OPERATIONAL HANDBOOK

Claude

the complete guide

From your first prompt to automation:
Claude Code, Cowork, Skills and the API

Gian Angelo Geminiani

2026 Edition · Rev. 3 · July 2026

Legal notes

Claude: the complete guide — 2026 Edition · Rev. 3 · July 2026

An **independent, unofficial** publication. This book is not produced, authorized, sponsored, endorsed or otherwise connected with Anthropic PBC or its affiliates. The opinions and interpretations are the author's own.

Trademarks. "Claude", "Anthropic", "Claude Code", "Cowork", "Claude Design" and the other product, service, logo and brand names mentioned belong to their respective owners. They are used for **identification and description only** (nominative / fair use), to refer to the products covered by the manual, and do not imply any sponsorship or endorsement. No third-party logo or figurative trademark is reproduced in this volume.

Data accuracy. Information about versions, commands, prices, menus and features refers to fast-moving third-party products: it is provided **"as is"**, verified on the date shown in each chapter and **subject to change** without notice. For up-to-date values, refer to the official sources (claude.com, support.claude.com, code.claude.com, platform.claude.com) and the companion repository.

Contents

F.1 — Preface

Front matter — Level 0. Product details verified on 22/06/2026 against official sources.

Goal

This preface explains who the book is for, how it is organized and how to read it. By the end you will know which level to start from and how to interpret the boxes, tags and figures you will encounter in the chapters.

Who it is for

The book is for anyone who wants to master Claude, from the first message all the way to advanced tasks. It assumes no technical skill in the early levels, and raises the bar as it goes. Three typical profiles:

- those starting from scratch who want to use the chat well;
- those installing and configuring Claude locally (Code, Cowork);
- those who automate, integrate and work in a team.

You don't have to read everything in order. Chapter F.4 offers three reading paths depending on what you need.

What the book is **not**: a generic programming course, a survey of competing products or a price list. When a fact is subject to change (prices, versions), the book flags it and points you to the up-to-date source, instead of faking a stability that isn't there.

Claude is not a single product

The first misconception to clear up: "Claude" is not just the chat in the browser. It is an ecosystem. There's the chat, there's the tool for programming (Claude Code), there's agentic work on your computer (Cowork), there's the design side, and there are the APIs for developers. Understanding how these pieces talk to each other is half the value of the book: chapter F.2 maps it out.

How it is structured

The content is divided into levels of increasing complexity. Each chapter declares its own level, so you know what to expect:

- **Level 1 — Foundations:** using the chat well.
- **Level 2 — Local installation:** Desktop and Code.

- **Level 3 — Daily work:** Cowork, Projects, documents.
- **Level 4 — Design:** from canvas to code.
- **Level 5 — Skills and identity:** giving Claude your voice.
- **Level 6 — Advanced:** automations, integrations, APIs.

Before the levels comes the front matter (this part); at the end, a complete project and the appendices.

How to use boxes, tags and figures

To read without surprises, keep three conventions in mind.

The **boxes** flag information outside the main flow: *Note* (a useful detail), *Warning* (a mistake to avoid), *Tip* (a shortcut).

The **tags** indicate how stable a piece of information is. *(EVERGREEN)* marks concepts that rarely change. *(VOLATILE)* marks facts that may change soon: versions, commands, prices, menu names.

The **figures** are numbered per chapter and always have a short caption and an alternative text, so they remain understandable even to those who use screen readers or read a version without images. The identifier combines the chapter and the sequence: the first figure of ch. L2.1 is Figure L2.1.1, the first table of front matter F.2 is Table F.2.1.

An honest note on a product that changes

Claude evolves fast: new models, new features, menus that move. A printed book risks being born out of date. That's why volatile facts are marked as such and collected, where useful, in an online **companion repo** that accompanies the book with up-to-date commands and errata. When you see a *(VOLATILE)* fact, consider the verification date shown and check the official source if you need the most recent value.

> **Note:** every chapter shows at the top the date on which its product details were verified. It's your reference for understanding how "fresh" a piece of information is.

Summary

1. The book goes from beginner to expert: start from the level that suits you.
2. Claude is not just the chat, but an **ecosystem** of products (ch. F.2).
3. The content is divided into **six levels** of increasing complexity.
4. **Boxes**, **tags** *(EVERGREEN)*/*(VOLATILE)* and **numbered figures** help you find your way through the reading.
5. Facts that change are dated and collected in the **companion repo**: always verify at the source when you need the current value.

Next step

In **ch. F.2 — The Claude ecosystem** we'll see the complete map of the products: what each one does, where it runs and when it's worth using over the others.

F.2 — The Claude ecosystem

Front matter — Level 0. Product details verified on 02/07/2026 against official sources.

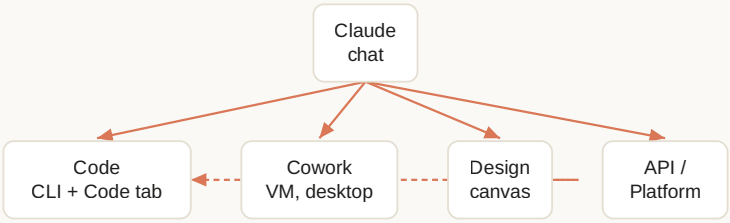
Goal

By the end you will have a mental map of the Claude products: what each one does, where it runs and when it's worth using over the others. It's the compass that orients the rest of the book.

One intelligence, many doors

Beneath the various products is the same engine: the Claude models. What changes is how we access it and what we can have it do. The chat is the main door; the other doors add autonomy (Cowork), code integration (Code), visual work (Design) or programmatic control (API).

Figure F.2.1 — The main doors to Claude. Alt text: vertical diagram with the chat at the top branching into Code, Cowork, Design and API; from Design a handoff goes toward Code.



The products, one by one

Claude (chat). The conversational assistant on web, desktop and mobile apps. It's the starting point for questions, writing, analysis and file generation. Available on all plans.

Claude Code. The agentic coding tool. It lives as a terminal command (CLI) and as the **Code** tab in the desktop app, on the same engine. It's for those who develop software. Requires a paid plan.

Claude Cowork. Brings agentic capabilities into the desktop app for non-programming work: research, documents, file organization. It runs in an isolated VM (a virtual machine separate from your system) and touches only the folders you connect. Desktop only (macOS/Windows), paid plans.

Claude Design. A canvas for creating UI, prototypes and presentations by conversing with Claude. It integrates with Code: when a design is ready, you "hand it off" to development instead of starting over from a screenshot. Beta, paid.

Claude API / Developer Platform. Programmatic access to the models, with SDK and Console. It's for those who integrate Claude into their own software. It's pay-as-you-go.

Add-in and browser. Claude also lives inside Excel/Word/PowerPoint (beta) and as a Chrome extension that acts on web pages (beta, paid).

The newest arrivals (beta). The ecosystem keeps growing. **@Claude (Claude Tag)** brings Claude into Slack as a team member: anyone in the channel can tag it and delegate a task to it (Team/Enterprise). **Claude Science** is a workbench for scientific research, with dozens of integrated databases and toolkits; it's also the first desktop product available on Linux as well as macOS (paid plans). Both are in beta: expect changes.

When to use what

The right question is not "which product is best", but "which is best suited to this task". The table summarizes the choice.

Table F.2.1 — Product, where it runs, when it's worth it.

Product	Where it runs	When to use it
Chat	web/desktop/mobile	questions, writing, analysis
Code	terminal + desktop	software development
Cowork	desktop (VM)	long tasks on your files
Design	web/desktop	UI, prototypes, slides
API	server/code	integrating Claude into your software

> **Note:** chat, Cowork and Code coexist in the same desktop app as three tabs. Switching tabs means switching the way you work, not the application.

How they talk to each other

The real value emerges when the products hand work off to one another. From Design you "hand off" an interface to Code, which actually builds it instead of starting from scratch. Cowork and Code share the same project folder and the same instructions. A Skill written once (Level 5) guides Claude the same way in the chat, in Cowork and in Code. Learning the individual products is useful; learning to make them collaborate is what makes Claude an ecosystem and not a sum of tools.

Summary

1. Beneath everything are the same Claude models: what changes is the door.
2. The **chat** is the general starting point.
3. **Code** is for programming; **Cowork** for autonomous tasks on your files.

4. **Design** covers the visual side and hands work off to Code.
5. The **APIs** are for integrating Claude into your own software.

Next step

In **ch. F.3 — Models and plans** we'll choose the right model and plan: which differences really matter and what each subscription unlocks.

F.3 — Models and plans

Front matter — Level 0. Product details verified on 02/07/2026 against official sources.

Goal

By the end you will be able to distinguish the families of Claude models and understand what each plan unlocks, so you can choose the right combination for you. The specific figures here are volatile: the criteria matter more than the numbers.

The three model families

Claude offers three main families, designed for different balances of intelligence, speed and cost. The name stays (Opus, Sonnet, Haiku), the version number changes over time.

Table F.3.1 — The families and the current version.

Family	Version	When to use it
Opus	4.8	complex reasoning
Sonnet	5	everyday use
Haiku	4.5	quick answers

In short: **Opus** is the most capable, suited to difficult tasks and long agentic work; **Sonnet** is the ideal balance between output and speed, the “everyday” choice (and the model proposed by default on the Free and Pro plans); **Haiku** is the fastest, perfect for short answers and simple tasks.

Above Opus there’s a class of more powerful models (the “Mythos”), designed for the most extreme reasoning and agentic work: **Fable 5** (a

version made more widely available) and **Mythos 5** (with restricted access). Their recent history alone explains why this book thinks in terms of families: launched in June 2026, they were **suspended after three days** by a government directive, then **reactivated** three weeks later, when the directive was lifted. And the return came with its own conditions: Fable 5 is included in paid plans only in part and for a limited period, after which you use it through usage credits (ch. L6.4); it has stricter safeguards than the other models, which sometimes block legitimate requests; and its data follows different retention rules. The moral: flagship models come and go, and each arrives with its own rules. For the current status, refer to the companion repo and the official sources.

> **Warning:** the names, strings and even the *availability* of the models change frequently. Don't memorize a model as "the right one forever": verify the current version when you need it.

How to choose the model

A practical rule that survives version changes: start from Sonnet. If the answer can't handle the complexity of the task, move up to Opus. If you only need speed on something simple, drop down to Haiku. Don't pick the most powerful model "just to be safe": you often pay more without any benefit.

The plans

The plan determines which products you can access and how much you can use them. Beyond the chat, each level adds something.

Table F.3.2 — What each plan unlocks.

Plan	For whom	What it adds
Free	trying out	Web search, files, 5 Projects
Pro	individuals	Code, Cowork, Design, Research
Max	heavy use	5x/20x usage, priority
Team	groups of 5-150	admin, SSO, deploy
Enterprise	companies	controls and security

The **Free** plan covers the chat, web search, file creation, Skills and up to five Projects well, but it doesn't include Code, Cowork or Design. **Pro** is the watershed: it unlocks exactly those three. **Max** doesn't add products but raises the usage limits considerably. **Team** and **Enterprise** add administration, security and centralized management for organizations.

> **Note:** the indicative prices (in USD) were, as of June 2026, around $17-20/month for Pro and from ~$100/month for Max. These are volatile figures: check the up-to-date value at claude.com/pricing.

Common mistakes

- **Choosing Free to use Code or Cowork.** They're not enough: you need at least Pro.
- **Paying for Max without needing it.** Max is for usage limits, not for unlocking new products: if you don't hit Pro's limits, you don't need it.
- **Fixing on a model name.** Versions change; think in terms of families.

Summary

1. Three families: **Opus** (power), **Sonnet** (balance), **Haiku** (speed).

2. Start from Sonnet; move up to Opus if needed, drop down to Haiku for simple tasks.
3. **Free** doesn't include Code, Cowork and Design; **Pro** does.
4. **Max** raises usage limits; **Team/Enterprise** add management and security.
5. Prices and versions are volatile: verify against official sources.

Next step

In **ch. F.4 — Reading paths** you'll choose where to start the book based on what you need, with three ready-made itineraries.

F.4 — Reading paths

Front matter — Level 0. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will know where to start and which chapters to follow based on your objective. The book covers a lot: this chapter saves you from reading everything in order when there's no need.

Read by jumping, not necessarily in a row

The chapters are ordered by increasing level, but you're not obliged to follow them linearly. Those who don't program can skip the development levels; those who develop can move quickly through the foundations. Below you'll find three ready-made itineraries, each designed for a different objective. They all start from the front matter (F) and close with the final project (C.1), so you see a complete case.

The three paths

Table F.4.1 — Three itineraries depending on the objective.

Path	For whom	Itinerary
Cowork only	non-technical	F → L1 → L3 → L5 → C.1
Code/Design	developers	F → L2 → L4 → L6 → C.1
From scratch	first project	F → L1 → L2 → L4 → C.1

Cowork only. For those who want to put Claude to work on their own files without touching the terminal. After the chat foundations (L1), move on to daily work with Cowork and Projects (L3), then to Skills to give Claude your voice (L5).

Code/Design only. For those who develop or design interfaces. Install the local tools (L2), work with Design and the bridge to code (L4), then push into advanced automations and integrations (L6).

From scratch to your first project. For those starting from zero who want to reach a concrete result. Foundations (L1), installation (L2) and the Design part useful for building something real (L4).

> **Tip:** if you don't know where you fit, follow "From scratch": it touches the essential points of every area and leaves you free to go deeper later.

The icon legend

In the chapters, a margin icon flags which path that content is most useful for. They help you find your bearings at a glance.

- **Cowork:** key content for the non-technical path.
- **Code/Design:** key content for those who develop or design.
- **From scratch:** recommended stop for those starting as beginners.

A chapter can have more than one icon: it means it's useful for more than one path. The absence of icons indicates cross-cutting material, valid for everyone.

Jumping without getting lost

Every chapter opens with a **Prerequisites** section: it tells you what to take as given and which chapters to return to if a piece is missing. It's the safety net that lets you jump. If you enter a chapter and the prerequisites are clear to you, go ahead; otherwise you recover only what you need, without rereading everything.

In the same way, the “see ch. X” cross-references are not obligations: they indicate where to find a deeper discussion if you need it at that moment. You can ignore them and come back later.

Where you end up

All paths converge on the **closing** of the book. **Ch. C.1** is an end-to-end project that runs across the levels on a real case: there you see the pieces working together. The **appendices** (ch. C.2) collect a glossary, troubleshooting and official links, to consult when needed rather than read in sequence.

Summary

1. The chapters are by level, but you can read them by jumping.
2. Three paths: **Cowork only**, **Code/Design**, **From scratch**.
3. They all start from F and close with the final project C.1.
4. A margin icon indicates which path a piece of content serves most.
5. When in doubt, follow “From scratch”.

Next step

Begin **Level 1** with **ch. L1.1 — First contact**: open Claude, send your first message and learn how to read an answer.

Chapter L1.1 — First contact

Level 1 — Foundations. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will have opened Claude on the platform you prefer, sent your first message, attached a file, and you'll know how to read an answer with a critical eye. It's the base everything else rests on.

Prerequisites

- A Claude account (see ch. F.3). The Free plan is enough to start.
- An active internet connection.

Where to find Claude

Claude is available on three surfaces, synchronized when you log in with the same account:

- **Web:** go to claude.ai in your browser. It's the fastest way to try it.
- **Desktop:** the app for macOS and Windows (see ch. L2.1 to install it).
- **Mobile:** the apps for iOS and Android, useful for continuing on the go.

Conversations, projects and preferences follow you across devices. You can start a chat on your phone and pick it up on your computer.

Which to choose? To try it right away, the **web** requires no installation: just a browser. The **desktop** app is worth it when you use Claude every day or you need local features like Cowork and Code (Level 2). **Mobile** is handy for capturing an idea on the fly or continuing on the go. It's not

a final choice: the account is the same and you switch from one to another whenever you want.

> **Note:** one exception to syncing are incognito chats (ghost icon): they're not saved and so don't appear on other devices. They're meant for throwaway questions.

The first message

It's straightforward: type in the text box and send. No special command or syntax required. Treat Claude as a competent collaborator who has no context on your work: the clearer you are, the better the answer. We'll go deeper into how to write good messages in ch. L1.2.

A useful first experiment: ask for something concrete, for example "Summarize this text in five points" by pasting in a paragraph. You'll immediately see how Claude structures the answer.

A concrete example

Imagine pasting in the text of an email you received and writing: "Summarize this email in three points and suggest a courteous reply." Claude responds with the three points and a draft ready to adapt. From here you can keep going in the same thread: "Make the reply more formal" or "Add a proposed date for the call." You don't start over: as long as you stay in the same chat, Claude keeps the context of the previous messages.

This is the underlying dynamic: a conversation, not a single command. Each message builds on the previous ones, and that's what makes the chat more useful than a plain search.

> **Tip:** when a result only partly satisfies you, course-correct with a short message instead of rewriting the whole request. Often a single line (“shorter”, “informal tone”) is enough to get where you want.

The interface in brief

The interface is minimal. Around the text box you’ll find a few commands worth recognizing right away:

- The **“+”** button opens attachments (files and photos).
- The **“Search and tools”** menu (sliders icon) gathers tools and styles, like Web search and the response style.
- The **ghost** icon starts an incognito chat: a temporary conversation, not saved in your history.

> **Note:** the exact position of the buttons can change between versions and across web, desktop and mobile. The names and functions, though, stay the same.

Attaching files and images

You can give Claude documents and images to analyze. The supported formats include PDF, DOCX, CSV, TXT, HTML, JSON and others; for images, JPEG, PNG, GIF and WebP. Excel files (XLSX) require that Code execution be active (see ch. L1.3).

Table L1.1.1 — Ways to attach and typical limits.

How	What	Indicative limit
“+” button	files and photos	~500 MB/file
Drag and drop	multiple files	~20 files/chat
Paste	images	from the clipboard

> **Warning:** from documents that aren’t PDFs, Claude usually extracts only the text, not the embedded images. For PDFs, the visual analysis is more complete under ~100 pages. The limits are volatile: verify them if you upload large files.

What Claude does with attachments

Attaching a file doesn’t just mean “feeding it” to Claude: you can ask it to extract data, summarize it, compare it with another document or translate it. The more precise you are about what you want it to do, the more on-target the answer. “Extract the monthly totals from the table” is better than “look at this file.” If a document is long, point to the part that matters: you help Claude focus and you reduce generic answers.

