# AI Master Guide

## Welcome

Most people open Claude, type a question, and start over every single time. Half of their tokens go to explaining who they are, what they're working on, and what they need. Then they hit the limit, open a new chat, and do it all again tomorrow.

This guide fixes that.

It's for anyone who pays for Claude, ChatGPT, or Gemini — or is about to — and senses there's more on the table than one-off answers. No coding background needed. If you can write an email, you can do everything in here.

By the end, you'll have a real system — folders, files, and prompts — that takes you from *"I asked Claude for help and got a generic answer"* to *"I point Claude at my project and it picks up where I left off."* Read it straight through once (about fifteen minutes), then come back with a real project and follow along with your hands on the keyboard.

Let's go.

# The Beginning

Three things to install. Do this once. You never redo it.

**1) An AI subscription.** Claude (recommended), ChatGPT, or Gemini. Any $20/month tier is enough to start. Free tiers hit limits fast once you're actually using these tools. Every example here uses Claude — the folder system is universal, only specific commands like `/plugin` change.

**2) A code editor.** This sounds scarier than it is. A code editor is just a smart text editor: it lets you open a folder, edit files, and run tools like Claude Code inside it. Pick **VS Code** (free, most popular), **Cursor** (VS Code with AI baked in), or **Antigravity** (newer, worth a look). Not sure? Pick VS Code. You can switch later.

**3) Node.js.** Required if you want to use Claude Code, the command-line tool this guide is built around. Free install from `nodejs.org`. If you plan to stay in the desktop or web app only, skip this.

Follow each installer. They walk you through every step. Nothing custom, nothing tricky. Once all three are running, you're done with setup forever.

# Your First Folder

The single biggest upgrade you can make to Claude is teaching it where to find context. That starts with a folder.

Create one somewhere you can find it again — Desktop works fine — and name it after the project you're working on. Inside it, create three plain-text files with the `.md` extension (that's just markdown: text with light formatting).

## a) CLAUDE.md — who Claude is working for

```
# Identity
You are helping {your name} with {what you do}.

## Rules
Write in plain, clear language.
Ask clarifying questions before making assumptions.
When you are unsure, say so.
```

Fill in your name and the kind of work you do. Don't overthink the rules — you'll refine them as you go.

## b) CONTEXT.md — what you're working on right now

```
# Current Project

## What are we building
{Describe the project in 2 or 3 sentences.}

## What good looks like
{What does a successful output look like? Paste an example if you have one.}

## What to avoid
{Common mistakes or things you don't want.}
```

This file changes over time as your project moves. That's the whole idea — keep it current.

## c) REFERENCES.md — background materials

```
# References

## Example of good work
{Paste an example or describe what you liked.}

## Relevant links
{URLs, docs, resources for this project.}

## Notes
{Anything else Claude should know.}
```

**Heads up:** REFERENCES.md is great for a first simple project. Once you graduate to the three-layer workspace setup later in this guide, references typically live inside each workspace's own folder instead of in one file at the root.

## Point Claude at your folder

**Claude Code (terminal):** navigate to the folder and run `claude`. Claude picks up CLAUDE.md automatically.

**Claude desktop app:** open the app, go to the "Code" section, and select your project folder.

Ask a quick test question — something like *"What's this project about, and what do you need from me to get started?"* — and you should immediately feel the difference. Claude responds as if it already knows you.

**Folder structure is the foundation.** Everything else in this guide is built on top of it. If you only ever do *this* part, you're already ahead of 90% of people using AI tools.

# Prompting

Every really useful prompt has up to five parts. You won't always use all five — but when something isn't working, this is the checklist to run.

## The 5-Part Framework

**1) Identity — who is Claude right now?** Tell Claude what role to fill. This shapes vocabulary, depth, and the assumptions it makes. *"You are a senior copywriter who writes for B2B SaaS companies."* If you set up CLAUDE.md, identity is already handled — but for one-off tasks or role shifts mid-conversation, put it at the top. Without identity, output is always generic.

