Claude Code as a Daily Assistant: Setup Guide

Despot Bitschnau

arcneo GmbH, Berlin

Last updated: September 2026

Abstract

This guide shows how I use Claude Code in the terminal as a personal everyday agent. Not for programming, but for email, research, notes, branding, and organization. It walks chronologically through the complete setup on a Mac, from installing the terminal to concrete use cases you can adopt right away. One important caveat: this is deliberately **not** my setup for coding agents. My coding setups include everything described here, but a lot more on top (project-specific rules, test and verification pipelines, CI integration, and additional tooling). This guide covers the setup for non-coding tasks only.

How to get the most out of this guide: Once the basic installation is in place (chapters 2 and 3), hand this document to your own Claude agent in the terminal. Have it store the guide, work through it step by step, and install and configure everything you are still missing: folder structure, CLAUDE.md, vault, connectors, plugins. From that moment on, "How do I set up X?" just works, and all you do is confirm. But do read the guide yourself anyway. Much of it is not about installation but about usage (workflows, habits, use cases), and you should understand what is happening on your machine and what is new. The two do not exclude each other: while you read, your agent can already start installing things in parallel.

---

1. Why Claude Code in the Terminal Instead of Chat in the Browser?

Claude Code is Claude with hands. The chat on claude.ai can write texts for you, but Claude Code can additionally:

- Read and write files on your machine (e.g. your Obsidian notes)
- Run programs, search the web, build PDFs
- Work directly in Gmail, Google Calendar, Notion, Figma, etc. via connectors
- Permanently remember who you are and how you work, via a CLAUDE.md file

The effect: instead of copying answers around, the agent gets things done. "Summarize today's emails and file the note in Obsidian" is a single sentence.

And Why Not Claude Cowork?

With **Claude Cowork**, Anthropic offers an agent variant directly inside the Claude desktop app, no terminal, no installation. For an absolute first taste it is fine, but for daily use as described here, Claude Code is the better choice:

- **Works directly on your real files.** Cowork runs in an isolated virtual machine; your files first have to go into that sandbox and results have to come back out. Claude Code works directly in your file system, for example live inside your Obsidian vault. This entire setup is built on exactly that.
- **Gets things done in the background without blocking you.** A Claude Code task keeps running in its terminal tab while you continue working normally on your machine. Claude never needs to take over your screen or your applications; it simply pings you when it is done.
- **Plugins, skills, and custom commands.** Claude Code is freely extensible: install plugins, build your own slash commands and hooks, fine-tune permissions. Cowork only ships with a fixed, pre-installed feature set.
- **Multiple instances in parallel, under control.** My core workflow, one named and color-coded iTerm2 tab per topic with its own session each, only works in the terminal. That way three or four agents run side by side without getting mixed up.
- **Full configurability.** CLAUDE.md files per folder, custom settings, language and voice configuration, status line: everything is adjustable.

In short: Cowork is the rental car with automatic transmission, Claude Code is your own car. If you just want a quick look at what the agent can do, start with Cowork. If you want the daily workflow described here, install Claude Code, and the rest of this guide shows that it takes less than 15 minutes.

---

2. Step 1: The Right Terminal

On a Mac, use **iTerm2**, not the pre-installed Terminal.app. Download: iterm2.com (free). Just drag the app into your Applications folder.

Why iTerm2:

- **Named tabs, automatically.** I usually have several tabs open in parallel, and each one carries the name of its topic. The agent sets the name itself via `/rename` (chapter 9 explains how). For the name to actually show up in the tab, iTerm2 needs two adjustments. First: stop setting tab titles by hand. A manually set title always takes precedence in iTerm2 and blocks every rename coming from Claude Code; the session is then named correctly internally, but the tab keeps showing the old name. If you still have manual titles, clear them once: right-click the tab, "Edit Tab Title", empty the field, confirm. Second: in Settings, Profiles, General, in the "Title" field tick only "Session Name" and untick "Job", otherwise every name gets a "(node)" appended. Colors I assign by hand: right-click, "Edit Tab Color".
- **Native notifications.** Claude Code can notify you via iTerm2 when a long task finishes while you are in another window. It happens twice over: a small pop-up in the top right corner of the screen plus a short notification sound. You miss nothing, no matter where you are looking.

- **Split panes.** Cmd+D splits the window, handy when an agent is working and you want to look something up alongside.
- **Paste images with Ctrl+V.** Screenshots or copied images go straight into the prompt when you press Ctrl+V in the empty input field (not Cmd+V, which only pastes text). For this to work in iTerm2, iTerm2 has to allow applications to access the clipboard, and that is off by default. Once, in Settings, General, Selection, tick "Applications in terminal may access clipboard", then quit iTerm2 completely and reopen it (Cmd+Q, not just closing the window). Without that checkbox, Ctrl+V simply does nothing and you end up looking for the problem in the wrong place.

Windows and Linux: Tabby. iTerm2 is Mac only. On Windows or Linux, use **Tabby** (tabby.sh, free and open source). Tabby does exactly what matters here: named and colored tabs, split panes, images from the clipboard. On Windows you open a PowerShell session inside it, which is where Claude Code runs natively. The pre-installed Windows Terminal works too, but with Tabby you stay closest to what this guide describes.

A typical beginner mistake: people stay in Terminal.app and wonder why notifications and colors do not work.

---

3. Step 2: Installing Claude Code

Mac and Linux. Open iTerm2 and paste this line (press Enter):

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

Windows. Open a PowerShell session in Tabby and paste this line:

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

So on Windows it is not the curl command, it is `irm`. Copy the curl line out of a Mac tutorial and you get an error message, then you look for the problem in the wrong place.

