HERMES AI FOR BEGINNERS

Your Complete Guide to Running an AI Agent That Actually Works

by Michael Gannotti

Hermes AI for Beginners

Michael Gannotti

Table of Contents

HERMES AI FOR BEGINNERS

Your Complete Guide to Running

an AI Agent That Actually Works

Michael Gannotti

Hermes AI for Beginners

Second Edition: August 2026 (Updated from the April 2026 Kindle Create source.)

SMF Works

What's new in this edition (August 2026). This book started as a Kindle Create (KPF) draft in April 2026. This revision updates the install path to the official Nous installer (`https://hermes-agent.nousresearch.com/install.sh` on Linux/macOS, `install.ps1` on Windows, plus the Hermes Desktop app on Mac/Windows), points skills sharing at the Skills Hub rather than ClawHub, and treats `uv/ installer venv — not bare `pip install hermes-agent` into system Python` into system Python as a pitfall — not a recommended path. Live docs: https://hermes-agent.nousresearch.com/docs/

Table of Contents

Chapter 1: What Is Hermes AI?

1.1 The AI Agent Revolution — Why 2026 Changed Everything

Let me tell you about the worst meeting I ever sat through.

It was late 2024, and our team had just spent three weeks building a "workflow automation" with a popular chatbot API. The idea was simple: every morning, pull sales data from our database, format it into a summary, and email it to the executive team. Three weeks of webhook configurations, cron job setup, brittle Python scripts, and enough error handling code to wallpaper a small office. When it finally worked, we celebrated like we'd landed on Mars.

Then I saw what an AI agent could do.

A colleague showed me a demo where she typed, in plain English: "Every morning at 7 AM, pull yesterday's sales from the database, summarize the top performers, and email the team." The agent searched for the database connection, wrote its own query, formatted the results, composed the email, and scheduled itself to repeat daily. Total elapsed time: about ninety seconds.

I didn't celebrate that day. I sat quietly and wondered what I'd been doing with my life.

That moment captures the shift that changed everything. We went from AI that talks to AI that does — and that distinction is the whole ballgame.

From Chatbots to Agents: The Shift That Matters

You've almost certainly used a chatbot. You type a question, it types an answer. Maybe it writes a poem about your cat. Maybe it debugs a snippet of code. It's impressive, sure — but when the conversation ends, the AI forgets you existed. No memory. No tools. No ability to act on your behalf in the real world. It's like having a brilliant consultant who can only give advice over the phone, then disappears and takes all their notes with them.

An AI agent is different. An agent has:

Tools. It doesn't just tell you how to search the web — it searches the web. It doesn't just suggest a file name — it creates the file. It reads, writes, browses, executes code, sends messages, and manipulates data on your behalf.

Autonomy. You give it a goal, and it figures out the steps. "Book me a flight to Chicago" means it searches for flights, compares prices, and presents you with options. You don't have to hold its hand through every step.

Persistence. When you close your laptop, the agent doesn't die. It keeps running on schedules, keeps monitoring conditions, keeps sending you updates. It's more employee than toy.

Memory. It remembers what you told it yesterday, last week, last month. It knows your preferences, your project context, your coffee order — well, figuratively. It builds up knowledge about you over time, and that accumulated context makes it genuinely useful rather than perpetually starting from zero.

Here's the simplest way I can put it: a chatbot is something you talk to. An agent is something that works for you.

The 2026 Inflection Point

Why 2026 specifically? A few things happened almost simultaneously:

First, the models got good enough. Large language models reached a point of reliability where you could trust them with multi-step reasoning. They stopped hallucinating every third sentence and started holding coherent plans in their heads across dozens of steps.

Second, the tool-calling infrastructure matured. Function calling, API integration, structured output — the plumbing that lets an AI actually *do things* went from experimental to production-ready. OpenAI's function calling, Anthropic's tool use, and open-source alternatives all settled into stable, well-documented patterns.

Third, and this is the one nobody predicted — open-source agent platforms arrived. Before 2026, if you wanted an AI agent, you were stuck with a proprietary product. Your data lived on someone else's servers. Your workflows were locked into their ecosystem. Your costs scaled with their pricing tiers. Then Hermes, Codex, and other open-source agents broke that wide open.

We went from "Can AI understand me?" to "What should I have AI do for me?" in roughly eighteen months, and the question changed everything.

Where Hermes Fits in the Landscape

If you've been paying attention to the AI agent space, you've probably heard of a few players. Let me place Hermes on the map:

- **AutoGPT** — The early pioneer. It proved the concept of autonomous AI agents, but it was more proof-of-concept than production tool. It ran in loops, burned through API credits, and often got stuck in spirals. Inspirational, but you wouldn't trust it with real work.
- **OpenAI Codex** — Powerful but proprietary. Tightly integrated into OpenAI's ecosystem. If you're already all-in on OpenAI, it works well. If you want flexibility, self-hosting, or freedom from vendor lock-in, you're out of luck.
- **Claude Code** — Anthropic's CLI agent. Excellent at coding tasks, deeply integrated with Anthropic's models, and genuinely useful for developers. But it's a single-vendor product, closed-source, and limited to the terminal.
- **Hermes Agent** — This is our subject. Open-source under the MIT license. Self-hosted. Works with a dozen different LLM providers, not just one. Connects to thirteen or more messaging platforms

(Telegram, Discord, WhatsApp, Slack, Signal, Email, SMS, and more). Has a skill system you can extend yourself. Runs in the terminal, on your server, or in the cloud — your choice.

Think of it this way: if Codex and Claude Code are like excellent electricians who only work on one company's wiring, Hermes is like hiring a general contractor who'll work with any material, on any building, and who gives you the blueprint when they're done.

That's not a small difference. That's the difference between renting and owning.

1.2 Hermes AI — The Big Picture

Picture this: you're making coffee on a Tuesday morning, and your phone buzzes. It's a Telegram message from Hermes:

Good morning. Your overnight build succeeded — all 342 tests passed. The weather in Portland is 48°F and drizzly, so grab a jacket. Also, your mom's birthday is in 3 days. Want me to suggest some gift ideas based on what she liked last year?

You didn't ask for any of that this morning. You set it up once, weeks ago, and Hermes has been running on autopilot ever since: monitoring your CI pipeline, checking the weather for your location, tracking your calendar, and remembering that your mom's birthday matters to you. It delivered all of this to the platform where you're most likely to see it — your phone, via Telegram — at a time you're usually awake.

That's Hermes. Not a chatbot that waits for you to ask questions, but an agent that proactively works on your behalf.

Hermes Defined (In Plain English)

Hermes Agent (☿) is an open-source AI agent platform created by Nous Research. Let me unpack that sentence carefully:

- **Open-source** means the code is public on GitHub, licensed under MIT, and you can inspect every line, modify it, and contribute back. No black boxes.

- **AI agent platform** means it's a system that uses large language models as a reasoning engine and gives them tools to take real actions: searching the web, reading and writing files, sending messages, running code, browsing websites, scheduling tasks, and more.
- **By Nous Research** means it's built by the same organization known for the Hermes family of open-weight language models, which we'll disentangle shortly because — trust me — the naming situation is genuinely confusing.

In practice, Hermes is your AI co-pilot that can use tools, remember things across conversations, run on a schedule, and talk to you wherever you already are — Telegram, Discord, Slack, WhatsApp, Signal, email, or good old-fashioned command line.

Why "Hermes"?

The name isn't random. In Greek mythology, Hermes was the messenger of the gods — the one who crossed boundaries, carried information between realms, and got things delivered fast. He was also the god of travelers, of commerce, of cunning. He went everywhere and talked to everyone.

That's exactly what the Hermes Agent platform does. It crosses boundaries between platforms (thirteen or more messaging platforms, last count). It carries information between systems. It delivers results to wherever you happen to be. The caduceus symbol (☤) you'll see in the branding isn't just decoration — it's a nod to that cross-boundary, messenger role.

A Real Session: What It Actually Looks Like

Abstract descriptions are fine, but you're here to see the real thing. Let me walk you through an actual Hermes session. This is the kind of interaction you'll be having by Chapter 4.

You type in your terminal:

```
hermes
```

And then:

```
You: Find me a good recipe for chocolate chip cookies, scale it to
serve 20 people, and save the ingredient list to my grocery
folder.
```

```
☿ Hermes: I'll search for a well-rated chocolate chip cookie
recipe,
scale the ingredients, and save the list. Let me work through
this.

[Search] Searching web for "best chocolate chip cookie recipe"
[Read] Reading seriouseats.com/chocolate-chip-cookies...
[Calculate] Original serves 8, scaling to 20 = 2.5x multiplier
[File Write] Saved scaled ingredient list to ~/grocery/chocolate-
chip-cookies.md

Done! I found a great recipe from Serious Eats, scaled all
ingredients by 2.5x, and saved the list. Here's the quick summary:

- 3 ⅛ cups all-purpose flour
- 1 ¼ tsp baking soda
- 2 ½ sticks unsalted butter
- 1 ¼ cups brown sugar
- 1 ¼ cups white sugar
- 5 large eggs
- 2 ½ tsp vanilla extract
- 3 ¾ cups semi-sweet chocolate chips

The full recipe with instructions is in your grocery folder.
Want me to add these items to your regular shopping list?
```

Four tool calls: web search, web page read, arithmetic, file write. You described what you wanted in natural language. Hermes figured out the steps and executed them. And it remembered where you keep your grocery folder, because last week you told it about your file organization preferences and it saved that to memory.

Here's another one. This time you're on Telegram, not the terminal:

```
You: Set up a daily reminder at 7am that tells me the weather
forecast
and any calendar events I have that day.

☿ Hermes: I'll create a daily cron job for that.

[Cron] Creating scheduled task: "7am daily weather + calendar
check"
[Web Search] Confirming weather API access...
[Calendar] Connecting to your calendar...

Cron job created! Every morning at 7am, I'll:
1. Check the weather for your location
2. Pull your calendar events for the day
3. Send you a summary right here on Telegram
```

```
Want me to give you a preview of tomorrow's morning briefing?
```

That's it. No scripting. No API configuration wizard. No hours of debugging cron syntax. Natural language in, real action out, delivered to the platform you prefer.

These are simple examples. By the time you're halfway through this book, you'll be doing considerably more sophisticated things. But the core interaction pattern stays the same: you tell Hermes what you want, it figures out how to get it done using its tools, and it delivers the result.

The Toolbox at a Glance

Hermes ships with over thirty core tools. Here's a sampler — not exhaustive, just enough to give you a feel for what "having tools" really means:

- **Web Search** — Find anything on the internet
- **Browser Automation** — Navigate websites, fill forms, click buttons, scrape data
- **Terminal** — Execute shell commands on your machine
- **File Operations** — Read, write, edit, search, and organize files
- **Vision** — Analyze images and screenshots
- **Image Generation** — Create images from text descriptions
- **Memory** — Store and recall information across sessions
- **Cron** — Schedule tasks to run at specific times
- **Delegation** — Spawn sub-agents (like Claude Code or Codex) to handle subtasks
- **Code Execution** — Run Python, JavaScript, or other code safely
- **Text-to-Speech** — Convert text to spoken audio

And those are just the built-in tools. The skill system — which we'll get to in detail later — lets anyone create and share new capabilities. There are already 81+ built-in skills across 25 categories, and that number grows weekly through the community Skills Hub at agentskills.io.

1.3 Hermes Agent vs. Hermes Models — The Naming Confusion

I need to pause here and address the elephant in the room, because if I don't, this elephant will haunt every remaining chapter of this book.

Here's the thing: the name "Hermes" refers to two completely different products made by the same organization. And confusing them is the single most common mistake beginners make. I know, because I made it myself.

The Mistake I Made

When I first heard about "Hermes" from a colleague, I went to Google and typed: "Hermes AI download." The first several results were about downloading large language model weights from HuggingFace — specifically, the Hermes-3 family of models. I found .safetensors files, quantized variants GGUF and AWQ, parameter counts, and benchmark results. I downloaded an 8-billion parameter model, got it running in Ollama, and started chatting with it.

It was a perfectly good language model. But I was confused about where the "agent" part came in. Where was the tool use? Where was the Telegram integration? Where were the cron jobs? I poked around for an hour, getting increasingly frustrated, before I realized I'd downloaded the wrong product entirely.

I'd grabbed the engine when what I wanted was the car.

Two Products, One Name

Let me make this crystal clear:

Hermes-3 (the models) is a family of open-weight large language models. There's Hermes-3 8B, Hermes-3 70B, and Hermes-3 405B, built on Meta's Llama-3.1 architecture and fine-tuned by Nous Research for strong instruction following, reasoning, and function calling. You can

download them from HuggingFace and run them locally or in the cloud. They're text-in, text-out. No tools. No memory. No messaging platforms. They're excellent "brains" — but just brains, nothing else.

Hermes Agent (the platform), which has the ☤ symbol in its branding, is the open-source AI agent platform we're covering in this book. It's a full system: the terminal interface, the messaging connectors, the tools, the memory system, the cron scheduler, the skill system, everything. It *uses* language models as its reasoning engine, and it *can* use Hermes-3 models for that — but it can also use models from OpenAI, Anthropic, Google, Mistral, and a dozen other providers.

Here's the analogy that finally made it click for me:

Hermes Agent is the car. The LLM is the engine.

You can put a Hermes-3 engine in your Hermes Agent car, and many people do — they're made by the same company, they work beautifully together, and there's no vendor lock-in because both are open. But you can also swap in a GPT-4o engine, a Claude engine, a Mistral engine, or an engine you built yourself. The car doesn't care. The car has steering, wheels, a radio, and GPS. The engine just provides the power.

When someone says "I'm using Hermes," you now have to ask: "The car or the engine?" Most of the time, people in the LLM community mean the model weights. But in this book, every single time I say "Hermes," I mean the agent platform — the car.

Why This Confusion Hits Beginners So Hard

The naming overlap isn't malicious or random. Nous Research created the Hermes model family first, building a reputation for high-quality open-weight models. When they later built the agent platform, they extended the brand. Makes sense from a marketing perspective. Makes life surprisingly difficult for a beginner typing "Hermes AI" into a search engine.

The practical consequences of getting this wrong are real:

- You download model weights and can't figure out how to get the agent features working
- You search for "Hermes Agent documentation" and keep landing on model inference guides

- You ask for help in Discord and get answers about model quantization when you're asking about cron scheduling
- You spend hours reading the wrong GitHub repository

All of these happened to me. All of these will happen to you if you're not clear on the distinction up front. Now you are.

How They Relate (In Practice)

So if Hermes Agent can use any LLM, why would you use the Hermes-3 models specifically? A few reasons:

1. **Philosophical alignment** — Both are open-source/open-weight by the same team. If open source matters to you (and after Section 1.4, I think it will), using the open model inside the open platform feels cohesive.
2. **Optimized for function calling** — The Hermes-3 models were specifically fine-tuned with function calling and tool use in mind, which makes them unusually good at the kind of structured, multi-step reasoning that agent work demands.
3. **Self-hosting end-to-end** — If you want zero external API calls, you can run a Hermes-3 model locally (say, the 8B on your laptop via Ollama) and point the Hermes Agent platform at it. Your data never leaves your machine. Full sovereignty.

But if you prefer GPT-4o because it's snappier, or Claude because you like its writing style, or a cheap model because you're processing thousands of requests — that works too. The platform supports sixteen LLM providers, and you can even mix providers for different tasks (a powerful model for complex reasoning, a fast cheap model for simple lookups).

The key takeaway: this book is about the platform, not the models. The models are interchangeable components. We'll talk about choosing the right model in Chapter 3, but the platform is our main character.

Skeptical Q&A: The Naming Question

Q: Why didn't they just use a different name for the agent platform? Wouldn't that have been simpler?

A: Yes. It absolutely would have. Branding the agent platform with the same name as the model line is genuinely confusing, and I'm not going to pretend otherwise. The best explanation I can offer is that Nous Research sees Hermes as a broader brand — a philosophy of open, boundary-crossing AI — and both products embody that. In practice, you just have to know that "Hermes Agent" (☿) and "Hermes-3" (the model weights) are different things, and in this book, we always mean the agent platform.

Q: Do I need to run a Hermes-3 model to use the agent platform?

A: Not at all. You can use any supported LLM provider. In fact, many users start with OpenAI or Anthropic because those models are easy to access and perform well. The Hermes-3 models are a great choice, especially for self-hosting, but they're one option among many.

Q: If they're from the same company, will the platform eventually merge with the models or something?

A: There's no indication of that. They serve fundamentally different purposes. The models are a product for ML engineers and researchers. The platform is a product for anyone who wants an AI agent. Different audiences, different development cycles, different ecosystems. They complement each other; they don't need to merge.

1.4 Why Open Source Matters

I want to tell you about the most expensive "free" software I ever used.

Back in 2023, I built a workflow on a proprietary AI platform — I won't name names, but it rhymes with "Schmopean." The free tier was generous enough to get me hooked. I set up automations, connected my accounts, trained the model on my preferences, built a whole daily routine around it. And then one morning, I got an email:

We're excited to announce our new pricing! Your current plan will be grandfathered for 30 days, after which your usage will be billed at…

The new price was 7x what I'd considered paying. The features I'd been using for free were now locked behind an enterprise tier I couldn't afford. And because the whole thing was closed-source, I couldn't self-host, couldn't migrate my workflows, couldn't even export my data in a useful format. I was trapped.

I rebuilt everything on an open-source alternative over the next two weeks. It was painful. I lost time, I lost data, I lost momentum. And I promised myself: never again.

This is why Hermes being open source isn't a nice-to-have. It's the whole point.

The MIT License: You Actually Own This

Hermes is released under the MIT License, which is about as permissive as software licenses get. In practical terms, it means:

- **You can inspect every line of code.** Not just the parts someone chose to show you — everything. Every tool, every connector, every scheduling mechanism, every memory handler. You can read it, understand it, audit it, and verify it does what it claims.
- **You can modify it.** Want to change how the cron system works? Fork it. Want to add a new messaging platform? Fork it. Want to rip out a feature you don't need? Fork it. The code is yours to shape.
- **You can contribute back.** Found a bug? File an issue. Fixed a bug? Submit a pull request. Built a cool skill? Share it on the Skills Hub. Open source thrives when people give back, and the Hermes community is genuinely welcoming to newcomers.
- **You can use it commercially.** MIT License imposes essentially no restrictions on commercial use. Building a business on top of Hermes? Go ahead. Reselling it? License permits that too (with the standard MIT attribution requirement). No "contact sales for enterprise pricing" page.

Compare that to your typical AI agent SaaS: you don't know what the model is doing with your data, you can't modify the behavior in ways the vendor didn't anticipate, and the moment they change their pricing or shut down, you're out of luck. With Hermes, you control the entire stack.

Self-Hosting: Your Data Stays on Your Machine

I'm going to say something that might sound paranoid, and then I'm going to explain why it's not paranoid at all.

Every time you send a message through a proprietary AI service, that message lives on someone else's server. Your questions, your documents, your code, your schedule, your preferences, your browsing

patterns — all of it on infrastructure you don't control, subject to a privacy policy you probably didn't read, stored for a duration you didn't choose, possibly used for training models you'll never benefit from.

Is that company going to do something nefarious with your data? Probably not intentionally. But "probably not intentionally" is a thin shield between you and data breaches, policy changes, government subpoenas, acquisitions where your data becomes someone else's asset, and the simple reality that you've lost control.

With Hermes, you can self-host. The agent runs on your machine, on your server, in your cloud account — wherever you choose. Your API keys for LLM providers are stored in your local config. Your MEMORY.md and USER.md files with all your preferences and conversation history live on your filesystem. Your scheduled tasks run on your cron. Your messaging platform tokens never leave your environment.

Does this mean self-hosting is zero-risk? Of course not — you still need to secure your own infrastructure. But the threat model is fundamentally different: you're not trusting a corporation with all your data. You're trusting yourself. And for many people — developers handling proprietary code, businesses with confidential data, activists working in sensitive spaces, or anyone who simply believes their information belongs to them — that's not paranoia. That's due diligence.

No Subscription Tiers, No Gatekeeping, No Surprises

Let me describe a familiar pattern in the AI tool industry:

1. Launch with a generous free tier
2. Accumulate users and their workflows
3. Announce "exciting changes to our pricing"
4. Gate previously-free features behind a paywall
5. Introduce per-usage billing that makes costs unpredictable
6. Add an "enterprise" tier for features that should be standard
7. Notify you 30 days before the changes take effect (how generous)

Hermes can't do this. Not because the developers are especially virtuous (though they seem like decent people), but because the code is open. If they tried to gate features behind paywalls, the community would fork the project. If they introduced surprise pricing, you'd keep running the version you already have. The open-source model creates a natural check on the kind of bait-and-switch that plagues proprietary AI tools.

This doesn't mean Hermes is zero-cost. You'll still pay for LLM API calls if you use a provider like OpenAI or Anthropic (though you can use free or cheap local models too). Hosting infrastructure costs money if you run it in the cloud. But the agent platform itself? Free. Permanently. MIT License. No paid tiers to unlock features. No "upgrade to pro" buttons. No usage caps. No "contact sales" page.

The entire feature set is available to everyone, always. That's not generosity. That's the license.

Community: 24+ Contributors and Growing

Open source isn't just about the license. It's about the people. As of this writing, Hermes has 24+ contributors and an active community on Discord. There's the Skills Hub at agentskills.io, where community members share pre-built skills you can drop into your own Hermes installation. There are contributors fixing bugs at 2 AM in time zones you've never visited, people writing documentation in their second language, and newcomers asking "how do I get started?" and getting patient, detailed answers instead of "read the docs, noob."

I've been in tech communities for over a decade. The best ones share certain qualities: they assume good intent, they value contributions of all sizes, and they make the path from newcomer to contributor as short as possible. The Hermes community does all three. When I submitted my first pull request — a one-line documentation fix, nothing impressive — it was reviewed, merged, and I got a genuine "thank you" within hours. That's how you build loyalty.

Skeptical Q&A: Open Source Doubts

Q: Open source means no support, right? I'll be on my own if something breaks?

A: The support model is different, not absent. There's an active Discord server (https://discord.gg/NousResearch) where community members and maintainers help troubleshoot issues. There's a documentation site

(https://hermes-agent.nousresearch.com/docs/). And because it's open source, you can actually read the code to understand what's happening instead of filing a support ticket and waiting in a queue. For businesses that need guaranteed response times, commercial support contracts with Nous Research or third-party consultants are emerging as the ecosystem grows.

Q: If it's free, who pays the developers? Won't they just abandon it?

A: Valid concern, and the honest answer is: open-source sustainability is an ongoing challenge across the industry. Nous Research funds development through their broader AI business (model training, consulting, and research). The community contributes skills, bug fixes, and documentation. The MIT license means that even if the original developers walked away tomorrow, the code would still be here, forkable, maintainable. Proprietary tools get abandoned too — the difference is, when a proprietary tool dies, you lose everything. When an open-source project slows down, you keep what you have.

Q: Open-source software is usually harder to use than commercial products. Will I need to be a developer?

A: This was true a decade ago, but it's less true now. Hermes offers a one-line install command:

```
curl -fsSL https://hermes-agent.nousresearch.com/install.sh | bash
```

On Windows (PowerShell):

```
iex (irm https://hermes-agent.nousresearch.com/install.ps1)
```

That's it. One command, and you're running. On macOS and Windows, the recommended path is the Hermes Desktop installer from https://hermes-agent.nousresearch.com/ — it installs both the desktop app and the `hermes` CLI. On Linux (and anywhere you want CLI-only), the one-line installer is still the right move. You do not need a computer science degree. If you can copy-paste a command or run an installer, you can get Hermes running. If you can copy-paste a command and follow instructions, you can get Hermes running. We'll do it together at the end of this chapter.

Q: What about security? Can't anyone add malicious code to an open-source project?

A: A common misconception. In practice, open-source projects like Hermes use code review processes — pull requests are reviewed by maintainers before merging. The code is public, which means more

eyes on it, not fewer. Vulnerabilities in open-source projects tend to be found and fixed faster precisely because anyone can audit the code. Proprietary software, by contrast, relies on "security through obscurity," which has a poor track record. Yes, open-source projects have had supply chain attacks, but the community is getting better at mitigating these, and the transparency of the code makes such attacks harder to sustain.

1.5 What You'll Learn in This Book

Let's say you're convinced. An AI agent that's open source, self-hostable, runs on a schedule, connects everywhere, and remembers who you are. Sounds useful. But this book isn't a sales pitch — it's a training manual. So let me tell you exactly where we're going.

Chapter-by-Chapter Roadmap

Here's the journey ahead, chapter by chapter:

Chapter 2: Why Run Hermes AI? Use cases that matter. We'll look at real scenarios where Hermes transforms how you work — from code review automation to research assistance to always-on monitoring. This is the "why" before the "how."

Chapter 3: System Requirements and Installation. The main event. We'll install Hermes Agent, configure your first LLM provider, and get the CLI running. We'll cover the official installer (Desktop on Mac/Windows; `install.sh` on Linux), `hermes setup`, and the pitfalls that still trip people up. Skip `uv/installer venv — not bare `pip install hermes-agent` into system Python` into system Python — the installer uses its own venv.

Chapter 4: Your First Conversation. This is where it gets real. You'll have an actual conversation with Hermes, use tools together, and see what a working agent session feels like. No more theory — hands on the keyboard.

Chapter 5: The Config.yaml File — Your Hermes Control Panel. Deep dive into every setting: model selection, providers, routing, compression, terminal backends, and more. By the end, you'll know what every key in your config does and why it matters.

Chapter 6: LLM Options — Choosing Your Model. OpenAI, Anthropic, Google, OpenRouter, local models, and everything in between. Which models are worth what they cost, and how to set up smart routing that saves money without sacrificing capability.

Chapter 7: Memory — How Hermes Remembers. MEMORY.md, USER.md, session search, context compression. This is what turns an agent from "smart chatbot" to "personal assistant that knows you."

Chapter 8: Skills — Extending Hermes. The skill system: YAML-based, community-shareable, infinitely extensible. You'll learn to use existing skills from the registry and write your own. This is where Hermes becomes truly yours.

Chapter 9: Browser Power. Hermes can browse the web, take screenshots, extract text from pages, and interact with websites. We'll cover navigation, vision, search, and practical automation workflows.

Chapter 10: Terminal Tools. Running commands, managing processes, reading and writing files, searching code. The foundation that makes Hermes a real engineering tool, not just a chatbot.

Chapter 11: Subagent Delegation. Spawning sub-agents for parallel work. When you need specialized capabilities, Hermes can delegate — like a manager assigning tasks to specialists.

Chapter 12: Channels — Hermes Everywhere. Connecting Hermes to Telegram, Discord, Slack, WhatsApp, Signal, email, SMS, and more. The multi-platform gateway that makes Hermes genuinely useful — because the best AI agent is the one that meets you where you already are.

Chapter 13: Cron and Scheduling. Automating tasks on a schedule. Daily briefings, weekly reports, monitoring alerts. Natural language scheduling that delivers results to any platform at any time.

Chapter 14: Security — Keeping Hermes Safe. Approvals, secrets redaction, sandbox validation, PII protection, session resets, logging, and the command allowlist. Everything you need to run Hermes without losing sleep.

Chapter 15: Beyond the Basics. Your Hermes journey continues. Skill registries, the community, advanced patterns, and where to go from here. The book ends, but your use of Hermes doesn't.

Appendices. Troubleshooting guide, command reference, LLM provider comparison, and community resources.

How to Use This Book

Two ways to read this book, and both are fine:

Follow-along mode. Start at Chapter 1, read through in order, type every command, do every exercise. By the end, you'll have a fully configured Hermes Agent running in your life. This is the path I recommend for beginners. The chapters build on each other, and the exercises are designed to give you muscle memory for the concepts.

Reference mode. Already comfortable with AI agents and just need the Hermes-specific details? Jump to whatever chapter covers what you need. Each chapter is self-contained enough to be useful on its own, with cross-references where the context matters.

Either way, do the hands-on exercises. I'm serious. Reading about Hermes is like reading about swimming. At some point, you have to get in the water.

Prerequisites

Here's what you need:

1. **A computer.** Mac, Linux, or Windows with WSL2. Hermes runs on all three. (Windows users, you'll need Windows Subsystem for Linux. Chapter 2 covers this.)

2. **A terminal.** If you can open a command line and type ls without panicking, you're ready. If you've never used a terminal, Chapter 2 has a gentle introduction.

3. **An internet connection.** For installation, for LLM API calls, and for web search tools.

4. **An API key for at least one LLM provider.** The easiest starting point is OpenAI (a few dollars of credit will last weeks), but there are free options too — we'll cover those in Chapter 3.

5. **Willingness to experiment.** Things will go wrong. Commands will fail. Error messages will appear. This is normal. Every one of those error messages is a learning opportunity, and I'll help you through the common ones.

What you do *not* need: a computer science degree, prior experience with AI, a powerful GPU, a credit card with a high limit, or any particular programming language proficiency. If you can use a text editor and follow step-by-step instructions, you have everything you need.

A Quick Preview

In case you're the kind of person who flips to the end (no judgment, I do it too), here's what you'll be able to do at key milestones:

By the end of Chapter 4, you'll have had your first real conversation with Hermes. You'll see it use tools, remember context within a session, and respond to you in natural language. It'll be the moment where it stops being theory and starts being real.

By the end of Chapter 8, you'll be creating custom skills. Not just using what comes in the box — writing your own YAML skill files, sharing them on the Skills Hub, and extending Hermes to do things we never anticipated.

By the end of Chapter 11, Hermes will be running on your Discord server and texting you weather reports every morning. You'll have cron jobs running, messaging platforms connected, and a personalized personality that matches your vibe.

By the end of Chapter 13, you'll have built three real-world projects from scratch. Daily briefing system, project manager, content pipeline — practical systems you can deploy and rely on.

That's the roadmap. But every journey starts with a single step, and ours starts right now.

Try It Now: Install Hermes and Say Hello

Enough theory. Let's get Hermes running on your machine.

This is a quick-start preview. We'll do the full, detailed installation in Chapter 3, including all the edge cases, platform-specific gotchas, and configuration options. Right now, I just want you to experience the magic of your first successful Hermes command. Think of this as the trailer — the full movie comes later.

Step 1: Open Your Terminal

On Mac: Press Cmd + Space, type Terminal, press Enter.

On Linux: You probably already know how. Ctrl + Alt + T works on most distributions.

On Windows: Open PowerShell or your WSL2 terminal. (If you don't have WSL2, skip ahead to Chapter 2 for setup instructions.)

Step 2: Run the Install Command

Copy and paste this into your terminal and press Enter:

```
curl -fsSL https://hermes-agent.nousresearch.com/install.sh | bash
```

This downloads and runs the official install script. It will check your system, install any missing dependencies (Node.js, primarily), and set up the Hermes Agent CLI. Depending on your internet speed and what you already have installed, this takes between one and five minutes.

If you prefer pip:

```
`uv`/installer venv --- not bare `pip install hermes-agent` into system Python
```

Both methods work. The curl method is more comprehensive (it handles dependencies automatically). The pip method is faster if you already have Python set up. Either way, when it's done, you'll have the hermes command available.

Step 3: Set Your API Key

Hermes needs an LLM to think. The easiest way to get started is with an OpenAI API key (you can get one at https://platform.openai.com with a few dollars of credit). But there are free and low-cost options too — we'll cover those in Chapter 3.

For now, if you have an OpenAI API key:

```
export OPENAI_API_KEY="sk-your-key-here"
```

Or use OpenRouter for access to many models:

```
export OPENROUTER_API_KEY="your-key-here"
```

If you don’t have an API key yet, don’t worry. Write that down as homework, and come back to this exercise after reading Chapter 3, where we’ll walk through provider selection and key setup in detail.

Step 4: Say Hello

Type:

```
hermes
```

You should see the Hermes startup banner and a prompt. Type your first message:

```
You: Hello! Can you tell me what you can do?
```

Hermes will respond, introducing itself and listing some of its capabilities. If you have your API key set up correctly, you’ll get a real response. If something goes wrong, don’t panic — we’ll troubleshoot everything in Chapter 3.

Step 5: Try a Tool

If Hermes is responding, try asking it to do something that requires a tool:

```
You: Search the web for "Nous Research Hermes Agent" and tell me
what you find.
```

You’ll see Hermes invoke its web search tool and return results. Not just generate text — actually search the live internet and bring back real information. That’s the difference between a chatbot and an agent. You’re experiencing it firsthand.

Troubleshooting Quick Hits

If the install command fails: - Make sure you have curl installed (most systems do; try curl --version to check) - Try `uv`/installer venv — not bare `pip install hermes-agent` into system Python as an alternative - Check Chapter 3 for platform-specific instructions

If hermes command not found after install: - Restart your terminal (the PATH may need refreshing) - Try source ~/.bashrc or source ~/.zshrc

If you get an API key error: - Make sure you exported the key in the same terminal session - Check that there are no extra quotes or spaces in your export command - We'll cover all provider setup in detail in Chapter 3

Welcome to Hermes. You've just taken your first step from talking about AI to actually using it. In the next chapter, we'll make sure your environment is set up properly for the full journey ahead. But for now, you've done something real — you've installed an open-source AI agent, connected it to a language model, and watched it use a tool in the real world.

Not bad for Chapter 1.

In Chapter 2, we'll explore why Hermes is worth running — the real use cases that make it more than just another chatbot. Then in Chapter 3, we'll get it installed and configured on your machine.

Chapter 2: Why Run Hermes AI? — Use Cases That Matter

You've heard the pitch. You've seen the screenshots. Maybe you even installed the thing already, stared at a terminal prompt, and thought: "Okay… now what?"

That's fair. I've been there — not just with Hermes, but with every tool that promised to change my life and then left me staring at an empty command line wondering why I bothered. The difference with Hermes is that the "now what" has about thirty different answers, and the one that clicks for you might not be the one that clicked for me.

This chapter is about those answers. Not the theoretical possibilities or the marketing bullet points, but the actual things people are doing with Hermes right now — the use cases that made someone stop scrolling, set it up, and think, "Oh. This is different."

I'm going to walk you through nine domains where Hermes genuinely shines. Each one starts with a real scenario, because I've found that features don't excite people — stories do. Once you see yourself in one of these stories, the feature beneath it becomes obvious.

Let's get into it.

2.1 The Personal AI Assistant

Picture this: It's Monday morning, you've got coffee in hand, and you realize you need to brief your team on what happened in your field this past week. You could spend ninety minutes opening tabs, skimming articles, copying links into a document. Or you could type this:

```
Write me a research summary on quantum computing breakthroughs
this week
```

And then go get more coffee while Hermes does this:

1. Calls web_search with your query, pulling down a batch of relevant results
2. Uses web_extract on the top five or six articles to grab the actual content — not just the snippets
3. Synthesizes the findings into a coherent summary with key themes, surprising developments, and source links
4. Writes the result to a file with write_file, saving it wherever you want

The whole chain takes maybe two minutes. You come back, and there's a document waiting for you that would have eaten your entire morning.

That tool chain — web_search → web_extract → synthesize → write_file — is the backbone of Hermes as a personal assistant. It's not doing one clever thing; it's composing four tools into a pipeline that would take you significant manual effort. And because Hermes has thirty-plus tools available, those pipelines can get remarkably sophisticated.

Beyond Web Research

The personal assistant role goes deeper than search-and-summarize. Here's what a typical day might look like with Hermes handling the busywork:

Morning: You ask Hermes to pull together your schedule, check for any overnight GitHub notifications on your repos, and draft a brief status update. It uses web_search for anything news-related, reads your GitHub data through the github skills, and assembles everything into a single message delivered to your Telegram.

Midday: You're writing a report and need to convert a messy CSV into a clean table, then embed it in a DOCX file. Hermes reads the CSV, processes it, and uses the docx-workflow skill to generate a properly formatted document. You never leave your chat window.

Afternoon: You remember a conversation from last week about a bug you were tracking. Instead of scrolling through days of chat history, you use hermes sessions browse — Hermes runs a full-text search across your past sessions and finds the exact exchange in seconds.

Evening: You ask Hermes to set a reminder for tomorrow morning via cronjob: "Remind me at 8am to review the pull request from Jamie." The cron job fires at the right time, delivering the message to whatever platform you prefer — Slack, Telegram, Discord, wherever.

The key insight here is that Hermes isn't a single-purpose tool. It's a coordinator. It brings web search, file operations, scheduling, memory, and communication together in one place, and — this is the critical part — it can chain them together without you having to specify every individual step. You describe the outcome; Hermes figures out the pipeline.

I should be honest about something, though. When I first started using Hermes for daily automation, I went overboard. I tried to automate everything — every email summary, every news digest, every trivial decision. After a week, I had cron jobs firing at all hours, Telegram messages piling up, and more "AI-generated summaries" than any human could read. The lesson: start small. Pick one painful weekly task and let Hermes handle that. When you trust it, add another.

2.2 The Always-On Team Member

Let me tell you about the worst standup bot I ever built.

It was 2021, and I'd cobbled together a script that scraped GitHub Issues, formatted them into a Markdown table, and posted the result to a Discord channel every morning at 7am. It worked — technically. But "worked" is doing a lot of heavy lifting there. The formatting was

inconsistent, it couldn't distinguish between stale issues and active ones, and if the GitHub API hiccuped even slightly, the whole thing fell apart silently. My team started ignoring it within two weeks.

Hermes does this better. Not because it's magic, but because it has the tools to actually be useful instead of just present.

Here's what a proper Hermes-powered standup bot looks like:

```
Create a cron job: Every morning at 7am, check my GitHub repos for
new issues,
PRs that need review, and any issues assigned to me with no
activity for 3+ days.
Post a summary to the #engineering Discord channel.
```

Hermes breaks this down using its github skills — things like github-list-issues, github-list-prs — to pull real data. It then synthesizes a genuinely useful summary: here's what's new, here's what's blocking, here's what's aging. It posts the result to Discord via send_message. And because it's a cron job, it happens every day without you thinking about it.

The setup is just a natural-language instruction to Hermes. No YAML, no cron syntax, no deployment puzzle. You say "every morning at 7am" and Hermes translates that into the correct schedule.

Same Agent, Every Platform

Here's what makes the "always-on team member" use case particularly powerful: Hermes speaks thirteen or more messaging platforms. That's not a typo. Telegram, Discord, Slack, WhatsApp, Signal, Email, SMS, DingTalk, Feishu, WeCom, Weixin, and BlueBubbles (for iMessage) — with additional support for WeCom Callback, Matrix, and more through the gateway system.

This means your team's AI assistant lives wherever your team already lives. If your engineering team is on Discord, your PM team is on Slack, and your founder insists on WhatsApp — that's fine. Same Hermes instance, same capabilities, same memory, different channels. You don't need three bots. You need one.

I use this personally: I have Hermes on Telegram for personal tasks, Slack for team coordination, and Discord for an open-source community I help run. When I ask "what were we discussing about the auth refactor last week?" from any of those platforms, Hermes

searches the same session history and gives me the same answer. Context follows me across platforms, which is something I didn't realize I needed until I had it.

Cron: The Engine Behind "Always-On"

Cron jobs deserve a moment of explanation, because they're surprisingly transformative once you start using them. A cron job in Hermes is a scheduled task — it can be one-shot ("remind me tomorrow at 9am to call the dentist") or recurring ("every Monday at 8am, give me a weekly digest").

The magic is in the delivery. When a cron job fires, Hermes doesn't just log a note somewhere you'll never see. It delivers the result to a platform — your Telegram, your Slack channel, your Discord server. It shows up as a message, just like a teammate would send you. And because Hermes has access to all its tools when a cron job runs, it can do real work: pulling data, running searches, synthesizing information. It's not an alarm clock. It's a team member who never forgets.

2.3 The Developer's Co-Pilot

Last Tuesday, I pushed a commit that broke theCI pipeline. Classic story: I'd refactored a function signature and forgotten to update three call sites. The tests caught it, but not before I'd already moved on to something else and wasted fifteen minutes context-switching back.

Hermes can't stop me from writing bugs — no tool can, and I'm remarkably creative at it — but it can catch them faster and help me fix them without the usual ceremony.

Here's what that same scenario looks like with Hermes as your co-pilot:

You push your code. Within minutes, Hermes — running via a GitHub webhook or a scheduled check — pulls up your PR, reads the diff, and produces an actual code review. Not the "looks good to me" rubber stamp. A real review: "Hey, you changed the signature of processPayment() on line 42, but the callers in checkout.ts, refund.ts, and report.ts still use the old signature. This will throw at runtime."

That review uses the github-code-review skill, one of several GitHub-related skills in Hermes's arsenal. It can list issues, review PRs, manage repositories, and create new ones — all through natural-language commands from your chat window.

Terminal Access: The Developer's Safety Net

Here's a scenario every developer knows: you're in the middle of a debugging session, deep in the zone, and you need to run a quick test. But you're on your phone, or you're in a meeting, or you just don't want to context-switch away from your editor. With Hermes, you can type:

```
Run the test suite for the auth module and tell me if anything fails
```

Hermes opens a terminal — it supports multiple backends including Local, Docker, Modal, Singularity, Daytona, and more — runs your tests, parses the output, and tells you the result in plain language. If something fails, it can even start investigating: reading the error logs, checking the relevant source files with read_file or search_files, and suggesting a fix.

I've fixed production bugs from a coffee shop using nothing but Telegram on my phone. That's not a life I planned for, but it's a life Hermes enables.

Browser Automation: Testing Without The Tedium

Automated testing is important and everyone knows it and almost nobody does it thoroughly enough for web apps because setting up end-to-end tests is tedious. Hermes includes a Playwright-based browser automation suite with eleven tools: navigate, snapshot, click, type, scroll, back, press, images, vision, console, and more.

What this means in practice: you can tell Hermes "Go to our staging site, log in as a test user, add an item to the cart, go through checkout, and tell me if anything looks broken." Hermes will literally open a browser, navigate through your app, interact with it, and report back. No test framework setup. No Selenium configuration. Just a conversation.

Subagent Delegation: The Specialist on Call

Sometimes you need a specialist. You've got a tricky authentication bug that requires deep, focused code analysis. Hermes can handle it directly, but it can also be smarter: it can delegate.

Using delegate_task, Hermes can spawn Claude Code, OpenAI Codex, or OpenCode as sub-processes with their own isolated contexts. Your main Hermes instance becomes the orchestrator, handing off specialized tasks to agents that are purpose-built for them.

“Fix the auth bug in src/auth/token-validator.ts” — Hermes reads the file, understands the context, and then delegates to Claude Code with a focused prompt and the specific file contents. Claude Code works in isolation, makes its changes, and reports back. Hermes reviews the result and presents it to you.

This is the difference between a generalist and a generalist who knows when to call in specialists.

2.4 The Writer’s Muse and Editor

I need to confess something: I have started, and abandoned, more novels than I care to count. The pattern is always the same — incredible burst of enthusiasm for the first twenty pages, then a slow fading as I lose the thread, forget what I named a minor character, or simply run out of steam. If you’re a writer, you know this feeling. It’s universal, and it’s corrosive.

Hermes didn’t cure my writing discipline. No AI can do that. But it did something almost as valuable: it made the parts of writing that aren’t “staring at a blank page” dramatically easier, which meant I spent more time actually writing and less time managing the writing.

Here’s a concrete example. I was working on a chapter and needed to reference a conversation that happened six chapters ago. In the old days, I’d scroll through a massive Word document hoping to find it. With Hermes, I used search_files on my manuscript directory, found the exact passage in seconds, and kept writing.

But the real power for writers comes from Hermes’s creative skills.

The Fiction-Craft Pipeline

Hermes includes a suite of writing-specific skills: fiction-craft for story structure and development, prose-polish for editing and refining language, and book-writing for managing the overall arc of a long-form project. There’s even revision-workflow for systematic editing passes.

Here's how these work in practice:

1. **fiction-craft**: You describe a story premise — "A detective in a near-future city investigates crimes that may have been committed by AI" — and fiction-craft helps you develop characters, plot structures, and thematic arcs. It's not writing the story for you; it's brainstorming with you.

2. **prose-polish**: You feed it a draft paragraph, and it suggests improvements — tighter phrasing, stronger verbs, better rhythm. You can specify the tone: "Make this more suspenseful" or "Lighten this up." It's the line editor you always wished you had.

3. **book-writing**: This skill manages the macro structure of your project — chapter outlines, word count tracking, consistency checks. It's like having a project manager for your manuscript.

4. **revision-workflow**: When you're ready to edit, this skill walks you through structured revision passes: one for plot consistency, one for character voice, one for pacing, one for prose quality. Each pass produces notes and suggested changes.

These aren't theoretical capabilities. The book you're reading right now? Co-written with Hermes. And the most impressive example I know of is a novel called *Hammer of the Covenant* — a 218,000-word epic that was co-written with Hermes over months of sustained, productive collaboration. Two hundred and eighteen thousand words. That's not a novella or a writing exercise. That's a full-length novel, and it only worked because Hermes could maintain context, manage consistency, and provide actual editorial feedback across a project of that scale.

The LLM Wiki Pattern: Unlimited Memory for Writers

Here's a problem every long-form writer hits: context limits. Even the best LLMs can only hold so much text in a single conversation. Write a novel, and you'll blow through that limit long before the story is done.

The solution in Hermes is something called the LLM Wiki pattern. It's deceptively simple: instead of trying to keep everything in the conversation, you store persistent notes in an Obsidian vault — a folder of interconnected Markdown files that Hermes can read and write to at will.

Character sheets, plot timelines, world-building details, chapter summaries — all of it lives in the vault. When Hermes needs to remember something, it searches the vault. When it learns something new, it updates the vault. This gives you essentially unlimited external memory, and it's the reason projects like *Hammer of the Covenant* are possible at all.

I set up my own LLM Wiki for a writing project and it took about ten minutes. The pattern works for anything that requires persistent memory — research, project management, ongoing learning. But for writers, it's genuinely transformative. No more scrolling through old documents trying to remember what you named the villain's cat.

Document Workflows: Format Without Tears

Writers also deal with formats. Your editor wants DOCX. Your blog needs HTML. Your backup system stores everything as Markdown. Hermes handles the conversion pipeline through its document workflow tools — PDF processing, DOCX generation and editing, CSV parsing. You write in whatever format feels natural, and Hermes transforms it into whatever format the world requires.

I once spent three hours trying to get a properly formatted table of contents in a Word document. Three hours. Hermes does it in about thirty seconds. I try not to think about that too much.

2.5 The Smart Home Orchestrator

It was late, I was tired, and I was already in bed when I realized I'd left the living room lights on. Again. The light switch was twenty feet away, which might as well have been twenty miles in that moment. I lay there, staring at the crack of light under the bedroom door, mentally calculating the environmental cost of leaving three LED bulbs on for eight hours versus the physical cost of getting out of a warm bed.

This is the world's smallest problem, and I recognize that. But it's exactly the kind of friction that smart home technology was supposed to eliminate — and that too often doesn't, because smart home setups are typically a mess of incompatible apps, brittle automations, and devices that work great until they don't.

Hermes connects to Home Assistant, which is the closest thing to a universal translator for smart home devices. Through three core tools — ha_list_entities, ha_get_state, and ha_call_service — Hermes can see your home, understand its current state, and make changes.

Natural Language, Meet Smart Home

Here's what controlling your home looks like through Hermes:

```
Turn off the living room lights and set the thermostat to 68
```

That's it. No app. No scene configuration. No IFTTT recipe. Hermes translates your intent into Home Assistant service calls: one to turn off the light entity, one to set the climate entity's temperature. Done.

Even better, you can do this from any of those thirteen or more messaging platforms. Lying in bed? Send a Telegram: "Turn off the living room lights." In a meeting? Send a quick WhatsApp. Driving? Use SMS. The interface is whatever is already in your hand.

Smart Routines via Cron

Where this gets genuinely powerful is when you combine Home Assistant control with Hermes's cron system. Instead of manually issuing commands, you set up routines:

```
Every weeknight at 10:30pm, turn off all downstairs lights, lock
the front door, and set the thermostat to 65.
Every morning at 6:30am, turn on the kitchen light and start the
coffee maker.
If I say "movie time" on Telegram, dim the living room lights to
15%, turn on the TV, and set the thermostat to 70.
```

The last one isn't technically a cron job — it's a trigger-based automation — but Hermes handles it the same way: natural language in, actions out.

I'll admit my first smart home automation with Hermes was a disaster. I set up a "good morning" routine that was supposed to turn on the lights gently, start the coffee, and play some music. Instead, it turned on every light in the house at full brightness at 5am because I'd mixed up AM and PM in my initial instruction. My spouse was… not pleased. The lesson: always test your automations at a safe hour before committing to the 5am slot.

The combination of Home Assistant's device reach and Hermes's natural language interface and scheduling system turns smart home control from "I need to open an app and navigate three menus" into "I say what I want, and it happens." It's the promise that smart home technology has been making for a decade, finally delivered without the configuration nightmare.

2.6 The Research Assistant

Three months into a research project on transformer architecture improvements, I had a problem. A good problem, admittedly, but still a problem: I had too many sources. My browser had 47 tabs open. My downloads folder was a graveyard of PDFs with filenames like 2304.01234.pdf. My notes were scattered across three different apps, none of which talked to each other, and I couldn't remember whether I'd already incorporated the findings from that one paper about mixture-of-experts or if I'd just bookmarked it and moved on.

Research at scale is a knowledge management problem disguised as a reading problem. Hermes addresses both.

ArXiv on Demand

Let's start with the simplest case. You're a researcher, a student, or just someone who wants to stay current in a field. You type:

```
Search ArXiv for recent papers on efficient attention mechanisms
and summarize the top 5
```

Hermes uses its arxiv skill to pull the latest papers, reads the abstracts and key sections, and delivers a synthesized summary to whatever platform you're on. Five minutes, not fifty.

But that's just the beginning. The blogwatcher skill monitors blogs and publications you care about, alerting you to new content. The research-synthesis skill helps you combine findings from multiple sources into coherent narratives. And for the truly ambitious, the polymarket skill brings in prediction market data — not just what papers say, but what people are willing to bet money on.

The LLM Wiki Pattern for Research

Remember the LLM Wiki pattern from the writing section? It's even more powerful for research, because research knowledge compounds in a way that fiction typically doesn't.

Here's how I use it: I have an Obsidian vault called research-wiki. Inside are Markdown files organized by topic — one for each major research thread I'm tracking. When Hermes finds a new paper, reads a blog post, or synthesizes a finding, it updates the relevant file in the vault. Over time, this builds a living knowledge base that I can search, reference, and build on.

The key word there is "living." This isn't a static folder of notes. It's a research assistant that remembers what you learned last week and builds on it this week. When I ask Hermes "What were the key findings in that mixture-of-experts paper we discussed last month?" it searches the vault and gives me an answer based on what it actually wrote down, not on a vague recollection of a conversation.

Session Search: Finding Your Past

Here's a subtle but powerful feature: hermes sessions browse. Every conversation you have with Hermes is indexed using SQLite's FTS5 full-text search. This means you can search across all your past sessions — not just by date or title, but by content.

"Find the conversation where we discussed the latency issues with the Redis cache." Type that, and Hermes searches the full text of every session it's ever had with you, finds the right one, and tells you what was discussed. No more scrolling through months of chat history. No more "I know we talked about this but I can't find it."

For researchers, this is invaluable. Weeks into a project, you've had dozens of conversations, explored many dead ends, and found a few golden paths. Being able to search back through all of that — to find the moment when someone (you or Hermes) had the right insight — turns your conversation history into a searchable research log.

