JUNE 2026 EDITION

YOU OUTGREW THE CHAT WINDOW

The field manual for turning Claude into dashboards, briefings, and working agents.

OFIR BLOCH | 10 PROJECT PLAYBOOKS | FREE, NO EMAIL

TABLE OF CONTENTS

What's in this guide

WHAT'S INSIDE

Built for people who don't write code

This guide is for marketing, operations, customer success, sales, and the people who run the work. You have ideas and projects. You have never needed a programming background to ship them, and now you need it less.

Claude builds things when you describe them in plain English. Reports, dashboards, automations, the work that used to wait six weeks in someone's backlog. This guide starts easy with Cowork and graduates to Claude Code, the surface that builds and runs real things. Every term gets explained the first time it appears.

▸ **HOW TO READ THIS**

Skim the parts you need. Each one is self-contained and ends with something you can use. The playbooks near the back are copy-and-run. Start anywhere.

LEGAL NOTICE

READ THIS PART ONCE

Then forget it and go build something.

- Educational use. The author is a Claude user, not affiliated with Anthropic beyond that.
- Product names and features reflect publicly available information as of June 2026 and will change.
- Results vary. Do not put sensitive or regulated data into any AI tool without proper review.
-

PART ONE

WHAT IS THIS THING?

The plain-English tour: what Claude Code is, where it lives, and what it takes to start.

- The one-paragraph version
- The five surfaces
- The model family in plain English
- What you need before you start

FIVE DOORS

The same Claude shows up in five places

Claude is one product with five doors, and they differ in how much they can touch. A chat window writes you a draft. The Code surface writes the draft, saves it, renames forty files around it, and schedules itself to do it again on Friday.

Cowork deserves a flag, because the old advice gets it wrong. It is a tab inside the Claude Desktop app, sitting next to Chat and Code. There is nothing separate to download.

SURFACE	WHAT IT IS	BUILT FOR
Chat (Desktop tab)	Conversation, drafts, thinking out loud	Everyday questions and writing
Cowork (Desktop tab)	Workspace with point-and-click connectors	Hooking up email, calendar, files
Code (Desktop tab or terminal)	The full agent: reads, writes, runs, schedules	Building things that persist
claude.ai (web)	Browser access with no install	Working from any machine
Mobile	Check on and nudge running agents	Approvals from the couch

▲ THIS WILL BITE YOU

Tabs do not share memory. A project you set up in Chat will not follow you into Code, and each tab starts fresh. Pick the surface where a workflow lives and keep that workflow there.

THE MODEL FAMILY

Four models, four pay grades, one dial

Under the hood, Claude is a family of models. Think pay grades: a fast junior for grunt work, a reliable manager for daily tasks, a senior specialist for hard problems, and a flagship above them all. Opus is the practical ceiling for everything in this guide.

You pick the model for the job, and you can switch mid-conversation without losing the thread. Most people leave the default alone and never think about it. The dial exists for the day volume or cost starts to matter.

MODEL	THINK OF IT AS	REACH FOR IT WHEN
Haiku 4.5	The fast junior	Quick lookups, sorting, bulk reading
Sonnet 4.6	The reliable manager	Drafts, analysis, most daily work
Opus 4.8	The senior specialist	Complex multi-step projects
Fable 5*	The chief	Flagship tier. Rarely needed; see the note

VERIFIED JUNE 2026

* Fable 5 is suspended as of June 2026 (see below). Cost per million tokens, in and out: Haiku 4.5 at $1/$5, Sonnet 4.6 at $3/$15, Opus 4.8 at $5/$25. Switch any time by typing /model haiku, sonnet, or opus. Fable 5, the flagship above Opus, launched June 9 2026 at $10/$50, twice the price of Opus, and was suspended on June 12 2026 under a US government directive, with Opus 4.8 as the fallback.

WHY BOTHER

Your bottleneck was never ideas

Every operator keeps a drawer of things they would build if a developer were on call: the tracker that updates itself, the dashboard, the report that assembles before the Monday meeting. What stood between you and them was a ticket queue and somebody else's roadmap.

That gap is the thing that closed. Describing work clearly is now enough to get it built, and describing work clearly is what you have done for a living all along.

THE THING YOU SHELVED	WHAT IT COSTS NOW
A status page for one messy project	One conversation
A folder that sorts and renames itself	Ten minutes, then it runs forever
A weekly report that drafts itself	One afternoon of setup
The small internal tool nobody would build	An evening

◆ WORTH KNOWING

Honest expectations: the first hour feels strange, version one lands about 80 percent right, and you still do the judging, the checking, and the deciding. What disappears is the part where good ideas die waiting for someone technical to have a free week.

BEFORE YOU START

The full shopping list is shorter than you think

There is no environment to set up, no configuration step, no prerequisite course. The list below is everything. If you sent an email today, you already meet the technical bar this book assumes.

1. A computer. Mac or Windows, whichever you already use
2. A paid Claude plan. Pro covers everything in this book
3. The Claude Desktop app, one download from claude.com
4. Thirty minutes nobody can take from you
5. Zero coding experience. The book assumes none and never will

▸ **DO THIS**

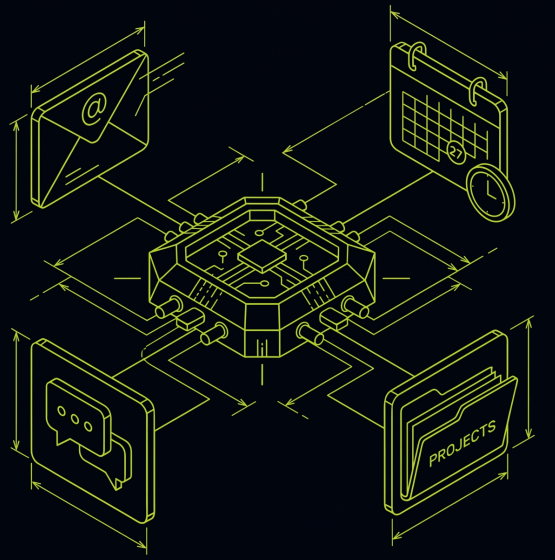
Download the Claude Desktop app and sign in before you turn the page. Part 2 opens with your first real task, and the only setup it needs is you, logged in.

PART TWO

START HERE: COWORK

Claude with hands. Your documents, your inbox, your calendar, one tab.

- What Cowork is
- How to open it
- Your first task
- Seven things Cowork is great at
- Connectors: point-and-click
- When to graduate to Claude Code



CLAUDE WITH HANDS

Cowork is a tab in the Desktop app, not a separate download

Open the Claude Desktop app. You will see three tabs at the top: Chat, Cowork, and Code. Cowork is the middle one. That is where you start. There is no separate Cowork installer, no second app, no different account.

The difference from the Chat tab is simple. Chat answers questions. Cowork takes actions: reads your files, connects to your calendar and inbox, produces documents, and saves output where you tell it to. You stay in control. Nothing sends without your say-so.

CHAT TAB	COWORK TAB
You ask, it answers	You ask, it works
No file access by default	Reads and writes files
No connectors	Gmail, Drive, Calendar, Slack, and more
Each session starts blank	Shared project context across sessions
Good for thinking and drafting	Good for getting things done

◆ WORTH KNOWING

Memory, projects, and MCP connections do not travel between tabs. A workflow you set up in Cowork stays in Cowork.

OPEN IT

Four steps from zero to your first Cowork session

If you have the Desktop app installed, you are already most of the way there. If not, download it from claude.ai. No separate Cowork install exists.

1. Download and open the Claude Desktop app from claude.ai. Sign in with your Claude account.
2. Click the Cowork tab at the top of the window. It sits between Chat and Code.
3. Create a new project or open an existing one. Give it a name that matches what you are working on.
4. Type your first request. Try something concrete: a document sitting on your desktop, a report you want cleaned up, a long email thread you need summarized.

▸ DO THIS

Before you pick a task, open the Connectors panel on the left side of the Cowork tab. You will see what is already available. The ones showing a green indicator are live. The ones showing a button are ready to connect in one click.

FIRST TASK

Turn a messy exported file into a clean report

Here is a worked example anyone can follow. You have a raw export from a project tool, a spreadsheet dump, or a set of unstructured meeting notes. You want a clean report with clear sections and action items pulled out.

Drag the file into the Cowork window. Type the request below. Review the draft before saving anything.

>_ COPY THIS

```
Here is a raw export from our project tracker. Please:
1. Summarize the current status in three bullets.
2. Pull out every open action item with its owner and due
date.
3. Flag anything that looks overdue or unassigned.
4. Format the output as a short report, one page maximum.

Do not change any numbers or names. If something is
unclear, ask me before guessing.
```

WHAT IT IS BUILT FOR

Seven things Cowork handles well

Cowork is general-purpose, but it shines on the kind of work that sits in your to-do list for weeks because it is tedious but requires judgment. These seven categories cover most of what non-developer teams need.

TASK	WHAT YOU GIVE IT
Report drafting	Raw data or notes, a format, who it is for
Document cleanup	A messy draft, a target style
Meeting summary	A transcript or notes file
Data extraction	A spreadsheet or export, what to pull
Inbox triage	Gmail connector, a priority filter
Calendar prep	Calendar connector, the meeting you are walking into
Research digest	A set of URLs or attached documents

◆ WORTH KNOWING

The pattern is the same for all of them: you supply the raw material and the judgment call, Claude supplies the labor. The edit pass is still yours.

CONNECTORS

Link your tools with a click, no JSON required

Cowork ships with dozens of built-in connectors. Gmail, Google Drive, Google Calendar, and Slack are the most common starting points. Connecting any of them takes about thirty seconds: open the Connectors panel, click the tool you want, log in with your existing account, and it is live.

Once a connector is active, Cowork can read from it and write to it inside the same conversation. Ask it to check your calendar before scheduling something, or pull the latest version of a doc from Drive. You stay in one window.

1. Open the Connectors panel on the left side of the Cowork tab.
2. Find the tool you want. Gmail, Drive, Calendar, and Slack are near the top.
3. Click Connect. An OAuth login screen opens in your browser.
4. Log in with the account you already use for that tool. The window closes.
5. The connector shows a green indicator. Start using it in the same conversation.

VERIFIED JUNE 2026

Cowork ships with dozens of built-in connectors, with more available through the plugin marketplace. Check the Cowork connectors panel for the current list. All Cowork connectors use OAuth authentication: click, log in, done, with no JSON files and no terminal commands required. The JSON configuration path exists only in the Claude Code CLI for custom or self-hosted tools.

THE NEXT LEVEL

When Cowork is not enough

For most knowledge workers — marketing, ops, customer success — Cowork is not the on-ramp. It is the main event. If 90% of your time ends up here, that is not a gap in your setup. That is the right call.

Claude Code, covered in Part 3, adds full file system access, the ability to write and run code, hooks that fire automatically at key moments, and scheduled tasks that run without you. The table below shows when the move makes sense. Not everyone needs to cross it.

STAY IN COWORK	MOVE TO CLAUDE CODE
Working with documents and connected tools	Writing or running code
Drafts, summaries, and formatted reports	Automating tasks on a schedule
Calendar, email, and inbox workflows	Building tools that run in the background
Team collaboration in a shared project	Custom hooks and automated safety checks
Point-and-click connector setup	Self-hosted or custom MCP servers

▸ DO THIS

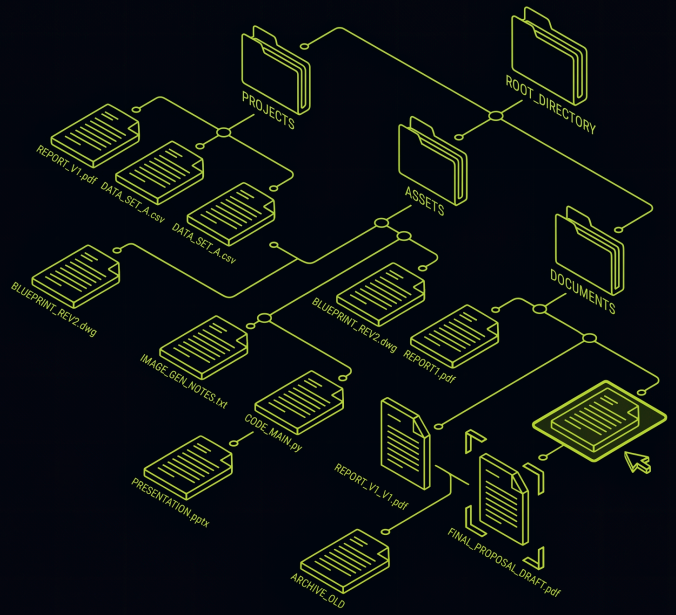
Pick one task you do manually every week. Open Cowork, connect the relevant tool if needed, and ask Claude to handle it. Time how long it takes. That number is the argument for everything that follows.

PART THREE

LEVEL UP: CLAUDE CODE

Where you start building things that run.

- What Claude Code is and where it lives
- Your first session and the four modes
- Permissions, checkpoints, and rewind
- A landing page from one sentence
- Project folders and meeting CLAUDE.md
- Interview first, plan as a file, build in gates



GET IT RUNNING

How to install Claude Code and open your first session

Desktop app users: the Code tab is already Claude Code. Pick a folder and skip to the next page.

Terminal version: open Terminal on Mac or PowerShell on Windows, then run these commands in order. Type each line exactly as shown and press Enter.

terminal

```
1. Check if Node.js is installed
> node -v
2. Install Claude Code
> npm install -g @anthropic-ai/claude-code
3. Launch and log in
> claude
4. Open a project folder, then start a session
> cd ~/Desktop/my-project
> claude
```

◆ WORTH KNOWING

If step 1 says 'command not found', Node.js is not installed. Go to nodejs.org, download the LTS version, accept all defaults, then return to step 2. After the full install, type claude --version to confirm it worked.