Reading an answer with judgment

Claude is useful, but not infallible. Three healthy habits from day one:

- **Check the facts that matter.** For dates, numbers and names, ask for the source or verify elsewhere. For recent information, turn on Web search (ch. L1.3).
- **Ask it to show its reasoning.** “Explain how you got there” makes the answer more transparent and easier to correct.
- **Iterate.** The first answer is a draft. Refine it with a second message instead of redoing everything from scratch.

- **Be wary of answers that are too confident on verifiable details.** A language model can state a wrong fact with confidence (the phenomenon of “hallucinations”): a confident tone doesn’t guarantee correctness.

It’s not a flaw to fear, but a way of working to learn. Claude is at its best when it reasons, drafts and transforms the material you provide, rather than when you ask it to recall facts “from memory.” For facts that change over time, Web search is the natural complement: it brings up-to-date information with sources.

In practice: your first conversation

1. Open claude.ai (or the app) and log in.
2. Write a concrete request, for example a summary or an email draft.
3. Attach a file with the “+” button and ask Claude to use it.
4. Read the answer and send a second one to improve it.
5. Try the incognito chat (ghost icon) for a throwaway question.
6. Repeat the same request on another surface (web or mobile) and notice that the conversation follows you.

Common mistakes

- **Too vague a request.** “Write me something” produces generic answers. State the purpose, audience and format (ch. L1.2).
- **A file not read the way you expect.** If it’s a document with images, remember that Claude extracts the text; describe in words what matters.
- **Trusting 100%.** Verify sensitive facts; don’t take a date or a number for granted without checking.
- **Using incognito for work you want to find again.** Those chats aren’t saved: if you need to pick them up, use a regular chat.

Summary

1. Claude is on **web, desktop and mobile**, synced with your account.
2. You use it by writing in the text box: no special syntax.
3. Recognize **“+”** (attachments), **Search and tools** (tools/styles) and the **ghost** icon (incognito).
4. You can attach many formats; watch out for the limits and the text-only extraction.
5. Read answers with judgment: verify, ask for the reasoning, iterate.

Next step

In **ch. L1.2 — Talking to Claude well** we’ll see how to write effective prompts, with before/after examples and the most common mistakes to avoid.

Chapter L1.2 — Talking to Claude well

Level 1 — Foundations. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will know how to write requests (prompts) that get useful answers on the first try: clear, with the right context and in the format you need. It's the skill that makes the difference between using Claude and using it well.

Prerequisites

- Having sent your first message (ch. L1.1).

The golden rule

Think of Claude as a new collaborator: skilled, but with no context on your habits. If your request would confuse a competent person who doesn't know your work, it will confuse Claude too. From this come five practical principles.

None of these principles requires technical jargon: they're the same common sense you'd use when delegating a task to a person. The difference is that with Claude the cost of retrying is almost nil, so you can afford to be explicit without fearing you're "asking too much."

1. Be clear and specific

Say explicitly what you want, for whom and with what constraints. Indicate the length, tone and format of the output. When the order matters, use numbered steps. "Write something about the product" leaves too

much room; “Write three lines of description for product X, professional tone, for a technical audience” guides the answer.

2. Give context and motivation

Explaining the why of a request helps Claude get it right. “Summarize this report **for an executive who has two minutes**” leads to a different result than a generic summary. Context isn’t an extra: it’s what guides the choices. Even a constraint, if framed positively, helps: instead of “don’t be technical,” write “use words a customer outside the field would understand.”

3. Use examples

Showing one or two examples of the result you want is one of the most reliable ways to guide format and tone. If you need a product sheet with a certain structure, paste in a finished sheet as a template. Three to five well-chosen examples are worth more than a thousand explanations.

4. Control the format

Say what you want, not what you don’t want. “Reply with a bulleted list of at most five items” is better than “don’t be wordy.” If you want a table, ask for it; if you want JSON or Markdown, specify it. Stating the format saves round trips: if you need a three-column table or a numbered list, say so up front instead of reformatting later.

5. Iterate

The first answer is a draft. Instead of starting over, course-correct: “Shorter”, “More informal tone”, “Add a concrete example”. Refining in two or three passes is faster than searching for the perfect prompt on the first try.

Giving role, structure and constraints

Beyond the five principles, two moves raise the quality on complex requests.

Assign a role. Indicating the point of view to answer from steers tone and depth: “Answer as an auditor” or “Explain it the way you would to a non-technical colleague.” It’s not a gimmick: it narrows the field of plausible answers to the ones you actually need.

Structure long requests. When a request has many parts, separate them: one block for the context, one for the task, one for the desired format. Even just using dashes or sub-headings helps Claude not miss anything. An orderly request produces an orderly answer.

Before and after

The table shows how to go from a weak request to an effective one. The principle is always the same: add purpose, audience and format.

Table L1.2.1 — Same intent, weak prompt and better prompt.

Intent	Weak prompt	Better prompt
Email	“write an email”	“short email, customer, courteous tone”
Summary	“summarize”	“5 points for a busy executive”
Code	“make a function”	“function that validates an email, with tests”

> **Tip:** if you don’t know where to start, write the objective first (“I want to get…”), then the context, finally the format. Three lines are enough.

A complete example

Let’s put the principles together on a real case. Start from a gut-instinct request:

> Write me a post to announce the new product.

It’s vague: no audience, no channel, no tone. Here’s the same request rewritten applying purpose, audience and format:

> Write a LinkedIn post (max 120 words) to announce the launch of product X to professionals in the field. Competent but warm tone, no superlatives. Close with a question that invites a comment.

The second version declares purpose, audience, channel, length, tone and even the closing. Claude has little left to guess, and the first answer is already close to what you wanted: the time spent writing the request well is time you get back on the corrections you won’t have to make.

Iterate or start over?

Iterating in the same chat preserves the context: it’s the right choice when you’re refining a result. It’s better, instead, to open a **new chat** when you change topic entirely, so the old context doesn’t “contaminate” the new answers. Practical rule: same objective, same chat; different objective, new chat.

Breaking up a big task

Huge requests (“write me the complete marketing plan”) produce generic answers, because Claude has to guess too much. It’s better to proceed in stages: first the outline, then one section at a time, finally the overall review. At each step you give a clear objective and check the

result before moving on. This way you keep control and get consistent quality, instead of a wall of text to fix all at once. This applies to texts, analysis and code alike: it's the difference between delegating a project and delegating a step.

Longer doesn't mean better

Being specific doesn't mean writing paragraphs. Often three well-aimed lines — objective, context, format — beat half a page of muddled instructions. Aim for clarity, not quantity: every word that doesn't add a constraint is noise. If you notice you're repeating yourself, cut instead of adding. That's how you know a good prompt: none of its parts can be deleted without losing something.

Common mistakes

- **A generic request.** Without a purpose and audience, the answer is anonymous.
- **Too many objectives at once.** Break complex tasks into steps.
- **Saying only what you don't want.** Indicate the desired result, not the prohibitions.
- **Not giving examples.** When the format matters, a template is worth more than a description.
- **Giving up at the first answer.** Iterate: it's part of the method, not a failure.

In practice: transform a weak prompt

1. Start from a vague request you'd use on instinct.
2. Add **purpose** ("it's for…") and **audience** ("for…").
3. Specify the **format** (length, structure, tone).
4. If you have an example of the desired result, paste it in.

5. Send, evaluate and **iterate** with a targeted correction.

Summary

1. Treat Claude as a competent colleague but without your context.
2. Be **clear and specific**: purpose, audience, constraints.
3. Give **context** and **examples**; they guide more than a thousand words.
4. Control the **format** by saying what you want, not what you avoid.
5. **Iterate**: the first answer is a draft to refine.

Next step

In **ch. L1.3 — Settings and styles** we'll configure tone, preferences and capabilities (Web search, Code execution, Memory) so you don't have to repeat them in every chat.

Chapter L1.3 — Settings and styles

Level 1 — Foundations. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will know how to configure Claude so it answers in your tone and with the right capabilities, without repeating the same instructions in every chat. We'll look at account preferences, Styles and the Web search, Code execution and Memory capabilities.

Prerequisites

- Knowing how to start a conversation (ch. L1.1).

Account-level instructions

In **Settings** → **Instructions for Claude** you set preferences that apply to all conversations: what your name is, what language to write in, the tone you prefer, recurring terms from your work. It's the right place for the things you'd say in every chat: writing them once saves you time.

It's different from an instruction given in a single chat, which applies only there: these apply everywhere. Useful examples to put here: "always write in English", "I prefer concise answers with examples", "I'm a developer", "don't use emoji". Keep them short and update them when your needs change: it's the most convenient level for stable preferences.

Response styles

Styles control *how* Claude communicates. There are a few presets — Normal, Concise, Formal, Explanatory — and you can create custom ones, even by uploading samples of your writing. You'll find them in the "Search and tools" menu next to the text box.

When to use them? **Concise** for quick, dense answers; **Explanatory** when you're learning and want more context; **Formal** for professional texts. The **custom** style is the most powerful: you upload some of your own texts and Claude imitates their register and rhythm. It's the first step toward "making Claude sound like you," which we go deeper into with Skills in Level 5.

> **Warning:** Styles are evolving toward Skills (see Level 5). Features and names may change: it's a volatile area, so check the current state in the settings.

The capabilities to know

Beyond tone, you can turn on capabilities that broaden what Claude can do. The table summarizes them; below, the detail.

Table L1.3.1 — Main capabilities and availability.

Feature	What it's for	Availability
Web search	up-to-date facts	all plans
Code execution	create files	all plans
Memory	remembers history	all plans
Chat search	search past chats	paid

Web search. Turns on live search on the web, with source citations. It's indispensable for recent information, where the model's "from memory"

knowledge isn't enough. It's activated from the "+" button or the tools menu. It consumes your usage limits. On Team and Enterprise it must be enabled by an administrator. In practice: ask "search for the latest news on X" and Claude brings results with the links, so you can trace the source. Without it, it answers with what it "knows" and on recent facts it can be wrong.

Code execution and file creation. Gives Claude a sandbox environment to run code and produce real files: Excel, PowerPoint, Word, PDF, charts. It's also needed to upload XLSX files. It's available on all plans, with an indicative limit of ~30 MB per file. It's what makes requests like "create an Excel with these expenses and a pie chart" possible: Claude writes the file and hands it back to you ready to download.

Memory and chat search. Memory builds a summary of your history, so Claude can draw on past context; it's available on all plans. **Search across past chats**, by contrast, is reserved for paid plans. Incognito conversations (ghost icon) are excluded from both.

Privacy and data control

Three things to keep in mind. **Incognito** chats don't end up in your history or in Memory: use them when you don't want to leave a trace. **Memory** is a summary, not a faithful copy: if a fact is important, repeat it in the chat instead of trusting it was remembered. Finally, on **Team** and **Enterprise** some capabilities are enabled by an administrator from *Admin → Capabilities*: if you don't see them, it's not your fault, ask whoever manages the organization.

> **Tip:** you don't have to turn everything on right away. Start from Web search and Code execution: they cover most everyday cases; you add the rest when needed.

Projects, in a nutshell

A **Project** is a workspace with its own knowledge base and dedicated instructions, shared across the chats of that project. It's useful when you work for a long time on the same topic. Free users can create up to five; sharing is reserved for Team and Enterprise. We go deeper into them in ch. L3.2.

Preferences, Styles or Projects?

They overlap, but they serve different purposes. **Instructions for Claude** are personal preferences that apply everywhere. **Styles** change how Claude communicates and you turn them on as needed. A **Project** isolates context and instructions for a specific topic or client, without polluting your other chats. Practical rule: stable preferences → Instructions; how to write → Styles; a standalone job → Project.

A concrete example: to write this book you could keep a Project with the style rules and the finished chapters, so each new chat starts "up to speed" without your having to repeat the context.

> **Note:** the settings seen here are personal. In an organization, some choices (security, data use for training) are defined by the administrator: your account inherits those rules.

In practice: configure your Claude

1. Open **Settings** → **Instructions for Claude** and write your name, language and tone.
2. Choose a **style** from the "Search and tools" menu (try Concise).
3. Turn on **Web search** and ask a question about a recent fact.
4. In **Settings** → **Capabilities**, verify that **Code execution** is active.

5. Check the **Memory** settings and decide whether to keep it on.
6. If you're in an organization, check in **Admin → Capabilities** what is enabled on your account.

Common mistakes

- **Repeating the same instructions in every chat.** Put them once in "Instructions for Claude".
- **Forgetting Web search on recent facts.** Without it, Claude answers "from memory" and can get dates or numbers wrong.
- **Expecting an Excel file without Code execution.** It must be turned on, otherwise no file generation.
- **Trusting Memory for everything.** It's a synthesis, not a precise archive: for important facts, repeat them in the chat.
- **Thinking it's a bug when a feature is missing.** On Team/Enterprise the administrator may have disabled it: check in Admin → Capabilities.
- **Creating a Project for every question.** Projects are for ongoing work; for an isolated request a regular chat is enough.

Summary

1. **Instructions for Claude** applies your preferences to every conversation.
2. **Styles** control tone; they're evolving toward Skills.
3. **Web search** is for up-to-date facts and consumes usage.
4. **Code execution** enables file creation; **Memory** remembers the history.
5. **Search across chats** is paid only; incognito stays excluded.

Next step

You've completed the foundations. In **Level 2** we move on to local installation: we start from **ch. L2.1 — Installing Claude Desktop**.

Chapter L2.1 — Installing Claude Desktop

Level 2 — Local installation. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will have the Claude Desktop app installed on your computer, you will know how to sign in, and you will understand what its three tabs are for: Chat, Cowork and Code. Claude Desktop is the application that brings Claude straight to the desktop, without going through the browser.

Prerequisites

- A Claude account (see ch. F.3). For **Chat** alone, signing in is enough; for **Cowork** you need a paid plan.
- A **macOS or Windows** computer. There is no Linux app: you use the CLI instead (see ch. L2.2).
- An active internet connection.

System requirements

The app runs on recent systems. On Mac a single build covers both Intel and Apple Silicon processors; on Windows there is a separate version for ARM64.

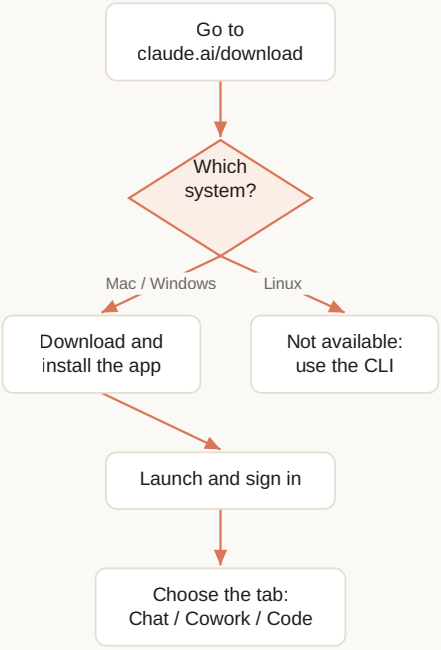
Table L2.1.1 — Supported systems.

System	Minimum version	Notes
macOS	11 (Big Sur)	Universal build
Windows	10	x64; ARM64 separate
Linux	—	none: use the CLI

Download and install

The path is the same for everyone: a download page, a file to open, a sign-in. Figure L2.1.1 sums up the flow, including the Linux case.

Figure L2.1.1 — Claude Desktop installation flow. Alt text: vertical diagram from download to choosing the tab.



> **Note:** the download page detects your system, but you can always pick the macOS or Windows version manually.

The app's three tabs

Once you are in, the app is split into three areas. It helps to know right away which one does what, so you don't go looking for a feature in the wrong place.

Table L2.1.2 — What each tab is for.

Tab	What it is for
Chat	conversations with Claude
Cowork	long agentic tasks and Dispatch
Code	software development in-app

Chat is the starting point. Cowork hands Claude multi-step work on a folder on your computer (we cover it in Level 3); **Dispatch** also starts here (sending a task off so it continues remotely, picked up again in Level 6). Code brings the CLI's capabilities into the app (Level 2 onward).

In practice: install and sign in

1. Open **claude.ai/download** and choose macOS or Windows.
2. **Open the file** you downloaded to complete the installation.
3. Launch Claude from **Applications** (Mac) or the **Start menu** (Windows).
4. **Sign in** with your account.
5. Open the tabs and try Chat with a first message.

> **Tip (Windows + Code):** the first time you open the **Code** tab on Windows you need **Git for Windows** installed. Install it and **restart** the app.

Cowork: what to know first

Cowork is included in the paid plans (Pro, Max, Team, Enterprise) and runs in an **isolated VM** (virtual machine) on your computer. It reads and writes only in the folders you connect, and the network follows your egress (outbound traffic) settings. On Windows you have to enable the **Virtual Machine Platform**.

Common mistakes

- **I'm on Linux and can't find the app.** Correct: it doesn't exist. Use the CLI (ch. L2.2).
- **Windows ARM PC.** Download the dedicated **ARM64 installer**, not the x64.
- **Cowork won't start (Windows).** Check the paid plan and that the **Virtual Machine Platform** is enabled.
- **The Code tab errors on first launch (Windows).** Git for Windows is missing: install it and restart the app.

Summary

1. Claude Desktop is available on **macOS 11+** and **Windows 10+**, not on Linux.
2. You install it from **claude.ai/download**: download, open, sign in.
3. The app has three tabs: **Chat**, **Cowork**, **Code**.
4. **Cowork** requires a paid plan and runs in an isolated VM.

5. On Windows, the **Code** tab requires **Git for Windows** on first launch.

Next step

In **ch. L2.2 — Installing Claude Code** we look at the command-line version (CLI), useful on Linux and for anyone automating, and the installation methods along with their upkeep.

Details verified on 22/06/2026 on support.claude.com (Install Claude Desktop) and code.claude.com/docs/en/desktop. The installation is graphical and tied to the account, so it was not run in the VM; the steps are reported faithfully from the official documentation.

Chapter L2.2 — Installing Claude Code

Level 2 — Local installation. Product details verified on 21/06/2026 against official sources.