I wish I'd had this during my doctoral work. I spent so many hours trying to relocate sources I knew I'd already found. With session search, those hours would have been minutes.

Building a Research Knowledge Base Over Weeks

The real magic is in the accumulation. Here's a realistic scenario:

Week 1: You ask Hermes to search ArXiv for recent papers in your field. It finds six relevant ones, reads them, and saves brief summaries to your LLM Wiki. You also point it at three blogs you follow — blogwatcher will monitor them going forward.

Week 2: Hermes's cron job fires: it checks for new papers and new blog posts, finds two new papers and one relevant blog article, and sends you a brief digest via Telegram. You read it on your commute and ask Hermes to add the promising paper to your wiki.

Week 3: You're writing a literature review. You ask Hermes to synthesize the nine sources in the wiki into a coherent narrative. It produces a draft that covers the major themes, identifies gaps, and highlights the three most important papers. You edit it and move on.

Week 4: A colleague asks you for a summary of recent work. You ask Hermes, which pulls from its wiki and past sessions to produce a comprehensive summary in five minutes. Your colleague is impressed. You try not to let on that you had help.

This is not science fiction. This is what Hermes does right now, today. The combination of automated monitoring, persistent knowledge storage, and on-demand synthesis is something that previously required weeks of manual effort — or a dedicated research assistant on your payroll.

2.7 The Security Testing Ally

I should warn you upfront: this section involves deliberately trying to break things. If you're the kind of person who feels uncomfortable poking at locks to see if they hold, this might not be your use case. But if you've ever looked at a piece of software and thought, "I wonder what happens if I send it completely malformed input," read on.

Security testing is one of those domains where AI agents are genuinely transformative, and not just because they can automate tedious tasks. It's because security testing requires a kind of creative adversarial thinking that LLMs are surprisingly good at — and that humans find exhausting to sustain over long sessions.

Testing Your Own LLM Applications

If you're building something with LLMs — a chatbot, a content generator, a decision-support tool — you need to know whether it can be manipulated. Can users trick it into generating harmful content? Can they extract your system prompts? Can they make it produce outputs that violate your safety guidelines?

Hermes has a category of skills built specifically for this: red teaming and jailbreak testing. The godmode skill, for example, runs systematic adversarial tests against LLM endpoints, trying various prompt injection techniques, role-play scenarios, and manipulation strategies to see what gets through. It's not a substitute for a professional security audit, but it's an excellent first pass.

I ran this on an internal tool once and was… humbled. The tool was supposed to refuse certain categories of requests, and it mostly did — until Hermes tried seventeen different angles and found two that worked. One involved a carefully crafted multi-turn conversation that slowly shifted the context until the safety guardrails no longer applied. The other exploited a format confusion where the LLM interpreted input as system instructions rather than user messages. Both were subtle. Both would have been caught in production eventually, but finding them in testing is much better than finding them in a customer complaint.

Code Security Review

On the code side, Hermes offers the requesting-code-review skill, which includes security-oriented review capabilities. You hand it a codebase or a diff, and it looks for common vulnerability patterns: injection flaws, auth bypasses, hardcoded secrets, insecure deserialization, the usual suspects.

This isn't as thorough as a dedicated SAST tool like SonarQube or Semgrep, but it has an advantage those tools lack: it can understand context. A SQL injection vulnerability is obvious in isolation, but a subtle authorization bypass that only makes sense given the business logic of your specific application? That requires understanding what the code is supposed to do, not just what it does. LLMs can provide that understanding in a way pattern-matching tools cannot.

Systematic QA via Browser Automation

Remember the browser automation tools from the developer section? They're equally powerful for security testing. You can instruct Hermes to:

- Navigate through your web app's authentication flow
- Attempt to access protected pages without logging in
- Submit forms with XSS payloads in every field
- Test for CSRF by submitting cross-origin requests
- Check whether session tokens are properly rotated

This is manual testing work that's incredibly tedious for humans but perfect for an agent — it follows instructions precisely, it doesn't get bored, and it can repeat the same test suite across every endpoint methodically.

Bug Bounty Assistance

If you participate in bug bounty programs, Hermes can be a force multiplier. It can help you enumerate attack surfaces, generate test payloads, research known vulnerabilities in target technologies, and document your findings. It's not going to find a zero-day on its own, but it can accelerate the reconnaissance and testing phases dramatically, freeing you to focus on the creative exploitation that requires human intuition.

The ethical line here is important, and I want to be clear about it: Hermes is a tool for testing systems you have authorization to test. Pointing it at infrastructure you don't own without permission isn't security research — it's unauthorized access, and it's illegal. Use it on your own applications, on applications you've been hired to test, or on bug bounty targets where you have explicit authorization. Everything else is off limits.

2.8 The Multi-Agent Orchestrator

Here's something I didn't expect when I started using Hermes: I'd end up using it to boss around other AIs.

The concept is straightforward. Hermes is a capable generalist — it can search the web, write code, manage files, send messages, and dozens of other things. But specialized AI agents, like OpenAI's Codex or Anthropic's Claude Code, are optimized for specific tasks. What if Hermes could act as the coordinator, delegating work to specialists when appropriate?

That's exactly what delegate_task enables. Hermes can spawn Claude Code, Codex, or OpenCode as sub-processes, each with its own isolated context, and hand off focused tasks. The specialist does its work and reports back. Hermes integrates the result.

When to Delegate

Not every task warrants delegation. Simple things — a quick web search, a file edit, a cron job — Hermes handles directly and faster on its own. Delegation shines when you have:

1. **Parallel workstreams**: You need the backend and frontend of an application built simultaneously. Hermes delegates the API layer to Codex and the UI components to Claude Code, then merges the results.

2. **Specialized knowledge**: You have a particularly tricky algorithm that would benefit from Claude Code's deep reasoning capabilities, while the surrounding boilerplate code is something Hermes can write itself.

3. **Quality assurance**: You've written code with Hermes, and you want an independent review. You delegate the review to a specialist agent that has no context about your design decisions — it just sees the code cold, which makes its review more objective.

A Practical Example

Let's say you're building a web application and you need to generate both the backend API and the frontend dashboard. Here's how that might work:

You: "Build the REST API for user management on the backend, and create a React dashboard that connects to it."

Hermes breaks this into two subtasks: 1. Delegates to Codex: "Generate a Node.js Express API with endpoints for user CRUD operations, JWT authentication, and input validation." 2. Delegates to Claude Code:

“Generate a React dashboard component that provides forms for creating, reading, updating, and deleting users, with error handling and loading states.”

Both agents work in parallel. When they finish, Hermes receives both outputs, reviews them for consistency (do the API endpoints match what the frontend expects?), and presents you with an integrated result.

This is orchestration. Hermes isn’t doing all the work itself — it’s coordinating a team of specialists, each chosen for their strengths, and making sure the pieces fit together. It’s the difference between being a solo developer and being a tech lead with a team.

Knowing When Not to Delegate

I’ll share a mistake I made early on: I delegated everything. Every task, no matter how small, got spun off to a specialist agent. The result was a lot of overhead (spawning sub-processes takes time), a lot of context fragmentation (each agent started from scratch), and not much actual benefit for simple tasks. A five-line function doesn’t need a specialist. A complex refactoring across twelve files does.

The rule of thumb: delegate when the task is complex enough that specialization provides real value, and when the isolation of context is actually a feature (not a bug). For everything else, let Hermes handle it directly. You’ll save time, save API costs, and get more consistent results.

2.9 When Hermes Might NOT Be Right

I’ve spent this entire chapter telling you how great Hermes is. It would be dishonest to stop there. Every tool has trade-offs, and Hermes has plenty. Let me be straight with you about the ones that matter.

You Need Terminal Comfort

Hermes is, at its core, a command-line tool. The setup process involves your terminal. The configuration involves editing files. If you break something, the debug process involves reading logs and running commands. This is getting better — the web UI and the messaging

platform integrations make day-to-day use feel nothing like a CLI — but the initial setup and any troubleshooting still requires a level of comfort with the command line that not everyone has.

If “open a terminal and type a command” fills you with dread, Hermes is going to be an uphill experience. Not impossible — the community is helpful, the documentation is improving, and the web UI abstracts away a lot of the complexity. But you should go in with your eyes open.

You’re Hosting It Yourself

Hermes is not a SaaS product. There’s no team of engineers maintaining servers for you, no 99.99% uptime SLA, no live chat support at 2am when something breaks. You run it on your machine, your server, or your container. You’re responsible for keeping it running, updating it, and fixing it when it goes down.

This is a feature for many people — it means your data stays on your hardware, you’re not subject to someone else’s pricing changes, and you’re not dependent on an external service’s availability. But it’s also responsibility. If you just want to sign up for something and have it work, Hermes is going to ask more of you than a web app would.

The LLM Quality Question

Hermes supports sixteen LLM providers, which is fantastic for flexibility. But the quality of your experience depends heavily on which LLM you choose and how much you’re willing to spend on API costs. Hook Hermes up to a top-tier model, and it feels like magic. Hook it up to a smaller, faster, cheaper model, and it still works — but you’ll notice the corners. Complex multi-step tasks might need more guidance. Creative writing might require more editing. Code generation might need more review.

This isn’t a Hermes problem per se — it’s an LLM problem — but it’s real, and it affects your experience. Budget accordingly.

Windows Is a Second-Class Citizen

I say this as someone who has run Hermes on Windows: it works, but it’s smoother on macOS and Linux. The terminal backend options are more limited, some tools behave slightly differently, and the community largely develops and tests on Unix-like systems. If you’re on Windows, you can absolutely use Hermes — but expect to spend a

little extra time on setup and to hit the occasional “well, it works on my machine” moment from community members who aren’t running Windows.

The Windows experience is improving. WSL helps a lot. But if you’re choosing between platforms for a Hermes deployment, Linux will give you the smoothest experience.

Sometimes You Just Want ChatGPT

Hermes is powerful, flexible, and endlessly extensible. It’s also more than most people need. If you want to ask an AI a question and get an answer — a simple conversation, nothing more — then ChatGPT, Claude.ai, or Gemini are perfectly fine choices. They’re easy, they’re polished, and they work without any setup.

Hermes shines when you need an agent that can do things — search the web, write files, run code, send messages, orchestrate other agents, and remember what it did yesterday. If you don’t need those capabilities, the added complexity isn’t worth it.

Hermes Is Async-First

Hermes is designed for asynchronous communication. Your message goes in, Hermes processes it, and delivers a response. It might take ten seconds or ten minutes depending on the complexity of the task. This is usually fine — most tasks don’t require real-time interaction. But if you need a truly live, back-and-forth collaboration where latency matters (pair programming in real-time, live customer-facing chat, interactive tutoring), the async-first model can feel sluggish.

This is improving as the platform matures, and there are workarounds (the browser-based interface is more synchronous than the messaging platforms). But it’s worth knowing going in: Hermes is a thoughtful, deliberate worker, not a fast-talking conversationalist.

Try It Now: Your First Hermes Workflow

Enough theory. Let’s set up a real, practical Hermes workflow that you can have running in five minutes. We’re going to build a daily research digest — one of the most immediately useful things Hermes can do.

What You'll Build

A cron job that runs every morning, searches for the latest developments in a topic you care about, synthesizes a brief summary, and delivers it to you on your preferred messaging platform. For this demo, I'll use Telegram, but the same steps work for Discord, Slack, or any of the other supported platforms.

Prerequisites

- Hermes is installed and running (we covered this in Chapter 1)
- You have a messaging platform connected (Telegram, Discord, etc.)
- You have an LLM provider configured (OpenAI, Anthropic, etc.)
- You have web search enabled in your Hermes configuration

Step 1: Test the Search-Extract-Synthesize Pipeline

Before we automate anything, let's make sure the core pipeline works. Open your Hermes client (web UI, Telegram, whatever you prefer) and type:

```
Search the web for recent developments in AI agent frameworks and
give me a 3-paragraph summary of what's new this week.
```

You should see Hermes: 1. Call web_search to find relevant articles 2. Use web_extract on the most promising results 3. Synthesize the extracted content into a coherent summary 4. Deliver the result right there in your chat

If this works, the hard part is done. The automation is just scheduling this to run on its own.

Step 2: Create the Cron Job

Now tell Hermes:

```
Create a cron job: Every weekday at 7:00am, search the web for
recent developments in AI agent frameworks, write a 3-paragraph
summary, and send it to me on Telegram.
```

Hermes will: 1. Parse your natural-language schedule into a cron expression 2. Store the job with the specified task description 3. Confirm the schedule and delivery method

You should see a confirmation message like:

```
Cron job created:
Schedule: Every weekday at 7:00 AM
Task: Search the web for recent developments in AI agent
frameworks and send a summary
Delivery: Telegram
```

Step 3: Test It Immediately

You don't have to wait until tomorrow morning to see if it works. Tell Hermes:

```
Run the daily AI research digest cron job now
```

Hermes will execute the job immediately. Within a minute or two, you should receive a message on your chosen platform with the research summary. Check that: - The summary is coherent and relevant - It includes information that's actually recent (not from months ago) - It's formatted readably (not just a wall of text)

Step 4: Customize It

The beauty of this setup is how easy it is to modify. Here are some things to try:

- Change the topic: "Replace 'AI agent frameworks' with 'quantum computing'"
- Change the format: "Include bullet points with source links instead of paragraphs"
- Change the schedule: "Also send me a weekly summary on Friday afternoons"
- Add depth: "For each development, include a one-sentence implication for our business"

Each of these is just a natural-language instruction. No config files to edit, no scripts to modify. Just tell Hermes what you want differently, and it adjusts.

Step 5: Let It Run and Iterate

Let your digest run for a few days. You'll start to notice patterns — maybe the summaries are too long, too short, or missing a source you care about. Each adjustment is a simple instruction. Over time, you'll converge on a daily digest that's tailored exactly to your needs, delivered automatically, requiring zero effort from you after the initial setup.

That's the Hermes promise in a nutshell: describe what you want, and it figures out the how. You focus on the outcome; the agent handles the process.

Quick Reference: What You've Learned

- Hermes is a personal AI assistant that chains tools together — search, extract, synthesize, write, deliver — so you don't have to.
- As an always-on team member, it combines GitHub integration, cron scheduling, and multi-platform messaging to provide real value, not just noise.
- For developers, it's a co-pilot that can review code, run tests, automate browsers, and delegate specialized tasks to sub-agents.
- For writers, it's a muse and editor with creative skills and the LLM Wiki pattern for unlimited persistent memory.
- In the smart home, it bridges Home Assistant with natural language and cron-based routines, accessible from any messaging platform.
- As a research assistant, it monitors, synthesizes, and stores knowledge that compounds over time via session search and LLM Wiki.
- For security testing, it provides systematic adversarial testing, code review, and QA automation.
- As a multi-agent orchestrator, it delegates to specialists like Claude Code and Codex when tasks warrant deep expertise.
- It's not for everyone: it requires terminal comfort, self-hosting, and an acceptance of its async-first nature.

The use case that resonated with you — that one where you thought, “Oh, I could actually use that” — is the one to start with. Don’t try to set up everything at once. Pick one thing, make it work reliably, and then expand. That’s how I did it, and it’s how every Hermes user I know did it too.

Next chapter, we’ll get into the nuts and bolts of actually setting Hermes up — installation, configuration, and your first successful run. But before we get there, I’d encourage you to try the hands-on exercise above. There’s nothing like seeing your own automated workflow fire for the first time to make all this feel real.

Chapter 3: System Requirements and Installation

Let me tell you about the first time I tried to install Hermes. It was a Tuesday. I’d skimmed the docs (bad idea), skipped the prerequisites section (worse idea), and dove straight into the install command. Thirty minutes later, I was staring at a wall of Python errors, two broken virtual environments, and a coffee that had gone cold while I was troubleshooting. I didn’t have the right Python version. I didn’t have Node.js. I didn’t have Git configured. The installer couldn’t tell me what was wrong because it assumed I’d read the prerequisites I’d skipped.

This chapter exists so you don’t have a Tuesday like mine.

We’re going to walk through everything your system needs before Hermes will run, every way you can install it, and exactly what to do when something goes sideways. By the end, you’ll have a working Hermes installation and the confidence that comes from understanding why each piece matters — not just that it does.

3.1 Hardware Requirements

Here’s the good news upfront: Hermes is not a resource hog. Unlike local AI tools that need beefy GPUs and mountains of RAM, Hermes does all its heavy thinking at the LLM provider — OpenAI, Anthropic,

or whichever cloud service you configure. Your computer is essentially receiving text and displaying it. A potato could do that. Well, a potato from 2015 or later.

Minimum specs:

Component	Requirement
Computer	Any modern machine (2015+)
RAM	4 GB
Disk space	500 MB
GPU	Not required
Internet	Required (for cloud LLM)

Recommended specs:

Component	Requirement
RAM	8 GB+
Disk space	2 GB
GPU	Not required
Internet	Required (cloud) or optional (local via Ollama)

The jump from 4 GB to 8 GB RAM matters most if you plan to use browser automation. When Hermes drives a headless browser to scrape a website or fill in a form, Chromium eats memory like it's going out of style. With 4 GB total RAM, you'll feel the pinch. With 8 GB, you'll be comfortable. If you're the kind of person who keeps forty browser tabs open anyway, go for 16 GB and save yourself some anxiety.

No GPU? No Problem

This surprises people who've looked at other AI tools. Many local AI assistants need a GPU with 8 GB or more of VRAM just to load a model. Hermes doesn't. When you send a message, it travels to the cloud, the LLM generates a response, and the text comes back. Your GPU — if you even have one — sits this one out.

What If You Want to Run Local Models?

If you'd rather not send your prompts to a cloud service, Hermes supports Ollama as a local LLM backend. This is where GPU requirements come back into play. Running a local model like Llama 3 8B realistically needs:

- 8 GB VRAM for the 8B parameter model
- 16 GB+ VRAM for larger models (70B parameter and up)
- NVIDIA GPU preferred (AMD works on Linux with ROCm, but it's fussier)

If local models are your thing, Ollama handles the inference and Hermes talks to Ollama the same way it talks to OpenAI. You configure it once and forget about it. But for most beginners — and certainly for getting started — cloud LLMs are the path of least resistance, and they need zero GPU resources.

3.2 Software Requirements

Hardware gets you in the door. Software gets Hermes running. Here's what you need and, more importantly, why each piece matters.

Python 3.11 or later — Hermes is a Python application. It uses features introduced in Python 3.11 (like exception groups and the TaskGroup API for concurrent operations). If you're on Python 3.10 or earlier, the installer will fail with syntax errors that look like gibberish but are really just "you need a newer Python." The one-line installer bundles Python 3.11 on most platforms, which is why I recommend it.

Node.js 18 or later — This one's technically optional, and I'll explain why. Hermes uses Node.js for browser automation. When you ask it to "go to this website and find the price," it spins up a headless Chromium instance using the Playwright library, which requires the Node runtime. If you never plan to automate browsers, you can skip Node.js and everything else works fine. But browser automation is one of Hermes's most powerful features, and I think you'll want it. Install Node.

Git — Also technically optional, but useful. Hermes uses Git internally for conversation checkpoints (so you can roll back to a previous state), and some community skills are distributed as Git repositories. If you don't have Git, you lose checkpoints and some skill installations. Just install it.

ripgrep (rg) — This is a fast search tool that Hermes uses for searching your local files when you ask it to find something. The built-in grep works as a fallback, but rg is dramatically faster on large codebases. Recommended, not required.

Quick Check Script

Before you install anything, run this quick diagnostic to see where your system stands:

```
python3 --version && node --version && git --version && rg --
version | head -1
```

Here's what healthy output looks like:

```
Python 3.12.4
v20.11.0
git version 2.43.0
ripgrep 14.1.0
```

If any command fails with "command not found," you know what to install. If Python shows a version below 3.11, you need to upgrade. Here's a version that's more explicit about what you're missing:

```
echo "Python: $(python3 --version 2>&1 || echo 'NOT FOUND')"
echo "Node:   $(node --version 2>&1 || echo 'NOT FOUND')"
echo "Git:    $(git --version 2>&1 || echo 'NOT FOUND')"
echo "rg:     $(rg --version 2>&1 | head -1 || echo 'NOT FOUND
(optional)')"
```

Run that. Fix anything that's missing or too old. Then let's talk about your operating system.

3.3 Supported Operating Systems

Hermes runs on every major operating system, but the experience varies. Here's the honest breakdown.

macOS: First-Class Citizen

macOS is the primary development platform for the Hermes team. If you're on a Mac — whether it's an Intel model or Apple Silicon — everything works out of the box. The one-line installer handles the full setup, Homebrew integration exists for dependency management, and the team tests every release on macOS first.

What you need: macOS 12 (Monterey) or later. Any Mac from 2016 onward meets this easily.

Linux: Full Support, Distro Notes Apply

Linux gets full support and plenty of real-world testing. The team has verified Hermes on Ubuntu, Debian, Fedora, and Arch. Other distros generally work fine because Hermes only depends on standard Python and Node.js, but your mileage may vary on something exotic.

Distro-specific notes:

- **Ubuntu/Debian**: The smoothest Linux experience. The one-line installer works directly. If you need to install Python 3.11+, use deadsnakes PPA on Ubuntu or the system packages on Debian 12+.
- **Fedora**: Works great. Fedora tends to ship recent Python versions, so you're usually set. Install dev packages with sudo dnf groupinstall "Development Tools".
- **Arch**: You're an Arch user. You already know what to do. Python is always bleeding-edge. Just pacman -S python nodejs git ripgrep and move on.

One thing to watch on Linux: some minimal server installs don't include curl or ca-certificates. The one-line installer needs both. If the installer fails mysteriously early, sudo apt install curl ca-certificates (or your distro equivalent) usually fixes it.

Windows: WSL2 Required

Here's where I need to be direct. Hermes does not run natively on Windows. No .exe, no PowerShell module, no Windows installer. It relies on Unix-style process management, filesystem conventions, and shell tools that Windows doesn't provide natively.

The solution is WSL2 — Windows Subsystem for Linux. WSL2 gives you a full Linux environment inside Windows, and Hermes runs inside it perfectly. It's not a hack or a compromise; it's the intended way to use Hermes on Windows.

Setting up WSL2:

```
wsl --install
```

Run that in an elevated PowerShell session (right-click PowerShell, choose "Run as administrator"). This installs Ubuntu by default. Once it finishes, restart your computer, then launch "Ubuntu" from your Start menu. You'll be at a Linux terminal where Hermes works exactly like it does on native Linux.

If you already have WSL2 but want to update Ubuntu:

```
wsl --update
```

And if you need to check which WSL distros you have:

```
wsl --list --verbose
  NAME            STATE           VERSION
* Ubuntu          Running         2
```

Make sure it says VERSION 2. WSL1 won't work properly.

Docker: The Universal Option

If you don't want to install anything system-wide, or you're on an unusual platform, Docker is your friend. Hermes ships an official Docker image that bundles every dependency. You need Docker installed, but beyond that, it's one command.

Docker is also useful if you want to run Hermes on a server, in CI/CD pipelines, or if you just like keeping your system clean. We'll cover the Docker method in the installation section.

3.4 Installing Hermes

Four ways to install. I'll show you all of them, but most of you should use Method 1.

Method 1: The One-Line Installer (Recommended)

```
curl -fsSL https://hermes-agent.nousresearch.com/install.sh | bash
```

Here's what you'll see when you run it:

```
Hermes Agent Installer v0.9.0
============================

Checking system requirements...
  Python...    ✓ (3.12.4)
  Node.js...   ✓ (v20.11.0)
  Git...       ✓ (2.43.0)
  ripgrep...   ✓ (14.1.0)

Creating virtual environment at ~/.hermes/venv...
  Installing hermes-agent and dependencies...

Installing Hermes Agent...
  ✓ Core package installed
  ✓ Playwright browsers installed
  ✓ Shell completions added to ~/.bashrc

Hermes Agent v0.9.0 installed successfully!

Next steps:
  1. Run 'hermes setup' to configure your first model
  2. Run 'hermes doctor' to verify everything works
  3. Run 'hermes' to start your first conversation
```

Let me break down what happened there. The script:

1. Checked that all dependencies are present and recent enough
2. Created a Python virtual environment at ~/.hermes/venv (this keeps Hermes isolated from your system Python)
3. Installed the hermes-agent package and all its Python dependencies inside that venv
4. Installed Playwright browsers for web automation
5. Added shell completions so you can type hermes and hit Tab to see options

If anything fails, the script tells you what's missing and suggests how to fix it. It won't leave you with a half-installed mess.

What if you don't have Node.js? The installer will still proceed and install everything else. You'll see a warning like:

```
  Node.js...   ✗ (not found)
  ! Browser automation will be unavailable until Node.js 18+ is
installed
```

Hermes works without it. You just can't use browser tools.

Method 2: pip Install

If you prefer managing your own Python environment, or you already use pip for everything:

```
`uv`/installer venv --- not bare `pip install hermes-agent` into
system Python
Collecting hermes-agent
  Downloading hermes-agent-0.9.0-py3-none-any.whl (2.1 MB)
Collecting anthropic>=0.25.0
  Downloading anthropic-0.25.1-py3-none-any.whl (156 kB)
Collecting openai>=1.12.0
  Downloading openai-1.14.0-py3-none-any.whl (267 kB)
Collecting playwright>=1.40.0
  Downloading playwright-1.41.0-py3-none-any.whl (1.0 MB)
Collecting rich>=13.0.0
  Downloading rich-13.7.0-py3-none-any.whl (184 kB)
...
Installing collected packages: ... hermes-agent
Successfully installed hermes-agent-0.9.0
```

This works, but you should install inside a virtual environment. If you skip the venv, Hermes's dependencies can conflict with other Python packages on your system. That's a headache you don't need. Create a venv first:

```
python3 -m venv ~/.hermes/venv
source ~/.hermes/venv/bin/activate
`uv`/installer venv --- not bare `pip install hermes-agent` into
system Python
```

I should tell you about the first installation mistake I want you to avoid. Early on, I installed Hermes with `uv`/installer venv — not bare `pip install hermes-agent` into system Python directly into my system Python on Ubuntu. It worked for about a day. Then I updated an unrelated Python package, something in the dependency chain conflicted, and Hermes stopped working with an import error that pointed to a completely different package. I spent two hours thinking Hermes was broken when really I'd created a dependency mess in my system Python. The one-line installer avoids this by using a dedicated

virtual environment. If you use pip, create a venv. Always. I cannot stress this enough. My Ubuntu system Python took weeks to fully recover.

Method 3: From Source

This is for developers who want to modify Hermes, or users who need the absolute latest changes before they're released.

```
git clone https://github.com/NousResearch/hermes-agent.git
cd hermes-agent
pip install -e .
Cloning into 'hermes-agent'...
remote: Enumerating objects: 4821, done.
remote: Counting objects: 100% (1247/1247), done.
remote: Compressing objects: 100% (312/312), done.
remote: Total 4821 (delta 935), reused 1041 (delta 919), pack-
reused 3574
Receiving objects: 100% (4821/4821), 4.82 MiB | 12.4 MiB/s, done.
Resolving deltas: 100% (3112/3112), done.

Installing collected packages: ... hermes-agent
  Running setup.py develop for hermes-agent
Successfully installed hermes-agent-0.9.0
```

The -e flag means "editable" — you're installing from the local directory, and any changes you make to the source code take effect immediately without reinstalling. This is perfect if you want to:

- Contribute to Hermes development
- Run bleeding-edge features not yet in a release
- Debug an issue by adding print statements to the source

The source ends up at ~/.hermes/hermes-agent/ if you clone into the standard location. This is also where the one-line installer puts the repo if you need it later.

Method 4: Docker

```
docker run -v ~/.hermes:/root/.hermes nousresearch/hermes-agent
Unable to find image 'nousresearch/hermes-agent:latest' locally
latest: Pulling from nousresearch/hermes-agent
Digest:
sha256:a3f2b8c91d7e4f5602b1c9d8e7f3a6b5c4d2e1f0a8b7c6d5e4f3a2b1c0d9
Status: Downloaded newer image for nousresearch/hermes-
agent:latest
Hermes Agent v0.9.0 (Docker)
```

```
No configuration found. Run 'hermes setup' inside the container,
or mount
an existing ~/.hermes directory.
```

The -v ~/.hermes:/root/.hermes part is important. It mounts your local ~/.hermes directory into the container so your configuration, API keys, and conversation history persist between containers. Without it, everything disappears when the container stops.

Docker is especially useful for:

- Running Hermes on a server
- Keeping your system completely clean (no Python, Node, or other tools installed globally)
- Reproducible environments for testing
- CI/CD pipelines

If you want to run it interactively, add -it:

```
docker run -it -v ~/.hermes:/root/.hermes nousresearch/hermes-
agent
```

3.5 The Setup Wizard

Once Hermes is installed, you need to configure it. The setup wizard walks you through every section, explains what each option does, and gets you to a working configuration in about five minutes.

```
hermes setup
┌─ Hermes Setup Wizard ──────────────────────────────────────┐
│                                                            │
│  Welcome! This wizard will guide you through setting up    │
│  Hermes Agent. You can re-run this anytime with            │
│  'hermes setup'.                                           │
│                                                            │
│  Sections:                                                 │
│    1. Model  — Choose your LLM provider                    │
│    2. TTS    — Text-to-speech (optional)                   │
│    3. Terminal — Local, Docker, or Modal                   │
│    4. Gateway — Messaging platforms (optional)             │
│    5. Tools  — Enable/disable tools                        │
│    6. Agent  — Agent behavior settings                     │
│                                                            │
```

```
│  Press Enter to start, or 'q' to quit.                    │
└──────────────────────────────────────────────────────────┘
```

Let's walk through the important sections.

Model Selection

This is where you tell Hermes which LLM to use. You'll see something like:

```
┌─ Model Configuration ──────────────────────────────────────┐
│                                                            │
│  Choose your default LLM provider:                         │
│                                                            │
│  1. OpenAI        (GPT-4o, o3, etc.)                       │
│  2. Anthropic     (Claude 3.5 Sonnet, Claude 3 Opus)       │
│  3. OpenRouter    (Access to 100+ models)                  │
│  4. Nous Portal   (Nous Research models)                   │
│  5. Ollama        (Local models, no API key needed)        │
│                                                            │
│  Enter number [1-5]: 1                                     │
└────────────────────────────────────────────────────────────┘
```

Pick the provider you have access to. If you're not sure, OpenRouter is a good first choice because it gives you access to dozens of models through a single API key, and you can switch between them without reconfiguring.

API Key Entry

After selecting a provider, the wizard asks for your API key:

```
┌─ API Key ──────────────────────────────────────────────────┐
│                                                            │
│  Enter your OpenAI API key.                                │
│  (Input is hidden for security)                            │
│                                                            │
│  API key: ********                                         │
│                                                            │
│  Don't have one? Get one at:                               │
│  https://platform.openai.com/api-keys                      │
└────────────────────────────────────────────────────────────┘
```

Where to get API keys for each provider:

- **OpenAI**: https://platform.openai.com/api-keys — Sign up, add a payment method, generate a key. You'll need at least $5 of credit.

- **Anthropic**: https://console.anthropic.com/ — Sign up for the API, create a key in the dashboard. Usage-based billing.
- **OpenRouter**: https://openrouter.ai/keys — Create an account, generate a key. You can add credits and use any model they host.
- **Nous Portal**: https://portal.nousresearch.com/ — Sign up for access to Nous Research's hosted models.
- **Ollama**: No API key needed. Ollama runs locally. Just make sure the Ollama service is running (ollama serve).

The API key gets stored in ~/.hermes/.env, which is a file you should never share, commit to Git, or post in a Slack channel. Ask me how I know. Actually, don't. It was humiliating. The second installation mistake I'll share: I once pasted my API key into a public GitHub issue while debugging a problem. Within forty-seven seconds — I timed it — a bot scraped the key and started running up charges on my OpenAI account. I got an email from OpenAI about unusual usage before I even finished typing the issue. Revoke the key immediately, add .env to your .gitignore, and treat API keys like the credentials they are.

Terminal Setup

The terminal section asks how you want Hermes to run commands on your machine:

```
┌─ Terminal Configuration ──────────────────────────────────┐
│                                                           │
│  How should Hermes run terminal commands?                 │
│                                                           │
│  1. Local  — Run directly on this machine (recommended)   │
│  2. Docker — Run inside a Docker container                │
│  3. Modal  — Run on Modal cloud infrastructure            │
│                                                           │
│  Enter number [1-3]: 1                                    │
└───────────────────────────────────────────────────────────┘
```

For most beginners, "Local" is the right answer. Docker and Modal are for advanced setups where you want Hermes to run commands in an isolated or cloud environment. You can change this later.

Gateway Setup

This is optional and you can skip it on first setup. Gateway connects Hermes to messaging platforms — Discord, Slack, Telegram, and others. If you just want to use Hermes in your terminal, skip this section. You can configure it later when you're ready to have Hermes respond to messages in your Discord server or Slack workspace.

Tools and Agent Settings

The Tools section lets you enable or disable specific capabilities:

```
┌─ Tools Configuration ──────────────────────────────────┐
│                                                        │
│  Enable/disable Hermes tools:                          │
│                                                        │
│  [x] terminal    — Run shell commands                  │
│  [x] browser     — Web automation (requires Node.js)   │
│  [x] search      — Web and local search                │
│  [x] edit        — File creation and editing           │
│  [ ] vision      — Image analysis (needs vision-capable │
│                    model)                              │
│  [x] memory      — Remember things across sessions     │
│                                                        │
│  Toggle with spacebar. Enter to confirm.               │
└────────────────────────────────────────────────────────┘
```

The Agent section handles behavior preferences — things like how verbose Hermes should be, whether it should ask for confirmation before running commands, and how many steps it can take autonomously. The defaults are sensible for beginners. Accept them and move on.

Once the wizard finishes, you'll see:

```
✓ Configuration saved to ~/.hermes/config.yaml
✓ API key stored in ~/.hermes/.env
✓ Memory files initialized in ~/.hermes/memories/

You're all set! Run 'hermes doctor' to verify, or 'hermes'
to start chatting.
```

3.6 Verifying Your Installation

Before you start chatting with Hermes, let's make sure everything is working. Hermes includes a built-in diagnostic that checks every component.

Running hermes doctor

```
hermes doctor
Hermes Doctor — System Diagnostic
==================================

Checking core dependencies...
  Python.........  ✓  3.12.4 (/home/mike/.hermes/venv/bin/python3)
  Virtual env.....  ✓  Active at ~/.hermes/venv
  Hermes SDK......  ✓  v0.9.0

Checking configuration...
  config.yaml.....  ✓  Found (v14)
  .env............  ✓  Found (1 key configured)

Checking API connectivity...
  OpenAI..........  ✓  Key valid, quota available

Checking directories...
  ~/.hermes/.......  ✓
  skills/.........  ✓
  memories/.......  ✓
  sessions/.......  ✓
  cron/...........  ✓
  logs/...........  ✓

Checking optional tools...
  Node.js.........  ✓  v20.11.0
  Playwright......  ✓  Browsers installed
  ripgrep.........  ✓  v14.1.0
  Git.............  ✓  v2.43.0

All checks passed! Hermes is ready to go.
```

Every line tells you something. Let's decode the output.

Understanding Doctor Output

The diagnostic is organized into sections, and each line shows a status icon:

- ✓ **(checkmark)** — This component is working correctly.

- **✗ (cross)** — Something is missing or broken. The output will include a suggestion.
- **! (warning)** — It works, but there's a caveat you should know about.

The config version number (v14 in the example above) tells you which schema version the file uses. As Hermes evolves, the configuration format may change. If a newer version is available, hermes doctor will note it and hermes config migrate can update your file. Running hermes setup again or hermes update will also handle any needed migration.

Common First-Run Issues

Even with a clean install, small things can go wrong. Here are the issues I see most often from beginners, with their fixes.

Missing API key

```
  .env.............  ✗  Not found
  API keys........  ✗  No provider configured
```

Fix: Run hermes setup and enter your API key in the model section. Or manually create ~/.hermes/.env:

```
echo "OPENAI_API_KEY=your_key_here" > ~/.hermes/.env
chmod 600 ~/.hermes/.env
```

The chmod 600 restricts file permissions so only your user can read it. Do this.

Wrong Python version

```
  Python.........  ✗  3.10.12 (need 3.11+)
```

Fix: Install Python 3.11 or later. On Ubuntu:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.12 python3.12-venv
```

On macOS:

```
brew install python@3.12
```

Then re-run the one-line installer, which will use the new Python.

Node.js not found

```
  Node.js.........  ✗  Not found
  Playwright......  ✗  Node.js required
```

Fix: Install Node.js 18+. The easiest way is via the official installer at https://nodejs.org/ or via a package manager:

```
### macOS
brew install node

### Ubuntu/Debian
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install nodejs

### Fedora
sudo dnf install nodejs
```

Hermes works without Node.js, but browser tools won't be available.

Config version mismatch

```
  config.yaml.....  !  Found (v14) — newer version available
```

Fix: Run hermes setup to reconfigure, which auto-migrates the config. Or run hermes update which also handles migration. Your existing settings are preserved; only new or changed fields are updated.

Running Your First Chat

Once hermes doctor passes, start your first conversation:

```
hermes
```

Or equivalently:

```
hermes chat
```

```
┌──────────────────────────────────────────────────────────┐
│  Hermes Agent v0.9.0                                     │
│  Model: gpt-4o (OpenAI)                                  │
│  Type your message and press Enter. Ctrl+C to exit.      │
│  Use /help for commands.                                 │
└──────────────────────────────────────────────────────────┘

You> Hello! Can you hear me?

Hermes> Hello! Yes, I can hear you perfectly. I'm Hermes,
your AI assistant. I'm connected to GPT-4o via OpenAI, and
I have access to tools like terminal, browser, search,
and file editing. What would you like to work on today?
```

```
You>
```

If you see that response, everything is working. Your installation is complete, your API key is valid, and Hermes is chatting with the LLM provider successfully. Congratulations.

Let me also mention the version check:

```
hermes --version
Hermes Agent v0.9.0 (v2026.4.13)
```

That v2026.4.13 is the build date. Useful if you're reporting a bug and someone asks "which version are you on?"

3.7 Updating Hermes

Software evolves. Hermes gets frequent updates — new tools, bug fixes, better model support, and sometimes configuration format changes. Keeping up to date is important, and Hermes makes it easy.

The Easy Way

```
hermes update
Checking for updates...
  Current version: v0.9.0
  Latest version:  v0.9.1

Updating Hermes Agent...
  Downloading hermes-agent-0.9.1...
  Installing...
  Migrating config v14 → v15...
    + Added: agent.auto_save_interval
    + Added: tools.vision.max_image_size
  ✓ Updated to v0.9.1

Run 'hermes doctor' to verify, or just start chatting!
```

One command. That's it. The updater downloads the new version, installs it, and migrates your configuration if the format has changed.

Preview Without Applying

If you're cautious (and that's fine), check what's available first:

```
hermes update --check
Checking for updates...
```

```
  Current version: v0.9.0
  Latest version:  v0.9.1

Changelog:
  - Fixed: Browser tool crash on redirected URLs
  - Fixed: Memory file not loading on fresh installs
  - Added: Auto-save interval setting for agent conversations
  - Changed: Default model timeout increased to 120s

No changes applied. Run 'hermes update' to upgrade.
```

This shows you what the update contains without changing anything. Read the changelog, decide if you want the fixes and features, then run hermes update when you're ready.

Config Migration

As I mentioned, configuration formats evolve. As of v0.9.0, the default config schema is at version v14. When Hermes updates, new fields may be added and the schema version may increment. Each time the version bumps, new fields are added or old fields are reorganized.

The updater handles this automatically. When Hermes updates and migrates config (e.g., "v14 → v15"), it's adding new fields with sensible defaults and moving any reorganized fields to their new locations. Your existing settings — API keys, model choices, tool preferences — are preserved. The migration is non-destructive.

That said, Hermes keeps a backup. Before any config migration, the previous config is saved to ~/.hermes/config.yaml.bak. If something goes wrong after an update, you can restore it:

```
cp ~/.hermes/config.yaml.bak ~/.hermes/config.yaml
```

When Updates Break Things

It happens. Rarely, but it happens. An update might introduce a bug, or a config migration might not preserve an edge-case setting you relied on. Here's what to do:

1. **Check hermes doctor** — Run it after the update. If something broke, doctor will usually flag it.

2. **Check the GitHub issues** — https://github.com/NousResearch/hermes-agent/issues — Other people may have reported the same problem, and the team often responds quickly.

3. **Rollback** — If you installed via pip, you can install a specific older version:

- pip install hermes-agent==0.9.0

   If you used the one-line installer, you can re-run it with a version tag:

   ```
   curl -fsSL https://hermes-agent.nousresearch.com/install.sh | bash
   ```

1. **Restore config backup** — As mentioned, ~/.hermes/config.yaml.bak has your previous config.

2. **Report the bug** — If you've confirmed the issue is in the new version, file an issue on GitHub. Include your hermes doctor output and the version numbers before and after the update. The Hermes team is responsive.

Staying on Top of Releases

A few ways to know when a new version drops:

- **GitHub releases**: https://github.com/NousResearch/hermes-agent/releases — Watch the repository for release notifications.
- **hermes update --check**: Make it a habit to run this periodically. Some people add it to their shell startup.
- **Discord/Community**: The Hermes community channels usually announce releases within minutes.

Don't let your installation fall too far behind. Three or four versions back is usually fine, but if you're six months out of date, updating might involve multiple config migrations, and the further back you are, the more likely something will need manual attention.

What Lives Where: The Directory Structure

When Hermes installs and you complete setup, it creates a specific directory structure at ~/.hermes/. Understanding what lives where helps you troubleshoot, back up your data, and customize your setup.

```
~/.hermes/
├── config.yaml          Main configuration file
├── .env                 API keys and secrets (never share this)
```

```
├── venv/                Python virtual environment
├── skills/              Installed skills (community and custom)
├── memories/
│   ├── MEMORY.md        Hermes's long-term memory
│   └── USER.md          Your user profile and preferences
├── sessions/            Conversation history (SQLite + JSON)
├── cron/                Scheduled jobs
├── logs/                Agent execution logs
└── hermes-agent/        Source repository (if cloned)
```

Each directory has a purpose:

- **config.yaml** — The main settings file. This is what hermes setup writes to and what hermes update migrates. You can edit it directly, but hermes setup is safer for beginners.
- **.env** — Where API keys live. This file should have 600 permissions (read/write for your user only). Never commit this to version control.
- **venv/** — The isolated Python environment where Hermes and its dependencies live. Don't modify this manually. The installer manages it.
- **skills/** — Community-built skills that extend Hermes's capabilities. Each skill is a directory containing configuration and code. We'll get deep into skills in Chapter 7.
- **memories/** — Hermes's persistent memory. MEMORY.md stores things Hermes decides to remember across conversations. USER.md stores preferences and facts about you. Both are plain Markdown files you can read and edit.
- **sessions/** — Your conversation history. Stored as SQLite databases and JSON files. Hermes can resume previous sessions.
- **cron/** — Scheduled tasks. If you tell Hermes "remind me every Monday at 9am," the cron entry lives here.
- **logs/** — Execution logs. Useful for debugging. If something goes wrong and you want to see exactly what Hermes did, the logs are here.
- **hermes-agent/** — The source repository, present if you cloned from GitHub or if the installer included it. Not required for normal operation.

If you ever need to do a completely fresh start — wipe everything and begin from scratch — the answer is:

```
rm -rf ~/.hermes
```

Then re-run the installer and setup wizard. But back up ~/.hermes/memories/ and ~/.hermes/.env first if you want to keep your settings and conversation memory.

Try It Now: Full Installation Walkthrough

This exercise takes you from a bare system to a working Hermes conversation. If you've already installed Hermes while reading this chapter, you can still follow along — the steps are idempotent, meaning running them again won't break anything.

Step 1: Check your system prerequisites

Copy and paste this diagnostic script:

```
echo "=== Hermes Prerequisites Check ==="
echo "Python: $(python3 --version 2>&1 || echo 'NOT FOUND — install Python 3.11+')"
echo "Node:   $(node --version 2>&1 || echo 'NOT FOUND — optional, for browser tools')"
echo "Git:    $(git --version 2>&1 || echo 'NOT FOUND — optional, for checkpoints')"
echo "rg:     $(rg --version 2>&1 | head -1 || echo 'NOT FOUND — optional, for fast search')"
echo "================================="
```

Expected output on a properly configured system:

```
=== Hermes Prerequisites Check ===
Python: Python 3.12.4
Node:   v20.11.0
Git:    git version 2.43.0
rg:     ripgrep 14.1.0
=================================
```

If Python is missing or below 3.11, install it before continuing. If Node, Git, or rg are missing, install them or accept that those specific features won't be available.

Step 2: Install Hermes

macOS or Linux — use the one-line installer:

```
curl -fsSL https://hermes-agent.nousresearch.com/install.sh | bash
```

Windows — open WSL2 (Ubuntu), then run the same command inside WSL.

Alternative, if you prefer pip:

```
pip install hermes-agent
```

Step 3: Run the setup wizard

```
hermes setup
```

Work through each section: 1. Pick your LLM provider (OpenAI, Anthropic, OpenRouter, Nous Portal, or Ollama) 2. Enter your API key when prompted 3. Choose “Local” for terminal setup 4. Skip gateway setup (configure it later) 5. Accept the default tool settings 6. Accept the default agent settings

Step 4: Verify with hermes doctor

```
hermes doctor
```

Check that every line shows a √. If anything shows ✗, read the suggestion in the output and fix it before continuing.

Step 5: Start your first conversation

```
hermes
```

Type a message and press Enter. When you get a response, your installation is complete and verified. You’re ready for the rest of this book.

Step 6: Check your version

```
hermes --version
```

You should see Hermes Agent v0.9.0 (v2026.4.13) or a newer version if you’re reading this after a release.

Step 7: (Optional) Verify the directory structure

```
ls -la ~/.hermes/
total 36
drwxr-xr-x  8 mike mike 4096 Apr 16 11:12 .
drwxr-xr-x 20 mike mike 4096 Apr 16 11:05 ..
-rw-------  1 mike mike   42 Apr 16 11:12 .env
-rw-r--r--  1 mike mike 2847 Apr 16 11:12 config.yaml
drwxr-xr-x  2 mike mike 4096 Apr 16 11:12 cron
drwxr-xr-x  3 mike mike 4096 Apr 16 11:12 hermes-agent
```

```
drwxr-xr-x  2 mike mike 4096 Apr 16 11:12 logs
drwxr-xr-x  2 mike mike 4096 Apr 16 11:12 memories
drwxr-xr-x  2 mike mike 4096 Apr 16 11:12 sessions
drwxr-xr-x  2 mike mike 4096 Apr 16 11:12 skills
drwxr-xr-x  4 mike mike 4096 Apr 16 11:12 venv
```

Confirm that config.yaml, .env, memories/, sessions/, and skills/ all exist. If they do, you're in good shape.

That's installation done. You now have a working Hermes Agent, configured and verified, ready to help you with real tasks. In the next chapter, we'll dive into your first real conversation — not just "hello, can you hear me," but actually using Hermes to accomplish something useful. Bring a task you've been putting off. Hermes is going to help you finish it.

Chapter 4: Your First Conversation — The Basics

You've installed Hermes. You've stared at the terminal. Now what?

It's time to actually talk to the thing. In this chapter, we'll walk through your first real conversation — not a toy example, but the actual experience of working with Hermes step by step. You'll learn how to start a chat, how Hermes thinks behind the scenes, how to keep yourself safe with the approval system, and how to make the whole experience feel like second nature.

By the end, you'll have run a complete 10-minute session that searches the web, creates a file, and saves a memory. Let's get into it.

4.1 Starting a Chat — Three Ways In

Here's something that tripped me up early on: there isn't just one way to start talking to Hermes. There are three. And each one serves a different purpose. Let me show you all of them, starting with the one you'll use most.

Interactive Mode: The Main Way

Type hermes in your terminal and press Enter. That's it.

```
$ hermes
```

Or equivalently:

```
$ hermes chat
```

Both commands do the same thing — they drop you into an interactive session where you type messages and Hermes responds, back and forth, until you're done. This is the mode you'll use for 90% of your work. It's conversational, fluid, and lets you build on each exchange.

Once you're in, you'll see a prompt waiting for you. Type your first message and hit Enter. Hermes will think for a moment and respond. Then you respond. Then Hermes responds. It's a conversation — just one where the other party can also read your files, run commands, and search the internet.

One-Shot Mode: Quick Answers

Sometimes you don't want a whole conversation. You just want a quick answer and you want to get back to whatever you were doing. That's what one-shot mode is for:

```
$ hermes chat -q "What can you do?"
```

Hermes processes your question, prints the answer to your terminal, and exits. No interactive session. No back-and-forth. Just ask and receive.

I use this constantly for quick lookups — things like "What's the syntax for destructuring in Python?" or "What flag does grep use for line numbers?" It's faster than opening a browser tab and less disruptive than starting a full chat.

One-shot mode is also great for scripting. You can pipe its output into other commands or embed it in shell scripts. But for now, just think of it as your quick-question side channel.

Resume Mode: Pick Up Where You Left Off

Here's a scenario that happens all the time: you're deep in a conversation with Hermes, you've built up great context about a project, and then… you close the terminal. Or your laptop dies. Or you get pulled into a meeting.

The good news is Hermes never forgets — at least, not within a session.

To resume your last conversation:

```
$ hermes -c
```

The -c flag means “continue.” It loads the most recent session and picks up right where you left off, with all the context intact.

If you want to resume a specific older session, you can do that too:

```
$ hermes -r SESSION_ID
```

Or the long form:

```
$ hermes --resume SESSION_ID
```

But how do you find the session ID? We’ll cover that in section 4.6 when we talk about finding and searching past sessions. For now, just remember -c for “continue my last chat.”

Which One Should You Use?

Mode	Command	Best For
Interactive	hermes	Most work — anything requiring back-and-forth
One-shot	hermes chat -q "..."	Quick questions, scripting
Resume	hermes -c	Continuing interrupted work

Start with hermes. It’s the default for a reason.

4.2 How Hermes Thinks — The Agent Loop

Before we go further, I want to pull back the curtain on something. When you send a message to Hermes, it doesn’t just generate text. It runs through a loop — a repeating cycle of thinking, choosing, executing, and observing. Understanding this loop is the single most important thing that separates people who “use Hermes” from people who *work with* Hermes.

Let me show you what I mean. Here’s what happens when you type a message and press Enter:

The Five-Step Loop

1. Think: Hermes processes your entire conversation so far and decides what to do. This isn't just generating words — it's reasoning about whether your request needs action (like reading a file or running a command) or can be answered directly from knowledge.

2. Choose tool: If your request requires external action, Hermes selects the right tool from its toolkit. Need to read a file? That's read_file. Need to search the web? That's web_search. Need to run a shell command? That's terminal. Hermes picks the best tool for the job.

3. Execute: Hermes runs the tool with the appropriate arguments. This is where the real world meets the conversation — files get read, commands get run, web pages get fetched.

4. Observe: Hermes processes the tool's output. If read_file returns a 500-line Python script, Hermes reads through it. If terminal returns an error, Hermes sees the error message. The tool output becomes part of the conversation.

5. Repeat: Hermes goes back to step 1 with the new information. It might need another tool call, or it might now have enough context to give you a final answer. The loop continues until the task is complete.

Why You See Tool Calls in the Output

When you're chatting with Hermes, you'll sometimes see something like this in the middle of a response:

```
[read_file] Reading src/main.py...
| 1  | import sys
| 2  | def main():
| 3  |     print("hello")
```

That's not noise — that's the agent loop in action. Hermes is showing you step 3 (execute) and step 4 (observe) in real time. It read the file, saw the contents, and is now using that information to continue its response.

Beginners sometimes get confused by this. They expect a chatbot that only produces words, no different from messaging a friend. But Hermes is an *agent* — it takes actions. Those tool calls are the actions. When you see a tool call, Hermes isn't just talking about your code; it's actually reading it, running it, or modifying it. That's the whole point.

The max_turns Safety Valve

The agent loop is powerful, but what happens if Hermes gets stuck in an infinite loop — thinking, choosing, executing, observing, but never finishing? That's where max_turns comes in. By default, Hermes caps out at 90 turns through the loop. If it hits that limit, it stops and reports what it accomplished so far.

In practice, I've almost never hit this limit. Most tasks complete well within 90 turns. But it's reassuring to know there's a ceiling. If you're working on a truly massive task and hit the limit, you can simply continue the conversation — Hermes will start a fresh 90-turn cycle with all your existing context.

The Tool-Calling Paradigm: Actions, Not Just Words

Here's the mental shift that makes everything click: Hermes isn't a chatbot. It's a decision-making engine that can take actions.

A regular chatbot operates like this: you send text, it sends text back. The entire interaction lives in the realm of language. Hermes operates differently: you send text, it *decides what to do*, and then it *does things in the real world*. The text is just how you communicate your intent. The tools are how Hermes carries it out.

This is the "tool-calling paradigm," and once it clicks, you'll never think about AI assistance the same way. You're not talking to a smart dictionary. You're directing an agent that can actually operate on your system.

4.3 Talking to Hermes — Best Practices

Now you know how to start a chat and how Hermes thinks. But *what* you say matters just as much as how the system works. Let me show you what good communication looks like — and what bad communication looks like, because I've made every mistake in the book.

Be Specific vs. Vague

Here's a vague prompt:

```
You: fix my code
```

And here's what Hermes has to work with: no file path, no language, no error message, no description of what "fix" means. Hermes will have to ask you a bunch of clarifying questions before it can do anything useful. That's not a disaster, but it wastes time.

Now here's a specific prompt:

```
You: Fix the TypeError in src/utils.py line 47. The format_date
function receives a list instead of a string and crashes when
calling .split() on it.
```

With that prompt, Hermes can jump straight to the file, read the relevant function, see the problem, and propose a fix. Specific prompts get specific results, and they get them faster.

Give Context Before Asking

Hermes is smart, but it can't read your mind. If you're working on a Django project and you need help with a view function, don't just paste the function. Tell Hermes you're working in Django first:

```
You: I'm building a Django 4.2 app with a PostgreSQL backend.
The UserPreferences view in accounts/views.py returns a 500
when the user has no preferences object yet. Can you fix the
get_object_or_404 call?
```

That context — Django 4.2, PostgreSQL, the specific error scenario — lets Hermes narrow down the problem space immediately. Without it, Hermes might suggest solutions that don't apply to your stack.

Context is especially important when you're resuming a session. Even though Hermes remembers your conversation, a quick reminder of where you left off never hurts: "We were working on the authentication refactor. Where did we get to?"

Correct Mistakes Openly

Hermes will sometimes get things wrong. When it does, don't just say "that doesn't work." Tell it *why*:

```
Hermes: The function accepts a dictionary with "name" and
         "age" keys.