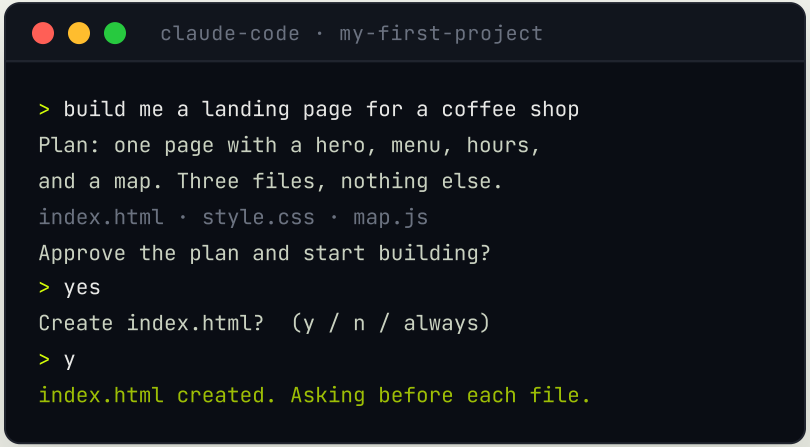
YOUR FIRST SESSION

What a session actually looks like

You type a request. Claude answers with a plan, asks before touching anything, and reports what it did. The whole rhythm fits on one screen: a sentence in, a plan out, a permission question, a file.

>_ COPY THIS

```
build me a landing page for a coffee shop. plan first.
```



◆ WORTH KNOWING

Nothing happens to a single file until you say yes. The first time you watch Claude ask permission, the fear goes away. That is the trade Claude Code offers: more power than chat, and a leash you hold.

THE FOUR MODES

Pick how much rope to give it

Shift+Tab cycles the modes. There are four. The first edition of this manual listed three, named Plan, Code, and Act, with great confidence. Consider that corrected.

Each mode answers one question: how much can Claude do before checking in with you. Start every new piece of work in Plan, then hand over rope as the plan earns it.

MODE	WHAT IT DOES	WHEN TO USE IT
Plan	Reads your files and proposes. Builds and changes nothing.	Always start here
Accept Edits	Edits files freely, still asks before running commands.	Once the plan is approved
Auto	Runs with minimal interruptions. A background check screens each action.	Long trusted jobs, where available
Default	Asks before most actions, one at a time.	Sensitive work, early days

▲ THIS WILL BITE YOU

Skipping Plan is how you end up with thirty enthusiastic edits to the wrong files. Let Claude describe the build first, read it like a contract, then move up a mode. Two minutes of reading beats an hour of untangling.

THE PERMISSION SYSTEM

Three answers cover every request

Outside Plan mode, Claude asks before acting: create this file, run this command. You have three answers, and together they are the entire security model.

Start cautious. Approve one action at a time until the requests turn boring. Allow always is a real timesaver for reads and routine commands, and a bad idea for anything that deletes, sends, or spends.

YOUR ANSWER	WHAT HAPPENS	USE IT FOR
Allow once	This single action runs. Claude asks again next time.	Everything, in your first week
Allow always	That command or tool is pre-approved in this project from now on.	Reads and routine commands you understand
Deny	Nothing runs. You say what you want instead.	Anything that surprises you

▸ DO THIS

When a request surprises you, deny it and ask Claude why it wanted that. The explanation teaches you more about your own project than any tutorial, and a wrong deny costs nothing.

CHECKPOINTS, REWIND, AND THE DIFF VIEW

Two commands that make experiments cheap

Claude Code saves a checkpoint as it works. Press Esc twice on an empty input line and a rewind menu opens. Pick a point and both your files and the conversation roll back to it. The worst case stopped being a ruined afternoon and became two keypresses.

Before you accept any change, type /diff. A viewer opens: every file that changed, how much, and which turn did it. You do not need to read code to read a diff.

```
claude-code · coffee-site

> /diff
3 files changed this session
index.html   41 lines added   turn 2
style.css    88 lines added   turn 2
map.js       17 lines added   turn 4
arrows to browse · enter for detail · esc to close
```

◆ **WORTH KNOWING**

Left and right arrows switch between the full session diff and what each individual turn changed. When something looks off, the per-turn view tells you exactly which request caused it, which is exactly what rewind needs. Rewind covers what Claude changed; edits you made yourself in other apps sit outside it.

YOUR FIRST PROJECT

A landing page, end to end

Time for the whole loop with no training wheels. One sentence of intent, a plan, a permission or two, and a page you can open in a browser. Twenty minutes start to finish, most of it watching.

1. Make a new empty folder. Call it coffee-site.
2. Open the Code tab and pick that folder. Shift+Tab to Plan.
3. Paste the prompt below and read the plan it returns.
4. Approve. Answer y as each file gets created.
5. Find index.html in the folder and open it in your browser.
6. It is a real website you built before lunch.

>_ COPY THIS

```
Build a one-page site for a neighborhood coffee shop. Sections: a hero with one strong headline, a short menu, opening hours, and a location map. Clean and mobile friendly. Show me the plan before creating anything.
```

MAKE IT YOURS

The edit loop you will use forever

The first version will be eighty percent right. The remaining twenty is conversation: one change at a time, read the plan, check the diff, rewind what lands wrong. This loop is the entire craft, and every later chapter is this loop wearing better clothes.

claude-code · coffee-site

```
> make the headline twice the size and move hours above the menu
Two edits: index.html and style.css.
Edit index.html?  (y / n / always)
> y
Done. 9 lines changed across 2 files.
> hmm, hours below the menu was better
Moved back. Esc Esc would also rewind it.
```

▸ DO THIS

Change three things on your page tonight, one at a time. Then rewind one of them on purpose. Knowing the undo works changes how you ask forever.

PROJECT SETUP

One folder is the whole project

Claude Code works inside the folder you opened, and only there. That folder is the project: the files Claude builds, the notes it reads, the settings it follows. Moving the folder moves the project. Backing it up backs everything up.

Keep one folder per project. Mixing the newsletter, the budget, and the website in one directory gives Claude a junk drawer to reason about, and you get junk drawer answers.

INSIDE THE FOLDER	WHAT IT IS
Your files	Pages, docs, and data Claude builds and edits
CLAUDE.md	Standing orders Claude reads at every session start
.claude/	Project settings, saved permissions, skills
Anything you drop in	Readable context, no pasting required

◆ WORTH KNOWING

The folder is also the safety boundary. Claude asks before touching anything outside it. A fresh folder per project keeps the blast radius of any mistake one directory wide.

MEET CLAUDE.MD

A note Claude reads before every session

One file in that folder deserves an introduction before this part ends. CLAUDE.md is a plain note Claude reads at the start of every session: who you are, what this project is, the rules that never change. Write it once and stop repeating yourself.

Type /init and Claude drafts one from what it sees in the folder. Three honest lines beat a blank file.

claude-code · coffee-site

```
> /init
Scanning the folder...
Drafting CLAUDE.md: what this project is,
how files are organized, what to leave alone.
Saved. Loads automatically every session.
> add: never touch style.css without asking
Added under Rules.
```

◆ WORTH KNOWING

This file is the seed of something bigger. Part 5 grows it into a full second brain: memory, a knowledge base, and handoffs between sessions. For now, /init plus your top three rules is plenty. If you have not built the coffee site yet, do it tonight and rewind something on purpose.

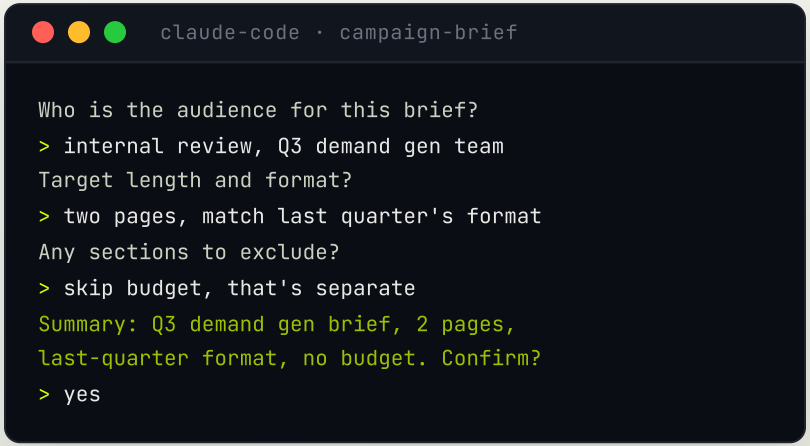
INTERVIEW FIRST

Ask Claude to interview you before it builds anything

The most common way to get the wrong output is to describe what you want without letting Claude ask what it needs to know. One addition to your prompt fixes this: tell Claude to interview you first. This works for reports, briefs, emails, and builds.

>_ COPY THIS

```
Before you start, interview me. Ask me one question at a
time until you understand exactly what I need. When you
have no more questions, summarize your understanding and
wait for my confirmation before doing anything else.
```



◆ WORTH KNOWING

Three good questions from Claude are worth more than an hour of revision. Most first-draft mismatches trace back to assumptions Claude made in the first thirty seconds. The interview stops those assumptions at the source.

PLAN.MD

Make the plan a file, not a chat message

After the interview, ask Claude to write PLAN.md. A plan written into chat disappears the moment you rewind or close the session. A file stays, can be edited directly, and doubles as a record of what was built and why.

The step gates keep you in control during execution: you approve each step before the next one begins.

1. After the interview, ask: write a step-by-step PLAN.md, one numbered step per line.
2. Open the file. Edit any step that is wrong before approving.
3. Say: execute step 1 of PLAN.md. Stop when step 1 is done and wait for my approval.
4. Review what step 1 produced. Approve or correct before moving to step 2.
5. Continue one step at a time. PLAN.md is the checkpoint record for the whole job.

>_ COPY THIS

```
Write a PLAN.md in this folder. Number every step. Each step should name what it does and what it produces when done. Stop after writing the plan. Wait for my approval before beginning step 1.
```

CLI VS DESKTOP

Claude Code CLI vs Desktop: same engine, two front doors

The terminal and the Desktop app run the same engine: same models, same features, same underlying architecture. Choosing between them is not choosing a better Claude. It is choosing a working environment.

DIMENSION	CLAUDE CODE CLI (TERMINAL)	CLAUDE CODE DESKTOP (GUI)
MCP connectors	Manual JSON config in .mcp.json. Full capability, requires editing config files.	Point-and-click OAuth setup for GitHub, Slack, Linear, Notion, Google Drive, and 38+ others. No JSON editing.
Beginner friendliness	Requires comfort with a terminal. No onboarding UI.	Recommended starting point for non-developers. Drag-and-drop layout, sidebar sessions, embedded app preview.
Best for	Power users, automation, CI pipelines, anyone who already works in a terminal.	Non-developers, mixed teams, anyone who wants visual diff review or point-and-click connectors without CLI setup.

◆ THE AUTHOR'S TAKE

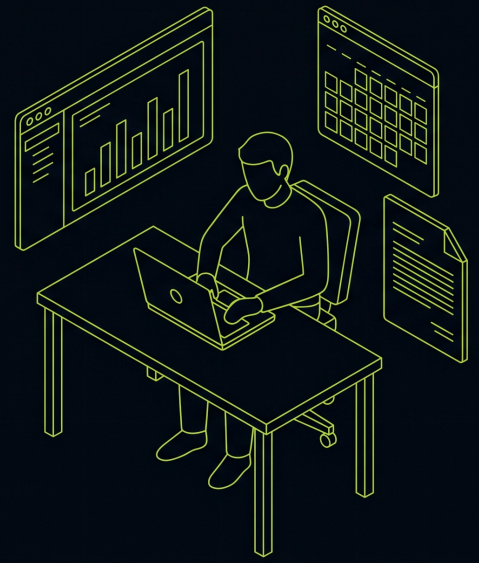
The terminal is my default because I can pipe output into other commands, run Claude headlessly inside scripts, and keep background agents running without extra CPU overhead. If I were onboarding someone who had never used a terminal, I would send them to Desktop and tell them to stay until the CLI felt like a natural next step.

PART FOUR

TALKING TO CLAUDE

How to ask so the first draft is the one you wanted.

- The freelancer on day one
- Interview, Plan, Build
- Plan mode and five prompting rules
- The good ask plus a kickoff skeleton



THE FREELANCER TEST

You would never brief a freelancer this way

Picture a freelancer on day one. Talented, fast, knows nothing about your business yet. You would never say make it pop and walk away. You would explain the audience, the goal, the format, the deadline, and what good looks like.

Claude is that freelancer at the start of every task. What comes back tracks the quality of the brief. A plain, complete brief outperforms the wittiest one-liner.

Voice works just as well as typing. Wispr Flow (wisprflow.ai) dictates into any app on Mac, Windows, iOS, and Android and strips filler words before the text lands.

WHAT A FREELANCER GETS	WHAT CLAUDE USUALLY GETS
The goal and who it is for	A one-line request
Examples of work you liked	No examples
A deadline and a format	"Make it professional"
A chance to ask questions	Silence until the rework

◆ WORTH KNOWING

Claude responds to the same things a sharp contractor responds to: a clear goal, real context, an example, and a definition of done.

INTERVIEW FIRST

Ask Claude to interview you before it builds anything

Flip the opening move. Instead of writing the perfect brief, ask Claude to pull the brief out of you. Tell it to interview you about the task before doing anything: one question at a time, until it could explain the job to someone else.

The questions do the work. They surface the constraint you forgot you had, the audience you never named, the format you assumed was obvious. Once the questions run out, it plans, and only after your approval does it build. Interview, Plan, Build, in that order.

claude-code · client-report

```
> Monthly client report. Interview me first.
Three questions before I touch anything.
1. Who reads it, and what do they decide with it?
> The ops lead. She decides budget renewals.
2. What did the last report get wrong?
> Too long. She wants the three numbers that moved.
One more question, then I will propose a plan.
```

>_ COPY THIS

```
Interview me about this task before doing anything. One question at a time. Stop when you could brief another person on it.
```

PLAN MODE

Review the plan before anything gets built

You met plan mode in Part 3: the read-only mode where Claude can look but cannot touch. Shift+Tab cycles into it. Claude studies the task, asks what it needs to ask, and presents numbered steps. Nothing executes until you approve.

The sixty seconds you spend reading that plan catch misunderstandings while they are still theoretical. A wrong plan costs one sentence of feedback. A wrong execution costs the runtime, the diagnosis, and the redo. Read every plan like a contractor's quote.