Goal

By the end you will have Claude Code installed on your computer, you will know how to choose the right method for your operating system, and you will have verified that it works. Claude Code is the command-line tool (CLI) that works directly on your project's files.

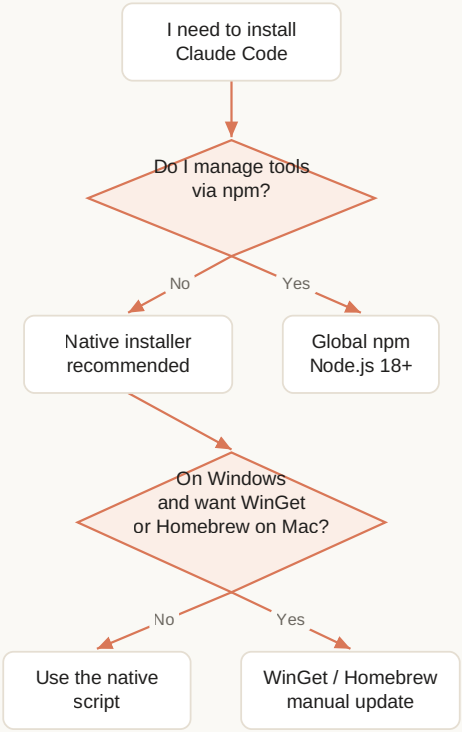
Prerequisites

- A paid account: **Pro, Max, Team, Enterprise or Console (API)**. The Free plan does not include Claude Code (see ch. F.3 and the ledger).
- An open terminal. If you have never used one, see ch. L2.1.
- An active internet connection.

Which method to choose

There are several ways to install, but for almost everyone the choice is simple: the **native installer**. It doesn't require Node.js and updates itself in the background. The other methods are for specific cases.

Figure L2.2.1 — Decision tree for the installation method. Alt text: vertical diagram guiding you through choosing the method.



> **Note:** native installer, Homebrew, WinGet and npm all install **the same binary**. All that changes is how it arrives and how it updates.

Installing with the native installer (recommended)

Open the terminal and run the command for your system.

macOS, Linux, WSL

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

Windows — PowerShell

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

Windows — CMD

The command goes on **a single line**; here it is split with `^` for printing.

```
curl -fsSL https://claude.ai/install.cmd ^
  -o install.cmd && install.cmd ^
  && del install.cmd
```

> **Warning (Windows):** if you see the error *"The token '&&' is not a valid statement separator"* you are in PowerShell, not CMD. The prompt shows `PS C:\` in PowerShell and `C:\` without `PS` in CMD.

On native Windows, **Git for Windows** is optional but recommended: it enables the Bash tool. Without Git, Claude Code uses the PowerShell tool.

Alternative methods

Table L2.2.1 — When to use a different method.

Method	Command	Update
Homebrew (mac)	`brew install --cask claude-code`	manual
WinGet (Win)	`winget install Anthropic.ClaudeCode`	manual
npm	`npm install -g @anthropic-ai/claude-code`	manual

npm needs **Node.js 18+**. Golden rule: **never** `sudo npm install -g`, because it creates permission problems and security risks. To update, use `npm install -g @anthropic-ai/claude-code@latest`, not `npm update -g`.

On Debian/Ubuntu, Fedora/RHEL and Alpine there are also signed repositories (apt, dnf, apk) on `downloads.claude.ai`: useful in enterprises, they update with the normal system package manager.

In practice: install and verify

1. Run the native installer command for your system.
2. **Open a new terminal** (to reload the PATH).
3. Check the version:

   ```
   claude --version
   ```

4. Run the installation and configuration diagnostics:

   ```
   claude doctor
   ```

5. Go into a **real project** folder (not empty) and start it:

   ```
   claude
   ```

   On first launch the browser opens for login: follow the instructions.

Common mistakes

- **`command not found` after installation.** Open a new terminal. The native binary sits in `~/.local/bin/claude`: make sure that folder is in the PATH.
- **Tab/login rejected.** Check that the account is paid: the Free plan doesn't include Claude Code.
- **Permission errors with npm.** Don't use `sudo`. Point the npm prefix to a folder of your own, or use nvm.
- **Git Bash not found (Windows).** Set the path in `settings.json`:

```
{
  "env": {
    "CLAUDE_CODE_GIT_BASH_PATH":
      "C:\\Program Files\\Git\\bin\\bash.exe"
  }
}
```

Summary

1. For almost everyone, the **native installer** is the right choice: no Node, auto-update.
2. Three different commands for macOS/Linux/WSL, PowerShell and CMD.
3. npm, Homebrew and WinGet install the same binary, but update manually.
4. You need a paid account; **never** `sudo` with npm.
5. Always verify with `claude --version` and `claude doctor`.

Next step

In **ch. L2.3 — Authentication and controls** we complete the login, look at using an API key in automated environments, and read the output of `claude doctor` to solve the most common problems.

Commands verified on code.claude.com/docs/en/setup on 21/06/2026. Runnability in the VM: the installation scripts require a network connection to claude.ai and an account, so they were not run here; the commands are reported faithfully from the official documentation.

Chapter L2.3 — Authentication and controls

Level 2 — Local installation. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will know how to sign in to Claude Code with the right method — subscription or API key — understand which credential is active, and diagnose the most common login and PATH problems. This is the step that turns an installation into a tool ready to use.

Prerequisites

- Claude Code installed and working (see ch. L2.2).
- A paid account or an API key from Console (see ch. F.3).

The first sign-in

After installation, go into a project folder and run:

```
claude
```

On first launch Claude Code opens the browser for login (OAuth, delegated sign-in without typing the password in the terminal). Sign in with your Claude.ai account and you return to the terminal already authenticated. If the browser doesn't open on its own, press `c` to copy the URL and paste it manually.

Depending on who you are, you sign in with a **Pro or Max** subscription (Claude.ai account), a **Team/Enterprise** account invited by the administrator, or **Claude Console** credentials if your organization bills via API.

> **Note:** to sign out and re-authenticate, type `/logout` at the Claude Code prompt; `/login` restarts the sign-in.

Two paths: subscription or API key

The two most common modes answer different needs. The **subscription** (browser login) is the normal choice when you work at the computer. The **API key** is for when there is no browser: servers, CI pipelines, automated scripts. There, interactive login is not practical, and you use a key generated in the Console.

Figure L2.3.1 — How to choose the sign-in method. Alt text: vertical diagram that goes from the claude command to three sign-in paths and then to verification with status.

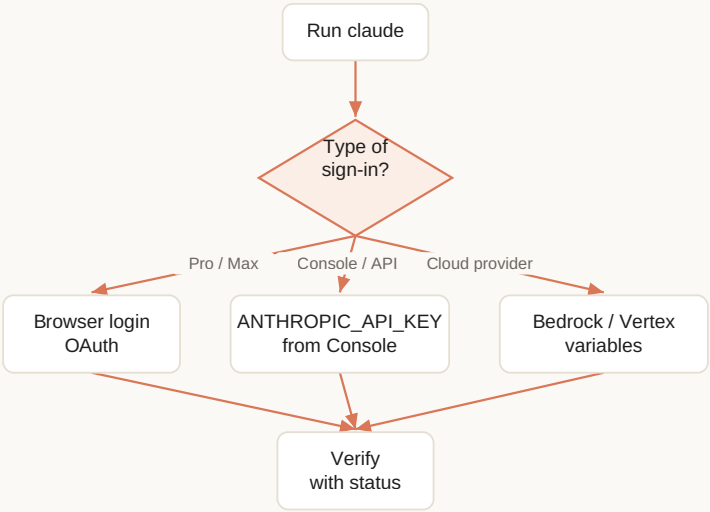


Table L2.3.1 — The authentication methods.

Method	For whom	How
OAuth	use at the computer	browser login
API key	headless / CI	ANTHROPIC_API_KEY
Cloud	Bedrock/Vertex	dedicated variables

Using a headless API key

Generate the key in the Console (platform.claude.com) and set it as an environment variable. On macOS and Linux:

```
export ANTHROPIC_API_KEY=sk-ant-...
```

On Windows (PowerShell):

```
$env:ANTHROPIC_API_KEY = "sk-ant-..."
```

In interactive mode, Claude Code asks you once to approve the key and remembers the choice. In non-interactive mode (`claude -p`) the key is always used, without asking: that is what you need in scripts.

Which credential wins

If multiple credentials are present, Claude Code picks one in a fixed order, from strongest to weakest:

1. cloud provider (`CLAUDE_CODE_USE_BEDROCK` and similar);
2. `ANTHROPIC_AUTH_TOKEN` (bearer for gateways or proxies);
3. `ANTHROPIC_API_KEY`;
4. `apiKeyHelper` script (rotating credentials);
5. OAuth subscription from `/login`.

Knowing the order explains the most frequent trap: if you have an active subscription **but** also `ANTHROPIC_API_KEY` set, the key wins. If that key

belongs to a disabled organization, the login fails even though you have a valid subscription.

> **Warning:** these variables only apply to the CLI in the terminal. Claude Desktop and remote sessions always use OAuth and ignore `ANTHROPIC_API_KEY`.

Where the credentials end up

Claude Code stores the login securely: on macOS in the encrypted **Keychain**; on Linux and Windows in the file `~/.claude/.credentials.json` (on Linux with `0600` permissions), or under `$CLAUDE_CONFIG_DIR` if set. You don't have to manage it by hand, but knowing where it is helps for backup or reset.

In practice: sign in and verify

1. Go into a project folder and run `claude`.
2. Complete the login in the browser (or press `c` to copy the URL).
3. Check which method is active with `/status`.
4. If you need the installation diagnostics:

   ```
   claude doctor
   ```

5. In an automated environment set `ANTHROPIC_API_KEY` and start with `claude -p`.

Common mistakes

- **Login failed with a valid subscription.** Likely an extra `ANTHROPIC_API_KEY`: run `unset ANTHROPIC_API_KEY` and re-check with `/status`.

- `command not found` **after installation.** Open a new terminal; the binary is in `~/.local/bin/claude`, which must be in the PATH (see ch. L2.2).
- **The browser doesn't open at login.** Press `c` to copy the URL and open it manually.
- **API key ignored in the Desktop.** That's normal: the Desktop uses only OAuth.

Summary

1. On the first `claude` you sign in from the **browser** with your Claude.ai account.
2. For **headless** environments you use `ANTHROPIC_API_KEY` from the Console.
3. With multiple credentials a **fixed order** wins: the API key beats the subscription.
4. `/status` tells you which method is active; `/logout` clears the sign-in.
5. `claude doctor` diagnoses installation and configuration.

Next step

In **ch. L2.4 — Configuring the project** we prepare `CLAUDE.md`, the `.claude` folder and the permissions, to start Claude Code already "on the ball" on a real repo.

Details verified on 22/06/2026 on code.claude.com/docs/en/authentication and support.claude.com (troubleshoot Claude Code). OAuth login and API key require a real account, so they were not run in the VM; commands and variables are reported faithfully from the official documentation.

Chapter L2.4 — Configuring the project

Level 2 — Local installation. Stable concepts; product details verified against official sources.

Goal

By the end you will know how to prepare a project folder so Claude Code works well in it: writing a `CLAUDE.md` file that gives the context, understanding what the `.claude/` folder is for, and setting permissions so Claude acts without asking you for confirmation at every step, but not behind your back either.

Prerequisites

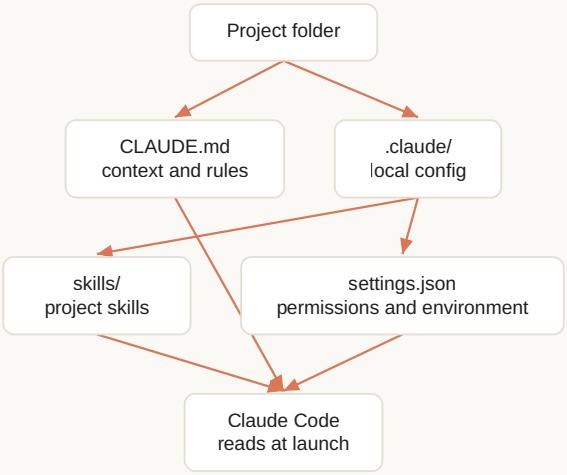
- Claude Code installed and working (see ch. L2.2).
- Login completed (see ch. L2.3).
- A **real** project folder: code, documents, anything you want to work on. Claude Code performs better on a real project than on an empty folder.

Why configuring matters

Claude Code starts out knowing nothing about your project. Every time you open a session, it rebuilds the context by reading the files. You can explain everything by voice at every launch, or write it once in a file it reads on its own. The second path is the one that pays off: fewer repetitions, more predictable behavior, and the same rules apply to anyone who opens the project.

Three elements make up the configuration: the `CLAUDE.md` file (the context), the `.claude/` folder (skills, commands, local settings) and the permissions (what Claude can do without asking).

Figure L2.4.1 — What Claude Code reads at launch in a project folder. Alt text: vertical diagram showing the project folder with CLAUDE.md and the .claude subfolder and their role.



The CLAUDE.md file

`CLAUDE.md` is a text file in Markdown format that you put in the project root. Claude Code reads it at every launch and treats its content as instructions to follow. It's the right place for the things you don't want to repeat:

- **What the project is:** a sentence or two on purpose and structure.
- **Commands you use often:** how to start it, how to test it, how to build it.
- **Conventions:** code style, names, things not to touch.

- **Constraints:** “don’t modify folder X”, “comments in English”.

The rule is: little and useful. A huge `CLAUDE.md` gets diluted; a focused one counts on every reply. Write the instructions the way you would tell them to someone new on the team on their first day.

> **Tip:** inside Claude Code, the `/init` command analyzes the project and proposes a first `CLAUDE.md`. It’s a good starting point to refine by hand, not a finished result.

A minimal example:

```
# Project: orders API

Node.js app. Start: `npm run dev`.
Test: `npm test`. Don't modify `db/migrations/`.
Code comments in English.
```

The .claude/ folder

Alongside `CLAUDE.md` you can have a `.claude/` folder. It gathers the project’s local configuration:

- `.claude/skills/` — the skills specific to this project (see Level 5).
- `.claude/settings.json` — the project’s permissions and environment variables.

The advantage is that this configuration lives **inside** the project: if it’s in a git repository, you share it with the team and version it like the rest. Whoever clones the project inherits the same rules.

Permissions: the happy medium

Claude Code asks for confirmation before actions that change something: modifying a file, running a command. It’s a safeguard, but con-

firming everything slows you down. Permissions are there to say in advance “these things you can do without asking, these never”.

You set them in `.claude/settings.json`. The basic schema distinguishes what is always permitted (`allow`) from what is always denied (`deny`):

```
{
  "permissions": {
    "allow": ["Bash(npm test)"],
    "deny": ["Bash(rm -rf *)"]
  }
}
```

The philosophy: grant the repetitive and safe actions (running tests, reading files), keep confirmation for those that move or delete. Don’t open everything “for convenience”: the point of the confirmation step is that you remain the last word on the actions that matter.

In practice: prepare a folder

1. Open the terminal and go into your project:

   ```
   cd ~/projects/my-app
   ```

2. Start Claude Code:

   ```
   claude
   ```

3. Generate a context draft:

   ```
   /init
   ```

4. Open `CLAUDE.md`, cut the superfluous, add the commands you use and the things not to touch.

5. If you have rules about recurring commands, create `.claude/settings.json` with the `allow` / `deny` permissions.
6. If the project is in git, make a commit: now the configuration travels with the code.

Common mistakes

- **CLAUDE.md too long.** It becomes noise. Keep only what changes behavior: purpose, commands, constraints.
- **Permissions too broad.** Putting everything in `allow` removes the safety net. Grant the repetitive and safe, not the destructive.
- **Configuration outside the repo.** If `.claude/` and `CLAUDE.md` aren't versioned, the team doesn't inherit them and the rules apply only to you.
- **Starting from an empty folder.** Claude Code gives its best on a real project. To learn, use a repo you already know.

Summary

1. `CLAUDE.md` is the context Claude Code reads at every launch: purpose, commands, conventions, constraints.
2. Keep it short and focused: little and useful beats long and generic.
3. The `.claude/` folder gathers the project's local skills and settings.
4. Permissions (`allow` / `deny`) balance speed and control: grant the safe, confirm the destructive.
5. Version `CLAUDE.md` and `.claude/` in the repo: that way the rules apply to everyone.

Next step

With the local installation complete, **Level 3 — Daily work** gets into everyday use. We start with **ch. L3.1 — Cowork: first steps**, where you delegate a task on a folder to Claude and learn to approve its actions.

Concepts (CLAUDE.md, permissions, project structure) verified on code.claude.com/docs on 24/06/2026. The `/init` command and starting `claude` require an active account and were not run here.

Chapter L3.1 — Cowork: first steps

Level 3 — Daily work. Product details verified on 27/06/2026 against official sources.

Goal

By the end you will know what Cowork is and how it differs from Chat and Claude Code, how to start a task on a folder of your computer, how to describe a result instead of a sequence of steps, and how to approve the actions Claude proposes while it works.

Prerequisites

- Claude Desktop installed (see ch. L2.1).
- A paid plan: **Pro, Max, Team or Enterprise**. Cowork is not included in the Free plan (see ch. F.3 and the ledger).
- A folder with files it makes sense to work on.

What Cowork is

Cowork is the desktop app's tab for **agentic non-code work**: you hand it a task on a folder and it carries it forward on its own — reads the files, reasons, creates or modifies documents, runs steps in sequence — stopping to ask you for confirmation when needed. It is available on the paid plans on macOS and Windows. (VOLATILE: the product's status evolves fast; see the ledger.)

It runs in an **isolated VM** (a virtual machine, a computer inside the computer) on your device. This is the key point for trust: Cowork reads and writes **only in the folders you connect to it**, and the network follows

the egress settings (what it can reach on the internet). It doesn't see the rest of your disk unless you give it to it.

Cowork, Chat and Code: who does what

The app's three tabs answer different needs. Choosing them well is half the job.

Table L3.1.1 — When to use which tab.

Tab	For what	Form
Chat	Questions, drafts, ideas	Back and forth
Code	Software development	On code files
Cowork	File tasks, multi-step	Agentic, long

In short: if you want an answer, Chat. If you're programming, Code. If you want someone to **do** an involved thing on your files — reorganize a folder, produce a report from several documents, prepare a series of files — Cowork.