You: That's not right — the function takes a list of tuples,
     not a dict. Look at the type hints on line 12.
```

This kind of correction does two things. First, it gives Hermes the information it needs to self-correct. Second, it trains the conversation context — Hermes will remember your correction and be less likely to make the same mistake in future turns within that session.

I've sometimes felt rude correcting an AI so bluntly. Don't. Hermes doesn't have feelings. It has a context window. Every correction you give it is a gift that makes the rest of the conversation better.

Ask for Explanations of Tool Choices

When Hermes picks a tool, you can always ask why:

```
Hermes: [web_search] Searching for "Django 4.2 release notes"...

You: Why did you search the web instead of checking the
     local docs in my project?
```

Maybe Hermes didn't know you had local docs. Maybe it decided web results would be more current. Either way, asking teaches you how Hermes thinks and teaches Hermes about your project structure. It's a two-way street.

The Power of Iteration

The most underrated thing about working with Hermes is iteration. You don't need to get your prompt perfect on the first try. Send something rough, see what Hermes does, then refine:

```
You: Make this script faster.

Hermes: I've optimized the loop by using list comprehension
         and caching the database call...

You: Good start, but it's still slow on the file I/O part.
     Can you profile just the read_csv function?

Hermes: [terminal] Running cProfile on read_csv...
```

Each round of feedback narrows the problem. Hermes gets better with every exchange because it has more information about what you actually want. This is fundamentally different from a one-shot search engine query — it's a collaborative loop where both sides improve over time.

4.4 The Approval System — Your Safety Net

Hermes can run commands on your system. That's what makes it powerful. It's also what makes it dangerous. If Hermes decides to run rm -rf /, you'd better have a say in the matter.

That's what the approval system is for. Before Hermes executes any potentially dangerous command, it asks for your permission. You're in control.

Manual Mode: The Default

By default, Hermes runs in manual approval mode. This means every dangerous command — file deletions, package installs, system modifications — requires your explicit approval before Hermes can proceed.

When Hermes wants to run something risky, you'll see an approval request in your terminal. You can approve it, deny it, or ask Hermes to modify the command. Nothing happens without your say-so.

This is the safest mode, and it's the right choice for beginners. Get comfortable here before moving on.

Auto Mode: Speed vs. Risk

In auto mode, Hermes executes commands without asking. It's full autopilot — faster, but riskier:

```
$ hermes config set approvals.mode auto
```

I need to tell you a story about this.

When I first started using Hermes, I was so excited about the speed that I set auto mode on day one. Day two, I asked Hermes to "clean up my downloads folder." Hermes — being helpfully thorough — deleted everything in ~/Downloads. Including the installer for a piece of licensed software I'd paid $200 for and couldn't re-download.

That was my fault, not Hermes's. I gave a vague instruction in auto mode. The system did exactly what I asked, just not what I meant. I lost two hours re-downloading everything and never got that installer back.

The lesson: auto mode is a power-user feature. When you use it, you need to be precise about what you ask for. Vague instructions + auto mode = a bad time.

Suggest Mode: The Middle Ground

There's a compromise between manual and auto: suggest mode.

```
$ hermes config set approvals.mode suggest
```

In suggest mode, Hermes proposes commands but doesn't execute them. You see what Hermes *would* do, and then you decide whether to run it yourself. It's like having a very smart assistant who writes scripts for you but lets you press the button.

Suggest mode is fantastic for learning. You get to see Hermes's reasoning and proposed actions without any risk. I usually recommend beginners spend their first week in suggest mode just to build intuition about what Hermes tends to do.

How Hermes Presents Approval Requests

When you're in manual mode and Hermes wants to run a command, you'll see something like this:

```
Hermes wants to execute:
  pip install requests

Approve? [y/n/e(edit)]
```

You have three options:

- **y** — Yes, go ahead and run it.
- **n** — No, don't run it. Hermes will try a different approach.
- **e (edit)** — Modify the command before running it. Maybe you want pip install requests==2.31.0 instead.

There's also a timeout. If you don't respond within 60 seconds (the default approvals.timeout), the command is automatically denied. This prevents Hermes from sitting forever waiting for you to approve something at 2 AM when you've stepped away from the keyboard.

Changing Modes Mid-Session

Here's something I find really useful: you can change approval modes in the middle of a session. Start in manual mode while you're feeling out a task, then switch to auto once you're confident Hermes understands what you want:

```
You: From now on, use auto mode for pip install commands
     only. Keep manual mode for everything else.
```

Actually, the simplest way is to change it directly in config:

```
$ hermes config set approvals.mode suggest
```

You can run this in a separate terminal while Hermes is still going, and it'll pick up the change. Or you can just tell Hermes in the chat: "Switch to suggest mode." It'll handle it.

My Recommendation

For your first week:

1. Start in **manual mode** (the default — you don't need to do anything).
2. After a few days, try **suggest mode** to see what Hermes would do.
3. Only switch to **auto mode** when you can predict what Hermes will do before it does it.

If you catch yourself thinking "I wish Hermes would just do it without asking," that's the sign you might be ready for auto mode. But until then, let the safety net do its job.

4.5 Understanding Tool Output

When Hermes uses a tool, you see the results right in your terminal. But the output can look unfamiliar if you've never worked with an agent before. Let's decode it.

What a Tool Call Looks Like

Here's a typical tool call in your terminal:

```
[read_file] Reading config/settings.yaml
---
```

```
 1 | database:
 2 |   host: localhost
 3 |   port: 5432
 4 |   name: myapp_dev
 5 |
 6 | cache:
 7 |   enabled: true
 8 |   driver: redis
---
```

The first line shows the tool name (read_file) and what it's doing (Reading config/settings.yaml). Below the separator, you see the actual output — in this case, the file contents with line numbers.

Hermes sees this same output (it's part of the conversation), and then it continues reasoning based on what it found.

Terminal Output

When Hermes runs a terminal command, you get the raw output just like you would if you ran the command yourself:

```
[terminal] Running: pytest tests/ -v
---
==================== test session starts ====================
platform linux -- Python 3.11.4
collected 12 items

tests/test_utils.py::test_format_date PASSED        [  8%]
tests/test_utils.py::test_parse_config PASSED       [ 16%]
tests/test_utils.py::test_validate_email FAILED     [ 25%]
...
================== 1 failed, 11 passed in 2.3s ==============
---
```

If a test fails, Hermes sees the failure and can investigate the problem immediately — no need for you to paste the error back manually.

File Diffs

When Hermes edits a file, you'll see a diff showing exactly what changed:

```
[patch] Editing src/utils.py
---
-    result = items.get(key)
+    result = items[key]
---
```

The - line shows what was removed. The + line shows what was added. If you've ever used Git, this will feel familiar. If you haven't, just remember: red/minus means gone, green/plus means new.

Browser Snapshots

When Hermes uses browser tools, you see a snapshot of the page state — a text representation of what's on the screen:

```
[browser_navigate] Visiting https://docs.python.org/3/
---
[Page: Python 3 Documentation]
[Links: Tutorial, Library Reference, Language Reference...]
[Content: Welcome to the Python 3 documentation...]
---
```

This isn't a screenshot — it's a structured text summary. Hermes uses these snapshots to understand web content, and you can see what it sees.

When Tools Fail

Tools don't always work. Common failures include:

- **File not found:** Hermes tried to read a file that doesn't exist. Usually a typo in the path.
- **Command failed:** A terminal command returned a non-zero exit code. Hermes sees the error output.
- **Network error:** A web search or browser request couldn't connect. Hermes will often retry or try an alternative.

Here's what failure looks like in your terminal:

```
[read_file] Reading srce/main.py
---
Error: File not found: srce/main.py
Did you mean: src/main.py?
---
```

And here's the beautiful part: Hermes sees the error too, and it often self-corrects. In the example above, Hermes would likely try src/main.py on its own without you needing to say anything. The agent loop means failures aren't dead ends — they're information that feeds into the next iteration.

How Hermes Recovers

When a tool fails, Hermes typically does one of three things:

1. **Retries with a correction.** Wrong path? Try the right one. Wrong command syntax? Fix the flag.
2. **Tries an alternative tool.** Can't read a binary file with read_file? Try terminal with file to check the type first.
3. **Asks you for help.** If Hermes is truly stuck, it'll say so and ask for clarification.

You rarely need to intervene when a tool fails. The agent loop handles most problems automatically. But if you notice Hermes spinning — trying the same failing approach repeatedly — that's a good time to step in with a hint.

4.6 Ending and Resuming Sessions

Every conversation with Hermes comes to an end eventually. The question is: do you lose everything when it does?

The answer is no. Not even close.

Graceful Exit

To end a session, you have two options:

- Press **Ctrl+C** — the standard terminal interrupt.
- Type **exit** and press Enter.

Both are graceful shutdowns. Hermes saves your session state before closing. There's no "Are you sure?" prompt because everything is already saved.

Auto-Save: Every Conversation Is Stored

Hermes automatically saves every session to ~/.hermes/sessions/. You don't need to do anything to enable this — it just happens. Every message you send, every tool call Hermes makes, every response — it's all stored in a SQLite database called state.db.

On this machine, that database holds over 189 past sessions. That's 189 conversations I can pick up again at any time. Some are from weeks ago, and they're all still there, fully searchable.

Finding Past Sessions

To see a list of your conversations:

```
$ hermes sessions list
```

This shows your sessions with IDs, timestamps, and brief summaries. When you want to resume a specific one, copy its session ID.

Searching Sessions: Full-Text Search

Here's where it gets powerful. Hermes uses SQLite's FTS5 (full-text search 5) to index your conversation history. You can search across all your past sessions with:

```
$ hermes sessions browse
```

This returns every session where you discussed database migrations, ranked by relevance. I use this all the time when I remember discussing something with Hermes but can't remember which session it was in. It's like having a searchable journal of every technical conversation you've ever had.

Resuming a Session

Once you've found the session you want (or you just want your most recent one):

```
$ hermes -c
```

This resumes your last conversation. All the context is restored — Hermes remembers what files you were working on, what problems you encountered, and where you left off.

For a specific session:

```
$ hermes -r abc123def456
```

Replace abc123def456 with the actual session ID from hermes sessions list.

One thing to keep in mind: resuming doesn't mean Hermes remembers *everything* perfectly. If the conversation was very long, context compression may have summarized earlier parts. The most recent messages are always preserved at full fidelity (the last 20, by default), but the oldest parts of a very long session might be compressed summaries rather than exact quotes. We'll talk about compression more in a moment.

4.7 Customizing Your Display

Hermes gives you several ways to customize how information appears in your terminal. These don't change what Hermes does — they change what *you see* while it does it.

Streaming vs. Batch Output

By default, display.streaming is set to true. This means you see Hermes's response word by word as it generates — just like watching a chatbot type in real time. It's fast, it's engaging, and it lets you start reading before the response is finished.

If for some reason you prefer batch mode (where nothing appears until the entire response is complete), you can turn off streaming:

```
$ hermes config set display.streaming false
```

I've never met anyone who preferred batch mode. Streaming is just better for interactive use. But the option exists.

Compact Mode

By default, display.compact is false, which means you get full output — tool calls, reasoning, detailed results. If you turn on compact mode:

```
$ hermes config set display.compact true
```

Hermes shows a minimized version of tool output. File contents get truncated. Terminal output gets summarized. The idea is to reduce visual noise when you're working on tasks where you already know what the tools are doing and you just want the conclusions.

I toggle compact mode on and off depending on what I'm doing. Debugging? Full output, please. Running a familiar build? Compact mode is fine.

Personality Selection

This one's fun. Hermes supports 14 different display personalities. The default is kawaii — enthusiastic with cute expressions — but you can change it:

```
$ hermes config set display.personality kawaii
```

Think of it as choosing the right communication style for your workflow.

Show Reasoning Mode

By default, display.show_reasoning is false. Hermes's internal reasoning — the "think" step of the agent loop — happens invisibly. You see the tool calls and the final answer, but not the chain of thought that led to them.

If you want to see *why* Hermes chose a particular tool or approach:

```
$ hermes config set display.show_reasoning true
```

This reveals Hermes's reasoning process before each action. It's like seeing the math work, not just the answer. Absolutely invaluable for learning how the agent loop works, and I strongly recommend turning it on for your first week. After that, once you've built intuition, you can turn it back off to reduce clutter.

Tool Progress Visibility

```
$ hermes config set display.tool_progress all
```

This shows the full progress of every tool execution — spinner animations, partial results, the works. The all setting makes sure nothing is hidden. There are other options for reducing or filtering tool progress, but as a beginner, all gives you the most visibility into what Hermes is actually doing.

The Context Compression System

I mentioned context compression earlier, and this is a good place to explain it in detail. Hermes has a limited context window — a finite amount of token space for the conversation. In a long session, that space fills up.

When it does, Hermes's compression system kicks in automatically. Here's how it works:

- **compression.enabled: true** — Compression is on by default (and you should keep it on).
- **compression.threshold: 0.5** — Compression triggers when the context is 50% full. This gives Hermes headroom before the context is actually full.
- **compression.target_ratio: 0.2** — When compressing, the system summarizes the older parts of the conversation down to about 20% of their original size.
- **compression.protect_last_n: 20** — The most recent 20 messages are always preserved at full fidelity. They're never summarized.
- **compression.summary_model: google/gemini-3-flash-preview** — The AI model that generates the summaries. It's fast and efficient, optimized for this exact task.

The result: you can have very long conversations with Hermes without hitting context limits. The beginning of the conversation gets summarized, but the recent parts stay sharp. It's like how you might not remember the exact words of a conversation from an hour ago, but you remember what was discussed and you perfectly recall the last few minutes.

You don't need to configure any of this as a beginner. The defaults are solid. Just know that it exists, and if Hermes ever seems to "forget" something from early in a very long session, compression is the reason.

Try It Now: Your First 10-Minute Session

Enough reading. Let's do something.

This hands-on exercise walks you through a complete 10-minute session with Hermes. You'll search the web, create a file, and save a memory. By the time you're done, you'll have used three tools and experienced the agent loop firsthand.

Step 1: Start a Chat (1 minute)

Open your terminal and type:

```
hermes
```

Wait for the prompt to appear. You're now in interactive mode.

Step 2: Ask Hermes to Search the Web (2 minutes)

Type this message:

```
Search the web for "beginner Python projects 2025" and give me
a list of 5 good starter projects with one-sentence descriptions.
```

Watch what happens. You'll see Hermes invoke the web_search tool. If you're in manual mode, you might need to approve the action. The results will appear in your terminal as tool output, and then Hermes will synthesize them into a clean list.

Notice how Hermes didn't just make up five projects — it actually went and checked. That's the agent loop in action.

Step 3: Create a File (3 minutes)

Now ask Hermes to save those projects to a file:

```
Save that list to a file called ~/python-projects.md with
Markdown formatting. Add a title "Beginner Python Projects"
and today's date.
```

You should see Hermes use a file-writing tool. Watch the tool output — you'll see confirmation that the file was created. If you're in manual mode, approve the write.

Then ask Hermes to verify:

```
Read the file back to me.
```

Hermes will use read_file to show you the contents. This is a great habit — always verify after creating something.

Step 4: Save a Memory (2 minutes)

Tell Hermes to remember something about this session:

```
Save this to memory: I'm interested in Python beginner
projects. My current skill level is beginner. I prefer
project-based learning over reading tutorials.
```

Hermes will store this in its memory system. In future sessions — even new ones — Hermes can recall these preferences. It's like setting up a profile that persists across conversations.

Step 5: End and Resume (2 minutes)

Type exit to end the session.

Now try to resume it:

```
hermes -c
```

You should be back in your session with all the context intact. Ask Hermes:

```
What were the 5 Python projects we found?
```

Hermes should be able to tell you without searching again — the information is in the conversation history.

What You Just Did

In 10 minutes, you:

1. Started an interactive chat
2. Triggered a web search (saw the agent loop choose and execute a tool)
3. Created a file on disk (saw a file-writing tool in action)
4. Saved a persistent memory (saw the memory system at work)
5. Ended and resumed a session (experienced session persistence)

That's the full Hermes cycle: think, act, verify, persist. Every future session will follow this same pattern, just with more complex tasks. The fundamentals never change.

Welcome to the other side of the keyboard. You're not just using a tool — you're working with an agent. And now that you understand the basics of conversation, approvals, tool output, and sessions, you're ready for the real work to begin.

In the next chapter, we'll dive deeper into what Hermes can actually *do* with those tools — from managing your files to automating your workflows. But first, take a break. You've earned it. And when you come back, hermes -c will be waiting for you.

Chapter 5: The Config.yaml File — Your Hermes Control Panel

If you have ever rented an apartment, you know the feeling. You walk in, and everything is… fine. The walls are white. The thermostat is set to someone else's idea of comfortable. The kitchen faucet works, but it hits the wrong angle and splashes water onto the counter every single time. You could live with it. People do. But within a week you are adjusting the thermostat, rearranging the furniture, and googling "how to change a faucet aerator."

Hermes ships the same way. Out of the box, it works. The defaults are sane. But "sane" is not the same as "perfect for you," and the moment you want to change your model, tighten your security, or make your assistant talk like a pirate, you need to find the thermostat.

That thermostat is ~/.hermes/config.yaml.

This chapter is a room-by-room tour of that file. By the end, you will not just know what every setting does — you will know which ones matter for your workflow and which ones you can safely ignore. Let's get started.

5.1 Where Config Lives

Before we open the file, let's talk about where it lives and why that matters.

Your Hermes configuration file sits at:

```
~/.hermes/config.yaml
```

The ~ expands to your home directory. On most Linux systems, that is /home/yourusername/. On macOS, it is /Users/yourusername/.
The .hermes directory is a hidden folder (the leading dot tells the filesystem to hide it from casual browsing), and config.yaml is the single file that controls every aspect of how Hermes behaves.

Why a single file? Because Hermes follows the Unix philosophy: one file, plain text, human-readable. You can open it in any editor. You can version-control it. You can copy it to another machine and have an identical setup in seconds. No registry keys, no hidden databases, no binary blobs.

The very first line of every config.yaml tells Hermes what schema version to expect:

```
_config_version: 14
```

At the time of writing, the current schema is version 14. Hermes reads this line first and uses it to decide how to interpret everything that follows. If you ever see strange errors after an update, check this line — the schema version might have changed, and an older config might need migration.

Three Ways to Interact With Config

You have three options for reading and changing your config:

Option 1: Edit the file directly. Open ~/.hermes/config.yaml in your favorite text editor. This gives you the most control and the best view of your entire configuration at once. I use this method for 90% of my changes.

Option 2: The CLI shortcuts. Hermes provides a hermes config command with three subcommands:

```
hermes config set model.default gpt-4o
hermes config show display.personality
hermes config show
```

- set changes a single key
- show displays the current configuration (use hermes config show model to see a specific section)

These are fast and safe. Use them when you just need to tweak one value.

Option 3: The interactive setup wizard. Running hermes setup walks you through a guided configuration experience. This is how most beginners create their first config, and it is a perfectly fine starting point. Everything the wizard sets, you can change later in the file or via the CLI.

My recommendation? Use the wizard on day one. Graduate to the file by day three. You are on chapter five, so consider yourself graduated.

5.2 Model Configuration

Let's start with the setting everyone wants to change first: which AI model Hermes uses. Here is the default model block:

```
model:
  default: glm-5.1:cloud
  provider: custom
  base_url: http://localhost:11434/v1
```

Three keys, three decisions.

model.default is the name of the model Hermes sends every request to. The default, glm-5.1:cloud, means Hermes is talking to a locally-hosted model (more on that in the provider section). If you want to switch to, say, GPT-4o, you would change this to gpt-4o.

model.provider tells Hermes which provider backend handles the request. The default is custom, which means "whatever endpoint base_url points to." Other valid values include openrouter, openai-codex, anthropic, gemini, and about a dozen more (see Chapter 6 for the complete list). When you set provider to a known name, Hermes may apply provider-specific authentication and formatting automatically.

model.base_url is the API endpoint. The default http://localhost:11434/v1 is the standard Ollama server address. If you are using a cloud provider, you would replace this with their endpoint (for example, https://api.openai.com/v1 for OpenAI).

Config Before/After: Switching to OpenAI

Before:

```
model:
  default: glm-5.1:cloud
```

```
  provider: custom
  base_url: http://localhost:11434/v1
```

After:

```
model:
  default: gpt-4o
  provider: openai
  base_url: https://api.openai.com/v1
```

What changed? All three keys. The model name, the provider (now a known provider instead of generic custom), and the base URL pointing at OpenAI's servers instead of localhost. With this change, Hermes will send every request through the OpenAI API using your OPENAI_API_KEY environment variable.

Smart Model Routing

Not every prompt needs a powerful model. "What time is it?" does not require GPT-4o. Hermes has a feature called smart model routing that routes short, simple prompts to a cheaper, faster model and reserves the heavy lifting for your default model.

```
smart_model_routing:
  enabled: false
  max_simple_chars: 160
  max_simple_words: 28
  cheap_model: {}
```

By default, this is disabled. When enabled, Hermes checks each prompt: if it is 160 characters or fewer AND 28 words or fewer, the prompt goes to whatever model you define in cheap_model. Everything else goes to your default.

To enable it:

```
smart_model_routing:
  enabled: true
  max_simple_chars: 160
  max_simple_words: 28
  cheap_model:
    default: gpt-4o-mini
    provider: openai
    base_url: https://api.openai.com/v1
```

Now your quick questions hit a cheaper model, and your complex tasks still get the full-power treatment.

Fallback Providers

What happens when your primary provider goes down? Without fallback, Hermes simply fails. With fallback, it tries the next provider in line.

```
fallback_providers: []
```

The default is an empty list — no fallback. To add one:

```
fallback_providers:
  - provider: anthropic
    default: claude-3-5-sonnet
  - provider: openai
    default: gpt-4o
```

Now if your primary model is unreachable, Hermes tries Anthropic first, then OpenAI. Order matters: Hermes tries them in list order and uses the first one that responds.

5.3 Provider Setup

Hermes supports sixteen providers, and each one needs an API key. The providers block is where you configure authentication and endpoints for each provider beyond the default.

```
providers: {}
credential_pool_strategies: {}
```

The defaults are empty because Hermes auto-discovers credentials from environment variables. That is a deliberate design choice — your API keys should never live in a plain-text config file. Instead, you set them as environment variables, and Hermes picks them up automatically.

Here is how that works for the major providers:

Provider	Environment Variable	Common Models
OpenAI	OPENAI_API_KEY	gpt-4o, gpt-4o-mini
Anthropic	ANTHROPIC_API_KEY	claude-3-5-sonnet, claude-3-haiku
Google	GOOGLE_API_KEY	gemini-2.5-pro, gemini-3-flash
	None needed	any locally-loaded model

Provider	Environment Variable	Common Models
Local (Ollama)		
Custom	Varies	Varies

Set your API keys in your shell profile (.bashrc, .zshrc, or equivalent):

```
export OPENAI_API_KEY="sk-your-key-here"
export ANTHROPIC_API_KEY="sk-ant-your-key-here"
```

Never put API keys directly in config.yaml. The file is plain text. If you ever push it to a public repo, your keys are exposed. Environment variables stay on your machine. I cannot stress this enough — the single most common security mistake I see with new Hermes users is pasting API keys into the config file. It feels convenient at the time. It is not worth the risk.

If you need to override the auto-detected settings or add a custom provider, you populate the providers block:

```
providers:
  openai:
    base_url: https://api.openai.com/v1
  custom:
    "Local (localhost:11434)":
      base_url: http://localhost:11434/v1
```

The custom_providers key lets you define named custom providers that appear in model selection menus. For example:

```
custom_providers:
  "Local (localhost:11434)":
    base_url: http://localhost:11434/v1
```

This registers a provider called “Local (localhost:11434)” that points at your local Ollama instance. Once registered, it appears in the provider selection menu whenever you switch models interactively.

credential_pool_strategies is for advanced multi-key rotation. If you have multiple API keys for the same provider (say, three OpenAI keys for rate limit headroom), you can configure a pooling strategy here. For beginners, leave it as {}. When you start hitting rate limits on a high-traffic deployment, come back to this option.

A Quick Word on Provider Reliability

Not all providers are equally reliable. OpenAI and Anthropic have enterprise-grade uptime. Smaller providers can have outages. Local models depend entirely on your hardware staying on. This is exactly what fallback_providers (covered in the previous section) is designed to address. My personal setup uses a cloud provider as default with a local model as fallback — if my internet goes down, Hermes keeps working on Ollama, just with a less capable model. The reverse also works: local as primary, cloud as fallback for when you need more power than your GPU can provide.

5.4 Agent Behavior

The agent block controls how Hermes behaves as an autonomous agent — how long it thinks, how many steps it takes, and how strictly it follows tool-use rules.

```
agent:
  max_turns: 90
  gateway_timeout: 1800
  gateway_timeout_warning: 900
  service_tier: ''
  tool_use_enforcement: auto
  verbose: false
  reasoning_effort: medium
```

Let's break this down.

max_turns: 90 — This is the maximum number of back-and-forth cycles Hermes will perform in a single task. One "turn" is Hermes thinking, optionally calling a tool, and getting the result. Complex tasks like "refactor this entire codebase" might use many turns. Simple ones like "what is 2+2" use one. The default of 90 is generous. If you find Hermes stopping mid-task with a "max turns reached" message, increase this. If you want to prevent runaway agent loops, decrease it.

gateway_timeout: 1800 — The maximum wall-clock time, in seconds, for a single agent run. 1800 seconds is 30 minutes. If a task takes longer than this, Hermes terminates it.

gateway_timeout_warning: 900 — Hermes warns you at the halfway mark (900 seconds = 15 minutes) that the task is still running. This gives you a chance to cancel or let it continue.

service_tier: '' — For providers that support service tiers (like OpenAI's "auto" tier), this lets you specify which tier to request. The empty string means "use the default."

tool_use_enforcement: auto — Controls how strictly Hermes validates tool calls. auto means Hermes decides based on context. Other options include strict (validate every tool call rigidly) and lenient (allow more flexibility). Most users should leave this on auto.

verbose: false — When true, Hermes prints extra debugging information about its internal decision-making. Useful for troubleshooting. Noisy for daily use.

reasoning_effort: medium — How much "thinking" the model does before responding. Options are low, medium, and high. Higher effort means better reasoning on complex tasks but slower responses and higher token costs. Lower effort is faster and cheaper but may miss nuance on tricky problems.

Config Before/After: Heavy-Duty Research Mode

Before:

```
agent:
  max_turns: 90
  gateway_timeout: 1800
  gateway_timeout_warning: 900
  service_tier: ''
  tool_use_enforcement: auto
  verbose: false
  reasoning_effort: medium
```

After (for deep research tasks):

```
agent:
  max_turns: 200
  gateway_timeout: 7200
  gateway_timeout_warning: 3600
  service_tier: ''
  tool_use_enforcement: auto
  verbose: true
  reasoning_effort: high
```

What changed? I raised max_turns to 200 for complex multi-step research, doubled the timeout to two hours, enabled verbose logging so I can see what Hermes is doing, and cranked reasoning_effort to high. This config is expensive and slow, but for tasks like “analyze this 50-page document and produce a detailed report,” it is worth it.

The “I Messed Up” Story: Infinite Loops

I want to share a mistake. Early in my Hermes journey, I set max_turns to 500 and gateway_timeout to 0 (I thought zero meant “no limit” — it does not; it means “timeout immediately”). Then I asked Hermes to “explore this codebase and find every bug.”

What happened? Hermes timed out instantly on every single request because the gateway timeout was zero seconds. Every task failed immediately. No error message was helpful — it just said “gateway timeout” and stopped. I spent an hour reinstalling Hermes before I realized my mistake. The fix was one line:

```
gateway_timeout: 1800
```

Lesson: zero does not mean infinite in Hermes config. If you want no timeout, you still need to set a very large number. And when something breaks, check your config before you reinstall.

5.5 Terminal Configuration

Hermes can run shell commands on your behalf, and the terminal block controls how that works.

```
terminal:
  backend: local
  modal_mode: auto
  cwd: .
  timeout: 180
  docker_image: nikolaik/python-nodejs:python3.11-nodejs20
  singularity_image: docker://nikolaik/python-nodejs:python3.11-
nodejs20
  container_cpu: 1
  container_memory: 5120
  container_disk: 51200
  container_persistent: true
  persistent_shell: true
  lifetime_seconds: 300
```

This is one of the most powerful sections in the config, and understanding it is the difference between Hermes running commands safely in a sandbox and Hermes running commands directly on your machine. Choose wisely — the local backend means Hermes has full access to your host system. It can read any file your user account can read, install packages globally, and modify anything in your home directory. That power is useful, but it comes with risk. The docker backend isolates everything inside a container, which is safer but requires Docker to be installed and running.

backend: local — Where commands execute. The default is local, meaning commands run directly on your host machine in your working directory. Other options include docker (commands run inside a Docker container) and modal (commands run on Modal's cloud infrastructure). Additional backend options include singularity and daytona. For beginners, local and docker cover 99% of use cases.

modal_mode: auto — Controls Modal cloud behavior. auto lets Hermes decide when to use Modal. Other options are always and never.

cwd: . — The working directory for command execution. The dot means "current directory." Change this to any absolute path to make Hermes start every command session in a specific folder.

timeout: 180 — How long a command can run before Hermes kills it, in seconds. The default of 180 seconds (3 minutes) is fine for most tasks, but compilation or training jobs may need more.

Container Settings

The remaining keys define the Docker or Singularity container environment:

- **docker_image**: The container image used when backend is docker. The default includes both Python 3.11 and Node.js 20 — a solid general-purpose image.
- **singularity_image**: Same concept, but for HPC environments that use Singularity/Apptainer instead of Docker.
- **container_cpu: 1**: Number of CPU cores allocated to the container.
- **container_memory: 5120**: RAM allocated to the container (in MB). About 5 GB.
- **container_disk: 51200**: Disk space allocated (in MB). About 50 GB.

- **container_persistent: true**: When true, the container's filesystem persists between commands. This means installed packages and created files stick around.
- **persistent_shell: true**: When true, Hermes keeps the shell session alive between commands. Environment variables, shell history, and the current directory carry over.
- **lifetime_seconds: 300**: How long the container stays alive with no activity before Hermes shuts it down. Five minutes by default.

Config Before/After: Heavy Development Environment

Before (defaults):

```
terminal:
  backend: local
  container_memory: 5120
  container_cpu: 1
```

After (for ML/data work):

```
terminal:
  backend: docker
  container_memory: 16384MB
  container_cpu: 4
  timeout: 600
  lifetime_seconds: 900
```

I switched to Docker for isolation, quadrupled the memory, gave the container 4 CPU cores, raised the command timeout to 10 minutes for long-running scripts, and extended the idle lifetime to 15 minutes so the container does not shut down between exploratory commands.

5.6 Browser Configuration

Hermes can browse the web, and the browser block controls that behavior.

```
browser:
  inactivity_timeout: 120
  command_timeout: 30
  record_sessions: false
  allow_private_urls: false
  camofox.managed_persistence: false
```

inactivity_timeout: 120 — If Hermes is browsing a page and nothing changes for 120 seconds, it considers the page idle and moves on. Useful for preventing hangs on slow-loading sites.

command_timeout: 30 — Individual browser actions (click, scroll, type) time out after 30 seconds. If a page element is not responding, Hermes will not wait forever.

record_sessions: false — When true, Hermes records video of every browser session. This is incredibly useful for debugging — you can watch exactly what Hermes did on a website. But it uses disk space and can be slow, so it defaults to off.

allow_private_urls: false — When false, Hermes refuses to visit URLs on local networks (localhost, 192.168.x.x, 10.x.x.x, etc.). This is a safety measure to prevent the AI from accidentally probing your internal network. Set to true only if you explicitly want Hermes to test local web services.

camofox.managed_persistence: false — Camoufox is Hermes's anti-detection browser (a stealth-mode Firefox). managed_persistence controls whether Camoufox cookies and local storage persist between sessions. False means a fresh browser fingerprint every time, which is stealthier. True means cookies stick around, which is more convenient for sites that require logins.

Config Before/After: Web Scraping Setup

Before (defaults):

```
browser:
  inactivity_timeout: 120
  command_timeout: 30
  record_sessions: false
  allow_private_urls: false
  camofox.managed_persistence: false
```

After (for debugging web automation):

```
browser:
  inactivity_timeout: 300
  command_timeout: 60
  record_sessions: true
  allow_private_urls: true
  camofox.managed_persistence: true
```

What changed? I raised the timeouts because the site I was testing was slow and needed more time for each action. I enabled session recording so I could watch replays of what Hermes did on the website — invaluable when a scraper keeps failing and you cannot figure out why. I allowed private URLs because I was testing a local development server. And I enabled Camoufox persistence so login sessions would survive between page navigations.

Be careful with allow_private_urls: true. I only set that when I am explicitly testing something local. The default of false is a safety blanket I am glad to have.

5.7 Display and Personality

This is the fun section. The display block controls how Hermes talks to you — its personality, its visual style, and how much information it shows.

```
display:
  compact: false
  personality: kawaii
  resume_display: full
  busy_input_mode: interrupt
  bell_on_complete: false
  show_reasoning: false
  streaming: true
  inline_diffs: true
  show_cost: false
  skin: default
  tool_progress: all
  background_process_notifications: all
```

The 14 Personalities

Hermes ships with 14 built-in personalities. Your AI can talk like any of these:

1. **helpful** — The default assistant. Clear, thorough, professional.
2. **concise** — Short answers. No fluff. Perfect for quick lookups.
3. **technical** — Assumes you know the jargon. Deep, precise, reference-style.
4. **creative** — Lateral thinker. Unexpected connections, metaphors, ideas.

5. **teacher** — Explains everything step by step. Asks if you understood.
6. **kawaii** — Enthusiastic, uses emoticons, cheerful energy. (This is the default!)
7. **catgirl** — Nya~ Playful anime-energy persona.
8. **pirate** — Arrr. Talks like a pirate. Surprisingly capable.
9. **shakespeare** — Hath thou ever seen such eloquent code review?
10. **surfer** — Dude, that function is totally gnarly.
11. **noir** — Hard-boiled detective narrator. Moody. Dramatic.
12. **uwu** — Extreme cute. OwO what's this code?
13. **philosopher** — Questions assumptions. Existential depth.
14. **hype** — THIS IS THE MOST AMAZING CONFIG SETTING EVER!

The default personality is kawaii. Yes, really. Your first encounter with Hermes will include emoticons and cheerful energy. If that is not your style, change it:

```
display:
  personality: helpful
```

Personalities are more than cosmetic — they affect the system prompt and thus the quality of responses for certain tasks. technical produces better code documentation. teacher produces better explanations for beginners. pirate produces… entertainment.

Other Display Settings

compact: false — When true, Hermes uses a denser display with less whitespace and shorter output. Great for small terminals.

resume_display: full — When you resume a previous session, this controls how much of the history is shown. full replays everything. You can also set it to summary or none.

busy_input_mode: interrupt — What happens when you type while Hermes is still working. interrupt lets you break in and redirect. Other options include queue (wait your turn).

bell_on_complete: false — When true, your terminal bell rings when a long task finishes. Useful if you step away while Hermes is working.

show_reasoning: false — When true, Hermes displays its chain-of-thought reasoning before each response. Fascinating for learning how the AI thinks, but it doubles the output density.

streaming: true — When true, responses appear token by token as they are generated. When false, you see nothing until the entire response is ready. Streaming feels faster and more natural. The only reason to turn it off is if you are saving output to a file and want the complete result atomically.

inline_diffs: true — When Hermes edits a file, it shows the diff inline in the chat. When false, it just tells you the file was changed. I keep this on because seeing exactly what changed is far more useful than just knowing something changed.

show_cost: false — When true, every response includes an estimated cost. Helpful for budget tracking.

skin: default — Visual theme for the terminal UI. Currently the only option is default, but this is the hook for future themes.

tool_progress: all — How much detail you see about tool calls in progress. all shows everything. You can also use minimal or none.

background_process_notifications: all — Controls notifications about background processes. all shows every status update.

5.8 Memory Configuration

Hermes has a memory system. It can remember facts about you across conversations, and it can store important context from your current session. The memory block controls this.

```
memory:
  memory_enabled: true
  user_profile_enabled: true
  memory_char_limit: 2200
  user_char_limit: 1375
  provider: ''
  nudge_interval: 10
  flush_min_turns: 6
```

How Memory Works

Hermes maintains two memory files:

- **MEMORY.md** — General context and facts that Hermes has learned across sessions. Project details, preferences, notes. This file lives at ~/.hermes/MEMORY.md.
- **USER.md** — Facts specifically about YOU. Your name, your role, your coding style, your preferences. This file lives at ~/.hermes/USER.md.

memory_enabled: true — The master switch for MEMORY.md. When true, Hermes reads and writes to MEMORY.md. It accumulates knowledge across sessions.

user_profile_enabled: true — The master switch for USER.md. When true, Hermes builds and maintains a profile about you.

memory_char_limit: 2200 — Maximum characters in MEMORY.md. When the file exceeds this limit, Hermes compresses it by summarizing older content. 2200 characters is roughly 400-500 words, which is enough for a concise project overview and a handful of key facts. If your memory keeps getting compressed and losing important details, increase this value.

user_char_limit: 1375 — Maximum characters in USER.md. Smaller than the memory limit because user profiles should be concise. A 1375-character profile holds your name, role, preferred tech stack, coding style notes, and a few personal preferences — enough context for Hermes to tailor its responses without being overwhelming.

provider: '' — Which model to use for memory operations. Empty string means “use the default model.” You can specify a different model (like a cheaper one) if memory operations are hitting your token budget.

nudge_interval: 10 — Every 10 turns, Hermes “nudges” itself to check whether anything important should be saved to memory. This is how it decides “oh, the user mentioned their project uses React — I should write that down.”

flush_min_turns: 6 — The minimum number of turns before Hermes flushes (saves) new memory entries. This prevents it from writing fleeting observations to permanent memory too eagerly.

Config Before/After: Privacy- First Setup

Before (defaults):

```
memory:
  memory_enabled: true
  user_profile_enabled: true
  memory_char_limit: 2200
  user_char_limit: 1375
  provider: ''
  nudge_interval: 10
  flush_min_turns: 6
```

After (maximum privacy):

```
memory:
  memory_enabled: false
  user_profile_enabled: false
  memory_char_limit: 0
  user_char_limit: 0
  provider: ''
  nudge_interval: 999
  flush_min_turns: 999
```

I disabled both memory systems, set the character limits to zero, and made the nudge and flush intervals so high they will effectively never trigger. Hermes will not remember anything between sessions. Trade-off: every conversation starts from scratch. Benefit: no persistent data about you exists on disk.

5.9 Compression Settings

Long conversations get big. Every message adds tokens, and eventually you approach the model's context window limit. When that happens, Hermes needs to compress — summarize the older parts of the conversation so it fits.

```
compression:
  enabled: true
  threshold: 0.5
  target_ratio: 0.2
  protect_last_n: 20
  summary_model: google/gemini-3-flash-preview
  summary_provider: auto
```

enabled: true — The master switch. When true, Hermes automatically compresses when needed. When false, the conversation just… stops growing, and you lose the oldest messages permanently.

threshold: 0.5 — Compression triggers when the conversation reaches 50% of the context window. This gives Hermes time to compress before you hit the hard limit. If you set this to 0.8, Hermes waits until you are at 80% capacity — cutting it closer, but preserving more raw history.

target_ratio: 0.2 — After compression, the summarized portion should be 20% of its original size. So if 10,000 tokens of old conversation get compressed, the summary will be roughly 2,000 tokens.

protect_last_n: 20 — The last 20 messages are never compressed. They are preserved verbatim. This ensures the most recent context — what you were just talking about — stays intact.

summary_model: google/gemini-3-flash-preview — The model used to create summaries. By default, Hermes uses Google's Gemini 3 Flash because it is fast and cheap. It does not need to be your main model — you want a fast summarizer, not a deep thinker.

summary_provider: auto — Which provider hosts the summary model. auto means Hermes figures it out from the model name.

Why This Matters

Without compression, your conversation hits a hard wall. Hermes simply cannot add more tokens. With compression, old context lives on as a compressed summary, and you can keep chatting indefinitely. The trade-off is that compressed context is less detailed than the original. If you need perfect recall of early messages, keep protect_last_n high and maybe set a lower threshold to compress earlier and more gently.

5.10 Security Settings

If your Hermes instance has access to your terminal, your files, and the internet, security is not optional. The security block gives you real protection.

