme rather than the live store:

```
shopify theme push --theme 123456789 --only sections/featured-collection.liquid
```

Replace the id with your development theme's id. Because there is no `--allow-live` flag, this cannot touch the published store.

Tips

- Keep tokens and keys private. Connectors sign you in without exposing them; the CLI manages its own login. Do not paste credentials into chats or files.
- Test on a development theme first. Push to live only when you have previewed the result and are happy.

- Before pushing anything to live, ask Claude to explain the change in plain English. If you do not understand the explanation, do not push it yet.

For the official reference, see the Shopify AI Toolkit documentation at https://shopify.dev/docs/apps/build/ai-toolkit.

13. Managing your chats: context, tokens, and slash commands

Once you are working with Claude day to day, one skill quietly separates a smooth session from a frustrating one: managing the conversation itself. This chapter explains how Claude holds a chat in mind, why long conversations get slower and pricier, and the handful of slash commands that keep things tidy. None of it is technical. Ten minutes here saves a lot of friction later.

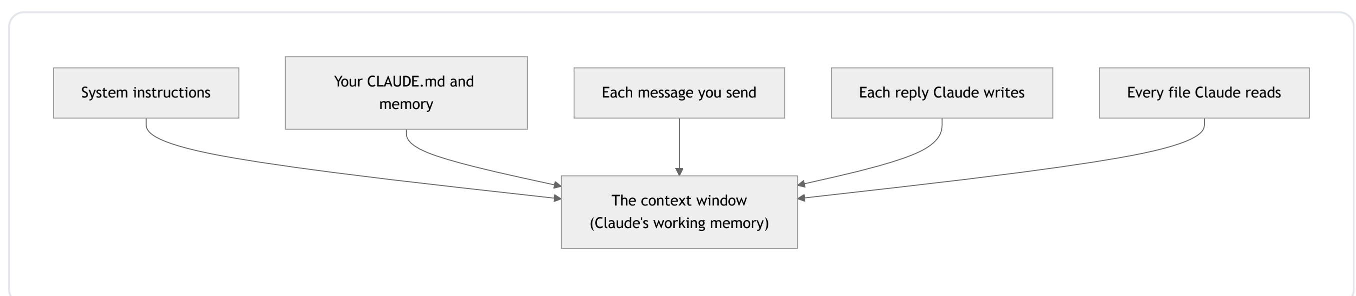
Tokens: how Claude measures text

Everything you and Claude exchange is measured in tokens. A token is a small chunk of text, very roughly three to four characters, or about three quarters of a word. A short sentence is a dozen or so tokens; a long file can run to tens of thousands. You never count them yourself, but it helps to know the unit, because everything below is measured in it.

The context window: Claude's working memory

The context window is everything Claude can hold in mind at once, measured in tokens. It is not the same as memory across sessions (that is CLAUDE.md and auto-memory, from Chapter 6); the context window is the live conversation in front of you right now.

The important thing to understand is that it is filling up before you even type. The moment a session opens it already holds the hidden system instructions, your CLAUDE.md, any saved memory, and the names of the tools available. Then every message you send, every reply Claude writes, and every file it reads piles on top.



In Claude Code the window holds about 200,000 tokens by default. Some models offer a much larger one (around a million tokens) on certain plans. Either way the number is large but not endless, and how full it is matters more than you might think.

Why long conversations go downhill

As the window fills, two things happen. First, quality slips: with a huge history to hold, Claude finds it harder to keep track of what matters, so it gets slower and can lose the thread. Second,

it costs more: each time you send a message, the whole conversation so far goes with it, so a long chat uses up more of your plan's allowance than a short, focused one. Anthropic put it plainly: longer conversations consume more of your usage.