End-state, not the steps

The most common mistake on first use is to guide Cowork step by step, the way you do in chat. The opposite works better: describe the **result you want** (the "end-state") and let it find the sequence.

- **Weak (steps):** "Open the file. Now read column B. Now copy…"
- **Better (end-state):** "From these three sales Excel files, create a summary by region with quarterly totals, saved as `summary.xlsx`."

Describing the result plays to what Cowork is made for: planning the path. You define the "where to get to" and the constraints; it finds the "how".

Approving the actions

While it works, Cowork stops before the actions that matter and shows you what it's about to do: create a file, modify one, run a command. You approve or correct. It's the same principle as Claude Code's permissions (see ch. L2.4): the repetitive part flows, the decisions stay yours.

Don't approve sight unseen. Read what it proposes, especially the first few times: that's how you learn to trust it — or to understand where you need to be more precise in the request.

Cowork has two **permission modes** that decide how often it stops:

- **Ask before acting:** it stops and asks for confirmation at every action. It's the right mode with new files or tools, or when you want to keep everything under control.
- **Act without asking:** it proceeds without stopping. Faster, but riskier: use it only while you are supervising, on files and sites you trust.

In **both** modes, Claude always asks for confirmation before **deleting** files permanently: deletion is never automatic.

In practice: your first task

1. Open Claude Desktop and go to the **Cowork** tab.
2. **Connect a folder**: choose one with real files, but of which you have a copy. Cowork will act only inside this folder.
3. Write the task as an **end-state**, with the constraints:

```
In this folder there are scattered .txt notes.
Group them by topic into subfolders and
create an INDEX.md listing what's where.
```

4. Let it plan. When it stops for an action, **read and approve**.

5. At the end of the task, check the result in the folder. If it's not what you wanted, refine the request and run it again.

Common mistakes

- **Guiding it step by step.** Wastes its strength. Describe the result, not the procedure.
- **Connecting the entire disk or critical folders.** Connect only what's needed, and work on copies until you trust it.
- **Expecting Cowork in the Free plan.** It needs a paid plan.
- **Approving without reading.** The first few times, check what it proposes: that's where you learn to give it better instructions.

Summary

1. Cowork is the desktop tab for agentic non-code work, in an **isolated VM** that sees only the connected folders.
2. Chat for answers, Code for software, Cowork for multi-step tasks on your files.
3. Describe the **result** (end-state) and the constraints, not the sequence of steps.
4. Approve the actions Cowork proposes: control stays yours.
5. Connect only the necessary folders and start from copies.

Next step

In **ch. L3.2 — Projects** we look at how to give recurring work a stable home: instructions, reference files and memory that persist from one session to the next.

Details on Cowork (isolated VM, plans, connected folders, permission modes) from the ledger, verified on 27/06/2026 on support.claude.com/en/articles/13345190 and code.claude.com/docs. The example task was not run here: it requires the desktop app with a paid account.

Chapter L3.2 — Projects

Level 3 — Daily work. Product details verified on 22/06/2026 against official sources.

Goal

By the end you will know what a Project is, when it’s worth creating one, how to give it instructions and reference files that persist from one conversation to the next, and what changes when you work in a team. We’ll use as an example the Project this book was written with.

Prerequisites

- Knowing how to converse in chat (see ch. L1.2).
- A recurring activity: something you come back to several times, not a one-off question.

What a Project is

A Project is a **workspace** dedicated to an activity: it gathers in one place the instructions that always apply, the reference files (the “knowledge base”) and the linked conversations. Every chat opened inside the Project starts already with that context, without you repeating it.

You’d already come across them in passing in the settings (ch. L1.3): here we put them to work. Projects are available on all plans; on the **Free** plan they are limited to **5**. Team sharing is reserved for **Team and Enterprise** (see the ledger).

When to create one

Not everything deserves a Project. The single question fits fine in a normal chat. It's worth creating a Project when these signals recur:

- **You come back several times** to the same work over time.
- **You repeat the same context** in every chat ("remember the style is…").
- **You have reference files** needed in every conversation.

If you find yourself pasting the same instructions or the same documents at the start of every chat, it's time for a Project.

Instructions and knowledge base

Two things make a Project useful:

- **Instructions:** they apply to every chat in the Project. Tone, constraints, format, things to do and not to do. It's the "who you are and how you work" of the project.
- **Knowledge base:** the files Claude can consult — documents, examples, reference material. They become part of the available context.

The difference from `CLAUDE.md` (ch. L2.4) is the context of use: `CLAUDE.md` governs Claude Code on the project's files; the Project instructions govern the **conversations** inside that Project. Same idea — write the stable context once — applied to different tools.

> **Note:** the knowledge base files have a per-file limit (around 30 MB) but no binding limit on number, as long as the content fits in Claude's context. It's a different limit from chat attachments (ch. L1.1): the knowledge base is designed for many reference files, not for a few huge files.

Memory and scope

The Project gives your work a boundary. The instructions and files stay **confined** to that Project: they don't mix with the other chats. This is a double advantage — the right context is always there, and the wrong one stays out. When you switch activities, you switch Project, and the model doesn't drag along material that doesn't belong.

Example: this book's Project

This manual was written inside a Project. Its shape shows the pattern well:

- **Instructions:** project rules — where to save the chapters, no cautionary notes in the published texts, voice and page-layout constraints.
- **Knowledge base:** the editorial plan, the facts ledger, the context files on voice and audience.
- **Conversations:** one per block of chapters, all already aligned to the same rules without repeating them.

Result: every writing session starts knowing what the book is like. It's the difference between explaining the project from scratch every time and having written it once.

In practice: create your Project

1. In Claude, open the **Projects** section and create a new one.
2. Give it a name that says what it's about.
3. Write the **instructions**: the stable context, the constraints, the format you want.
4. Upload the recurring reference files into the **knowledge base**.

5. Open a chat inside the Project and work: the context is already active.
6. When a rule recurs across several chats, move it into the Project's instructions.

Common mistakes

- **A Project for every chat.** They're for recurring work, not the single question. Too many Projects become clutter.
- **Instruction-floods.** It applies as with `CLAUDE.md`: little and focused. Excess gets diluted.
- **Stale knowledge base.** Reference files age. Update them, or Claude works on old material.
- **Expecting sharing on Free/Pro.** Sharing a Project in a team is Team/ Enterprise.

Summary

1. A Project is a workspace with instructions, reference files and linked chats, all in one place.
2. Create it when you come back to a piece of work several times and repeat the same context.
3. The instructions apply to every chat in the Project; the knowledge base is the material Claude consults.
4. The scope keeps the right context in and the irrelevant out.
5. Free up to 5 Projects; team sharing on Team/Enterprise.

Next step

In **ch. L3.3 — Connectors** we connect Claude to the apps you already use — Drive, Slack, task managers — so it can read your data and act within them.

Details on Projects (Free limits, team sharing) from the ledger, verified on 22/06/2026 on support.claude.com. The example of this book's Project is real; the operational steps were not run here.

Chapter L3.3 — Connectors

Level 3 — Daily work. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know what Connectors are, how to link one from the directory, how to activate them in a conversation, and what precautions to keep in mind — both as a user and, in a company, as an administrator.

Prerequisites

- Knowing how to open a chat (see ch. L1.1) and use the tools menu (see ch. L1.3).
- An account on a service you want to connect (Drive, Slack, a task manager…).

What connectors are

A connector gives Claude access to an app or service: reading your data and **performing actions** inside it. It links Claude to Linear to open issues, to Slack to send messages, to Google Drive to search your files.

The point to grasp clearly is the permission model: **Claude inherits your permissions** in the connected service. If you can't see a given file, channel or record at the source, the connector won't reach it from Claude. A connector doesn't give you more access than you already have — it only brings it into the conversation.

Web connectors are available to **all users** on Claude, Cowork, Desktop and Mobile; they also work with Claude Code and via API.

The connectors directory

The available connectors live in the **Connectors Directory** (claude.ai/connectors). Each connector has a card with use cases, read/write capabilities and availability. Some carry the **Interactive** badge: they render live interfaces — dashboards, boards, tools — inside the conversation.

Open the directory in two ways:

- **From a chat:** the **"+"** button (or "/") > "Connectors" > "Manage connectors" > "+".
- **From settings: Customize > Connectors** > "+".

Connecting and activating a service

Connecting a service and using it are two distinct steps.

Connecting (once): from the directory, click the connector, review its capabilities, "Connect"/"Install", follow the authentication prompts (usually OAuth, meaning you authorize Claude from the service's site), configure the permissions.

Activating (per conversation): "+" (or "/") > "Connectors", then toggle on the services you want for that chat. From there Claude uses them when relevant — and it can propose them on its own, without you having to name them every time.

> **Tip:** if you have many connectors, from "Connectors" > "Tool access" choose how they load. The default **Auto** is fine for almost everyone; with 10 or more active connectors, **On demand** leaves more room for the conversation.

In a company: the admin's role

On **Team and Enterprise** a connector isn't usable until an **Owner** or **Primary Owner** enables it for the organization (Organization settings > Connectors > "Browse connectors" > "Add to your team"). Enabling does **not** give anyone access: everyone still authenticates on their own.

The admin can also **restrict the actions** of a connector for the whole org. From Customize > Connectors, on the connector, the **Tool permissions** are split by type (read-only, write/delete) and for each you choose *Always allow*, *Needs approval* or *Blocked*. It applies to everyone and can't be bypassed by an individual.

Table L3.3.1 — Examples of action restrictions.

Service	Allow	Block
Email	Search and summarize	Send messages
Drive	Read files	Create/edit
Linear	View issues	Create or change them

These restrictions work **together** with the service's permissions: even where Claude can write, you still need permission at the source. Restricting in Claude never grants more than the source system allows — at most it restricts.

Precautions

- **Connect only what you trust and what you need.** Linking a service means giving Claude access to your data in it.
- **Review the requested access** during connection, and **disconnect** what you no longer use (Customize > Connectors).
- **Custom connector = not verified by Anthropic.** You can add custom connectors (remote MCP), but only connect servers from trusted

organizations. We go into custom connectors in detail at Level 6 (ch. L6.2).

- **Team/Enterprise:** connectors only work in private Projects, and chats with synced content can't be shared.

In practice: connect Google Drive

1. In chat, "+" (or "/") > "Connectors" > "Manage connectors" > "+".
2. Search for the connector in the directory and open it.
3. Read the capabilities, then "Connect" and authorize with your account (OAuth).
4. Go back to chat, "+", "Connectors", and toggle on the service.
5. Try it:

```
Search my Drive for the March quote
and summarize the main line items for me.
```

Common mistakes

- **"I connected it but it isn't using it."** Connecting isn't enough: toggle it on in the chat's "Connectors" menu.
- **Expecting access to everything.** Claude only sees what you see in the service.
- **(Team) "The connector isn't there."** An Owner must enable it for the org first; then you authenticate.
- **Custom connector that times out.** The MCP server is reached from Anthropic's cloud, not from your device: it must be on the public internet (see ch. L6.2).

Summary

1. A connector gives Claude access to an app to read data and act, **with your own permissions** at the source.
2. Connectors live in the Connectors Directory; some are **Interactive**.
3. Connecting (OAuth, once) and activating (toggle, per conversation) are two distinct steps.
4. In Team/Enterprise an Owner enables the connector and can restrict its actions for the whole org.
5. Connect only what you trust, review the access, disconnect the useless.

Next step

In **ch. L3.4 — Documents** we use Claude to generate professional files — Word, PowerPoint, Excel, PDF — starting from the right content.

Connector data (directory, OAuth, admin role, action restrictions) from the ledger, verified on 24/06/2026 on support.claude.com/en/articles/11176164. The operational steps were not executed here.

Chapter L3.4 — Documents (Word, PowerPoint, Excel, PDF)

Level 3 — Daily work. Product details verified on 22/06/2026 against official sources.

Goal

By the end you'll know how to have Claude generate professional documents — Word, Excel, PowerPoint, PDF — by asking for them the right way, applying the "content first, then the file" principle, and starting from a template when you want a repeatable result.

Prerequisites

- Knowing how to write clear requests (see ch. L1.2).
- Having **Code execution / file creation** active (see ch. L1.3): it's the sandbox Claude uses to create files. It's on every plan.

How Claude creates files

Claude doesn't "hand-write" a .docx: it uses a code execution sandbox that generates the actual file, with the formatting needed. That's why the documents produced are real files — openable in Word, Excel, PowerPoint — not plain pasted text. The per-file limit is around **30 MB**.

The typical formats: **.docx** (Word), **.xlsx** (Excel), **.pptx** (PowerPoint), **.pdf**. The rule for choosing is the recipient: a report to be reread and edited is a .docx; data and calculations are an .xlsx; something to present is a .pptx; a final document to distribute as is is a .pdf.

Asking well

A document is only as good as the request. Three things raise the quality of the result:

- **The recipient and the purpose:** “a note for the board”, “a flyer for customers”. It changes tone, length, structure.
- **The structure you want:** the sections, the order, what to highlight.
- **The final format:** explicitly say .docx, .xlsx, .pptx or .pdf.

Compare:

- **Weak:** “Make me a document about sales.”
- **Better:** “Create a one-page .docx report on Q1 sales for management: an opening summary, a table by region, three comment points. Sober tone.”

Content first, then the file

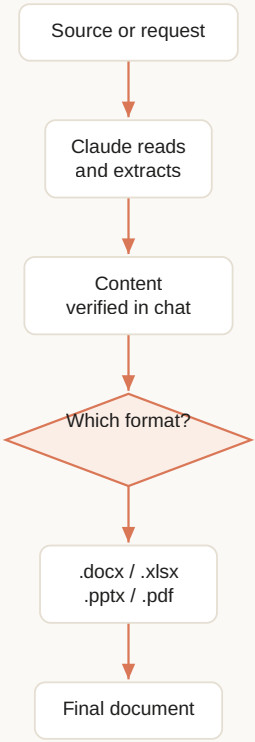
The principle that makes the difference is “**read before create**”: first the content is settled, then it’s laid out. A document starts badly if you ask at the same time “find the data and make me the PDF”: the risk is a nice layout on wrong content.

It pays to separate the two moments:

1. **Content.** Have Claude produce or verify the data, the text, the numbers. Check them in chat.
2. **Document.** Only when the content is right, ask to put it into the file with the desired formatting.

If you start from an existing file — a report to summarize, data to rework — the same holds: first Claude **reads** the source and extracts what’s needed, then builds the new document. Laying out first means redoing the work.

Figure L3.4.1 — The “content first, then the file” flow. Alt text: vertical diagram from verified content to format choice to the final document.



Starting from a template

If you produce the same kind of document often — a monthly report, a proposal — don’t start from scratch every time. Give Claude a **template**: a sample file already set up, or a precise description of the structure. It pours the new content into it, keeping form and style.

It’s the natural bridge to Projects (ch. L3.2): put the template in the knowledge base and every document comes out consistent with the

previous one. For voice and recurring style rules, Skills (Level 5) make all of this even more repeatable.

In practice: a report in two moments

1. In chat, have the content prepared and **check it**:

   ```
   From the data I'm pasting, write the summary and
   the table by region. Show them to me, don't make
   the file yet.
   ```

2. Fix what's needed until the content is right.
3. Ask for the document, with format and structure:

   ```
   Now put it in a one-page .docx: title,
   summary, table, three comment points.
   ```

4. Open the file and check. For touch-ups, ask for targeted edits instead of regenerating everything.

Common mistakes

- **Asking for data and layout together.** Separate them: first the right content, then the file.
- **Not stating the format.** Without ".docx" or ".xlsx" you risk a generic output. Be explicit.
- **Regenerating for every touch-up.** Ask for targeted edits ("change only the table"): faster and safer.
- **Forgetting the sandbox.** If file creation doesn't start, check that Code execution is active (ch. L1.3).

Summary

1. Claude generates real files (.docx, .xlsx, .pptx, .pdf) through the code execution sandbox.
2. Choose the format based on the recipient: report, data, presentation, final document.
3. Ask well: recipient, structure, explicit format.
4. “Content first, then the file”: verify the content in chat, then lay it out.
5. For recurring documents start from a template; with Projects it becomes repeatable.

Next step

In **ch. L3.5 — Slides and Excel** we get into the two most requested cases: building a presentation and a spreadsheet with a method that avoids redoing the work, and using the add-in inside Office.

Data on Code execution / file creation (plans, ~30 MB limit) from the ledger, verified on 22/06/2026 on support.claude.com. The examples were not executed here.

Chapter L3.5 — Slides and Excel

Level 3 — Daily work. Product details verified on 22/06/2026 against official sources.

Goal

By the end you'll know how to build a presentation with a three-step method (structure, content, style), set up a spreadsheet moving from data to formulas to charts, and understand when the add-in inside Office is worth it instead of generating in chat.

Prerequisites

- Having read how to ask for documents (ch. L3.4).
- Code execution / file creation active (see ch. L1.3).

Slides: structure, then content, then style

A presentation built all in one shot almost always comes out muddled. The method that works has three steps, and follows the same "content first" principle as the previous chapter:

1. **Structure.** Decide the slide sequence first: how many, with which title, in what order. It's the backbone of the talk. Have it reviewed until the thread holds.
2. **Content.** Fill each slide: few points per slide, one idea per slide. Here what matters is what you say, not how it looks.
3. **Style.** Only at the end, the appearance: theme, colors, visual consistency.

Reversing the order is costly: polishing the style of slides you'll later rewrite is wasted work. An example of a first move:

```
Prepare the OUTLINE of an 8-slide presentation
on project X for management: titles only, plus
one line of content per slide.
```

Once the outline holds, you move to the content slide by slide, and only at the end do you ask for the `.pptx` with the theme.

Excel: data, then formulas, then charts

For spreadsheets the right order is just as clear:

1. **Data.** First clean, well-structured data: columns with clear headers, one row per record. Everything else rests on this.
2. **Formulas.** Then the calculations: totals, percentages, aggregations. On well-structured data the formulas come out simple and correct.
3. **Charts.** Finally the visualization, built on the data and calculations already in place.

If the data is messy, formulas and charts will inherit the mess. It's worth spending the first step tidying up the columns.