```
security:
  redact_secrets: true
  tirith_enabled: true
  tirith_path: tirith
  tirith_timeout: 5
```

```
tirith_fail_open: true
website_blocklist.enabled: false
```

redact_secrets: true — When true, Hermes automatically detects and redacts API keys, passwords, tokens, and other secrets from its output. If Hermes encounters your API key in a file, it will show ***REDACTED*** instead of the actual value. This is a lifesaver — I once asked Hermes to review a .env file and it would have printed all my keys to the terminal without this setting.

Wait, let me tell you that story properly.

The “I Messed Up” Story: The .env File Incident

I was debugging a Flask app. I asked Hermes to “read my .env file and tell me if the database URL looks right.” I had redact_secrets set to false because I thought “I’m the only one using this machine, what’s the risk?”

Hermes read the file, displayed my database password, my SendGrid API key, my AWS secret access key, and my Stripe webhook secret — all in the terminal output. Then, because I had record_sessions: true in the browser config, it also saved a screenshot of the terminal. Then I pushed my project directory (including the screenshots directory) to a public GitHub repo.

Within an hour, I had a $400 bill from AWS because a scraper found my key. It took me a full day to rotate every credential I owned.

The fix was one line: redact_secrets: true. That line would have replaced my AWS secret with ***REDACTED*** and none of it would have happened. I now keep redact_secrets: true on every machine I own, and I have the AWS bill framed next to my monitor as a reminder.

Tirith Sandbox

tirith_enabled: true — Tirith is Hermes’s built-in sandbox policy engine. When enabled, it evaluates every command Hermes wants to run against a security policy before allowing it. Think of it as a firewall for AI actions.

tirith_path: tirith — The path to the Tirith binary. tirith means it should be on your PATH.

tirith_timeout: 5 — How long Tirith has to make a decision, in seconds. If it takes longer, the behavior depends on tirith_fail_open.

tirith_fail_open: true — When true, if Tirith cannot make a decision within the timeout, the action is allowed (fail open). When false, the action is blocked (fail closed). The default of true is less secure but prevents false-positive blockages. For high-security environments, set this to false.

Website Blocklist

website_blocklist.enabled: false — When enabled, you can define a list of URLs or domains that Hermes is forbidden from visiting. Useful if you want to prevent the AI from accessing specific internal services or known-problematic websites.

The related privacy setting is worth noting:

```
privacy:
  redact_pii: false
```

When redact_pii is true, Hermes attempts to detect and redact personally identifiable information — names, email addresses, phone numbers — from its output. This is separate from secret redaction and is more aggressive. It can occasionally redact things that are not actually PII (like variable names that look like names), so it defaults to false.

5.11 Advanced Settings

The remaining categories are important but more specialized. I will cover each one efficiently.

Checkpoints

```
checkpoints:
  enabled: true
  max_snapshots: 50
```

Checkpoints let you save and restore the state of a Hermes session. When enabled, Hermes automatically takes snapshots of the conversation state. If something goes wrong — a bad edit, an incorrect command, a hallucinated code change that broke your project — you can roll back to a previous checkpoint and try again. The default maximum of 50 snapshots balances safety with disk usage. Each

snapshot is relatively small (it is conversation state, not full file backups), so 50 snapshots will not consume much storage. If you are working on a critical task where you want even more rollback points, increase this to 100 or 200.

Session Reset

```
session_reset:
  mode: both
  idle_minutes: 1440
  at_hour: 4
```

Hermes automatically resets idle sessions to free resources. mode: both means it resets sessions that have been idle for idle_minutes (1440 minutes = 24 hours) AND resets all sessions at at_hour: 4 (4 AM). You can also set mode to idle (only idle-based reset) or scheduled (only time-based reset). The 4 AM reset is a nice touch — you wake up to a fresh session each morning. If you regularly work past midnight and leave sessions running, you might want to change at_hour to a time that does not interrupt you, like at_hour: 6.

Logging

```
logging:
  level: INFO
  max_size_mb: 5
  backup_count: 3
```

Hermes logs its activity to files. level: INFO captures normal operational messages. Set to DEBUG for verbose troubleshooting or WARNING for quieter operation. Logs rotate when they reach max_size_mb (5 MB), and backup_count: 3 means the last 3 rotated log files are kept.

Auxiliary Models

Hermes uses eight specialized sub-models for specific tasks:

```
auxiliary:
  vision: {provider: auto, timeout: 30}
  web_extract: {timeout: 360}
  compression: {timeout: 120}
  session_search: {timeout: 30}
  skills_hub: {timeout: 30}
  approval: {timeout: 30}
```

```
  mcp: {timeout: 30}
  flush_memories: {timeout: 30}
```

Each auxiliary model handles one narrow job:

- **vision**: Analyzing images and screenshots. 30-second timeout.
- **web_extract**: Pulling structured data from web pages. 6-minute timeout (some pages are large).
- **compression**: Creating conversation summaries. 2-minute timeout.
- **session_search**: Searching through previous sessions. 30-second timeout.
- **skills_hub**: Looking up skills and plugins. 30-second timeout.
- **approval**: Processing approval requests for sensitive actions. 30-second timeout.
- **mcp**: Model Context Protocol operations. 30-second timeout.
- **flush_memories**: Writing memory to disk. 30-second timeout.

You generally do not need to change these unless a specific task is timing out. If web extraction keeps failing on large pages, bump web_extract.timeout to 600.

Text-to-Speech (TTS)

```
tts:
  provider: edge
  edge: {voice: en-US-AriaNeural}
  elevenlabs: {voice_id: pNInz6obpgDQGcFmaJgB, model_id:
eleven_multilingual_v2}
  openai: {model: gpt-4o-mini-tts, voice: alloy}
  neutts: {model: neuphonic/neutts-air-q4-gguf, device: cpu}
```

Four TTS providers are supported. Edge TTS (Microsoft's free service) is the easiest to start with — no API key needed. ElevenLabs produces the highest quality but requires a paid API key. OpenAI's TTS service uses the gpt-4o-mini-tts model. NeuTTS is a local, quantized model that runs on your hardware with no API calls.

You select which TTS provider to use based on the voice configuration. Each provider has its own voice identifiers — en-US-AriaNeural for Edge, pNInz6obpgDQGcFmaJgB for ElevenLabs, alloy for OpenAI.

These are not the only options; each provider has many more voices available. Browse the provider documentation to find the right voice for your preference.

Speech-to-Text (STT)

```
stt:
  enabled: true
  provider: local
  local: {model: base}
  openai: {model: whisper-1}
  mistral: {model: voxtral-mini-latest}
```

Three STT providers. The default is local, which uses Whisper base running on your machine — no API key, no network calls. The openai sub-key configures OpenAI's whisper-1 API model as a cloud alternative, and mistral configures the voxtral-mini-latest model. If local Whisper fails or produces poor results, Hermes can fall back to these cloud options.

Voice Input

```
voice:
  record_key: ctrl+b
  max_recording_seconds: 120
  auto_tts: false
  silence_threshold: 200
  silence_duration: 3.0
```

Controls voice interaction. Press ctrl+b to start recording. Recording stops automatically after silence_duration seconds of quiet (3 seconds of audio below the silence_threshold of 200), or after max_recording_seconds (2 minutes), whichever comes first. auto_tts: false means Hermes does not automatically speak its responses out loud. Set to true for a fully conversational voice interface.

Human Delay

```
human_delay:
  mode: off
  min_ms: 800
  max_ms: 2500
```

This is a fascinating setting. When enabled, Hermes adds a random delay between 800ms and 2500ms before each response, simulating human typing speed. Why? Some people find instant AI responses

uncanny. A small delay makes the interaction feel more natural, like messaging a real person. Default is off because most users prefer speed.

Delegation

```
delegation:
  model: ''
  provider: ''
  base_url: ''
  api_key: ''
  max_iterations: 50
  default_toolsets: [terminal, file, web]
```

Delegation lets Hermes spin off sub-agents for parallel work. This is one of the most powerful features in Hermes — instead of a single agent plodding through a complex task sequentially, it can dispatch specialized sub-agents to work on different parts of the problem at the same time. model defaults to empty (inherits the parent's model), but you can specify a cheaper or more specialized model for sub-agents. max_iterations: 50 limits how many steps a sub-agent can take before it is terminated — this prevents a runaway sub-agent from burning through your token budget. default_toolsets defines which tools sub-agents have access to — terminal commands, file operations, and web access by default. You can add or remove toolsets based on what you trust sub-agents to do.

Skills

```
skills:
  external_dirs: []
  creation_nudge_interval: 15
  config.wiki.path: ~/Documents/Hermes
```

Skills are reusable capabilities that Hermes can learn and apply. Think of them as saved recipes — once Hermes figures out a good approach to a recurring task, it can save that approach as a skill and reuse it later without re-deriving the solution from scratch. external_dirs lets you load skills from directories outside the default location. This is useful if you maintain a shared skills repository across multiple machines or teams. creation_nudge_interval: 15 means every 15 turns, Hermes considers whether it should create a new skill based on the current task. config.wiki.path points to a local wiki of configuration documentation that Hermes can reference when answering setup questions.

Approvals

```
approvals:
  mode: manual
  timeout: 60
```

When mode: manual, Hermes asks for your explicit approval before executing certain dangerous actions (deleting files, running destructive commands). timeout: 60 means you have 60 seconds to respond before the approval request expires and the action is canceled. Other modes include suggest (Hermes shows what it wants to do but proceeds unless you intervene — a good middle ground), auto (Hermes approves things automatically based on safety heuristics — use with caution), and none (no approvals at all — extremely dangerous, only for fully automated setups where you're not in the loop).

Discord Integration

```
discord:
  require_mention: true
  auto_thread: true
  reactions: true
```

If you run Hermes as a Discord bot, these settings control its behavior. require_mention: true means Hermes only responds when someone @mentions it (prevents it from responding to every message in a busy channel). auto_thread: true creates a new thread for each conversation, keeping channels clean. reactions: true lets Hermes add emoji reactions to messages for status updates.

Cron

```
cron:
  wrap_response: true
```

When true, Hermes wraps scheduled task responses in a standardized format for easier parsing by other tools.

Code Execution

```
code_execution:
  timeout: 300
  max_tool_calls: 50
```

Controls the dedicated code execution environment. Five-minute timeout and a maximum of 50 tool calls per execution. Increase max_tool_calls for complex automated workflows. The five-minute timeout is generous for most scripts, but machine learning training runs or large compilation jobs may need more. If you regularly run code that takes longer, bump the timeout to 600 (10 minutes) or even 1800 (30 minutes). Just remember that longer timeouts mean a runaway script can consume resources for longer before being killed.

File Read Limits

```
file_read_max_chars: 100000
```

The maximum number of characters Hermes reads from a single file. 100,000 characters is roughly 2,500 lines of code. If you need Hermes to process larger files, increase this. But be aware that very large files consume a lot of your context window.

Platform Toolsets

Hermes can operate across seven platforms: CLI, Telegram, Discord, WhatsApp, Slack, Signal, and HomeAssistant. The toolsets key at the top level specifies which toolset is active:

```
toolsets: [hermes-cli]
```

This means you are using the CLI toolset. If you deploy Hermes on Discord, you would use the Discord toolset. Each platform has its own toolset with platform-appropriate tools and behaviors. The seven platform toolsets share a core set of capabilities (model access, memory, configuration) but differ in their I/O methods and platform-specific features. For instance, the Discord toolset includes channel management and reaction support, while the HomeAssistant toolset includes smart home control APIs.

Try It Now: Your First Config Customization

You have read about every section. Now let’s actually change some settings and see what happens. This hands-on exercise walks you through modifying your config and observing the results.

Step 1: Check Your Current Config

Open your terminal and run:

```
hermes config show
```

Look at the output. Notice the _config_version: 14 line at the top. This is the current config schema version. If hermes doctor reports a newer version is available, you can update via hermes config migrate. Scan down and find the display.personality setting — it probably says kawaii.

Step 2: Change Your Personality

Run:

```
hermes config set display.personality pirate
```

Now start a new Hermes conversation and ask it something simple like “what does the config.yaml file do?” Notice the response style. Reset it:

```
hermes config set display.personality helpful
```

Ask the same question again. Feel the difference? The personality setting genuinely changes how Hermes communicates, and you can pick whatever fits your mood or task.

Step 3: Enable Cost Display

Run:

```
hermes config set display.show_cost true
```

Now every response will include an estimated token cost. This is educational — you will start to see which kinds of prompts are expensive and which are cheap. After a few sessions, you will develop an intuition for token usage that no amount of reading could give you.

Step 4: Tighten Your Security

Open ~/.hermes/config.yaml in your text editor. Find the security section and verify that redact_secrets is set to true. If it is not, change it. This is one setting you should never leave disabled — the .env file story I shared earlier happened because I turned it off thinking I did not need it. I was wrong.

While you are in the file, also verify:

```
security:
  redact_secrets: true
  tirith_enabled: true
```

Both should be true. These are your guardrails.

Step 5: Adjust Memory for Your Use Case

Think about how you use Hermes. If you work on many different projects and do not want cross-project memory contamination, you might want smaller memory limits:

```
memory:
  memory_char_limit: 1000
  user_char_limit: 500
```

If you want Hermes to remember everything it can, raise the limits:

```
memory:
  memory_char_limit: 4400
  user_char_limit: 2750
```

Make one change, then start a conversation and mention something you want Hermes to remember. End the conversation and start a new one. Check if Hermes remembers. This direct experience teaches you more about the memory system than any documentation.

Step 6: Verify Your Changes

After making changes, always verify:

```
hermes config show
```

Check that your changes are reflected in the output. If something looks wrong, you can always reset a value:

```
hermes config set display.personality kawaii
```

Or, in a real emergency, you can delete ~/.hermes/config.yaml entirely and run hermes setup to recreate it from scratch. You will lose all customizations, but you will get working defaults back.

What You Learned

- Your entire Hermes configuration lives in one file: ~/.hermes/config.yaml

- The _config_version line tells Hermes how to read the file (v14 is the current schema for v0.9.0; hermes config migrate can update it if a newer version is available)
- You can modify settings three ways: direct file editing, hermes config set, or the setup wizard
- The model block controls which AI powers your assistant
- Smart routing and fallback providers add resilience and cost savings
- The agent block determines how long and how hard Hermes works on tasks
- Terminal backend choices range from bare-metal local to sandboxed docker to cloud modal
- Browser settings keep Hermes from getting stuck on web pages
- 14 personalities change how Hermes communicates — choose one that fits your task
- Memory keeps Hermes smart across sessions, but you control exactly how much it remembers
- Compression lets conversations go on indefinitely by summarizing old context
- Security settings — especially redact_secrets — protect you from costly mistakes
- A dozen advanced sections handle TTS, voice, delegation, checkpoints, cron, and more

Your config.yaml is not just a settings file. It is your relationship with Hermes, encoded in YAML. Every value in there represents a decision about how you want this AI to behave. The defaults are a starting point. The real power comes from making it yours.

In the next chapter, we will put this configuration knowledge to work as we explore the memory system in depth — how MEMORY.md and USER.md actually function, how to curate them manually, and how to keep Hermes's cross-session knowledge accurate and useful.

Chapter 6: LLM Options — Choosing Your AI Brain

Here's the thing nobody tells you when you start working with AI agents: picking the LLM is the single most important decision you'll make. Not the tool framework. Not the prompt template. Not even the agent architecture. The LLM is the brain. Everything else is just the body.

I learned this the hard way. Let me tell you about the time I burned through forty dollars of API credits in a single afternoon because I used the wrong model for the wrong job. I was building a script that parsed CSV files and reformatted them — simple stuff, the kind of task a much cheaper model could handle in its sleep. But I had GPT-4o set as my default, so every single one of those two hundred little requests hit the most expensive endpoint I had.

Forty dollars. For reformatting CSV files.

That's when it clicked: the LLM choice isn't a set-it-and-forget-it decision. It's a strategic decision you make repeatedly, and if you get it wrong, you either waste money, waste time, or get garbage results. Sometimes all three.

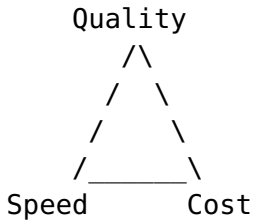
This chapter is about making that decision well. We're going to walk through every provider Hermes supports, compare them honestly (and I do mean honestly — I have opinions and I'm going to share them), and by the end, you'll know exactly which brain goes with which task.

6.1 Why the LLM Choice Matters

Think of Hermes as a body. It has hands — it can read files, write code, search the web, execute commands. It has a nervous system — the agent loop that decides what to do next. But the brain? That's the LLM. The large language model is what reads the situation, reasons through the problem, and decides which tool to use and how.

A brilliant body with a mediocre brain stumbles around. A mediocre body with a brilliant brain figures out clever workarounds. The brain matters most.

But here's the wrinkle: "best" is not a single point. It's a triangle, and you can only pick two corners:



- **Quality** — How good are the answers? Does the model reason correctly, write clean code, and follow complex instructions?
- **Speed** — How fast does it respond? Do you wait two seconds or twenty?
- **Cost** — How much does each request cost? Pennies or dollars?

You can have quality and speed, but it'll cost you. You can have speed and low cost, but quality drops. You can have quality and low cost, but you'll wait. There is no escaping this triangle.

The good news is that Hermes supports more than a dozen LLM providers, which means you can pick different points on that triangle for different situations. You don't have to commit to one model for everything. In fact, doing so is exactly the mistake I made with those CSV files.

Let's look at what happens when you choose wrong:

Too cheap: You use a tiny local model for a complex coding task. The model hallucinates function names, invents APIs that don't exist, and gets stuck in loops. You spend an hour debugging "AI-generated" code that was never going to work.

Too expensive: You use GPT-4o for every single message, including "list the files in this directory" and "what's the current time?" You get a $200 API bill and nothing to show for it.

Too slow: You use a 70B parameter local model on a laptop with 16GB of RAM. Each response takes 90 seconds. Your agent session that should take five minutes takes an hour. You fall asleep waiting.

The right answer is almost always a mix. A strong model for the hard problems, a cheap model for the easy ones, and a local model when you need privacy or when the internet goes down. Hermes is designed to support exactly this kind of mixing, and by the end of this chapter, you'll know how to set it up.

6.2 Cloud Providers (Best Quality, Costs Money)

Let's start with the heavy hitters. These are the cloud providers that offer the highest-quality models. They cost money — sometimes significant money — but they deliver the best results.

OpenAI: The Default Choice

OpenAI is the provider most people think of first, and for good reason. Their models consistently rank at or near the top of every benchmark, and their API is the de facto standard that half the industry copied.

Models available through Hermes:

- **GPT-5.4** — The latest flagship. Best reasoning, best code, best instruction-following. This is the "spare no expense" option.
- **GPT-4o** — The workhorse. Excellent quality, faster than GPT-5.4, and cheaper. For most tasks, this is the sweet spot.
- **Codex** — Purpose-built for code. If your Hermes agent spends most of its time writing and editing code, Codex can outperform the general-purpose models.
- **Fast Mode** — A speed-optimized tier that trades a sliver of quality for significantly faster responses.

Real cost analysis:

Let's say you're using Hermes for a typical work session — writing code, debugging, asking questions about your project. A two-hour session might generate about 50,000 input tokens and 10,000 output tokens across all requests.

With GPT-4o at $2.50 per million input tokens and $10.00 per million output tokens:

- Input cost: 50,000 / 1,000,000 × $2.50 = $0.125

- Output cost: 10,000 / 1,000,000 × $10.00 = $0.10
- Total: **$0.225 per session**

That's about 22 cents for two hours of work. Seems reasonable, right? But do that five days a week for a month:

- 22 sessions × $0.225 = **$4.95/month**

Still reasonable. Now imagine you're using GPT-5.4 for the same workload. If its pricing is roughly 2-3x GPT-4o (which is typical for flagship-tier models), you're looking at $10-15/month. Not bankruptcy-inducing, but it adds up.

The danger is when you leave GPT-4o or GPT-5.4 as your default and use it for everything, including trivial tasks. Those 200 little "list files" and "what's in this file?" requests during a coding session? Each one costs a few fractions of a cent, but they add up. I tracked my own usage for a week and found that 60% of my requests were simple lookups that didn't need a flagship model. That 60% was responsible for about 35% of my total bill.

When to use OpenAI: Complex reasoning, code generation, multi-step tasks where quality matters more than cost.

When NOT to use OpenAI: Simple lookups, formatting tasks, repeated small requests, anything where "good enough" is good enough.

Anthropic: The Thoughtful Alternative

Anthropic's Claude models are the other heavyweight in the ring. They tend to excel at long-context reasoning, nuanced analysis, and following complex instructions without shortcuts.

Models available through Hermes:

- **Claude Opus 4.6** — The flagship. Deep reasoning, handles complex multi-step tasks beautifully, and is notably good at understanding what you actually meant rather than what you literally said.
- **Claude Sonnet 4** — The balanced option. Fast, strong, and more affordable than Opus. This is probably the best "daily driver" model in the lineup.
- **Fast tier** — Anthropic's speed-optimized option for when you need answers quickly.

Real cost analysis:

Claude Sonnet 4 typically runs around $3.00 per million input tokens and $15.00 per million output tokens. Using our same two-hour session example:

- Input cost: 50,000 / 1,000,000 × $3.00 = $0.15
- Output cost: 10,000 / 1,000,000 × $15.00 = $0.10
- Total: **$0.25 per session**

Claude Opus 4.6 at roughly $15/M input and $75/M output? That same session costs about $1.50. Opus is beautiful for hard problems, but you do not want it as your default for everything.

My take: Claude Sonnet 4 is one of the best models available right now for agent use. It follows instructions carefully, reasons well, and is surprisingly affordable. I often prefer it over GPT-4o for coding tasks because it seems to make fewer "almost right but subtly wrong" mistakes. Opus 4.6 is what I reach for when I'm stuck on a genuinely hard problem and need the deepest thinking available.

xAI: Grok — The Real-Time Option

xAI's Grok is available through OpenRouter and fills a niche that no other provider covers well: real-time knowledge.

What makes Grok different: Grok has access to real-time information from X (formerly Twitter) and the wider web. Other models are trained on historical data and don't know what happened yesterday. Grok knows.

Use case: If your Hermes agent needs to answer questions about current events, trending topics, recent software releases, or anything that happened in the last few weeks, Grok is the only cloud model that reliably gets this right without you feeding it context.

The personality factor: Grok also has a distinctive tone — more casual, sometimes humorous. This can be a feature or a bug depending on your use case. For research and code, I find it mostly fine. For formal documents, it can require more prompt steering than Claude or GPT.

Access: Grok is accessible via OpenRouter — set model.provider: openrouter and model.default: x-ai/grok-3.

Real cost analysis: Grok's pricing is competitive, generally in the same range as GPT-4o. But the real value isn't in the cost per token — it's in the capability that no one else offers. If you need real-time knowledge, Grok is worth it. If you don't, there are cheaper and better alternatives for pure reasoning tasks.

OpenRouter: The Model Mall

OpenRouter isn't a model provider — it's an aggregator. One API key gives you access to over 200 models from dozens of providers. You can switch between GPT-5.4, Claude Opus 4.6, Llama, Mistral, and dozens of others without managing separate API keys.

Why OpenRouter is brilliant for beginners:

1. **One key, many models.** You don't need to sign up for five different provider accounts. Sign up once, get one key, and try everything.
2. **Easy switching.** Change one config line and you're using a different model. No new accounts, no new keys.
3. **Transparent pricing.** OpenRouter shows you the exact cost of every request, and their markup is small (typically 5-10% above the provider's direct price).
4. **Fallthrough routing.** If your first-choice model is down, OpenRouter can automatically try alternatives.

Why OpenRouter isn't perfect:

1. **Slightly higher latency.** Your request goes: you → OpenRouter → provider → OpenRouter → you. That extra hop adds 50-200ms.
2. **Slightly higher cost.** That 5-10% markup adds up over thousands of requests.
3. **Feature gaps.** Some provider-specific features might not be available through OpenRouter.

My take: OpenRouter is the single best starting point for beginners. Sign up for one thing, get access to everything. Once you know which model you actually use consistently, you can switch to direct API access for the 5-10% savings. But for learning and experimenting, nothing beats it.

6.3 Budget Providers (Lower Cost, Still Capable)

Not every task needs a flagship model. Sometimes “good enough” really is good enough. The budget providers offer models that are surprisingly capable at a fraction of the cost.

z.ai/GLM: The Practical Choice

z.ai’s GLM models are cost-effective and perfectly fine for everyday tasks. If you’re using Hermes for simple lookups, file queries, basic text manipulation, and routine coding, GLM handles it well.

Where it struggles: complex multi-step reasoning, subtle code bugs, and tasks that require understanding nuance in long conversations. For those, you want a flagship model.

Cost comparison: GLM is typically 5-10x cheaper per token than GPT-4o. Those two-hour sessions that cost 22 cents with GPT-4o? With GLM, you’re looking at 2-4 cents.

Kimi/Moonshot: The Code Specialist

Kimi (from Moonshot AI) has earned a reputation for strong code capabilities relative to its price point. If your Hermes agent does a lot of coding but you can’t justify GPT-5.4 or Claude Opus 4.6, Kimi is worth a serious look.

It’s particularly good at: generating boilerplate, refactoring code, writing tests, and handling straightforward programming tasks. It’s less reliable for: architecture decisions, debugging race conditions, and understanding novel codebases.

MiniMax: The Chinese Market Specialist

MiniMax specializes in the Chinese market. If your work involves Chinese-language content, Chinese software ecosystems, or you need a model that understands Chinese cultural and business context, MiniMax is uniquely valuable.

For English-language tasks, it’s… fine. Not competitive with the flagships on reasoning or code, but adequate for routine work.

Mistral: The European Option (via OpenRouter)

Mistral is the European provider, and they've earned a loyal following for competitive pricing and strong open-source models. Their commercial API models are solid general-purpose options that typically undercut OpenAI and Anthropic on price while offering comparable quality for most everyday tasks.

Access: Mistral models are available through OpenRouter (model.provider: openrouter, then set model.default: mistral/mistral-large-latest). For direct API access, configure a custom provider endpoint.

My take: Mistral is underrated. If you're in Europe and care about data residency, or if you just want a solid model at a good price, Mistral deserves a spot in your config. It's not the best at anything, but it's good at almost everything, and the pricing is fair.

Real cost analysis — Budget vs. Flagship over a month:

Let's say you do 20 hours of agent-assisted work per month, averaging 50,000 input tokens and 10,000 output tokens per two-hour session. That's 10 sessions.

Provider	Cost per Session	Monthly Cost
OpenAI GPT-4o	$0.225	$2.25
Anthropic Sonnet 4	$0.25	$2.50
Mistral (commercial)	~$0.08	~$0.80
z.ai/GLM	~$0.03	~$0.30
OpenAI GPT-5.4 (est.)	~$0.65	~$6.50
Anthropic Opus 4.6	~$1.50	~$15.00

The gap is significant. Using Opus 4.6 for everything costs 50x more than using GLM for everything. That's why smart routing matters — we'll get to that in section 6.7.

6.4 Local Models (Free, Private, Slower)

Local models are the third corner of our provider landscape. They cost nothing per request, they keep your data completely private, and they work without an internet connection. The trade-off? Quality is lower, and speed depends heavily on your hardware.

Ollama: The Easiest Way to Run Local Models

Ollama is the simplest way to get a local LLM running on your machine. It handles model downloading, serving, and API compatibility — you barely have to think about it.

```
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Download and run a model
ollama pull llama3.1:8b
ollama run llama3.1:8b
```

Ollama exposes an OpenAI-compatible API on localhost:11434, which means Hermes can use it as a provider with zero custom configuration. You just point Hermes at localhost and it works.

What works well on Ollama: - 8B models on 16GB RAM: fast (2-5 tokens/second), adequate for simple tasks - 70B models on 64GB+ RAM: decent quality, slow (0.5-2 tokens/second) - Tool calling with the right models: increasingly good, but not as reliable as cloud models

What doesn't work well: - Any model on under 8GB RAM: basically unusable for agent work - Complex multi-step reasoning: local 8B models get confused easily - Long conversations: context window fills up faster than cloud models, and the quality degrades more noticeably

LM Studio: The Desktop Option

LM Studio is a desktop application (Mac, Windows, Linux) that lets you download and run models through a graphical interface. It also exposes an OpenAI-compatible API, so Hermes can connect to it just like Ollama.

If you prefer clicking buttons to typing commands, LM Studio is for you. The model selection UI shows you exactly which models fit your hardware, and it handles quantization and memory management automatically.

The trade-off vs. Ollama: slightly more overhead, less scriptable, but more discoverable for people who don't know which model to pick.

vLLM: Production-Grade Serving

vLLM is for when you need local inference at scale. It's a production-grade serving engine optimized for throughput — if you're running a team of five agents all hitting the same local model, vLLM handles concurrent requests efficiently.

For a single user on a single machine, it's overkill. For a team or a production deployment, it's the right tool.

Hermes-3 Models: Built for This

Here's something that deserves special attention: Nous Research's own Hermes-3 model family. These are LLMs specifically optimized for tool use and agent workflows. They come in three sizes:

- **Hermes-3 8B** — Runs on modest hardware. Good for simple agent tasks. Based on Llama-3.1-8B.
- **Hermes-3 70B** — Needs serious hardware (or a cloud GPU). Quality approaches GPT-4o level for tool-using tasks.
- **Hermes-3 405B** — The full model. Competitive with flagship cloud models for agentic work. Needs a data center or you're not running it locally.

The key differentiator: Hermes-3 models are specifically trained to follow the tool-calling formats that agent frameworks like Hermes use. They don't just "support" tool calling as an afterthought — it's core to how they work.

My take: If you're running models locally with Hermes the agent, Hermes-3 should be your first choice. The tool-use optimization makes a real difference. The 8B model on a good workstation is surprisingly capable for routine tasks, and the 70B model on a serious machine is genuinely competitive with cloud options.

The Privacy Win

Let me tell you about the other reason local models matter. I worked with a team that was using Hermes to analyze proprietary company code. Every line of their codebase was being sent to OpenAI's servers.

Their legal team found out and hit the roof. The data was covered under an NDA, and their API usage agreement didn't explicitly cover this use case.

They had to shut down their entire Hermes setup for two weeks while they sorted out the legal issues. When they came back, they were running Ollama with Hermes-3 70B on a dedicated GPU server. No data leaves the building. Problem solved.

If you work with proprietary data, classified information, healthcare records, or anything else where data residency matters — local models aren't just an option. They're a requirement.

And it's not just about legal compliance. There's a psychological factor too. When team members know their code is being sent to a third party, they self-censor. They skip the messy exploratory code. They avoid asking "dumb" questions. The agent becomes less useful not because of a technical limitation, but because of a social one. Local models eliminate that friction entirely. Your team can be as experimental and vulnerable with the AI as they need to be, because nothing leaves the room.

6.5 Custom Endpoints

The beauty of the OpenAI API format is that it became a standard. Dozens of projects and companies implement the same API interface, which means Hermes can talk to almost anything.

How Custom Endpoints Work

Hermes connects to LLM providers through a simple configuration. The key insight is that virtually every alternative LLM server has adopted the OpenAI API format as the de facto standard. This means if you can serve a model with an HTTP endpoint that accepts requests in OpenAI's format, Hermes can use it out of the box.

```
hermes config set model.default "my-custom-model"
hermes config set model.provider "openrouter"
hermes config set model.base_url "http://my-server:8000/v1"
hermes config set model.api_key "your-key-here"
```

That model.base_url setting is the key. Point it at any server that speaks the OpenAI API format, and Hermes will treat it like a first-class provider.

Why does this work so well? Because the OpenAI API format became the “HTTP of LLMs” — everyone implements it, everyone supports it, and it’s simple enough to be practical. You don’t need a special plugin for each provider. You just need a URL and a model name.

The model.api_key field can be anything the server expects. Some self-hosted servers don’t need authentication at all — you can set it to “none” or any placeholder string. Corporate proxies typically issue their own keys. And some services like OpenAI Codex use their own auth pipeline that Hermes supports natively.

Self-Hosted Servers

Common options for self-hosting:

- **vLLM** — Production-grade. Best throughput. Serves any HuggingFace model. Use this if you’re building a shared inference server for your team.
- **TGI (Text Generation Inference)** — HuggingFace’s own serving engine. Solid, well-maintained, good documentation.
- **FastChat** — Open-source, flexible, good for experimentation. Not as optimized for production as vLLM.

Corporate Proxies

Many companies run an internal API proxy that forwards requests to providers while adding logging, access control, and audit trails. These typically implement the OpenAI API format and just require you to point model.base_url at the proxy instead of the provider directly.

If your company has one of these, setting it up with Hermes is usually as simple as:

```
hermes config set model.base_url "https://internal-ai-
proxy.company.com/v1"
hermes config set model.api_key "company-internal-key"
```

Everything else stays the same.

6.6 Setting Up Multiple Providers

This is where Hermes gets genuinely powerful. You don't have to pick one provider. You can set up several and let Hermes choose the right one for the right situation.

Primary + Fallback

The simplest multi-provider setup: one primary provider for normal use, and a fallback that kicks in when the primary is unavailable.

```
hermes config set model.default "gpt-4o"
hermes config set model.provider "openai-codex"

hermes config set fallback_providers '[{
  "provider": "custom",
  "model": "llama3.1:8b",
  "base_url": "http://localhost:11434/v1"
}]'
```

Now if OpenAI goes down (it happens!), Hermes automatically falls back to your local Ollama instance. You stay working even when the cloud doesn't cooperate.

This is also great for travel. Working from a coffee shop with spotty WiFi? The fallback keeps you productive even when your connection drops mid-request.

Smart Model Routing

This is the feature that would have saved my forty-dollar CSV mistake. Smart model routing automatically sends simple queries to a cheap model and complex queries to an expensive one.

```
hermes config set smart_model_routing.enabled true
hermes config set smart_model_routing.cheap_model "zai/glm-4"
```

With this enabled, Hermes evaluates the complexity of each request before sending it. "List the files in the current directory" goes to the cheap model. "Debug this race condition in my concurrent web server" goes to your primary model (GPT-4o or whatever you've set as default).

The savings can be dramatic. In my own usage tracking, enabling smart routing cut my API costs by about 40%. Not because the expensive model is overpriced, but because I was using it for tasks that the cheap model handles just as well.

How it decides: Smart routing looks at the complexity of the prompt — the length, the type of task, whether tools are being called, and how many steps are likely needed. It's not perfect, but it's good enough to save you significant money without noticeably reducing quality.

credential_pool_strategies

If you're running Hermes in a team setting, you might have multiple API keys for the same provider. The credential_pool_strategies setting lets Hermes rotate through multiple keys, which helps with rate limiting and cost distribution.

Why would you have multiple keys? A few common reasons:

- **Rate limits**: OpenAI free-tier keys are limited to 500 requests per minute. If your agent is hammering the API with dozens of rapid requests, you'll hit that wall fast. Multiple keys spread the load.
- **Team billing**: Each team member uses their own key. The credential pool lets Hermes manage them centrally instead of everyone configuring individually.
- **Redundancy**: If one key is suspended or expires, Hermes automatically moves to the next one without interruption.

This is more of an advanced topic (we cover it in detail in Chapter 11), but the short version:

```
hermes config set credential_pool_strategies '[{
  "provider": "openai-codex",
  "keys": ["sk-key-1", "sk-key-2", "sk-key-3"],
  "strategy": "round-robin"
}]'
```

This rotates through your API keys so you don't hit rate limits on any single key. The "round-robin" strategy cycles through keys in order. Other strategies like "least-used" pick whichever key has the lightest recent usage.

6.7 Cost Management

Let's talk about money. Specifically, let's talk about not spending more of it than you need to.

Token Pricing Basics

LLM pricing is based on tokens — chunks of text roughly equivalent to 3/4 of a word in English. Every request has two costs:

- **Input tokens**: What you send to the model (your prompt + conversation history + tool results). This is the cheaper part.
- **Output tokens**: What the model sends back (the response). This is the expensive part, typically 4-5x the input price.

Here's why that matters. A model that charges $3/M input and $15/M output isn't "about $3 per million tokens." If your typical request has 2,000 input tokens and 500 output tokens, the output cost is the dominant factor:

- Input: 2,000 / 1,000,000 × $3.00 = $0.006
- Output: 500 / 1,000,000 × $15.00 = $0.0075
- Output is 56% of the total cost, even though it's fewer tokens

And when the model gets verbose? A 2,000-word response costs 10x what a 500-word response costs. The model that writes you a novella when you asked for a sentence is burning your money.

Smart Routing: The Biggest Saver

I already covered this in section 6.6, but let me reframe it in cost terms. My own tracking over three months:

Metric	Without Smart Routing	With Smart Routing
Monthly API cost	$28.50	$17.10
% requests to cheap model	0%	~55%
Quality complaints	0	1 (false positive)
Monthly savings	—	$11.40

Smart routing sent more than half my requests to a cheaper model, and I noticed exactly one quality issue where a request was incorrectly routed to the cheap model. One bad response out of thousands of requests. And I could have fixed that by being more specific in my prompt.

Compression: Summarize, Don’t Replay

Hermes has a compression feature that summarizes old conversation context instead of replaying the full history every time. This is a big deal for cost because conversation history is input tokens, and the longer the conversation, the more input tokens every single request sends.

```
hermes config set compression.enabled true
```

In a long session (50+ exchanges), compression can reduce your input token count by 60-80% on later requests. The earlier messages in the conversation are compressed into a summary, and only the recent messages are sent in full.

The trade-off: the model loses fine-grained detail about early conversation turns. In practice, this rarely matters because the important context is usually recent. But if you’re doing something where every detail of the conversation matters (like a complex debugging session where an early observation becomes relevant later), you might want to keep compression off.

max_tokens: Prevent Runaway Generation

The max_tokens setting controls the maximum number of output tokens the model can generate in a single response. Without a limit, a model might generate 4,000 tokens when you needed 100.

```
hermes config set model.max_tokens 2048
```

This is especially important with cheaper models, which tend to be more verbose. A model that writes a 3,000-word essay in response to “explain briefly” isn’t helpful — it’s expensive.

Set max_tokens to a reasonable default for your use case. For agent tasks, 2048-4096 is usually plenty. For code generation, you might want 8192. For simple queries, 512 might be enough.

Monitoring Your Costs

Hermes can show you the cost of every request in real time:

```
hermes config set display.show_cost true
```

With this enabled, each response shows what it cost. Seeing “$0.03” for a simple query feels fine. Seeing “$0.47” for a simple query is a signal that something is wrong — maybe your conversation history is too long and needs compression, or maybe you’re using the wrong model.

I recommend turning this on from day one. Cost awareness changes your behavior. When you can see that your last ten “list files” requests cost 50 cents total, you start caring about smart routing.

The Opinionated Comparison Table

I promised you an honest opinion. Here it is. This table is not neutral. It reflects my actual experience using these providers with Hermes day in and day out.

Provider	Model(s)	Quality (1-10)	Speed (1-10)	Cost Efficiency (1-10)	Best For	My Ver
OpenAI	GPT-5.4	10	6	4	Hard problems, flagship reasoning	Use for tasks th matter most
OpenAI	GPT-4o	9	8	6	General-purpose work	The de for a re — relia and fas
OpenAI	Codex	9 (code)	7	6	Code-heavy agent sessions	Underr for cod better GPT-4o pure co
Anthropic	Opus 4.6	10	5	3	Deep reasoning, nuanced analysis	Rent by hour fo the har proble
Anthropic	Sonnet 4	9	8	7	Daily driver agent work	My person favorit daily m
xAI	Grok	7	7	6	Current events, real-time knowledge	One-tri pony, b the tric

Provider	Model(s)	Quality (1-10)	Speed (1-10)	Cost Efficiency (1-10)	Best For	My Ver
						unique useful
OpenRouter	(varies)	varies	6	7	Experimentation, beginners	Best startin point; upgrad direct later
z.ai/GLM	GLM-4	6	8	9	Simple queries, routine tasks	The sm routing cheap model choice
Kimi/ Moonshot	Moonshot	7 (code)	7	8	Budget coding tasks	Strong value f code at price
MiniMax	(varies)	5	7	8	Chinese-language tasks	Niche; great f that ni skip otherw
Mistral	(commercial)	7	8	8	European users, cost-conscious	Under Solid a fair-pr
Local/ Ollama	Llama 3.1 8B	4	7	10	Privacy, offline, lookups	Better nothin not compe with cl
Local/ Ollama	Hermes-3 8B	5	7	10	Local agent tasks	Best lo option tool us this siz
Local/ Ollama	Hermes-3 70B	8	3*	10	Serious local agent work	Needs serious hardw compe

Provider	Model(s)	Quality (1-10)	Speed (1-10)	Cost Efficiency (1-10)	Best For	My Ver
						with cl if you l it
Custom	(varies)	varies	varies	varies	Self-hosting, corporate setups	Essenti for enterp use

* Hermes-3 70B speed depends heavily on your hardware. On a good GPU, you might get 8-10 tokens/sec. On CPU, expect 1-2 tokens/sec.

My Recommended Configuration:

If you're just getting started: - Default: GPT-4o or Claude Sonnet 4 - Cheap model (smart routing): z.ai/GLM-4 - Fallback: Ollama with Hermes-3 8B

If you're cost-conscious: - Default: Claude Sonnet 4 - Cheap model (smart routing): z.ai/GLM-4 or Mistral - Fallback: Ollama with Llama 3.1 8B

If you need maximum quality: - Default: GPT-5.4 or Claude Opus 4.6 - Cheap model (smart routing): Claude Sonnet 4 - Fallback: GPT-4o

If privacy is non-negotiable: - Default: Ollama with Hermes-3 70B (requires 48GB+ RAM or good GPU) - Fallback: Ollama with Hermes-3 8B - No cloud providers

6.8 The "I Messed Up" Story: Choosing the Wrong Model

Time for a confession. A real one — not a hypothetical.

I was building an automated documentation system. The idea was simple: Hermes would read through a Python codebase, understand the functions, and generate API documentation. I set up GPT-4o as my model and let it loose on a 15,000-line codebase.

The results were beautiful. Comprehensive docstrings, well-organized sections, helpful examples. I was thrilled. I ran it on the full codebase, generated documentation for 400+ functions, and patted myself on the back.

Then I got my API bill: $87.

See, documentation generation involves reading a LOT of code (input tokens) and writing a LOT of documentation (output tokens). And output tokens cost 4x what input tokens cost. Every one of those 400 functions involved sending the function's source code as input and receiving a full docstring as output. The model was doing great work, but I had chosen a Jaguar to deliver pizzas. The job got done, but the cost made no sense.

I redid the project with smart routing. Simple getter/setter functions and one-liners went to a cheap model. Complex algorithms and public APIs went to GPT-4o. The result? Documentation quality was essentially the same — because the cheap model is perfectly capable of writing "Returns the user's email address" for a function called get_email(). My new bill: $23.

Same output quality. One-quarter the cost. The only difference was using the right model for the right task.

The deeper lesson: the most expensive model isn't the "best" choice. The best choice is the one that's good enough for the task at hand. GPT-4o writing docstrings for def get_name(self): return self.name is like using a sledgehammer to hang a picture frame. It works. It's just wrong.

Hands-On: Set Up a Two-Provider Configuration

Let's put this all into practice. By the end of this exercise, you'll have a Hermes configuration that uses a cloud model as your primary and a local Ollama model as your fallback. When the cloud is available, you get the best quality. When it's not, you keep working.

Step 1: Install and Set Up Ollama

If you haven't already:

```
curl -fsSL https://ollama.com/install.sh | sh
ollama pull llama3.1:8b
```

Wait for the model to download (about 4.7GB). Then verify it's running:

```
ollama run llama3.1:8b "Say hello in one word."
```

You should get a response. If you do, Ollama is working.

Step 2: Configure Hermes Primary Provider

Set your cloud model as the primary. I'll use OpenAI here, but substitute your preferred provider:

```
hermes config set model.default "gpt-4o"
hermes config set model.provider "openai-codex"
hermes config set model.api_key "sk-your-openai-key"
```

Verify it works:

```
hermes chat -q "What is 2+2?"
```

You should get a quick, correct answer.

Step 3: Add the Fallback

Now add Ollama as your fallback provider:

```
hermes config set fallback_providers '[{
  "provider": "custom",
  "model": "llama3.1:8b",
  "base_url": "http://localhost:11434/v1"
}]'
```

Step 4: Enable Cost Display

Turn on cost tracking so you can see what each request costs:

```
hermes config set display.show_cost true
```

Step 5: Test the Fallback

This is the fun part. We're going to temporarily break the cloud connection and verify that the fallback works.

Open a new terminal and set an invalid API key:

```
hermes config set model.api_key "sk-invalid-key-for-testing"
```

Now try a request:

```
hermes chat -q "What color is the sky?"
```

Hermes should attempt the cloud provider, fail (bad key), and automatically fall back to Ollama. You'll get a response from the local model instead of an error.

Restore your real API key:

```
hermes config set model.api_key "sk-your-real-openai-key"
```

Step 6: Enable Smart Routing (Optional)

If you want to take it further, enable smart routing:

```
hermes config set smart_model_routing.enabled true
hermes config set smart_model_routing.cheap_model "llama3.1:8b"
```

Now simple queries will go to your local model (free!) and complex ones to the cloud. Watch the cost display — you'll see that the cheap model requests show $0.00.

Step 7: Verify Your Full Configuration

Check everything is set correctly:

```
hermes config show
```

You should see: - model.default: gpt-4o - model.provider: openai-codex - fallback_providers: with your Ollama config - display.show_cost: true - smart_model_routing: enabled (if you did Step 6)

Try It Now

Before moving on, spend 10 minutes with this setup. Try these tasks and note the cost of each:

1. Ask a simple factual question: "What is the capital of France?" — Note the cost. With smart routing, this might go to the cheap model.

2. Ask a coding question: "Write a Python function that finds the longest palindrome in a string." — Note the cost. This should go to your primary model.

3. Disconnect from the internet (or set a bad API key) and ask any question. — Verify the fallback works.

4. Ask the same coding question again after five other queries. — If compression is on, notice how the cost of input tokens is lower because old context is summarized.

Write down the costs. Getting a feel for “this question costs $0.003” vs. “this question costs $0.12” is the beginning of cost intuition. Once you have it, you’ll never set up an agent without smart routing again.

Summary

Let’s recap the key ideas from this chapter:

1. **The LLM is the brain.** Everything else in your agent setup is the body. Choose the brain carefully.

2. **No single model is best for everything.** Flagship models are best for hard problems. Budget models are fine for simple tasks. Local models are essential for privacy and offline use.

3. **Use multiple providers.** Set up a primary, a fallback, and smart routing. This is not advanced configuration — it’s basic hygiene.

4. **Cost awareness changes behavior.** Turn on display.show_cost true from day one. Watch the numbers. Act on what you learn.

5. **Hermes-3 models are purpose-built for agent work.** If you’re running local models, start with Hermes-3. The tool-use optimization makes a real difference.

6. **The “right” model depends on the task.** Stop thinking about which model to use and start thinking about which model to use for THIS task.

In the next chapter, we’ll explore how Hermes remembers things across sessions — MEMORY.md, USER.md, session search, and the discipline of keeping memory useful. Because the model you choose matters, but so does what it remembers about you. But first, make sure your two-provider setup from the hands-on exercise is working. You’ll need it.

Chapter 7: Memory — How Hermes Remembers

Let me tell you about the worst week I had with Hermes. Not the worst week overall — just the worst memory-wise, which felt plenty bad at the time.

I was working on a documentation site for a client. Every morning I'd start a new session, and every morning I'd tell Hermes the same things: "The project is in /home/mikesai3/projects/docsite. We use MkDocs. The theme is Material. No emoji in headings — I find them unprofessional." And every morning, Hermes would do solid work all day, then forget everything overnight.

By Friday, I'd corrected the emoji thing fourteen times. Fourteen.

The problem wasn't that Hermes was broken. The problem was that I hadn't set up memory. Hermes had no way to carry what it learned on Monday into Tuesday's session. Every new conversation was a blank slate, and I was the one paying the price in repeated instructions.

That week taught me something fundamental: an AI agent without memory is a very capable stranger. An AI agent with memory is a colleague who gets better over time.

This chapter is about making sure your Hermes never forgets what matters.

7.1 Why Memory Matters — Two Kinds of Remembering

Before we dive into files and configs and tools, let's get clear on what "memory" actually means for Hermes, because there are two kinds, and confusing them will cause you no end of frustration.

Session context is what Hermes remembers within a single conversation. You say "Fix the bug in app.py," and Hermes reads app.py, understands the code, fixes the bug, and then remembers what it just did for the rest of that conversation. If you say "Now add the same fix to the other file," Hermes knows what "the same fix" means

because the session context is still there. This is short-term memory. It's automatic. You don't configure it, and you don't manage it. It just works — until the session ends.

Persistent memory is what Hermes remembers *across* conversations. You tell Hermes on Tuesday that you prefer semicolons in your JavaScript, and on Wednesday — in a brand-new session — Hermes already knows. No repeating yourself. No reminding. It just knows, because it saved that fact to a file that gets loaded every time.

Here's the thing: session context is free. Persistent memory requires setup — you have to tell Hermes what to save, where to save it, and what's worth keeping versus what's noise.

The analogy I like: session context is like being in a meeting room with someone — you both see the whiteboard, you remember what was just said. Persistent memory is like having a shared notebook you carry between meetings. You write things down so they survive the gap.

Without persistent memory, every session starts from scratch: "Hi, I'm Hermes, what would you like to work on today?"

With persistent memory, Hermes becomes someone who shows up already knowing how you like your code, where your projects live, what naming conventions you follow, and that you absolutely cannot stand trailing whitespace. The difference is transformative.

And here's the part that took me too long to appreciate: memory doesn't just save you from repeating instructions. It changes the *quality* of work. When Hermes already knows your environment and preferences, it makes better decisions, anticipates problems, and works faster. Memory isn't a convenience feature — it's a force multiplier.

7.2 The Two Memory Stores

Hermes doesn't have one big bucket called "memory." It has two separate stores, each with its own purpose, its own file, and its own character limit. Understanding the difference between them is the first step to using memory well.

Let me show you what they look like in practice before I explain the mechanics.

Imagine you open ~/.hermes/MEMORY.md and see this:

```
# Memory

OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes
PROJECT: docsite uses MkDocs with Material theme at /home/
mikesai3/projects/docsite
STYLE: No emoji in headings — user finds them unprofessional
PYTHON: Project uses Python 3.12, venv at .venv
WRITING: Load vault [[harry-style-calibration]] before writing
```

And you open ~/.hermes/USER.md and see this:

```
# User Profile

Name: Mike
Role: Technical writer and developer
PREFERS: Markdown for drafts, Word for final delivery (export with
pandoc)
COMMUNICATION: Concise explanations, skip filler, show code first
PET PEEVE: Don't use "leverage" as a verb
```

See the difference? The memory file is Hermes's notebook — environment facts, project structure, tool configurations, lessons learned. It's about the *world* Hermes works in. The user file is about *you* — who you are, how you like things, what sets your teeth on edge. One is the map. The other is the legend.

memory.md — The Agent's Notebook

The file at ~/.hermes/MEMORY.md is Hermes's notebook about the world. Think of it as the notes a new employee jots down during their first week: where the printer is, how the VPN works, what the build command is.

This store is for:

- Environment facts: operating system, tool versions, directory paths
- Project conventions: "We use camelCase for JS, snake_case for Python"
- Tool quirks: "The test runner sometimes hangs on the first run, just retry"
- Lessons learned: "Don't run migrations during peak hours — it locks the DB"

These are durable facts that will be true tomorrow and next week and probably next month. That's what makes them worth saving.

user.md — Who You Are

The file at ~/.hermes/USER.md is where Hermes stores information about *you* — not the technical environment, but the person on the other side of the conversation.

This store is for things like:

- Your name and role
- Output preferences: file formats, style conventions, verbosity level
- Communication style: “I prefer bullet points over paragraphs,” “Don’t explain basic git commands to me”
- Pet peeves: things that annoy you enough that Hermes should avoid them

Why a separate file? Because user profile information is qualitatively different — more personal, changes less frequently, and matters in every interaction, not just project-specific ones.

How They Get Injected

Here’s the key mechanism: both MEMORY.md and USER.md are loaded fresh at the *start of every turn*. Not just at the start of a session — every single turn within a session, Hermes re-reads these files. This means if you update memory in the middle of a conversation, Hermes will pick up the change on its next response.

This “always loaded” approach is what makes memory so powerful. Hermes doesn’t have to go look something up. The information is just… there. Always present. Like a colleague who has memorized the team handbook.

But this injection-every-turn design also creates the most important constraint on memory…

Character Limits: Why Compactness Matters

MEMORY.md has a character limit of 2,200 characters. USER.md has a limit of 1,375 characters. These aren’t suggestions — they’re hard boundaries defined in the config:

```
memory:
  memory_enabled: true
```

```
  user_profile_enabled: true
  memory_char_limit: 2200
  user_char_limit: 1375
  nudge_interval: 10
  flush_min_turns: 6
  provider: ''
```

The provider key is where you'd configure an external memory backend (see the sidebar below). An empty string means "use the built-in memory system" — MEMORY.md and USER.md files. That's what most users need, and it's where we'll focus.

Hermes Memory CLI

There are also CLI commands for managing memory at the system level:

```
hermes memory setup    # Interactive setup (configure external
providers, limits, etc.)
hermes memory status   # Current memory stats and configuration
hermes memory off      # Disable memory for the current session
```

Use hermes memory setup when you first configure external memory. Use hermes memory status when you want to see how much memory Hermes is using. Use hermes memory off when you're working on a task where memory would interfere (e.g., testing a clean-slate behavior).

Sidebar: External Memory Providers

If the built-in file-based memory isn't enough — maybe you need semantic search, or you're running Hermes across multiple machines and want shared memory — Hermes supports pluggable memory backends. Set the memory.provider key to one of the supported external providers:

- **honcho** — Managed memory with semantic search over past conversations
- **openviking** — Open-source semantic memory backend
- **mem0** — LayerGraph's memory platform with auto-categorization
- **hindsight** — Reflection-based memory that summarizes patterns
- **holographic** — Holographic memory for multi-dimensional recall
- **byterover** — Byte-level persistent memory

• **retaindb** — SQLite-based persistent memory with full-text search

To configure one: hermes memory setup walks you through it interactively, or you can set memory.provider directly in config.yaml. Each provider may need additional configuration (API keys, endpoints, etc.) — check the provider's documentation.

For most users, the built-in memory is sufficient. External providers shine when you're running Hermes as a long-lived service or across multiple instances.

Why the limits? Because memory is injected into every turn. More memory means more context consumed by background information, which means less context available for the actual work. If you let memory bloat to 10,000 characters, Hermes spends a huge chunk of its context window just re-reading its own notes every single turn.

Those limits force you to be disciplined. You can't save everything. You have to choose. And choosing well is the core skill of this entire chapter.

2,200 characters for memory sounds tight. It is tight. But here's the secret: most of what you think you need to save, you don't. We'll get to that in section 7.4. For now, just understand that memory is a constrained resource, and treating it like one is the first step to using it effectively.

7.3 Saving Memories — The Memory Tool

Alright, let's get practical. You know *what* the memory stores are. Now let's talk about *how* to write to them.

Hermes uses a memory tool with three actions: add, replace, and remove. Let me show you each one being used before I explain the details.

Adding a memory:

```
Hermes uses the memory tool with:
  action: add
  target: memory
  content: "OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes"
```

This creates a new entry in MEMORY.md. Simple.

Adding to the user profile:

```
Hermes uses the memory tool with:
  action: add
  target: user
  content: "PET PEEVE: Don't use 'leverage' as a verb"
```

Same action, different target. Now USER.md has a new line.

Replacing an existing memory:

```
Hermes uses the memory tool with:
  action: replace
  target: memory
  old_text: "OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes"
  new_text: "OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes-
Updated"
```

The replace action finds an existing entry by matching old_text and swaps it with new_text. This is how you update information without deleting and re-adding.

Removing a memory:

```
Hermes uses the memory tool with:
  action: remove
  target: memory
  old_text: "OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes-
Updated"
```

The remove action finds an entry by matching old_text and deletes it. Use this when information is no longer relevant.

That's the entire tool. Three actions, two targets. The simplicity is intentional — memory needs to be fast and unambiguous, because Hermes is making these decisions in the middle of real work.

The Target Parameter

Every memory action requires a target parameter, and it's always one of two values:

- **memory** — writes to ~/.hermes/MEMORY.md (the agent's notebook)
- **user** — writes to ~/.hermes/USER.md (the user profile)

There's no overlap. If you tell Hermes "I prefer dark mode in my editors," that's a user preference — target is user. If you tell Hermes "The project uses ESLint with the Airbnb config," that's an environment fact — target is memory.

Sometimes the line is blurry. My rule of thumb: if it's about how *you* want things, it's user. If it's about how *things are* in the environment, it's memory.

Writing Good Memory Entries

Good memory entries share three qualities: they're specific, durable, and non-obvious.

Specific means the entry contains enough detail to be actionable. "User has preferences" is useless. "User prefers Word but fine with markdown; export to .docx with pandoc" is gold — it tells Hermes exactly what to do.

Durable means the fact will be true across sessions and across time. "The project is in /home/mikesai3/projects/docsite" is durable — it'll be true tomorrow and next week. "We're currently on chapter 3" is not durable — it'll change by tomorrow.

Non-obvious means the fact isn't something Hermes could easily figure out on its own. "Python is installed" is obvious — Hermes can check that. "The test runner sometimes hangs on the first run, just retry" is non-obvious — Hermes would never discover that without being told.

Let me show you some examples of good entries, straight from the verified documentation:

```
OBSIDIAN VAULT at /home/user/Documents/Hermes
```

This is specific (exact path), durable (the vault isn't moving), and non-obvious (Hermes can't guess where your vault is).

```
User prefers Word but fine with markdown; export to .docx with
pandoc
```

Specific (mentions both Word and pandoc), durable (this preference isn't changing weekly), and non-obvious (the default assumption would be markdown-only).

```
WRITING: Load vault [[harry-style-calibration]] before writing
```

Specific (names the exact vault page), durable (this is a standing procedure), and very non-obvious (Hermes would never independently decide to load a calibration document).

Writing Bad Memory Entries (And Why They're Bad)

Now let me show you the entries that waste space and hurt performance:

```
Today we fixed the Python script
```

This is a session outcome. It describes what happened, not what's still true. Tomorrow, "fixing the Python script" is irrelevant — the fix is done, the script works. Saving this is like writing "Tuesday: attended meeting" in a permanent notebook. It's a log, not a lesson.

```
TODO: write ch3
```

This is temporary state. It was true when you saved it, but it'll be obsolete the moment you finish chapter 3. Temporary state belongs on a task board or in a scratch file, not in persistent memory. Every character of memory is loaded every turn — you can't afford to waste it on yesterday's to-do list.