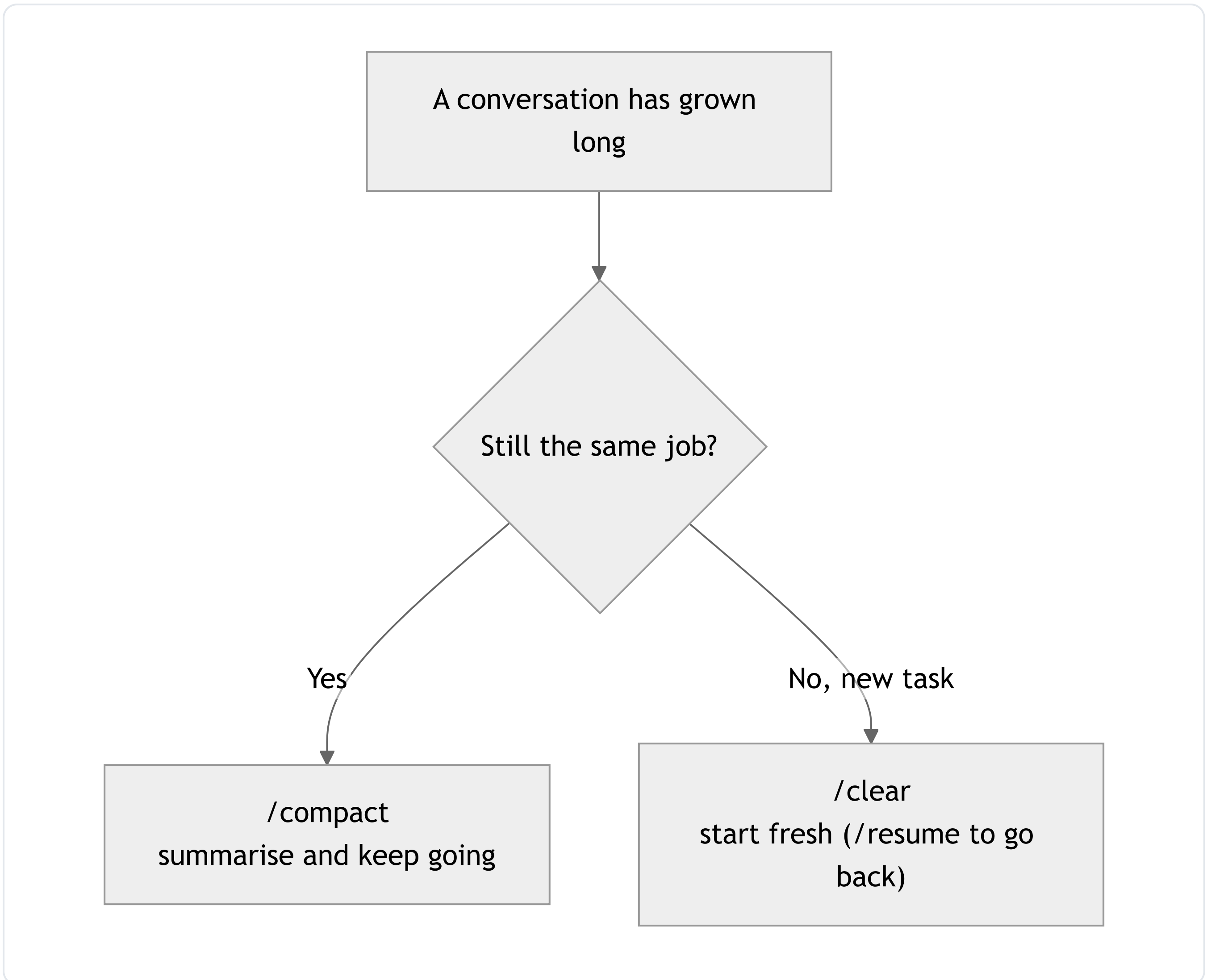
The lesson is simple. Keep a conversation to one job. When that job is done, or when a chat has wandered and grown long, reset it. The next two commands are how.

/compact and /clear: the two you will use most

These are slash commands: you type them straight into the Claude prompt, beginning with a forward slash.

- **`/compact`** summarises the conversation so far and replaces the long history with that summary, freeing up room while keeping the thread. You stay on the same job; Claude simply carries a lighter version of the story. Use it when a conversation has grown long but you are not finished. You can even point it: `/compact focus on the about-page rewrite` keeps the summary tight on what matters.
- **`/clear`** wipes the conversation and starts fresh, with an empty window. Use it when you move to a genuinely different task. The old conversation is not lost; `/resume` brings it back if you need it.

The rule of thumb: same job, lighten it with `/compact`; new job, start clean with `/clear`.



You will not always have to remember. Claude Code watches the window and compacts automatically as it nears full, so a long session does not simply stop. Running `/compact` yourself just does it sooner, and on your terms.

Seeing where it all goes: /context and /usage

Two commands let you look under the bonnet.

- **`/context`** shows a visual map of what is filling the window right now, broken down by part: the system instructions, your CLAUDE.md, the files read, Claude's replies, and so on. It is the quickest way to see what is taking up room.
- **`/usage`** shows how much of your plan's allowance you have used, and the session's cost. Reach for it if you are wondering how close you are to a limit.

How usage limits work

Your plan's allowance runs on two clocks: a five-hour window that opens with your first message of a session, and a longer weekly cap. Both are shared across Claude chat and Claude Code, so

heavy use in one leaves less for the other. The paid tiers differ by how much room they give: Pro is the baseline, and the Max plans give five or twenty times as much.

You do not need to watch this nervously. The point is only that long, sprawling sessions cost more of your allowance than short focused ones, which is one more reason to compact or clear when a job is done. Check `/usage` any time, or open Settings, then Usage, in the Claude app.

The slash commands worth knowing

Type `/` on its own to see everything available, including commands from any plugins you have installed. The exact list varies a little by plan, platform, and version. These are the ones a beginner gets the most from.

Command	What it does
`/help`	Lists the available commands
`/clear`	Wipes the chat and starts fresh (old chat kept; reopen with `/resume`)
`/compact`	Summarises and compresses the current chat, in place
`/context`	Shows what is filling the context window
`/usage`	Shows your plan usage and the session cost
`/resume`	Returns you to an earlier conversation
`/rewind`	Rolls back to an earlier point if Claude went the wrong way
`/model`	Switches the AI model
`/config`	Opens settings (theme, model, preferences)
`/init`	Creates a starter CLAUDE.md for a new project
`/status`	Shows your version, model, and sign-in
`/doctor`	Checks your installation for problems

A habit that keeps sessions healthy

When you finish a piece of work, ask Claude for a short recap of what you did and what is next (there is a ready-made prompt for this in Appendix D), then `/clear` to start the next job clean, or `/compact` to carry on lighter on the same one. That single habit keeps every session fast, focused, and easy on your allowance.

14. Tips and tricks

This chapter collects habits that make Claude more useful day to day. None of them are complicated. Together they are the difference between fighting the tool and working with it. Try a few, keep the ones that fit how you work.

Working well with Claude

Ask for a plan first. For anything beyond a one-line request, ask Claude to plan before it acts, then read the plan. Claude Code and Cowork can both lay out the steps they intend to take. A plan is cheap to read and cheap to correct; undoing the wrong files is not. This is also how a Cowork session is meant to run: you describe the task, Claude plans, you review, Claude executes, you steer.