Either way, Claude Code is installed natively, without needing Node.js or Homebrew first. Afterwards, close the terminal completely and reopen it.

If you later get "command not found" (on Windows: "is not recognized as the name of a cmdlet"), the terminal was not restarted completely. Do not just open a new tab, quit the program and start it again; the installer only registers the path for new sessions.

The Folder First, Then Claude

Before you start Claude for the first time, create the folder your assistant is going to live in. It looks like a detail and it is the most important switch in the whole setup: Claude reads identity, rules, and memory

from the folder you started it in. Start it somewhere else and it is a blank sheet.

Name the folder after your assistant, or simply `Assistant`. Mine is called `Assistant`, and the agent inside it is Falco.

Mac and Linux:

```
mkdir ~/Assistant
cd ~/Assistant
claude
```

Windows (PowerShell):

```
mkdir $HOME\Assistant
cd $HOME\Assistant
claude
```

From here on, without exception: Claude gets started in that folder and nowhere else.

Signing In and Choosing a Team

On the first start of `claude`, the login opens in your browser:

1. Sign in with your **Claude account** (the same one as on claude.ai). You need a Pro, Max, or Team plan.
2. Careful, the second classic mistake: at login there are two paths, "Claude account" (subscription) and "Console account" (API with credits). For daily use you want **Claude account**, otherwise you pay per request instead of through your subscription.
3. If you belong to several organizations (personal + company): after logging in, select the right **team**. If you ended up in the wrong one, type `/login` in Claude and sign in again; you can switch organizations there.

Quick sanity check: start `claude` and ask something simple ("Which files are on my desktop?"). If Claude answers and asks for permission before acting, everything is working.

Your Own Start Command

You will type `cd ~/Assistant && claude` diligently for a few days and then, out of convenience, start Claude somewhere else. So set up your own command right away, one that does both at once: change into the folder and start Claude. Name it after your assistant; mine is `falco`.

Mac and Linux (zsh):

```
echo 'falco() { ( cd "$HOME/Assistant" && claude "$@" ) }' >> ~/.zshrc
source ~/.zshrc
```

Windows (PowerShell): create and open the profile once,

```
if (!(Test-Path $PROFILE)) { New-Item -ItemType File -Path $PROFILE -Force }
notepad $PROFILE
```

then write this line into it, save, and restart the terminal:

```
function falco { Push-Location $HOME\Assistant; claude @args; Pop-Location }
```

From then on `falco` is all it takes, no matter which folder you are in. The command grows with you later: mine also attaches two Google Drive folders that Claude is allowed to access, via `--add-dir`.

```
falco() {
  ( cd "$HOME/Assistant" && claude --add-dir "$HOME/path/to/folder1" --add-d
}
```

---

4. Step 3: Basic Configuration

In Claude Code you type commands that start with `/`. The most important ones for setup:

/config: The Control Center

`/config` opens the settings. There you configure:

- **Theme:** dark or light color scheme, depending on your terminal look. Just try them out.
- **Language:** set it to your preferred language and Claude will always answer in it, no matter how you ask. (The config file `~/.claude/settings.json` will then contain e.g. `"language": "German"`.)
- **Notifications:** set them to iTerm2 so you get pinged when a task is done, as a small pop-up in the top right plus a short sound (see chapter 2).

Model and Effort: Which Level for What

Two dials decide how hard the agent works, and most people only know one of them.

Dial 1: the model. With `/model` (or in `/config`) you pick between the Claude models. Roughly: **Sonnet** is the fast worker, **Opus** the strong all-rounder, **Fable** the top tier for tasks where a mistake really hurts. On top of that there is `/fast`, which puts Opus into a faster output mode without dropping down to a smaller model.

Dial 2: the effort. Independent of the model, you set how much thinking budget the agent gets per answer. In daily use this is often the bigger lever than switching models: a mid-tier model with more effort regularly beats a top-tier model answering in a rush.

This is how I run it every day, and almost always on **medium effort**:

- **Sonnet, medium:** routine. Drafting emails, building lists, filing documents, calendar items, small stuff. Nobody needs a heavyweight for that.
- **Opus, medium:** my default for anything that needs a bit of thinking. Researching a competitor and then implementing right away, analyses, multi-step assignments. 90 percent of my day happens here.
- **Fable, medium:** when it counts. Hunting for security holes, a final pass over a document before it goes out, a contract text, a delicate investor email. Whenever an overlooked mistake gets expensive.
- **Fable plus ultrathink or ultracode:** planning and architecture. How do we build this new app, what structure, what tasks, what security concept. Those are the moments where the highest escalation level actually pays off (more on that in chapter 9).

The rule of thumb I built from this: **turn up the effort first, switch the model second.** And pick the level by the cost of an error, not by gut feeling. A wrong list is fixed in ten seconds. An email to an investor or a missed security hole is not.

One thing to keep an eye on: the high levels burn through your quota, and `ultracode` with its swarm of subagents burns a lot more than anything else. Run at maximum all day and you are out of budget by the afternoon.

And a point for teams: when you build your own skills (chapter 6), write down **which model tier the skill was built and tested for.** Anything with external impact, so emails, LinkedIn posts, client documents, is otherwise hard to reproduce, because the same script comes out differently on Sonnet than on Fable. We put it right at the top of every skill: our team skills are tested on **Fable 5.0 at medium effort** and run just as cleanly on **Opus 4.8**. Run them on a smaller tier and you will get results, but not necessarily the ones the skill was calibrated for.