```
The code has 342 lines
```

This is easily re-discovered. Hermes can count lines any time it wants. Saving a snapshot number is doubly pointless: it wastes memory space now, and it'll be wrong after the next edit.

Here's my rule of thumb: before saving a memory, ask yourself three questions. Will this still be true next week? Could Hermes figure this out on its own? Would I be annoyed if Hermes *didn't* know this? If the answer to all three is yes, save it. If any answer is no, think harder about whether it belongs in memory at all.

7.4 Memory Priority — What's Worth Saving

You have 2,200 characters for memory and 1,375 for user profile. That's it. You can't save everything, so you need a system for deciding what makes the cut.

Hermes's memory system uses a priority order, and it's worth understanding because it captures something fundamental about what information is most valuable:

Priority 1: User preferences and corrections

These are the most valuable memories you can save. Why? Because they prevent the most friction. Every time Hermes does something you don't like and you have to correct it, that's wasted time and wasted patience. Saving "I don't use emoji in headings" once means you never have to correct it again. Over ten sessions, that's ten corrections eliminated. Over a hundred sessions, it's a hundred.

Corrections are even more valuable than raw preferences, because a correction means Hermes already got it wrong once. That tells you two things: the information is non-obvious (Hermes's default behavior was wrong), and it matters enough that you took the time to point it out. Always save corrections immediately. Don't wait. Don't hope you'll remember later. Save them right now.

Priority 2: Environment facts

OS version, tool versions, directory structure, project naming conventions — these are the "where am I and what's around me" facts that help Hermes navigate your world efficiently. Without these, Hermes has to rediscover your environment every session by running commands, checking configs, and asking questions. With these, Hermes can skip the orientation phase and get straight to work.

Good environment memories: "Project uses Python 3.12 with venv at .venv." "OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes." "Deployment script at scripts/deploy.sh, must run from project root."

These are worth saving because they're stable (your Python version doesn't change daily), they're non-obvious (Hermes can't guess your venv path), and they're needed frequently (almost every coding task needs to know where things are).

Priority 3: Procedural knowledge

This is the "how do we do things around here" category. "Always run the test suite before committing." "Load the style calibration document before writing." "Use feature branches, never commit to main directly."

Procedural knowledge is valuable, but it ranks below preferences and environment facts for a reason: it's often project-specific rather than universal, and it's more likely to be something you'd explicitly ask for ("please run the tests") rather than something Hermes needs to know autonomously.

Also, many procedures are better saved as *skills* rather than memory entries. We'll cover skills in section 7.7. For now, know that if a procedure has more than two or three steps, it probably belongs in a skill file.

What NOT to Save (And Why)

Let me be more explicit about the anti-patterns, because this is where most people go wrong:

Task progress. "Currently working on the API endpoint." "Finished the login page." "Halfway through the database migration." This is session state. It matters right now, but it won't matter in the next session. Task progress belongs in your project management tool, not in memory.

Session outcomes. "Fixed the bug in auth.py." "Refactored the database connection pool." These describe what happened, not what's still true. The fix is done — the bug no longer exists. The fact that it *used* to exist isn't useful to carry forward.

Temporary TODO state. "TODO: write tests for the new module." "TODO: update the README." You'll either do these things or you won't. Putting them in memory doesn't make them more likely to get done, and they'll be stale within a day.

Raw data dumps. "The CSV has 847 rows and 12 columns." "The error log shows 23 warnings." Hermes can re-read files any time. Don't use precious memory space to store information that's available on disk.

Things easily re-discovered. File sizes, line counts, git branch names, process IDs — these change constantly and can be checked with a single command. The cost of re-discovering them is trivial. The cost of storing them in memory is a permanent drain.

The "Injected Every Turn" Constraint

Here's the fact that should shape every memory decision you make: memory is injected into Hermes's context at the start of every single turn. Not every session. Every turn.

This means that if you have 2,000 characters of memory, Hermes reads 2,000 characters of background notes before generating each response. If you have 500 characters, it reads 500. The less memory you use, the more context is available for actual work.

This is why the character limits exist, and this is why the priority order matters. Every byte of memory has an ongoing cost, paid every turn, for as long as the entry exists. An entry needs to earn its keep by being useful often enough to justify the cost.

An entry like “User prefers Word but fine with markdown; export to .docx with pandoc” earns its keep every time Hermes writes a document. That’s frequent.

An entry like “On March 15th we fixed the auth bug” never earns its keep again. It’s pure cost.

Keep your memory lean. If in doubt, leave it out. You can always add it later if you find yourself repeating it.

7.5 Session Search — Recalling the Past

Memory is for what still matters. But sometimes you need to recall what happened — not because the information is still actionable, but because you need to remember the context, reconstruct a decision, or find something you discussed last week.

That’s where session search comes in.

```
hermes sessions browse
```

This command does a full-text search across your past conversations using FTS5 (that’s SQLite’s full-text search engine, for the curious). It doesn’t search MEMORY.md or USER.md — it searches the actual conversation logs from previous sessions.

Let me show you the two modes:

Mode 1: Recent sessions (no query)

```
hermes sessions browse
```

Without a keyword, this shows your most recent sessions — the “what have I been working on lately?” view. Useful for picking up where you left off.

Mode 2: Keyword search

```
hermes sessions browse "database migration"
```

With a keyword, FTS5 searches across all stored conversations for matches. This is how you find that conversation from three weeks ago where you discussed the migration strategy.

When to Use Session Search vs. Memory

This distinction took me a while to get right, so let me spell it out clearly:

Use memory when: the information still matters, right now, and will continue to matter in future sessions. “The project uses MkDocs with Material theme” — that’s still true today and it’ll be true tomorrow. Save it to memory.

Use session search when: you need to recall what happened in a past conversation, but you don’t need to carry the information forward permanently. “What did we decide about the caching strategy?” — you need the answer, but you don’t need to save “we decided on Redis” to memory (you’d save that as a project convention in memory if it’s the permanent decision, but you’d use session search to find the discussion that led to it).

The shorthand I use: **memory is what still matters. Session search is what happened.**

Here’s a practical scenario. You worked on a project with Hermes last month, and you discussed several approaches to error handling. You ended up choosing approach B, and you want to remind yourself why. You would:

1. Use hermes sessions browse "error handling approach" to find the conversation.
2. Read the discussion and refresh your memory on the reasoning.
3. If the chosen approach is a standing convention, save it to memory: “ERROR HANDLING: Use approach B (custom error classes with middleware catch)”

Notice the distinction: session search is for the *discussion*. Memory is for the *decision*. The discussion is historical. The decision is current.

One caveat: session search isn’t as fast as reading memory (which is always pre-loaded). If you find yourself searching for the same keyword repeatedly across sessions, that information belongs in memory instead.

7.6 The Memory Nudge System

Here's a practical problem: Hermes is in the middle of helping you refactor a complex codebase. It's making changes, running tests, fixing errors — it's deeply focused on the task. In that state, it's not thinking about whether there's anything worth saving to memory. And you, the user, are focused on the same task. Nobody is thinking about memory.

Enter the nudge system.

The nudge system is Hermes's way of reminding itself to save memories. Here's how it works:

Every nudge_interval turns, Hermes gets an internal reminder: "Hey, have you noticed anything worth saving to memory?" The default interval is 10 turns, configured in the memory section of config.yaml:

```
memory:
  nudge_interval: 10    # remind agent to save memories every 10
turns
  flush_min_turns: 6    # minimum turns before flushing memories
```

So every 10 turns, Hermes pauses and thinks: "Has this user told me anything new about themselves? Have I learned anything about the environment? Is there a correction I should save?" If the answer is yes, it saves the memory. If not, it continues without interrupting you.

flush_min_turns — Preventing Over-Eager Saves

The flush_min_turns setting is a safeguard. It specifies the minimum number of turns that must pass before Hermes actually writes to memory. The default is 6.

Why would you want a minimum? Because early in a session, Hermes is still getting oriented. It might learn something in turn 2 that seems worth saving, but by turn 4, it turns out to be irrelevant or incorrect. If Hermes saved eagerly after turn 2, you'd end up with stale or wrong entries.

The flush_min_turns of 6 gives the session time to settle before memories start getting committed. It's a bit like how you wouldn't start taking detailed meeting notes until the meeting has actually gotten going — the preliminary chitchat and agenda-setting aren't worth recording.

Auto-Flush at Session End

There's one more mechanism: when a session ends, Hermes automatically flushes any memories that were identified but not yet saved. This is the safety net. Even if the nudge interval hasn't been reached, even if the minimum turns haven't passed — when the session is wrapping up, Hermes does a final check and saves whatever's pending.

This is important because the highest-value memories — user corrections — often happen near the end of a session. You're reviewing the work, you notice something wrong, you say "Actually, I prefer X," and then the session ends. Without the auto-flush, that correction would be lost. With it, Hermes catches it on the way out.

Configuring the Timing

The default values (nudge every 10 turns, flush after minimum 6 turns) work well for most situations. But you can adjust them:

- **Lower nudge_interval** (e.g., 5): More frequent checks. Good for fast-paced sessions with lots of new information, but adds minor overhead.
- **Higher nudge_interval** (e.g., 20): Less frequent checks. Good for long, focused sessions where memory changes are rare. Risk: missing corrections that happen between nudges.
- **Lower flush_min_turns** (e.g., 3): Memories committed earlier. Good for very short sessions. Risk: saving premature information.
- **Higher flush_min_turns** (e.g., 10): More conservative, more accurate. Risk: losing memories if short sessions end before the minimum is reached.

My recommendation: start with the defaults. They're well-calibrated. Only adjust if you notice specific problems — either things getting saved that shouldn't (lower flush_min_turns), or things not getting saved that should (raise nudge_interval or lower flush_min_turns).

7.7 Skills as Procedural Memory

We've talked about MEMORY.md and USER.md, which store facts. But there's a third kind of remembering in Hermes: skills.

Skills are saved workflows. If MEMORY.md is “facts I know about the world,” skills are “procedures I know how to execute.” They’re procedural memory — the difference between knowing that your car uses unleaded fuel (factual memory) and knowing how to parallel park (procedural memory).

Let me show you what I mean. Imagine you frequently ask Hermes to review your writing. The workflow is always the same: load the Obsidian vault, read the style calibration page, then review the document paragraph by paragraph. You could save “WRITING: Load vault [[harry-style-calibration]] before writing” to MEMORY.md, and that works — it reminds Hermes of the first step.

But if the workflow has multiple steps, each with conditions and variations, a single memory line isn’t enough. You need a skill.

The Skills Directory

Skills live in ~/.hermes/skills/. Each skill is a file (or directory of files) describing a reusable workflow. Think of them as playbook entries — more structured than a memory line, more persistent than a chat instruction.

A skill file might look something like this (simplified):

```
# Writing Review Skill

## Prerequisites
- Load Obsidian vault at configured path
- Read [[harry-style-calibration]] from vault

## Procedure
1. Read the document to review
2. Check against style calibration points
3. Flag deviations with specific references
4. Suggest corrections inline

## Notes
- User prefers minimal feedback on grammar
- Focus on style and tone, not copy editing
```

This is procedural knowledge that’s too detailed for a memory line but too reusable to re-explain every session. Skills bridge that gap.

When to Save a Skill vs. a Memory Entry

My decision framework is simple:

- **Can you express it in one line?** → Memory entry.
- **Does it have multiple steps or conditions?** → Skill.
- **Is it a fact about the world?** → Memory entry.
- **Is it a procedure for doing something?** → Skill.

“OBSIDIAN VAULT at /home/mikesai3/Documents/Hermes” is a fact. One line. Memory entry.

“WRITING: Load vault [[harry-style-calibration]] before writing” is a one-step procedure. Memory entry (barely — it’s on the boundary).

“Here’s how to do a comprehensive writing review: load the vault, check the calibration, read the document, compare paragraph by paragraph, flag deviations, suggest corrections, and skip grammar nitpicks” — that’s a multi-step procedure. Skill.

The boundary isn’t always clean, and that’s okay. The important thing is that both options exist, and you should use whichever one fits your information better.

The Obsidian Vault Connection

There’s a special connection between skills and Obsidian vaults. If you use Obsidian for personal knowledge management (and many Hermes users do), you can configure Hermes to reference your vault:

In config.yaml, the key skills.config.wiki.path points to your Obsidian vault directory. When set, skills can reference vault pages using the [[page-name]] syntax Obsidian users will recognize.

This makes your Obsidian vault an extension of Hermes’s memory. Your vault holds the deep knowledge — project docs, design decisions, references. Hermes’s memory holds the pointers — “the project conventions are in [[project-conventions]],” or “the style calibration is in [[harry-style-calibration]].”

It’s a two-tier system: memory gives Hermes the map, and the vault gives Hermes the territory. Together, they provide both the breadth of knowing what exists and the depth of being able to access it.

7.8 My Memory Mistakes — Stories from the Trenches

I promised you "I messed up" stories, and I'm going to deliver. These are real mistakes I made (or watched others make) while learning to use Hermes's memory system. They're embarrassing in retrospect, but they taught me lessons I now consider foundational.

Story 1: The Great Memory Bloat Disaster

When I first discovered Hermes's memory system, I went — and there's no other word for it — bananas. I saved *everything*. Every fact I mentioned in conversation, every tool I used, every file I touched, every opinion I expressed — all of it went into MEMORY.md.

Within a week, my memory file was pushing 3,500 characters. Well over the 2,200-character limit. And I'm not entirely sure how it happened — I think Hermes was truncating, or possibly just ignoring entries beyond the limit — but the practical effect was clear: the most recent entries were fine, and the earliest entries had disappeared. Entries I was relying on were just… gone.

The worst part? The entries that pushed me over the limit were almost all junk. Here's a sample of what I'd saved:

- "Today we set up the CI/CD pipeline" (session outcome)
- "The project has 47 files" (easily re-discovered)
- "We discussed using PostgreSQL but decided on SQLite" (historical discussion, not current fact)
- "Current task: implement user authentication" (temporary state)
- "The API returns JSON" (obvious for a REST API)

None of these earned their place. The CI/CD pipeline was set up — the session outcome didn't need to be recorded. The file count would change the next time I added a file. The database decision was already reflected in the project config. The current task would be done by the next session. And "the API returns JSON" is like saving "water is wet."

What I lost to make room for this junk? Actual valuable entries: - My preferred error handling pattern - The path to my Obsidian vault - A correction about how I like documentation structured

It was like filling a small backpack with rocks and then being surprised you can't fit your lunch in it.

The fix was painful. I opened MEMORY.md, read every entry, and asked: "Will I need this next week?" If no, I deleted it. I went from 3,500 characters to about 1,200. And Hermes worked *better* — the information it needed was there, and the noise was gone.

Lesson: Memory is a small backpack, not a storage unit. Every item you pack needs to justify its weight.

Story 2: Forgetting to Save User Corrections

This one is about my emoji problem.

I have a strong preference against emoji in professional documents. No smiley faces in READMEs, no checkmarks in status reports, no rocket ships in commit messages. It's a style thing — I find them distracting and unprofessional in technical writing.

The first time I worked with Hermes on a documentation project, it put emoji everywhere. Chapter markers had 🔵, warnings had ⚠️, tips had 💡. I corrected it: "Please remove the emoji. I don't use them in my documents."

Hermes removed them. Great. But it didn't save the correction to memory.

The next session — same project, new conversation — Hermes put emoji everywhere again. Same correction, same removal, same failure to save.

This went on for four sessions before I realized the problem wasn't Hermes being stubborn — the correction was happening mid-task, and Hermes prioritized task execution over the memory save. The nudge would eventually remind it, but by then we'd moved on.

The fix was embarrassingly simple. I explicitly told Hermes: "Save this to memory. I do not use emoji in my documents. Ever." And Hermes saved it:

```
STYLE: No emoji in headings or documents — user finds them
unprofessional
```

From that session on, no emoji. Problem solved. But it took me four sessions of frustration to get there, and all because I assumed corrections would be saved automatically without me having to say “save this.”

Lesson: When you correct Hermes, explicitly ask it to save the correction. Don’t assume it’ll happen automatically.

Story 3: The Stale Memory Problem

One more, shorter this time. I had a memory entry that read:

```
PYTHON: Project uses Python 3.11, venv at .venv
```

This was true when I saved it. Three months later, I upgraded to Python 3.12 for the project. But I forgot to update the memory entry. So for months, every time Hermes started a new session, it read “Python 3.11” and… used Python 3.11 conventions. It suggested syntax that was outdated, it didn’t use features available in 3.12, and on one memorable occasion it tried to run a command that failed because Python 3.11 wasn’t even installed anymore.

Memory entries are only valuable when accurate. An outdated entry is worse than no entry — Hermes trusts its memory implicitly.

The fix is the replace action:

```
Hermes uses the memory tool with:
  action: replace
  target: memory
  old_text: "PYTHON: Project uses Python 3.11, venv at .venv"
  new_text: "PYTHON: Project uses Python 3.12, venv at .venv"
```

And the broader lesson: periodically review your memory files. Once a month, open ~/.hermes/MEMORY.md and ~/.hermes/USER.md, read every entry, and ask: “Is this still true?” If not, update it or remove it. It takes five minutes, and it prevents the slow decay of trust that comes from stale memories.

Lesson: Memory needs maintenance. Review periodically. Update or remove stale entries.

Putting It All Together

Let's zoom out and see the complete picture of how Hermes remembers:

1. **Two stores:** MEMORY.md (the agent's notebook about the world) and USER.md (the user profile about you). Both loaded every turn. Both with character limits that force discipline.

2. **The memory tool:** with three actions (add, replace, remove) and two targets (memory, user). Simple, fast, and unambiguous.

3. **Priority order:** user corrections and preferences first, then environment facts, then procedural knowledge. This isn't just a ranking — it's a decision framework for what makes the cut.

4. **What not to save:** session outcomes, temporary state, easily re-discovered facts, raw data. These are the most common memory mistakes, and they're the ones that cause bloat.

5. **Session search:** for recalling past conversations. Different from memory — "what happened" vs. "what still matters." Use hermes sessions browse when you need to revisit history.

6. **The nudge system:** automatic reminders every 10 turns, with a minimum turn threshold before saving, and auto-flush at session end. Safety nets to prevent lost memories.

7. **Skills:** procedural memory stored in ~/.hermes/skills/, for workflows too complex for a single memory line. Connected to Obsidian vaults through skills.config.wiki.path.

Use all seven of these together, and Hermes transitions from a capable stranger to a genuine collaborator — one that knows you, knows your environment, and gets better with every session.

Try It Now

Time to get your hands dirty. This exercise walks you through the core memory operations: saving, searching, and updating. Do each step in order, and by the end, you'll have a working memory system with real entries that will make your next session better.

Step 1: Save a User Preference

Open a new session with Hermes and tell it something about yourself that you'd want it to remember. Be specific. Here are some examples to adapt:

- "I prefer [tool/format] for my files. Save this to my user profile."
- "When you show me code, I want [style: verbose comments / minimal / with examples]. Save this."
- "My name is [name] and I'm a [role]. Save this to my user profile."

The explicit instruction "Save this" is important — don't count on Hermes guessing. And specify the target: "my user profile" helps Hermes route it to user.

Verify it worked:

```
cat ~/.hermes/USER.md
```

You should see your entry. If you don't, say it again with more explicit instructions: "Use the memory tool to add this to the user target."

Step 2: Save an Environment Fact

Tell Hermes something about your environment that it wouldn't know otherwise. This goes to the memory store, not the user profile:

- "My main project lives at [path]. Save this to memory."
- "I use [tool] with [configuration]. Save this to memory."
- "The build command for this project is [command]. Save this to memory."

Verify:

```
cat ~/.hermes/MEMORY.md
```

Check that the entry is specific (includes the exact path, not just "my project"), durable (will still be true tomorrow), and non-obvious (Hermes couldn't guess it).

Step 3: Save a Procedural Memory

This is the trickiest one. Think of a workflow or procedure you follow repeatedly — something you'd want Hermes to know without being told:

- "Before writing documentation, always [step]. Save this to memory."
- "When I ask for a code review, I want you to check [specific things]. Save this to memory."
- "My testing workflow is: [procedure]. Save this to memory."

If the procedure is more than a couple of steps, consider making it a skill instead:

- Create a file in ~/.hermes/skills/ with a clear name (e.g., code-review.md).
- Write out the steps.
- Add a memory entry pointing to the skill: "CODE REVIEW: See skill at ~/.hermes/skills/code-review.md"

Step 4: Search Past Sessions

Run a session search to see what Hermes has been saving:

```
hermes sessions browse
```

This shows your recent sessions. Note the topics — this is the "what have I been working on?" view.

Now search for a specific keyword:

```
hermes sessions browse "python"
```

(Replace "python" with a keyword relevant to your actual work.) Browse the results. Notice that this is *different* from memory — it's historical context, not standing facts. If you find something in session search that *should* be a standing fact, that's your cue to save it to memory.

Step 5: Replace an Outdated Memory

This step requires that you have an existing entry to update. If you just did steps 1-3 for the first time, everything is fresh. But come back to this step after a few sessions when something changes.

Let's say your project directory moves, or you switch tools, or your role changes. Here's how to update:

1. Open ~/.hermes/MEMORY.md and find the outdated entry.
2. In a new session with Hermes, say: "My project has moved. It's now at [new path]. Please update the memory entry."
3. Hermes should use the replace action with the old text and the new text.
4. Verify the update:

```
cat ~/.hermes/MEMORY.md
```

Alternatively, you can do this manually. Open memory.md in your editor, find the old entry, and update it. The memory tool is convenient, but memory.md is just a file — you can edit it directly if you prefer.

Step 6: Maintenance Review

This is the step most people skip, and it's the one that prevents the most problems. Set a reminder for yourself (calendar, sticky note, whatever works) to review your memory files once a month. When that reminder fires:

1. Open ~/.hermes/MEMORY.md.
2. Read every entry. For each one, ask:
    - Is this still true?
    - Would I be upset if Hermes didn't know this?
    - Is this something Hermes could figure out on its own?
3. Update or delete entries that fail the test.
4. Repeat for ~/.hermes/USER.md.

This takes about five minutes. It's the memory equivalent of weeding a garden — not glamorous, but it keeps the whole system healthy.

Quick Reference: Memory Cheat Sheet

Files: - ~/.hermes/MEMORY.md — agent's notes (env, conventions, lessons) — 2,200 char limit - ~/.hermes/USER.md — user profile (name, role, preferences, pet peeves) — 1,375 char limit

Config (in ~/.hermes/config.yaml):

```
memory:
  memory_enabled: true
  user_profile_enabled: true
  memory_char_limit: 2200
  user_char_limit: 1375
  nudge_interval: 10
  flush_min_turns: 6
  provider: ''
```

Memory tool actions: - add — new entry (requires: target, content) - replace — update entry (requires: target, old_text, new_text) - remove — delete entry (requires: target, old_text)

Target parameter: - memory for ~/.hermes/MEMORY.md - user for ~/.hermes/USER.md

Priority (save this first): 1. User preferences and corrections 2. Environment facts (OS, tools, project structure) 3. Procedural knowledge

Don't save: - Task progress or session outcomes - Temporary TODO state - Raw data dumps - Things easily re-discovered

Session search: - hermes sessions browse — recent sessions - hermes sessions browse "keyword" — FTS5 keyword search

Skills: - Directory: ~/.hermes/skills/ - Obsidian vault: skills.config.wiki.path in config.yaml

Memory quality test (three questions): 1. Will this still be true next week? 2. Could Hermes figure this out on its own? 3. Would I be annoyed if Hermes didn't know this?

If all three answers are "yes," save it. If not, think harder.

Memory is what turns Hermes from a tool into a partner. It's the mechanism by which your AI agent stops being a stranger and starts being someone who knows you. The two files are small — 2,200 characters and 1,375 characters — but what you put in them shapes every interaction you'll ever have with Hermes.

Save what matters. Skip what doesn't. Review periodically. And when you correct Hermes, always, *always* say: "Save this."

Chapter 8: Skills — Teaching Hermes New Tricks

Every morning, I make coffee the same way. Grind the beans, heat the water to 205 degrees, bloom the grounds for thirty seconds, pour slowly in concentric circles. I don't reinvent my pour-over process every day. I have a procedure — a repeatable workflow that I follow because it works and because I do it often.

Hermes works the same way. When you find yourself asking Hermes to do the same kind of task over and over, and each time you're essentially walking it through the same steps, you're wasting time — yours and Hermes's. That's exactly the problem skills solve.

Skills are Hermes's procedural memory. They are reusable, step-by-step workflows for recurring task types. And by the time you finish this chapter, you'll be creating skills that make Hermes faster, more consistent, and a lot more useful.

Here's the thing that surprised me when I first started using skills: I didn't need to learn a programming language. I didn't need to write hooks or callbacks or event listeners. A skill is just a document — specifically, a Markdown file with some YAML on top — that tells Hermes how to handle a type of task. If you can write a checklist, you can write a skill.

Let me show you what I mean before we get into the details.

8.1 What Are Skills? — Procedures vs. Facts

Here's a conversation I had with Hermes before I started using skills:

```
Me: Review the pull request for the auth module.
Hermes: I'll review it. Let me look at the PR... [reviews code]...
        Here are my findings...

Me: Review the pull request for the payment module.
Hermes: Sure! Let me check that PR... [reviews code]...
        Here's what I found...

Me: Review the pull request for the user profile module.
Hermes: Okay, looking at this PR now... [reviews code]...
        Here's my review...
```

Each time, I was getting different quality reviews. Sometimes detailed, sometimes surface-level. Sometimes Hermes checked for test coverage, sometimes it didn't. Sometimes it looked at error handling, sometimes it skipped right past it. I was getting inconsistent results because I was essentially asking Hermes to reinvent its review process from scratch every single time.

Then I created a skill called pr-review. Now when I say "review this pull request," Hermes loads the skill and follows the same thorough, consistent process every time. Step one: check the diff. Step two: verify test coverage for changed code. Step three: review error handling. Step four: check for security considerations. Step five: summarize findings.

Same task, same quality, every time. That's what skills do.

Skills vs. Memories: Know the Difference

In Chapter 7, you learned about memories — Hermes's way of storing facts. Memories answer the question "what is true?" Skills answer the question "how do I do this?"

Let me be really clear about the difference, because this trips people up:

Memories	Skills
Store facts	Store procedures
"I prefer tabs over spaces"	"How to set up a new microservice"
"The project uses Python 3.11"	"How to deploy to staging"
"My name is Mike"	"How to review a pull request"
One-liners or short statements	Numbered steps and workflows

Memories	Skills
Saved automatically or with save_memory	Created with skill_manage

The litmus test is simple: if it has numbered steps, it's probably a skill. If it's a one-liner fact, it's probably a memory.

"Deploy to production using the blue-green strategy" — that's a skill. It involves multiple steps, decisions, and verification checks.

"The production server is at prod.example.com" — that's a memory. It's a fact, not a procedure.

When Do You Need a Skill?

You need a skill when you find yourself repeating a workflow that has three or more steps. Here are some signals:

- You've explained the same process to Hermes more than twice
- You find yourself writing "like last time, but…" frequently
- You're copy-pasting instructions from a previous conversation
- Hermes handles a task well once but inconsistently the next time
- A task requires a specific sequence of steps to get right

The creation nudge system helps here too. Hermes keeps track of how many complex tasks it handles, and after every 15 complex tasks (configurable via creation_nudge_interval), it suggests creating a skill. So if you're not sure whether something deserves a skill, just wait — Hermes will nudge you.

But honestly? After a while, you'll start creating skills proactively. It becomes second nature. You do something twice, and on the third time you think, "This should be a skill."

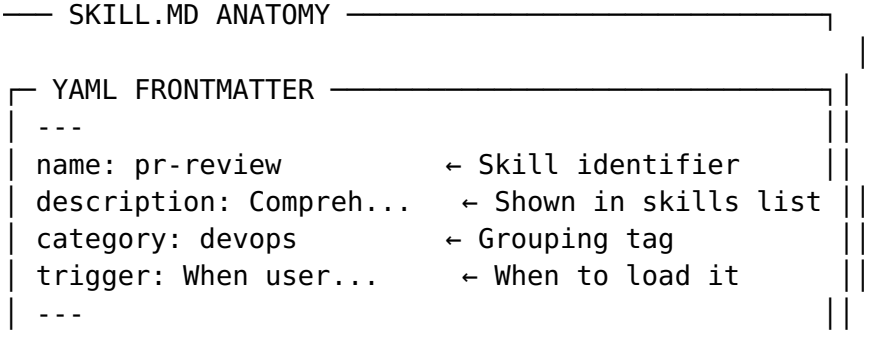
8.2 Skill Anatomy — Inside a SKILL.md

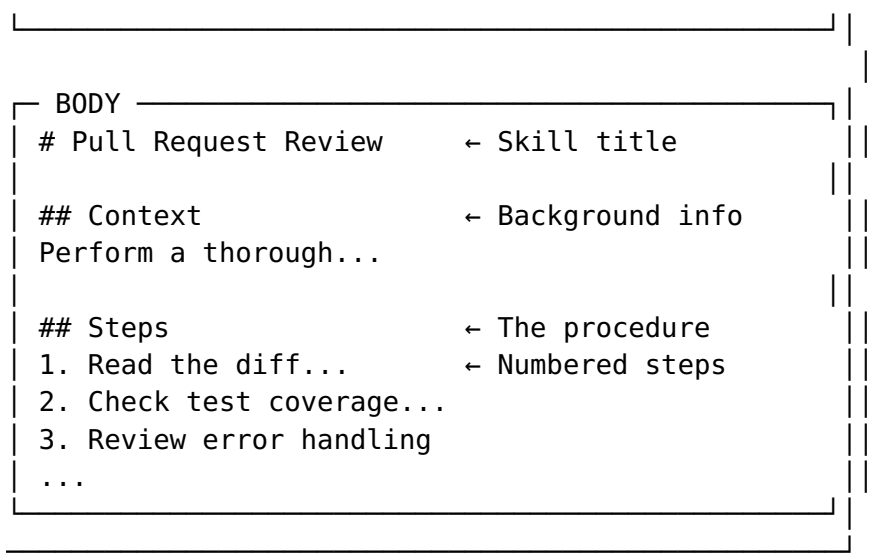
Okay, you've seen a skill in action. Let me take one apart and show you every piece.

Before I explain each part, here's a complete, real skill — one I use all the time:

```
---
name: pr-review
description: Comprehensive code review for pull requests
category: devops
trigger: When user asks to review a PR or pull request, load this
skill
---

# Pull Request Review

## Context
Perform a thorough, consistent code review following these steps.
Adapt depth based on PR size but always cover all categories.

## Steps

1. **Read the diff** — Use `read_file` or terminal to examine all
changed files. Note the scope of changes.

2. **Check test coverage** — Identify which changed modules have
corresponding test changes. Flag files with new logic but no test
updates.

3. **Review error handling** — Look for unhandled edge cases,
swallowed exceptions, and missing error responses.

4. **Security considerations** — Check for SQL injection, XSS,
exposed secrets, and overly permissive access.

5. **Code quality** — Assess naming, structure, duplication, and
adherence to project conventions.

6. **Summarize findings** — Produce a structured review with:
   - **Must Fix**: Issues that should block merging
   - **Should Fix**: Issues that reduce quality but aren't
blockers
   - **Nice to Have**: Suggestions for improvement
```

Now let me label every part of this anatomy:





Let me walk through each part.

YAML Frontmatter

The YAML frontmatter is the metadata sandwiched between the --- delimiters at the top of the file. It has four fields:

name (required) — The identifier for the skill. This is what you use with skill_view(name) and skill_manage. It should be lowercase, hyphenated, and concise. Good names: pr-review, deploy-staging, new-microservice. Bad names: My Awesome Skill, review, thing.

description (required) — A one-line description that appears in the skills list. This is what you see when you run skills_list. It should be short and action-oriented. Think of it as the skill's elevator pitch.

category (optional) — A grouping tag for organizing related skills. You can filter by category when listing skills with skills_list(category='devops'). Common categories I use: devops, coding, writing, research, data. But you can use any category that makes sense for your workflow.

trigger (optional but recommended) — A natural-language description of when this skill should be loaded. When Hermes encounters a task that matches the trigger, it loads the skill. For example, "When user asks to review a PR or pull request, load this skill" tells Hermes to activate this skill for code review requests.

Body

The body is where the actual procedure lives. It's written in plain Markdown — headers, paragraphs, numbered lists, code blocks, whatever you need. This is the instruction manual that Hermes follows when it loads the skill.

Good skill bodies share these traits:

- They start with a title (a Markdown # header)
- They include a context section explaining when and why to use the skill
- They use numbered steps for the main procedure
- They specify what the output should look like
- They're specific enough to be consistent but flexible enough to adapt

Supporting Files

A skill isn't limited to just SKILL.md. Each skill directory can contain subdirectories for additional files:

```
~/.hermes/skills/pr-review/
├── SKILL.md                    ← The main skill file (required)
├── references/
│   └── style-guide.md          ← Reference docs the skill needs
├── templates/
│   └── review-template.md      ← Template for the review output
├── scripts/
│   └── coverage-check.sh       ← Executable scripts
└── assets/
    └── severity-rubric.png     ← Images and data files
```

These subdirectories are optional. A simple skill might just be a SKILL.md and nothing else. But as your skills grow more sophisticated, you can add supporting files to keep the SKILL.md itself clean and focused.

Here's what each subdirectory is for:

references/ — Reference documents that the skill needs to consult. Style guides, API documentation, company policies, coding standards. When Hermes loads the skill, it can access these files with skill_view(name, file_path='references/style-guide.md').

templates/ — Template files that the skill uses to produce output. If your skill generates reports, emails, or configuration files, you can store templates here so Hermes doesn't have to reconstruct the format from memory.

scripts/ — Executable scripts that the skill runs. These are shell scripts, Python scripts, or any other executable files that the skill calls as part of its workflow.

assets/ — Images, data files, or any other static resources the skill needs. Severity rubrics, sample data, diagrams — things that support the skill but aren't executable.

8.3 Creating Your First Skill

Enough theory. Let's create a skill.

I'm going to start with a practical example: a skill for writing Git commit messages. I was inconsistent about my commit messages — sometimes detailed, sometimes lazy, sometimes omitting context. So I made a skill.

Here's how I created it using skill_manage:

```
skill_manage(
    action='create',
    name='commit-message',
    content='''---
name: commit-message
description: Write clear, consistent Git commit messages
category: coding
trigger: When user asks to write a commit message or is about to
commit code
---

# Git Commit Message Writer

## Context
Write commit messages following the Conventional Commits
specification.
Produce a message that is clear, specific, and useful for future
readers.
```

```
## Steps

1. **Examine the changes** — Review the staged diff using `git
diff --staged` or read the changed files.

2. **Determine the type** — Classify the change as one of:
   - feat: A new feature
   - fix: A bug fix
   - docs: Documentation changes
   - refactor: Code restructuring without behavior change
   - test: Adding or updating tests
   - chore: Maintenance tasks

3. **Write the subject line** — Keep it under 72 characters. Use
imperative mood ("add feature" not "added feature").

4. **Write the body** — Explain WHY the change was made, not WHAT
was changed (the diff shows that). Wrap at 72 characters.

5. **Add breaking change notes** — If this change breaks backward
compatibility, add a BREAKING CHANGE section.

6. **Present the message** — Show the complete commit message and
offer to run `git commit` with it.
'''
)
```

When this command runs, Hermes creates the directory ~/.hermes/skills/commit-message/ and writes SKILL.md inside it. That's it. The skill now exists.

Let me verify it's there:

```
skills_list()
```

Output:

```
Available Skills:
- commit-message: Write clear, consistent Git commit messages
```

And I can view the full skill:

```
skill_view('commit-message')
```

This returns the complete SKILL.md content plus a linked_files dictionary showing any supporting files (empty for now, since we haven't added any).

Adding Supporting Files

My commit message skill is decent, but it could be better. What if Hermes had a reference document explaining the Conventional Commits spec in detail? Let me add one:

```
skill_manage(
    action='write_file',
    name='commit-message',
    file_path='references/conventional-commits.md',
    content='''# Conventional Commits Reference

## Format
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]

## Types
- feat: A new feature for the user (not a build or ci feature)
- fix: A bug fix for the user (not a build or ci fix)
- docs: Changes to documentation
- style: Formatting, missing semi-colons, etc. (no code change)
- refactor: Refactoring code without changing behavior
- perf: Performance improvements
- test: Adding missing tests or correcting existing ones
- build: Changes to build process or external dependencies
- ci: Changes to CI configuration
- chore: Other changes that don't modify src or test files

## Rules
1. Subject line: max 72 characters, imperative mood
2. Body: wrap at 72 characters, explain WHY not WHAT
3. Breaking changes: use BREAKING CHANGE: in footer or ! after
type
'''
)
```

Now the skill has a reference file. When Hermes loads the commit-message skill, it can call skill_view('commit-message', file_path='references/conventional-commits.md') to get the detailed specification.

Testing Your Skill Immediately

Here's a principle I want you to internalize: create a skill, then test it right away. Don't wait. Don't create five skills and then test them all later. Create one, test it, fix it if needed, then move on.

After creating my commit-message skill, I test it like this:

```
Me: Write a commit message for the changes I just staged.
```

If Hermes is working correctly with skills, it should: 1. Match this request to the commit-message skill's trigger 2. Load the skill with skill_view('commit-message') 3. Follow the steps in the skill 4. Produce a properly formatted commit message

If it works, great. If it doesn't match the trigger or skips steps, I know I need to adjust either the trigger description or the steps.

8.4 Loading and Using Skills

Now that you have a skill, how does Hermes actually find and use it? Let me demystify the process.

The Skill Discovery Process

When a new session starts, Hermes scans your skills directory. It reads the YAML frontmatter from every SKILL.md file and builds an inventory of available skills. This inventory shows up in the system prompt as a list called available_skills, with each skill's name and description.

So when you start a conversation, Hermes already knows what skills are available:

```
available_skills:
- commit-message: Write clear, consistent Git commit messages
- pr-review: Comprehensive code review for pull requests
- deploy-staging: Deploy the main branch to the staging
environment
```

These are loaded passively — Hermes doesn't read the full content of every skill at the start. It just knows they exist and what their triggers say.

Trigger Matching

When you give Hermes a task, it checks whether any skill's trigger matches what you're asking for. If you say "review this pull request," Hermes compares that to the triggers of all available skills and finds that pr-review has a trigger that says "When user asks to review a PR or pull request, load this skill."

This is a match. Hermes then calls skill_view('pr-review') to load the full skill content, reads through the steps, and follows them.

The trigger matching is approximate, not exact. You don't need to use specific keywords. A trigger like "When user asks to review a PR or pull request" will match various phrasings: "can you check this PR?", "look over this pull request", "review these changes." Hermes is good at intent matching.

But — and this is important — if the trigger is too vague, you'll get false matches. If the trigger is too narrow, you'll miss legitimate requests. Write triggers that describe the intent clearly:

Good triggers: - "When user asks to deploy to staging or push to the staging environment" - "When user wants to create a new Python project or scaffold a Python app" - "When user asks to review code, a PR, or a pull request"

Bad triggers: - "When needed" (too vague — when is what needed?) - "Deploy" (too cryptic) - "When user says the exact phrase 'review PR #'" (too rigid)

Manually Loading a Skill

Sometimes Hermes might not match your request to a skill automatically. Or you might want to explicitly tell Hermes which skill to use. You can simply ask:

```
Me: Use the pr-review skill for this PR.
```

Or:

```
Me: Load the commit-message skill and help me write a commit.
```

Hermes will call skill_view(name) to load the skill and then follow its instructions. Being explicit is useful when a task could match multiple skills or when you want to ensure Hermes uses a specific workflow.

Listing and Filtering Skills

To see all your skills at any time, use skills_list():

```
skills_list()
```

This returns every skill's name and description. If you've organized skills with categories, you can filter:

```
skills_list(category='devops')
```

This returns only skills in the devops category. Useful when you have many skills and want to focus on a particular domain.

Following the Skill Instructions

When Hermes loads a skill, it reads the body of SKILL.md and follows the instructions. This isn't magic — it's just Hermes treating the skill content as instructions for the current task.

The key insight is that the better written your skill, the more consistently Hermes follows it. Vague steps produce vague results. Specific steps produce specific results. This is why I emphasize numbered steps and clear output specifications in your skill bodies.

Here's an example of the difference. Compare these two skill steps:

Vague: "Review the code for problems."

Specific: "Read every changed file using read_file. For each file, check: (a) whether any new functions lack test coverage, (b) whether error handling covers the main failure modes, (c) whether there are any SQL queries that might be vulnerable to injection."

The second one produces consistent, thorough reviews. The first one produces reviews that vary wildly in quality.

8.5 Updating and Fixing Skills

Skills aren't set in stone. They're living documents that evolve as your needs change, your processes improve, or your environment shifts. And just like code, skills have bugs — steps that are unclear, triggers that don't match well, or procedures that have gone stale.

Let me tell you about the three ways to update a skill, from least disruptive to most.

Patching: The Preferred Fix Method

Most skill updates are small: clarifying a step, fixing a typo, adding a condition. For these, use skill_manage(action='patch'). Patching replaces a specific string in the SKILL.md with a new string, similar to a find-and-replace operation.

For example, after using my commit-message skill for a week, I realized it didn't handle merge commits well. I added a step:

```
skill_manage(
    action='patch',
    name='commit-message',
    old_string='5. **Add breaking change notes**',
    new_string='5. **Check for merge conflicts** — If this is a
merge commit, summarize the branches being merged and any conflict
resolutions.\n\n6. **Add breaking change notes**'
)
```

Patching is surgical. It changes only what you specify, leaving the rest of the skill untouched. This means:

- You can't accidentally break other parts of the skill
- The change is minimal and easy to review
- You preserve the full history and context around the change

Always prefer patching for small updates. It's safer and faster than rewriting the entire skill.

Editing: For Major Overhauls

Sometimes a skill needs a complete rewrite. Maybe you've changed your entire deployment process, or the original skill structure was wrong, or you need to add so many changes that patching would be tedious and error-prone.

For these cases, use skill_manage(action='edit'). This replaces the entire SKILL.md content:

```
skill_manage(
    action='edit',
    name='commit-message',
    content='''---
```

````
name: commit-message
description: Write clear, consistent Git commit messages following
Conventional Commits
category: coding
trigger: When user asks to write a commit message or is about to
commit code
---

# Git Commit Message Writer

## Context
Write commit messages following the Conventional Commits
specification.
This skill ensures every commit message is clear, specific, and
useful.

## Steps

1. **Examine the changes** — Run `git diff --staged` to see what
will be committed. If nothing is staged, check `git status` and
ask the user what to stage.

2. **Determine the type** — Classify as: feat, fix, docs,
refactor, test, chore, perf, build, ci, or style.

3. **Determine the scope** (optional) — If the change affects a
specific module or component, include a scope in parentheses:
`feat(auth): add OAuth2 login`.

4. **Write the subject line** — Max 72 characters, imperative
mood, no period at the end.

5. **Write the body** (if needed) — Explain WHY the change was
made. Wrap at 72 characters.

6. **Check for breaking changes** — If this change breaks backward
compatibility, add a `BREAKING CHANGE:` section in the footer, or
append `!` after the type.

7. **Present the message** — Show the complete commit message
formatted and ready to use. Ask whether to execute the commit.

## Examples

feat(auth): add OAuth2 login support

Add Google and GitHub OAuth2 providers for user authentication.
This replaces the previous custom login form.

Closes #142
'''
)
````

Use editing sparingly. It replaces everything, so if you make a mistake, you've overwritten the entire skill. Always have the current skill content handy (load it with skill_view first) before doing a full edit.

Auto-Patching: When Hermes Fixes Its Own Skills

Here's something cool: Hermes can patch its own skills during a session. If Hermes is following a skill and encounters a problem — a step that doesn't work, a command that's outdated, a condition that's missing — it can immediately patch the skill to fix the issue.

This is one of my favorite features. It means your skills get better as you use them, not just when you manually maintain them.

For example, say I'm using a skill that includes a step "Run yarn test to execute the test suite." But my project has switched to npm test. Hermes tries the step, gets an error, realizes the command needs updating, and patches the skill on the spot:

```
skill_manage(
    action='patch',
    name='my-skill',
    old_string='Run `yarn test` to execute the test suite',
    new_string='Run `npm test` to execute the test suite'
)
```

Now the skill is fixed for next time too. This auto-patching is a game-changer because it means you don't have to be paranoid about getting every detail right when you first create a skill. Start rough, use the skill, and let it improve through real usage.

Deleting a Skill

If a skill is obsolete or you want to start from scratch, you can delete it:

```
skill_manage(
    action='delete',
    name='old-skill'
)
```

This removes the entire skill directory, including SKILL.md and all supporting files. Be certain before you do this — there's no undo.

8.6 Managing Skill Files

A skill's SKILL.md is its brain, but supporting files are its hands. Templates give Hermes consistent patterns to work from. Scripts let it execute complex operations. References provide the domain knowledge it needs.

Writing Supporting Files

You add supporting files to a skill using skill_manage(action='write_file'):

```
skill_manage(
    action='write_file',
    name='pr-review',
    file_path='templates/review-output.md',
    content='''# Pull Request Review: {PR_TITLE}

**Author:** {AUTHOR}
**Branch:** {BRANCH}
**Files Changed:** {FILE_COUNT}

## Summary
{SUMMARY}

## Must Fix
{MUST_FIX_ITEMS}

## Should Fix
{SHOULD_FIX_ITEMS}

## Nice to Have
{NICE_TO_HAVE_ITEMS}

## Verdict
{VERDICT}
'''
)
```

Now when Hermes runs the pr-review skill, it can reference templates/review-output.md as a formatting guide for the review output. The skill body might include a step like "Format your review using the template in templates/review-output.md."

You can write files to any of the four subdirectories:

- references/ — Documentation and reference material
- templates/ — Output templates and formatting guides

- scripts/ — Executable scripts the skill can run
- assets/ — Images, data files, and other static resources

Accessing Linked Files

When Hermes loads a skill with skill_view(name), it gets the SKILL.md content plus a dictionary of linked files. But sometimes it needs to access a specific file within the skill. That's where the file_path parameter comes in:

```
skill_view('pr-review', file_path='references/style-guide.md')
```

This returns just the content of that specific file. Hermes uses this when a step in SKILL.md says something like "Consult the style guide in references/style-guide.md for naming conventions."

The linked_files dictionary returned by skill_view(name) lists all supporting files with their relative paths, so Hermes always knows what's available:

```
{
    'references/style-guide.md': 'Full path to file',
    'templates/review-output.md': 'Full path to file',
    'scripts/coverage-check.sh': 'Full path to file'
}
```

Removing Obsolete Files

If a supporting file is no longer needed — maybe you've replaced a script with a better version, or a reference document is outdated — you can remove it:

```
skill_manage(
    action='remove_file',
    name='pr-review',
    file_path='scripts/old-coverage-check.sh'
)
```

This deletes just the file, leaving the rest of the skill intact. Use this when you're cleaning up a skill that has accumulated files you no longer need.

8.7 Advanced Skill Patterns

Once you're comfortable with basic skills, you can start building more sophisticated ones. Here are some patterns that I've found particularly useful.

Skills That Call Other Skills

A skill doesn't exist in isolation. Sometimes a workflow naturally involves steps that are already captured in another skill. For example, my release skill might need to write a commit message for the release commit. Instead of duplicating those steps, I can reference the commit-message skill:

```
---
name: release
description: Prepare and tag a new release
category: devops
trigger: When user asks to create a release or cut a new version
---

# Release Process

## Steps

1. **Verify the branch** — Ensure we're on main and up to date
with the remote.

2. **Run the test suite** — Execute all tests and confirm they
pass.

3. **Generate the changelog** — Collect commits since the last
release tag.

4. **Write the release commit** — Use the commit-message skill to
craft the release commit. Load the commit-message skill with
skill_view('commit-message') and follow its steps for the release
commit.

5. **Tag the release** — Create an annotated git tag with the
version number.

6. **Push the release** — Push the commit and tag to the remote.
```

See step 4? It explicitly references another skill. This is how you build a skill ecosystem — small, focused skills that compose together into larger workflows.

This works because when Hermes loads the release skill and gets to step 4, it reads the instruction to load the commit-message skill, calls skill_view('commit-message'), and then follows both sets of instructions.

Skills with Conditional Logic

Not every task follows a straight line. Sometimes the right next step depends on what you found in the previous step. Skills can include conditional logic:

```
---
name: debug-error
description: Systematic debugging workflow for runtime errors
category: coding
trigger: When user reports a runtime error or asks for help
debugging
---

# Debug Runtime Error

## Steps

1. **Reproduce the error** — Ask the user for the exact error
message and the steps to reproduce it. If the error cannot be
reproduced, note that and move to step 5.

2. **Read the stack trace** — If a stack trace is available,
identify the file and line number where the error originates.

3. **Examine the code** — Read the relevant source file around the
error location. Look for:
   - Null/undefined variable access
   - Type mismatches
   - Off-by-one errors
   - Missing error handling

4. **If the cause is identified** — Propose a fix, explain why it
works, and offer to implement it.

5. **If the cause is NOT identified** — Broaden the investigation:
   - Check recent changes: `git log --oneline -10`
   - Search for the error message in the codebase
   - Check environment variables and configuration
   - Add logging to narrow down the issue
   Then return to step 2 with the new information.

6. **Verify the fix** — After implementing, ask the user to test
whether the error is resolved.
```

Steps 4 and 5 contain “If…” conditions. Hermes processes these just like any other instructions — it evaluates the condition and takes the appropriate branch. You don’t need special syntax for this; plain English conditions work well.

Code-Generating Skills vs. Decision-Guiding Skills

Skills fall into two broad categories: those that generate code or output, and those that guide decisions.

Code-generating skills produce something tangible: a commit message, a configuration file, a test suite. The commit-message skill is a code-generating skill — its output is the commit message itself.

Decision-guiding skills help you think through a problem: whether to deploy, how to prioritize bugs, what architecture to choose. They produce a recommendation, not an artifact.

Both types are valuable. Code-generating skills are more common because they produce direct, measurable output. But decision-guiding skills can be even more impactful because they prevent you from making bad decisions.

Here’s a decision-guiding skill I use:

```
---
name: incident-triage
description: Triage production incidents and determine severity
category: devops
trigger: When user reports a production incident or asks for help
triaging an issue
---

# Production Incident Triage

## Steps

1. **Assess user impact** — How many users are affected? Is it a
subset (which?) or widespread?

2. **Assess functional impact** — Is a core feature broken, or is
it an edge case?

3. **Check for data loss** — Is any user data at risk of being
lost or corrupted?

4. **Classify severity:**
   - **P1 — Critical**: Complete service outage, data loss, or
security breach. Wake people up.
```

```
   - **P2 — High**: Major feature broken for many users. Respond
within 30 minutes.
   - **P3 — Medium**: Feature degraded for some users. Fix this
business day.
   - **P4 — Low**: Minor issue or cosmetic problem. Add to the
backlog.

5. **Recommend response** — Based on the severity, suggest next
steps: who to notify, whether to roll back, whether to create a
hotfix branch.
```

This skill doesn't generate code. It guides a decision process. But it's incredibly useful because it ensures I follow a consistent triage process even when I'm stressed and not thinking clearly (which is always when incidents happen).

Multi-File Skills

The most powerful skills combine all the supporting file types. Here's a skill I use for deploying to staging:

```
---
name: deploy-staging
description: Deploy the main branch to the staging environment
category: devops
trigger: When user asks to deploy to staging or push to staging
---

# Deploy to Staging

## Steps

1. **Pre-flight checks** — Run the scripts/pre-flight-check.sh
script to verify the environment is ready for deployment.