1. Every item it will create or change is named, none are vague
2. The steps run in an order that makes sense
3. Side effects are listed: new tools, new folders, anything contacted
4. It names what it is unsure about
5. The scope matches what you asked for, nothing extra

▲ **THIS WILL BITE YOU**

A fifteen-step plan for a three-step ask is scope creep, caught early. Do not approve it and hope. Reject and restate: focus only on X, do it in four steps. Smaller plans execute more reliably.

FIVE RULES

Five rules cover almost every prompt you will write

There is no secret syntax. Good prompting is good delegation, and delegation has rules people have known for a century. These five are the whole playbook. Most weak results trace back to skipping exactly one of them.

In practice: "Write a competitor summary" gets a generic draft. "Summarize these 3 competitor sites for a sales team, one page, pricing and positioning, match @april-summary.md" gets a usable draft on the first pass.

THE RULE	WHAT IT SOUNDS LIKE
Be specific	"Three slides for a renewal call" beats "make a deck"
Give context	Who it is for, why now, what happened last time
Show examples	"Match the tone and structure of @last-month.md"
Define done	"Done means one page, saved to /reports, zero jargon"
Let it ask	"Ask me anything unclear before you start"

▸ DO THIS

Take one prompt that let you down this week. Score it against the five rules. Add the missing ones and run it again.

SHOW, DON'T DESCRIBE

An example teaches faster than an explanation

Rule three from the previous page — show examples — is the most underused technique. Tell Claude a founder writes with sophisticated but light humor and it produces something generic. Feed it three emails the founder actually wrote and it matches the pattern.

Description is never as precise as the thing itself.

INSTEAD OF DESCRIBING	SHOW THIS
"Match our brand voice"	Paste 2-3 paragraphs from a piece you liked
"Same structure as last month"	Paste the previous output or @last-month.md
"Professional but not stuffy"	An email you sent that hit the right register
"Like a slide deck, not an essay"	A slide you liked: header length, bullet count

▸ DO THIS

Pick any task where Claude missed the tone. Find one real example of what you wanted. Add it with @filename or paste it directly. Run the same request again.

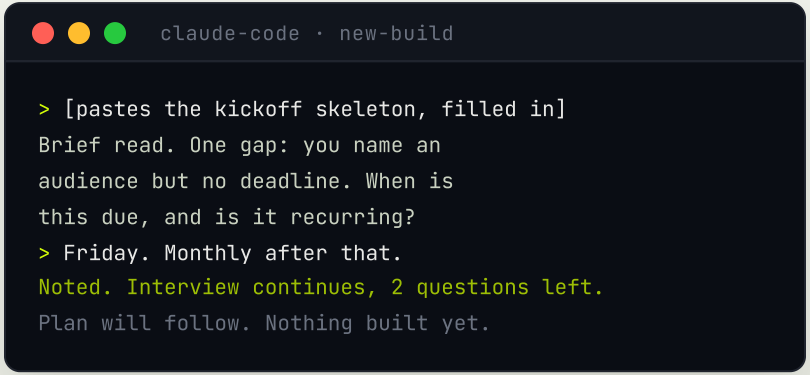
COPY THIS

A kickoff skeleton for any build

Paste this at the start of any real task: a report, a tracker, a deck, a cleanup. Fill the brackets, delete what does not apply. It packs the interview, the five rules, and the plan review into one block.

>_ COPY THIS

```
I want to build [thing].

## Context
- For [audience], who will use it to [decide or do what]
- Why now: [the trigger]
- Related files: [@file, or none]

## What done looks like
- [Format, length, where it lives]
- [What it must include]

## Process
1. Interview me first. One question at a time.
2. Show me a numbered plan. Wait for approval.
3. Build it. Flag anything you had to guess.
```



▸ DO THIS

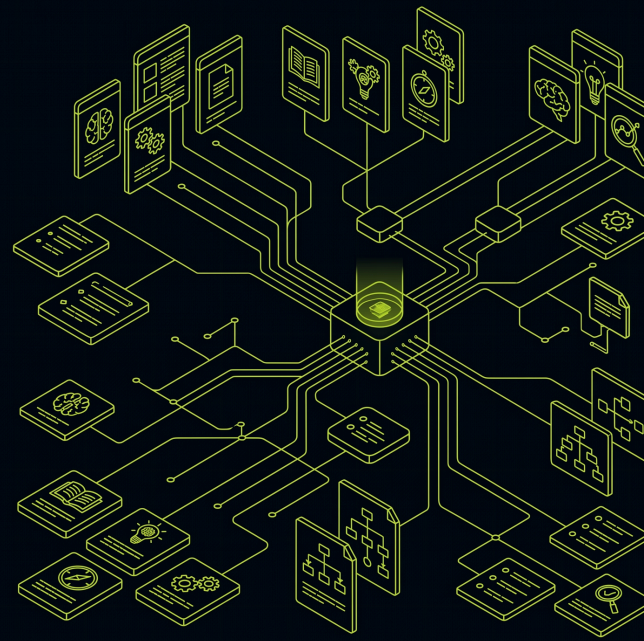
Save the skeleton as kickoff.md in your work folder. On your next real task, open with @kickoff.md and fill it in live.

PART FIVE

YOUR SECOND BRAIN

A small set of files that lets Claude wake up already knowing you.

- Memory and CLAUDE.md
- The knowledge base
- Saving and sharing context
- Ending sessions and context hygiene



THE AMNESIA TAX

Claude forgets you the moment the session ends

Every session, Claude is brilliant. And every session it wakes up knowing nothing about you: your name, yesterday's project, the decision you reached after three rounds of iteration last week. All gone.

So you re-explain, re-decide, and watch it suggest the approach you rejected on Tuesday. That re-orientation is the real tax, and most people pay it every morning without noticing.

WHAT YOU BUILT IN THE SESSION	WHAT SURVIVES THE CLOSE
Decisions and their reasoning	Nothing
Context you explained	Nothing
Approaches you ruled out	Nothing
Files Claude saved to disk	Everything

◆ WORTH KNOWING

The fix is a small set of files Claude reads automatically at startup. This part builds the full kit: standing orders, auto memory, a knowledge base it can read, and a five-minute close ritual.

TWO MEMORY SYSTEMS

What Claude remembers across sessions, and where

Claude Code has two memory systems, and both load before you type a word. The first is CLAUDE.md: files you write, holding your rules and context. The second is auto memory: notes Claude writes for itself as it learns how you work.

CLAUDE.md comes in scopes. A personal file applies everywhere you work. A project file lives in the work folder and travels with the team. Claude reads them all at startup, broadest first.

MEMORY	WHO WRITES IT	WHERE IT LIVES
User CLAUDE.md	You	~/.claude/CLAUDE.md, every project
Project CLAUDE.md	You or your team	./CLAUDE.md, shared with the project
Local CLAUDE.md	You	./CLAUDE.local.md, this project, your eyes only
Auto memory	Claude	~/.claude/projects/<project>/memory/MEMORY.md

VERIFIED JUNE 2026

Auto memory is on by default in Claude Code v2.1.59 and later. The first 200 lines or 25KB of MEMORY.md load at every session start. Older entries move to topic files Claude reads on demand. Type /memory to see everything currently loaded.

STANDING ORDERS

CLAUDE.md is the file that ends repeat explanations

Think of CLAUDE.md as standing orders for a very capable new hire. Anything you would say in every single session belongs there: who you are, what you are working on, how you want output formatted, what is off limits.

The rule for what earns a line: if you have explained it twice, write it down. Every repeated explanation is unpaid onboarding. Five sections cover almost everyone.

1. Who I am and what I do, in two or three sentences
2. Current projects, one line each, with the phase
3. How I like to work: tone, formats, tools
4. Standing decisions, settled once, never relitigated
5. Off limits: things Claude never does without asking

▲ THIS WILL BITE YOU

Long files get ignored. Community consensus, backed by observed behavior, puts the ceiling around 200 lines: above it, adherence drops and instructions start slipping. If you cannot skim your CLAUDE.md in two minutes, cut it. Keep rules here and move step-by-step procedures into separate files.

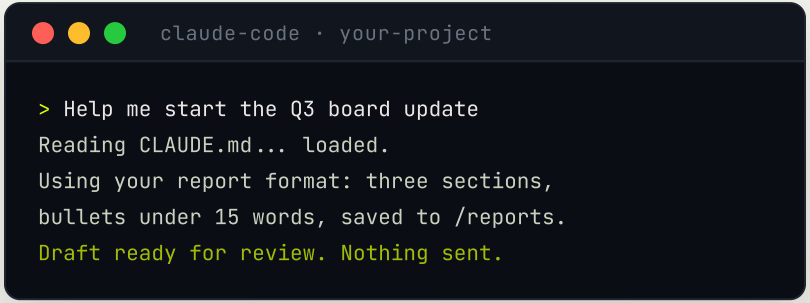
COPY THIS

A starter CLAUDE.md you can paste today

Create a file named CLAUDE.md in your work folder. Paste this skeleton, replace the brackets, delete what does not apply.

>_ COPY THIS

```
# CLAUDE.md

## Who I am
[Role, team, what you produce. Two sentences.]

## Current projects
- [Project]: [phase, what done looks like]
- [Project]: [phase]

## How I work
- Tone: [direct / formal / casual]
- Deliverables: [format, length, where they go]
- File naming: YYYY-MM-DD-topic.md

## Standing decisions
- Save all output to [folder].

## Off limits
- Never send, post, or delete anything without asking.
- Never invent numbers. Flag anything unsourced.
```



THE KNOWLEDGE BASE

A folder of notes becomes a brain Claude can read

If you keep notes in Obsidian, Notion exports, or any folder of markdown files, run Claude from that folder. It reads everything in its working directory. The folder is the integration. No connector to configure, nothing to install.

The loop is capture, organize, recall. Write down anything you had to explain twice. Give each category one home: decisions in a log, project facts in project notes, preferences in CLAUDE.md. At startup, Claude reads its way in and recalls all of it.

SECOND BRAIN (PERMANENT)	CHAT (TEMPORARY)
Who you are and how you work	Today's task details
Active projects and their phase	Drafts mid-iteration
Decisions, with the reasoning	Back-and-forth clarifications
Lessons and corrections	Research for this job only

◆ WORTH KNOWING

The test for any piece of information: will you need it again? Yes means it belongs in a file. If it only matters right now, let it die with the chat. A short CLAUDE.md at the top of your notes folder, describing structure and naming, lets Claude navigate the whole vault unguided.

SESSION HANDOFFS

Context can survive the session boundary

When a session ends, hand the next one a note. A rolling handoff file survives anything.

>_ COPY THIS

```
# handoff.md

## What we are trying to do
[Goal in one paragraph]

## Status
- Done: [...]
- In progress: [...]
- Not started: [...]

## Decisions made, with reasoning
- [Decision]: [why, and what we ruled out]

## Failed approaches
- [What did not work, and why]

## Open questions
- [Unresolved item, current thinking]

## First move next session
[One concrete action]
```

THE CLOSE RITUAL

Five minutes at close beats fifteen at the next start

Most of the value people lose with AI is lost at the close, when a finished conversation evaporates. The session-end ritual converts it into permanent records before the window shuts. Capture five categories, in this order, every time.

Each capture has a home. Decisions go to a decision log with their reasoning. Commitments go to a commitments file you actually check. Everything else lands in the handoff file under today's date, ready for tomorrow.

1. Decisions made, with the reasoning and what you ruled out
2. Commitments: anything promised to another person, with the deadline
3. What changed: files created, documents updated, messages sent
4. Open questions you could not resolve, and why
5. Next steps: one to three actions, concrete enough to start cold

>_ COPY THIS

```
Before we close: summarize this session as 1) decisions made with reasoning, 2) commitments and deadlines, 3) what changed on disk or got sent, 4) open questions, 5) next steps. Format it for my handoff file.
```

CONTEXT ROT

Long sessions degrade, and the tells show up early

A session has a working memory, and it fills. As it does, early content loses influence. Claude asks questions you already answered, re-proposes rejected approaches, or invents files that do not exist. That is context rot. It is predictable and manageable.

You do not need token counts. The behavior is the signal. At the first tell, run /compact with a focus instruction: it compresses history while keeping decisions and file paths.

THE TELL	THE MOVE
Asks a question you answered earlier	/compact now
Re-proposes an approach you rejected	/compact, name what to preserve
References files that do not exist	/clear, reload from your handoff file
Responses turn short and generic	Finish the step, then /clear
Ignores a standing rule from CLAUDE.md	/compact. CLAUDE.md reloads from disk after

◆ WORTH KNOWING

You can steer compaction: '/compact preserve our decisions on naming and format' keeps that thread and drops the noise. It typically frees around half the working space.

WHAT YOU HOLD

You now have a brain that outlives the session

Claude still wakes up blank tomorrow. The difference is that it now wakes up inside your files: standing orders, accumulated memory, a readable knowledge base, and a handoff that says exactly where the work stood. Orientation takes seconds.

From here the system compounds. Every correction becomes a permanent line. Every project leaves a record.

1. A CLAUDE.md under 200 lines, skimmable in two minutes
2. Auto memory on, reviewed now and then with /memory
3. One folder of notes Claude reads at startup
4. A rolling handoff.md in every multi-session project
5. The five-minute close ritual after any real work
6. /compact at the first behavioral tell, /clear between tasks

▸ DO THIS

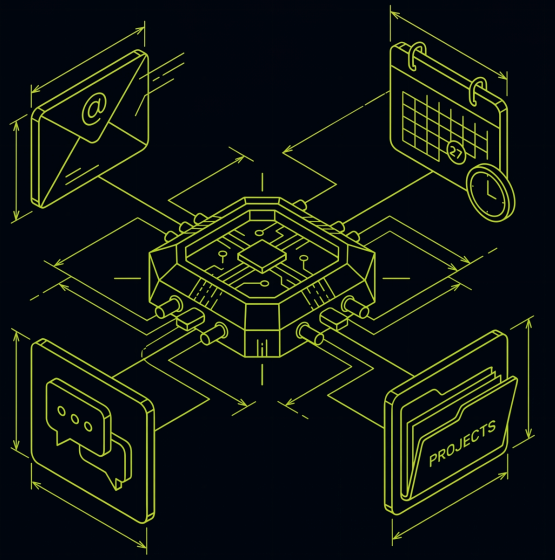
Run the close ritual once tonight, before your next session ends. Tomorrow, open with @handoff.md and time how fast the work resumes. One morning is enough to make the habit stick.