Voice: Talking to Claude

This changed my workflow more than anything else: I dictate almost everything instead of typing. Enable **Voice Mode** in `/config`. I use the "Hold" mode: hold a key, speak, release, done. Dictating is much faster than typing, especially for longer tasks ("Write an email to X, mention Y, keep the tone casual..."). Combined with your language setting, it feels like a conversation with an assistant.

If you want to go one step further, get **SuperWhisper** (superwhisper.com): a standalone dictation tool for the Mac based on the Whisper models. Its recognition is another notch better than the built-in Voice Mode, especially with technical terms and mixed German/English. Two things to know: the **free version is limited** (the better models and the full feature set are behind the subscription), and the real killer feature is that it works **system-wide**, not just in Claude. Once set up, you dictate everywhere: emails, notes, messages, any text field. Once you are used to dictating with AI, you will not want it confined to the terminal anyway.

Permissions

At first, Claude asks for permission before every action. That is good in the beginning, to get a feel for it. Later you can choose "always allow" in the permission dialog or set a more relaxed mode in `/config`. My recommendation: confirm everything for the first week, then loosen up.

The single most important move: **Shift+Tab**. It switches the mode within the running session; on "auto-accept", Claude executes actions directly without asking every time. I do this **first thing in every session**, otherwise you just sit there clicking confirmations away. The mode only applies to the current session, so make Shift+Tab your first keystroke after starting. What makes this safe: underneath the mode sits a fixed deny list in my setup that applies even in auto mode, see chapter 12.

The CLAUDE.md: Giving the Agent an Identity

In your working folder (e.g. `~/Assistant/`) you create a file called `CLAUDE.md`. Claude reads it automatically on every start and follows it. Mine has grown to nine sections over the months:

1. **Identity and persona.** Who the agent is and how it sounds. Mine is called Falco and answers with a dry Viennese streak; a dedicated section at the end collects phrases and no-gos that I keep extending. That is a matter of taste, but it makes the daily work more pleasant.
2. **About me.** Name, email addresses, company, languages. The basics the agent would otherwise have to ask for in every session.
3. **How we work together.** How answers should look: short and direct, no filler, make assumptions instead of asking, end with one or two sentences on what happened.
4. **Context folder.** A curated knowledge folder for the agent (about me, the company, reference material) with a README as index that it reads first on relevant questions.
5. **Inbox workflow.** What happens when I drop a file into the `inbox/` folder: read it, file it into the right context folder, update the index, report in one sentence.
6. **Filing rules.** Which content belongs in my private vault, which in the company's team vault, and what stays scratch material (more on the vaults in the Obsidian chapter).
7. **Wiki maintenance.** The rules by which the agent treats the vault as a maintained wiki: set cross-links, flag contradictions, keep a changelog (more on that in the Obsidian chapter too).
8. **External resources.** Where things live and how the agent reaches them: the two Google Drives, the vault, the connectors.
9. **Hard boundaries.** What never happens without checking back: no sending emails, company drive read-only, never delete permanently.

You do not have to start with nine sections. My first version was four lines; the file grows with every rule you would otherwise have to explain twice. The more precise the CLAUDE.md, the less you have to re-explain in every session.

My everyday start command: `falco` (the custom start command from chapter 3), or one tab simply stays open in the Assistant folder permanently.

The Memory: What the Agent Remembers on Its Own

Besides the CLAUDE.md, which I write and curate myself, Claude Code keeps its own memory directory: small Markdown files, one per fact, that the agent creates on its own whenever it learns something about me or how I like to work. I do not have to do anything except correct it occasionally. Mine holds around 40 entries by now, in four groups. An excerpt:

About me

1. **Dictation facts.** Certain speech-to-text mishearings always mean the same word; the agent knows that and no longer asks.
2. **Signature.** Available as an image file, inserted into documents on request.
3. **Third email address.** Besides the two main accounts there is a third address, including the context of what it is for.
4. **F1 interest and space interest.** Personal topics, so research on them lands in the right vault folder right away.
5. …

My feedback rules

1. **Never send emails.** Emails are only ever prepared as drafts; sending happens only after my OK.
2. **No em dashes.** The character does not appear in texts written for me.
3. **Sheets cell by cell.** Google Sheets are edited cell by cell, never cleared and rewritten.
4. **Dev in English.** Everything developer and GitHub related (commits, READMEs, docs) in English.
5. **LinkedIn through the team skill.** Posts are always created through our shared skill, never freehand.
6. …

Company

1. Domain facts and wording rules around arcneo that the agent must keep consistent in texts. The contents themselves do not belong in this guide.
2. …

Projects and references

1. **Calendar.** Which calendar connector runs through which account, and which one is read-only.
2. **Personal hub.** My private dashboard project whose data the agent maintains.
3. …

The difference from the CLAUDE.md: the CLAUDE.md is maintained by me, the memory grows on its own. The effect is the same as with a good employee, every correction sticks the first time, and after a few weeks the agent feels like it knows you. Wrong entries I simply have deleted, that is one sentence in the chat.

One detail worth knowing: the memory directory does not live in the vault but globally under `~/.claude/`. It therefore does not follow you to a second machine on its own. In my setup it is symlinked into the repo, how that works is in chapter 7.