**2) Task — what needs to get done?** Be specific. *"Help me write something"* is vague. *"Write a 200-word product description for our new onboarding feature, targeting mid-market HR directors"* is not. Good tasks have a clear action (write, analyze, compare, fix), a defined scope (how long, how many, what format), and enough detail that a stranger could start on it without five follow-up questions.

**3) Context — what does Claude need to know?** The background, the constraints, the audience, prior decisions. If you're using the folder system, CONTEXT.md handles project-level context. For individual prompts, give context directly. **Guessing is where AI outputs go sideways.** If an output feels generic or off-target, the fix is almost always more context — not a cleverer prompt.

**4) Constraints — what should Claude avoid?** This is where most people leave value on the table. *"Don't use jargon. Write at an 8th-grade level."* *"No solutions that require a paid API."* *"Under 300 words, no bullet points."* *"Don't start with 'In today's world.'"* Every constraint you set is a mistake Claude won't make. Every common annoyance you have with AI output is an unset constraint.

**5) Output Format — what should the result look like?** Tell Claude the shape of the answer. A list? A table? Three options? A draft with placeholders? *"Three headline options, each under 10 words, with a one-sentence explanation."* *"A markdown table with columns for Task, Owner, Deadline, Status."* Output format is the difference between something you can use immediately and something you have to reformat for 20 minutes.

## One Prompt, Five Parts — a full example

```
(Identity) You are a technical writer who explains complex topics
to a non-technical audience.

(Task) Write a 300-word explanation of how API keys work and why
someone would need one.

(Context) This is for a community learning AI tools. Most have never
written code. They're encountering API keys for the first time because
they're setting up Claude Code.

(Constraints) Do not use jargon without explaining it. Do not assume
they know what a server is. Keep sentences short.

(Output Format) Start with a one-sentence analogy, then explain the
concept, then give them three steps to get their first API key. End
with a one-line reassurance that it's easier than it sounds.
```

You won't write prompts this long every time. But when responses feel vague or unhelpful, run this checklist: *what am I missing?*

## Cheat sheet

| Task Type | Prompt Parts |
|---|---|
| Simple task | Task only. |
| Creative work | Identity + Task + Constraints + Format |
| Complex work | All five. |
| Ongoing projects | Identity and Context in files; Task and Constraints with each prompt. |

## Chunking — breaking up big jobs

If your project is bigger than a single prompt can handle, break it into steps. **Each prompt should ask for one clear thing.**

*Too much at once: "Write me a full marketing strategy with a content calendar, email sequence, and social posts for next quarter."* Claude will try to do all of it, and the output will be shallow across the board.

*Chunked into steps:* outline Q2 themes → [you review, pick a direction] → draft a four-week calendar for theme 1 → [you review] → write the first email in the nurture sequence. Each step builds on the last. When something goes wrong, you redo one step, not the whole project.

## Feeding Claude large inputs

For a long document or dataset, break the input too. **Step 1:** give Claude the structure first — *"I'm going to give you a 40-page report in sections. Here's the table of contents."* **Step 2:** feed sections in order, and after each one ask Claude to confirm what it picked up. **Step 3:** after the last section, ask Claude to work across everything.

This works better than pasting 50 pages into one message — Claude handles structured, sequential input much more reliably than a wall of text. If your data is already structured (spreadsheet, table, CSV), **don't convert it to prose.** Structure is good.

# Workspaces & Routing

You've got a folder and good prompts. What happens when a project has many kinds of work inside it — writing, building, publishing — or when you've got five projects running at once? This is where the real system kicks in.

## Tokens, and why this matters

A **token** is roughly three-quarters of a word. Sometimes a whole word. Sometimes a long word like *"hamburger"* counts as three tokens. There are only so many tokens an AI can hold in its **context window** — its active memory — before it starts forgetting, slowing down, or failing. Every file Claude reads, every message you send, every response it writes — all tokens.

Which means **what Claude reads matters as much as what you ask it.** That's why folder structure is the single biggest upgrade you can make.

## The 3-layer routing system

Instead of one giant conversation or one massive CONTEXT.md, you break your work into separate **workspaces**. Each workspace handles a different kind of work. The system has three layers.