PART SIX

SKILLS, PLUGINS, AND CONNECTORS

A small set of files that makes repeatable work run itself.

- What a skill is and when to make one
- A starter skill you can paste today
- Installing skill packs and the marketplace
- Plugins: bundles that do more
- Connectors and MCP in plain language
- Skill safety: required invocation and permission scoping
- Chrome Extension: connecting to a live browser session
- Browser access and the trust hazard



WHAT A SKILL IS

A skill is a plain text file Claude reads before it starts a task

Skills live in a folder inside your project or your home directory. Each skill is a plain text file: a name, a description, a set of phases. When you invoke one, Claude reads it and follows it for that task only. Other tasks run without it.

The creation rule is blunt: if you have explained the same multi-step process twice, write it down. A skill ends the tax.

WHAT BELONGS IN CLAUDE.MD	WHAT BELONGS IN A SKILL
Rules always active (tone, format, off-limits)	Procedures invoked on demand
Standing decisions already made	Workflows that change by context
Things that apply to every task	Things that apply to one kind of task
"Never send without asking"	"Run the weekly planning workflow"

◆ WORTH KNOWING

Skills outrank same-named slash commands when both exist. If you have old /commands files, a skill with the same name takes over automatically. You can move them in without breaking anything.

THE LIFECYCLE

Skills come from doing the task wrong twice and writing down why

A skill does not start as a skill. It starts as a task you repeat until you notice the pattern.

The non-obvious part: good skills carry a blocked patterns section alongside the instructions. When an approach fails once, it gets written in. Future runs never re-propose it.

```
claude-code · your-project

> /create-skill weekly-planning
Creating .claude/skills/weekly-planning/
Writing SKILL.md with frontmatter...
Scaffold ready. Open SKILL.md to write phases.
Add triggers: so Claude invokes it automatically.
> /weekly-planning
Loading skill: weekly-planning
Phase 1: gathering this week's open items...
Skill running. No re-explaining required.
```

◆ WORTH KNOWING

Skills can have a model: field in their frontmatter. A session-end skill that only writes files pins model: haiku. A content review skill that needs judgment pins model: opus. The skill, not the session, decides which model runs the work.

COPY THIS

A starter skill you can paste today

Create a folder named after your workflow inside .claude/skills/. Paste this into SKILL.md. Fill in the phases from how you already do the task. The blocked patterns section is empty until a run goes wrong, and then it is never empty again.

>_ COPY THIS

```
---
name: my-workflow
description: One sentence. Trigger phrases.
model: sonnet
triggers: ["run my workflow", "weekly review"]
---

## Phase 1: Load context
[What to read before starting]

## Phase 2: Collect
[What data to gather, from where]

## Phase 3: Produce
[What the output looks like, format and location]

## Blocked patterns
- Do not do X (rejected YYYY-MM-DD, reason)
```

THE MARKETPLACE

Other people have already written skills you need

The /plugin command opens a four-tab browser: Browse, Installed, Updates, Recommended. Before you install anything, Claude Code shows you how much context that plugin adds to every conversation. That number is the cost you pay for every session it loads, whether you use it or not.

A popular community example is Superpowers, a methodology pack that encodes TDD, spec-driven development, systematic debugging, and code review as individual skills.

1. Type /plugin to open the marketplace browser
2. Browse or search for the pack you want
3. Check the context cost shown before installing
4. Install: /plugin install superpowers@claude-plugins-official
5. Invoke a skill it added, for example /superpowers:using-superpowers to start

VERIFIED JUNE 2026

Superpowers (obra/superpowers) is a popular community methodology pack for Claude Code. Key commands: /superpowers:writing-plans, /superpowers:executing-plans, /superpowers:brainstorming. Install via /plugin install superpowers@claude-plugins-official.

SKILL SAFETY

Two frontmatter settings for skills that touch important systems

As your library grows, two additional frontmatter fields prevent skills from running in the wrong moment or with more access than they need.

FRONTMATTER FIELD	WHAT IT DOES
disable-model-invocation: true	Claude cannot auto-invoke this skill via trigger match. Only an explicit call by name starts it. Use on any skill that sends, publishes, or deletes.
allowed-tools: Read, Grep	Limits which tools this skill can use. A community skill requesting allowed-tools: Bash(*) has full command execution access. Restrict to exactly what the skill needs.

▶ DO THIS

Before installing any community skill, read its frontmatter for allowed-tools. Read and Grep are low-risk. Bash(*) means the skill can run any command on your machine. If you do not recognize the author and Bash(*) is in the list, do not install it.

PLUGINS EXPLAINED

A plugin ships skills, hooks, and connectors as one unit

MCP (Model Context Protocol) is the standard that lets Claude reach external tools. In Cowork the setup is point-and-click OAuth; the full walkthrough is in Part 2.

A skill is one workflow. A plugin can include skills, subagents, hooks, MCP server configurations, and output style definitions, all installed together from a single source. When you install Superpowers, you get 14 methodology skills and the hooks they depend on, as one unit.

Plugins install at three scopes. User scope applies everywhere you work. Project scope applies to one project and travels with it in version control. Managed scope is what an IT team deploys for all users in an organization, with no individual install step required.

PLUGIN SCOPE	WHERE IT INSTALLS	WHO IT APPLIES TO
User	~/.claude/plugins/	You, across all projects
Project	.claude/plugins/	Everyone working in this project
Managed (org)	System-level path, IT-deployed	All users on the organization account

▲ THIS WILL BITE YOU

Every plugin you install adds its context cost to every conversation, whether you invoke it that session or not. Audit with /usage to see which plugins are consuming the most context. A plugin you installed once and forgot adds its context cost to every session after that.

COMMON CONNECTORS

A short table of what you can wire up today

MCP (Model Context Protocol) is the standard that lets Claude reach external tools. In Cowork the setup is point-and-click OAuth; the full walkthrough is in Part 2.

The connectors below are available in the official Cowork marketplace with point-and-click OAuth. Developer tools connect via the CLI path with a JSON block in .mcp.json.

CONNECTOR	WHAT CLAUDE CAN DO WITH IT	SETUP PATH
Gmail	Read inbox, draft replies, search threads, create labels	Cowork OAuth
Google Calendar	Read events, create and update meetings, check availability	Cowork OAuth
Google Drive	Read and summarize documents, create files, search	Cowork OAuth
Slack	Read channels, draft messages, search conversations, react	Cowork OAuth
Linear / Jira	Read and create issues, update status, search projects	Cowork OAuth
Notion	Read and write pages, search workspace, create databases	Cowork OAuth

● VERIFIED JUNE 2026

Cowork ships with dozens of first-party connectors. The community marketplace adds more. The exact catalog changes with each release. Open the Connectors tab in Cowork to see what is currently available on your account.

CHROME EXTENSION

Connect Claude to your already-open Chrome session

The Claude in Chrome extension gives Claude read access to your live browser: the page you are looking at, its DOM, and its console output. This is different from the Playwright plugin, which launches a separate controlled browser. With the extension, Claude sees what you see, including pages you are already logged into.

1. Search the Chrome Web Store for "Claude" by Anthropic, or navigate directly to the extension page. The listing name is Claude, published by Anthropic (claude.com).
2. Click the extension icon and sign in with your Anthropic account.
3. Open the Claude Code Desktop app and enable browser tools in Settings.
4. The extension icon turns green when Claude is connected to your browser.
5. Claude can now read your active Chrome tab and interact with it on request.

◆ **WORTH KNOWING**

Chrome and Edge only. The extension does not work in Brave, Arc, or Firefox. Requires a paid Claude account (Pro, Max, Teams, or Enterprise). In the Claude Code CLI, browser integration is configured via MCP in .mcp.json rather than a command-line flag.

LIVE BROWSER USE

Five things you can do with a live session that plugins cannot

The extension accesses pages you are already logged into, without re-authentication and without giving Claude your credentials. Every action still requires your approval before anything executes.

TASK	WHAT YOU SAY
Debug a broken page	Read the console errors on this tab
Review a live design	Does this match the brand guide in BRAND.md?
Work in a logged-in app	Fill the form from the data in my notes
Extract live data	Pull the table on this page into a CSV
Test a deploy	Check that my staging URL shows the new header

▲ THIS WILL BITE YOU

The extension shares login state. Any logged-in tab Claude can see, it can interact with. Use a dedicated Chrome profile for Claude work. In the extension settings, restrict access to specific sites. Never run the extension alongside open email, banking, or HR tabs.

BROWSER ACCESS

Claude can open a browser and act in it. Two risks are worth naming

Claude Code can open a browser, navigate pages, fill forms, and extract content. The setup in Cowork is graphical. In the CLI, you point it at a running Chrome or Brave instance via a CDP port. Either way, Claude can read the page it sees and take actions on your behalf.

The practical uses are real: pulling data from a dashboard that has no API, filling out a form, verifying that a page loads correctly after a deploy. The risks are also real, and they are specific.

1. Confirm which tab is active before any form fill or click
2. Check the URL Claude reports before it takes any action
3. Never give browser access to a session also holding API keys or credentials
4. After autonomous browsing, review the action log before treating results as final
5. Treat any page that cannot be re-navigated as a one-shot: verify before Claude proceeds

▲ THIS WILL BITE YOU

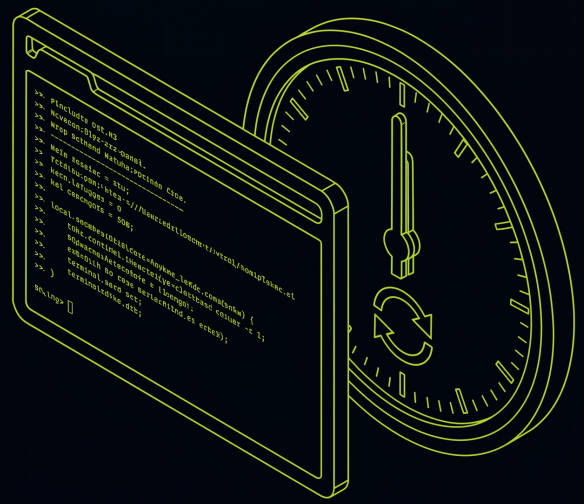
Plugins and skills install and run code on your machine. An untrusted plugin is untrusted code. A cloned repository that ships a .claude/ folder can pre-load hooks that run before you type anything. Before opening any repo in Claude Code, look at its .claude/ directory first. This is not hypothetical: security researchers have publicly demonstrated this exact path, where a cloned repo whose .claude/ folder runs code that exfiltrates secrets before you type anything.

PART SEVEN

PUT IT ON A SCHEDULE

Stop being the alarm clock for your own automation.

- From running tasks to owning systems
- Build the morning briefing
- Where it runs: cloud, desktop, terminal
- Make it fail loud
- Set yours up tonight



THE PROMOTION

You were the bottleneck. A schedule fixes that.

Tuesday, 6:55am. You are asleep. A job wakes up anyway, reads your calendar, scans last night's email, and writes a briefing about a day you have not started. At 7:10 you read it over coffee and fix one mislabeled priority.

Every workflow in this book so far began with you opening Claude and asking for something. The asking was the bottleneck. You were the trigger, every single time. From here on, a system runs the task on a schedule and your job is to review what it produced. The work still happens every morning. You are no longer the reason it happens.

BEFORE A SCHEDULE	AFTER A SCHEDULE
You open Claude, explain context, ask	The job reads context, runs, delivers
Work starts when you remember to start it	Work starts whether or not you remember
Output quality depends on your morning energy	Output quality depends on the prompt you wrote once
You own the task	You own the system that runs the task

◆ WORTH KNOWING

One scheduled job teaches you more than reading about scheduled jobs will. The morning briefing is the best first one: you already know what good looks like, the inputs are connectors you built in the last chapter, and the output is short enough to check in ninety seconds.

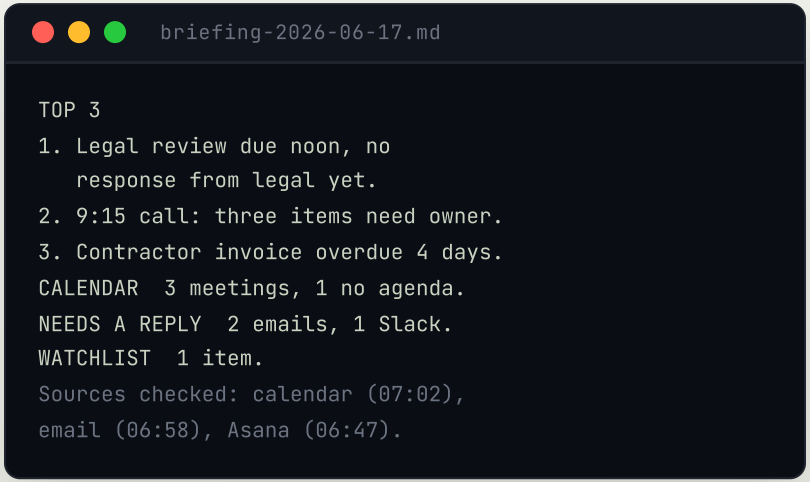
BUILD THE BRIEFING

The prompt is the whole job. Four rules shaped it.

The prompt below is built around four rules, each of which came from a briefing that had to be deleted. Forced ranking. Hard cap at 300 words. A signature line naming every source and its timestamp.

>_ COPY THIS

```
Every weekday at 6:55am, build my morning briefing.
1. FRESHNESS FIRST. Pull calendar, email (last 18 hrs), and
[your tool] (since 6pm yesterday). If any source returns
nothing or is older than 24h, write SOURCE STALE: [name].
Skip it.
2. Write four sections: TOP 3. CALENDAR (flag no-agenda
meetings). NEEDS A REPLY (oldest first). WATCHLIST.
3. Under 300 words. No preamble.
4. Deliver as [draft email to me / briefing-{date}.md in my
Briefings folder].
5. SIGN IT. Last line: Sources checked: calendar
({timestamp}), email ({timestamp}), [tool] ({timestamp}).
6. If any step fails, deliver a short note saying exactly
what failed.
```



WHERE IT RUNS

One question decides the surface.