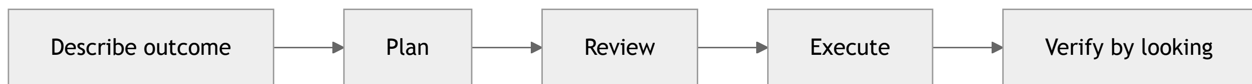
Be specific. Say the outcome you want, the constraints that matter, and where the files are. "Rewrite the about page so it is shorter and warmer, keep the headings, the file is in 03 web" gives Claude what it needs. "Make it better" does not. The more precisely you describe the finish line, the closer the first attempt lands.

Dictate instead of typing. Speaking is much faster than typing, and you tend to give Claude more detail when you talk than when you type. A dictation tool runs quietly in the background; you press a hotkey, speak, and your words appear wherever your cursor is, including the Claude prompt box, cleaned up with punctuation and capitals. Two are worth knowing. Wispr Flow works on Mac, Windows, iPhone and Android, and polishes what you say as you go, so it is the easy choice if you are on a PC or move between devices. Superwhisper is Mac-first, also on Windows and iPhone, and runs entirely on your own machine, so your audio never leaves your computer, which suits anything private. Both have a free tier to try. For the longer prompts in Chapter 7 and Appendix D, this turns a minute of typing into a few seconds of talking.

Use a CLAUDE.md. A short markdown file at the root of a project tells Claude who you are, how you want it to work, and your house rules every time it opens that folder. It saves you re-explaining yourself. See chapter 6 for how to write one.

Let it use tools. Claude can search the web, look up documentation, and reach real services through MCP connectors. A lookup beats a guess. If you want current facts, ask it to search; if a service is connected, ask it to check the live data rather than estimate. Connected tools (web search, docs lookup, MCP connectors) are almost always better than the model working from memory alone.

A healthy task runs as a short loop. Describe what you want, let Claude plan, review the plan, let it execute, then confirm the result with your own eyes.



Staying in control

Keep the context clean. When a conversation has wandered across several jobs it gets cluttered, and clutter makes Claude slower and less accurate. Use `/clear` to start fresh on an unrelated task, or `/compact` to keep going on the same job with a lighter history; Chapter 13 explains both. For a big side-quest inside a larger job, ask Claude to use a subagent: a separate Claude that runs one focused task and reports back, which keeps your main conversation tidy.

Find skills and commands by typing a slash. In Claude Code, type `/` (or `/help`) to see the skills and slash commands available to you, including ones from plugins you have installed and your own project skills. The handful worth learning first, `/clear`, `/compact`, `/context` and `/usage`, get their own chapter (Chapter 13). The exact list varies a little by plan, platform and version.

Match effort to the problem. Use a higher effort level for genuinely hard problems where you want Claude to think carefully, and a faster mode for simple, low-stakes work; you switch model and effort with `/model` and `/config`. Spending more effort on a tricky problem usually saves you time fixing a rushed answer, but higher effort and long sessions also use up more of your plan allowance, so save the heavy settings for when they earn it (see Chapter 13).

Set permissions deliberately. You can pre-approve safe, repeated commands so Claude stops asking each time, which removes friction from routine work. Keep approvals tight for anything that changes or deletes data. For a tool from a connector or MCP server, the stance "Ask" (prompts each time) is the safe default for anything that writes; reserve "Allow" for actions that only read. Review planned actions before allowing them, especially on sensitive files.

Save what works. When Claude gets something right, lock it in. Ask it to remember a preference so it applies next time, or to write a reusable skill in your project's `.claude/skills/` folder for a task you repeat. A preference written to your CLAUDE.md or saved as memory is reloaded from disk every session, so it survives even when you clear or compact a conversation, unlike anything that lived only in the chat. The second time is then faster than the first.

When something breaks

Verify by looking. Do not trust "done" on its own. Ask Claude to run the thing and show you the output, or to take a screenshot of the web page, rather than reporting success without evidence. Seeing the result is the only reliable check.

Use git. Keep your work in a git repository so every change is recorded and reversible. If an edit goes wrong, you can return to the last good state instead of trying to remember what changed. This single habit removes most of the fear from letting Claude edit files.

Run a health check. When Claude Code is behaving oddly, run the built-in check.

```
claude doctor
```

It inspects your installation and reports problems, which is the fastest first move before you dig into anything more involved.

15. Troubleshooting

This chapter is a reference. Find the symptom, apply the fix. Most problems at this stage are small: a terminal that needs reopening, a sign-in that has expired, or a folder that was never connected. Work through the relevant entry, and if it persists, the last section points you to deeper help.

Install and terminal problems

"claude: command not found" after installing Claude Code. The install worked, but your terminal has not picked up the new command yet.

1. Close the terminal window completely and open a fresh one. This reloads the PATH (the list of places your computer looks for commands).
2. Run `claude --version` again.
3. On Mac, if it still fails, re-run the installer (`curl -fsSL https://claude.ai/install.sh | bash`), then open a new terminal.
4. Check you are in the terminal you installed into. If you installed in Terminal but are typing in another app, switch back.

"irm is not recognised" on Windows. You are in Command Prompt (CMD), but `irm` is a PowerShell command. Open PowerShell and run the PowerShell install line (`irm https://claude.ai/install.ps1 | iex`). Alternatively, use the CMD install line instead.

"&&" gives an error on Windows. The opposite of the above: you are in PowerShell, but the `&&` CMD install line was written for Command Prompt. Open Command Prompt (CMD) and run it there, or use the PowerShell line in PowerShell.