**Layer 1 — The Map (CLAUDE.md).** Your top-level file, at the root of your project. The first thing Claude reads every time. CLAUDE.md describes what the project is, what the folder structure looks like, naming conventions, where things go. Most importantly, it holds a **routing table**: for a given task, these are the files to read, the files to skip, and the skills to use. This is the most important pattern in the whole system.

**Layer 2 — The Rooms (Workspace Context).** Each workspace has its own CONTEXT.md. When Claude goes to work in a specific room, it reads that room's CONTEXT: what the workspace is for, what the process looks like, what files live there, which skills to use. Plain English. A few paragraphs.

**Layer 3 — The Tools (Skills).** Pre-built recipes you plug into a workspace. Don't load every skill in every workspace — you can have 100 skills available in a project, but each workspace loads only the ones it needs. This is the plug-and-play idea.

## Routing saves tokens

Remember the token limit. Here's what it looks like in practice.

**Without routing:** Claude enters your project, reads CLAUDE.md, then (to be safe) opens all four of your CONTEXT.md files plus a few reference documents. Roughly **8,000 tokens** burned before you've even asked a question.

**With routing:** Claude reads CLAUDE.md, sees the routing table say *"for writing tasks, read only /script-lab/CONTEXT.md,"* and opens one file. Roughly **2,000 tokens.** Same answer. A quarter of the cost. A lot more room for your actual work.

## A worked example — the content creator

Say you make videos, write posts, and manage your own social presence. Your work cycles through ideas, scripts, production, and publishing.

```
my-content-project/
├── CLAUDE.md            # what this project is and how
│                          to route between workspaces
│
├── script-lab/
│   ├── CONTEXT.md       # voice, audience, process from
│   │                      idea to finished script
│   ├── ideas/
│   ├── drafts/
│   └── final/
│
├── production/
│   ├── CONTEXT.md       # production process, tools,
│   │                      visual standards
│   ├── briefs/
│   ├── specs/
│   └── output/
│
└── distribution/
    ├── CONTEXT.md       # platforms, cadence,
    │                      per-channel rules
    ├── platforms/
    └── scheduling/
```

The CLAUDE.md at the root might look like this:

```
# My Content Project
I create {type of content} for {audience}.

## Workspaces
- /script-lab      ideas, writing, drafts
- /production      building and producing content
- /distribution    publishing, scheduling, repurposing

## Routing
| TASK                | GO-TO         | READ       |
| ------------------- | ------------- | ---------- |
| Write or brainstorm | /script-lab   | CONTEXT.md |
| Build or produce    | /production   | CONTEXT.md |
| Publish or repurpose | /distribution | CONTEXT.md |

## Naming Conventions
- Drafts: topic-name_draft.md
- Final scripts: topic-name_final.md
- Published: YYYY-MM-platform-topic.md
```

Now *"Let's work on the new video script in /script-lab"* is all you need. Claude reads the main CLAUDE.md, routes to `/script-lab/CONTEXT.md`, and ignores production and distribution entirely. Tight, focused, cheap on tokens, and Claude already knows the voice and audience.

## Designing your own — four steps

**1) List your workspaces.** The 2–4 major areas of your work. Writing and Building are different modes. Client A and Client B are different clients. *If you've ever wished Claude would "forget what it was just doing," that's workspace territory.*

**2) Write a CONTEXT.md for each.** Describe what happens in this workspace, what the process is, what files live there, what good looks like. Keep each one under a page.

**3) Write your CLAUDE.md.** List the workspaces. Build the routing table. Include naming conventions so Claude organizes files without any code — just by following the pattern you gave it.

**4) Start working.** Point Claude at the folder and give it a real task. Notice how much more context it has. Adjust your CONTEXT files based on what Claude gets right and wrong. The system gets better every iteration.

## Skills and MCP

The word *skill* sounds technical. It really isn't. **A skill is a pre-packaged recipe Claude can follow** — someone figured out the best way to write a client proposal or audit a landing page, packaged the instructions, and shared them. When you *add* a skill, you're handing Claude a new recipe.