Three surfaces can hold a schedule. The choice comes down to one question: does this job need your machine to be on?

Cloud Routines run on Anthropic's infrastructure. Your laptop can be in a drawer. For a briefing fed entirely by cloud connectors, this is the right default for most readers. Desktop scheduled tasks run on your machine, which means they inherit its limits: if the lid is closed at 6:55, there is no briefing at 6:55. The terminal path gives the most control and the most plumbing. Start with a Routine.

SURFACE	RUNS WHILE LAPTOP OFF?	BEST FOR
Cloud Routine	Yes	Sources that are all cloud connectors
Desktop scheduled task	No	Sources your logged-in browser can reach
Terminal (OS scheduler)	No	Local files mixed with cloud sources

VERIFIED JUNE 2026

Cloud Routines launched April 2026. They run on Anthropic's infrastructure on a schedule and can also fire from an API call or a GitHub webhook. Plan limits and the minimum interval change, so check your account for the current numbers. Desktop scheduled tasks run without an open session, but only while the machine is on and awake. In the CLI, /loop reruns a prompt on an interval, built for short-horizon watching rather than standing daily jobs. Connector permissions vary, so confirm what each one can do before trusting it unattended; the Gmail connector, for one, has created drafts rather than sending.

FAIL LOUD

Nobody is in the chair. Weld the verify step into the job.

In a live session you are the verifier. You read the output, you catch the weirdness, you ask the follow-up. A scheduled job runs the entire loop with nobody watching. Which means the verify step has to be welded into the job itself.

The briefing prompt does this in three places. Step 1 refuses to summarize any source whose newest item is older than a day. Step 5 signs every briefing with source timestamps. Step 6 tells the job to report its own failure rather than guessing around it. Three blunt checks. Together they let you stop watching.

1. Step 1: refuse to summarize any stale source. Name it in capitals at the top.
2. Step 5: sign every briefing with a sources-checked line and each source's timestamp
3. Step 6: if any step fails, deliver a short failure note rather than a partial briefing

▲ THIS WILL BITE YOU

The most dangerous scheduled job is one that runs perfectly on broken inputs. A dead connector does not announce itself. It returns nothing, and a summarizer with nothing to summarize will report a quiet day, on time, in full confidence. You will believe it precisely because it has been right for weeks. Build the freshness check into the prompt on day one. Read the Sources checked line before you read anything else, every single morning.

TONIGHT

Twenty minutes, six steps.

Somewhere in the second week, the briefing shows up whether you remembered to start it or not. You catch yourself planning the day around a document you never asked for that morning, because you asked for it once, properly, weeks ago. Chapter 8 hires it some colleagues.

1. Pick three sources you already connected: calendar, email, one work tool.
2. Pick the landing zone inside an existing habit. A draft email to yourself is the safest start.
3. Paste the COPY THIS prompt. Swap in your tool names, your times, your delivery choice.
4. Pick the surface. Cloud Routine if every source is a cloud connector. Desktop scheduled task if the job needs your machine. Terminal scheduler if that is where you already live.
5. Schedule it for a weekday time 15 minutes before you normally pick up your phone.
6. Run it once manually today and read the output before you trust a schedule with it.

▸ **DO THIS**

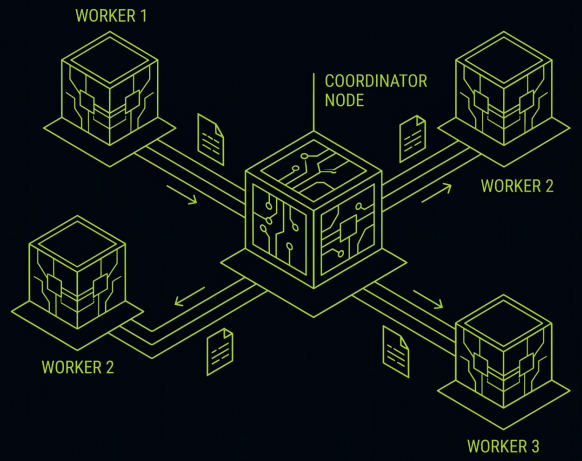
Run step 6 tonight, before you schedule anything. Paste the briefing prompt into a live session, point it at your calendar and email, and read what comes back. A manual run reveals what a scheduled run will miss. Fix it now, while you are in the chair.

PART EIGHT

HIRE A TEAM, NOT AN ASSISTANT

One session can dispatch parallel workers and collect their results.

- What a subagent is
- The hiring metaphor: match the model to the job
- Parallel dispatch and when it helps
- A worked research example
- Red-teaming and the routing close



THE ONE-AT-A-TIME LIMIT

Claude can spin up helpers that work while you wait

When you give Claude a large task, it does not have to work alone. It can spin up separate Claude instances, each with its own context and tools, to handle parts of the task simultaneously. The main session collects their results and produces the final answer.

You still type to one session. Behind it, other instances may already be running. This is what makes research-heavy tasks fast: instead of reading ten sources one by one, ten helpers read them in parallel and report back in the time it takes to read one.

ONE CLAUDE, NO HELPERS	ONE CLAUDE, WITH HELPERS
Reads ten documents in sequence	Reads ten documents at once
All context in one window	Each helper has its own window
Fast on simple tasks	Fast on tasks that fan out
Cost: one model, one task	Cost: depends on which models you assign

◆ SAY IT PLAINLY

Subagent: a separate Claude instance that a parent Claude spins up to handle part of a task. The parent collects the results. The subagent just does its assigned job. Also called a helper or worker in plain conversation.

FOUR EMPLOYEES, FOUR SALARIES

Match the model to the job

Four models are available as of June 2026. They differ in capability and cost. The right call is not always the smartest model. It is the model whose capability matches what the task actually requires. An intern files the paperwork while the director handles the call that needs judgment.

Route your helpers by job type. Data gathering, file reads, and counting go to the intern tier. Analysis and drafting go to the manager or director. Reserve the chief for the hardest reasoning, the final synthesis, or anything a human will read and act on.

ROLE	MODEL	ASSIGN WHEN...
Intern	Haiku	Reading files, counting items, extracting facts, formatting
Manager	Sonnet	Most daily analysis, email drafts, summaries, comparisons
Director	Opus	Complex reasoning, multi-step research, code review
Chief	Fable	Hardest problems, creative strategy, work going straight to a human

VERIFIED JUNE 2026

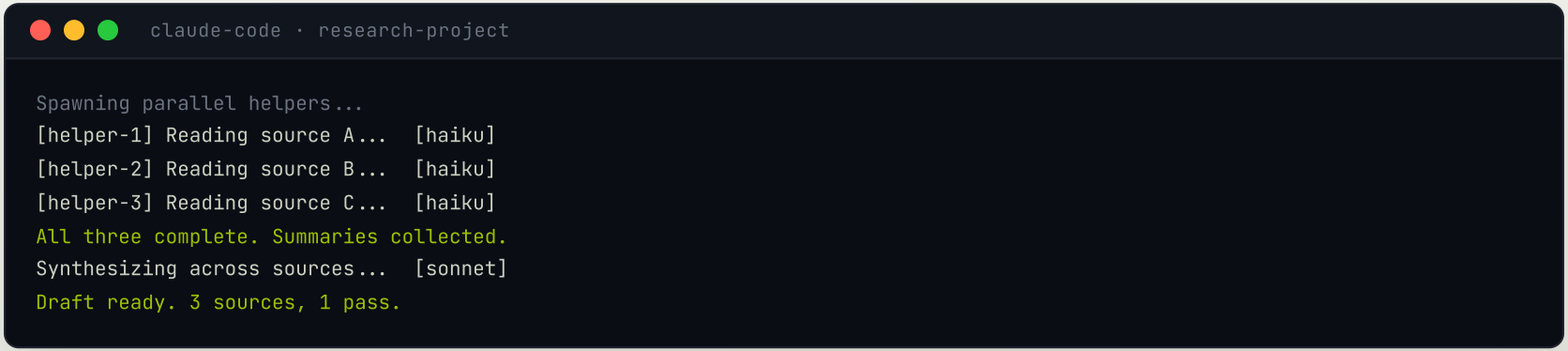
Model pricing per million tokens as of June 2026: Haiku $1 in, $5 out // Sonnet $3 in, $15 out // Opus $5 in, $25 out // Fable $10 in, $50 out. Cost ratio across the four tiers is roughly 1:3:5:10. Full model IDs and plan access at code.claude.com. Note: the chief tier, Fable 5, costs twice as much as Opus and was suspended under a US government directive in June 2026, with Opus as the fallback.

PARALLEL DISPATCH

Fan out, collect, synthesize: the three-step pattern for parallel work

The pattern is always the same. Fan out data-gathering tasks to multiple Haiku helpers running at the same time. Collect their summaries into the main session. Synthesize on a higher-tier model that never had to see the raw sources.

In practice: one helper per document, all reading at once. Each returns a compact summary. The main session sees only the summaries and produces the comparison.



>_ COPY THIS

```
Read [document name] and return: 1) the three main commitments, 2) any pricing terms, 3) any risk flags or conditions. Keep your response under 200 words. Do not include section headers or formatting.
```

SECOND OPINIONS AND THE ROUTING RULE

A second agent reads the first one's output with no attachment to how it was made

A second instance carries no attachment to the reasoning that produced the draft. Ask it to find the problem.

1. Does the task require reading many sources? Fan it out to Haiku helpers, synthesize on Sonnet
2. Is the output going to a human who will act on it? Route the final draft through the Director or Chief tier
3. Is this a high-stakes deliverable? Spin a second Claude to red-team it before it leaves your desk
4. Is the task a simple lookup, a count, or a format? Haiku alone, every time
5. Did you set an explicit model on every helper? If not, go back and set one

▸ **DO THIS**

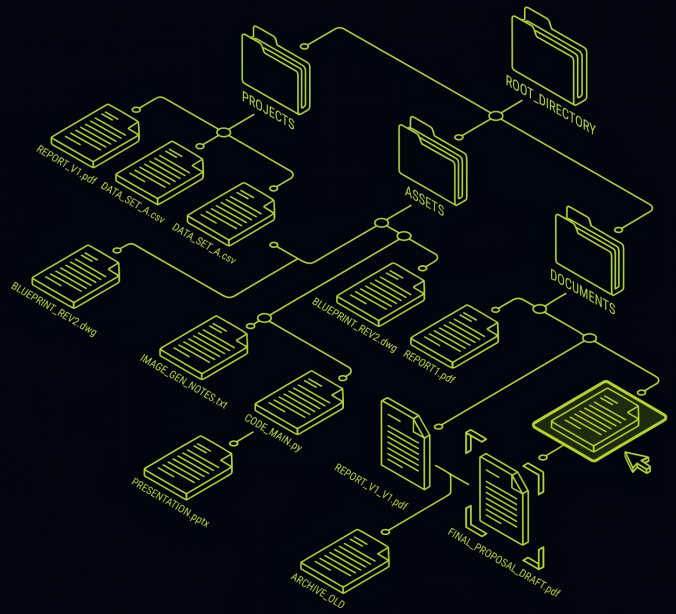
Before your next research or analysis task, label every step with the model it needs: Haiku for data gathering, Sonnet for synthesis, Opus for the final output. Run it once and check the cost in the API dashboard.

PART NINE

THE SAFETY STUFF

Most mistakes here are recoverable. This is how you keep them that way.

- The mental model: recoverable vs. not
- Permission discipline
- Secrets and sensitive data
- Hooks as guardrails
- Verification doctrine



RECOVERABLE BY DESIGN

Most mistakes Claude makes are fixable. The ones that are not all share a pattern: something went out before anyone checked it.

The unrecoverable failures all share one pattern: something was sent, posted, or deleted before anyone looked at it. That is the gap to close. A mechanical one: require approval before any action that cannot be undone.

RECOVERABLE	NOT RECOVERABLE WITHOUT PREPARATION
Wrong file created or overwritten	Email sent to the wrong person
Bad draft that was never sent	Post published to a public channel
Incorrect analysis or summary	Customer data exposed in a prompt
Misformatted output	Credentials pasted and logged

◆ WORTH KNOWING

Claude Code creates automatic checkpoints during a session. Press Esc twice to open the rewind menu, or type /rewind. If a run goes sideways, roll back before it compounds. Plan mode (Shift+Tab) puts Claude into read-only review before any action begins. These are native features: no setup, no extra cost.

PERMISSION DISCIPLINE

Start cautious. Open up only when you understand what you are opening up.

Claude Code asks for permission before it acts. The instinct to click allow everything and get moving is understandable. It is also where most safety problems start. Permissions compound: a broad allowance given once applies to every session that inherits it.

The discipline is simple. Allow once when you want to see what Claude will do. Allow always only after you have seen it work correctly several times.

1. Start in default mode: Claude asks before most actions, you approve each one
2. When a category of action feels routine, allow it once and watch the result
3. If the result is correct and the action is genuinely low-risk, allow always for that category
4. Deny anything you do not recognize. Then ask what it was going to do and why
5. Use plan mode on any task where the scope is unclear: Claude reads and plans, nothing runs

▸ **DO THIS**

Before your next unfamiliar task, press Shift+Tab to enter plan mode. One checkpoint here prevents a half-hour cleanup later.

WHAT NEVER GOES IN

Some categories of information should never appear in any AI prompt.

Claude processes everything you paste. That means everything you paste enters a conversation that travels over a network, gets logged, and may be used by infrastructure you do not control. The categories that cause problems are narrow but serious.

NEVER PASTE THIS	WHY
Passwords, API keys, tokens, credentials	One log line exposes them permanently
Customer names, emails, IDs, or records	Data regulation applies regardless of intent
Regulated personal data (health, financial, legal)	Compliance exposure survives the conversation ending
Internal systems details with access implications	Social engineering surface whether or not you mean it
Confidential contracts or unreleased product details	Disclosure is disclosure

▲ THIS WILL BITE YOU

The most common version of this mistake: pasting a real customer email as an example to show Claude the format. The data is now in the prompt. It does not matter that the task was legitimate. Sanitize examples before you paste them. Replace real names, addresses, IDs, and numbers with clearly fictional stand-ins. Do this once, build the habit, and it stops being a thing you have to think about.