Node and npx errors

If you installed via npm, or a tool uses `npx`, and you see Node-related errors, your Node.js is missing or too old. Claude Code and several MCP servers need Node.js 18 or newer. Install or update Node.js, reopen the terminal, and try again. Where possible, prefer the native installer over npm to avoid this entirely.

Sign-in and connector problems

Symptom	Fix
A plugin or MCP server says it needs authentication	Run its sign-in step again, or re-connect the connector. For CLI tools, check the API key is set and correct.
A connector stopped working	Re-authorise it. Go to Settings, then Connectors, and reconnect (sign in again).
Shopify CLI cannot log in	Run `shopify auth login` again. Confirm your account has access to that store.

API keys can expire or be revoked, and sign-in sessions time out. Re-authorising is normal maintenance, not a sign something is broken.

Permissions and behaviour

Claude keeps asking permission for the same safe command. Each tool from a connector or MCP server has a permission stance: Allow, Ask, or Blocked. If a command is safe and you run it often, change its stance to Allow (add it to your allowed list) so it runs without asking. Keep "Ask" for anything that changes or deletes real data.

Cowork cannot see a file. Cowork can only read and write inside folders you have connected. If a file is missing, its folder is not connected. Connect the folder (or move the file into a folder that is already connected), then ask again. Remember Claude can permanently delete files in connected folders, so connect only what you need.

Slow, stuck, or confused

If Claude becomes slow, repeats itself, or loses the thread, the conversation has usually grown too long. Run `/compact` to summarise and compress the history while staying on the same job, or `/clear` to start fresh on a new task; the old conversation is still reachable with `/resume`. Claude Code also compacts automatically as the window nears full, so a long session does not simply stop. For a deeper health check of your Claude Code setup, run `claude doctor`. Chapter 13 explains context and these commands in full.

Deeper help

For anything not covered here:

- Claude Help Center: https://support.claude.com
- Claude Code "Troubleshoot install" and setup docs: https://code.claude.com/docs

When asking for help, note your operating system, exactly what you typed, and the exact error text. That is usually enough to get a precise answer.

16. Resources and where to learn more

Keep this page to hand. Every link below points to an official source, an installer you may need, or a place to learn more. Each entry has a one-line note so you can find the right one quickly.

Official documentation and help

- Claude Code docs (code.claude.com/docs): the main reference for Claude Code, including the setup guide, the VS Code extension page, and plugins and marketplaces.
- Claude Help Center (support.claude.com): account, billing, and how-to articles, including the Cowork get-started and use-safely guides.
- Claude Cowork product page (claude.com): what Cowork is and who it is for.
- Claude Cowork on anthropic.com: the official overview from Anthropic.
- Cowork course, free (anthropic.skilljar.com): a short, free course that walks you through using Cowork.
- Model Context Protocol (modelcontextprotocol.io): the open standard behind connectors and MCP servers, for when you want to understand how tools plug in.

Plugins and tools

- Official plugin marketplace (github.com/anthropics/claude-plugins-official): the built-in marketplace; source for superpowers, imessage, and swift-lsp.
- ecc, Everything Claude Code (github.com/affaan-m/everything-claude-code): a large toolkit of subagents, skills, and bundled MCP servers; see also ecc.tools.
- superpowers (github.com/obra/superpowers): a software-development methodology in plugin form: brainstorming, planning, test-driven development, and review.
- Shopify AI Toolkit (shopify.dev/docs/apps/build/ai-toolkit): the developer reference for connecting Claude to a Shopify store.

Software you install

- Visual Studio Code (code.visualstudio.com): the free code editor that hosts the Claude Code extension.
- Homebrew (brew.sh): the package manager for macOS, used to install some tools by command.
- Git for Windows (git-scm.com/downloads/win): recommended on native Windows so Claude Code can use its Bash tool.
- GitHub (github.com): a free account is recommended for code projects and for some plugins.

Dictation and creative tools

- Wispr Flow (wisprflow.ai): voice dictation for Mac, Windows, iPhone and Android; speak your prompts instead of typing them.
- Superwhisper (superwhisper.com): on-device voice dictation, Mac-first (also Windows and iPhone); your audio stays on your computer.
- HyperFrames (github.com/heygen-com/hyperframes): an open-source tool from HeyGen for turning HTML into animated video from Claude Code. Install with `npx skills add heygen-com/hyperframes --all` (needs Node.js 22+ and FFmpeg).

Recommended viewing and community

We strongly recommend Nate Herk for clear, beginner-friendly walkthroughs of Claude Code, MCP servers, skills, and AI automation.

- Nate Herk, AI Automation (youtube.com/@nateherk): practical video walkthroughs; a good place to start watching.
- nateherk.com: his website, with more resources.
- His community, AI Automation Society, runs on Skool for people who want to go deeper.

When in doubt, start with the official docs and Help Center above. They are kept current, and they are the safest place to check a command or a setting before you run it.

Appendix A: A real setup, end to end (worked example)

The rest of this booklet stays neutral on purpose. This appendix is the one place we show a real, named setup, so you can see how the pieces fit together in practice rather than in the abstract. The example is the author's own studio, an independent fashion, jewellery and creative business that runs under the JCD brand, with a Shopify store at its centre. Nothing sensitive is shown here. Anywhere a secret would normally go, you will see a placeholder: the store is written as `your-store.myshopify.com`, API tokens as `<api-token>`, and the store password as `<store-password>`.

Read it as a story of one working setup, not a rule. The same shape works for almost any small business.