5. Step 4: Connectors (MCP), Plugging Claude into Your Tools

Connectors link Claude to external services. Most of them you set up once on **claude.ai** (Settings, Connectors); after that they are automatically available in Claude Code in the terminal as well. In Claude Code, `/mcp` shows you what is connected.

My connectors and what I use them for:

Connector	What for
Gmail	Search emails, summarize threads, create drafts in my own style
Google Calendar	Find and create events, time suggestions ("When am I free on Friday?")
Google Drive	Search and read documents (in my case: company drive read-only)
Notion	Read and create pages, query databases
Figma	Read AND create designs (my LinkedIn posts and pitch decks are made this way)
HubSpot	CRM queries, contacts, campaigns
GitHub	Repos, issues, pull requests (in my case via the `gh` CLI, which Claude operates directly)
Vercel	Deploy and manage websites, push changes live straight from the terminal
Supabase	Query and manage databases and backends
Brave Search	Web search straight from the terminal
Context7	Up-to-date documentation for tools and libraries
Namecheap	Manage the DNS records of my domain (set up locally)

A tip from practice: only connect what you actually use. Every connector is an access the agent has. For sensitive systems (e.g. the company drive), put a read-only rule in your CLAUDE.md.

---

6. Step 5: Plugins and Skills

Plugins extend Claude Code with ready-made capabilities ("skills"). Install them in Claude Code via `/plugin`, where you can browse the marketplace. What I run:

- **Superpowers** (official): **the absolute must-have, install it first.** Superpowers fundamentally changes *how* Claude works: instead of charging ahead, Claude first clarifies the requirements, plans, then implements and finally verifies its own result. The difference is noticeable on every non-trivial task: less rework, fewer "that is not what I meant" moments. If you install only a single plugin, make it this one.
- **Firecrawl** (CLI + skills): **the second clear recommendation, specifically for research.** My web toolbox: searching, cleanly extracting web pages, downloading entire documentation sites, monitoring competitors' pricing pages. The highlight is the deep-research skills, which deliver reports with real source citations instead of half-knowledge from the model's memory. Firecrawl is the foundation for all the research use cases further below. Installation: register at firecrawl.dev, then install the CLI with the skills; from then on Claude can "read the web" at any time.
- **Frontend-Design** (official): ensures high-quality design that does not look templated when Claude builds documents, pages, or graphics.
- **arcneo-branding** (our own plugin): my brand skills, which produce LinkedIn cards, one-pagers, and pitch-deck slides directly in Figma in the company CI. The point behind it: you can build **your own plugins** for your own recurring tasks and share them within the team. For branding tasks in particular, this pays off enormously.
- **Game-Sounds** (a toy): sound effects on certain events. Not necessary, but fun.
- **handelsregister** (our own plugin, open source): pulls German commercial register documents for any German company directly from the official portal, i.e. shareholder lists, articles of association, and register excerpts (AD/CD/SI). Useful for KYC and due diligence. It exists because the portal has been free since 2022 but has no API. The code is public on GitHub (DespotB/handelsregister-cli). Anyone can install it straight from there, entirely without our Hub: `/plugin marketplace add DespotB/handelsregister-cli`, then `/plugin install handelsregister@handelsregister-cli`. After that, a single `/handelsregister Beispiel GmbH` is enough and the documents land in a folder, cleanly named and deduplicated.

On top of that come **slash commands**: custom shortcuts like `/note` (write a daily note into the vault) or `/dns` (DNS change with a before/after check). They are simple text files in the `.claude/commands/` folder; anyone can build their own, just ask Claude to do it.

From Solo Setup to Team

How to extend this plugin principle to the whole company (building and maintaining vetted skills centrally, instead of everyone tinkering with their own variants on their own machine) is covered in chapter 8.

---

7. Step 6: Obsidian as Memory (Second Brain)

Claude Code can read and write files. Obsidian stores notes as plain Markdown files in a folder (a "vault"). This combination is the core of my setup: **Claude works directly in your Obsidian vault**, no

plugin, no interface, simply through the file system.

1. Install Obsidian: obsidian.md (free)
2. Create a vault, e.g. at `~/Assistant/vault/`
3. Choose a sensible folder structure. Mine:

```
vault/
├── Daily/                    Daily notes and journal
├── Mail/                     Email style analysis, contacts, archive
├── Work/
│   ├── arcneo/               Current company, subfolders mirror the shared
│   │   ├── 04 HR
│   │   ├── 06 Legal
│   │   ├── 07 Partners
│   │   ├── 08 Marketing & Sales
│   │   ├── 10 Events
│   │   ├── 11 Funding
│   │   └── 14 Agents-Workspace
│   └── Karriere & Zukunft/   Work-related, but not current
└── Privat/
    ├── Freunde & Familie/    Friends & family
    ├── Wohnung & Verträge/   Home & contracts
    ├── Hobbys & Freizeit/    Hobbies & leisure, e.g. DJ/, Gaming/
    ├── Bildung & Wissen/     Learning & knowledge
    └── Organisation/         Occasions, planning, speeches
```

A tip from practice: name the work folder exactly like the company's shared drive. Then Claude finds things in both places under the same path name.

For context: this is my **private** vault: my structure, my machine, my backup. Alongside it there is a second Obsidian at our company: the **shared team vault**, a common Git repo that is part of the arcneo Hub (more on that in chapter 8). It is deliberately cut differently, not by areas of life but by knowledge types: `Meetings/` (all minutes, named `YYYY-MM-DD Topic.md`), `Wissen/` (completed research, concepts, decisions), `Projekte/`, `Vorlagen/`, `Archiv/` (outdated material gets flagged instead of deleted), and a `00 Inbox` as a drop zone. The dividing line is hard: private matters stay in the private vault, company topics belong in the team vault, and neither one links into the other.