You may also hear **MCP server**. MCP is just the plumbing that lets Claude talk to outside tools — Google Drive, your calendar, a database. We'll cover MCP in depth in a future guide. For now, just know it exists.

### Adding your first skill in Claude Code:

1. Start a Claude Code session.
2. Type `/plugin` and press enter.
3. Browse the marketplace. Skill names look like `market-landing`, `ads-google`, `market-emails`. Each has a short description.
4. Pick one that matches work you actually do, and install it.
5. In the relevant workspace's CONTEXT.md, add a line: *"For this work, use the `market-landing` skill."*

That's the whole process. No coding. No config files.

> **Rule for beginners: add one skill at a time.** Test it on a real task. See if the output feels right. *Then* add another. Stacking five skills on day one is how you lose the plot — when something goes wrong, you won't know which one caused it.

# The 7 Common Mistakes

Most problems with Claude come from one of these seven patterns. Skim now, come back when something feels off.

**1) Making CLAUDE.md too long.** Burns tokens. Muddies context. Loses the strictness of what matters most — especially the routing table. *Target: one page, max.*

**2) Skipping the routing table.** Without it, Claude guesses what to read or opens everything. Both are bad. Keep it simple: three columns (Task, Go-to, Read), four once skills are added.

**3) Too many workspaces.** Every workspace is a mental shift for you and a file to maintain. *Start with 2–3. Expand only when a real, repeated need shows up — not a theoretical one.*

**4) Writing CONTEXT files about AI instead of about the work.** *"Claude should be creative and bold"* is almost useless. *"Our audience is CTOs at 50–500 person companies; they've heard every AI pitch and win with concrete numbers, not adjectives"* actually changes the output. **80% describing the work. 20% or less on behavioral instructions.**

**5) Never updating CONTEXT.** Projects evolve; context needs to evolve with them, or it quietly becomes wrong — and wrong context is worse than no context. Add a "Last Updated" line. *This is the single highest-leverage thing you can do — five minutes a week keeps the whole system sharp.*

**6) Putting everything in one folder and hoping.** See #3. When things feel tangled and Claude keeps asking the same clarifications, it's almost always a workspace problem, not a prompt problem.

**7) Building the whole system before using it.** Don't design a perfect six-workspace architecture before you've tried a simple one. **Start rough. Let real tasks tell you what to add.** Perfection is a trap. Iteration is the move.

# Your First Week

You just read a lot. Here's how to turn it into a working habit.

**Day 1 — Subscribe and install.** Pick a subscription (Claude recommended). Any $20/month tier is enough. Install the desktop app and your code editor.

**Day 2 — Create your first folder.** Open your code editor. Make a folder named after a real project you're working on right now. **Not a sample. Not a test. A real project.** You'll only feel the system click on something you actually care about.

**Day 3 — Write CLAUDE.md and CONTEXT.md.** Use the templates. Keep each file under a page. Don't overthink it — the point is to have something to iterate on, not to get it perfect.

**Days 4–7 — Use it for real.** Give Claude one real task you were going to do this week anyway. Use the 5-part prompt framework. See what happens. Then update CONTEXT.md based on what Claude got right and wrong.

That's the whole move. You're not trying to master the system this week — you're trying to feel the difference between prompting Claude with no context and prompting Claude with structure.

## If you get stuck

- **TerraByte HQ on Discord** — ask the community. Someone has probably hit the same wall.
- **terrabyte.vip** — more guides, templates, resources.
- **@terrabyte.dev on Instagram** — quick tips and short-form breakdowns.

## One last thing

It gets easier every week. Your first CLAUDE.md will feel clumsy. Your second will be better. By the fourth iteration, you'll wonder how you ever worked without one. Iterate weekly. Fix what's wrong. Keep what works.

**What comes next:** a follow-up guide on **Skills and Workspace Templates** — going deeper on Layer 3, sharing templates across projects, and automating the parts of your workflow that repeat the most.

Until then — build your first folder. That’s the whole game.