HOOKS AS GUARDRAILS

CLAUDE.md is advice, and hooks are enforcement.

When you write a rule in CLAUDE.md, Claude reads it and tries to follow it. That is advice. It can be ignored, misread, or lost to context pressure in a long session. A hook runs whether Claude remembers the rule or not.

For anything where the rule genuinely matters, hooks are the right tool: preferences go in CLAUDE.md, consequences go in hooks.

>_ WORTH KNOWING

Hooks use exit codes to control what happens:

Exit 0: the hook passed, Claude continues
Exit 2: a blocking error. On a PreToolUse hook this stops the action and feeds the hook's message back to Claude
Any other non-zero code: a non-blocking warning Claude sees and continues past

◆ WORTH KNOWING

The community's practical split: use CLAUDE.md for tone, format, and workflow preferences. Use a PreToolUse hook for anything involving send, post, delete, or publish. If the cost of Claude ignoring the rule is a bad afternoon, put it in a hook.

SHOW ME IT WORKS

Done means you checked it, not that Claude said it worked.

The most common safety failure in AI-assisted work is accepting output without checking it. A successful push command and a working deploy are two different things.

Verification is the discipline of asking one question before approving anything that matters: how would I know if this went wrong? Ask it every time.

TASK TYPE	THE CHECK
Document or report	Open the file and read the first three paragraphs
Spreadsheet update	Spot-check two or three rows against the source data
Email or message draft	Confirm it is saved as a draft, not sent
Scheduled task or automation	Trigger it once manually and read the output
Deploy or publish	Confirm the endpoint or URL returns what you expect
File deletion or rename	Verify the before and after state in the folder

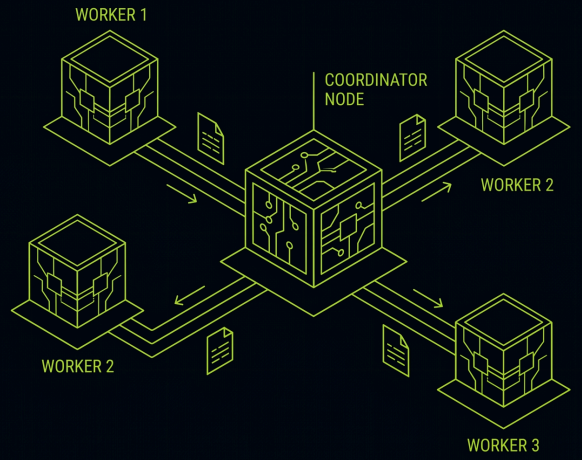
>_ COPY THIS

```
Before we close: tell me what was done, what was sent or published, what was saved to disk, and what still needs human verification.
One sentence per item. Flag anything you are not certain completed correctly.
```

THE PLAYBOOKS

Exact prompts for ten real projects. Pick one and run it this week.

- Spreadsheet to Dashboard
- Marketing Website
- Monday Morning Report
- Competitive Intel Tracker
- Customer FAQ Bot
- Event Landing Page
- Team Wiki
- Lead Scoring
- Finance Tracker
- Email Campaign Builder



PLAYBOOK 01

Spreadsheet to Dashboard

You have a spreadsheet full of numbers and a stakeholder who wants a chart by Friday. Every week the ask is the same and every week you open Figma or Google Sheets charts and spend 45 minutes making something passable. This playbook builds a standalone HTML dashboard that reads your data file and renders the charts automatically. You hand Claude the spreadsheet, describe which metrics matter, and it writes the page. Next week you drop in a new CSV and refresh.

```
claude-code · pipeline-dashboard

> @pipeline-data.csv  Build a dashboard showing
>  leads by week, conversion rate, and top sources.
Reading pipeline-data.csv... 847 rows, 12 columns.
Plan: bar chart (leads by week), gauge (conv rate),
horizontal bar (top 5 sources). Single HTML file.
dashboard.html ready. Open it in any browser.
To update: swap the CSV and reload the page.
```

▶ **DO THIS**

Export your most-used report as a CSV first. Tell Claude which three numbers your audience actually asks about. Let it pick the chart types. You can overrule them after you see the first draft. 45 min · ops, marketing, or finance teams · beginner

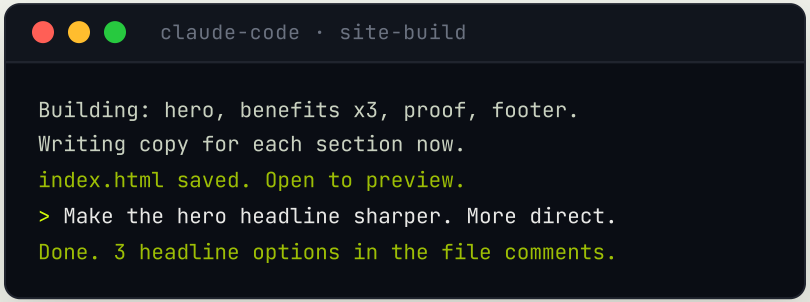
PLAYBOOK 02

Marketing Website

A product or service needs a page, and the agency quote came in at four figures and six weeks. This playbook gets you to a real, deployable site in an afternoon. You give Claude a one-paragraph brief: what you are selling, who it is for, what you want the visitor to do. It builds the HTML, writes the copy, styles it, and saves everything to a folder you can push to any static host. No CMS, no subscriptions, no waiting on anyone.

>_ COPY THIS

```
Build a one-page marketing site for [product].
Audience: [who they are, two sentences].
The one action I want visitors to take: [CTA].
Tone: [punchy / professional / warm].
Sections needed: hero, three benefits, social proof
placeholder, CTA footer.
Save as index.html with embedded CSS. No external
dependencies.
```



◆ WORTH KNOWING

Ask for three headline options before you accept the first one. The third usually beats the first. Copy revision is free; design revision is not. 2-3 hr · marketers, founders, ops leads · beginner

PLAYBOOK 03

Monday Morning Report

Every Monday you compile the same numbers from three different places, write the same summary paragraph, and paste it into the same Slack channel. This playbook builds a script that does it in 90 seconds. Monday morning becomes a button, not a task.

claude-code · monday-report

```
> Build a script that reads this week's numbers
>  from metrics.csv, writes a 3-bullet summary,
>  and saves it to reports/YYYY-MM-DD-weekly.md.
Reading metrics.csv... 6 columns, current week found.
Writing summary: leads, pipeline delta, win rate.
reports/2026-06-09-weekly.md saved.
Run again next Monday. Same command, new date.
```

▲ **THIS WILL BITE YOU**

The first run will probably hit a column-name mismatch or a missing file path. That is expected. Show Claude the error message and it fixes it in one pass. Build this on a Thursday so you have two days to shake out the edge cases before Monday counts. 1 hr setup · ops, RevOps, team leads · beginner

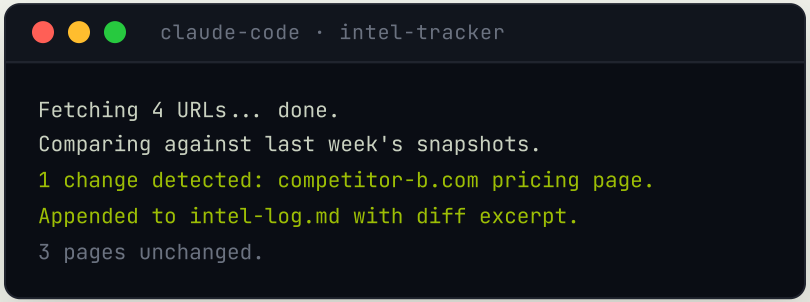
PLAYBOOK 04

Competitive Intel Tracker

A competitor changed their pricing, launched a feature, or updated their website. You found out from a customer. This playbook builds a tracker that monitors a list of competitor URLs and flags changes. You give Claude a list of pages to watch, describe what changes matter (pricing, product names, job postings), and it builds a script that runs on a schedule and saves a diff log. You find out from the tracker, before a customer finds out for you.

>_ COPY THIS

```
Build a competitor monitoring script.
URLs to watch: [list them, one per line].
For each URL, extract: [page title, pricing text, CTA
copy].
Run weekly. Compare against last week's snapshot.
If anything changed, append a dated note to intel-log.md.
Save snapshots to snapshots/YYYY-MM-DD/ folder.
```



◆ WORTH KNOWING

Start with five URLs at most. Pages with heavy JavaScript may need a browser-based fetch instead of a plain HTTP request. Tell Claude if a snapshot comes back empty and it will switch methods automatically. 90 min · product marketing, competitive teams · intermediate

PLAYBOOK 05

Customer FAQ Bot

Your support inbox has the same 12 questions in it every week. This playbook turns your existing documentation into a searchable FAQ bot that lives on a webpage. You paste in your docs or a list of Q+A pairs, and Claude builds a page with a search box that finds the right answer without a server or a subscription. Customers stop waiting for a reply to a question your docs already answer.

```
claude-code · faq-bot

> @faq-source.md  Build a searchable FAQ page.
>  Fuzzy search, instant results, mobile-friendly.
Parsed 34 Q+A pairs from faq-source.md.
Building: search index, result cards, no-match fallback.
faq.html ready. All 34 entries searchable.
To add entries: edit faq-source.md and rebuild.
```

▶ **DO THIS**

Pull the last 30 support tickets and ask Claude to extract the unique questions. That becomes your faq-source.md. You will probably find that eight questions cover 70 percent of the volume. 1 hr · CS, support, product teams · beginner

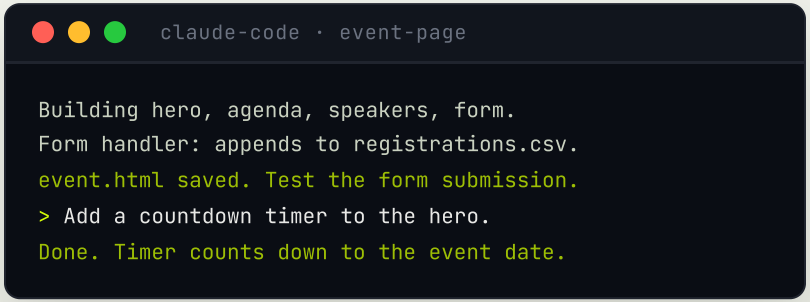
PLAYBOOK 06

Event Landing Page

An event is three weeks out and the landing page is a Notion link someone cleaned up. This playbook builds a proper registration page: headline, agenda, speaker section, a form that collects names and emails to a CSV, and a confirmation message. You give Claude the event details and any design preferences, and it builds the whole thing as a single deployable file. No platform fees, no login walls for your attendees.

>_ COPY THIS

```
Build an event landing page.
Event: [name, date, location or virtual link].
Audience: [who should attend].
Sections: hero with date/CTA, agenda (3-5 items), speaker
bios, registration form.
Form collects: first name, last name, email, company.
On submit: show a thank-you message and append a row to
registrations.csv.
Style: [clean and minimal / bold and dark / your hex
colors].
```



◆ WORTH KNOWING

The CSV form handler works perfectly on most static hosts. If your host does not allow server-side processing, ask Claude to wire the form to a free Formspree endpoint instead. Two-minute swap, no code knowledge needed. 2 hr · event managers, marketers · beginner

PLAYBOOK 07

Team Wiki

The onboarding doc is a Google Doc that has not been updated since the last person quit. Process notes are in someone's Notion. The real answer to most questions is in Slack, if you can find it. This playbook builds a local markdown wiki with a generated index, consistent formatting, and a search page. You paste in the raw content, and Claude structures it, cross-links the pages, and builds the navigation. New hires stop asking the same questions on day three.

claude-code · team-wiki

```
> @raw-docs/  Build a wiki from these files.
>  Add a home page, sidebar nav, and search.
Found 11 files. Categorizing by topic.
Structure: Getting Started, Processes, Reference.
Adding cross-links between related pages.
wiki/ folder ready. Open wiki/index.html.
```

▸ DO THIS

Dump everything into a raw-docs folder first and do not worry about quality. Let Claude sort it. You can delete or merge pages after you see what it generates. Starting with a clean structure is the wrong order. 2-3 hr · ops, HR, team leads · beginner

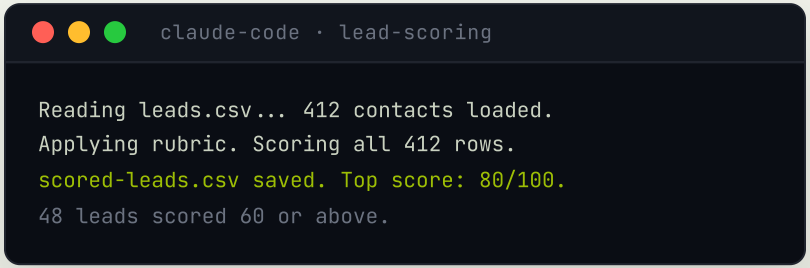
PLAYBOOK 08

Lead Scoring

Your CRM has 400 contacts and your team is calling all of them in no particular order. This playbook builds a scoring script that reads a CSV export of your contacts, applies a rubric you define (company size, title, industry, engagement signals), and outputs a ranked list with a score and a one-line reason for each lead. You spend the next week calling the top 50 and you are calling the right leads first, before you change anything else about your process.