Important: my Obsidian needs **no community plugins**. Everything plugins would normally do (templates, auto-linking, imports) is done by Claude. Obsidian is just the nice viewer with graph view and linking; Claude is the author.

Three expansion stages for the vault (GitHub backup, second machine, the vault as a maintained wiki) are collected in chapter 12.

---

8. Step 7: The arcneo Hub, a Notion for the Agents

Since August 2026 we have a central, private Git repo called the **arcneo Hub** (`arcneo-gmbh/arcneo-hub`). The shortest definition: **what Notion is for the humans on the team, the Hub is for the agents**: the one place holding everything that every Claude agent in the company needs to know and be able to do. The principle behind it: build once, use across the whole team. The repo is the single source of truth; skills, prompts, and team knowledge are maintained and versioned there, not on individual machines.

What Is Inside

Folder	Contents
`skills/`	Vetted team skills, installable as the `arcneo-skills` plugin (LinkedIn posts, meeting minutes, investor FAQ, …)
`prompts/`	Prompt library for everyone who only works with the chat on claude.ai (no Claude Code required)
`vault/`	The shared Obsidian vault: `Meetings/`, `Wissen/`, `Projekte/`, `Vorlagen/`, `Archiv/` (see chapter 7)
`onboarding/`	The Agent Workspace: template + guided setup for the personal assistant
`docs/`	How-tos: How do I write a skill? What goes where?
`.claude-plugin/`	The marketplace definition, makes the repo installable as a plugin source

Installation per person: `/plugin marketplace add arcneo-gmbh/arcneo-hub`, then `/plugin install arcneo-skills@arcneo-hub`. The marketplace also ships the branding plugin and the handelsregister plugin along with it. Updates come via `/plugin marketplace update arcneo-hub`. New and improved skills reach everyone automatically that way.

Why We Have It

- **One vetted version instead of ten homegrown ones.** A skill is built once, reviewed, and then used by everyone. If someone improves a skill, the whole team gets it with the next update.
- **Shared memory.** The team vault (all meeting minutes, all completed knowledge) is automatically available context for every agent on the team. A colleague's agent knows what was decided in my meeting last week.
- **Onboarding in hours instead of days.** New team members clone the repo, start `claude` inside it, and say "Run the onboarding". The agent sets up plugins, vaults, and connectors itself.
- **Usable without a terminal too.** The prompt library covers those who only work with the chat. The Hub is for the whole team, not just the power users.

What Goes Where (The Filing Matrix)

The Hub replaces neither Notion nor Google Drive. The systems have clear roles, and one of the most important pages in the Hub is exactly this decision matrix (`docs/ablage-regeln.md`), which humans **and** agents follow. The rule of thumb: **Markdown knowledge into the repo, living things into Notion, binaries into Drive.**

Why not simply put all knowledge into Notion? Because the two places are built for different readers. Agents simply find their way around a vault of Markdown files better than around Notion: they read the file system directly, search hundreds of pages in seconds, follow links without an API detour, and via Git they even see when which piece of knowledge changed. That means faster answers on a better information basis. And the side effect matters just as much: Notion stays clean. Knowledge that primarily exists for agents (research, process documentation, background context) would only clutter Notion; that is where humans should find their living boards and databases, not a hundred reference pages no human ever opens.

Content	Place
Meeting minutes, completed knowledge (research, concepts, decisions)	**Hub vault**
Running tasks, task boards, living databases (e.g. legal-questions tracker, pipelines)	**Notion**
Lead lists, contact data	**Notion / HubSpot** (personal data does not belong in a Git repo)
Binary files: PDFs, contracts, renderings, presentations	**Drive**
Credentials, API keys	**Password manager** (never in the repo, Notion, or Drive)
Private matters	**Your own local vault**, never in team systems

Some things deliberately live in two places, but only following the **master-copy pattern**: one place is always the master, the other a dated copy. Example: a living Notion database that agents need as context gets a dated Markdown export in the vault; in case of conflict, Notion wins. Or: contracts stay as PDFs in Drive, an index in the vault links to them, and only what agents constantly need as text is additionally converted to Markdown, with a source reference. What is not allowed: maintaining the same information in two places in parallel. That creates two truths, and eventually neither is correct.

How to Build This for Your Own Company

The Hub is not a product, it is a pattern. And rebuilding it is itself a Claude task, not a weekend project:

1. **Create a private GitHub repo** (e.g. `yourcompany/yourcompany-hub`), invite the team members.
2. **Three core folders:** `skills/`, `prompts/`, `vault/`.
3. **Have Claude build the marketplace definition:** "Turn this repo into a Claude Code plugin marketplace that ships the skills from `skills/` as a plugin." From then on the repo is installable via `/plugin marketplace add`.
4. **Distill the first two or three skills** from the tasks that recur most often in the team (for us: meeting minutes, LinkedIn posts, investor FAQ).
5. **Write a filing-rules page** so it is clear from the start what goes into the repo, what into Notion, what into Drive.
6. **Onboard the team:** clone the repo, add the marketplace, open `vault/` in Obsidian (with the community plugin Git for auto-sync).

One rule applies from day one, without exception: no passwords, no private content, no sensitive personnel data in the repo. The rest grows by itself. Every meeting note and every new skill makes the Hub more valuable.