1. The project folder

Everything the studio does lives in one project folder, organised into numbered top-level folders so any piece of work has an obvious home:

```
00 brand        01 flagship project   02 products   03 web
04 email        05 applications       06 correspondence
07 print        08 social             09 strategy    10 finance
```

Each folder has a short README that says what belongs in it, so files do not drift to the wrong place. File names are clear and dated or versioned, which makes the latest version easy to find six months later.

2. The CLAUDE.md brain

At the root of the project sits a single `CLAUDE.md` file. Claude Code loads it automatically at the start of every session in that folder, so it never has to be re-explained. It holds:

1. Who the brand is, in a few honest lines.
2. The voice and the house rules (British English, no hype words, what never to do).
3. Links to the source-of-truth files (the brand bible, the style guide, the site brief) rather than copies of them.
4. What Claude's job is: a working creative director and operator, not a yes-machine.

It is kept short and updated whenever something material changes.

3. Plugins enabled

Two plugins are switched on at the toolkit level. ecc (Everything Claude Code) supplies the broad set of specialist subagents and skills for code and operations. superpowers supplies the working method: brainstorming, writing a plan, test-driven development, and systematic debugging, so the studio's coding work follows a discipline rather than improvising.

4. Custom skills and subagents

On top of the plugins, the studio has built its own skills in the project's `.claude/skills/` folder for jobs it does over and over: an image-generation prompt scaffold, an icon-set generator, and a set of web-theme helpers. Alongside them sit custom subagents, including a Shopify expert, a voice reviewer that checks copy against the house rules, and a research aggregator. Each is invoked by name or simply by asking for the thing it does.

5. Connectors and MCP

Three outside connections do the heavy lifting. The Shopify connector (switched on in the app, signed in with OAuth) gives plain-English access to live store data: orders, products, inventory, reports. An image-generation MCP server produces product visuals. A browser-automation MCP (Playwright) lets Claude open and check the live site after a change. Anything that writes or deletes is kept on "Ask" so nothing happens to real data without a look first.

6. The Shopify theme workflow

Theme code is never edited straight on the live site. The studio works in this order:

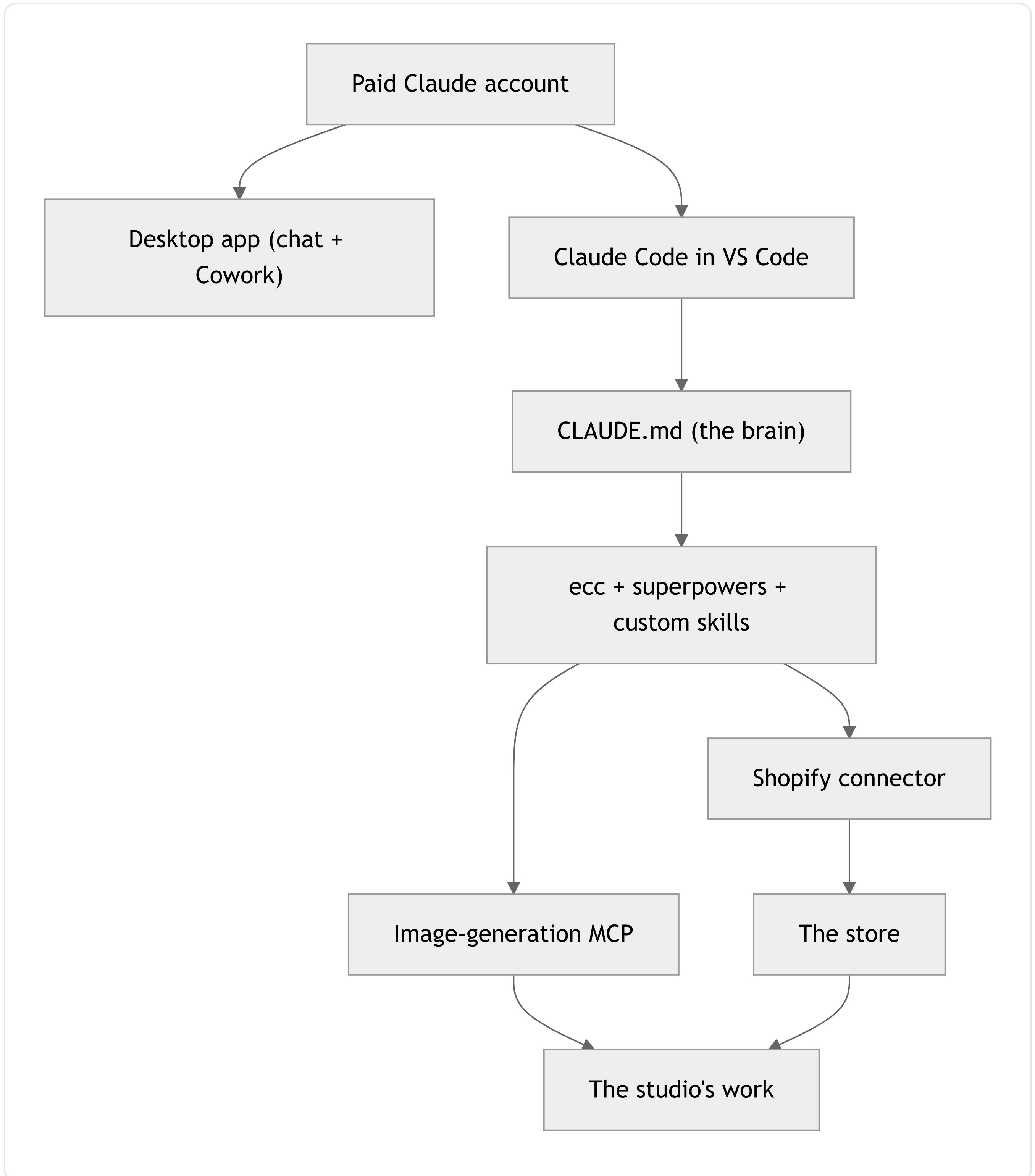
1. Edit a theme section locally with the Shopify CLI.
2. Preview it on an unpublished development theme with `shopify theme dev`.
3. Push to the live theme only when it is right, with `shopify theme push --allow-live`.