>_ COPY THIS

```
Build a lead scoring script for leads.csv.
Scoring rubric:
- Company size 200-500 employees: +20 pts
- Title contains Director or VP: +25 pts
- Industry is [target industry]: +15 pts
- Last activity within 30 days: +20 pts
- Missing email: -30 pts
Output: scored-leads.csv with columns: name, company,
score, reason.
Sort descending by score.
```



▲ THIS WILL BITE YOU

Run the script and then spot-check the top ten results by hand. If a lead that clearly should not rank highly is showing up near the top, your rubric has a gap. Adjust one rule at a time until the top 20 match your gut. 1 hr · sales, RevOps, demand gen · beginner

PLAYBOOK 09

Finance Tracker

You have a bank export CSV and no clear picture of where the money is going. This playbook builds a personal finance tracker using synthetic categories you define. Claude reads the transaction export, maps each row to a category by merchant keyword, calculates monthly totals, and produces a summary page. Everything runs locally on your machine. No app gets your bank credentials. The output is a clean HTML page and a categorized CSV you own.

```
claude-code · finance-tracker

> @transactions-may.csv  Categorize and summarize.
>  Categories: groceries, dining, transport, bills, other.
Reading transactions-may.csv... 143 rows.
Categorizing by merchant keyword. 6 unmatched rows.
> The 6 unmatched are all from 'ACME MKT'. That is groceries.
Updated. Summary: groceries $612, dining $287, transport $94.
```

◆ WORTH KNOWING

Use synthetic or anonymized data the first time you run this. Export a real file only once you have verified the script does exactly what you expect. The categories and keyword rules become a config file you can reuse every month. 45 min · anyone with a bank export · beginner

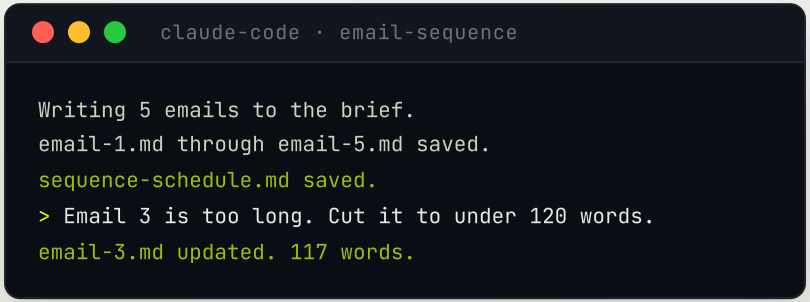
PLAYBOOK 10

Email Campaign Builder

You need five emails for a nurture sequence: a welcome, three value-delivery emails, and a soft ask. Writing all five in one sitting takes most of a day. This playbook gets them done in 90 minutes. You give Claude the audience profile, the product or service, the sequence goal, and the tone. It drafts all five with subject lines and preview text, saves each to its own file, and produces a schedule doc showing which email fires on which day. You review, edit, and hand the folder to your email tool.

>_ COPY THIS

```
Write a 5-email nurture sequence.
Audience: [who they are, what they care about].
Product or service: [one sentence].
Goal of the sequence: [trial signup / booked call /
purchase].
Tone: [conversational / professional / direct].
Email 1: welcome, set expectations.
Emails 2-4: one practical insight each, no sell.
Email 5: soft ask with a single CTA.
Save each email as email-N.md with subject line and preview
text at the top.
Also create sequence-schedule.md: email, send day, goal.
```



▸ DO THIS

Read each email once before it goes out. 90 min · marketers, founders, CS teams · beginner

PLAYBOOK 11

A/B Variant Generator — produce 3 versions of anything, in 30 minutes

Most content gets one shot. AI makes variation free. Every piece you produce should have at least two alternatives to test. This playbook builds that habit.

1. Paste your content piece — landing page, email, ad copy — and your target audience into Cowork or Code.
2. Ask Claude for three distinct variants. Specify the angle: one leading with pain, one with the outcome, one with a credibility signal.
3. For each variant, ask Claude: 'What is this optimizing for?' That sentence becomes your test hypothesis.
4. After results, paste the winner and stats back to Claude: 'What pattern should I apply going forward?' Add the answer to your knowledge file.

◆ WORTH KNOWING

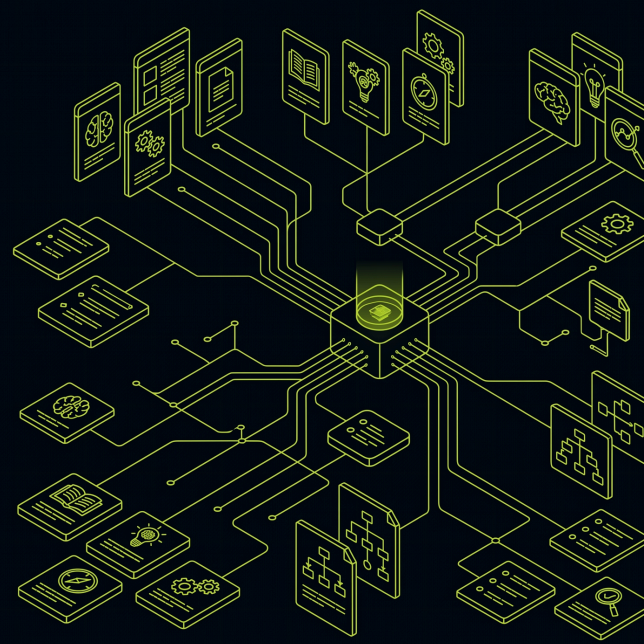
The goal is not just more outputs — it is more data. Each test teaches something the next prompt can use. Over time your knowledge file accumulates patterns that actually work for your audience.

PART ELEVEN

SUPER PROMPTS

Copy-paste prompts that extract value most people never find.

- The self-reflection prompt
- Make it a repeatable skill
- Plan before you build
- Find what you are missing
- Turn messy notes into a decision
- Start the day already oriented



THE MIRROR PROMPT

Ask Claude to critique how you use Claude

Most people never ask their AI assistant to analyze the person giving the instructions. That is the unlocked room. Claude has watched every session: where you stall, where you re-explain the same context, where you ask for things you never use. Ask it to report back.

The prompt below works best after several weeks of consistent use. Run it in a session where CLAUDE.md and auto memory are loaded. The result is not a compliment.

>_ COPY THIS

```
Look at how I have been working with you across our recent sessions. Tell me: the patterns in how I delegate, where I consistently lose time, what I re-explain when I should have written it down once, and three concrete ways I could get more out of you. Be specific and a little blunt.
```

ONCE, NOT EVERY TIME

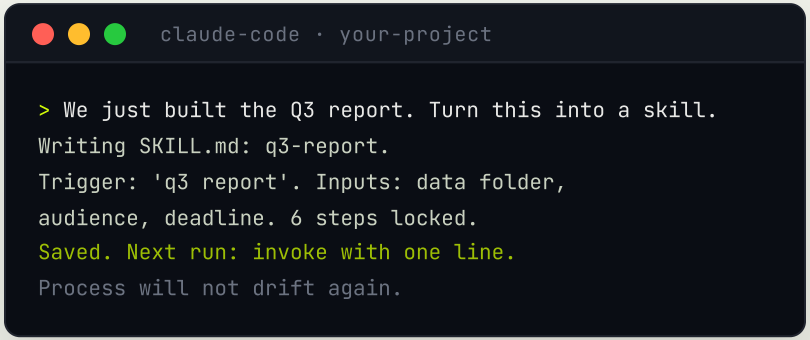
Turn any repeated process into a skill Claude follows automatically

The tax on repeated work is not just time. It is the drift between this week's version and last week's. A skill file eliminates both. The prompt below extracts the procedure from a conversation you already had and writes the file for you.

Paste it after any session where Claude helped with something you will do again.

>_ COPY THIS

```
We just worked through [name the process]. I want to turn this into a skill file so I never have to explain it again. Write a SKILL.md that captures: the trigger phrase I will use to invoke it, the inputs it needs from me each time, the step-by-step process we just followed, and any decisions or constraints I want locked in permanently. Make it specific enough that you can run it cold, six months from now, with no other context.
```



THE CONTRACT BEFORE THE WORK

Get alignment on scope before a single line is written

The most expensive rework happens when the work is done and the goal turns out to have been different from what either party understood. One prompt at the start forces the ambiguity out before it costs anything.

Run this before any task that will take more than one round of back-and-forth. The output is a one-page plan you approve or correct before work begins.

1. What are we actually trying to produce, and for whom?
2. What does done look like, precisely?
3. What constraints are non-negotiable?
4. What have I already ruled out, and why?
5. What would make this a waste of time after the fact?

>_ COPY THIS

```
Before we start: summarize what you understand the task to be, what the output looks like when it is done, the constraints I mentioned, and anything you are uncertain about. Show me this plan in one paragraph. I will correct it before you write a word.
```

THE ADVERSARIAL READ

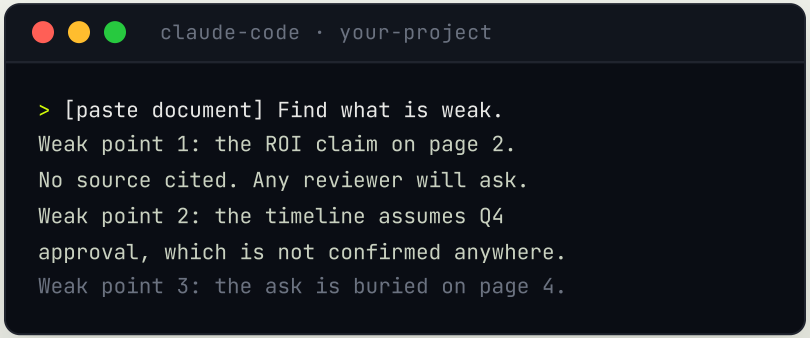
Find the gap in any document or decision before someone else does

Every plan has a weak assumption its author cannot see. Every document has a line that will get challenged in the meeting. The fastest way to find them is to ask Claude to attack the document before you send it.

Paste the document or decision below the prompt. The result is not a list of praise with one mild concern at the bottom.

>_ COPY THIS

```
Read this document as a skeptical reviewer whose job is to
find the three weakest points: the claim with no evidence
behind it, the assumption the whole thing rests on that I
have not stated, and the question a smart reader will ask
that I have not answered. Do not soften the findings. Tell
me what is actually weak, not what could theoretically be
stronger.
```



FROM NOTES TO DECISION

Two prompts that work together: clear the mess, start the day

Raw notes accumulate. They hold useful thinking trapped in noise. The first prompt below extracts the decision from the mess. Paste any rough notes, voice-to-text output, or scratchpad into it.

The second prompt is the start-of-day equivalent: it rebuilds context from whatever files are loaded and tells you what to touch first. Cross-reference Part 5 for the handoff file that makes this prompt land correctly, and Part 7 for the scheduled version.

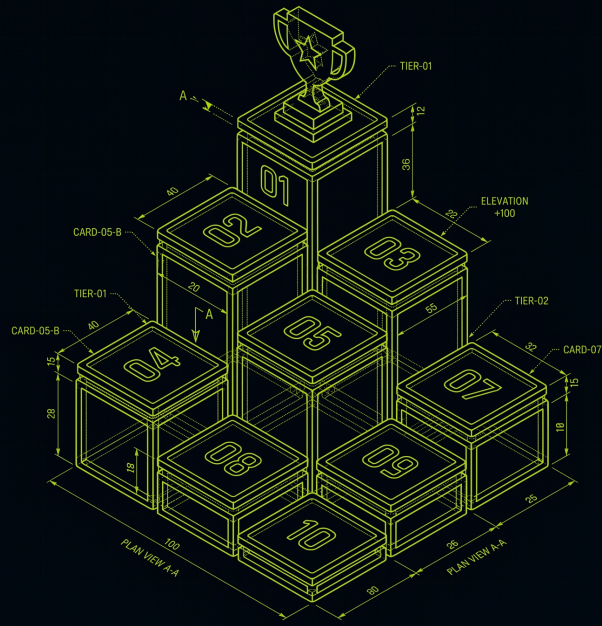
WHEN TO USE	COPY THIS
You have rough notes and need a decision	Here are my raw notes on [topic]. Extract: the actual decision I am facing, the two or three real options, what I already know that should constrain the choice, and what is still missing. Output as a decision brief, not a summary.
Start of day, context loaded from files	Read what is in context. Tell me: the one thing that most needs my attention today, any commitment I made that has not moved, and the first concrete action. Do not give me a full agenda. Give me the first move.

PART TWELVE

OFIR'S TOP 10

Not the ones on every list. The ones people find late and wish they'd found earlier.

- The discipline pair that changes how Claude works before it types a word
- The output tax you didn't know you were paying
- The session failures nobody talks about and the fixes
- The quality gate most setups skip entirely
- The app that makes this guide work for people who avoid terminals



INSTALL THESE FIRST

Three tools that change the foundation

TOOL	BY	THE JACKPOT	INSTALL
superpowers	Jesse Vincent (obra)	Claude's worst habit is starting the moment you describe your goal. This makes it plan first and build second, so sessions stop going sideways before they start.	/plugin install superpowers@claude-plugins-official
token-optimizer	Alex Greenshpun	Your Claude Code bill includes invisible charges from plugins you forgot and instructions that conflict. This finds all of it.	/plugin marketplace add alexgreensh/token-optimizer then /plugin install token-optimizer@alexgreensh-token-optimizer
caveman	Julius Brussee	Claude writes too much: preambles, summaries, a note telling you it's done. Caveman strips all of that. Same answer, a third of the words.	curl -fsSL https://raw.githubusercontent.com/JuliusBrussee/caveman/main/install.sh | bash

▸ DO THIS

Install superpowers and token-optimizer from the official marketplace. Caveman is a community install via the curl command above. Run all three before your next session.

INSTALL THESE WHEN SESSIONS LOSE THE THREAD

Three fixes for failures nobody talks about

The most expensive Claude Code failure is not Claude getting it wrong. It is Claude starting in the wrong direction, losing context mid-session, or forgetting everything by next week.

TOOL	BY	THE JACKPOT	INSTALL
grill-me	Matt Pocock	Before this, you'd describe a project and Claude would immediately start building the wrong thing. With it, Claude asks questions first and waits for your confirmation before touching anything.	npx skills add mattpocock/skills skill=grill-me -y -g
handoff	Matt Pocock	Claude's memory resets when a session ends. Handoff writes a summary before the window closes so you can paste it into your next session and pick up exactly where you stopped.	npx skills add mattpocock/skills skill=handoff -y -g
claude-mem	thedotmack	Every new session, Claude has forgotten your project. Claude-mem remembers your context, preferences, and what you already tried, automatically, without you re-explaining.	npx claude-mem install

◆ WORTH KNOWING

Grill-me and handoff come from the same skills repo. Use grill-me to start any session where direction matters. Use handoff before closing any session you plan to continue.

THE QUALITY AND ACCESS LAYER

Four tools most setups skip

Code-review and commit-commands handle what happens after Claude builds. Claude-md-management and opcode handle the rest.