---

9. My Daily Workflow

This is what a normal day looks like for me:

1. **iTerm2 with named, color-coded tabs.** One tab per context, e.g. blue = "Falco" (personal assistant), green = "arcneo" (company), orange = "Research". Each tab is its own Claude session with its own conversation history. The agent names the tabs: in Claude Code, the command `/rename <word>` renames the session in both places at once, in the terminal tab at the top and in the session list you see in Remote Control on your phone. Without renaming, Claude assigns random names there like "Polish Blossom" that tell you nothing. So I added a rule to my agent's CLAUDE.md: as soon as the first topic of a new session is clear, it automatically names the session with exactly one concise word (technically via a small script that types `/rename` into its own iTerm2 window using AppleScript). The effect is simple but worth gold: the tab at the top always matches the session name exactly, and when I check in from my phone via Remote Control, I immediately know which session belongs to which topic and which tab.
2. **Voice instead of typing.** Hold the key, dictate the task, release. Especially on the move between meetings, it is the fastest way.
3. **Everything important lands in the vault.** Claude writes results (research, briefings, analyses) as Markdown notes with links into the matching Obsidian folder. In the evening I browse through Obsidian; everything is there and linked.
4. **Long tasks run in the background.** Claude keeps working, iTerm2 pings when it is done. Meanwhile, the next topic in the next tab.
5. **Keep the context window clean, always.** This matters more than it sounds: Claude has a limited "memory" per session (the context window). The longer a session runs and the more topics get mixed into it, the slower, more expensive, and less focused the agent becomes. Hence the iron rule: **new**

topic = new tab, or `/clear` in the existing tab. `/clear` empties the conversation history and starts fresh (the CLAUDE.md and all settings are of course preserved). Never squeeze a new topic into a full session.

6. **For very demanding tasks: ultrathink.** A magic keyword: write the word "ultrathink" anywhere in your task and Claude switches into its deepest thinking mode. Claude then gets the maximum budget of "thinking time" and deliberates extensively before answering or acting: it weighs alternatives, checks its own logic, and plans several steps ahead. The word even gets rainbow-colored in the terminal so you can see it was recognized. Unnecessary for everyday tasks (it takes longer), but for complex things, say a delicate negotiation email, a strategic plan, or a tricky analysis, it makes a noticeable difference in quality. Example: "ultrathink: Draft the reply to the investor, here is the thread ..."
7. **And the highest escalation level: ultracode.** The second magic keyword. If "**ultracode**" appears in your task, Claude no longer works alone but orchestrates a whole swarm of subagents: the task gets decomposed, worked on in parallel, and the results check each other before anything comes back. For context: Superpowers already does exactly this on a small scale anyway (individual subagents for research or review). **ultracode** turns it up to dozens of agents, correspondingly more expensive and slower. So reserve it for extremely demanding tasks, say a full audit, a large piece of research across many sources, or an analysis that truly needs to go deep.
8. **The agent improves itself.** Whenever something goes wrong or I correct a preference, I have Claude update the CLAUDE.md. Every mistake happens only once.

---

10. Use Cases from Practice (To Rebuild)

The Personal Email Style Guide

My favorite example. I gave Claude my complete email archive (around 2,500 emails from three years, via a Google Takeout export). Claude read all of it and distilled a **style bible** from it: how I address people (when "Hallo", when "Sehr geehrte"), how I sign off, sentence structure, favorite words, what I never do (emojis, stock phrases), even the differences between my German and English emails. The result is a note `Mail/Stil.md` in the vault, plus a contact list and few-shot examples.

Since then the rule is: whenever Claude drafts an email in my name, it reads the style guide first. The drafts sound like me, not like AI. One afternoon of effort, once; value, daily.

Competitor Research: Charts for the Fintech Platform

For our fintech platform we wanted to know how competitors visualize fixed-interest bonds: which chart types they use, how interest curves and repayment are displayed, what is standard and what is not. Claude researched the competitors' platforms and wrote the results straight into Notion, as a complete page with an overview per competitor, the chart types they use, and a recommendation for our platform.

Afterwards, Claude created first example charts in our branding in Figma, as a basis for the design discussion. From question to presentable page in one session, without me doing any of the research myself.

Branding and Content Directly in Figma

Branding and content are a **dedicated plugin** for us (`arcneo-branding`, shipped via the Hub marketplace) with three skills that work directly in Figma: **LinkedIn cards** (posts, carousels, speaker cards, banners, and related formats), **pitch-deck slides** (16:9), and the **one-pager** (A4 fact sheet for investors). The foundation underneath: a self-written **branding guidebook** (colors, typography scale, layout patterns, format specs) that each of the three skills reads before working, plus a collection of real **example slides and designs** that the skills recycle or use for orientation. "Make an announcement post for the trade fair" thus turns into a finished design in the correct CI (colors, fonts, logo placement), ready for fine-tuning instead of building from scratch. For the **copy** there is the separate team skill `linkedin-post` in the Hub, which can write various post types (trade-fair announcements, product updates, milestones, founder posts), based on a style bible with real reference posts, analogous to the email style.

Guides from Your Own Learnings

The meta use case: this very guide came about exactly that way. Claude wrote it, out of everything that had accumulated in the setup over the months (vault, CLAUDE.md, workflows, solved problems). A single prompt is enough ("Write a beginner's guide based on my setup"), because the knowledge is already with Claude: there is nothing to research, only condensing and putting it into a readable order. The pattern works for anything you want to pass on: internally as how-we-work documents for the team (e.g. how we work sales, how we follow up on meetings), externally as a guide like this one. Once written down, such a guide lives on as a normal Markdown file in the vault or hub and simply gets maintained along with future changes.