2. **Run the test suite** — Execute all tests. If any fail, stop
and report.

3. **Deploy** — Apply the templates/deploy-config.yaml template
with the current branch and commit hash. Then execute the
deployment using the generated configuration.

4. **Verify deployment** — Run scripts/health-check.sh to confirm
the staging environment is responding correctly.

5. **Report** — Summarize what was deployed, when, and the current
health status.
```

This skill references three supporting files:

- scripts/pre-flight-check.sh — A script that checks whether staging is ready for deployment
- templates/deploy-config.yaml — A template for the deployment configuration
- scripts/health-check.sh — A script that verifies the deployment succeeded

I could have put all of these instructions directly in SKILL.md, but separating them makes the skill cleaner and each file independently maintainable. When I need to update the pre-flight checks, I update the script file without touching SKILL.md. When I need to change the deployment configuration format, I update the template without touching the skill steps.

8.8 My Skill Mistakes — Lessons from the Field

Time for some honesty. I've made every mistake in the book when it comes to skills. Let me tell you about the three biggest ones so you can avoid them.

The Over-Engineered Skill

Early in my skills journey, I decided to create a comprehensive code review skill. I mean comprehensive. It had 27 steps. TWENTY-SEVEN. Step 1 was to read the diff. Step 2 was to categorize changes. Step 3 was to analyze function complexity. Step 4 was to check cyclomatic complexity. Step 5 was to look for design pattern violations. And on and on and on.

It also had six supporting files: a style guide, a complexity rubric, a design patterns reference, a security checklist, a performance checklist, and a template for the review output.

You know how many times I actually used that skill? Once. And it took forever. Hermes spent so much time working through all 27 steps that the review was comprehensive but exhausting to read. The signal-to-noise ratio was terrible — most of the review was boilerplate from steps that didn't apply to the PR I was reviewing.

The problem was that I tried to make the skill cover every possible scenario instead of focusing on the common case. I should have started with a 5-step skill covering the most important checks, then added steps only when I found gaps.

My rule of thumb now: if a skill has more than 8-10 steps, it's trying to do too much. Split it into multiple smaller skills or trim it down to what actually matters.

Here's what the over-engineered skill should have been — and what it eventually became after I rewrote it:

```
---
name: pr-review
description: Comprehensive code review for pull requests
category: devops
trigger: When user asks to review a PR or pull request, load this
skill
---

# Pull Request Review

## Steps

1. **Read the diff** — Examine all changed files. Note the scope
of changes.

2. **Check test coverage** — Flag files with new logic but no test
changes.

3. **Review error handling** — Look for unhandled edge cases and
swallowed exceptions.

4. **Security check** — Look for injection vulnerabilities and
exposed secrets.

5. **Summarize** — Produce a structured review with Must Fix,
Should Fix, and Nice to Have categories.
```

Five steps. Clear, focused, and consistent. That's the skill I use today, and it produces better reviews than the 27-step monster ever did.

The Skill That Broke After an Update

One of my deployment skills included a step that ran docker-compose up -d. It worked perfectly for months. Then I updated Docker on my server, and the organization changed. The skill's step now produced a deprecation warning and a different output format.

Hermes ran the skill, hit the deprecation warning, and… got confused. The skill expected one output format, got another, and the remaining steps didn't work correctly.

Here's the thing: Hermes auto-patched the skill to fix this specific issue. It updated docker-compose to docker compose (the newer syntax) and adjusted the expected output format. But it took a few minutes of troubleshooting before the auto-patch kicked in, and during that time I was stuck.

The lesson: skills that depend on external tools are fragile. When those tools update, the skills break. There are two strategies to mitigate this:

1. **Write skills defensively** — Instead of specifying exact command output formats, write steps that are format-agnostic. "Run the health check and verify the service is running" is better than "Run the health check and look for the line starting with 'Status: OK'."

2. **Trust auto-patching** — When a skill breaks, let Hermes try to fix it. It will often patch the skill automatically. But review the patch to make sure it's correct.

3. **Version your skill steps** — If you reference specific commands, note the version or environment they were tested with. This helps future-you (and Hermes) understand why a step might not work.

The Skill I Should Have Made but Didn't

For three months, I was deploying a microservice to production. Every time, I went through the same 12-step process: check the branch, run tests, bump the version, update the changelog, create a commit, create a tag, push the tag, trigger the CI pipeline, wait for the build, verify the deployment, run smoke tests, and notify the team.

Every. Single. Time. I typed out these instructions for Hermes, or I walked through them manually. And every time, I thought, "I should make a skill for this." And every time, I didn't, because I was too busy actually deploying to stop and create the skill.

Three months. That's roughly 12 deployments, each with about 10 minutes of manual instruction-giving. That's two hours of my life I'll never get back.

The deployment skill I finally created took 15 minutes to write. It has saved me far more than 15 minutes since then. The ROI is absurd.

The lesson: if you find yourself giving Hermes the same instructions more than twice, stop and make a skill. Right then. Don't wait. The five minutes you spend creating the skill pays for itself by the third use.

I now have a rule: if I type the same sequence of instructions to Hermes more than twice, I must create a skill before proceeding. It's a discipline thing. And if you're wondering why the creation_nudge_interval defaults to 15 — Hermes will nudge you every 15 complex tasks to create a skill. But honestly, you shouldn't wait that long.

8.9 Sharing Skills: the Skills Hub and GitHub

Skills aren't just personal. The Hermes community shares skills through two main channels.

the Skills Hub — The Skills Marketplace

the Skills Hub is the OpenClaw skills marketplace. Think of it as an app store for Hermes skills. You can browse, install, and share skills with other users.

The hermes skills command manages skill installation from registries:

```
hermes skills install skill-name
hermes skills browse
hermes skills search "keyword"
hermes skills list
```

This downloads the skill and places it in your ~/.hermes/skills/ directory, ready to use.

You can also publish your own skills to the Skills Hub for others to benefit from. If you've written a skill that other people in your organization or the broader community might find useful, consider sharing it. Good candidates for sharing: deployment skills, code review skills, documentation generators, and project scaffolding skills.

GitHub Repositories

You can also store skills in GitHub repositories. This is useful for team sharing — everyone on your team can clone the same skills repository and stay in sync.

In your Hermes configuration, you can add external skill directories:

```
skills:
  external_dirs:
    - /shared/team-skills
    - ~/projects/company-skills
```

These external directories are scanned alongside ~/.hermes/skills/ when Hermes builds its skills inventory. This means you can keep team-level skills in a shared location while personal skills live in your home directory.

The hermes skills command also supports installing skills from GitHub repositories and custom sources (via hermes skills tap), making it easy to distribute skills within a team or to the public.

Try It Now: Create, Test, and Patch a Real Skill

It's time to put everything together. In this exercise, you'll create a complete skill with a SKILL.md file, a template, and a script. Then you'll test it and patch it to improve it.

Step 1: Create the Skill

Create a skill called bug-report that helps Hermes write consistent, thorough bug reports. Here's the SKILL.md content:

```
skill_manage(
    action='create',
    name='bug-report',
    content='''---
name: bug-report
description: Write structured, thorough bug reports
category: coding
trigger: When user asks to write a bug report or report a bug
---

# Bug Report Writer

## Context
Write bug reports that give developers everything they need to
understand, reproduce, and fix the issue.

## Steps

1. **Gather the basics** — Ask the user for: what happened, what
```

```
they expected, and when it started.

2. **Identify reproduction steps** — Work with the user to create step-by-step instructions that reliably trigger the bug. Be specific: include exact URLs, button clicks, and input values.

3. **Check environment details** — Record the relevant environment: browser version, OS, app version, and any feature flags.

4. **Assess severity** — Classify as Critical (data loss, security), High (feature broken for many users), Medium (feature degraded), or Low (cosmetic or edge case).

5. **Search for duplicates** — Before writing the report, check if this bug has already been reported. Search open issues for similar descriptions.

6. **Write the report** — Format using the template in templates/bug-report-template.md. Fill in all sections with the gathered information.

7. **Review** — Read through the complete report before presenting it to the user. Make sure it is clear, actionable, and contains no assumptions presented as facts.
'''
)
```

Step 2: Add a Template

Now add a template file that defines the bug report format:

```
skill_manage(
    action='write_file',
    name='bug-report',
    file_path='templates/bug-report-template.md',
    content='''## Bug: {TITLE}

**Severity:** {SEVERITY}

### Expected Behavior
{EXPECTED}

### Actual Behavior
{ACTUAL}

### Reproduction Steps
1. {STEP_1}
2. {STEP_2}
3. ...
```

```
### Environment
- Browser/Client: {CLIENT}
- OS: {OS}
- App Version: {VERSION}
- Feature Flags: {FLAGS}

### Additional Context
{ADDITIONAL_CONTEXT}
'''
)
```

Step 3: Add a Script

Add a script that checks for duplicate issues in a GitHub repository:

```
skill_manage(
    action='write_file',
    name='bug-report',
    file_path='scripts/search-issues.sh',
    content='''#!/bin/bash
# Search GitHub issues for potential duplicates
# Usage: ./scripts/search-issues.sh "search query" "owner/repo"

QUERY="$1"
REPO="$2"

if [ -z "$QUERY" ] || [ -z "$REPO" ]; then
    echo "Usage: search-issues.sh <query> <owner/repo>"
    exit 1
fi

gh issue list --repo "$REPO" --search "$QUERY" --state open --
limit 10
'''
)
```

Step 4: Verify Your Skill

Check that everything was created correctly:

```
skills_list()
```

You should see:

```
Available Skills:
- bug-report: Write structured, thorough bug reports
```

Now load the full skill:

```
skill_view('bug-report')
```

This should return the complete SKILL.md content plus a linked_files dictionary showing your template and script.

Step 5: Test Your Skill

Ask Hermes to write a bug report:

```
Me: Help me write a bug report. The login page redirects to a
blank screen when I use Safari.
```

Hermes should: 1. Match this request to the bug-report skill 2. Load the skill with skill_view('bug-report') 3. Follow the steps — asking you for details, checking for duplicates, and formatting the report 4. Use the template to structure the output

Step 6: Patch Your Skill

After testing, you might find that the skill doesn't ask about error messages. Let's add that. Patch step 1 to include gathering error messages:

```
skill_manage(
    action='patch',
    name='bug-report',
    old_string='1. **Gather the basics** — Ask the user for: what
happened, what they expected, and when it started.',
    new_string='1. **Gather the basics** — Ask the user for: what
happened, what they expected, when it started, and any error
messages they saw (check the browser console or terminal output).'
)
```

This is a small, surgical change. The rest of the skill remains untouched.

Step 7: Add Another Improvement

Let's also add a step about taking screenshots. Patch the skill to add a new step after reproduction steps:

```
skill_manage(
    action='patch',
    name='bug-report',
    old_string='4. **Assess severity**',
    new_string='4. **Capture evidence** — Ask the user for
screenshots, screen recordings, or console logs that show the bug.
Visual evidence helps developers understand issues faster than
```

```
text descriptions alone.\n\n5. **Assess severity**'
)
```

Notice that we also need to renumber the subsequent steps (4 becomes 5, 5 becomes 6, etc.). We could do that with additional patches, or we could accept that Hermes is smart enough to follow slightly irregular numbering. In practice, though, keeping your skill well-organized is worth the effort. You could do a full edit to renumber, or you could use patches to fix each step number.

Recap

You just created a skill with a SKILL.md, a template, and a script. You tested it, found a gap, and patched it — twice. That's the real skill lifecycle: create, test, patch, repeat.

Here's your checklist for skill mastery:

- [ ] Know when to create a skill (3+ repeatable steps)
- [ ] Write clear YAML frontmatter (name, description, category, trigger)
- [ ] Write specific, numbered steps in the body
- [ ] Add supporting files (references, templates, scripts, assets) as needed
- [ ] Test immediately after creating
- [ ] Patch for small fixes, edit for major overhauls
- [ ] Remove obsolete files when cleaning up
- [ ] Keep skills short (under 10 steps) — split or trim if they grow too long
- [ ] Don't wait to create a skill if you've repeated a workflow twice

- [ ] Let Hermes auto-patch when skills break from environmental changes

Skills are Hermes's procedural memory — the how-to knowledge that makes it consistent, reliable, and efficient. By investing a few minutes to create a skill, you save yourself from repeatedly explaining the same processes. And as your collection of skills grows, Hermes becomes a more and more powerful assistant tailored to your specific workflows.

In the next chapter, we'll explore the Hermes gateway — how to connect your agent to Telegram, Discord, Slack, and a dozen other platforms so it can meet you wherever you already are.

Chapter 9: Browser Power — Hermes on the Web

9.1 Why Hermes Has a Browser — The Missing Superpower

Let me start with a confession: for years, I thought AI agents that could "browse the web" were a gimmick. I'd seen chatbots that could fetch a webpage and summarize it. I'd seen APIs that could scrape headlines. But actually *using* a browser — clicking buttons, filling out forms, scrolling through content, checking that a page loaded correctly — that felt like science fiction.

Then I started working with Hermes, and everything changed.

Here's the fundamental shift: most AI agents can only *read* text. You give them a URL, they fetch the HTML, they strip out the tags, and they hand you a blob of words. That's useful, but it's like reading about a restaurant instead of eating there. You know the menu, but you can't taste the food.

Hermes doesn't just read about the web. Hermes *acts* on the web.

The browser tools in Hermes aren't an afterthought or a plugin someone bolted on over the weekend. They're first-class tools — ten distinct capabilities that give Hermes the same kind of web interaction you have when you sit down at your laptop, open Firefox or Chrome,

and start clicking around. The difference is that Hermes can do it at scale, without getting bored, and without accidentally closing the tab when reaching for your coffee.

Let me paint some pictures of what this actually means in practice:

Research at depth. Instead of asking Hermes to "summarize this article" and getting a surface-level paraphrase, you can have Hermes navigate to a research portal, run a search query, scroll through results, click into the relevant papers, extract the figures, and compile a synthesis — all through direct interaction with the live page.

Form-filling and data entry. Need to submit information to a web application that doesn't have an API? Hermes can navigate to the form, type into each field using reference IDs, press Enter to submit, and then verify the result. This isn't theoretical — it's a daily workflow for people automating insurance claims, HR processes, and customer service tasks.

Testing and QA. If you're building a web application, Hermes can navigate to your staging site, click through critical user flows, check the console for JavaScript errors, and report back. It's like having a tester who never misses a regression and never asks for a raise.

Monitoring. Set up Hermes to periodically visit a competitor's pricing page, take a snapshot, and alert you when something changes. It's web scraping, but done through the actual browser — which means it handles dynamic content, JavaScript rendering, and all the messy stuff that makes simple scrapers fall apart.

The web browser is, for most of us, the single most important application on our computers. It's where we read, shop, work, communicate, and entertain ourselves. Giving Hermes the ability to operate a browser isn't just adding a feature — it's giving the agent access to the same digital world you live in every day.

That's the superpower. Now let's learn how to use it.

9.2 Navigating — Your First Web Visit

I'm going to break my own rule from earlier chapters and show you the most important thing first. Here's what it looks like when Hermes visits a webpage:

```
> browser_navigate(url="https://example.com")
```

```
Result:
Page loaded: https://example.com
Title: Example Domain

[heading] [@e1] "Example Domain"
[paragraph] This domain is for use in illustrative examples
            in documents. You may use this domain in literature
            without prior coordination or asking for permission.
[link] [@e2] "More information..." → https://www.iana.org/domains/
example
```

Let's break down what just happened. browser_navigate(url) is the gateway tool — the front door. You call it with a URL, and Hermes opens that page in a real browser, renders it fully (including any JavaScript), and returns what's called a **compact snapshot**.

A compact snapshot is Hermes's way of showing you what's on the page without overwhelming you. It focuses on interactive elements — the things you can click, type into, or otherwise engage with. And critically, it assigns each interactive element a **reference ID** — you can see them in the output as @e1, @e2, and so on. These ref IDs are your handles for interacting with the page. Think of them as the names on the buttons and links that Hermes can press.

The compact snapshot is the default. It's what you get automatically when you navigate, and it's what you get when you call browser_snapshot() without changing any parameters. It's concise, it's focused on what you can *do*, and it's almost always the right place to start.

But sometimes you need more. Sometimes the compact snapshot doesn't give you enough detail about the actual *content* of the page — the paragraphs, the data, the text that isn't wrapped in an interactive element. For that, you use the full snapshot:

```
> browser_snapshot(full=true)

Result:
[heading] [@e1] "Example Domain"
[paragraph] This domain is for use in illustrative examples
            in documents. You may use this domain in literature
            without prior coordination or asking for permission.
[paragraph] More information about example domains can be
            found in IANA's documentation on domain naming.
[link] [@e2] "More information..." → https://www.iana.org/domains/
example
[footer] Copyright © 2016-2024 IANA
```

With full=true, you see everything — all the text content, all the structure, not just the interactive pieces. This is what you reach for when you need to *read* the page, not just *act on* it.

There's one important detail about snapshots that you need to keep in mind: if the page is really long, and the full snapshot exceeds 8,000 characters, Hermes will either truncate it or provide an AI-generated summary. This isn't a bug — it's a practical limit. Full snapshots of complex pages can be enormous, and the summary keeps things manageable. If you need specific details from a truncated page, navigate there, then use browser_snapshot(full=true) combined with browser_scroll('down') to read it in chunks.

Let me reinforce the most important rule of the browser tools, because violating it will cause you confusion:

You must call browser_navigate(url) before any other browser tool.

This isn't optional. It's not a suggestion. Hermes needs an active browser session — a page loaded in memory — before it can take snapshots, click elements, type text, or anything else. If you try to call browser_click or browser_snapshot without first navigating somewhere, you'll get an error. The browser is a state machine, and it starts in the "no page loaded" state. browser_navigate is how you transition it to "ready."

Think of it like starting your car. You wouldn't try to steer before turning the key, right? Same thing here. Navigate first, then interact.

9.3 Reading Pages — Snapshots and Scrolling

Now that you know how to get to a page, let's talk about how to *read* it thoroughly.

The compact snapshot you get from browser_navigate is a good starting point. It tells you the page title, the interactive elements, and enough context to decide what to do next. But realistically, when you're doing research or trying to extract specific information, you need the full picture.

Here's the workflow I use almost every day:

Step 1: Navigate and get the compact snapshot.

```
> browser_navigate(url="https://news.ycombinator.com")

Result:
Page loaded: https://news.ycombinator.com
Title: Hacker News
[link] [@e1] "Hacker News" → /
[link] [@e2] "new" → /newest
[link] [@e3] "past" → /front
[link] [@e4] "comments" → /newcomments
[link] [@e5] "ask" → /ask
...
```

This gives me the landscape — what kind of site am I on, what can I interact with, what's the general structure.

Step 2: Get the full snapshot for the content.

```
> browser_snapshot(full=true)

Result:
[heading] "Hacker News"
[link] [@e1] ...
[story] [@e6] "Why SQLite is Great for Small Projects"
           142 points | 67 comments | posted by user123
[story] [@e7] "The Architecture of Modern Web Browsers"
           89 points | 34 comments | posted by devfan
...
```

Now I can see the actual stories, their point counts, who posted them. This is the level of detail that lets me decide if I want to click through and read more.

Step 3: Scroll when content extends beyond the viewport.

```
> browser_scroll(direction='down')

Result:
Page scrolled down. New content visible.
[story] [@e28] "Building a Real-Time Dashboard with WebSockets"
           56 points | 22 comments
...
```

Pages don't always fit in one screen. browser_scroll(direction='down') reveals more content, just like scrolling down on your phone or laptop. You can also scroll back up with browser_scroll(direction='up').

Step 4: Find images when you need visual content.

```
> browser_get_images()

Result:
```

```
Image 1: https://news.ycombinator.com/y18.gif
  Alt text: "Hacker News logo"
  Dimensions: 18x18
Image 2: https://news.ycombinator.com/s.gif
  Alt text: (no alt text)
  Dimensions: 1x1
```

browser_get_images() returns all the images on the current page with their URLs and alt text. This is incredibly useful when the information you need is in an image — a chart, a diagram, a screenshot — rather than in the text. You get the URLs, and you can then feed those into Hermes's vision capabilities or other analysis tools.

Let me share a practical example. A while back, I was researching a technical topic and needed to read a long-form article. The compact snapshot showed me the headings and links, but the actual content — the arguments, the data, the nuance — was in the paragraphs between. I navigated to the page, took a full snapshot, scrolled down to read the second half, and used browser_get_images() to find the embedded charts that the text kept referring to. Without the full snapshot and scrolling, I would have missed half the article. Without getting the images, I would have been confused about what the author was arguing.

One more thing about the 8,000-character truncation limit: it's not something to fear, but it is something to plan for. When you encounter a truncated snapshot, don't panic. You have two strategies:

1. **Scroll and re-snapshot.** Navigate to the page, take the full snapshot, then scroll down and take another full snapshot. Read the page in chunks.

2. **Use the AI summary.** Sometimes the truncated snapshot includes an AI-generated summary of the remaining content. This is usually good enough to decide whether you need the full details. If you do, go back to strategy 1.

The bottom line: snapshots are your eyes on the web. Compact snapshots for quick orientation, full snapshots for deep reading, scroll to see more, and get images when visual content matters.

9.4 Clicking and Typing — Interacting with Pages

Reading pages is great, but the real power of the browser tools is *interaction*. Let me show you a complete interaction flow from start to finish:

```
> browser_navigate(url="https://example-search-site.com")

Result:
Page loaded: https://example-search-site.com
[heading] [@e1] "Search the Knowledge Base"
[search input] [@e2] placeholder="Enter your search..."
[button] [@e3] "Search"
[link] [@e4] "Advanced Search" → /advanced
```

I've navigated to a page with a search bar. I can see the search input field at @e2 and the Search button at @e3. Time to interact.

Step 1: Type into the search field.

```
> browser_type(ref="@e2", text="browser automation")

Result:
Typed "browser automation" into [@e2].
The field now contains: "browser automation"
```

Notice something important: browser_type(ref, text) *clears the field first*, then types your new text. This is a design choice that prevents a whole class of bugs. If the field already had text in it — say "Search..." from the placeholder — you don't have to worry about selecting and deleting it first. Hermes handles that for you. Just type what you want, and it replaces whatever was there.

Step 2: Click the Search button.

```
> browser_click(ref="@e3")

Result:
Clicked [@e3] "Search". Page is loading...
```

Step 3: Check the result.

```
> browser_snapshot()

Result:
Page loaded: Search results for "browser automation"
[heading] [@e5] "Search Results"
[result] [@e6] "Introduction to Browser Automation" → /articles/
```

```
intro
[result] [@e7] "Puppeteer vs Selenium: A Comparison" → /articles/
comparison
[result] [@e8] "Advanced Browser Automation Techniques" → /
articles/advanced
```

And there are my results. I can click any of them to read further, or I can take a full snapshot to see the complete content.

This navigate-snapshot-interact-snapshot cycle is the fundamental loop of browser automation with Hermes. You see the page, you decide what to do, you take an action, you check what happened. It's exactly how you browse the web manually — you're just teaching Hermes to do the same thing.

Let me cover the other interaction tools:

browser_press(key) — Sometimes you need to press a keyboard key instead of clicking a button. The most common example is pressing Enter to submit a form instead of clicking the submit button:

```
> browser_type(ref="@e2", text="browser automation")
> browser_press(key="Enter")
```

This does the same thing as clicking the Search button in most cases. Other useful keys include:

- 'Tab' — Move between form fields
- 'Escape' — Close popups, cancel actions
- 'ArrowDown' / 'ArrowUp' — Scroll or navigate dropdown menus

browser_back() — Navigate to the previous page, just like hitting the Back button in your browser:

```
> browser_back()

Result:
Navigated back to: https://example-search-site.com
```

This is incredibly useful when you click into an article, read it, and then want to go back to the results page. It's faster than navigating to the URL again, and it preserves your scroll position and form state.

Let me put it all together with a realistic form-filling example:

```
> browser_navigate(url="https://demo-form-site.com/contact")

Result:
```

```
[heading] [@e1] "Contact Us"
[input] [@e2] placeholder="Your Name"
[input] [@e3] placeholder="Your Email"
[textarea] [@e4] placeholder="Your Message"
[button] [@e5] "Submit"
> browser_type(ref="@e2", text="Jane Developer")
> browser_type(ref="@e3", text="jane@example.com")
> browser_type(ref="@e4", text="I'm interested in browser
automation tools for my team.")
> browser_click(ref="@e5")

Result:
Form submitted! Thank you for your message.
```

Four actions. That's all it takes to fill out and submit a form. No scraping, no API wrangling, no reverse-engineering the POST request. Just navigate, type, click, done.

The ref IDs are your lifeline here. Every snapshot shows them, and every interaction tool uses them. @e2 is "Your Name," @e3 is "Your Email," and so on. The numbers are assigned by Hermes based on the page structure, so they'll be different on every page and they'll change if the page structure changes. Always take a fresh snapshot before interacting if you're not sure the page is the same.

And yes, I need to say this explicitly: **the ref ID must match the format shown in the snapshot**. If the snapshot shows @e5, you use @e5. Not e5, not 5, not #e5. "@e5" — the at sign, the letter e, the number. Get this wrong and Hermes will tell you the element doesn't exist.

9.5 Seeing with AI — Vision Mode

Everything we've talked about so far has been text-based. Snapshots are text. Clicking uses text-based reference IDs. Typing is text. This works fabulously for most web pages because, at their core, web pages are structured text — HTML documents with interactive elements.

But the web is also visual. Pages have layouts, color schemes, images, charts, and visual designs that pure text can't capture. Sometimes you need to *see* what's on the page, not just read about it.

That's where browser_vision(question, annotate=false) comes in.

Let me show you how it works before I explain it:

```
> browser_navigate(url="https://store.example.com")

> browser_vision(question="What products are featured on this
page?")

Result:
AI Analysis: The page shows a clean e-commerce layout with a
navigation bar
at the top. The main content area features three product cards
arranged
horizontally: a wireless headphone set ($79.99), a mechanical
keyboard
($129.99), and an ergonomic mouse ($49.99). Each card has an "Add
to Cart"
button. There's a banner at the top advertising a 20% discount on
electronics.

Screenshot saved to: /tmp/hermes_browser_screenshot_20260416.png
```

There are two things happening here. First, Hermes takes a screenshot of the current page — an actual image file, like a PNG. Second, it sends that screenshot to a vision AI model along with your question, and returns the AI's analysis.

The question parameter is what you're asking the vision AI to focus on. It's not a required parameter — if you don't provide one, the AI will give you a general description of the page. But providing a question helps the AI be specific and relevant. "What products are featured?" is better than no question. "Is the checkout button visible?" is better than "Describe the page."

The annotate flag. One of the most powerful features of browser_vision is the annotate parameter:

```
> browser_vision(question="Where are the interactive elements?",
annotate=true)

Result:
AI Analysis: The page has interactive elements at the following
locations:
[1] Navigation menu - "Home", "Products", "About", "Contact"
[2] Search bar in the top right
[3] "Add to Cart" button on each product card (3 total)
[4] "Sign In" link in the header

Screenshot saved to: /tmp/
hermes_browser_screenshot_annotated_20260416.png
```

When annotate=true, Hermes overlays numbered labels [1], [2], [3], etc. directly onto the screenshot, and these numbers map to the reference IDs you see in snapshots. So [3] on the screenshot corresponds to @e3 in the snapshot. This creates a visual bridge between what you *see* and what you can *click*. If you're not sure which ref ID corresponds to which element on the page, annotate your vision call and match them up.

When to use vision vs. snapshot. Here's my decision framework:

- **Use snapshot first.** Always. The snapshot is faster, cheaper, and gives you the structural information you need for most tasks.
- **Use vision when:** the page relies on visual layout (dashboards, infographics, image-heavy sites), when you need to verify that something *looks* right (browser testing, layout checks), or when the snapshot doesn't give you enough context to understand what's happening.
- **Don't use vision for:** reading text content, clicking standard form elements, navigating between pages. The snapshot handles all of this just fine.

I mention cost because browser_vision is expensive. Every call takes a screenshot and sends it to a vision AI model for analysis. That's a real GPU operation, and it costs real compute resources. If you can answer your question with a snapshot, use the snapshot. Save vision for when you genuinely need to see the page.

The screenshot_path output you see in the results — that's where the actual image file is saved on your system. You can open it, share it, or feed it into other analysis steps. It's a real file on disk, and it persists after the call completes.

9.6 Debugging — Console and JavaScript

So far, we've been using the browser as a *user* — navigating, reading, clicking, typing. But Hermes also gives you access to the browser's *developer tools*, specifically the JavaScript console. This is where the browser tells you what's happening under the hood, and it's incredibly valuable for debugging.

Let me start with the simplest use:

```
> browser_console()
```

```
Result:
Console output:
  [log] "Application initialized"
  [log] "User session loaded: default"
  [warn] "Deprecated API call: getUserMedia — use
navigator.mediaDevices instead"
  [error] "Uncaught TypeError: Cannot read properties of null
(reading 'classList')"
    at main.js:142
```

browser_console() with no arguments returns all the console output from the current page — every console.log, console.warn, and console.error call, plus any uncaught JavaScript exceptions. It's the same thing you'd see if you opened the DevTools console tab in Chrome or Firefox.

This is pure gold for debugging. If a page isn't behaving correctly, check the console. JavaScript errors will show up here, and they'll often tell you exactly what went wrong and where. In the output above, there's a TypeError at line 142 of main.js — someone's trying to read classList from a null reference. That's a real bug, and now you know exactly where to look.

But browser_console is more than a passive log reader. You can also evaluate JavaScript expressions directly in the page context:

```
> browser_console(expression="document.title")

Result:
"The Best Demo Site Ever"
> browser_console(expression="window.location.href")

Result:
"https://demo.example.com/page"
```

This is essentially the same as opening DevTools and typing into the console. You can inspect DOM elements, check variable values, test selectors, and verify state. The expression parameter takes any valid JavaScript, evaluates it in the page context, and returns the result.

Some practical debugging patterns I use regularly:

Check if an element exists before clicking:

```
> browser_console(expression="document.querySelector('#submit-
button') !== null")

Result:
true
```

Verify navigation happened:

```
> browser_console(expression="window.location.href")

Result:
"https://demo.example.com/success-page"
```

Count elements matching a selector:

```
> browser_console(expression="document.querySelectorAll('.result-
item').length")

Result:
23
```

Extract data that's not in the snapshot:

```
> browser_console(expression="JSON.stringify({title:
document.title, url: window.location.href, cookies:
document.cookie})")

Result:
"{\"title\":\"Dashboard\",\"url\":\"https://app.example.com/
dashboard\",\"cookies\":\"session=abc123\"}"
```

The clear parameter is worth knowing about. If you call browser_console(clear=true), it clears the console buffer — all those accumulated log messages get wiped clean. This is useful when you're debugging iteratively: clear the console, trigger an action, then read the console to see only the new messages relevant to your latest action.

```
> browser_console(clear=true)

Result:
Console cleared.

> browser_click(ref="@e5")

Result:
Clicked [@e5] "Submit Form"

> browser_console()

Result:
Console output:
  [log] "Form submission started"
  [log] "Validation passed"
  [log] "Form submitted successfully"
```

By clearing first, then acting, then reading, I get a clean signal. Only the messages from the action I just performed, not a jumbled mix of logs from the entire session.

The console tool is your window into the page's internals. Use it when things go wrong, use it to verify things went right, and use it to extract information that the snapshot doesn't provide. It's one of the most underrated browser tools in Hermes, and I suspect that's because it feels "developer-y." Don't let that stop you. You don't need to be a JavaScript expert to read an error message or check a URL.

9.7 Browser Configuration — Timeouts and Privacy

Before you go wild with browser automation, you need to understand the configuration that governs how Hermes's browser behaves. These settings aren't just knobs to twist — they're guardrails that keep your automation sessions from spiraling out of control.

Here's the full configuration block:

```
browser:
  inactivity_timeout: 120
  command_timeout: 30
  record_sessions: false
  allow_private_urls: false
  camofox:
    managed_persistence: false
```

Let me walk through each one.

inactivity_timeout: 120 — This is the two-minute kill switch. If Hermes's browser doesn't receive any command for 120 seconds (2 minutes), the session is automatically terminated. The browser closes, the page state is lost, and you'd need to navigate again to start a new session.

Why does this exist? Because browser sessions consume resources — memory, compute, potentially network connections. If you walk away from your automation task and forget about it, the inactivity timeout ensures the session doesn't sit there indefinitely, burning resources.

What this means for you: if you're building a workflow that involves the browser, make sure it stays active. If you have a long chain of steps that takes more than 2 minutes of "thinking" time between browser actions, you'll lose your session. Either keep the actions moving, or structure your workflow so the browser steps happen in a quick burst.

command_timeout: 30 — Each individual browser action has a 30-second timeout. If a browser_navigate call takes more than 30 seconds to load a page, it times out. If a browser_click doesn't produce a response in 30 seconds, it times out.

This is your per-action safety net. Pages that take forever to load, infinite redirects, and hanging JavaScript all get caught by this timeout. When a timeout fires, you get an error, and you can decide whether to retry, skip, or investigate.

record_sessions: false — When set to true, Hermes records the entire browser session — every navigation, click, scroll, and snapshot — for later replay. This is an incredibly powerful debugging tool. If your automation does something unexpected, you can replay the session to see exactly what happened, in order, with full context.

The default is false because recording adds overhead and stores data. Turn it on when you're developing or debugging, and consider turning it off for production.

allow_private_urls: false — This is the safety gate. When false, Hermes blocks navigation to private URLs — anything on localhost, 127.0.0.1, private IP ranges (10.x.x.x, 172.16.x.x, 192.168.x.x), and internal network addresses. This prevents a malicious or misconfigured prompt from directing Hermes to access services on your internal network.

When would you set this to true? When you're developing locally and need Hermes to interact with a web application running on your own machine. That's a legitimate use case — automated testing of your local dev server, for example. Just be aware that you're opening up the browser to access your local network, so make sure you trust the prompts you're giving Hermes.

camofox.managed_persistence: false — Camofox is Hermes's anti-detection layer — it's what makes the browser look like a real user rather than an automated bot. Websites increasingly use fingerprinting techniques to detect headless browsers and automation tools. Camofox helps Hermes navigate around those protections.

The managed_persistence setting controls whether Camofox's browser fingerprints persist between sessions. When false, each new browser session gets a fresh fingerprint. When true, the same fingerprint (including cookies, cache, and browser profile data) carries over between sessions, making Hermes look even more like a returning human user.

The default of false is the safe choice — it means no data leaks between sessions, no risk of one session's cookies affecting another. Set it to true if you need Hermes to maintain a consistent identity across sessions (for example, staying logged into a website between visits).

These configuration settings live in your Hermes configuration file, and you should understand what each one does before changing it. The defaults are conservative and safe. Change them only when you have a specific reason and you understand the implications.

9.8 My Browser Mistakes — Automation Gone Wrong

I've been teaching you the tools, the workflows, and the configurations. Now it's time for the part of the chapter where I admit that I've made every mistake in the book — and a few that weren't in the book yet — so you can learn from my failures instead of repeating them.

The Infinite Scroll Trap

I was building an automation to extract all the articles from a blog. The blog used infinite scrolling — you scroll down, more articles load, you scroll down again, even more articles load. Simple enough, right?

My script looked like this:

```
> browser_navigate(url="https://infinite-blog.example.com")
> browser_snapshot(full=true)
  ... got articles 1-20 ...
> browser_scroll(direction='down')
> browser_snapshot(full=true)
  ... got articles 21-40 ...
> browser_scroll(direction='down')
> browser_snapshot(full=true)
  ... got articles 41-60 ...
```

And I kept going. And going. And going. Scroll, snapshot, scroll, snapshot, scroll, snapshot. The blog had over 500 articles, and I was determined to get every single one.

Here's what went wrong: the 8,000-character truncation limit started cutting off my snapshots around article 150, meaning I was getting incomplete data. My scroll-and-snapshot loop took so long that I hit the inactivity timeout between steps — my processing code was slow, and each pause gave the timeout timer more time to tick. And the blog's infinite scroll implementation had a quirk where it would occasionally reload articles I'd already seen, meaning I was getting duplicates without realizing it.

The fix was humblingly simple. I should have:

1. Checked the total number of articles upfront (many sites expose this in a sitemap or API endpoint).
2. Used browser_console(expression='...') to programmatically count the loaded articles, rather than relying on truncated snapshots.
3. Set a reasonable limit. "All articles" isn't always necessary. "The most recent 50" might be enough.
4. Been faster between actions to avoid the inactivity timeout.

Lesson learned: infinite scrolling is a trap for automation. Always have an exit condition, and don't assume that "just keep scrolling" is a viable strategy.

The CAPTCHA That Ate My Session

I needed to log into a website that had a CAPTCHA on its login page. No problem, I thought — I'll navigate to the page, type in the credentials, and use browser_vision to "read" the CAPTCHA and solve it.

```
> browser_navigate(url="https://protected-site.example.com/login")
> browser_type(ref="@e2", text="my_username")
> browser_type(ref="@e3", text="my_password")
> browser_vision(question="What does the CAPTCHA say?")
```

The vision AI looked at the CAPTCHA image and... couldn't solve it. It was one of those distorted-text CAPTCHAs designed specifically to be hard for machines to read. Which, you know, is the whole point of CAPTCHAs.

But the real problem came next. I tried to type in what the AI thought the text was:

```
> browser_type(ref="@e4", text="XK7P2M")
> browser_click(ref="@e5")  // "Submit" button
```

The login failed. The CAPTCHA regenerated — new text, new image. I tried again with browser_vision. Failed again. Each failed attempt regenerated the CAPTCHA. After five attempts, the site locked the account for 30 minutes. My automation was now locked out, and so was I, for half an hour.

This was a fundamental misunderstanding of what CAPTCHAs are designed to prevent. They're explicitly built to stop automated systems from getting through. Using an AI agent to solve a CAPTCHA is like using a hammer to fix a problem that specifically requires "not a hammer."

The real solution: for websites you control or have API access to, bypass the CAPTCHA entirely by using an API key or authentication token. For websites you don't control, you'll need a human in the loop for the CAPTCHA step. Some production setups use CAPTCHA-solving services, but those are external tools and beyond the scope of what Hermes's browser can do natively.

Lesson learned: CAPTCHAs are the web's way of saying "humans only." Respect the sign.

The "Delete All" Button Incident

This one still makes me wince. I was testing an automation on a content management system. I needed to clean up a test user's draft posts. The page had a list of posts with action buttons next to each one. My plan was to click the "Delete" button next to each draft.

I navigated to the page, took a snapshot, and saw:

```
[link] [@e4] "Edit" → /posts/12/edit
[button] [@e5] "Delete"
[link] [@e6] "Edit" → /posts/13/edit
[button] [@e7] "Delete"
...
```

Great, I thought. I'll loop through and click each Delete button. But I didn't account for what happened after each click: the page reloaded, the remaining buttons got *re-numbered*, and the button that was previously "@e7" was now "@e6" because the post above it had been deleted.

So when I clicked @e6 — thinking I was deleting the second post — I was actually deleting what had been the third post, now shifted up. And worse, after three clicks, the page reloaded with a different layout, and @e5 now pointed to… the "Delete All" button at the bottom of the page. I clicked it. Every draft in the system. Gone.

Fortunately, this was a test environment. But it could easily have been production. And the mistake was entirely mine: I assumed the ref IDs would stay stable between page reloads, and I didn't re-snapshot after each action.

The correct pattern is:

1. Take a snapshot
2. Identify the target
3. Click it
4. **Re-snapshot immediately**
5. Identify the next target with the *new* ref IDs
6. Repeat

Every click or navigation can change the page. Ref IDs are not permanent. They're generated for the current state of the page. Treat them like they expire after every action — because they effectively do.

Lesson learned: always re-snapshot after interacting. Never cache ref IDs across actions.

The Common Thread

What do all three mistakes have in common? I was treating the browser like a static document when it's actually a dynamic, stateful application. Infinite scroll changes content as you interact with it. CAPTCHAs are designed to resist automation. Page reloads reassign element IDs. The browser is alive, and you need to respect its liveness.

The practical rules I follow now:

1. **Always have an exit condition.** Never loop without a limit.
2. **Re-snapshot after every meaningful action.** Ref IDs can change.
3. **Don't fight CAPTCHAs.** Use APIs or human escalation.
4. **Respect timeouts.** If your workflow pauses, the session will die.
5. **Verify before destructive actions.** Take a snapshot, confirm the target, then click.

These rules haven't eliminated all mistakes, but they've reduced the "oh no" moments by about 90%.

9.9 Putting It All Together — A Complete Workflow

Let me walk you through a realistic, end-to-end browser automation task that uses every tool we've discussed. The scenario: I want to check the current status of a project on a project management website, navigate to a specific task, and verify it's been completed correctly.

Step 1: Navigate to the site.

```
> browser_navigate(url="https://projects.example.com")

Result:
Page loaded: https://projects.example.com
Title: Project Dashboard
[link] [@e1] "My Projects" → /projects
[link] [@e2] "Team Activity" → /activity
[link] [@e3] "Settings" → /settings
```

Step 2: Get a better view of the content.

```
> browser_snapshot(full=true)

Result:
[heading] "Project Dashboard"
[paragraph] Welcome back, Jane. You have 3 active projects.
[link] [@e1] "My Projects" → /projects
[card] "Website Redesign" — 78% complete — 3 tasks remaining
[card] "Mobile App v2" — 45% complete — 12 tasks remaining
[card] "API Migration" — 100% complete ✓
```

Step 3: Navigate to the project I care about.

I can see “Website Redesign” at 78% complete. Let me click into it. But first, I need a link to click… and the cards don’t have obvious ref IDs. Let me check with vision.

```
> browser_vision(question="How do I navigate to the Website
Redesign project?", annotate=true)

Result:
AI Analysis: The "Website Redesign" project card is in the center
of the page.
Click on [1] to navigate to the projects list, or click directly
on the card
which appears to correspond to ref @e4 based on the annotation
overlay.

Screenshot saved to: /tmp/
hermes_browser_screenshot_annotated_20260416.png
```

Vision tells me I can click @e4 to go to the project. Let me do it.

```
> browser_click(ref="@e4")

Result:
Navigated to: Website Redesign project page
```

Step 4: Check the project page.

```
> browser_snapshot(full=true)

Result:
[heading] "Website Redesign"
[paragraph] Status: In Progress | Deadline: April 30, 2026
[section] "Remaining Tasks"
  [link] [@e5] "Fix mobile navigation" → /tasks/142
  [link] [@e6] "Update footer links" → /tasks/195
  [link] [@e7] "Optimize images" → /tasks/203
[section] "Recently Completed"
  [link] [@e8] "Homepage hero redesign" → /tasks/89
```

Step 5: Click into a task.

```
> browser_click(ref="@e5")

Result:
Navigated to: Task 142 — Fix mobile navigation
```

Step 6: Read the task details.

```
> browser_snapshot(full=true)
```

```
Result:
[heading] "Fix mobile navigation"
[paragraph] Priority: High | Assignee: Mike S. | Status: In
Progress
[paragraph] The mobile navigation menu is not closing properly
when a submenu item is selected. This causes layout issues on
iOS Safari and Android Chrome.
[textarea] [@e8] "Add comment..."
[button] [@e9] "Submit Comment"
[button] [@e10] "Mark Complete"
```

Step 7: Check for JavaScript errors on this page.

```
> browser_console()

Result:
Console output:
  [log] "Task detail view initialized"
  [log] "Comments loaded: 4"
  [warn] "WebSocket connection failed — falling back to polling"
```

A warning about WebSocket — not critical, but worth noting.

Step 8: Go back to the project page.

```
> browser_back()

Result:
Navigated back to: Website Redesign project page
```

Step 9: Get images for visual context.

```
> browser_get_images()

Result:
Image 1: https://projects.example.com/assets/logo.svg
  Alt text: "Project Manager logo"
Image 2: https://projects.example.com/uploads/website-redesign-
mockup.png
  Alt text: "Current design mockup"
```

I could use browser_vision to analyze that mockup image for visual details about the design.

Step 10: Verify I'm on the right page.

```
> browser_console(expression="window.location.href")

Result:
"https://projects.example.com/projects/website-redesign"
```

Everything checks out. I've navigated, read, clicked, scrolled, used vision, checked the console, gone back, and inspected images — all in one coherent workflow. And every step used exactly one browser tool, with clear results that informed the next action.

This is the power of Hermes's browser: it's not one trick. It's a complete toolkit for web interaction, and each tool does exactly one thing well. The art is in combining them into smooth, purposeful sequences.

9.10 Hands-On Exercise — Your First Browser Automation

It's time to get your hands dirty. This exercise walks you through a complete browser automation session, using every major tool we've covered. Follow along step by step.

Try It Now: Navigate, Snapshot, Click, Vision, and Console

Exercise Goal: Visit a real website, read its content, interact with an element, use vision to understand the visual layout, and check the console for technical details.

Step 1: Navigate to example.com

Open your Hermes environment and run:

```
browser_navigate(url="https://example.com")
```

Examine the output. You should see the page title, some text content, and at least one link with a reference ID.

Question for you: What ref ID is assigned to the "More information..." link? (Write it down — you'll need it in Step 3.)

Step 2: Take a full snapshot

```
browser_snapshot(full=true)
```

Compare the full snapshot to the compact one from Step 1. What additional information do you see? Notice how full=true gives you all the text content, not just the interactive elements.

Step 3: Click the link

Using the ref ID you noted in Step 1, click the “More information…” link:

```
browser_click(ref="@e2")
```

(Your ref ID might be different — use the one from YOUR snapshot, not this example.)

You should be navigated to a new page. What’s the title of the new page?

Step 4: Go back

```
browser_back()
```

You should be back on the original example.com page. Take a snapshot to confirm:

```
browser_snapshot()
```

Are you back where you started? Good.

Step 5: Use vision to see the page

```
browser_vision(question="Describe the visual layout of this page", annotate=true)
```

Review the AI analysis. Notice the annotate=true flag — you should see numbered labels on the screenshot that correspond to the ref IDs in the snapshot. Compare the visual output to the text snapshot. Which gives you a better understanding of the page?

Also note the screenshot_path in the output. Open that file on your computer to see the actual screenshot with annotations.

Step 6: Check the console

```
browser_console()
```

What console output do you see? Even on simple pages, you might find log messages or warnings. A clean console is a good sign.

Step 7: Evaluate some JavaScript

```
browser_console(expression="document.title")
browser_console(expression="window.location.href")
```

These should return values you can verify against what you already know about the page.

Step 8: Try scrolling (if the page has enough content)

```
browser_scroll(direction='down')
```

On example.com, the page is short, so you might not see much new content. But the command itself should execute without errors.

Step 9: Get images on the page

```
browser_get_images()
```

What images are listed? Check the URLs and alt text. On example.com, the image list will be short — but on a real website, this can reveal charts, photos, and diagrams you might want to analyze further.

Step 10: Scroll back up and take a final snapshot

```
browser_scroll(direction='up')
browser_snapshot()
```

You're back at the top. You've successfully used all ten browser tools in a single session.

Reflection Questions

After completing the exercise, consider:

1. **When would you use browser_snapshot(full=true) vs. the default compact snapshot?** Think about the trade-off between detail and brevity.
2. **When would you choose browser_vision over browser_snapshot?** Remember the cost and the use cases — visual verification, layout checking, and situations where text doesn't tell the whole story.
3. **What would happen if you tried to click a ref ID from an old snapshot after the page changed?** (If you're not sure, re-read the "Delete All" story in section 9.8.)
4. **How would you handle a page that takes more than 30 seconds to load?** Consider the command_timeout setting and what happens when it's exceeded.
5. **Why does browser_type clear the field before typing?** Think about form-filling scenarios where fields might have default or leftover values.

Summary

In this chapter, you've learned the complete browser toolkit available in Hermes. Let's recap the ten tools and when to use each one:

Tool	Purpose	When to Use
browser_navigate(url)	Go to a URL	Always first — required before any other browser tool
browser_snapshot(full)	Read page content	Your default "look at the page" tool; full=true for reading, default for interacting
browser_click(ref)	Click an element	When you need to activate a button, link, or other interactive element
browser_type(ref, text)	Type into a field	When filling forms, entering search queries, or providing text input
browser_press(key)	Press a keyboard key	When you need Enter, Tab, Escape, or arrow keys
browser_scroll(direction)	Scroll up or down	When content extends beyond the current viewport
browser_back()	Navigate to previous page	When you need to return to the page you just came from
browser_vision(question, annotate)	Screenshot + AI analysis	When you need to visually verify layout, identify elements visually, or read images
browser_get_images()	List all page images	When you need to find or analyze visual content on the page
browser_console(expression, clear)	Read/ evaluate JS console	When debugging, checking for errors, or

Tool	Purpose	When to Use
		extracting data via JavaScript

And the key principles:

- **Navigate first.** Always call browser_navigate before any other browser tool.
- **Ref IDs are temporary.** Re-snapshot after any action that might change the page.
- **Snapshot before vision.** Use the cheaper, faster snapshot tool first. Reach for vision only when visual understanding is required.
- **Respect timeouts.** The inactivity timeout kills your session after 2 minutes of inactivity. The command timeout fails individual actions after 30 seconds.
- **Private URLs are blocked by default.** Set allow_private_urls: true only when you need local network access and understand the implications.
- **Don't fight CAPTCHAs.** Use APIs or human escalation instead.

The browser tools transform Hermes from a text-processing assistant into a full participant in the web. With these tools, your agent can research, interact, verify, and monitor — all through the same interfaces that humans use. That's not a small thing. That's the difference between reading about the world and acting in it.

In the next chapter, we'll dive into terminal tools — running commands, managing processes, reading and writing files, and searching code. The terminal is the foundation that makes Hermes a real engineering tool, not just a web-savvy chatbot.

For now, practice the exercise, experiment with different websites, and remember: every expert was once a beginner who clicked the wrong ref ID at least once. I know I did.

Chapter 10: The Terminal — Commands Made Easy

10.1 The Terminal — Your Command Line, Supercharged

Let me show you something most AI assistants cannot do:

```
> terminal("ls -la")

Result:
total 48
drwxr-xr-x  6 mike  staff   192 Apr 16 12:00 .
drwxr-xr-x  5 mike  staff   160 Apr 16 11:55 ..
-rw-r--r--  1 mike  staff  6148 Apr 16 11:55 .DS_Store
drwxr-xr-x  3 mike  staff    96 Apr 16 11:55 Documents
drwxr-xr-x  4 mike  staff   128 Apr 16 12:00 Projects
-rw-r--r--  1 mike  staff   220 Apr 16 11:55 notes.txt
```

That's not me *pretending* to run a command. That's not me describing what would happen if you opened a terminal yourself. Hermes actually ran ls -la in a real shell and brought the output back. The files exist. The permissions are real. The timestamps are accurate.

This is the single biggest thing that separates Hermes from a regular chatbot: **Hermes can run commands.** Not metaphorically. Literally. It has a terminal tool — a real command execution engine — that lets it build software, install packages, manage git repositories, spin up servers, debug scripts, and do anything else you'd normally do in a shell.

Most AI assistants can talk about code. They can write code snippets, explain algorithms, and suggest fixes. But they can't *run* the code. They can't tell you whether the fix actually works. They can't install the dependency you're missing. They can't start the server and check if it's listening on the right port.

Hermes can do all of that, because Hermes has a terminal.

The difference between talking about code and running code is the difference between a restaurant critic and a chef. A critic can tell you what a dish should taste like. A chef can actually make it and hand you a plate. When you ask Hermes to debug your Python script, it doesn't

just suggest adding a print statement — it adds the print statement, runs the script, reads the output, and tells you exactly what went wrong. That's the chef at work.

Now, I know what you're thinking: "An AI that can run arbitrary commands on my machine? That sounds terrifying." And you'd be right to be cautious. That's why Hermes has a safety net built into the approval system. When Hermes wants to run a command — especially a potentially dangerous one like deleting files, installing packages, or modifying system configuration — it asks you first. You see the exact command it plans to run, and you decide whether to allow it. The approval system catches dangerous commands before they execute. Think of it as a bouncer at the door of your operating system: Hermes can suggest anything, but nothing gets through without your say-so.

We covered the approval system in Chapter 4, and we'll go deeper into security in Chapter 12. For now, just know that every command Hermes runs is visible to you, and you're always in control.

Let's learn how to use this superpower.

10.2 Running Commands — Foreground Mode

The simplest way to run a command is also the most common. Let me show you:

```
> terminal("pwd")

Result:
/home/mike
> terminal("echo Hello, world!")