That way the public storefront only ever sees finished work.

7. Email

Email marketing runs through a connected email-marketing tool, so welcome flows, post-purchase notes and broadcasts are drafted, reviewed and sent from the same place as the rest of the studio's work, in the same voice.

The map, filled in



The same shape works for any small business: keep the folder structure, the CLAUDE.md brain and the working method, and swap the folder names and connectors for your own.

Appendix B: Quick-start checklist (print this page)

Print this page. Tick each box as you go. Pick the column for your machine, then do the shared section at the bottom.

Before you start

- [ ] You have a paid Claude plan (Pro, Max, Team, Enterprise, or Console)
- [ ] You have your Claude sign-in details to hand
- [ ] Your machine meets the requirements (macOS 13.0+ or Windows 10 1809+, 4 GB+ RAM, internet)
- [ ] A free GitHub account is set up if you plan to do code projects

Mac column

- [ ] Open Terminal (Applications, Utilities, Terminal)
- [ ] Install Homebrew (optional, the macOS package manager):

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
```

- [ ] Install Claude Code (recommended, auto-updates):

```
curl -fsSL https://claude.ai/install.sh | bash
```

- [ ] Verify the install:

```
claude --version
```

- [ ] Install the desktop app from https://claude.ai/download
- [ ] Install VS Code from https://code.visualstudio.com (optional)

Windows column

- [ ] Open Windows PowerShell (Start menu, type PowerShell)
- [ ] Install Git for Windows from https://git-scm.com/downloads/win
- [ ] Install Claude Code (recommended, auto-updates):

```
irm https://claude.ai/install.ps1 | iex
```

- [ ] Verify the install:

```
claude --version
```

- [ ] Install the desktop app from https://claude.ai/download
- [ ] Install VS Code from https://code.visualstudio.com (optional)

After install (do these on both Mac and Windows)

- [ ] Open the desktop app and sign in with your Claude account
- [ ] In your terminal, go to your project folder and run `claude`, then log in through the browser prompt
- [ ] Open your project folder in VS Code (optional)
- [ ] Create a `CLAUDE.md` file at the root of your project (your brief and house rules)
- [ ] Install the superpowers plugin:

```
/plugin install superpowers@claude-plugins-official
```

- [ ] Add the ecc marketplace, then install ecc:

```
/plugin marketplace add affaan-m/everything-claude-code
/plugin install ecc@ecc
```

- [ ] Connect Shopify if you need it (desktop app, Settings, Connectors, sign in)
- [ ] Run a final deeper check:

```
claude doctor
```

You are done when

`claude doctor` passes, the desktop app is signed in, and a `CLAUDE.md` file exists in your project.

Appendix C: Reference tables

This appendix collects the exact commands, names, and links from the rest of the booklet in one place. Use it as a quick lookup once you understand what each item does. Every value here is taken from the official sources current as of June 2026.

Install commands (Claude Code)

Platform / method	Command
macOS / Linux / WSL (native installer, recommended)	`curl -fsSL https://claude.ai/install.sh | bash`
Homebrew (macOS)	`brew install --cask claude-code`
Windows PowerShell (native installer)	`irm https://claude.ai/install.ps1 | iex`
Windows CMD	`curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del install.cmd`
WinGet (Windows)	`winget install Anthropic.ClaudeCode`
npm (needs Node.js 18+, no sudo)	`npm install -g @anthropic-ai/claude-code`
Verify the install	`claude --version`
Deeper health check	`claude doctor`

The Homebrew cask does not auto-update; upgrade it with `brew upgrade claude-code`. The native installers do auto-update.

Plugins (the four in this reference setup)

Name	What it does	Install command
superpowers	Software-development methodology: brainstorming, plans, test-driven development, debugging, code review, worktrees	`/plugin install superpowers@claude-plugins-official`
ecc (Everything Claude Code)	Large toolkit: 60+ subagents, 200+ skills, hooks, bundled MCP servers	`/plugin marketplace add affaan-m/everything-claude-code` then `/plugin install ecc@ecc`
imessage	macOS only: reads iMessage history and sends replies (needs Full Disk Access)	`/plugin install imessage@claude-plugins-official`
swift-lsp	Swift language server (SourceKit-LSP) for Swift code intelligence	`/plugin install swift-lsp@claude-plugins-official`

Key Claude Code commands

Action	Command
Browse and discover plugins	`/plugin`
Add a marketplace from a GitHub repo	`/plugin marketplace add owner/repo`
Install a plugin	`/plugin install name@marketplace`
Run a skill	`/skill-name`
Start Claude Code / sign in	`claude`
Show the installed version	`claude --version`
Run the diagnostic check	`claude doctor`
Add an MCP server	`claude mcp add <name> ...`

Session and context commands

Type these straight into the Claude prompt during a session. Availability varies a little by plan, platform, and version. Chapter 13 explains them in context.

Command	What it does
`/help`	Lists the available commands (or type `/` on its own)
`/clear`	Wipes the chat and starts fresh; the old chat is kept, reopen it with `/resume`
`/compact`	Summarises and compresses the current chat in place, keeping the thread
`/context`	Shows what is filling the context window
`/usage`	Shows your plan usage and the session cost
`/resume`	Returns you to an earlier conversation
`/rewind`	Rolls back to an earlier checkpoint
`/model`	Switches the AI model
`/config`	Opens settings (theme, model, preferences)
`/init`	Creates a starter CLAUDE.md for a new project
`/status`	Shows your version, model, and sign-in
`/doctor`	Checks your installation for problems

Connectors and MCP servers

Name	Type	Connects to
Shopify	Connector	Your Shopify store (orders, products, inventory, reports)
Gmail	Connector	Your Gmail mailbox
Google Drive	Connector	Your Google Drive files
Notion	Connector	Your Notion workspace
Microsoft 365	Connector	Microsoft 365 (mail, files, calendar)
Adobe for Creativity	Connector	Adobe creative tools
Playwright	stdio MCP (via npx)	A controllable web browser
Flora	http MCP (`https://agents.flora.ai/mcp`)	Image generation
Shopify Dev MCP	stdio MCP (via npx)	Shopify docs, GraphQL validation, store operations

Connectors are switched on in the app under Settings then Connectors (or the "+" in Cowork). MCP servers and connectors can act on real accounts; only connect what you need and prefer the "Ask" permission for anything that changes or deletes data.

Shopify commands

Action	Command
No-code route	Use the Shopify connector in the Claude app (Settings, Connectors, add Shopify, sign in), then ask in plain English
Developer MCP (Claude Code)	`claude mcp add --transport stdio shopify-dev-mcp -- npx -y @shopify/dev-mcp@latest`
Install the Shopify CLI	`npm install -g @shopify/cli@latest`
Sign in	`shopify auth login`
Local theme preview	`shopify theme dev`
Download theme files	`shopify theme pull`
Upload theme files	`shopify theme push`

When pushing a theme, `--theme <id>` targets a specific theme, `--only <files>` limits the upload, and `--allow-live` pushes to the published theme. Work on an unpublished development theme, preview, then push to live only when ready.

Companion tools (optional)

Tool	What it is	Where to get it
Wispr Flow	Voice dictation (Mac, Windows, iPhone, Android): speak your prompts instead of typing	wisprflow.ai
Superwhisper	On-device voice dictation (Mac-first, also Windows and iPhone): audio stays on your machine	superwhisper.com
HyperFrames	Turn HTML into animated video from Claude Code (needs Node.js 22+ and FFmpeg)	`npx skills add heygen-com/hyperframes --all`

These are optional extras, not part of a basic setup. Dictation makes long prompts far quicker; HyperFrames is for when you want to make video.

Official links

Resource	URL
Claude Code docs	https://code.claude.com/docs
Claude Help Center	https://support.claude.com
Claude Cowork	https://claude.com/product/cowork
Cowork course (free)	https://anthropic.skilljar.com
Model Context Protocol	https://modelcontextprotocol.io
Shopify AI Toolkit	https://shopify.dev/docs/apps/build/ai-toolkit
Official plugin marketplace	https://github.com/anthropics/claude-plugins-official
ecc plugin	https://github.com/affaan-m/everything-claude-code
superpowers plugin	https://github.com/obra/superpowers
VS Code	https://code.visualstudio.com
Homebrew	https://brew.sh
Git for Windows	https://git-scm.com/downloads/win
GitHub	https://github.com
Nate Herk (YouTube)	https://www.youtube.com/@nateherk
Nate Herk (site)	https://www.nateherk.com
Desktop app download	https://claude.ai/download
Wispr Flow (dictation)	https://wisprflow.ai
Superwhisper (dictation)	https://superwhisper.com
HyperFrames (HTML to video)	https://github.com/heygen-com/hyperframes

Appendix D: The prompt pack (copy and paste)

A library of prompts you can paste straight into Claude. They are starting points, not magic words. Change anything in square brackets to fit you, and reword the rest freely. Chapter 7 introduces the core setup prompts in context; this appendix is the fuller set, grouped by when you would reach for each.

On screen, every block has a Copy button. There is also a plain-text copy of this pack in the downloads folder (see Appendix E), so you can keep it open in a window beside Claude.

Getting set up (your first hour)

Set me up from scratch, and teach me as you go:

```
I have just installed Claude Code and I am brand new to it. I want you
to help me set up this project, and teach me what you are doing as we
go. Work in small steps and stop for me after each one. Start by
explaining what a CLAUDE.md file is, ask me a few short questions about
me and my work, and draft one from my answers. Do not create any files
until I have seen the draft and said go ahead.
```

Build my folder structure:

```
Help me set up a tidy folder structure for this project: numbered
top-level folders so they stay in order, and a short README in each.
Ask me what kind of work this covers, suggest folders, let me adjust
them, and only create the folders and READMEs once I confirm. Explain
the naming convention as you go.
```

Walk me through connecting a tool:

```
I want to connect [Shopify / Gmail / Google Drive / Notion] to Claude.
Walk me through it one step at a time, tell me what I am agreeing to
when I sign in, and once it is connected, suggest two safe, read-only
things I could ask for to check that it is working.
```

Check my setup is healthy:

```
Walk me through checking my Claude Code setup is healthy. Run, or tell
me to run, /doctor and /status, and explain the output in plain English.
Then show me how to check my connected tools (/mcp), my plugins
(/plugins), and my plan usage (/usage). For each one, say what a healthy
result looks like and what to do if something is missing or signed out.
```

Learning as you go

Teach me as we work:

```
While we work, briefly explain what you are doing and why, especially
anything I could reuse another time. If there is a simpler or more
standard way to do what I asked, tell me before you start. Treat this
as me learning the tool, not just getting the job done.
```

What can you actually do here:

```
Look at this project folder and tell me, in plain English, the kinds of
jobs you could help me with here. Give me five concrete examples based
on what you can actually see, not generic ones, and mark which are
quick wins.
```

Explain that like I am new:

```
Explain [the last thing you did / this term / this command] as if I am
new to all of this. Keep it short and plain, and tell me when it
matters in practice.
```

Build me a glossary:

```
Make me a short glossary of the words I keep seeing here, for example
CLAUDE.md, plugin, skill, subagent, MCP, connector, terminal. One
plain-English line each, in the order a beginner would meet them.
```

Everyday working

Start a task well, brief first:

```
Before you start, tell me back what you understand the goal to be, and
your plan in a few steps. Flag anything you are unsure about. Wait for
me to confirm before you do the work.
```

Plan first, then do:

```
Plan this before you change anything. List the steps, what each one
touches, and anything risky. Once I approve the plan, carry it out and
tell me what you actually did at the end.
```

Review my work:

```
Review [this file / this draft / what we just did] as a careful second
pair of eyes. Tell me what is strong in a line, then what is weak or
unclear with specifics, then the single most useful change to make.
```

Tidy this folder:

```
This folder has got messy. Suggest a tidier structure and a consistent
way of naming the files. Show me the before and after first. Do not
move or rename anything until I approve, and never delete anything
without asking.
```

Write it in my voice:

```
Write [the thing] in my voice. Match how I write in [point to a file,
or paste an example]: the same plainness, the same length of sentence,
the same words I would and would not use. No hype, no filler.
```

Keeping Claude sharp (memory)

Save this as memory:

```
That is worth remembering for next time. Save it as a short, clear note
so you recall it in future sessions, and tell me where you saved it.
```

Update my CLAUDE.md:

```
We have changed how this project works. Read my CLAUDE.md, tell me which
parts are now out of date, and propose edits. Show me the changes before
you save them.
```

End-of-session recap:

```
We are wrapping up. Give me a short recap: what we did, what is left,
and the first thing to pick up next time. If anything from today should
go into my CLAUDE.md or memory, say so.
```

Use this just before you `/clear` to start a fresh task, so nothing important is lost. To keep going on the same job with a lighter history instead, `/compact` does a similar summarise step in place (you can add a focus, for example `/compact focus on the about-page rewrite`). See Chapter 13.

Connecting your tools

Shopify, read-only to start:

```
My Shopify store is connected. To start, only read, do not change
anything. Show me my orders from the last seven days and my five
best-selling products, and tell me where the numbers came from.
```

Shopify, a careful change:

```
I want to [describe the change] in my Shopify store. Before you touch
anything, tell me exactly what you are about to change and show me the
current values. Make the change only after I confirm, then read it back
so I can see it worked.
```

Working safely

Ask before anything risky:

```
For the rest of this session, check with me before anything that sends,
publishes, changes or deletes data. Reading and drafting are fine to do
on your own. When you do want to make a change, show me what and where
first.
```

Dry run first:

```
Show me what this would do without doing it. List every file or setting
it would change and how. I will tell you which parts to actually carry
out.
```

If you find yourself pasting the same prompt often, ask Claude to turn it into a custom skill or a saved command for your project, so it is one short word next time. See Chapter 10 for how skills work.

Appendix E: Downloadable starter files

Alongside this booklet is a `downloads` folder with ready-made files, to save you typing. You can use these instead of, or alongside, the prompts in Chapter 7. The prompts teach you as you go; the files just get you there fast. Either route is fine, and you can mix them.

File	What it is	How to use it
`CLAUDE.md`	A starter brain for a project, with guidance and blanks to fill in.	Save it at the top of your project as `CLAUDE.md`, then replace the bracketed prompts with your own words.
`setup-claude-workspace.sh`	A Mac or Linux script that builds a numbered folder structure, a README in each folder, a starter `CLAUDE.md`, and a basic `.gitignore`.	See "Running the folder-setup script" below.
`setup-claude-workspace.ps1`	The same script for Windows PowerShell.	See "Running the folder-setup script" below.
`starter-prompts.md`	The whole prompt pack from Appendix D as a plain text file.	Open it in a window beside Claude and copy whichever prompt you need.

Running the folder-setup script

Two things to know first. The script only creates folders and starter files; it does not delete or overwrite anything you already have. And, as with anyone's script, open it and read it before you run it.

On a Mac or Linux, open Terminal and run:

```
cd /path/to/your/project
bash setup-claude-workspace.sh
```

On Windows, open PowerShell and run:

```
cd C:\path\to\your\project
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
.\setup-claude-workspace.ps1
```

The `Set-ExecutionPolicy` line lets the script run for this one PowerShell window only; it changes nothing permanently.

You get a set of numbered folders, a short README in each, a starter `CLAUDE.md` if you do not already have one, and a basic `.gitignore`. If you want different folder names, open the script and edit the short list near the top before you run it.

Or let Claude do it

If you would rather not run a script at all, paste Prompt 2 from Chapter 7 instead, and Claude will build the same structure with you, explaining each step as it goes. Same result; you choose the way you learn best.