> **Tip:** ask Claude to **explain the formulas** it adds. A sheet you don't understand is a sheet you can't trust: having them commented gives you a check.

Table L3.5.1 — The right order, at a glance.

Output	First	Last
Slides	Structure	Style
Excel	Data	Charts

The add-in inside Office

Beyond generating files in chat, Claude lives **inside** Office as an add-in for **Excel, Word and PowerPoint** (Claude for Microsoft 365). It's in beta (research preview) for the **Max, Team and Enterprise** plans.

The practical difference is where you work:

- **In chat:** Claude **creates** the file from scratch. Handy when you start from nothing or produce many similar documents.
- **In the add-in:** Claude works **on the open document** in Office. Handy when the file already exists, is large, or you want to stay in the tool where you'll then refine it by hand.

Quick rule: if the document is being created now, chat is just fine; if you're editing something already in Excel or PowerPoint, the add-in saves you the back-and-forth between apps.

In practice: a presentation in three steps

1. Ask for the **outline** (titles only + one line). Adjust it.
2. For each slide, ask for the **content**: few points, one per idea.
3. Ask for the **.pptx** with a consistent theme:

   ```
   Generate the .pptx from the approved outline:
   restrained theme, one accent color, consistent titles.
   ```

4. Open the file. For touch-ups, targeted edits (ch. L3.4) or, if you have it, the PowerPoint add-in.

Common mistakes

- **Starting from style.** Theme and colors come last. First structure and content.

- **Too much per slide.** One idea per slide. Walls of text don't present.
- **Dirty data in Excel.** Formulas and charts inherit the mess: tidy the columns first.
- **Trusting formulas you don't understand.** Have them explained: it's your check.
- **Looking for the add-in on the wrong plan.** It's beta on Max/Team/Enterprise.

Summary

1. Slides in three steps: **structure** → **content** → **style**. Style last.
2. Excel in three steps: **data** → **formulas** → **charts**. Clean data first of all.
3. Have the formulas explained: a sheet you don't understand isn't reliable.
4. In chat Claude **creates** the files; in the add-in it works **on the open document** in Office.
5. The M365 add-in is beta on Max/Team/Enterprise.

Next step

With daily work wrapped up, **Level 4 — Design** opens the canvas: generating and iterating on interfaces, starting from your own brand and bringing the result into Claude Code. We begin with **ch. L4.1 — Design: the canvas**.

Data on the Microsoft 365 add-in (products, beta, plans) from the ledger, verified on 22/06/2026 on claude.com and support.claude.com. The examples were not executed here.

Chapter L4.1 — Design: the canvas

Level 4 — Design. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know what Claude Design is, how to generate your first project on the canvas by conversing, and how to refine it in the three ways the product offers: from chat, with inline comments and by editing directly on the canvas. You'll also learn to ask for variants and to save a direction before trying another.

Prerequisites

- Knowing how to write effective requests (ch. L1.2): they apply identically here.
- A paid plan: **Pro, Max, Team or Enterprise**. Design is in beta and on Enterprise it's **off by default** (an admin enables it).

What Claude Design is

Claude Design is the tool for creating interfaces, interactive prototypes and presentations by **conversing** with Claude, instead of drawing by hand. You use it on the web (claude.ai/design) or from the desktop app's sidebar; it's not on mobile.

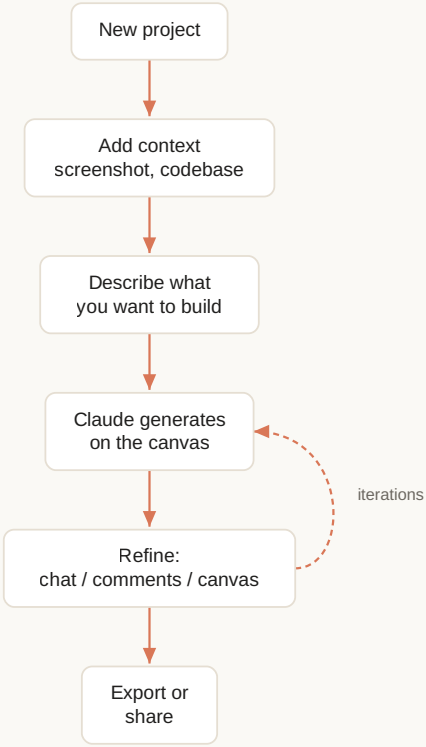
The screen has two areas: the **chat on the left**, where you describe what you want, and the **canvas** on the right, where Claude generates the design. From there you iterate: refine by conversing, leave comments on specific points, or step in by hand on the canvas. It's the

same principle as chat — describe the result, then correct it — applied to a visual output.

The workflow

A Design project almost always follows the same path.

Figure L4.1.1 — The flow of a project in Claude Design. Alt text: vertical diagram from the new project to the description, to generation on the canvas, to refinement, to export.



The project automatically inherits your organization's design system, if configured (we'll see this in ch. L4.2): colors, fonts and components are already in the right place, without uploading anything.

Writing a good prompt

You don't need to be a designer. You need to be specific. A good prompt states four things:

- **Goal:** what you're building.
- **Layout:** how to arrange the elements.
- **Content:** what information to show.
- **Audience:** who will use it.

For example: "Create a dashboard that shows monthly revenue, with filters by region and product line." If something important is missing, Claude asks questions before generating.

Three ways to refine

The first generation is a starting point. The real value is in the iteration, and the product offers three routes — using them at the right moment saves time.

Table L4.1.1 — Which tool for which change.

Tool	For what	Example
Chat	broad, structural changes	"darker theme"
Inline comments	targeted, on a component	"wider padding"
Direct editing	quick visual touch-ups	drag, resize

Chat is for changes that affect the whole or need an explanation ("reorganize the dashboard", "add a panel on the right"). **Inline comments**

are applied by clicking directly on the element: faster than describing in words where it is. **Direct editing** on the canvas is for moving, resizing and aligning on the fly.

> **Warning:** sometimes an inline comment disappears before Claude reads it. It's a known issue: if the edit isn't picked up, paste the same feedback into chat.

Variants and versions

Two useful moves for when you haven't settled on a direction yet. To explore alternatives, ask: "Show me 2-3 different layouts for this page" — comparing is faster than guessing. To change course without losing the work done, tell Claude: "Save what we have and try a completely different approach." Claude keeps the current version and confirms where it saved it, so you can come back to it.

In practice: your first project

1. Open **claude.ai/design** or the Design sidebar in the app, and create a project.
2. Add context if you have any: a reference screenshot, a codebase.
3. Write the prompt with **goal, layout, content, audience**:

```
Landing page for our new API product:
hero, code examples, pricing section.
Audience: developers.
```

4. Look at what it generates on the canvas.
5. Refine: chat for the big changes, comments for the details, freehand for the alignments.

6. Ask for 2-3 layout variants and choose from there.

Common mistakes

- **Vague prompt.** “Make me a site” produces generic results. State goal, layout, content, audience.
- **Everything from chat.** To move an element or change a padding, inline comments and direct editing are faster.
- **Feedback not picked up.** If an inline comment disappears, paste it into chat.
- **Looking for Design on mobile.** It’s web and desktop only.

Summary

1. Claude Design creates UI, prototypes and slides by **conversing**; chat on the left, canvas on the right. Web and desktop only.
2. The flow: project → context → prompt → generation → refinement → export.
3. A good prompt states **goal, layout, content, audience**.
4. Refine with **chat** (broad changes), **inline comments** (targeted), **direct editing** (visual touch-ups).
5. Ask for **variants** and **save a version** before changing direction.

Next step

In **ch. L4.2 — Design system import** we see how to make Design start from your real brand: importing colors, fonts and components from a codebase or a deck, and reducing the risk of an anonymous output.

Data on Claude Design (areas, flow, refinement, beta/plans) verified on 24/06/2026 on support.claude.com/en/articles/14604416. The canvas requires a paid account, so the steps were not executed here.

Chapter L4.2 — Design system import

Level 4 — Design. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know why to make Claude Design start from your brand instead of from scratch, which sources to import a design system from, what Claude extracts from it, and how to avoid the anonymous, generic output — the so-called "AI slop".

Prerequisites

- Having used the canvas (ch. L4.1).
- Having at least one source of your brand: a codebase, a deck, or graphic assets.

Why import the brand

Without a design system, Claude generates with a default style: correct but neutral, recognizable as "made by AI". It's the problem of **AI slop**: plausible results with no identity, all looking the same. The cure is to give Claude your brand as a foundation, so every screen starts with your colors, fonts and components instead of the defaults.

Importing a design system is the step that turns Design from a generator of generic drafts into a tool that produces material that's already "yours".

What Claude extracts and from where

Claude analyzes the sources you give it and extracts a **reusable design system**:

- **Color palette:** primary, secondary and accent colors.
- **Typography:** font families, sizes, weights.
- **Components:** buttons, cards, navigation elements and other patterns.
- **Layout:** spacing, grids, page structures.

The possible sources are more than one, and just one is enough to start from:

Table L4.2.1 — Sources to import from and what they give.

Source	What it contains	Result
Codebase	real components	the most faithful
Deck / PDF	colors, layout	good for the look
Brand assets	logo, palette	minimal base

A **codebase** (for example a React component library) is the richest source: Claude reads the real components and reuses them. A well-made **deck** or PDF is enough to extract colors and layout. Even just a logo and a palette give a starting point. The more sources you provide, the more Claude has to work from.

How to set it up

Configuring the design system is an operation for a **brand owner or designer**, to be done **once**: afterwards, the team's projects inherit it automatically (Team/Enterprise). The steps, in brief:

1. In Claude Design, select or create your **organization**.

2. Upload the brand **assets** (codebase, deck, graphic assets).
3. Claude generates a **UI kit**: review it by creating a test project.
4. When you're satisfied, turn on the **"Published" toggle**: from then on, the org's new projects use your design system.

> **Note:** in teams, the **Claude Design Admin** role lets you approve a standard design system and lock down changes, so the output always stays within the company guidelines.

Importing from Claude Code

If your design system lives in code, you can import it **from the terminal** with Claude Code's `/design-sync` command: it brings the local codebase's design system into Design, so every screen starts from your real components. It's the bridge between code and canvas, which we go deeper into in ch. L4.3.

Reducing AI slop

The import is worth only as much as its source: a messy codebase or an incomplete file shows in the output. A few moves raise the quality:

- **Give real examples, not just specs.** A finished landing page says more than an isolated color palette: it conveys the "feel" of the brand.
- **Iterate on the extraction.** If the first result doesn't capture the brand, upload different or additional assets.
- **Name the components in the prompt.** "Use the Primary Button component" or "Apply the Card pattern" guides Claude toward your system instead of toward a default.

In practice: bring in your brand

1. Gather the best source you have: ideally the codebase, otherwise a curated deck.
2. Create/select the organization in Claude Design and upload the assets.
3. Review the generated UI kit with a test project (“Create a landing page for our product”).
4. If it doesn’t capture the brand, add assets and iterate.
5. Turn on **Published** and check that new projects use your style.

Common mistakes

- **Starting without a design system.** You get AI slop: anonymous style. Import the brand first.
- **Messy source.** A confused codebase or incomplete file is reflected in the output: clean the source or pick a better one.
- **Only specs, no examples.** A palette alone says little: add a real page.
- **Not naming the components.** If you know a component exists, call it by name in the prompt.

Summary

1. Without a design system the output is anonymous (**AI slop**); importing the brand is the cure.
2. Claude extracts **colors, typography, components and layout** from the sources.
3. The most faithful source is a **codebase**; decks and assets give a base.
4. The setup is for a brand owner, **once**: the team inherits the system.

5. The import is worth as much as the source: give real examples and iterate to reduce AI slop.

Next step

In **ch. L4.3 — From Design to code** we walk the complete bridge: the `/design-sync` command in both directions and the handoff to Claude Code that builds real software starting from the canvas.

Data on the design system import (sources, extraction, setup, admin role) verified on 24/06/2026 on support.claude.com/en/articles/14604397. The setup requires a paid account and was not executed here.

Chapter L4.3 — From Design to code (/design-sync)

Level 4 — Design. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll understand the bridge between Claude Design and Claude Code: how `/design-sync` connects canvas and code in both directions, how to pass a design to Claude Code so it becomes real software without starting over from a screenshot, and how to start a Design project directly from the terminal. It's the chapter that makes Design and Code a single cycle instead of two separate tools.

Prerequisites

- Claude Code installed and configured (ch. L2.2, L2.4).
- Having used the canvas (ch. L4.1) and, ideally, imported a design system (ch. L4.2).

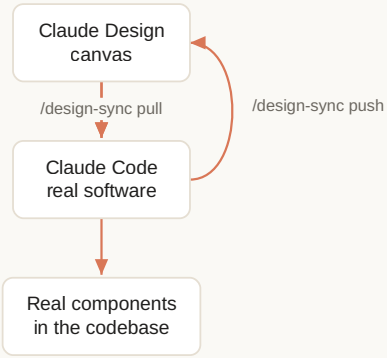
The problem it solves

The classic pain point of design→development work is the handoff jump: the designer hands over a screenshot or a file, and whoever develops **rebuilds everything from scratch**. Time and fidelity are lost. The bridge between Design and Code removes this step: Claude Code continues from the existing work, not from a photo of the result.

The Design ↔ Code cycle

Design and Code are not a one-way chain, but a loop. The design generates code; the code, as it evolves, updates the design. `/design-sync` keeps the two sides aligned.

Figure L4.3.1 — The two-way cycle between Design and Code. Alt text: vertical diagram showing the canvas and the code connected by pull and push, with the handoff toward Claude Code.



/design-sync in both directions

The `/design-sync` command runs **inside Claude Code** and works in two directions.

- **Pull (code → canvas):** imports your local codebase's design system into Claude Design, so every screen you generate starts from your real components.
- **Push (canvas → code):** after you've implemented a design in code, `/design-sync` sends the current state back into the canvas, keeping Design aligned with what you've actually built.

In practice: you start from the code to give Design the right foundations (pull), build and iterate, then send the result back so that canvas and product don't drift apart (push).

Table L4.3.1 — The two directions of `/design-sync`.

Direction	From → To	What it's for
Pull	code → canvas	design on real components
Push	canvas → code	canvas aligned to the build

The handoff to Claude Code

When a design is ready to become software, you **hand it off** to Claude Code. Code continues from the existing work instead of rebuilding it from a screenshot. The handoff can also be started from Design's **Export** button (see ch. L4.5) and can target the local coding agent or Claude Code Web.

Starting from Claude Code: /design

If you prefer to stay in the terminal, you don't have to open the canvas at all. With the `/design` command you start, edit and sync a Design project **without leaving Claude Code**: import a design into the codebase, export the code as a live prototype, or let Claude build everything from scratch.

> **Note:** if you don't see `/design` or `/design-sync` in Claude Code, run `/update` to update the skills. The commands appear only in **new sessions**, not in the one already open.

In practice: from codebase to canvas and back

1. In Claude Code, inside your project, run:

```
/design-sync
```

2. Choose **pull**: you import the codebase's design system into Design.
3. In Design, generate and iterate on the screens: they'll use your real components.
4. When a screen is ready, do a **handoff** to Claude Code and build.
5. After the implementation, `/design-sync` again in **push** to realign the canvas to the code.

> **Tip:** if the commands don't appear, `/update` and then open a new session.

Common mistakes

- **Expecting the commands in the current session.** After `/update`, `/design` and `/design-sync` are only in new sessions.
- **Handoff from a screenshot.** Not needed: the handoff brings the real work, not a photo. Pass the project, not an image.
- **Canvas and code diverging.** After building, remember the **push**: without it, Design falls behind the product.
- **Skipping the initial pull.** Without it, Design generates with the default style and not with your components.

Summary

1. The Design↔Code bridge eliminates the "rebuild from screenshot".
2. `/design-sync` runs in Claude Code in two directions: **pull** (code→canvas) and **push** (canvas→code).
3. The **handoff** passes the design to Code, which continues from the real work.

4. With `/design` you start and sync a Design project **from the terminal**.
5. If the commands are missing, `/update` and open a **new session**.

Next step

In **ch. L4.4 — Design inside Cowork** we see how to use Design's outputs as material for agentic tasks, and how Skills make visual work repeatable.

Data on `/design-sync`, handoff and `/design` verified on 24/06/2026 on support.claude.com/en/articles/14604416. The commands require Claude Code with a paid account, so they were not executed here.

Chapter L4.4 — Design inside Cowork

Level 4 — Design. Concepts of composition between products; details verified on 24/06/2026.

Goal

By the end you'll know how to use what you produce in Claude Design as **material** for Cowork's agentic tasks, and you'll understand how Skills make visual work repeatable instead of having to re-explain it every time. This is a chapter about composition: not a new tool, but two you already know put to work together.

Prerequisites

- Knowing how to start a task in Cowork (ch. L3.1).
- Having produced something in Design (ch. L4.1) and exported an output (ch. L4.5).

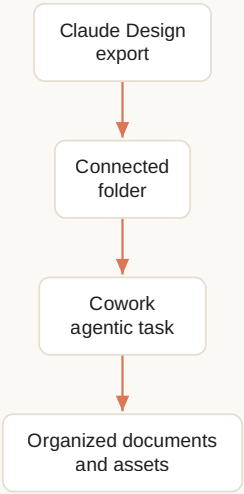
Design produces, Cowork works

Design and Cowork answer different needs. Design **creates** visual material: screens, prototypes, slides, the code of a prototype. Cowork **acts** on a folder of files across multiple steps. The point of contact is natural: Design's outputs become Cowork's **inputs**.

The link runs through the folder. You export from Design (for example a ZIP, a standalone HTML, or a prototype's code, see ch. L4.5), put it in a folder connected to Cowork, and from there Cowork works on it: it reorganizes the assets, generates the documentation, prepares the files for

delivery. Remember that Cowork sees **only the folders you connect** (ch. L3.1): an asset exists for it only once you've put it there.

Figure L4.4.1 — From Design's outputs to Cowork's tasks. Alt text: vertical diagram from the Design export to the connected folder to Cowork's agentic task.



Skills make design repeatable

The real leap in quality isn't moving a file: it's making the way visual work is handled **repeatable**. This is where Skills come in (Level 5): a skill writes the rules once — how to name assets, what folder structure to use, what checks to run on an export — and Claude applies them the same way in chat, in Cowork and in Code.