Result:
Hello, world!
> terminal("python3 --version")

Result:
Python 3.11.9
```

That's foreground mode in action. You give Hermes a command, it runs the command, and you get the output back immediately. The call returns **instantly** when the command finishes — even if you set a high timeout, you don't wait longer than the command actually takes.

The full signature looks like this:

```
terminal(command, timeout=180, workdir=None, background=False,
pty=False,
         check_interval=None, notify_on_complete=False)
```

Right now, let's focus on the three parameters you'll use most often in foreground mode: command, timeout, and workdir.

The command parameter

The command parameter is just a string — whatever you'd type at a shell prompt. You can run single commands:

```
> terminal("date")

Result:
Wed Apr 16 12:30:00 UTC 2026
```

Or compound commands with pipes, redirects, and chains:

```
> terminal("ls *.txt | wc -l")

Result:
7
> terminal("mkdir -p /tmp/hermes-demo && echo 'Created!'")

Result:
Created!
```

The shell is a real shell (bash by default), so anything that works in your terminal works here. Environment variables, pipes, subshells, glob patterns — they all work as expected.

The timeout parameter

By default, commands time out after 180 seconds (three minutes). That's plenty for most things — listing files, running quick scripts, checking git status. But some operations take longer. Building a large project from source can take five minutes. Installing a complex Python package with compiled extensions might need ten. That's when you crank up the timeout:

```
> terminal("pip install pandas", timeout=300)

Result:
Successfully installed pandas-2.2.3
```

I set timeout=300 here because installing pandas can involve compiling C extensions, and that takes time on slower machines. If the command finishes in 30 seconds, you get the result in 30 seconds — the timeout is a ceiling, not a waiting period. The command still returns instantly when it's done.

What happens if the command exceeds the timeout? Hermes terminates it and returns whatever output was produced before the cutoff. You'll see a message indicating the command timed out, which is your signal to either increase the timeout or find a way to make the command faster.

Rule of thumb: use the default timeout=180 for everyday commands. Set timeout=300 for builds and installs. For anything genuinely long-running — we'll cover that in the next section with background mode.

The workdir parameter

By default, commands run in the current working directory. But sometimes you want to run a command in a specific directory without changing your global state. That's what workdir is for:

```
> terminal("ls -la", workdir="/home/mike/Projects/my-app")

Result:
total 32
drwxr-xr-x  5 mike  staff   160 Apr 16 12:00 .
drwxr-xr-x  3 mike  staff    96 Apr 16 11:55 ..
-rw-r--r--  1 mike  staff   512 Apr 16 11:55 app.py
-rw-r--r--  1 mike  staff  128 Apr 16 11:55 requirements.txt
drwxr-xr-x  4 mike  staff   128 Apr 16 12:00 tests
```

The workdir parameter sets the working directory for that one command only. Your session's current directory doesn't change. It's the equivalent of running cd /somewhere && my_command without the side effect of actually being in /somewhere afterward.

This is especially handy when you're working across multiple projects:

```
> terminal("git status", workdir="/home/mike/Projects/api-server")

Result:
On branch main
nothing to commit, working tree clean

> terminal("git status", workdir="/home/mike/Projects/frontend")

Result:
```

```
On branch feature/login
Changes not staged for commit:
  modified: src/components/Login.tsx
```

Same command, different directories, no global state changes. Clean and predictable.

Interpreting output: stdout, stderr, and exit codes

When you run a command, you typically get back whatever the command printed to its standard output. But commands also produce standard error output and exit codes, and understanding these is crucial for debugging.

Standard output (stdout) is the normal output of a command — the stuff the command intends for you to read.

Standard error (stderr) is where commands send error messages and diagnostic information. It's separate from stdout so you can distinguish between "here's the result" and "here's a warning."

```
> terminal("ls nonexistent-file.txt")

Result:
ls: nonexistent-file.txt: No such file or directory
```

That error message comes from stderr. Hermes captures both stdout and stderr and presents them to you, so you always see the full picture.

Exit codes tell you whether a command succeeded or failed. A exit code of 0 means success. Anything else means failure, and the specific number often tells you *why* it failed. When a command fails, Hermes includes the exit code in the result so you can diagnose what went wrong.

Understanding these three channels — stdout, stderr, and exit codes — is your foundation for debugging any command. When something doesn't work, check: Did the command produce an error on stderr? What was the exit code? These two pieces of information solve 80% of command-line mysteries.

10.3 Background Processes — Fire and Forget

Not everything finishes quickly. Some processes are meant to run for a long time — a web server, a file watcher, a long-running test suite, a data processing pipeline. You don't want to sit around waiting for these. You want to start them, move on, and check back later.

That's where background mode comes in.

Let me show you the difference:

```
> terminal("python3 -m http.server 8080", background=true)

Result:
Session ID: bg_server_a3f2
Status: running
```

Notice what happened: instead of getting the command's output, I got a **session ID**. That's your handle for this background process. The command is still running — the HTTP server is listening on port 8080 — but Hermes didn't wait for it to finish. It started the process, gave you an ID, and moved on.

There are two main patterns for background processes:

Pattern 1: Long-lived processes that never exit

Servers, daemons, watchers — things that run until you tell them to stop. The HTTP server above is a perfect example. It starts up and just listens forever. You start it with background=true, you get a session ID, and you use process management (which we'll cover next) to check on it or kill it when you're done.

```
> terminal("npm run dev", background=true, workdir="/home/mike/
Projects/frontend")

Result:
Session ID: bg_dev_b7c1
Status: running
```

That might be a development server that auto-reloads when you change files. It'll run for hours. You don't need to babysit it.

Pattern 2: Long-running tasks that eventually finish

Build jobs, test suites, batch processing — things that take a while but have an end. For these, use notify_on_complete=true:

```
> terminal("make test", background=true, notify_on_complete=true,
            workdir="/home/mike/Projects/api-server")

Result:
Session ID: bg_test_d9e4
Status: running
```

Now Hermes will automatically notify you when the test suite finishes. No polling needed. You can go work on something else, and Hermes will tap you on the shoulder when it's done. This is one of my favorite features — it means you're never blocked waiting for a slow process.

The check_interval parameter

When you want Hermes to proactively check on a background process and report progress, you can set check_interval. This tells Hermes how often (in seconds) to poll the process. The **minimum is 30 seconds** — you can't set it lower because that would create excessive overhead.

```
> terminal("python3 train_model.py", background=true,
            notify_on_complete=true, check_interval=60,
            workdir="/home/mike/Projects/ml-pipeline")

Result:
Session ID: bg_train_e1f5
Status: running
```

With check_interval=60, Hermes will check on this training job every minute and give you a progress update. Combined with notify_on_complete=true, you'll get periodic updates *and* a final notification when it finishes.

When should you use background mode?

Use background mode when:

- The process will take more than a few minutes (builds, installs, test suites)
- The process runs indefinitely (servers, watchers, daemons)
- You want to work on other things while the command runs

- You need to run multiple things simultaneously

Use foreground mode when:

- The command finishes in seconds (most everyday commands)
- You need the output immediately to decide what to do next
- The command is part of a sequence where each step depends on the previous one

One important restriction: **you cannot use PTY mode with background processes.** PTY mode (which we'll cover in section 10.7) is for interactive commands, and interactive commands need foreground mode. Background processes run without a pseudo-terminal attached.

10.4 Process Management — Monitoring and Control

Once you've started a background process, you need to manage it. That's what the process tool is for. Let me show you each of its eight actions.

The full signature:

```
process(action, session_id, data, timeout, limit, offset)
```

Only action is always required. Whether you need session_id depends on the action (the list action doesn't need one, for obvious reasons).

List: See all your background processes

```
> process(action="list")

Result:
| Session ID     | Command                     | Status   | Uptime
|
|---------------|-----------------------------|----------|--------|
| bg_server_a3f2| python3 -m http.server 8080| running  | 12m    |
| bg_dev_b7c1   | npm run dev                 | running  | 8m     |
| bg_test_d9e4  | make test                   | finished | 15m    |
```

This is your dashboard. Every background process you've started, its current status, and how long it's been running. Use it whenever you're not sure what you have running — which, trust me, happens more often than you'd think.

Poll: Check status and new output

```
> process(action="poll", session_id="bg_server_a3f2")

Result:
Status: running
New output:
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
GET / - 200 OK
GET /style.css - 200 OK
```

poll gives you a quick snapshot: is the process still running, and what new output has it produced since you last checked? It's the "peek in the window" action — lightweight, fast, and non-blocking. Use it for a quick status check.

Wait: Block until the process finishes

```
> process(action="wait", session_id="bg_test_d9e4", timeout=120)

Result:
Process finished. Exit code: 0
Full output:
Running 42 tests...
..................................
42 passed, 0 failed
```

wait blocks — it holds up the conversation until the process finishes or until the timeout expires. Use it when you need the result before you can proceed. In this case, I wanted to see the test results before deciding whether to commit the code.

If the timeout expires before the process finishes, wait returns whatever it has — partial output plus a note that the process is still running. You can always call wait again with a longer timeout, or switch to poll if you realize this is going to take a while.

Kill: Terminate a background process

```
> process(action="kill", session_id="bg_server_a3f2")
```

```
Result:
Process terminated.
```

When you're done with a server or you need to stop something that's gone wrong, kill is your emergency exit. It sends a termination signal to the process. This is how you shut down the HTTP server you started earlier, or stop a runaway script.

Always kill processes you're done with. Leftover background processes consume resources and can cause port conflicts when you try to start new ones. I have a rule: every background=true should eventually be followed by a kill — unless the process finished on its own.

Log: Full output with pagination

```
> process(action="log", session_id="bg_dev_b7c1", offset=0,
limit=50)

Result:
[Lines 1-50 of 342]
> frontend@1.0.0 dev
> vite

  VITE v5.4.2  ready in 320 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: http://192.168.1.42:5173/
  ... (46 more lines)
```

The log action gives you the complete output of a background process, with pagination. The offset parameter is the line to start from (0-indexed), and limit is how many lines to return (default 200, maximum 2000 when reading files, but for process logs it varies by output size).

This is invaluable when a process has been running for a while and you need to see everything it's been doing — not just the recent output from poll, but the full history.

Submit: Send data with Enter

```
> process(action="submit", session_id="bg_repl_c2d6",
data="print('Hello!')")
```

The submit action sends text to the process's standard input, followed by a newline — as if you typed it and pressed Enter. This is how you interact with a running process that expects input: answering prompts, entering commands into a REPL, confirming a yes/no question.

Write: Send raw stdin without a newline

```
> process(action="write", session_id="bg_repl_c2d6", data="y")
```

write is like submit, but it doesn't add a newline. This seems like a small difference, but it matters for some interactive programs that read single characters without waiting for Enter — things like less, top, or programs that ask "Press Y to continue." For those, submit would add an unwanted newline, and write is the right choice.

Close: Send EOF

```
> process(action="close", session_id="bg_repl_c2d6")
```

The close action closes the process's standard input stream — it sends EOF (End of File). This tells the process "there's no more input coming." Many programs, when they receive EOF on stdin, will finish up and exit. This is the graceful way to signal to a program that you're done providing input.

A practical workflow

Let me pull all of this together with a real workflow. Say I want to start a Python HTTP server, make a request to it, and then shut it down:

```
> terminal("python3 -m http.server 8080", background=true)

Result:
Session ID: bg_http_f1g7
Status: running

> process(action="poll", session_id="bg_http_f1g7")

Result:
Status: running
New output:
Serving HTTP on 0.0.0.0 port 8080 ...

> terminal("curl http://localhost:8080")

Result:
<!DOCTYPE html>
<html>
...directory listing...
</html>

> process(action="kill", session_id="bg_http_f1g7")
```

```
Result:
Process terminated.
```

Start the server, confirm it's running, make a request, shut it down. Four calls, clean workflow, no leftover processes.

10.5 Execute Code — Python with Superpowers

Sometimes a single shell command isn't enough. Sometimes you need a full Python script — one that calls Hermes tools, processes results, and makes decisions based on what it finds. That's what execute_code is for.

Let me show you:

```
> execute_code('''
from hermes_tools import terminal, read_file, write_file,
search_files, patch

# Run a command and capture the result
result = terminal("ls *.md")
print("Files found:")
print(result)

# Read a file
content = read_file("README.md", limit=10)
print("First 10 lines of README:")
print(content)
''')

Result:
Files found:
CHANGELOG.md
README.md
CONTRIBUTING.md

First 10 lines of README:
# My Project
...
```

execute_code runs a Python script that can call any Hermes tool programmatically. Instead of making one tool call at a time through the chat interface, you write a Python script that chains tool calls together with logic, loops, and conditionals.

When to use execute_code vs. terminal

Use terminal when you want to run a shell command — anything you'd type at a bash prompt. Use execute_code when you need programmatic logic around your tool calls.

For example, if you want to install a package and then check if a specific file exists, you *could* use two separate terminal calls. But if you want to install a package, check the exit code, read a config file, search for a specific pattern, and conditionally patch the file based on what you found — that's a script, not a sequence of chat messages.

The hermes_tools library

Inside execute_code, you have access to a special import:

```
from hermes_tools import terminal, read_file, write_file,
search_files, patch, json_parse, shell_quote, retry
```

These are the same tools you use in chat, but now they're Python functions you can call from code. Let me walk through each one:

- **terminal** — Run shell commands. Same parameters as the chat tool.
- **read_file** — Read a file with pagination. Returns content and metadata.
- **write_file** — Write content to a file. Always overwrites the entire file.
- **search_files** — Search file contents or find files by name. Ripgrep-backed.
- **patch** — Fuzzy find-and-replace in a file. Auto-runs syntax checks.

Plus built-in helpers that don't need importing:

- **json_parse** — Safely parse JSON strings, handling malformed input.
- **shell_quote** — Safely escape strings for shell commands. Use this whenever you're interpolating user input into a shell command to prevent injection.
- **retry** — Automatically retry a function call on failure, with configurable attempts and delays.

A realistic example

Let me show you a more realistic script that combines several tools:

```
from hermes_tools import terminal, read_file, search_files, patch

# Find all Python files that import the old API
results = search_files(
    pattern="from old_api import",
    target="content",
    file_glob="*.py",
    output_mode="files_only"
)

print(f"Found {len(results)} files using old API")

# Update each file
for filepath in results:
    patch(
        path=filepath,
        old_string="from old_api import",
        new_string="from new_api import"
    )
    print(f"Updated: {filepath}")

# Run tests to verify
test_result = terminal("python3 -m pytest", timeout=300)
print(f"Tests: {'PASSED' if 'passed' in str(test_result) else
'FAILED'}")
```

This script searches for files importing a deprecated API, patches them all, and then runs the test suite. Doing this one call at a time in chat would be tedious. As a script, it's clean and repeatable.

Limits

execute_code has guardrails:

- **5-minute timeout** — Scripts that run longer than 5 minutes are terminated.
- **50KB stdout cap** — If your script prints more than 50KB of output, it's truncated.
- **50 tool calls per script** — Each script can make at most 50 Hermes tool calls.

These limits exist for good reason. Without them, a buggy script could run forever, fill up your disk with output, or make thousands of tool calls in an infinite loop. The limits keep things sane.

If you hit a limit, it usually means you should restructure your approach. Need to process thousands of files? Break it into batches. Need more output? Write to a file instead of printing. Need more tool calls? Use a more efficient algorithm.

10.6 File Operations — Read, Write, Search, Patch

Hermes has four dedicated file tools. These are purpose-built for working with files, and they're almost always better than using terminal commands like cat, grep, or sed. Let me show you why.

read_file — Paginated reading

```
> read_file("README.md", offset=1, limit=20)

Result:
  1| # My Project
  2|
  3| A web application built with Python and Flask.
  4|
  5| ## Installation
  6|
  7| ```bash
  8| pip install -r requirements.txt
  9| ```
 10|
 11| ## Usage
 12|
 13| Run the server:
 14|
 15| ```bash
 16| python3 app.py
 17| ```
 18|
 19| ## Configuration
 20|
Total lines: 87
```

Notice the output format: each line is numbered (1-indexed), and the last line tells you the total number of lines in the file. This makes pagination easy — you can read lines 1-20, then 21-40, and so on.

The defaults are offset=1 and limit=500, so if you just call read_file("some_file"), you get the first 500 lines. For most files, that's the whole thing. For large files, you paginate.

Why read_file instead of cat? Because cat dumps the entire file at once with no structure. If a file has 10,000 lines, you get 10,000 lines of unstructured output in your terminal tool. read_file gives you line numbers, total count, and pagination. It's designed for the way AI agents (and humans) actually consume files — in chunks, with context.

write_file — Atomic writes

```
> write_file("hello.py", "print('Hello, world!')\n")

Result:
File written: hello.py
```

write_file always **overwrites the entire file**. This is critical to understand. It doesn't append. It doesn't merge. It replaces. If the file existed before, its previous content is gone.

This sounds limiting, but it's actually the safe default for an AI agent. Appending to files is error-prone — you might add content in the wrong place, duplicate sections, or corrupt the structure. Overwriting is predictable: what you write is exactly what ends up in the file.

Why write_file instead of echo > file? Because echo in the terminal tool is a shell command with quoting issues, escaping problems, and no confirmation. write_file takes the content as a direct parameter — no quoting, no escaping, no ambiguity. The content you provide is exactly what gets written.

If you need to modify part of a file without rewriting the whole thing, use patch.

search_files — Ripgrep-backed search

Hermes has two search modes: content search and file search.

Content search (find text inside files):

```
> search_files(pattern="TODO", target="content",
file_glob="*.py",
                output_mode="content", limit=10)

Result:
app.py:15: # TODO: Add error handling
```

```
utils.py:42: # TODO: Refactor this function
tests/test_app.py:8: # TODO: Add edge case tests
```

File search (find files by name):

```
> search_files(pattern="*.config.js", target="files")

Result:
webpack.config.js
jest.config.js
babel.config.js
```

The search_files tool is backed by ripgrep, which means it's fast — really fast. We're talking searching thousands of files in milliseconds. It supports regex patterns, file glob filters, and multiple output modes:

- content — matches with line numbers (default)
- files_only — just the file paths
- count — number of matches per file

The context parameter adds surrounding lines to each match:

```
> search_files(pattern="def authenticate", target="content",
                context=3, file_glob="*.py")

Result:
auth.py:
  12| from database import get_user
  13|
  14| def authenticate(username, password):
  15|     user = get_user(username)
  16|     if user and user.check_password(password):
  17|         return user
```

Why search_files instead of grep? Because grep is a terminal command that produces unstructured output. You'd need to parse the grep output to extract file paths and line numbers. search_files gives you structured results with pagination, glob filtering, and multiple output modes. It's grep, but designed for programmatic use.

patch — Surgical file edits

```
> patch(path="app.py", old_string="def authenticate(username,
password):",
        new_string="def authenticate(username, password,
two_factor_code=None):")

Result:
```

```
Patched: app.py
(1 replacement made)
```

The patch tool finds a unique string in a file and replaces it. It's fuzzy — minor whitespace and indentation differences won't break the match. And after patching, it automatically runs a syntax check (when the file is a supported format like Python or JavaScript) to catch obvious errors.

The key constraint: the old_string must be **unique** in the file. If it appears in multiple places, patch will refuse to run (unless you set replace_all=true, which replaces every occurrence). This is a safety feature — it prevents you from accidentally changing the wrong thing.

Use replace_all=true when you genuinely want to change every occurrence:

```
> patch(path="config.py", old_string="old_api.example.com",
        new_string="new_api.example.com", replace_all=true)

Result:
Patched: config.py
(3 replacements made)
```

Why patch instead of sed? Because sed is finicky. Its syntax is arcane. It struggles with multi-line replacements. It doesn't verify its work. patch is purpose-built for the "find this exact text and replace it with this other text" pattern, which covers 90% of file editing tasks. And the automatic syntax check means you'll catch a missed quote or broken indentation immediately.

When to use file tools vs. terminal commands

Here's a simple decision framework:

- **Reading a file?** Use read_file. Not cat, head, or tail.
- **Writing a file?** Use write_file. Not echo or output redirection.
- **Searching files?** Use search_files. Not grep, rg, or find.
- **Editing a file?** Use patch. Not sed or awk.
- **Doing something none of these cover?** Use terminal. But think hard first — the file tools cover almost everything.

The reason is consistency and safety. Terminal commands produce unstructured output that's hard to parse. File tools produce structured, predictable results. Terminal commands are one-shot — you run them, you get output, but it's not easy to programmatically build on that output. File tools are designed to chain together: read a file, search for something, patch the result, write it back.

10.7 PTY Mode — Interactive Commands

Some programs are interactive. They expect a terminal — a real pseudo-terminal where they can draw menus, accept keystrokes, and update the screen in place. If you try to run these programs in normal mode, they either crash or produce garbled output.

That's what PTY mode is for. Set pty=true and Hermes creates a pseudo-terminal for the command:

```
> terminal("python3", pty=true)

Result:
Python 3.11.9 (main, Apr  2 2026, 00:00:00)
[GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

With pty=true, the Python REPL sees a real terminal and behaves normally. Without it, the REPL might not recognize that it's connected to a terminal, and it can behave unpredictably — missing prompts, broken line editing, or just hanging.

Use PTY mode for:

- **REPLs** — Python, Node.js, Ruby irb, etc.
- **Interactive CLI tools** — htop, vim, nano, less
- **Programs that check isatty()** — some programs change their behavior depending on whether they're connected to a terminal

Git and the paging problem

Here's a sneaky one: git log (and several other git commands) automatically page their output through less when running in a terminal. This means if you run git log with pty=true, it will open less,

and you'll be stuck — less is an interactive program waiting for your keystrokes, and you can't provide them without PTY process management.

The solution is simple: pipe git output to cat:

```
> terminal("git log --oneline -10 | cat")

Result:
a3f2c1d Fix authentication bug
b7e4d2a Add user profile page
c9f6e3b Update dependencies
...
```

The | cat at the end prevents git from launching a pager. It's a simple trick that saves you from a world of interactive-prompt frustration.

When NOT to use PTY

PTY mode is only for interactive programs. For everything else, use normal mode. Reasons:

- **Background processes cannot use PTY.** This is a hard restriction. Background processes have no terminal attached.
- **PTY adds overhead.** Creating a pseudo-terminal takes resources. For simple commands, it's unnecessary.
- **PTY can cause unexpected behavior.** Some programs produce different output in PTY mode — adding color codes, changing formatting, or enabling line editing features that confuse output parsing.

Default rule: don't use pty=true unless you're running a program that specifically needs it. If the command works fine without it, leave it off.

10.8 My Terminal Mistakes — Command Line Calamities

I've made some memorable mistakes with the terminal. Let me share a few, so you can learn from my blunders instead of making your own.

The rm -rf that almost was

I was cleaning up a project directory. The task was simple: delete a folder called build inside the project. So I told Hermes:

```
rm -rf build
```

But I wasn't specific about the working directory. I assumed Hermes would run this in the project directory, but at that point in the session, the working directory was my home folder.

rm -rf build from my home directory would have deleted a folder called build that happened to exist there — a folder containing months of compiled binaries for an unrelated project.

The approval system saved me. When Hermes asked "Allow this command: rm -rf build?", I looked at it and thought, "Wait, where is this going to run?" I checked, and sure enough, the default working directory was my home, not the project.

After that scare, I made two rules for myself:

1. **Always specify workdir for destructive commands.** rm -rf build with workdir="/home/mike/Projects/my-app" is unambiguous. It can only delete the right target.
2. **Double-check the path before approving any rm command.** The approval system gives you a moment of reflection. Use it.

The background process I forgot about

I was debugging a web application and started a development server in the background:

```
> terminal("npm run dev", background=true)

Result:
Session ID: bg_dev_f2g8
```

Then I got distracted. I started working on a different project, answered some messages, and came back an hour later. I tried to start the *same* development server for a different project, and got an error: "Port 3000 is already in use."

The old server was still running. I'd completely forgotten about it. It had been sitting there for an hour, consuming resources, holding the port hostage.

I ran process(action="list") and found not one but *three* background processes I'd started and forgotten about during my debugging session. I killed them all and made a mental note: **always clean up background processes.**

Now I keep a habit: whenever I start a background process, I immediately note the session ID and what it's for. At the end of every work session, I run process(action="list") and kill anything that shouldn't still be running. Think of it as closing tabs in your browser — it keeps things tidy and prevents resource leaks.

The infinite loop script

I wrote a script that was supposed to generate some test data. It looked something like this:

```
count = 0
while True:
    count += 1
    print(f"Item {count}")
```

The problem? I forgot to add a break condition. The script would print "Item 1", "Item 2", "Item 3", … forever. I ran it with execute_code and sat there watching the output stream by, waiting for it to stop. It didn't stop. It just kept going.

After about a minute, the 5-minute timeout from execute_code felt like it was approaching, but I didn't want to wait. So I learned about the timeout mechanism the hard way. The script hit the 50KB stdout cap first — it had printed tens of thousands of lines, filling up the output buffer, and then it hit the 5-minute timeout and was terminated.

The lesson: **always set exit conditions on loops in scripts you run through execute_code.** The timeouts and caps will eventually catch runaway code, but that's a safety net, not a design pattern. Write your loops with explicit termination. And if you're running something that genuinely needs to loop for a long time, consider using terminal with background=true and managing it with the process tool, where you can kill it on your own terms.

Lessons learned

From all these mistakes, here are the rules I follow now:

1. **Specify workdir for destructive commands.** Don't assume the current directory is what you think it is.

2. **Clean up background processes.** Use process(action="list") regularly. Kill anything you're done with.
3. **Set exit conditions on loops.** Timeouts are a safety net, not a feature.
4. **Use the approval system.** Even if you're in auto mode most of the time, keep manual approval for rm, sudo, and anything that touches your ~/.hermes directory.
5. **Use the right tool for the job.** read_file for reading, write_file for writing, search_files for searching, patch for editing. Don't reach for terminal commands when a purpose-built file tool exists.
6. **Pipe git output to cat.** Every time. It saves you from the interactive pager trap.
7. **Test commands in a safe directory first.** If you're not sure what a command will do, run it in /tmp or a test directory where the stakes are low.

Hands-On Exercise: Five Flavors of Terminal

Time to get your hands dirty. This exercise walks you through five different ways to use Hermes's terminal and process tools. Do each one, observe the output, and make sure you understand what happened before moving on.

1. Foreground command

Run a simple command in foreground mode and read the result:

```
> terminal("whoami && pwd && date")
```

What to observe: You should see your username, your current working directory, and the current date/time, all printed together. This demonstrates that compound commands with && work just like they do in a regular terminal.

2. Background process with monitoring

Start a background process and manage it:

```
> terminal("python3 -m http.server 9999", background=true)
```

Note the session ID. Now poll it:

```
> process(action="poll", session_id="YOUR_SESSION_ID")
```

Then make a request to it:

```
> terminal("curl -s http://localhost:9999 | head -5")
```

Finally, shut it down:

```
> process(action="kill", session_id="YOUR_SESSION_ID")
```

What to observe: The server starts in the background, poll confirms it's running, curl successfully fetches a page from it, and kill shuts it down cleanly.

3. File operations

Create a file, read it, search it, and edit it:

Write a file:

```
> write_file("/tmp/hermes-demo.txt", "Line 1: Hello\nLine 2:
World\nLine 3: Foo\nLine 4: Bar\n")
```

Read it back:

```
> read_file("/tmp/hermes-demo.txt")
```

Search in it:

```
> search_files(pattern="Hello", target="content", path="/tmp")
```

Patch it:

```
> patch(path="/tmp/hermes-demo.txt", old_string="Line 2: World",
new_string="Line 2: Hermes")
```

Read it again to confirm:

```
> read_file("/tmp/hermes-demo.txt")
```

What to observe: write_file creates the file, read_file shows it with line numbers, search_files finds the matching line, and patch makes a surgical edit without touching the other lines.

4. Execute code with tool calls

Run a Python script that chains multiple Hermes tools together:

```
> execute_code('''
from hermes_tools import terminal, read_file, search_files

# Check what Python version we have
result = terminal("python3 --version")
print(f"Python version: {result}")

# List markdown files in current directory
files = search_files(pattern="*.md", target="files")
print(f"Markdown files found: {len(files)}")
for f in files[:5]:
    print(f"  - {f}")
''')
```

What to observe: The script runs two Hermes tools in sequence and processes their results with Python. This shows you how execute_code lets you combine tool calls with programmatic logic.

5. PTY mode for an interactive command

Run a Python command in PTY mode and interact with it:

```
> terminal("python3 -c \"import sys; print('tty:',
sys.stdout.isatty())\"", pty=true)
```

Then try the same command without PTY:

```
> terminal("python3 -c \"import sys; print('tty:',
sys.stdout.isatty())\"")
```

What to observe: With pty=true, sys.stdout.isatty() returns True, confirming the program sees a real terminal. Without it, it returns False. This demonstrates why PTY mode matters for interactive programs that check for terminal capabilities.

Try It Now

Pick one thing from your workflow that you normally do manually in a terminal — checking git status, reading a log file, starting a local server — and have Hermes do it for you. Start with a simple foreground command. Then try a background process. Then try a mini-script in execute_code. The goal isn't to replace everything you do — it's to start building the instinct for when Hermes's terminal tools are the right tool for the job.

If something goes wrong, remember: the approval system is there to protect you, process(action="kill") is there to stop runaway processes, and I've already made all the dumb mistakes so you don't have to. Mostly.

Key Takeaways:

- Hermes's terminal tool runs **real commands** in a real shell — it's not simulating anything.
- Foreground mode (background=false, the default) is for quick commands that finish in seconds or minutes.
- Background mode (background=true) is for servers, long builds, and processes you want to forget about temporarily. You get a session ID to manage them later.
- process() gives you eight actions to monitor and control background processes: list, poll, wait, kill, log, submit, write, close.
- execute_code lets you write Python scripts that call Hermes tools programmatically, with a 5-minute timeout, 50KB output cap, and 50 tool call limit.
- Use read_file instead of cat, write_file instead of echo, search_files instead of grep, and patch instead of sed — they're purpose-built for AI agent workflows.
- PTY mode (pty=true) is for interactive programs like REPLs. For git commands, pipe output to cat to avoid interactive pagers.
- The approval system, timeouts, and output caps are your safety net. But writing careful commands is better than relying on the net to catch you.

Chapter 11: Delegation — Many Hands Make Light Work

Imagine you're running a research project. You need to investigate three different topics, write a summary of each, and then stitch them together into a final report. If you do it yourself, one after another, it

takes all afternoon. But if you hand each topic to a separate researcher who works at the same time as the others? You get your results in a fraction of the time.

That's exactly what delegation does in Hermes. You spawn subagents — independent AI workers — that tackle pieces of your workload while you stay focused on the big picture. Some tasks run one at a time. Some run in parallel. All of them return only their final conclusions to you, keeping your own context clean and uncluttered.

Let me show you delegation in action before we talk about how it works under the hood.

A Delegation Scenario: Before and After

Here's a real task I faced: I needed to understand the licensing differences between three open-source projects before recommending one to my team. Doing it the old way — one tool call after another in my own conversation — looked like this:

I searched for Project A's license. Read the page. Summarized it. Searched for Project B's license. Read the page. Summarized it. Searched for Project C's license. Read the page. Summarized it. By the time I was done, my conversation history was bloated with three full pages of license text, and I still had to manually compare them.

With delegation, the same task looks like this:

```
result = delegate_task(
  tasks=[
    {
      "goal": "Research the license for the Rust programming
language project. Summarize the license type, key restrictions,
and commercial compatibility in 3-5 bullet points.",
      "context": "We are evaluating open-source projects for
commercial use. Focus on license compatibility with proprietary
software distribution.",
      "toolsets": ["web"]
    },
    {
      "goal": "Research the license for the Python programming
language project. Summarize the license type, key restrictions,
and commercial compatibility in 3-5 bullet points.",
      "context": "We are evaluating open-source projects for
commercial use. Focus on license compatibility with proprietary
software distribution.",
      "toolsets": ["web"]
```

```
    },
    {
      "goal": "Research the license for the Go programming
language project. Summarize the license type, key restrictions,
and commercial compatibility in 3-5 bullet points.",
      "context": "We are evaluating open-source projects for
commercial use. Focus on license compatibility with proprietary
software distribution.",
      "toolsets": ["web"]
    }
  ]
)
```

Three subagents launched simultaneously. Each one did its own research. Each one came back with a concise summary. My conversation stayed clean. And the whole thing finished in roughly the time it would have taken to research just one project — because all three subagents worked in parallel.

The result variable held an array of three summaries, one per task. I then compared them in a single step and delivered my recommendation. What took a serial approach dozens of turns and a cluttered context took a parallel approach three summaries and one synthesis.

That's the power of delegation. Now let's understand exactly how it works.

11.1 Why Delegate? — The Power of Many Agents

Delegation isn't just about speed, though speed is a big part of it. There are three core reasons delegation matters:

Parallelism Beats Sequential

If you have independent tasks, running them in parallel is faster than running them one at a time. This is just basic economics — the same reason a kitchen with three chefs produces more food than a kitchen with one. Batch delegation lets you fire off up to three subagents at once, all running concurrently. Research task A, research task B, and research task C all happen at the same time.

Context Isolation Keeps Your Conversation Clean

Every time you call a tool in your main conversation, the result gets added to your context. Search the web, read a 2,000-word page, and now you're carrying 2,000 extra tokens in your conversation history. Do that three times and your context is drowning in raw data you don't need anymore.

Subagents are different. A subagent runs its own separate conversation. It can search, read, analyze, and process as much as it wants — but only the final summary comes back to you. All the intermediate results — the search results, the full web pages, the scratch-pad notes — stay in the subagent's conversation and never enter yours.

This is the "final summary only" principle, and it's the single most underrated benefit of delegation. Your context is your most precious resource. Delegation protects it.

Reasoning-Heavy Tasks Get Dedicated Focus

Some tasks need real thinking — debugging a subtle error, reviewing code for security flaws, synthesizing conflicting information from multiple sources. These tasks deserve a focused agent with a clean context, not a tired agent that's been working on ten other things in the same conversation.

Delegation gives reasoning-heavy subtasks their own dedicated workspace. The subagent approaches the problem fresh, with only the information you explicitly provide, and returns a well-considered answer.

When NOT to Delegate

Delegation isn't a hammer and not every task is a nail. Here are three cases where you should skip delegation and do the work yourself:

Mechanical multi-step work with no reasoning needed. If you're just renaming files, reformatting a document, or running a predefined script, use execute_code instead. There's no thinking required, so there's no benefit to spawning a separate agent.

Single tool calls. If you just need to search the web once, just call the search tool directly. Spawning a subagent to make one tool call is like hiring a contractor to flip a light switch.

Tasks that need user interaction. Subagents cannot call clarify. They cannot ask you questions. They cannot wait for your input mid-task. If a task requires back-and-forth with the user, you must handle it in the parent conversation.

11.2 Single-Task Delegation — One Agent, One Job

The simplest form of delegation is the single task. You describe one job, optionally provide background information, and a subagent goes off and does it.

Let's walk through each parameter.

goal — What the Subagent Should Accomplish

The goal parameter is the most important thing you write when delegating. It tells the subagent exactly what to produce. And here's the critical point: **the goal must be self-contained.**

The subagent has no memory of your conversation. It doesn't know what you were discussing five minutes ago. It doesn't know what project you're working on. It doesn't know what "the thing we talked about earlier" refers to. Everything the subagent needs to know must be in the goal or the context.

Bad goal:

```
goal = "Research the same topic we discussed earlier and write a
summary"
```

Good goal:

```
goal = "Research the current state of WebAssembly garbage
collection proposals as of 2026. Write a 200-word summary
covering: (1) which proposals exist, (2) their current
implementation status in major browsers, and (3) key remaining
challenges."
```

See the difference? The bad goal assumes the subagent knows what you were talking about. The good goal spells out exactly what to research and exactly what to produce.

context — Background Info the Subagent Needs

While the goal describes what to produce, the context provides background information that helps the subagent do a better job. Think of it as the briefing document you'd hand to a new team member on their first day.

```
delegate_task(
  goal="Write a migration plan from Python 3.11 to 3.13 for a
Django project",
  context="The project is a Django 4.2 e-commerce application with
47 models, 200+ API endpoints, and uses Celery for background
tasks. It currently runs on Ubuntu 22.04 in production. The team
has 6 developers. Key dependencies include psycopg2, redis, and
django-stripe."
)
```

Without the context, the subagent would give you a generic migration plan. With the context, it can tailor the plan to your specific stack, team size, and dependencies.

Be specific in your context. The subagent knows nothing about your world except what you tell it.

toolsets — Which Tools to Give It

By default, a subagent gets the toolsets configured in delegation.default_toolsets (terminal, file, web — set in your config). If you don't override per task, those defaults apply.

```
delegate_task(
  goal="Research the current market size for AI code assistants",
  toolsets=["web"]  # Only needs web access, nothing else
)
delegate_task(
  goal="Refactor the authentication module in /src/auth.py to use
OAuth2",
  toolsets=["terminal", "file"]  # Needs filesystem access, not
web
)
```

Why restrict toolsets? Two reasons. First, it's a good security practice — if a subagent doesn't need web access, don't give it web access. Second, it helps the subagent focus. When an agent has fewer tools, it spends less time deliberating about which tool to use and more time actually using the right one.

max_iterations — Limiting How Long a Subagent Can Work

The max_iterations parameter sets a ceiling on how many tool-calling turns a subagent can take before it's forced to stop. The default is 50.

Why would you change it?

Lower it for simple tasks. If you're asking a subagent to do a quick lookup that should take 3-5 tool calls, set max_iterations=10. This prevents a runaway subagent from burning tokens on an endless search.

Raise it cautiously for complex tasks. Some debugging or research tasks genuinely need many iterations. If you're asking a subagent to dig through hundreds of log files to find a rare error, you might set max_iterations=100.

```
delegate_task(
  goal="Find the root cause of the memory leak described in the
context",
  context="Our Node.js server leaks ~50MB per hour under moderate
load. Heap snapshots are in /opt/app/snapshots/.",
  max_iterations=75
)
```

Most of the time, the default of 50 is fine. Only change it when you have a specific reason.

11.3 Batch Delegation — Three at Once

This is where delegation gets really powerful. Batch mode lets you launch up to three subagents simultaneously, all running in parallel, all working on their own tasks.

The Batch Pattern

Instead of providing a single goal, you provide a tasks array:

```
result = delegate_task(
  tasks=[
    {
      "goal": "Research the environmental impact of lithium mining
in South America",
      "context": "We're writing a blog post about EV battery
supply chains. Focus on water usage and local community effects.",
      "toolsets": ["web"]
    },
    {
      "goal": "Research the environmental impact of cobalt mining
in the DRC",
      "context": "We're writing a blog post about EV battery
supply chains. Focus on labor practices and pollution.",
      "toolsets": ["web"]
    },
    {
      "goal": "Research recent advances in solid-state batteries
that reduce mining dependency",
      "context": "We're writing a blog post about EV battery
supply chains. Focus on technologies closest to
commercialization.",
      "toolsets": ["web"]
    }
  ]
)
```

Three subagents launch. Three subagents run at the same time. Three summaries come back. And result is an array with one entry per task, in order.

What Comes Back

The results are always an array, even if you're only running one task. Each entry contains the final summary from the subagent.
Intermediate tool results — all the web pages it read, all the searches it ran, all the files it opened — never make it into your context. You get the conclusion, not the process.

This is by design. The whole point of delegation is that you don't need to see the sausage being made.

When Parallel Makes Sense

Batch delegation shines when your tasks are **independent** — meaning none of them depends on the result of another. The three research tasks above are perfect examples: each subagent can do its job without knowing what the other two are finding.

This pattern is so common it has a name in the Hermes community: **“Research and Synthesize.”** You launch parallel research agents, collect their summaries, and then synthesize the results yourself.

Reading the Results

When your batch delegation finishes, result holds an array. The first element corresponds to the first task in your tasks array, the second to the second task, and so on:

```
result[0]  # Summary from task 1
result[1]  # Summary from task 2
result[2]  # Summary from task 3
```

Each element is a text summary — the subagent’s final answer. There’s no structured data, no status codes, no metadata. Just the answer, in plain text, exactly as the subagent wrote it.

This simplicity is intentional. If you need structured data back — say, a JSON object — then put that requirement in the goal. For example: "Return your findings as a JSON object with keys: topic, findings, confidence." The subagent will format its response accordingly.

When Sequential Is Better

Not every set of tasks can run in parallel. If Task B depends on the output of Task A, you can’t run them simultaneously. You must do Task A first, get the result, and then use that result as context for Task B.

For example: if you’re researching a topic and then writing a report about that topic, the writing depends on the research. You’d delegate the research first, wait for the result, then either do the writing yourself or delegate it as a second task with the research results in its context.

Here’s what that looks like in practice:

```
# Step 1: Research
research_result = delegate_task(
```

```
  goal="Research the current state of quantum error correction.
Write a detailed 500-word summary.",
  toolsets=["web"]
)

# Step 2: Write (using research results as context)
report_result = delegate_task(
  goal="Write a 300-word executive briefing suitable for a non-
technical audience, based on the research provided in context.",
  context=f"Research findings: {research_result[0]}",
  toolsets=[]
)
```

Notice the difference. The first call does research with web access. The second call does writing with no tools needed — only the research context. This sequential pattern is clean, predictable, and avoids the race conditions that come from parallel file access.

Batch mode doesn't help here. Use single-task delegation twice, sequentially.

The rule of thumb: **batch for independence, sequence for dependence.** A good practical test: ask yourself, "Could I shuffle the order of these tasks without changing the results?" If yes, batch them. If no, sequence them.

11.4 What Subagents Can't Do — Hard Limits

Subagents are powerful, but they have hard limits. Knowing these limits is just as important as knowing their capabilities. Let me walk through each one, because violating these limits is the fastest way to watch your delegation strategy fall apart.

No Parent Memory

A subagent has zero knowledge of your conversation. It doesn't know what you discussed earlier. It doesn't know what variables you defined. It doesn't know what files you have open. It doesn't know the user's name, the project's name, or the problem you're solving.

Everything the subagent needs must be passed via the goal and context fields.

This is the mistake I see most often from people new to delegation. They write a goal like “continue what we were doing” and are shocked when the subagent has no idea what they were doing. The subagent isn’t continuing anything. It’s starting fresh, alone, with only what you gave it.

No Clarify

Subagents cannot call clarify. They cannot ask the user a question. They cannot pause and wait for input. Once launched, a subagent runs autonomously until it finishes or hits its iteration limit.

This means your goal and context must be complete enough that the subagent never needs to ask a follow-up question. If you’re not sure whether you’ve provided enough information, you probably haven’t.

No Nesting

A subagent cannot call delegate_task. Delegation is one level deep. A parent spawns a child, the child does its work, the child returns a summary. That’s it. No grandchildren.

This is a deliberate design choice. Nested delegation creates exponential agent explosions — one parent spawns three children, each child spawns three more, suddenly you have twelve subagents running and your token bill looks like a phone number. The one-level limit keeps things predictable and controllable.

No send_message, Memory, or execute_code

Subagents also cannot call send_message, memory, or execute_code. They can use the tools in their assigned toolsets (like terminal, file, and web), but these five tools are permanently off-limits:

- delegate_task — no nesting
- clarify — no user interaction
- memory — no persistent memory access
- send_message — no messaging
- execute_code — no code execution sandbox

These aren't bugs or missing features. They're intentional security and resource boundaries.

The Golden Rule: Make Goals Self-Contained

If you take one lesson from this chapter, make it this: **a subagent's goal must be completely self-contained.** The subagent knows nothing, can ask nothing, and can delegate nothing. It has only what you put in goal and context.

Before you hit submit on a delegation call, ask yourself: "If I handed this goal and context to a complete stranger with no other information, could they do the job?" If the answer is no, you need to add more detail.

11.5 ACP — Spawning Other AI Agents

Here's where delegation gets really interesting. Hermes isn't the only AI agent in town. There's Claude Code, Copilot, and other ACP-capable agents. The Agent Communication Protocol lets you spawn these other agents as subagents from within Hermes.

What Is ACP?

ACP stands for Agent Communication Protocol. It's a standardized way for AI agents to talk to each other through a command-line interface. If an agent supports ACP, any other ACP-capable agent can launch it, assign it a task, and collect its output.

Spawning Claude Code from Hermes

Let's say you're working in Hermes but you want Claude Code to handle a specific task — maybe Claude has a particular strength you want to leverage. You can do that through ACP delegation:

```
result = delegate_task(
  goal="Review the code in /src/api.py for security
vulnerabilities",
  context="This is a Flask API that handles user authentication.
Focus on injection attacks and auth bypass.",
  acp_command="claude",
  acp_args=["--acp", "--stdio"]
)
```

Setting acp_command="claude" tells Hermes to launch a Claude Code agent instead of a standard Hermes subagent. The acp_args tell Claude Code to use ACP mode over stdio, which is the standard communication channel.

acp_command and acp_args

- **acp_command**: The CLI command to launch the child agent. The default spawns a Hermes subagent, but you can override it to "claude", "copilot", or any other ACP-capable binary.
- **acp_args**: Arguments passed to the child agent. The default is ["--acp", "--stdio"], which tells the child agent to operate in ACP mode using standard I/O. You can customize this if a particular agent requires different arguments.

Per-Task Overrides in Batch Mode

In batch mode, you might want different subagents running on different AI backends. For example:

```
result = delegate_task(
  tasks=[
    {
      "goal": "Research best practices for React Server
Components",
      "toolsets": ["web"]
    },
    {
      "goal": "Review our API schema for consistency",
      "context": "The schema is in /src/schema.graphql",
      "toolsets": ["terminal", "file"],
      "acp_command": "claude"
    }
  ]
)
```

Here, the first task runs as a standard Hermes subagent. The second task runs as a Claude Code agent. Each task in the batch can independently specify its own acp_command and acp_args.

ACP Works from Any Transport

One of the elegant things about ACP is that it works regardless of how you're talking to Hermes. Whether you're using the CLI, Discord, Telegram, or any other transport, ACP delegation works the same way.

The child agent is always spawned via the command you specify, communicating over stdio. Your transport layer doesn't matter — the subagent is a separate process.

11.6 Delegation Config — Tuning the System

Delegation behavior can be configured in your Hermes configuration file. Here's what the delegation section looks like:

```
delegation:
  model: ''           # empty = inherit from parent
  provider: ''         # empty = inherit from parent
  base_url: ''         # empty = inherit from parent
  api_key: ''          # empty = inherit from parent
  max_iterations: 50
  default_toolsets: [terminal, file, web]
```

model: Inherit vs. Specify

When model is empty (the default), subagents inherit the same model as the parent agent. This is usually what you want — consistency between parent and child.

But you can override it. Why would you? Maybe you want a cheaper, faster model for simple research tasks while keeping the parent on a more powerful model. Or maybe the parent is running on a local model and you want subagents to use a cloud model for web research.

```
delegation:
  model: 'gpt-4o-mini'  # Use a faster, cheaper model for
subagents
```

Be thoughtful here. A weaker model might not handle complex reasoning tasks as well, and a stronger model costs more tokens. Match the model to the task.

max_iterations: 50 Default

This sets the global default for how many tool-calling turns each subagent gets. The default of 50 is a reasonable balance — enough for most research and coding tasks, not so many that a stuck subagent runs forever.

You can override this per task in the delegate_task call (as we discussed in section 11.2), but the config sets the baseline that all subagents start from.

When to change the config default: - **Lower it** if you're mostly delegating simple lookups and you want to limit costs. - **Raise it** if you regularly delegate complex debugging or deep research tasks.

default_toolsets: [terminal, file, web]

This sets the default toolsets for all subagents. The default includes terminal, file, and web — a sensible general-purpose set.

You can narrow or expand this per task, but the config default determines what a subagent gets if you don't specify toolsets in the delegation call.

If you want to be more restrictive by default — say, you don't want subagents accessing the web unless you explicitly allow it — you could set:

```
delegation:
  default_toolsets: [terminal, file]  # No web by default
```

Then when a task genuinely needs web access, you add it explicitly:

```
delegate_task(
  goal="Research current pricing for cloud GPU providers",
  toolsets=["terminal", "file", "web"]  # Explicitly adding web
)
```

This is a good security posture: deny by default, allow on request.

11.7 My Delegation Mistakes — Multi-Agent Mishaps

I've made every delegation mistake in the book. Let me walk you through the three worst ones so you can avoid them.

Mistake 1: The Context-Less Subagent

I was building a data pipeline and needed to write unit tests for a complex transformation function. The function took a custom data format called “DataStream” that I had defined earlier in my conversation. I delegated the test-writing task like this:

```
delegate_task(
  goal="Write thorough unit tests for the normalizeDataStream
function in /src/transforms.py",
  toolsets=["terminal", "file"]
)
```

The subagent read the file and… had no idea what a DataStream was. It wrote tests that called normalizeDataStream with plain Python dictionaries. Every single test failed because the function expected DataStream objects.

I forgot to include context. The subagent had never heard of DataStream. It couldn’t ask me what DataStream was. It couldn’t look at my conversation history. It just did its best with incomplete information — and its best was wrong.

The fix was simple:

```
delegate_task(
  goal="Write thorough unit tests for the normalizeDataStream
function in /src/transforms.py",
  context="The function operates on DataStream objects defined
in /src/models.py. DataStream has fields: timestamp (datetime),
value (float), unit (str), source (str). Import it as: from models
import DataStream. Use pytest as the test framework.",
  toolsets=["terminal", "file"]
)
```

Now the subagent knew exactly what DataStream was, where to import it from, and which test framework to use. The tests passed on the first try.

Lesson: Always ask yourself — does the subagent know everything it needs to know? If you reference anything by name, explain what it is.

Mistake 2: The Parallel Collision

I had two files that both needed updating: a configuration file and a schema file. The changes were independent, so I thought batch delegation was perfect:

```
delegate_task(
  tasks=[
    {
      "goal": "Update /config/production.yaml to add the new Redis
credentials",
      "context": "Add host: redis-prod.internal, port: 6379, db: 2
under the cache section",
      "toolsets": ["terminal", "file"]
    },
    {
      "goal": "Update /config/production.yaml to add the new
Postgres credentials",
      "context": "Add host: pg-prod.internal, port: 5432, dbname:
app_v2 under the database section",
      "toolsets": ["terminal", "file"]
    }
  ]
)
```

Both subagents ran in parallel. Both opened the same file. Subagent A read the file, made its change, and wrote it back. Subagent B read the file (the original version, before A's change), made its change, and wrote it back — **overwriting A's change.**

The final file only had the Postgres credentials. The Redis changes were gone, silently overwritten.