Administration and Small Stuff

- **DNS and domain management:** through two connectors, the Vercel MCP and the Namecheap MCP, Claude is wired directly into my domains (personal and company). Concretely, that means: I can edit and publish content on my websites straight from the terminal, and I can change DNS records on demand, with a built-in before/after check. This also covers the email security part: Claude sets up and verifies SPF, DKIM, and DMARC properly on the domains, so my emails do not land in spam and nobody can send in my name.
- **Event radar:** Claude watches industry events and maintains a list in the vault.
- **LinkedIn network upkeep:** a small startup check reminds Claude once a day to grow my LinkedIn network via the Chrome extension (a handful of connection requests to fitting profiles, with a hard daily limit and a log in the vault). The pattern behind it is reusable: a hook checks at startup whether a daily routine has already run, and nudges Claude if not.
- **Meeting follow-up:** I dictate notes, Claude structures them and files them in the right company folder.

---

11. The Most Common Beginner Mistakes

1. **Not restarting the terminal after installation** → "command not found". Just quit iTerm2 completely and reopen it.
2. **Signing in with the Console/API account instead of the Claude subscription** → unexpected per-request costs. Choose "Claude account" at login; fix it with `/login`.
3. **Landing in the wrong team/organization** → sign in again with `/login` and choose the right organization.
4. **Working without a CLAUDE.md** → every session starts from zero. The half hour spent on a good CLAUDE.md is the best investment in the whole setup.
5. **Starting Claude in the home directory instead of the working folder** → Claude does not see its context. Always `cd ~/Assistant` (or wherever your setup lives) first, then `claude`. Easiest with the custom start command from chapter 3, which does both in one.
6. **Doing everything in one single session** → the history gets long and messy. One tab per topic; on a topic switch, `/clear` or a new session.
7. **Connecting every connector "because you can"** → unnecessary access. Only connect what is needed, and grant write permissions deliberately.
8. **Ctrl+V does not paste an image** → either you pressed Cmd+V instead of Ctrl+V, or iTerm2 is not allowed to access the clipboard. Tick "Applications in terminal may access clipboard" (Settings, General, Selection) and restart iTerm2 completely (see chapter 2).
9. **Writing hard limits only into the CLAUDE.md** → that is a request to the model, not protection. Whatever must never happen (sending mail, recursive deletes, production deploys) also belongs in the settings as a deny rule or hook (chapter 12).

---

12. Advanced: Once the Basic Setup Is Running

Everything in this chapter is optional. It pays off once the setup from chapters 2 to 8 has been running in daily use for a few weeks and you notice where it pinches: backup, a second machine, a vault that maintains itself, and permissions that hold even when the agent reads content from strangers. The order is also my recommended order for tackling them.

Advanced: Backing Up the Vault as a GitHub Repo

Once the basic setup is running, it is worth going one step further: turn the vault folder into a **private GitHub repository**. You do not need to know how to do this yourself; it is itself a Claude task ("Turn my vault into a private GitHub repo and set it up so changes are backed up automatically"). What you get:

- **Backup and future-proofing.** Every note is versioned. Nothing gets lost anymore, even if your machine dies, and you can restore any old version of any note.

- **Multiple machines.** On a second Mac (personal/work), clone the repo and the same vault with the same CLAUDE.md is available there. But that is only part of what the agent knows about you. What comes along and what does not is covered in the next section.
- **Claude commits by itself.** My CLAUDE.md contains the rule that Claude automatically commits and pushes after every change in the vault. The backup happens as a side effect, without me ever thinking about it.

Sensitive folders (e.g. the private email archive) can be excluded from the repo via `.gitignore`; they then stay local only.

Advanced: Second Machine and Migration, What Comes Along and What Does Not

The repo covers only part of what Claude Code knows about you. The rest lives in your home directory under `~/.claude/` and does not sync by itself. How it splits in my setup:

- **Comes along via the repo:** the vault, the project CLAUDE.md, the project commands and hooks in the `.claude/` folder of the working directory.
- **Lives globally in `~/.claude/` and does not come along by itself:** the automatically grown memory (chapter 4), the global CLAUDE.md, the settings (language, model, status line, permission rules and hooks), your own helper scripts, installed plugins and skills.
- **Stays local on purpose:** the file `~/.claude.json`, which sits directly in the home directory (not inside `~/.claude/`) and holds login, MCP servers including API keys and the trust state of your folders, plus the session transcripts.

Two pitfalls you need to know. First, `~/.claude.json` sits outside `~/.claude/`. If you only copy the folder, you lose login and MCP servers. Second, Claude Code names the memory directory and the session transcripts after the absolute path of the working directory (in my case `~/.claude/projects/-Users-despot-b-Assistant/`). A different username or a different folder path on the second machine, and the memory is there but never found. So use the same username and the same path on every device.

My solution: the parts that should come along live in the repo in a folder `claude-home/` (memory, global CLAUDE.md, scripts, a snapshot of the settings) and are symlinked into `~/.claude/`. The agent thus writes its memory straight into the repo, and the commit rule from the CLAUDE.md backs it up along with everything else. On a new machine that means: clone the repo, run the script `claude-home/link.sh` (creates the symlinks), start `claude`, log in, set up MCP servers and plugins again. Claude set this up for me, including script and README, one request in the chat. For a one-time move without a repo, rsync is enough: copy `~/.claude/` and `~/.claude.json` to the new machine. With the same username and path everything keeps working, only the login is due once more, because the credentials live in the macOS keychain and not in the files.