Without a skill, you repeat the instructions for every task ("put the assets in `/img`, rename them like this, generate the index"). With a skill, you say them once and they always hold. It's the difference

between a result that depends on how precise you were today and a result that's consistent every time.

A concrete example

Imagine producing the mockups of a new section of the site in Design and exporting them as standalone HTML. The composed flow:

1. You export the mockups from Design into a project folder.
2. You connect the folder to Cowork.
3. You give Cowork a task as an **end-state** (ch. L3.1): "Organize these mockups by page, generate an `INDEX.md` with previews and notes, and prepare a `delivery/` folder ready to send."
4. A project skill ensures that names, structure and checks are always the same, mockup after mockup.

The result: Design does the creative part, Cowork the repetitive, organizational part, and the skill holds the rules together.

In practice: composing Design and Cowork

1. In Design, export the output you need (ch. L4.5) into a folder.
2. In Cowork, **connect** that folder.
3. Write the task as the desired result, not as a sequence of steps.
4. If you repeat the same kind of work, wrap the rules in a project **skill** (Level 5).
5. Re-run it on new assets: the skill keeps things consistent.

Common mistakes

- **Expecting an automatic link.** Design doesn't "enter" Cowork by itself: you go through an export and a connected folder.

- **Forgetting to connect the folder.** Cowork sees only what you connect: if the asset isn't there, it doesn't exist as far as Cowork is concerned (ch. L3.1).
- **Repeating the rules for every task.** If the work recurs, put it in a skill instead of re-explaining it.
- **Guiding Cowork step by step.** It holds here too: describe the end-state.

Summary

1. Design **creates** visual material; Cowork **acts** on the files: the outputs of the first are inputs to the second.
2. The link runs through an **export** into a **folder connected** to Cowork.
3. **Skills** make the handling of visual work repeatable, the same in chat, Cowork and Code.
4. Typical pattern: Design creates, Cowork organizes, the skill holds the rules.
5. In Cowork always describe the **result**, not the steps.

Next step

In **ch. L4.5 — Export and Canva** we close Level 4 with Design's output formats — PDF, PPTX, HTML, Canva and the others — and with the criterion for choosing between exporting and doing a handoff.

Chapter on composition between Design (ch. L4.1, L4.5) and Cowork (ch. L3.1). The capabilities of Skills are introduced here and explored in depth at Level 5. No command was executed here.

Chapter L4.5 — Export and Canva

Level 4 — Design. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know how to export a Claude Design project in the right format, send it to the tools you already use (Canva and others), and choose sensibly between **exporting** a file and doing a **handoff** to code. This is the chapter that takes the work off the canvas.

Prerequisites

- A project ready on the canvas (ch. L4.1).
- For the handoff, Claude Code installed (ch. L2.2) and the concept of the Design ↔ Code bridge (ch. L4.3).

Where export lives

When the design is ready, the **Export** button at the top right opens the ways out. The right format depends on what you need to do with it: gather feedback, hand off to development, or present to a group.

The output formats

The destinations group by purpose. The table sums up the main ones.

Table L4.5.1 — Export formats and when to use them.

Purpose	Format	When
Review	PDF	feedback, print
Present	PPTX / Canva	slides to refine
Publish	HTML / ZIP	prototype, web

Beyond these, Design exports as **.zip**, sends **to Canva** and produces **standalone HTML**. It can also send the project to external tools — Adobe, Base44, Canva, Gamma, Lovable, Miro, Replit, Vercel, Wix — with other destinations on the way. For sharing within the organization there's a **link** with three-level permissions: view only, comment, edit.

Canva and the external tools

"Send to Canva" brings the design into Canva, where you refine it with layout tools you already know. The same logic applies to the other destinations: Design isn't trying to replace your tools, but to **hand them** a starting point already set up on your brand, instead of making you start over from a blank page.

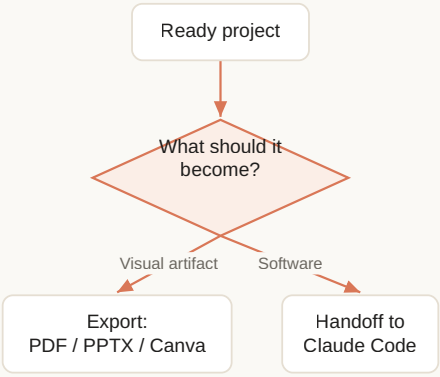
Export or handoff?

This is the choice that matters most, and it depends on where the work is headed.

- **Export** when the output is the goal: a PDF for an opinion, a PPTX to present, a Canva file to refine. The design stays a visual artifact.
- **Do a handoff** when the design has to become **software**. The handoff to Claude Code (ch. L4.3) continues from the real work, without rebuilding from a screenshot, and targets the local coding agent or Claude Code Web.

Rule of thumb: if the next person to touch it opens a viewer, export; if they open a code editor, do a handoff.

Figure L4.5.1 — Export or handoff: the fork. Alt text: vertical diagram that from the ready project separates the export branch from the handoff branch toward code.



In practice: take your design out

1. Open the project and press **Export** at the top right.
2. For an opinion: **PDF**. To present: **PPTX** or **Send to Canva**.
3. For a web prototype: **standalone HTML** or **.zip**.
4. If it has to become software, choose **Handoff to Claude Code** instead of a file.
5. To share with the team, generate a **link** with the right permission (view, comment or edit).

Common mistakes

- **Exporting a screenshot for development.** If it becomes code, do a handoff: you carry the real work, not a photo.

- **Wrong format for the purpose.** PDF for an opinion, PPTX/Canva to present, HTML/ZIP for the web.
- **Link with too broad a permission.** For a simple opinion, “view only” or “comment” is enough, not “edit”.
- **Looking for destinations that aren’t there yet.** The list grows over time: check which are available at the moment.

Summary

1. The **Export** button (top right) opens Design’s ways out.
2. Formats: **PDF, PPTX, HTML, .zip**, sending to **Canva** and to other tools.
3. **Send to Canva** and the like hand over a starting point already on your brand.
4. **Export** for a visual artifact; **handoff** when it has to become software.
5. Share within the org with a **link** with permissions: view, comment, edit.

Next step

You’ve completed Level 4. In **Level 5 — Skills and identity** we look at what Skills really are — already met as the glue of Cowork and Design — starting from **ch. L5.1 — Anatomy of a skill**.

Data on export, destinations and sharing verified on 24/06/2026 on support.claude.com/en/articles/14604416. The features require a paid account, so they were not executed here.

Chapter L5.1 — Anatomy of a skill

Level 5 — Skills and identity. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know what a skill is, how it differs from a prompt, how a `SKILL.md` file is made and why its `description` is the most important part. It's the conceptual foundation for building one of your own in the next chapter.

Prerequisites

- Knowing how to write effective requests (ch. L1.2).
- Having **Code execution** active (ch. L1.3): it's required for Skills.

Skill or prompt?

A prompt is good for one conversation: you write it, you get the answer, it ends there. A **skill** is reusable knowledge that Claude loads **on its own** when the context calls for it, without you pasting it every time. You write a rule once, and Claude applies it every time it's needed.

The practical difference: the prompt is a throwaway instruction; the skill is a permanent competence. If you notice yourself re-pasting the same instructions chat after chat, that text is a candidate to become a skill.

A folder and a file

A skill, in its minimal form, is a **folder** that contains a `SKILL.md` file. Only that file is mandatory: it alone is enough to make a working skill. The file has two parts:

- a **frontmatter** at the top (metadata between two `---` lines);
- a **body** in Markdown with the actual instructions.

> **Note:** by convention the file is called `SKILL.md`. The documentation also refers to it as `skill.md`: it's the same file. In Claude Code, project skills live under `.claude/skills/` (ch. L2.4).

The frontmatter: two mandatory fields

The frontmatter declares what the skill is. Two fields are mandatory:

- **name** — the readable name, maximum **64 characters**.
- **description** — what the skill does and **when to use it**, maximum **200 characters**.

There are optional fields (for example `dependencies` for software packages), but name and description are enough to get started.

Here's a complete minimal skill:

```
---
name: meeting-minutes
description: From raw notes, create minutes
  with decisions and action items.
---

# Meeting minutes

Turn messy notes into minutes:
1. List the decisions made.
2. Extract action items with an owner.
3. Close with the points still open.
```

The description is everything

Between the two fields, the `description` is the one that really counts. Claude reads it to decide **whether** to activate the skill: with dozens or hundreds of skills available, it's the description that triggers the right one at the right moment. A vague description ("helps with documents") never triggers precisely; a specific one ("from raw notes, creates minutes with decisions and action items") does.

Here's where the mechanism of **progressive disclosure** comes in: Claude first reads only the metadata — name and description — to understand whether the skill is needed, and loads the full body **only if** it decides to use it. It's efficient: a few lines are enough for it to choose, without reading everything. That's why the description should be written with an eye to when you want the skill to activate, not just to what it does.

The body: instructions and examples

Under the frontmatter, the Markdown body contains the instructions Claude follows when the skill is active: what to do, in what order, with what constraints. Examples help a lot — showing an input and the ex-

pected output is worth more than an abstract explanation. If the instructions grow too long, you can move part of them into separate resource files (for example a `REFERENCE.md`) and reference them from `SKILL.md`.

Common mistakes

- **Confusing skill and prompt.** The prompt is for one chat; the skill is reusable and activates on its own.
- **Vague description.** It's the field that triggers the skill: be specific about the "when".
- **Going over the limits.** A name over 64 or a description over 200 characters isn't valid.
- **Putting everything in the body.** If it's too much, use separate resource files.

Summary

1. A skill is **reusable** knowledge that Claude loads on its own; a prompt is good for a single chat.
2. In its minimal form it's a **folder** with a `SKILL.md` file.
3. The frontmatter has two mandatory fields: **name** (≤64) and **description** (≤200).
4. The **description** decides when the skill triggers: write it with the "when" in mind.
5. The Markdown body contains instructions and examples; extra resources go in separate files.

Next step

In **ch. L5.2 — Your first skill** we build a skill from scratch, by hand and with the help of skill-creator, and we learn to test that it activates when it should.

Data on SKILL.md, frontmatter and limits verified on 24/06/2026 on support.claude.com/en/articles/12512198 and code.claude.com/docs/en/skills. The SKILL.md example was not executed here.

Chapter L5.2 — Your first skill

Level 5 — Skills and identity. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll have created a working skill: the folder structure, a `description` that "triggers" when it should, and a way to test it. We'll see both the manual route and the help of **skill-creator**, the skill that assists you in building others.

Prerequisites

- Understanding the anatomy of a skill (ch. L5.1).
- **Code execution** active (ch. L1.3).

The folder structure

A skill is a folder whose name matches the name of the skill, with at least the `SKILL.md` file inside. If you add resources or scripts, they go in the same folder.

Figure L5.2.1 — Structure of a skill folder. Alt text: vertical diagram of the skill folder with SKILL.md and a resources subfolder.

To use it in chat or in Cowork, the folder must be compressed into a **ZIP** with the folder itself as the root (not the loose files in the ZIP), and then uploaded. In Claude Code, on the other hand, you put it under the project's `.claude/skills/` (ch. L2.4).

Writing a description that triggers

This is the step that decides everything. The `description` must say **what** the skill does and **when** Claude should use it, in at most 200 characters. Two examples side by side:

- **Weak:** "Helps with emails." — too generic, triggers randomly or never.
- **Strong:** "Writes polite, short email replies to customers, starting from the message received." — it states the task and the context of use.

The rule: include the signals you want to trigger the skill. If it should activate on minutes, name minutes; if on customers, name customers.

Building it with skill-creator

You aren't forced to start from a blank page. **skill-creator** is an official skill that guides you in creating one: it sets up the structure, helps you write the description and can generate test prompts (evals) to check that the skill triggers when it should — and, importantly, that it does **not** trigger when it shouldn't.

It's the quickest way to a solid first skill: you describe the workflow you want to automate, and skill-creator produces the skeleton that you then refine.

Testing before and after upload

A skill is judged in the field. Before uploading it, re-read the `SKILL.md`, check that the description really reflects when it's needed, and verify that the referenced files exist. After uploading and enabling it in **Customize > Skills**, try it with several prompts that should activate it, and check in Claude's "thinking" that it's loading it. If it doesn't trigger when you expect, the thing to fix is almost always the **description**.

In practice: create and test a skill

1. Create a folder with the name of the skill (e.g. `meeting-minutes`).
2. Inside, write `SKILL.md` with frontmatter (name, description) and body.
3. Take care with the **description**: what it does and when to use it, under 200 characters.
4. For chat/Cowork, compress the folder into a **ZIP** (folder as root) and upload it; enable it in **Customize > Skills**.
5. Try it with prompts that should activate it; check that it loads.
6. If it doesn't trigger, **iterate on the description** and try again.

Common mistakes

- **ZIP with loose files.** The ZIP must contain the **folder** as root, not the files directly.
- **Folder name different from the skill name.** Make them match.
- **A description that doesn't trigger.** Add the "when" signals; don't describe only the "what".
- **Testing it only once.** Test with several prompts, including some that should **not** activate it.
- **Hardcoding secrets.** Never keys or passwords inside a skill.

Summary

1. A skill is a **folder** (name = skill name) with `SKILL.md` inside.
2. For chat/Cowork it's uploaded as a **ZIP** with the folder as root; in Code it goes in `.claude/skills/`.
3. The **description** that triggers names what it does **and** when to use it.
4. **skill-creator** scaffolds the skill and generates activation tests.
5. Test before and after upload; if it doesn't trigger, iterate on the description.

Next step

In **ch. L5.3 — Skills in operation in Cowork** we look at Skills at work in autonomous tasks: differences between chat and Cowork, how to split them, and team sharing. We'll use as an example the three skills this book is written with.

Data on structure, ZIP packaging, testing and skill-creator verified on 24/06/2026 on support.claude.com/en/articles/12512198. No skill was created or executed here.

Chapter L5.3 — Skills in operation in Cowork

Level 5 — Skills and identity. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know how to use Skills in autonomous work: how they behave in chat versus Cowork, why it's worth splitting them into several focused skills instead of one big one, and how to share them in a team. The guiding example is the three skills this book is written with.

Prerequisites

- Knowing how to create a skill (ch. L5.2).
- Knowing Cowork (ch. L3.1).

In chat and in Cowork

The same skill is valid in chat, in Cowork and in Claude Code: you write it once and it guides Claude the same way everywhere. What changes is the context it works in.

In **chat** the skill refines a single response: tone, format, structure. In **Cowork**, where Claude runs a long task on a folder across multiple steps, the skill counts for more: it governs a behavior repeated over many files, without your having to repeat the rules at every step. The more autonomous and long the work, the more a well-written skill makes the difference between a consistent result and one that depends on how precise you were today.

Splitting instead of piling up

The temptation is to write one skill that does everything. It's almost always a mistake. Several **focused** skills compose better than a monolithic one, for two reasons: the `description` of a targeted skill triggers precisely (ch. L5.1), whereas that of a do-it-all skill is necessarily vague; and Claude can **combine** several skills automatically when they're needed together.

Rule of thumb: one skill, one workflow. If you notice a skill has two distinct purposes, split it. Skills can't call one another, but Claude uses several in the same task: the composition is done by it.

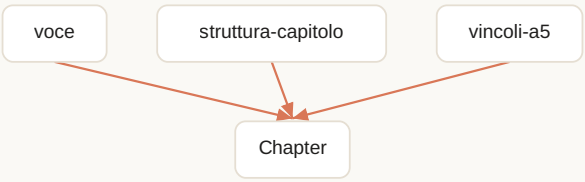
Example: the three skills of this book

This manual is written with three separate skills instead of one big one:

- **voce** — tone and terminology: plain, direct prose, product terms in English, no brochure-style enthusiasm.
- **struttura-capitolo** — standard structure, length target, use of the facts ledger, definition of done.
- **vincoli-a5** — layout constraints: narrow code, three-column tables, vertical diagrams.

Each has a clear purpose and a description that triggers on its scope. When I write a chapter, Claude uses **all three together**: voice governs the text, structure the skeleton, constraints the form. A single skill that did the work of all three would be harder to write, to trigger and to update.

Figure L5.3.1 — Three focused skills that Claude composes on a chapter. Alt text: vertical diagram with three skills converging on the writing of a chapter.



Sharing in a team

A useful skill shouldn't stay on your account. On the **Team and Enterprise** plans an administrator can provide and manage skills for the whole organization, so everyone works with the same rules. It's the way to make a standard — of voice, of format, of process — something that holds for the group and not just for whoever wrote it.

> **Tip:** before sharing a skill in a team, have someone else try it on real prompts. A description that triggers well for you may trigger badly for someone who uses different words.

In practice: bring your skills into Cowork

1. Start from a workflow you repeat often in Cowork (e.g. organizing assets, producing a report).
2. Write its rules in a **focused** skill (ch. L5.2).
3. If the purposes are two, make **two** skills, not one.
4. Activate it and launch a task in Cowork: verify that the rules are applied.
5. If you're in a team, ask the admin to make it available to the organization.

Common mistakes

- **The do-it-all skill.** Vague description and hard to maintain. Split by workflow.
- **Expecting one skill to call another.** It doesn't happen: it's Claude that composes them. Write them self-contained.
- **Sharing without having it tested.** In a team, a description should be tested on several ways of writing the same intent.
- **Repeating the rules in the prompt.** If they're in a skill, there's no need to restate them in every task.

Summary

1. The same skill is valid in chat, Cowork and Code; in Cowork it weighs more because it governs long, autonomous tasks.
2. Several **focused** skills compose better than a monolithic one.
3. Skills don't call one another: the **composition** is done by Claude.
4. The three skills of this book (voce, struttura, vincoli) are the example of the pattern.
5. In **Team/Enterprise** an admin shares the skills with the whole organization.

Next step

In **ch. L5.4 — Making Claude sound like you** we put together all the identity tools — instructions, Projects, context files and Skills — to give Claude your voice in a stable way.

Data on use, composition and sharing of skills verified on 24/06/2026 on support.claude.com/en/articles/12512198. The example of the three skills is real (they're the ones from this book's project). No skill executed here.