TOOL	BY	THE JACKPOT	INSTALL
code-review	Anthropic	When Claude says it's done, it might still be wrong. Code-review runs four independent checks and reports back honestly. Like asking a second person to read before you send.	/plugin install code-review@claude-plugins-official
commit-commands	Anthropic	After Claude builds, you still need to commit, push, and open a PR. This collapses those three steps into one command.	/plugin install commit-commands@claude-plugins-official
claude-md-management	Anthropic	CLAUDE.md is your project's instruction file and it gets stale fast. This reads your project and flags what's out of date.	/plugin install claude-md-management@claude-plugins-official
opcode	winfunc	Claude Code runs in a terminal. Opcode wraps the whole thing in a proper app: buttons to click, past sessions in a list, no command line required.	github.com/winfunc/opcode

▲ THIS WILL BITE YOU

Every plugin installed adds its context cost to every session whether you use it or not. More than six or seven active plugins degrades performance noticeably. Install what you use. Audit the rest with token-optimizer.

PART TEN

WHEN THINGS GO WRONG

Hallucinations, context rot, and knowing when to stop.

- Hallucinations and how to catch them
- When to start fresh vs. keep going
- Failure by symptom
- When to stop and ask a human

THE CONFIDENT WRONG ANSWER

Claude will state a fabrication as fact, and its tone will give nothing away

Hallucinations are the most dangerous failure mode because they are invisible. Claude does not slow down, add a hedge, or lower its voice when it is making something up. The tone is identical whether the claim is sourced or invented.

Three categories show up most often: fabricated references (a paper, a link, a statistic that does not exist), wrong facts stated with precision (a price, a date, a name that is close but wrong), and invented file contents (claiming to have read a file, then describing what was not in it). In each case, the failure looks like success until someone checks.

1. For any claim you will repeat to someone else, ask: show me where this came from
2. For any number or date, ask Claude to quote the source sentence exactly
3. For any file Claude says it read, ask it to paste a specific line
4. For any link, open it yourself before forwarding
5. Treat high specificity as a yellow flag, not green

▲ **THIS WILL BITE YOU**

The show-me-where habit feels slow until the day you forward a fabricated statistic to a senior stakeholder. Ask once, verify once. The alternative is explaining later why the number came from nowhere.

START FRESH OR KEEP GOING

The decision depends on what is still load-bearing

Not every problem calls for /clear. Some sessions are worth saving. The question is whether the context still alive in the window is earning its keep.

SITUATION	DO THIS
Single task still in progress, context useful	Keep going
Behavioral tells but task not done	/compact with focus, continue
Task done, new task starting	/clear, new session
Claude is inventing files or decisions	/clear, reload from handoff.md
You cannot tell what Claude thinks the goal is	/clear, restate the goal explicitly
Context window above 80 percent, long work ahead	/compact now, before it gets worse
Responses contradicting each other within same session	/clear, start from a clean brief

WHEN TO STOP AND ASK A HUMAN

Not every failure is a prompting problem

There are categories of problem where continuing without a human makes things worse.

SYMPTOM	WHAT IS LIKELY HAPPENING	DO THIS
Output is technically correct but keeps missing the tone	Voice calibration needs a human example, not more prompting	Paste one example of what right looks like
Claude keeps asking the same clarifying question	The source material genuinely does not contain the answer	Get the answer from a human and paste it in
Three attempts at the same thing, all wrong in the same way	Prompt framing is the issue, not the model	Restate the task from scratch, or ask a colleague to read the prompt
Decision requires approval, policy, or sign-off	This is a governance step, not a research step	Route to the right person before proceeding
Output will reach an external audience and you cannot verify the facts	High-stakes output with no ground truth in the session	Human verification before it leaves the building

▸ DO THIS

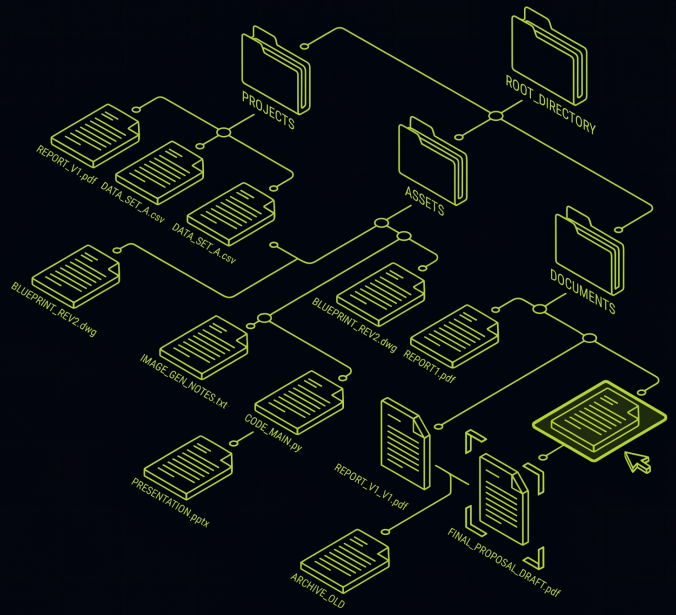
Set a personal three-strikes rule. If a task has failed the same way three times, stop asking Claude to try again. Change the input, the prompt, or the model. If none of those work, the answer is not in the session.

APPENDIX

QUICK REFERENCE

Shortcuts, commands, models, costs, and terms.

- Keyboard shortcuts: Claude Code controls
- Keyboard shortcuts: line editing
- Slash commands: session and work control
- Slash commands: memory, config, and utilities
- Model aliases
- Cost ladder and effort levels
- Glossary
- Pricing and plans
- Skills and hooks



KEYBOARD SHORTCUTS PART 1

Claude Code controls: the screenshot-worthy page

These are the controls you reach for during a session. Option shortcuts on Mac require your terminal to send Option as Meta. In iTerm2: Settings, Profiles, Keys, set Left Option Key to Esc+. In Ghostty, WezTerm, Kitty, Warp, and VS Code terminal, Shift+Enter works without extra setup.

ACTION	KEYS	WHAT IT DOES
Stop Claude mid-response	Esc	Halts the current response immediately
Rewind to prior checkpoint	Esc Esc (input empty)	Opens the rewind menu
Cycle permission modes	Shift+Tab	Cycles: default, acceptEdits, plan
Switch model	Option+P (Mac) / Alt+P	Opens model picker. Press s to apply session-only.
Paste image from clipboard	Ctrl+V (Mac) / Alt+V (WSL)	Pastes an image into the conversation
Redraw screen	Ctrl+L	Fixes garbled terminal display, preserves input
Cancel or exit	Ctrl+C / Ctrl+D	First press clears input. Second exits session.

● VERIFIED JUNE 2026

Confirmed against claude-code-complete-guide.md (v2.1.168) and keybindings documentation at code.claude.com/docs.

KEYBOARD SHORTCUTS PART 2

Line editing and multiline input

These are standard terminal line-editing bindings. They work inside the Claude Code prompt the same way they work in bash or zsh. Ctrl+K and Ctrl+Y are the ones most people do not know they need until they need them.

ACTION	KEYS	WHAT IT DOES
Jump to line start	Ctrl+A	Moves cursor to the beginning
Jump to line end	Ctrl+E	Moves cursor to the end
Delete previous word	Ctrl+W	Deletes one word backwards
New line (most terminals)	Shift+Enter	Multiline input without backslash
Run a shell command	! at the start	Output added to the conversation
File path autocomplete	@ anywhere	Attaches a file to the conversation

◆ WORTH KNOWING

Ctrl+B is the tmux prefix. In a tmux session, press Ctrl+B twice to pass it through to Claude Code. Ctrl+A is the GNU screen prefix and has the same double-press workaround.

SLASH COMMANDS PART 1

Session, context, and work control

Type / to see all available commands. Type / followed by letters to filter. These are the ones you reach for every day.

COMMAND	WHAT IT DOES
/clear	Starts a new conversation. Prior session stays in /resume. Aliases: /reset, /new
/compact [focus]	Summarizes conversation to free context. Pass focus to control what is kept.
/context	Shows how much context you have used and where it is going
/resume	Opens picker to return to a prior conversation. Alias: /continue
/plan [description]	Switches to plan mode. Claude proposes changes without making them.
/rewind	Rolls code and conversation back to a prior checkpoint. Aliases: /checkpoint, /undo
/effort [level]	Sets reasoning effort: low, medium, high, xhigh, max, ultracode
/model [model]	Switches AI model. No argument opens picker.
/usage	Shows session cost and usage breakdown. Aliases: /cost, /stats

SLASH COMMANDS PART 2

Memory, config, and useful utilities

These are the commands that keep the system healthy: memory inspection, config, diagnostics, and the ones that save you a step.

COMMAND	WHAT IT DOES
/memory	View and edit CLAUDE.md files, toggle auto-memory
/init	Generates a starter CLAUDE.md for the current project
/permissions	Manages allow, ask, and deny rules for tool calls. Alias: /allowed-tools
/config	Opens settings: theme, model, output style. Alias: /settings
/skills	Lists available skills. Press t to sort by token count.
/mcp	Views and manages connected MCP servers and OAuth authentication
/copy [N]	Copies last response to clipboard. N picks an earlier response.
/doctor	Diagnoses installation, settings, and MCP connections. Press f to auto-fix.
/status	Shows version, model, account, and connectivity

MODEL ALIASES

What you type vs. what runs

Every alias resolves to a specific model. Type /model followed by the alias to switch mid-session without losing conversation context. Press Option+P on Mac to open the model picker keyboard-first.

ALIAS	RESOLVES TO	CONTEXT WINDOW
haiku	Claude Haiku 4.5	200k tokens
sonnet	Claude Sonnet 4.6	1M tokens
opus	Claude Opus 4.8	1M tokens
fable	Claude Fable 5	1M tokens
best	Fable if available, else Opus	1M tokens

VERIFIED JUNE 2026

Aliases confirmed on CLI v2.1.170. No fableplan alias exists. For Fable-grade planning with cheaper execution: /model fable to plan, then /model sonnet before execution. Switching mid-session warns you because the new model re-reads full history without cached context. Note: as of June 2026 Fable 5 is suspended, so best resolves to Opus 4.8.

COST LADDER AND EFFORT LEVELS

The model you pick is a cost dial

Haiku is 50-60x cheaper per input token than Opus. The right model for reading a file is not the right model for writing a strategy brief. Set effort level with /effort or the --effort flag at launch.

MODEL	INPUT PER MTOK	OUTPUT PER MTOK	USE WHEN
Haiku 4.5	$1	$5	File reads, lookups, counting, data extraction
Sonnet 4.6	$3	$15	Writing, analysis, synthesis: most daily tasks
Opus 4.8	$5	$25	Complex reasoning, strategy, architecture
Fable 5	$10	$50	Flagship tier, twice the price of Opus, availability varies (see note)

VERIFIED JUNE 2026

A note on the flagship tier: Fable 5 ($10 in, $50 out) launched June 9 2026 and was suspended on June 12 2026 under a US government directive, with Opus 4.8 as the fallback. This guide treats Opus as the working ceiling, and the Haiku, Sonnet, Opus ladder covers everything here. Set effort with /effort: low, medium, high, xhigh, max.

GLOSSARY

One line per term. No jargon to explain the jargon.

These are the terms that appear throughout the book. Each one carries a precise meaning in Claude Code.

TERM	WHAT IT MEANS
Agent	A Claude session running autonomously, completing multi-step tasks without constant input
MCP	Model Context Protocol: a standard that lets Claude connect to external tools, databases, and APIs
Skill	A folder with a markdown instruction file that adds a slash command and a reusable workflow
Plugin	A packaged bundle of skills, hooks, settings, and MCP servers installed as one unit
Connector	A pre-built MCP integration for a specific service (Slack, Google Drive, GitHub, etc.)
Context	Everything Claude can see in the current session: files, conversation, instructions, and memory
Token	The unit Claude reads and writes in. Roughly 0.75 words. Pricing is per million tokens.
CLAUDE.md	A plain-text file Claude reads at startup, containing your standing instructions and preferences
Routine	A scheduled cloud task that runs automatically on a cron schedule without a local session

PRICING AND PLANS

What you pay for and where the limits live

Claude Code usage flows through one of two billing paths: a Claude.ai subscription (Max plan) with included usage and rate limits, or direct Anthropic API billing at per-token rates. The terminal and the desktop app share the same usage pool on a subscription plan.

PATH	WHO IT IS FOR	HOW BILLING WORKS
Claude.ai Max plan	Individuals using the desktop app and terminal together	Monthly subscription with included usage. Overage moves to API rates.
Anthropic API (direct)	Teams, automation, production workflows	Per-token, per-model. No monthly floor. Pay for what you use.
Batch API	Any workload processing 20 or more items asynchronously	50 percent discount on per-token rates for async batches

VERIFIED JUNE 2026

Plan names, tier limits, and overage policies change. Check anthropic.com/pricing for current rates. Use /usage inside any session to see your spend and remaining quota in real time. The Batch API 50 percent discount applies to /v1/messages/batches endpoint calls verified as of June 2026.

SKILLS AND HOOKS QUICK REFERENCE

Where they live and how to wire them up

Skills add slash commands. Hooks add automatic behavior at lifecycle events. Both are plain files you control.

THING	WHERE IT LIVES	HOW IT ACTIVATES
Project skill	.claude/skills/skill-name/SKILL.md	Type /skill-name in chat
Personal skill	~/.claude/skills/skill-name/SKILL.md	Available in every project, same invocation
SessionStart hook	settings.json under hooks.SessionStart	Runs once when a session opens. Good for context injection.
PreToolUse hook	settings.json under hooks.PreToolUse	Runs before a tool call. Exit code 2 blocks the action.
PostToolUse hook	settings.json under hooks.PostToolUse	Runs after a tool call. Good for logging or validation.
Stop hook	settings.json under hooks.Stop	Runs when Claude finishes responding. Good for session-end capture.
UserPromptSubmit hook	settings.json under hooks.UserPromptSubmit	Runs on every user message before Claude sees it

>_ COPY THIS

```
Reload skills without restarting: type /reload-skills. List what is loaded: /skills.
```