Advanced: The Vault as a Maintained Wiki (Three Layers)

The next maturity level, inspired by Andrej Karpathy's "LLM wiki" idea: the vault is not just storage but a wiki that the agent actively maintains. The setup has three layers:

1. **Raw sources (`vault/raw/`).** Articles, PDFs, and web clips land here and are never modified. A practical feeder: the **Obsidian Web Clipper** browser extension turns any article into a Markdown file. In Obsidian, set the attachment folder to `raw/assets/` and bind the "Download attachments for current file" action to a hotkey (Ctrl+Shift+D in my case); then all images are stored locally too and Claude can look at them.
2. **The wiki (the rest of the vault).** The notes written by Claude, linked to each other. The rule that makes the difference: when ingesting a new source, Claude does not just file a summary, it also **updates the existing pages**. It adds cross-references, flags contradictions with older knowledge, and creates dedicated pages for recurring topics. Knowledge gets condensed once and stays current, instead of being re-assembled for every question (that is the difference from classic RAG or NotebookLM-style tools).
3. **The schema (the CLAUDE.md).** That is where exactly these maintenance rules live, so every session follows them automatically. You never write the wiki yourself: you supply sources and ask questions, Claude does the bookkeeping. Obsidian is the reading view (the graph view shows you the structure), Claude is the author.

Two small helpers go with this, which Claude builds for itself on request:

- `vault/log.md`: an append-only chronicle (what was ingested, researched, checked, and when). It gives every new session instant context about the state of the wiki.
- `/lint-vault`: a custom command that periodically checks the wiki for orphaned pages, dead links, contradictions, and missing cross-references, and proposes fixes.

The effect: good answers no longer disappear into the chat history; they get written back into the wiki as linked notes. Knowledge compounds, every piece of research makes the next one better. (Karpathy's original idea: gist.github.com/karpathy/442a6bf555914893e9891c11519de94f)

Advanced: Enforcing Permissions Technically

Chapter 4 says Shift+Tab is my first keystroke in every session. That is convenient, but with the connector set from chapter 5 it is also a different order of magnitude than writing notes: the agent can send mail, write to databases, deploy websites, and change DNS records. At the same time it constantly reads content from strangers (mail, web clips, scraped pages), and foreign text can contain instructions that did not come from you. The hard rules in the CLAUDE.md ("never send mail", "company Drive is read-only", "no rm -rf") are no protection against that. They are a request to the model, which it follows almost always, but not guaranteed.

Three things actually hold, all in the file `~/.claude/settings.json`:

1. **Deny rules.** A list of tools and commands Claude may never run, whatever the mode. Mine includes the send functions of the Gmail connector, recursive deletes (`rm -rf`), `git push --force`, any write into the company Drive, and reading key files such as `~/.ssh`. A deny rule always wins, even against "always allow" and even in auto mode.
2. **Ask rules.** A list of actions where Claude always asks, even with Shift+Tab active. Mine: DNS changes, database migrations, deployments to the live website, deleting mail or calendar events,

sharing Drive files, reading `.env` files with credentials. Everything that acts outward or is hard to undo.

3. **A hook.** Deny rules only check the beginning of a command. What comes after (`cd x && rm -rf y`, a script inside a heredoc) they do not see. That is what hooks are for: a small script that runs before every shell command, receives the full command, and can reject it. Mine checks by regex for recursive deletes in any spelling, force pushes, writing commands containing the path of the company Drive (reading and copying out of the Drive stays allowed), and sending mail via AppleScript or the command line.

This is the core of my file (abridged):

```
{
  "permissions": {
    "deny": [
      "Bash(rm -rf*)",
      "Bash(git push --force*)",
      "Edit(//Users/despot_b/Library/CloudStorage/GoogleDrive-despot@arcneo.
      "mcp__claude_ai_Gmail__send_message",
      "Read(~/.ssh/**)"
    ],
    "ask": [
      "mcp__namecheap__update_dns_record",
      "mcp__claude_ai_Supabase__apply_migration",
      "Bash(vercel deploy --prod*)",
      "mcp__claude_ai_Gmail__trash_message"
    ]
  },
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "/Users/despot_b/.claude/hooks/gua
        ]
      }
    ]
  }
}
```

Three things worth knowing:

- **Global rather than per project.** The file `~/.claude/settings.json` applies in every working folder and also to every subagent Claude starts itself (otherwise those only inherit the session's allow list). Project-specific rules go into `.claude/settings.json` in the respective working folder. That fits the one-tab-per-topic workflow from chapter 9: the coding folder may do different things than the assistant folder, because each folder has its own rules file.

- **This is a Claude task.** You do not need to learn the rule syntax. "Read my CLAUDE.md and turn every hard limit in it into a deny rule, ask rule, or hook, with tests" is enough. Then have it checked: "Try to copy a file into the company Drive." The rejection has to come, with a reason.
- **The rule of thumb.** Every "never" in the CLAUDE.md gets a technical counterpart. When a new hard rule is added, it goes into the settings the same day. Part of that is least privilege for access: my DNS connector can only read and write DNS records, not cancel domains; the company Drive is read-only by rule; write access goes only to the connector that needs it (see chapter 5).

One side effect: the hook also rejects commands that merely *contain* the forbidden words, for example when Claude wants to write a text with "osascript" and "send" into a file. Claude then falls back to its file tools, which is intended. Better rejected once too often than once too little.