Chapter L5.4 — Making Claude sound like you

Level 5 — Skills and identity. Stable concepts; product details from the ledger (verified 24/06/2026).

Goal

By the end you'll know how to compose the identity tools — account instructions, Styles, Projects, context files and Skills — to give Claude a stable, recognizable voice, and you'll do a practical exercise to extract yours. It's the chapter that closes Level 5 by lining up what you've learned.

Prerequisites

- Settings and Styles (ch. L1.3), Projects (ch. L3.2), Skills (ch. L5.1-5.3).

Why voice matters

Claude, without guidance, writes in a correct but neutral register. If you use it to produce texts that carry your name — emails, articles, documents — that neutrality shows: it's the sign that the text isn't "yours". Giving Claude your voice isn't an aesthetic indulgence, it's what makes the result publishable without rewriting it.

Voice doesn't live in one single place. It's composed of several tools, each with a different reach. Choosing them well avoids putting everything everywhere.

The scale of tools

From the lightest to the most structured, here's where identity lives and what it governs.

Table L5.4.1 — The identity tools, from light to structured.

Tool	Reach	For what
Instructions	every chat	stable preferences
Styles	as needed	how to communicate
Projects	a topic/client	isolated context
Skills	wherever needed	repeatable rules

- **Instructions for Claude** (ch. L1.3): personal preferences valid in every chat — name, language, base tone. The most convenient level for what always holds.
- **Styles** (ch. L1.3): they govern *how* Claude communicates; you activate them when needed, and a custom style can learn from your text.
- **Projects** (ch. L3.2): they isolate context and instructions for a topic or a client, without polluting the other chats.
- **Skills** (ch. L5.1-5.3): they package repeatable rules that trigger on their own, in chat, Cowork and Code.

Context files

There's a level that precedes all the others: the **context files**. They are documents where you gather the raw material of your voice — examples of your texts, a list of words to avoid, "before/after" pairs. They aren't a product in themselves: they're the raw material from which a custom Style, a Project's instructions or a voice skill are then born.

This book works that way: a context file gathers examples and style rules, and from there it feeds the `voce` skill and the Project's instructions. Writing the context first avoids scattering the voice across many small, disconnected instructions.

Compose them, don't pile them up

The mistake is putting the same rule in all the tools. Better to assign each one its reach: the always-valid preferences in the Instructions; the "how to write" in the Styles or in a `voce` skill; a client's material in a Project. A rule lives in one place only — the right one — so that, when it changes, you update it once.

Exercise: extract your voice

A concrete path for giving Claude your voice:

1. **Gather** 3-5 real texts of yours (emails, articles, notes). They're the most precious material.
2. **Have the traits extracted:** ask Claude "From these texts of mine, describe my register, the recurring words and the ones I avoid".
3. **Write the context file:** save those traits, with examples and a list of words to avoid.
4. **Choose the container:** for general use, a **custom Style**; for recurring work, a **`voce` skill**; for a client, a **Project**.
5. **Test and correct:** have a text produced, compare it with yours, and refine the context until it "sounds" like you.

> **Tip:** the voice test is simple. Have Claude write a short text and ask yourself: would you sign it? If not, a trait is missing: add it to the context.

Common mistakes

- **Leaving Claude its default register.** On texts that carry your name it shows. Give it your voice.
- **The same rule everywhere.** Assign each tool its reach; one rule, one place.
- **Describing the voice in words.** Real examples are worth more than a thousand adjectives: start from your texts.
- **Stopping at the first rendering.** Voice is refined by iterating on the context file.

Summary

1. Giving Claude your voice makes texts publishable without rewriting them.
2. The tools have different reach: **Instructions** (always), **Styles** (how to write), **Projects** (a topic), **Skills** (repeatable rules).
3. **Context files** are the raw material from which Style, Project and voice skill are born.
4. **Compose** the tools: one rule in one place only.
5. Extract the voice from your **real texts**, then choose the container and iterate.

Next step

You've completed Level 5. In **Level 6 — Advanced** we move on to fine control of the tools, starting from **ch. L6.1 — Advanced Claude Code**: hooks, sub-agents and granular permissions.

Synthesis chapter between Instructions/Styles (ch. L1.3), Projects (ch. L3.2) and Skills (ch. L5.1-5.3). Product details from the ledger, verified on 24/06/2026. No command executed here.

Chapter L6.1 — Advanced Claude Code

Level 6 — Advanced. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know three advanced levers in Claude Code: **hooks** to attach your own commands to events, **sub-agents** to delegate tasks to separate contexts, and granular **permissions**. These are the tools that turn Claude Code from an assistant into an integral part of your workflow.

Prerequisites

- Claude Code installed and a project configured (ch. L2.2, L2.4).
- Comfortable with `settings.json` and basic permissions (ch. L2.4).

Hooks: attaching commands to events

A **hook** is a shell command that Claude Code runs automatically at a given **event** in its lifecycle: before using a tool, after, when a prompt is sent, and so on. They're there to enforce rules you don't want to leave to good intentions: run the linter after every edit, block writes to certain files, log what happened.

You configure them in `settings.json`, organized by **matcher** (the tool's name, a regex, or empty for all). The main events are `PreToolUse` (before a tool is called) and `PostToolUse` (after).

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit",
        "hooks": [
          { "type": "command",
            "command": "npm run lint" }
        ]
      }
    ]
  }
}
```

The exit-code trap

The thing that trips everyone up is the hook's **exit code**. It's not a detail: it changes Claude's behavior.

- **exit 0:** all good, carry on.
- **exit 2: blocking** error. stdout is ignored; **stderr** goes back to Claude as an error message. On `PreToolUse`, the tool call is blocked.
- **other non-zero codes: non**-blocking error, shown to you.

So if you want a hook to **stop** an action and explain to Claude why, you must exit with **exit 2** and write the reason to **stderr**. Getting the code wrong is the most common cause of hooks that "don't block."

Sub-agents: delegating to a separate context

A **sub-agent** is a specialized assistant that Claude Code calls on for a task, with its **own context window**, its own system prompt, and its own tool permissions. It's there to isolate a piece of work (a code review, for example) without cluttering the main conversation.

You define it as a Markdown file with frontmatter in `.claude/agents/` (project) or `~/.claude/agents/` (personal). Claude decides which sub-agent to delegate to by reading the `description`, just like with Skills.

```
---
name: code-reviewer
description: Reviews changes for quality
  and correctness. Use it before committing.
tools: Read, Grep, Glob
---

You are a careful reviewer. Look for logic
errors, edge cases, and readability issues.
```

> **Tip:** the `/agents` command guides you through creating a sub-agent and can generate a first draft from a description.

Granular permissions

Basic permissions (`allow` / `deny`, ch. L2.4) decide what Claude can do without asking. Hooks raise the bar: a `PreToolUse` hook can **evaluate** the individual action and decide at runtime whether to allow it, deny it, or ask for confirmation. So you move from static rules ("never `rm -rf`") to dynamic ones ("block writes outside `src/`"). The principle stays the one from ch. L2.4: grant the repetitive and safe, guard the destructive.

In practice: a hook that blocks

1. Open `.claude/settings.json`.
2. Add a `PreToolUse` hook with the matcher for the tool you want to watch.
3. In the command, check the condition; if it should be blocked, write the reason to stderr and exit with **exit 2**.
4. Start Claude Code and try an action that should be blocked.

5. Check that Claude receives the message and stops.

Common mistakes

- **Hook that doesn't block.** Missing the **exit 2**: with other codes the action proceeds.
- **Message in the wrong place.** On a block, write to **stderr**, not stdout: stdout is ignored.
- **Sub-agent without a clear description.** Claude won't know when to delegate the task: take care with the `description`.
- **Permissions too broad.** Opening everything removes the safety net (ch. L2.4).

Summary

1. **Hooks** run your commands on Claude Code's events, from `settings.json` by matcher.
2. The **exit-code trap**: only **exit 2** blocks, and the reason goes to **stderr**.
3. **Sub-agents** delegate a task to a separate context; Claude picks based on the `description`.
4. Granular permissions: hooks evaluate at runtime what to allow.
5. Philosophy unchanged: grant the safe, guard the destructive.

Next step

In **ch. L6.2 — MCP custom** we'll see how to connect to Claude what doesn't already have a ready connector, and the pattern that pairs a skill with an MCP server.

Data on hooks, sub-agents, and permissions verified on 24/06/2026 against code.claude.com/docs (hooks, sub-agents). The configuration examples were not run here.

Chapter L6.2 — MCP custom

Level 6 — Advanced. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll understand what MCP is in plain terms, when you need a custom MCP server instead of a ready connector, how to add one to Claude Code, and the pattern that combines a skill with an MCP. It's the way to connect Claude to what doesn't already have an integration.

Prerequisites

- Having used connectors (ch. L3.3).
- Claude Code installed (ch. L2.2), for the hands-on part.

What MCP is, in plain terms

MCP (Model Context Protocol) is an **open standard** for connecting Claude to external tools and data. Think of a universal power socket: instead of a custom-built integration for each app, MCP defines a common way for Claude to talk to a service. An **MCP server** is the component that exposes that service's actions to Claude.

The connectors in ch. L3.3 are already MCP under the hood: a **custom connector** is a **remote MCP** (a server reachable over the network). MCP is therefore the technical layer that sits beneath connectors.

When you need a custom MCP

The connector directory covers the most common apps. A custom MCP is needed when:

- the app you need **has no ready connector**;
- you want to connect one of your **internal tools** or a company database;
- you need a specific action that no existing connector offers.

In short: first you search the directory; if it's not there, you build or connect an MCP.

Local or remote

An MCP server can run in two ways, and Claude Code handles both.

Table L6.2.1 — Local and remote MCP compared.

Type	Where it runs	How
Local (stdio)	on your computer	child process
Remote (HTTP)	on a server	over the network

The **local** one (`stdio` transport) is started by Claude Code as a child process that communicates over standard input/output: handy for tools that live on your machine. The **remote** one (`http` transport) is a server reachable over the network: it's the form of custom connectors, reached from Anthropic's cloud (ch. L3.3).

Adding an MCP to Claude Code

From the terminal, the base command is `claude mcp add`. For a local server:

```
claude mcp add --transport stdio \
  my-tool -- npx -y my-mcp-server
```

The flags (`--transport`, `--env`, `--scope`) go **before** the name; the `--` separates the name from the command that starts the server. Environment variables are passed with `--env` (for example a service's API key).

> **Warning:** a custom MCP gives Claude access to an external service. Only connect servers you trust, and don't put secrets in the clear: use environment variables.

The skill + MCP pattern

MCP and Skills solve different problems and complement each other. An **MCP** gives Claude the **capability** to access a service (the data, the actions). A **skill** (Level 5) gives Claude the **method**: when to use that capability, in what order, with what rules.

Example: an MCP exposes your management software; a skill describes the workflow "how to prepare the monthly report from the management software's data." The MCP is the arm, the skill is the procedure. Together they make an integration not just possible, but **repeatable**.

In practice: connect a tool

1. First check the connector directory (ch. L3.3): if it's there, use that.
2. If it's not, obtain or write an MCP server for your tool.
3. Add it with `claude mcp add`, choosing `stdio` (local) or `http` (remote).
4. Pass credentials with `--env`, never in the clear.

5. If you use it recurrently, write a **skill** that describes its workflow.

Common mistakes

- **Building an MCP that already exists.** Check the connector directory first.
- **Flags after the name.** They go before; the `--` separates them from the server's command.
- **Secrets in the clear.** Use `--env` /environment variables, don't write them in the command.
- **MCP without a skill for repeated use.** The MCP gives the capability; the skill gives the method.

Summary

1. **MCP** is the open standard that connects Claude to tools and data; an MCP server exposes a service's actions.
2. **Custom connectors** (ch. L3.3) are **remote MCPs**.
3. You need a custom MCP when no ready connector exists or for internal tools.
4. In Code: `claude mcp add` with `stdio` (local) or `http` (remote) transport.
5. **Skill + MCP**: the MCP gives the capability, the skill the repeatable method.

Next step

In **ch. L6.3 — Automations and remote control** we'll see how to make Claude work on its own over time: scheduled tasks, cloud routines, and control from your phone.

Data on MCP and `claude mcp add` verified on 24/06/2026 against code.claude.com/docs/en/mcp and support.claude.com (custom connector). The commands require Claude Code and an MCP server, so they were not run here.

Chapter L6.3 — Automations and remote control

Level 6 — Advanced. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know how to make Claude work over time without sitting in front of it: **Scheduled Tasks** in Cowork, Claude Code's cloud **Routines**, and **Dispatch**, which turns your phone into a remote control. Above all you'll understand which to choose, because they run in different places.

Prerequisites

- Cowork (ch. L3.1) for scheduled tasks; Claude Code (ch. L2.2) for routines.
- A paid plan.

Three tools, three places

The difference that matters isn't what they do, but **where they run**: that's what decides whether they work with the computer off, and how suited they are to non-technical work.

Table L6.3.1 — Where each tool runs.

Tool	Where it runs	With computer off
Scheduled Task	Cowork, Desktop	no
Routine	Anthropic cloud	yes
Dispatch	your computer	no

Scheduled Tasks

Scheduled Tasks run a Cowork task automatically, on a schedule or on demand: “every morning check my email,” “every Friday prepare the report.” They live in **Cowork on Desktop**, on paid plans.

The limit to know: they run **only with the computer on and the Desktop app open**. If the computer is off or the app is closed at the scheduled time, the task is **skipped** and runs on its own as soon as you turn the computer back on or reopen the app. They’re perfect for someone who leaves the computer on, less so for someone who needs a guarantee that it runs no matter what.

Routines

Claude Code’s **Routines** are saved configurations — a prompt, one or more repositories, and your connectors — that run on **Anthropic’s cloud**, not on your computer. That’s why they work even with the machine off.

Their strong point is **combinable triggers**: the same routine can start on a schedule, in response to an API call, and on a GitHub event, all at once. They’re the tool for development automations that need to be reliable and independent of your machine.

Dispatch

Dispatch turns your phone into a remote control for Claude on your computer. You send a message from the mobile app and Claude runs the task **on your machine**: it reads local files, pulls data from connectors, searches the web, and reports the result back to you. Between the mobile app and the Desktop a **single, persistent thread** opens up, like a walkie-talkie: you assign remotely, Claude works locally.

It's the answer to "I'd like to kick off that job while I'm out": the computer has to be on, but you don't.

Which to choose

- **Non-technical, recurring** work, and you keep the computer on → **Scheduled Task**.
- **Development** automation that must run **always**, even with the machine off, with various triggers → cloud **Routine**.
- You want to **kick off a job remotely** that uses your local files → **Dispatch**.

In practice: schedule a recurring task

1. In Cowork, describe the task as you would for normal work (ch. L3.1).
2. Set the recurrence (for example "every morning at 8").
3. Leave the computer on and the Desktop app open at the scheduled time.
4. For independent development automations, consider a **Routine** in Claude Code instead.
5. To start it from outside, use **Dispatch** from the mobile app.

Common mistakes

- **Scheduled Task with computer off.** It's skipped and runs on reopening: for guaranteed execution use a cloud Routine.
- **Confusing Routine and Scheduled Task.** Routines run on the cloud (Code), Cowork tasks on your Desktop.
- **Expecting Dispatch with the machine off.** It runs locally: the computer has to be on.

Summary

1. The key difference is **where** the tool **runs**.
2. **Scheduled Tasks**: Cowork on Desktop; only with computer on and app open.
3. **Routines**: Anthropic's cloud (Code); always run, combinable triggers.
4. **Dispatch**: you start it from your phone, Claude runs it on your computer (which must be on).
5. Choose based on the reliability required and the type of work.

Next step

In **ch. L6.4 — Managing usage limits** we'll see what consumption depends on and how to work for a long time without hitting the limits.

Data on Scheduled Tasks, Routines, and Dispatch verified on 24/06/2026 against support.claude.com (scheduled tasks, dispatch) and code.claude.com/docs (scheduled-tasks/routines). The features require a paid account, so they were not run here.

Chapter L6.4 — Managing usage limits

Level 6 — Advanced. Product details verified on 02/07/2026 against official sources.

Goal

By the end you'll understand the difference between **usage limit** and **length limit**, what consumption depends on, and how to work for a long time without hitting the limits. This is what lets you use Claude sustainably, especially on heavy tasks.

Prerequisites

- Knowing the plans (ch. F.3) and Projects (ch. L3.2).

Two different limits

Claude has two types of limit, which work in distinct ways:

- **Usage limit:** *how much* you can use Claude in a period. It's your "conversation budget." When it runs out, you wait for the reset.
- **Length limit:** *how long* a single chat can get, that is, the **context window** (Claude's working memory in that conversation).

The first is about quantity over time; the second about the depth of a single chat. Confusing them leads to the wrong choices when something "gets stuck."

What consumption depends on

Usage isn't consumed in fixed chunks. Several factors weigh in: the **length and complexity** of the conversation, the active **features**, the

model chosen, and the **effort level** (how deeply it reasons). A detail that surprises people: all surfaces — claude.ai, Claude Code, Claude Desktop — draw on the **same** usage. They're not separate budgets.

> **Note:** with Code execution on, long chats trigger **automatic context management**: Claude summarizes older messages to keep going. Convenient, but long conversations consume **more** usage.

The context window

In chat, on paid plans, the context window goes up to **1M tokens** with Sonnet 5 (the first model to offer it in chat), is **500K** for the other flagship models (Opus 4.6/4.7/4.8 and Sonnet 4.6) and **200K** for the rest. In **Claude Code** the flagship models go up to **1M tokens**: on Pro, for the million with Opus, you need to enable usage credits. You can't enlarge it at will, but you can use it better: that's where the difference is decided between a chat that holds up and one that saturates.

Cheaper tasks: cutting consumption

A few moves lower consumption for the same result:

- **Use Projects.** They leverage RAG (targeted retrieval): they load into context only what's needed, instead of everything.
- **Short instructions.** Concise Projects and prompts weigh less and deliver more.
- **Remove what you don't need.** Unneeded files in Projects, and unnecessary **tools/connectors**: they're token-intensive, so turn them off when not in use.
- **Lower the effort** and disable **extended thinking** on routine tasks.
- **Choose the right model.** Don't use the most powerful one "just to be safe" (ch. F.3): often you pay more with no benefit.

What to do at the limit

When you hit a limit, the move depends on which one:

- **Usage limit reached:** wait for the **reset**, **upgrade** your plan, or buy **usage credits** (on paid plans) to continue right away.
- **Length limit (chat too long):** open a **new conversation**, or move the material into a **Project** to work on it more efficiently.

> **Note:** usage credits aren't just for "at the limit": some flagship models are used *mostly* this way. When Fable 5 returned (July 2026), for example, the model was included in paid plans only in part and for a few days; once that period ended, access remained via usage credits only.

> **Tip:** if you're in a long chat and near the usage limit, it's often worth starting over in a new chat: you restart with the essential context instead of dragging the whole history along.

In practice: work for a long time without getting stuck

1. For recurring work, put it in a **Project** instead of endless chats.
2. Keep instructions short and remove files you no longer use.
3. Turn off tools and connectors not needed for that session.
4. On simple tasks, lower the effort and turn off extended thinking.
5. If you near the limit in a long chat, **open a new one**.

Common mistakes

- **Confusing the two limits.** If it's the chat that's full, no need to wait for the reset: open a new chat.

- **Keeping everything on.** Unneeded tools and connectors consume context and usage: turn them off.
- **Oversized model.** More power than needed costs more with no better output.
- **Endless chats.** Dragging a huge history along consumes usage: split into chats or use Projects.

Summary

1. **Usage limit** = how much you use over time; **length limit** = how long a chat is (context window).
2. Consumption depends on length, features, **model**, and **effort**; all surfaces count against the **same** usage.
3. Context window in chat: up to **1M** (Sonnet 5), **500K** for the other flagship models, **200K** for the rest; in Claude Code up to **1M** (with usage credits on Pro for Opus).
4. Cut consumption with Projects, short instructions, fewer active tools, lower effort.
5. At the limit: reset/upgrade/**credits** for usage; **new chat**/Project for length.

Next step

In **ch. L6.5 — Claude at work, safely** we'll see how to introduce Claude into an organization: governance, roles, data, and when not to use a personal account.

Data on usage/length limit and context window verified on 02/07/2026 against support.claude.com/en/articles/11647753 and /articles/8606394. Values subject to change: see the ledger and the official sources for updates.

Chapter L6.5 — Claude at work, safely

Level 6 — Advanced. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know what changes when Claude enters an organization: who governs the capabilities, how roles and data are managed, and why a personal account isn't the right choice for business work. This is the chapter that separates individual use from team use.

Prerequisites

- Knowing the plans (ch. F.3) and connectors (ch. L3.3).

Personal or company account

The first decision is also the most important: for the organization's work you use an account **managed by the organization** (Team or Enterprise), not a personal one. The reason isn't a formality. On a company account the data, the access, and the rules are controlled by whoever administers it; on a personal one they stay in the hands of the individual, outside the company's policies. Putting work material on a personal account means taking it outside the perimeter the organization can protect and manage.

Who governs the capabilities

In Team and Enterprise not everything is on for everyone. An **Owner** or **Primary Owner** decides what's available, from **Admin > Capabilities**: Web search, connectors, and other functions are enabled at the organ-

ization level. So if you don't see a function, it's often not an error: it's a choice by whoever administers the account (we already saw this for Web search in ch. L1.3 and for connectors in ch. L3.3).

The same goes for **connector actions**: an admin can allow read-only and block writes, for the entire organization (ch. L3.3).

Roles, identity, and data

Organizations need controls that a single account doesn't have. The table sums up what the business plans add.

Table L6.5.1 — Controls for organizations.

Each column is additive: Enterprise includes what Team has, and adds to it.

Area	Team	Enterprise (extra)
Access	admin, SSO	RBAC, SCIM
Data	no training*	retention, audit log
Network	—	IP allowlisting

*On both Team and Enterprise, by default data isn't used for training; Enterprise adds retention and audit log.

In short: **Team** brings administration, SSO (single sign-on), and controlled deployment; **Enterprise** adds granular roles (RBAC), automatic user provisioning (SCIM), audit log, data retention management, IP allowlisting, and dedicated security features. These are the tools that make use compliant with company policies, not left to the individual.

The permission model, again

One principle runs through the whole book: Claude **inherits your permissions**. It holds for connectors (it sees only what you see at the source, ch. L3.3) and for file work (it touches only the connected folders, ch. L3.1). In a company this is a guarantee: the controls of the source system stay in force, and Claude doesn't override them. Restricting in Claude never grants more access than the source allows.

In practice: introducing Claude into a company

1. Use an account **managed by the organization**, not a personal one.
2. Have an Owner define the available **capabilities** (Admin > Capabilities).
3. Set up **access**: SSO and, in Enterprise, roles (RBAC) and provisioning (SCIM).
4. Configure **connector restrictions** based on what the team can read or modify (ch. L3.3).
5. Agree on the rules for **data** and retention before putting in sensitive material.

Common mistakes

- **Business work on a personal account.** It takes data outside the organization's control: use the managed account.
- **Thinking it's a bug when a function is missing.** Often an admin has disabled it (ch. L1.3, L3.3).
- **Connectors without restrictions.** In a team, define what's read-only and what isn't (ch. L3.3).
- **Postponing the data rules.** Agree on them first, not after uploading sensitive material.

Summary

1. For business work use an account **managed by the organization**, not a personal one.
2. An **Owner** governs the capabilities from **Admin > Capabilities**.
3. **Team** adds admin and SSO; **Enterprise** adds RBAC, SCIM, audit log, retention, IP allowlisting.
4. Claude **inherits permissions**: the source's controls stay in force.
5. Define access, connector restrictions, and data rules before you start.

Next step

In **ch. L6.6 — Integrating via API** we close the technical level: how to move from interfaces to code, with the messages endpoint and a first example.

Data on Team/Enterprise controls (RBAC, SCIM, SSO, audit, retention) verified on 24/06/2026 against claude.com/pricing and support.claude.com. Independent publication, not affiliated with Anthropic.

Chapter L6.6 — Integrating via API

Level 6 — Advanced. Product details verified on 24/06/2026 against official sources.

Goal

By the end you'll know when to move to the API, what a call to the messages endpoint looks like, and how **tool use** works. It's the bridge from someone who *uses* Claude to someone who *integrates* it into their own software.

Prerequisites

- An API key from the Console (ch. L2.3, F.3).
- Familiarity with a programming language.

When you need the API

The interfaces (chat, Cowork, Code) are enough for personal work. The **API** is for when you want to put Claude **inside** a product of yours: an app that answers customers, a process that classifies documents, a service that generates text on demand. It's programmatic access, pay-as-you-go by token (ch. F.3).

The messages endpoint

The heart of the API is one endpoint: you send a list of messages, Claude generates the next one. The raw call, with `curl`:

```
curl https://api.anthropic.com/v1/messages \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "content-type: application/json" \
  -d '{
    "model": "claude-opus-4-8",
    "max_tokens": 1024,
    "messages": [
      {"role": "user",
       "content": "Hello"}
    ]
  }'
```

Three headers are mandatory: `x-api-key` (your key), `anthropic-version` (e.g. `2023-06-01`), and `content-type`. The body declares at least `model`, `max_tokens`, and `messages`. (VOLATILE: the model string changes, see the ledger.)

With the SDK

In practice you don’t use `curl`, but an official SDK (Python or TypeScript), which handles headers and formats. In Python:

```
import anthropic

# Reads the key from ANTHROPIC_API_KEY
client = anthropic.Anthropic()

msg = client.messages.create(
    model="claude-opus-4-8",
    max_tokens=1024,
    messages=[
        {"role": "user",
         "content": "Hello, who are you?"},
    ],
)
print(msg.content[0].text)
```

Tool use, in brief

Tool use lets Claude use tools you provide — a function that queries a database, calls an API, runs a calculation. The flow is a back-and-forth in three steps:

1. You define the tools in the request and send the message.
2. If a tool is needed, Claude responds with `stop_reason: "tool_use"` and a block that says **which** tool and with **which** arguments.
3. Your code runs the operation and returns the result as a `tool_result`; Claude uses it for the final answer.

Figure L6.6.1 — The tool use cycle. Alt text: vertical diagram from the message to the tool request, to execution in your code, to the result, to the final answer.

The key point: the tool **you run yourself**, in your software. Claude decides when to use it and with which arguments, but the code stays under your control.

In practice: your first call

1. Generate an **API key** in the Console and set it as an environment variable (ch. L2.3).
2. Install the SDK for your language (Python or TypeScript).
3. Make a minimal call to the messages endpoint and read the response.
4. To integrate actions, define a **tool** and handle the `tool_use` → `tool_result` cycle.
5. Keep the key out of the code: use environment variables (ch. L2.3).

Common mistakes

- **Missing headers.** You need `x-api-key`, `anthropic-version`, and `content-type`.
- **Hardcoded model string.** It changes over time: take it from the ledger or the official sources.
- **Key in the code.** Never: use an environment variable (ch. L2.3).
- **Expecting Claude to run the tool.** You run it; Claude decides when and how to call it.

Summary

1. The API is for integrating Claude **inside** your software; pay-as-you-go.
2. Endpoint `POST /v1/messages` with headers `x-api-key`, `anthropic-version`, `content-type`.
3. In practice you use an **SDK** (Python/TS), not `curl`.

4. **Tool use** is a cycle: Claude asks for a tool, your code runs it and returns the result.
5. The key lives in an **environment variable**, never in the code.

Next step

You've completed Level 6. In the **Closing**, **ch. C.1 — End-to-end project** runs through all the levels on a real case, from design to code to automation.

Data on the endpoint, headers, and tool use verified on 24/06/2026 against platform.claude.com/docs (messages, tool use). The examples were not run here; they require a valid API key.

Chapter C.1 — End-to-end project

Closing — variable level. Synthesis concepts; product details from the ledger (verified 24/06/2026).

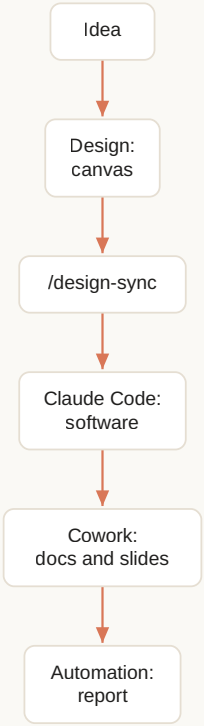
Goal

This chapter runs through all the levels of the book on a real case: from an idea to a small product, passing through Design, Claude Code, Cowork, and automation. It introduces nothing new: it shows the pieces you've already seen working **together**.

The case

Imagine you want to build a small internal app: a page where the team logs customer requests and tracks their status. We'll take it from the idea all the way to a report that updates on its own. Each stage points back to the chapter where you learned it.

Figure C.1.1 — The complete flow, from idea to automation. Alt text: vertical diagram connecting idea, Design, /design-sync, Code, Cowork, and automation in sequence.



1. Idea and foundations

You start from chat (Level 1). Describe the idea and get help bringing it into focus: what the page should do, for whom, with which fields. The rule from ch. L1.2 applies — purpose, audience, format — and so does the one from ch. L1.1: the first answer is a draft to refine. Out of this comes a short requirements document.

2. From design to code

You open **Claude Design** (ch. L4.1) and generate the page on the canvas, starting from your design system if you've imported it (ch. L4.2). You refine with chat, comments, and direct edits. When the look holds

up, you use `/design-sync` and the **handoff** to **Claude Code** (ch. L4.3): the design becomes real software, without rebuilding it from a screenshot.

In Claude Code you work on the real project, with `CLAUDE.md` and permissions configured (ch. L2.4). For the delicate steps, a **hook** enforces the checks and a **sub-agent** reviews the code before the commit (ch. L6.1). If the app needs to read from one of your internal tools, you connect it via **MCP** (ch. L6.2).

3. Documents and organization

With the app up and running, you need supporting material: a usage guide, a presentation for the team. Here **Cowork** comes in (ch. L3.1): you connect the project folder and hand it the work as an **end-state** — "from these files, generate a `.docx` guide and a `.pptx` presentation." The methods from Level 3 apply: content first, then the file (ch. L3.4), and structure → content → style for the slides (ch. L3.5).

A project **skill** (Level 5) holds the rules together — voice, structure, file names — so the output is consistent every time (ch. L5.3).

4. Automation

Finally, the report that updates on its own. With a **Scheduled Task** in Cowork (ch. L6.3) you have Claude reread the requests every Friday and produce a summary. If you need it independent of your machine, you use a cloud **Routine**; if you want to kick it off from outside, **Dispatch** from your phone. And if the app grows to the point of needing a real integration, you move to the **API** (ch. L6.6).

Throughout the journey you keep an eye on the **usage limits** (ch. L6.4): Projects for context, clean sessions, tools on only when they're needed.

What it demonstrates

The value isn't in the individual tool, but in the handoff: Design passes to Code, Code works with Cowork on the same folder, a skill makes it all repeatable, automation closes the loop. It's the difference, seen all the way back in ch. F.2, between a sum of tools and an ecosystem.

In practice: try a mini-project

1. In chat, bring a small, concrete idea into focus (ch. L1.2).
2. Generate the interface in Design and hand it off to Code (ch. L4.3).
3. Configure the project in Claude Code (ch. L2.4).
4. In Cowork, generate a guide and slides from the folder (ch. L3.4, L3.5).
5. Add an automation for the recurring report (ch. L6.3).

Summary

1. A real project runs through all the levels: chat, Design, Code, Cowork, automation.
2. **Design** → **/design-sync** → **Code** takes the interface to software without rebuilding it.
3. **Cowork** produces documents and slides from the project's own folder.
4. A **skill** makes the output consistent; **automation** keeps it up to date.
5. The value is in the handoff between the products, not in the individual tool.

Next step

In **ch. C.2 — Appendices** you'll find the glossary, a troubleshooting table, and the official links, to consult when you need them.

Synthesis chapter: it recalls the products covered in Levels 1-6. No command run here; the cross-references point to the chapters where each step is explained.

Chapter C.2 — Appendices

Closing — reference material. Product details verified on 24/06/2026 against official sources.

How to use these appendices

They aren't read in sequence: they're consulted. You'll find a glossary of the recurring terms, a troubleshooting table for the most common problems, and the official links from which to verify the volatile data.

Glossary

- **Agentic:** a way of working in which Claude carries a task forward over several steps, not just a single answer.
- **API:** programmatic access to the models, to integrate Claude into your own software (ch. L6.6).
- **Canvas:** Claude Design's canvas where the generated interfaces appear (ch. L4.1).
- **Connector:** a link that gives Claude access to an app, with your permissions (ch. L3.3).
- **Context window:** the working memory of a single chat, measured in tokens (ch. L6.4).
- **Cowork:** non-code agentic work in the desktop app, in an isolated VM (ch. L3.1).
- **Design / `/design-sync`:** the canvas and the command that syncs design and code both ways (ch. L4.3).
- **Dispatch:** starting from your phone a task that Claude runs on your computer (ch. L6.3).
- **Effort level:** how deeply Claude reasons; higher consumes more usage (ch. L6.4).

- **Handoff:** passing a design to Claude Code so it becomes software (ch. L4.3).
- **Hook:** a command attached to a Claude Code event (ch. L6.1).
- **MCP:** open standard that connects Claude to external tools and data (ch. L6.2).
- **Native installer:** the recommended method to install Claude Code, without Node (ch. L2.2).
- **OAuth:** delegated access via the browser, without typing the password in the terminal (ch. L2.3).
- **Prompt:** the request you write to Claude (ch. L1.2).
- **Project:** a workspace with instructions and a knowledge base shared across chats (ch. L3.2).
- **Routine:** a Claude Code automation that runs on the cloud (ch. L6.3).
- **Scheduled Task:** a recurring Cowork task, runs with the computer on (ch. L6.3).
- **Skill /** `SKILL.md` **:** reusable knowledge that Claude loads on its own (ch. L5.1).
- **Sub-agent:** an assistant with its own context and permissions, that Code delegates to (ch. L6.1).
- **Tool use:** the use, via API, of tools you provide (ch. L6.6).
- **Usage limit:** how much you can use Claude in a period (ch. L6.4).
- **Isolated VM:** the virtual machine Cowork runs in; it sees only the connected folders (ch. L3.1).

Troubleshooting

Table C.2.1 — Common problems and first solution.

Problem	Cause	Solution
`command not found`	PATH	new terminal (L2.2)
Login failed	extra API key	`unset` the key (L2.3)
Function missing	admin	Admin > Capabilities (L1.3)
Hook doesn't block	exit code	use exit 2 (L6.1)
Connector inert	not activated	toggle in chat (L3.3)
Task skipped	PC off	reopen the app (L6.3)
Chat saturated	length limit	new chat (L6.4)

Official links

From here you verify the data that changes over time. These are the only sources this book relies on for product details.

- **claude.ai** — access to Claude (web).
- **claude.com** and **claude.com/pricing** — products and plans.
- **support.claude.com** — official guides and troubleshooting.
- **code.claude.com/docs** — Claude Code documentation.
- **platform.claude.com/docs** — API and Console documentation.
- **agentskills.io** — the open standard for Skills.
- **github.com/anthropics/skills** — example skills.

Legal notes

Independent publication, not affiliated with or endorsed by Anthropic. Claude and Anthropic are trademarks of their respective owners. The product details (versions, commands, prices, menus, plans) are **verified as of the date indicated** in each chapter and in the ledger, and are **subject to change**: for current values refer to the official sources above and to the companion repo.

Summary

1. The **glossary** collects the recurring terms, with the cross-reference to the chapter.
2. The **troubleshooting** table gives the first move for common problems.
3. The **official links** are the only sources for the volatile data.
4. The publication is **independent** and not affiliated with Anthropic.
5. The data is verified as of the date indicated and should be rechecked against the sources.

The end

You've gone through the whole journey: from your first messages in chat all the way to the API and automation. The product will keep changing — that's why the volatile data is dated and collected in the companion repo. The underlying ideas, on the other hand, stay: describe the result, give context, iterate, and make the pieces of the ecosystem work together.

Reference material. The glossary and cross-references are evergreen; troubleshooting, links, and product details verified on 24/06/2026 against the official sources listed.