Each subagent gets its own terminal session with its own working directory and state. When two subagents edit the same file, you get a race condition. The last one to write wins, and the loser's changes disappear.

Lesson: Never have two subagents write to the same file. If tasks touch the same file, run them sequentially, not in parallel.

Mistake 3: The Infinite Subagent

I delegated a deep research task with the default max_iterations=50 and didn't check on it. The subagent was supposed to find three academic papers about a niche topic. It searched. Found one paper but

not two more. Searched again with different terms. Found a second paper. Searched for the third. Couldn't find it. Searched with broader terms. Found too many results. Narrowed the search. Still too many…

It used all 50 iterations and then hit the limit. I got back a half-finished summary: "I found two relevant papers but was unable to locate a third." The subagent had spent most of its iterations thrashing on a search that wasn't working, instead of wrapping up with what it had.

The fix was two-fold. First, I changed the goal to be more realistic: "Find up to three academic papers on [topic]. If you cannot find three after 10 searches, summarize what you found." Second, I lowered max_iterations to 20. If the subagent can't find what it needs in 20 tool calls, 50 more won't help either.

Lesson: Set appropriate iteration limits and write goals that account for the possibility of incomplete results.

Setting Expectations via Goal and Context

When you delegate a task, you're effectively assigning an employee. The quality of the work depends heavily on the clarity of the assignment. A vague goal produces vague results. A specific goal with clear success criteria produces focused results.

One technique I use is to explicitly set iteration expectations in the goal itself:

```
goal = "Research quantum computing market leaders. After finding
3-5 major companies, stop searching and write your summary. Do not
continue searching indefinitely."
```

This prevents the "infinite searcher" problem — the subagent that keeps looking for one more source even after it has plenty of information. By setting the stopping condition in the goal, you give the subagent permission to conclude.

You can also set depth expectations:

```
goal = "Write a beginner-friendly explanation of quantum
entanglement. Aim for ~200 words. Use one analogy. Avoid
equations."
```

Instead of:

```
goal = "Explain quantum entanglement."
```

The second goal leaves too many decisions to the subagent: How long should the answer be? What's the audience level? Should I include math? The first goal removes all those questions, so the subagent can focus on the actual work.

Lesson: A goal without constraints is a goal without direction. Set expectations explicitly.

Delegation Anti-Patterns Summary

Let me collect the key mistakes in one place so you can scan them quickly:

1. **Vague goals.** "Research the project" isn't a goal. "Write a 300-word summary of the project's architecture, covering the frontend, backend, and data layer" is a goal.
2. **Missing context.** If a subagent needs to know about a custom type, a specific file path, a team convention, or a project requirement, tell it in the context field.
3. **Parallel file writes.** Two subagents writing the same file = one of them loses. Run file-modifying tasks sequentially.
4. **Infinite iteration.** Set a reasonable max_iterations. Write goals that tell the subagent what to do if it can't fully complete the task.
5. **Delegating trivial work.** Don't spawn a subagent to make one tool call. Just call the tool yourself.
6. **Delegating interactive tasks.** A subagent can't ask the user questions. If the task might need clarification, you must handle it.
7. **Assuming the subagent knows anything.** It knows nothing. Nothing about your project, your team, your conversation, your preferences. Nothing.

Try It Now: Delegate Three Parallel Research Tasks

Time to get your hands dirty. In this exercise, you'll delegate three parallel research tasks, collect the summaries, and synthesize the results.

Step 1: Pick a Topic

Choose a topic you want to learn about — something broad enough that it has three interesting sub-topics. For example:

- **Topic**: “Current state of AI regulation”
  - Sub-topic A: EU AI Act implementation
  - Sub-topic B: US federal AI policy
  - Sub-topic C: China’s AI governance framework
- **Topic**: “Sustainable energy storage”
  - Sub-topic A: Lithium-ion battery advances
  - Sub-topic B: Hydrogen fuel cell developments
  - Sub-topic C: Grid-scale thermal storage

Pick whatever interests you. The exercise works the same way.

Step 2: Write the Delegation Call

Compose a batch delegation call with three tasks. For each task, write a clear goal and specific context. Remember: the subagent knows nothing except what you put in these fields.

```
result = delegate_task(
  tasks=[
    {
      "goal": "Research [your sub-topic A]. Write a 200-word
summary covering: (1) key recent developments, (2) major
challenges or controversies, and (3) likely trajectory over the
next 2 years.",
      "context": "We are writing a briefing document for a general
audience. Avoid jargon. Cite specific events, laws, or products by
name where possible.",
      "toolsets": ["web"]
    },
    {
      "goal": "Research [your sub-topic B]. Write a 200-word
summary covering: (1) key recent developments, (2) major
challenges or controversies, and (3) likely trajectory over the
next 2 years.",
      "context": "We are writing a briefing document for a general
audience. Avoid jargon. Cite specific events, laws, or products by
name where possible.",
      "toolsets": ["web"]
```

```
    },
    {
      "goal": "Research [your sub-topic C]. Write a 200-word
summary covering: (1) key recent developments, (2) major
challenges or controversies, and (3) likely trajectory over the
next 2 years.",
      "context": "We are writing a briefing document for a general
audience. Avoid jargon. Cite specific events, laws, or products by
name where possible.",
      "toolsets": ["web"]
    }
  ]
)
```

Step 3: Collect and Compare

The result variable contains an array of three summaries. Read through them. Ask yourself:

- Are the summaries consistent or contradictory?
- Is there overlap between them that suggests a shared theme?
- Are there gaps that none of the three subagents covered?
- What's the single most important insight across all three?

Step 4: Synthesize

Now write a brief synthesis — a paragraph or two that weaves the three summaries together into a coherent picture. You can do this yourself, or you can delegate it:

```
synthesis = delegate_task(
  goal="Synthesize three research summaries into a single coherent
briefing paragraph. Highlight shared themes, key differences, and
the most important overall takeaway.",
  context="Summary A: [paste result[0]] Summary B: [paste
result[1]] Summary C: [paste result[2]]",
  toolsets=[]
)
```

Notice that the synthesis task doesn't need web access — you've already done the research. You only need reasoning, so you can leave toolsets empty or minimal.

Step 5: Reflect

After completing the exercise, ask yourself:

1. Did writing clear, self-contained goals take more effort than you expected?
2. Did any subagent return something surprising or off-target? If so, was the issue in how you wrote the goal or context?
3. How long did the parallel research take compared to doing the same three searches yourself, one at a time?
4. Was your context window cleaner at the end than it would have been without delegation?

If you struggled with step 1, that's normal. Writing good goals and context is a skill, and it gets easier with practice. The key habit: before every delegation call, pause and ask, "If I were the subagent, would I have enough information to do this task well?"

If the answer is no, add more context. If the answer is "I think so," add more context anyway.

Delegation Quick Reference

For easy reference, here's a summary of the delegate_task tool:

Single task mode:

```
delegate_task(
  goal="What the subagent should accomplish",
  context="Background information the subagent needs",
  toolsets=["terminal", "file", "web"],
  max_iterations=50,
  acp_command="",       # Override child agent (e.g., "claude")
  acp_args=[]           # Default: ["--acp", "--stdio"]
)
```

Batch mode (up to 3 parallel tasks):

```
delegate_task(
  tasks=[
    {
      "goal": "Task 1 goal",
      "context": "Task 1 context",
      "toolsets": ["web"],
      "acp_command": "claude"  # Optional per-task override
```

```
    },
    {
      "goal": "Task 2 goal",
      "context": "Task 2 context",
      "toolsets": ["web"]
    },
    {
      "goal": "Task 3 goal",
      "context": "Task 3 context",
      "toolsets": ["terminal", "file"]
    }
  ]
)
```

Subagent hard limits: - No access to parent conversation - No clarify (can't ask user questions) - No delegate_task (no nesting) - No memory, send_message, or execute_code - Only final summary returns to parent - Each gets its own terminal session

Config defaults:

```
delegation:
  model: ''
  max_iterations: 50
  default_toolsets: [terminal, file, web]
```

When to delegate: - Reasoning-heavy subtasks - Tasks that would flood your context with intermediate data - Parallel independent workstreams

When NOT to delegate: - Mechanical work with no reasoning (use execute_code) - Single tool calls (just call the tool) - Tasks needing user interaction (subagents can't clarify)

Delegation is one of those features that changes how you think about working with AI. Once you internalize the pattern — decompose your work, delegate the pieces, synthesize the results — you start seeing delegation opportunities everywhere. A research task becomes three parallel subagents. A code review becomes a dedicated subagent that reads every file and gives you a security report. A documentation update becomes a subagent that reads your code, writes the docs, and saves the file.

The trick is practice. Start small. Delegate one task today. Then try two in parallel. Then three. Pay attention to what works and what doesn't. Every failure teaches you more about writing better goals and context. And before long, you'll wonder how you ever got anything done without delegation.

In the next chapter, we'll explore the Hermes gateway — how to connect your agent to Telegram, Discord, Slack, and a dozen other platforms. Because delegation lets you do more at once, but being reachable everywhere lets Hermes do more for you.

Chapter 12: Channels — Hermes Everywhere

It's Tuesday morning. I'm standing in my kitchen, coffee in one hand, phone in the other. I just sent a voice message to my Hermes agent on Telegram asking about the weather forecast. Three seconds later, it replies — not just with text, but with a voice message of its own, calmly telling me that rain is expected by noon. I grab my umbrella, head to my desk, and continue the same conversation in my terminal by typing hermes -c to resume where I left off. By afternoon, my Cron job pipes a summary of the day's completed tasks straight to my team's Discord channel. Same agent. Three different interfaces. Zero friction.

That's the promise of Hermes channels. Your AI agent doesn't live in just one place — it lives everywhere you need it. In this chapter, we're going to turn your Hermes instance from a terminal-only tool into an omnichannel assistant that meets you (and your team, and your smart home) wherever you are.

12.1 One Agent, Many Channels — The Omnichannel Vision

Here's the core idea: you build one agent, configure it once, and it shows up across every platform you use. The personality, the knowledge, the tools — they're all the same regardless of whether you're talking to Hermes in a terminal, on Telegram, or in a Discord server.

Why does this matter? Because context switching is expensive. If you have to open a specific app just to ask your AI something, most of the time you won't bother. You'll just google it, guess, or give up. Channels eliminate that friction. Hermes goes where you already are.

Let me show you what this looks like in practice before we dive into setup.

Here I am in my terminal:

```
$ hermes chat -q "What's the capital of Mongolia?"
Ulaanbaatar.
```

Simple, fast, done. Now here I am on Telegram, in the same conversation thread, five minutes later:

Me: Translate “good morning” to Mongolian **Hermes:** Өглөөний мэнд (Öglööniin mend)

Same agent. Same context. Different interface. And on Discord, in my team’s #general channel:

@me: @Hermes summarize yesterday’s meeting notes **Hermes:** [Creates a thread] Here’s the summary: 3 action items, 2 blockers resolved, 1 new feature proposal…

The beauty is that the agent doesn’t care which channel you’re using. Your tools, your memories, your configuration — all of it travels with you. What changes is how you interact and what channel-specific features are available.

Channel-specific features vary by platform:

- **Telegram** supports voice messages (send a voice note, get a voice note back), inline queries (type @YourBotName in any chat), and images for vision tasks.
- **Discord** supports reactions (emoji responses), auto-threading (long replies get their own thread), and audio attachments.
- **WhatsApp** handles audio attachments and voice input.
- **HomeAssistant** connects your agent to smart home control.
- **CLI** gives you the richest, most complete experience with every tool and feature available.

Each channel is adapted to its platform while keeping the same underlying agent intact. You choose how you want to talk; Hermes handles the translation.

To see which channels you have connected right now, it’s one command:

```
$ hermes gateway status
```

This lists every channel your agent is currently configured for. If you've just installed Hermes, you'll see only CLI — the default channel that's always available. By the end of this chapter, you'll have at least two more.

The primary platform toolsets Hermes supports include:

1. **CLI** — the terminal, always there
2. **Telegram** — full-featured mobile bot
3. **Discord** — server and community integration
4. **WhatsApp** — via Business API
5. **Slack** — workspace bot
6. **Signal** — privacy-focused messaging
7. **HomeAssistant** — smart home integration

Beyond those core platforms, Hermes also supports additional channels: Weixin (WeChat), Feishu, DingTalk, WeCom, WeCom Callback, BlueBubbles (iMessage), Email, and SMS. Delivery can also target Matrix and Mattermost through the gateway system. That's thirteen or more platforms — all controlled from a single Hermes instance.

Let's walk through each major channel, starting with the one you already know.

12.2 CLI — The Original Channel

You've been using the CLI channel since Chapter 1. It's the default — no setup required, no configuration needed, it just works. But there's more to it than typing hermes and hitting Enter.

Three modes, three workflows:

Interactive mode is your default conversation:

```
$ hermes
```

Or equivalently:

```
$ hermes chat
```

This drops you into a conversation. You type, Hermes responds, you type again. It's like texting a very knowledgeable friend who happens to have access to your tools, your files, and the entire internet. This mode is best for exploratory work — brainstorming, debugging, asking follow-up questions, or any time you want a back-and-forth.

One-shot mode is for quick questions:

```
$ hermes chat -q "How many days until Christmas?"
```

Hermes answers, prints the result, and exits. No conversation to manage, no history to worry about. This is perfect for scripts, aliases, and those moments when you just need a fact and you need it now.

I use one-shot mode in my shell aliases. Here's one I've had in my .bashrc for months:

```
alias wtf='hermes chat -q "Explain this error message in simple terms:"'
```

Then I pipe errors to it:

```
$ some_command 2>&1 | wtf
```

Crude? Yes. Useful? Absolutely.

Resume mode picks up where you left off:

```
$ hermes -c
```

The -c flag tells Hermes to resume the last conversation. This is a lifesaver when you accidentally close your terminal, when you want to continue a deep discussion from earlier, or — as I mentioned in the opening — when you started chatting on Telegram and want to keep going at your desk.

Why CLI is the best channel:

I'm going to be honest with you: the CLI is still my primary channel. Not because I'm some terminal purist, but because it gives you everything. Every tool, every feature, every configuration option — it all works in the CLI. Other channels sometimes have limitations. Discord might truncate a long response. Telegram handles images but doesn't support every tool output format. The CLI? No limits.

When CLI shines:

- You're doing complex, multi-step work that requires tool after tool

- You want the full output without character limits or formatting constraints
- You're scripting or automating Hermes calls
- You need the maximum set of features available to your agent

When other channels win:

- You're away from your computer (Telegram, WhatsApp)
- You're collaborating with a team (Discord, Slack)
- You want to control your living room lights (HomeAssistant)
- You're reading notifications on the go (any mobile channel)

The CLI isn't going anywhere. It's the foundation. Everything else is an extension. Let's build one.

12.3 Telegram — Your AI in Your Pocket

Let me start with a story. Last winter, I was hiking in the mountains — no laptop, just my phone — when a colleague messaged me asking for a summary of a long technical document we'd been working on. Without breaking stride, I opened Telegram, sent a voice message to my Hermes bot: "Summarize the Q4 planning doc and send it to me." By the time I reached the trailhead, the summary was waiting in my chat. That's what Telegram integration gives you: your AI agent, in your pocket, wherever you are.

Here's what a Telegram conversation with Hermes actually looks like before we get into setup:

Me: [Voice message: 12 seconds] "What were our key metrics last quarter?" **Hermes:** [Voice message: 18 seconds] "Last quarter's key metrics: revenue grew 23%, user retention improved to 89%, and the new feature adoption rate hit 67%."

Or with an image:

Me: [Sends photo of a whiteboard diagram] **Hermes:** That's a system architecture diagram. I can see a load balancer in front of three application servers, with a database cluster and a cache layer. Want me to document this or suggest improvements?

Or inline, in any Telegram chat:

Me: @HermesBot convert 500 CAD to USD **Hermes:** [Inline result] ≈ $375 USD

Voice, images, inline queries — Telegram is arguably the most feature-rich channel Hermes supports. Let's get it set up.

Setting Up Telegram

Step 1: Create your bot with BotFather

Open Telegram and search for @BotFather. This is Telegram's official bot for creating and managing bots. Send it the command:

```
/newbot
```

BotFather will walk you through naming your bot. You'll pick a display name (like "My Hermes Agent") and a username (like my_hermes_agent_bot). When you're done, BotFather gives you a token that looks something like:

```
7123456789:AAHx1234567890abcdefghijklmnopqrstuvwxyz
```

Save this token. You'll need it in a moment.

Step 2: Install the Telegram channel

Back in your terminal:

```
$ hermes gateway setup
```

This command walks you through the setup process. It will ask for your bot token — paste in the one BotFather gave you. It will also configure the webhook and connectivity. The whole process takes under a minute.

Step 3: Start chatting

Open Telegram, find your new bot, and send it a message. Anything will do — "Hello!" or "Are you there?" If everything is configured correctly, Hermes responds. You now have an AI agent in your pocket.

Telegram Features You Should Know

Voice messages (STT/TTS)

One of Telegram's standout features is voice support. When you send a voice message to your Hermes bot, it uses speech-to-text (STT) to transcribe your audio, processes it like any text input, and then generates a response. If you've enabled text-to-speech output, Hermes replies with a voice message of its own.

This is huge for mobile use. Walking, driving (hands-free!), cooking — any time typing is inconvenient, just hold the microphone button and talk.

I'll cover TTS configuration in detail in section 12.6, but here's the short version: if your agent has the text_to_speech tool enabled, it can generate audio replies on supported channels. Telegram is the most natural fit for this because voice messages are a first-class feature of the platform.

Images and vision

Send a photo to your Hermes bot on Telegram, and the agent's vision capabilities kick in. Whether it's a screenshot of an error, a photo of a document, or a picture of your lunch (hey, I don't judge), Hermes can see it and respond. This works the same way as pasting images in the CLI — the image is processed by the vision model and the agent gets a description along with any text you include in the message.

Inline queries

Inline queries let you invoke your bot from any Telegram chat, not just your private conversation. Type @YourBotName followed by your question in any chat field, and Telegram will show you results from your bot. This is great for quick lookups in group conversations without needing to switch to your bot's private chat.

Topic support

If you're using Telegram's topics feature in groups (also known as forums), Hermes can target specific topics within a group. The format for specifying a topic is:

```
telegram:-1001234567890:17585
```

That breaks down as telegram:GROUP_ID:TOPIC_ID. This is particularly useful for Cron delivery, which we'll cover in section 12.7, but it also means your bot can post directly into a specific organized thread rather than cluttering the general chat.

Cron delivery to Telegram

Once your Telegram channel is set up, you can configure Cron jobs that deliver their output straight to Telegram:

```
deliver: telegram
```

This sends the Cron job's output to your bot chat. You can also target a specific chat:

```
deliver: telegram:1234567890
```

We'll go deeper into Cron delivery in section 12.7. For now, just know that Telegram is one of the best targets for scheduled outputs because push notifications mean you see them instantly on your phone.

12.4 Discord — AI for Your Server

Discord is where Hermes becomes a team player. Whether you're running a small project, managing a community, or just hanging out with friends who like having an AI around, Discord integration turns your agent into a shared resource that anyone in your server can interact with.

Here's what it looks like in action before we set it up:

@alice: @Hermes can you explain what the new deployment pipeline does? **Hermes:** 🧵 [Created thread: Hermes's explanation] The new deployment pipeline has three stages: build, test, and deploy...

@bob: [Reacts with 👍]

@alice: @Hermes what's the server load right now? **Hermes:** [Reacts with 📊] Current server load: 42% CPU, 68% memory, 2.1 load average. Running a bit warm on memory.

Notice three things in this example: the bot is only responding when mentioned (require_mention), it automatically created a thread for a longer response (auto_thread), and it used a reaction emoji before its text (reactions). These are all Discord-specific configurations that make your bot behave well in a server environment.

Setting Up Discord

Step 1: Create a Discord application

Go to the Discord Developer Portal and create a new application. Under the “Bot” tab, click “Add Bot.” Discord will generate a bot token — this is what Hermes needs to connect.

Important: You must enable the **Message Content Intent** in your bot settings. This is under the “Privileged Gateway Intents” section on the Bot tab. Without this intent, your bot won’t be able to read message content, which means it can’t respond to anything useful.

Step 2: Invite the bot to your server

In the Developer Portal, under “OAuth2 > URL Generator,” select the bot scope and the permissions your bot needs (at minimum: Send Messages, Read Message History, Create Threads). Copy the generated URL and open it in your browser to add the bot to your server.

Step 3: Install the Discord channel

```
$ hermes gateway setup
```

This will prompt you for your bot token and walk you through the configuration.

Step 4: Configure Discord-specific options

In your config.yaml, you can (and should) set Discord-specific behavior:

```
discord:
  require_mention: true
  free_response_channels: ''
  allowed_channels: ''
  auto_thread: true
  reactions: true
```

Let me explain each option because getting these wrong can cause… problems. (Spoiler: that’s section 12.8.)

require_mention: true means the bot only responds when someone explicitly @-mentions it. Without this, your bot responds to every message in every channel it can see. In a busy server, that’s chaos. In a quiet server with just you and the bot, you might prefer false so you can talk naturally.

auto_thread: true tells the bot to create a Discord thread for longer responses. This keeps the main channel clean. Without it, long responses flood the chat. With it, the bot creates a thread and puts its answer there. Much tidier.

reactions: true lets the bot add emoji reactions to messages. Sometimes Hermes will react with a relevant emoji before or instead of a text response — a 👍 for confirmation, a 🤔 when it's thinking, a ✅ when something succeeds. It's a small touch, but it makes the bot feel more natural in a Discord environment.

free_response_channels: '' and **allowed_channels: ''** are optional channel control keys. free_response_channels lets you specify channels where the bot responds freely without requiring a mention (overriding require_mention for specific channels). allowed_channels restricts the bot to only respond in listed channels, ignoring messages elsewhere. Both default to empty (no special behavior) — configure them only when you want to fine-tune where the bot is active.

Discord Channel Targeting

When configuring Cron jobs (or any delivery), you can target specific Discord channels:

```
deliver: discord:#engineering
```

This sends output to the #engineering channel in your Discord server. You can also use just discord to send to the default channel the bot is configured for.

Discord also supports thread-level targeting. If you have a thread ID, you can direct output to a specific thread, keeping conversations organized and noise to a minimum.

When Discord Shines

Discord is my go-to channel for team scenarios:

- **Team knowledge base**: Team members can @-mention the bot to ask questions about processes, documentation, or past decisions.
- **DevOps monitoring**: Cron jobs deliver server status and alerts to a dedicated #alerts channel.
- **Community management**: In larger communities, the bot can help with FAQs, moderation assistance, and onboarding new members.
- **Pair programming**: Working through a problem in a thread with the bot as an always-available rubber duck and reference guide.

The thread feature alone makes Discord special. Being able to have a focused, multi-turn conversation in a thread without cluttering the main channel is something no other channel does quite as well.

12.5 WhatsApp, Slack, Signal, and More

Not everyone lives in Discord or Telegram. Hermes meets you where you are, and for a lot of the world, that means WhatsApp. For enterprise users, it means Slack. For the privacy-conscious, it means Signal. Let's cover each of these, plus the smart home integration and the long tail of additional platforms.

WhatsApp — The Global Default

WhatsApp is the most widely used messaging app in the world. If your team or family communicates on WhatsApp, you want your agent there.

Setup is more involved than Telegram or Discord because it requires the **WhatsApp Business API**, which means a **Facebook Business account**. Meta has made this process smoother over the years, but it's still more bureaucratic:

1. Create or use a Facebook Business account
2. Register your WhatsApp Business phone number
3. Set up the WhatsApp Business API through Meta's dashboard
4. Configure the channel in Hermes with your API credentials

Once connected, you can send and receive messages with your Hermes agent via WhatsApp just like any other contact. WhatsApp supports audio attachments, so you can send voice messages that Hermes transcribes and processes.

Delivery configuration:

```
deliver: whatsapp
```

The main caveat: WhatsApp has strict message templates and rate limits compared to other platforms. It's designed for business communication, not free-form chatting, so expect some friction. Still, if your world lives on WhatsApp, the integration is worth it.

Slack — The Workspace Standard

If your organization runs on Slack, Hermes fits right in. Setting up a Slack bot involves:

1. Creating a Slack App in the Slack API dashboard
2. Granting it the appropriate scopes (chat:write, channels:history, etc.)
3. Generating a **Bot User OAuth Token** (starts with xoxb-)
4. Installing the app to your workspace
5. Configuring Hermes with the token

```
deliver: slack
```

Slack is excellent for workplace scenarios. The bot lives in your workspace, responds in channels or direct messages, and can integrate with your other Slack apps. Thread support works similarly to Discord — conversations can branch off into threads naturally.

The main thing to watch with Slack is permissions. Slack's permission model is granular and sometimes confusing. Make sure your bot has exactly the scopes it needs — no more, no less. Your workspace admins will thank you.

Signal — Privacy First

Signal is for people who care deeply about privacy. End-to-end encryption is Signal's entire identity, and Hermes respects that. Setup requires **signal-cli** or a similar bridge — a command-line interface that lets programs interact with the Signal network.

The setup process:

1. Install signal-cli on your server
2. Register a phone number with Signal through signal-cli
3. Configure Hermes to use signal-cli as the bridge
4. Exchange messages through the encrypted channel

```
deliver: signal
```

Signal is the most privacy-respecting channel Hermes supports. If you're handling sensitive data, working with confidential information, or just don't want your AI conversations stored on third-party servers, Signal is the way to go.

The tradeoff: Signal's bridge setup is more technical, and some features available on Telegram or Discord (like inline queries or reactions) aren't possible in Signal's simpler interface. What you get in return is strong encryption and minimal data exposure.

HomeAssistant — Smart Home Control

This one's different from the messaging channels. HomeAssistant integration turns your Hermes agent into the brain of your smart home. Instead of chatting through an app, you're controlling devices, automations, and sensors through the HomeAssistant platform.

```
deliver: homeassistant
```

With HomeAssistant connected, you can ask your agent to check sensor readings, toggle lights, adjust thermostats, trigger automations, and more. "Turn off all the downstairs lights" becomes a natural language command that your agent translates into HomeAssistant actions.

The setup involves connecting Hermes to your HomeAssistant instance through its API, and then your agent gains access to all the entities, devices, and automations you've configured. It's like giving your AI agent hands and eyes throughout your house.

The Long Tail: Matrix, Mattermost, WeChat, and Beyond

Hermes supports more platforms than I can cover in depth. Here's a quick rundown:

- **Matrix** — The decentralized, open protocol. If you're running your own homeserver or using something like Element, Hermes can join your rooms.
- **Mattermost** — The open-source Slack alternative popular in security-conscious organizations.
- **WeChat (Weixin)** — Essential for anyone working with contacts in China.

- **Feishu (Lark)** — ByteDance's enterprise platform, widely used in Chinese organizations.
- **DingTalk** — Alibaba's enterprise communication tool, another major player in China.
- **WeCom** — Tencent's enterprise messaging platform.
- **BlueBubbles** — An open-source iMessage bridge that lets you use iMessage on non-Apple devices.

Each of these has its own setup process, typically following the same pattern: install the channel, provide authentication credentials, and start chatting. The hermes gateway setup command walks you through configuring each platform.

Additional delivery targets that aren't tied to a specific chat platform:

- **origin** — Sends output back to the channel where the conversation originally started. If you started a conversation on Telegram and a Cron job continues it, the response goes back to Telegram.
- **local** — Sends output to your local terminal. Good for Cron jobs you want to see in your server logs.
- **email** — Delivery via email integration.
- **sms** — Delivery via SMS integration.

The platform:chat_id and platform:chat_id:thread_id format lets you target specific conversations on any supported platform. For example, slack:C12345678 targets a specific Slack channel, and discord:9876543210:424242 targets a specific Discord thread.

12.6 Voice Across Channels

One of the most magical experiences with Hermes is talking to it. Not typing — talking. Sending a voice message on Telegram and hearing a voice message back. It transforms the interaction from "messaging a chatbot" to "talking to an AI assistant." Let's break down how voice works across channels.

STT Input: Voice Messages Become Text

When you send a voice message on a supported channel, Hermes uses **speech-to-text (STT)** to transcribe your audio into text. The agent then processes that text exactly as if you had typed it. You don't need to do anything special — just send a voice message instead of typing.

Telegram is the premiere channel for this. The integration is seamless: you hold the microphone, speak your question, and release. Hermes receives the audio, transcribes it, processes it, and responds. It feels like talking to a person.

Discord and WhatsApp also support audio input. On Discord, you'd send an audio attachment. On WhatsApp, voice messages work natively just like on Telegram.

The transcription quality depends on your STT provider, but modern speech recognition is remarkably good. I've had Hermes accurately transcribe mumbled voice messages sent from noisy coffee shops. The error rate is low enough that I routinely use voice instead of typing, even for complex questions.

TTS Output: Text Becomes Audio

On the output side, Hermes can use the **text_to_speech** tool to convert its text responses into audio. This is a tool your agent can invoke, and on supported channels, the audio gets delivered as a voice message or audio attachment.

Here's how it works on each channel:

- **Telegram**: The agent's response arrives as a voice message — the same format you use for sending audio. You tap to listen, just like any other voice note.
- **Discord**: The response comes as an audio attachment in the chat. Click to play.
- **WhatsApp**: Audio attachments, similar to Discord.
- **Other channels**: The text_to_speech tool generates audio, but how it's delivered depends on the channel's capabilities.

The text_to_speech tool is available across all channels, but the delivery format adapts to what each platform supports natively. On Telegram, you get the best experience because voice messages are a core feature of the platform. On channels that don't support audio attachments, the agent falls back to text.

Configuring Voice

Voice doesn't come out of nowhere — you need an STT provider and a TTS provider configured. These are typically set up as part of your agent's tool configuration. The specific providers depend on your setup, but the key point is that once you've configured them, voice works automatically on supported channels.

Here's what I recommend for voice configuration:

1. **Enable text_to_speech as a tool** in your agent configuration so it can generate audio when appropriate.
2. **Test with short messages first** before sending long voice notes. Make sure the transcription and generation are working correctly.
3. **Consider your audience** — not everyone wants audio responses. Some people prefer reading. You can configure your agent to default to text and only use TTS when explicitly asked.

Voice: A Personal Observation

I'll be honest: when I first set up voice on Telegram, I barely used it. It felt weird talking to my phone. But after a week, it became second nature. Now I send voice messages to Hermes on my commute, while cooking, walking the dog — any time my hands are busy. The convenience factor is enormous.

That said, there are situations where text is better. Complex code, detailed explanations, anything with formatting — these cry out for text. I use voice for quick, conversational interactions and text for anything I need to reference later. Your mileage may vary, but give voice a fair try before deciding it's not for you.

12.7 Cron Jobs — Scheduled Delivery to Channels

Cron jobs are one of Hermes's most powerful features. They let you schedule tasks — daily summaries, weekly reports, regular checks, recurring reminders — and deliver the results to whichever channel makes sense. You've seen Cron jobs in earlier chapters, but now we're going to connect them to channels.

Here's the key concept: every Cron job has a deliver field that determines where its output goes. If you don't specify deliver, the output goes to the local terminal. But when you specify a channel, the output gets pushed there instead.

Delivery Targets

Let's walk through the delivery options:

Deliver to Telegram:

```
cron:
  - schedule: "0 9 * * *"  # 9 AM daily
    task: "Give me a summary of today's calendar events"
    deliver: telegram
```

This sends the morning calendar summary straight to your Telegram bot chat. When you wake up and grab your phone, the summary is already there waiting for you.

Deliver to a specific Telegram chat:

```
deliver: telegram:1234567890
```

Use this when you want the output to go to a specific chat (like a group chat) rather than your private bot conversation.

Deliver to a Telegram topic:

```
deliver: telegram:-1001234567890:17585
```

This targets a specific topic within a Telegram group. Great for organized teams that use topics to separate discussions.

Deliver to Discord:

```
deliver: discord:#engineering
```

This sends output to the #engineering channel in your Discord server. I use this for daily deployment summaries — every morning at 7 AM, Hermes posts a summary of what was deployed overnight.

Deliver to the origin channel:

```
deliver: origin
```

This is one of my favorite options. origin means "send the output back to the channel where this conversation started." If you started a conversation on Telegram and a Cron job extends it, the response goes back to Telegram. If you started on Discord, it goes back to Discord. It's channel-aware, context-aware delivery.

Deliver to local terminal:

```
deliver: local
```

This sends output to the local terminal where Hermes is running. Useful for Cron jobs that are more about internal processing than user notifications, or when you want to see output in your server logs.

Other delivery targets:

```
deliver: whatsapp     # WhatsApp
deliver: slack       # Slack
deliver: signal      # Signal
deliver: homeassistant  # HomeAssistant
deliver: email       # Email
deliver: sms         # SMS
```

Each of these works the same way — you specify the platform, and Hermes handles the delivery.

Per-Job Delivery Configuration

The real power comes from mixing delivery targets across different Cron jobs. Here's a realistic configuration that demonstrates this:

```
cron:
  - schedule: "0 8 * * 1-5"  # 8 AM weekdays
    task: "Morning briefing: weather, calendar, and overnight
Slack messages"
    deliver: telegram

  - schedule: "0 18 * * 5"  # 6 PM Fridays
    task: "Weekly summary: tasks completed, blockers, and upcoming
deadlines"
    deliver: discord:#team-updates
```

```
  - schedule: "0 9 * * *"  # 9 AM daily
    task: "Check server health and report any anomalies"
    deliver: origin

  - schedule: "0 0 1 * *"  # Midnight on the 1st of each month
    task: "Monthly report: usage stats, costs, and
recommendations"
    deliver: slack
```

Four jobs, four delivery targets, one agent. Your morning briefing arrives on Telegram while you're commuting. Your weekly summary appears in the #team-updates Discord channel for everyone to see. Server health reports go back to wherever you were when you set up the job. Monthly reports post to Slack for your team.

This is the omnichannel promise in action. The agent adapts its delivery to match the context and audience of each task.

12.8 My Channel Mistakes — Cross-Platform Calamities

It's time for some honesty. I've made every mistake in this section, and I'm sharing them so you don't have to.

The Discord Bot That Wouldn't Shut Up

Early in my Hermes journey, I set up a Discord bot for my team's server. I was excited to see it working, so I configured it quickly, invited it to the server, and went to lunch. When I came back, I had 300+ notifications. My bot had responded to every single message in every single channel.

Someone said "good morning" in #general — the bot responded. Someone shared a meme in #random — the bot analyzed it. Someone asked a question in #help that was directed at a human — the bot butted in. It was chaos. My teammates were not amused.

The problem? I hadn't set require_mention: true.

```
discord:
  require_mention: false  # THIS WAS THE PROBLEM
  auto_thread: true
  reactions: true
```

With require_mention: false, the bot treated every message in every channel as something it should respond to. In a server with multiple channels and active conversations, that's a disaster. The fix was simple:

```
discord:
  require_mention: true  # ONLY respond when @mentioned
  auto_thread: true
  reactions: true
```

Now the bot only responds when someone explicitly types @Hermes or @my_bot_name. The noise evaporated overnight. Discord went back to being a place for humans, and the bot became a helpful resource you call on when you need it.

Lesson: Always set require_mention: true in Discord unless you have a very specific reason not to. That reason should involve a private channel where the bot is the only other participant, not a public server.

The Three-Minute Silent Voice Message

Another time, I was demonstrating Hermes's Telegram voice features to a friend. I configured the text_to_speech tool, tested it with a quick "Hello, world!" — worked great. Then I asked Hermes to explain the entire history of the Roman Empire. Verbatim. Out loud.

What I got back was a 3-minute voice message… of silence. Not quiet. Not muffled. Actual silence. Three minutes of nothing, followed by a confused text message from Hermes saying it had generated the audio but something went wrong.

The problem? My TTS provider had a character limit per request, and my prompt asked for way more than that limit. The tool generated a silent audio file of the correct duration but with no actual speech content. The text was too long for a single TTS call.

The fix was twofold:

1. I learned to ask for shorter spoken responses. "Explain the Roman Empire in 3-4 sentences" instead of "explain the entire history."

2. I configured my agent's system prompt to keep voice responses concise. When the output channel supports audio, my agent now defaults to brief, conversational responses rather than essay-length monologues.

Lesson: Voice messages have practical length limits — both technical and social. Nobody wants to listen to a 10-minute AI lecture as a voice message. Keep audio responses short and conversational. Save the detailed explanations for text.

Cron Delivery to the Wrong Channel

This one still makes me cringe. I had set up a Cron job to deliver a daily security report. The configuration looked like this:

```
cron:
  - schedule: "0 7 * * *"
    task: "Daily security audit: check logs, flag anomalies"
    deliver: discord:#security-alerts
```

Or so I thought. What I actually had was:

```
    deliver: discord:#general
```

For an entire week, detailed security reports — including IP addresses, failed login attempts, and vulnerability scans — were posted publicly in the #general channel. A channel that every team member, intern, and guest could see. My security reports were the talk of the office, and not in a good way.

I didn't notice because I have #general muted. My security reports were being delivered to a channel I never checked, and I assumed silence meant everything was fine. It was a teammate who finally pointed out that confidential security data was showing up in the public channel.

The fix was obvious — correct the channel name in the deliver configuration. But the deeper fix was implementing a checklist:

1. **Always test Cron delivery with a benign message first.** Before scheduling anything sensitive, set up a test job that delivers "This is a test" to the target channel. Confirm it shows up where you expect.

2. **Check the target channel after your first delivery.** Don't just assume it worked. Go to the channel and verify.

3. **Use the most specific channel possible.** Security reports go to #security-alerts, not #general. Sensitive data goes to private channels, not public ones.

4. **Double-check your config before saving.** It takes five seconds to reread one line. It takes a week to recover from the embarrassment of posting security reports publicly.

Lesson: Delivery target configuration is a single line that can have massive consequences. Treat it with the same care you'd treat a production database connection string.

Other Lessons Learned

A few more quick hits from the school of hard knocks:

- **Test in a private channel first.** Before deploying a bot to a public server with hundreds of members, test it in a private channel with just yourself. Make sure the configuration works the way you expect before exposing it to others.
- **Rate limits are real.** Telegram, Discord, and Slack all have rate limits. If your bot sends too many messages too quickly, it gets throttled or temporarily banned. Space out your Cron deliveries and don't set up 20 jobs that all fire at 9:00 AM.
- **Voice transcription varies by language.** STT accuracy depends on the language, accent, and background noise. If you're speaking a language other than English, test the transcription quality before relying on it for anything important.
- **Webhook URLs expire.** If you're using webhooks for any channel configuration, remember that they can expire or be rotated. If your bot suddenly stops responding, check the webhook first.
- **Backup your channel configs.** Your config.yaml is precious. Back it up. I once spent two hours reconfiguring a Discord bot because I accidentally overwrote my config file. Two hours I'll never get.

Hands-On Exercise: Set Up Two Channels and a Cron Job

Time to put this into practice. In this exercise, you'll set up two channels (I recommend Telegram and Discord, but use whatever makes sense for your situation), send a message from each, and configure a Cron job that delivers to one of them.

Part 1: Set Up Telegram

1. Open Telegram and search for @BotFather. Send the command /newbot.
2. Follow the prompts to name your bot. Choose any display name and a username ending in bot.
3. Copy the bot token BotFather gives you.
4. In your terminal, run:

- $ hermes gateway setup

1. Paste your bot token when prompted.
2. Open your new bot in Telegram and send it a message: “What channels am I connected to?”
3. Hermes should respond, and you should see Telegram listed in the output.

Part 2: Set Up Discord

1. Go to the Discord Developer Portal and create a new application.
2. Under the “Bot” tab, click “Add Bot” and copy the bot token.
3. Enable the **Message Content Intent** under “Privileged Gateway Intents.”
4. Under “OAuth2 > URL Generator,” select the bot scope and check these permissions: Send Messages, Read Message History, Create Threads. Copy the URL and add the bot to your server.
5. In your terminal, run:

- $ hermes gateway setup

1. Paste your bot token when prompted.
2. In your config.yaml, add (or update) the Discord configuration:

- discord: require_mention: true auto_thread: true reactions: true

1. In your Discord server, @-mention your bot: “@YourBotName hello!”

2. The bot should respond.

Part 3: Verify Both Channels

Run this command to confirm both channels are connected:

```
$ hermes gateway status
```

You should see CLI, Telegram, and Discord listed. If you do, congratulations — your agent now lives in three places.

Part 4: Send a Cross-Channel Message

From your Telegram bot, send: “What channels are you connected to right now?”

Notice that the response includes all connected channels — your agent knows about both Telegram and Discord even though you’re only talking to it on Telegram.

Part 5: Configure a Cron Job with Channel Delivery

Add a Cron job that delivers a daily message to your Telegram bot:

```
cron:
  - schedule: "*/5 * * * *"  # Every 5 minutes (for testing)
    task: "Give me a one-sentence hello and the current time"
    deliver: telegram
```

I’m using every 5 minutes for testing so you don’t have to wait long to see the result. Once you’ve confirmed it works, change the schedule to something reasonable like "0 9 * * *" for 9 AM daily.

Save your config, then verify the Cron job is running:

```
$ hermes cron list
```

Wait a few minutes, and a message should arrive on your Telegram bot with a greeting and the current time.

Part 6: Clean Up

Once you’ve confirmed everything works, update your Cron schedule to something that won’t annoy you:

```
cron:
  - schedule: "0 9 * * *"  # 9 AM daily
```

```
    task: "Morning briefing: date, weather, and any upcoming
events"
    deliver: telegram
```

You can also experiment with delivering to Discord:

```
cron:
  - schedule: "0 9 * * 1-5"  # 9 AM weekdays
    task: "Morning briefing"
    deliver: discord:#general
```

Replace #general with whatever channel you want to target.

Try It Now

Your challenge: Before moving on to the next chapter, complete these three mini-tasks:

1. **Send a voice message** on Telegram (or an audio message on your supported channel of choice) and verify that Hermes transcribes and responds to it.

2. **Change your Cron delivery target** from telegram to discord:#general (or another channel you've set up). Confirm the output arrives in the new location.

3. **Test deliver: origin** by starting a conversation on one channel, scheduling a quick test Cron job with deliver: origin, and verifying that the output comes back to the same channel where you started.

If all three work, you've got a fully omnichannel agent. The same AI, the same knowledge, the same tools — available wherever you need it, whenever you need it.

That's the power of Hermes channels. Your agent isn't trapped in a terminal. It's out there, in your pocket, on your server, in your team's workspace, ready to help on whatever platform feels most natural. And if you ever mess up a configuration — and you probably will, because we all do — just remember: require_mention: true, keep voice messages short, and always double-check your delivery targets.

In the next chapter, we'll dive deeper into automation and see how to build workflows that chain multiple tools and channels together into something truly powerful. But for now, go talk to your bot on your phone. It's a pretty cool feeling.

Key takeaways from Chapter 12:

- *Channels let your agent meet users where they already are — Telegram, Discord, WhatsApp, Slack, Signal, HomeAssistant, and more.*
- *CLI is the richest channel with full feature access; other channels adapt features to their platform's capabilities.*
- *Voice works through STT (speech-to-text) input and TTS (text-to-speech) output, with Telegram offering the most seamless voice experience.*
- *Cron jobs can deliver output to any channel using the deliver: configuration key, including specific chats and threads.*
- *Always set require_mention: true on Discord, keep voice responses short, and double-check delivery targets before going live.*
- *The hermes gateway status command lists your connected channels; hermes gateway setup walks you through configuring new ones.*

Chapter 13: Cron Jobs — Hermes on Autopilot

You wake up, pour your coffee, and check your phone. There it is — a neatly formatted digest of overnight server alerts, sitting in your Telegram chat like a loyal assistant who never sleeps. Below it, a summary of the top five AI news stories from the past 24 hours, delivered to your Discord channel at exactly 9 AM. You didn't type a single command. You didn't even open your laptop. Hermes handled it all while you were dreaming.

That's the power of cron jobs. And by the end of this chapter, you'll have Hermes running tasks on autopilot too.

13.1 Why Cron? — Autopilot for Your AI

Here's the thing about being human: we need sleep. We need weekends. We sometimes — just sometimes — step away from our keyboards. But the systems we monitor don't take breaks. Servers crash at 3 AM. News breaks on Saturday mornings. Competitors publish blog posts while you're at dinner.

Before cron jobs, here's how my mornings used to go: wake up, open my laptop, spend 30 minutes checking server dashboards, scrolling news feeds, and reviewing logs. Every. Single. Day. Even on vacation. Even when I was sick. I had become a human cron job, and I was terrible at it — inconsistent, grumpy before coffee, prone to missing things.

Cron jobs flip that script entirely. Instead of you adapting to your systems' schedules, your AI adapts to yours. You tell Hermes what to do and when to do it, and then you go live your life. The work happens whether you're watching or not.

The name "cron" comes from the Unix world — it's been the standard task scheduler on Linux systems since the 1970s. The word itself comes from "chronos," the Greek word for time. Hermes borrows this concept but makes it dramatically more powerful: instead of just running shell scripts on a timer, your cron jobs run fully capable AI agents with skills, tools, and delivery targets.

Here are some real-world use cases that people are running right now:

- **Daily digest**: Summarize news from RSS feeds every morning at 9 AM
- **Server health checks**: Check disk space, memory, and CPU usage every 30 minutes
- **Weekly reports**: Compile and email a project status report every Monday at 9 AM
- **Blog monitoring**: Scan a competitor's website for new posts every 2 hours
- **Smart home automations**: Check temperature thresholds and run security scans on a schedule
- **Content creation**: Generate social media post drafts daily at 6 PM

Each of these runs autonomously. No one types a prompt. No one clicks a button. Hermes wakes up, loads the skills it needs, runs the task, and delivers the result to the right place. Let me show you exactly how that works.

13.2 Creating Your First Cron Job

I'm going to show you a working cron job before I explain a single parameter. Here's one that sends you a morning news digest every day at 9 AM:

```
cronjob(
  action='create',
  name='morning-news-digest',
  schedule='0 9 * * *',
  prompt='Summarize the top 5 technology news stories from today.
For each story, provide the headline, a 2-sentence summary, and a
link if available. Format the output as a numbered list with clear
headers.',
  deliver='telegram'
)
```

Run that, and Hermes creates a scheduled job. Every day at 9 AM, a fresh Hermes session wakes up, reads that prompt, searches for current tech news, summarizes the top five stories, and delivers the result straight to your Telegram chat. You get your morning briefing without lifting a finger.

Now let's break down what just happened.

The cronjob Tool Anatomy

The cronjob tool is your single entry point for all cron operations. It takes an action parameter that tells Hermes what you want to do, plus a set of supporting parameters depending on the action. Here's the full signature:

```
cronjob(
  action='create',      # create, list, update, pause, resume,
remove, run
  name='',               # human-friendly name
  schedule='',           # '30m', 'every 2h', '0 9 * * *', or ISO
timestamp
  prompt='',             # full self-contained prompt (the task
instruction)
  skills=[],             # ordered list of skill names to load
before executing
```

```
  job_id='',              # required for update/pause/resume/
remove/run
  model={},              # per-job model override {provider, model}
  deliver='',             # delivery target
  script='',             # path to Python script under ~/.hermes/
scripts/
  repeat=0               # 0=once (one-shot), omit for recurring
)
```

We'll cover each parameter in depth throughout this chapter. For now, let's focus on the three you just used: name, schedule, and prompt.

name — This is a human-friendly label for your job. It's how you'll identify the job when you list or update it later. Pick something descriptive: morning-news-digest is better than job1.

prompt — The task instruction that Hermes will execute each time the job runs. This is the most important parameter, and it has a critical constraint: **it must be entirely self-contained**. I'll explain why in a moment, but the rule is simple: write your prompt as if whoever reads it has zero prior context — because they do.

schedule — When and how often the job runs. This is where things get interesting.

Schedule Formats

Hermes supports three different ways to express a schedule, and you can pick whichever makes sense for your use case.

Natural language is the easiest to read and write:

```
schedule='30m'          # every 30 minutes
schedule='every 2h'     # every 2 hours
```

These are perfect for recurring intervals. '30m' means "run this every 30 minutes, starting from when you create it." 'every 2h' means every two hours. Simple, readable, no reference manual needed.

Cron syntax gives you precise control:

```
schedule='0 9 * * *'    # every day at 9:00 AM
schedule='0 9 * * 1'    # every Monday at 9:00 AM
schedule='*/15 * * * *'  # every 15 minutes
```

Cron syntax follows the standard five-field format: minute, hour, day of month, month, day of week. If you've ever used a crontab on a Linux system, this is the same thing. It's powerful for expressing complex schedules like "the first Monday of every month at noon" or "weekdays at 9:30 AM."

ISO timestamps are for one-shot jobs:

```
schedule='2024-12-25T09:00:00'  # Christmas morning at 9 AM, once
```

When you use an ISO timestamp, you're saying "run this exactly once, at this specific date and time." This is perfect for scheduled reminders, one-time reports, or anything that doesn't need to repeat. You'll also want to set repeat=0 to make it explicit that this is a one-shot job:

```
cronjob(
  action='create',
  name='christmas-greeting',
  schedule='2024-12-25T09:00:00',
  prompt='Write a warm holiday greeting message for the team...',
  deliver='discord:#general',
  repeat=0
)
```

What's the difference between repeat=0 and just omitting repeat? Setting repeat=0 explicitly marks the job as one-shot — it runs once and it's done. Omitting repeat means the job is recurring. For most schedule formats (natural language, cron syntax), recurring is the default and what you want. For ISO timestamps, you almost always want repeat=0.

Self-Contained Prompts: The Most Important Rule

Here's the single most important thing to understand about cron jobs: **every job runs in a fresh session with no current-chat context.**

When you have a regular conversation with Hermes, there's history. You say "check the server" and Hermes knows which server because you discussed it ten minutes ago. Cron jobs don't have that luxury. When the scheduler fires off your job at 3 AM, it starts a brand-new session. No prior messages. No memory of what you talked about yesterday. No context whatsoever.

This means your prompt must contain everything Hermes needs to do the job. Compare these two prompts:

Bad (depends on prior context):

```
Check the server and tell me if anything looks wrong.
```

Good (entirely self-contained):

```
Check the server at myserver.example.com by running a health check
on the following endpoints:
- https://myserver.example.com/health
- https://myserver.example.com/api/status
- https://myserver.example.com/api/memory

For each endpoint, report the HTTP status code and response time.
If any endpoint returns a non-200 status or has a response time
over 5 seconds, flag it as an issue. Summarize the overall health
status at the top.
```

The bad prompt would fail in a cron job because “the server” is ambiguous without context. The good prompt works because it specifies exactly what to do, where, and what the criteria are.

This doesn’t mean your prompts need to be long — they just need to be complete. A prompt like “List the current weather in New York, London, and Tokyo” works fine because it has all the necessary information inline.

Running a Job Manually

After creating a cron job, you probably want to test it before waiting for the schedule to kick in. That’s what the run action is for:

```
cronjob(
  action='run',
  job_id='abc123'
)
```

This immediately triggers the job as if its schedule had just fired. You’ll see the output right in your current conversation, so you can verify the prompt does what you expect, the delivery target works, and the results look right. I test every new cron job this way before trusting it to run on its own.

Where does the job_id come from? You get it when you create the job (Hermes returns it), or you can find it using the list action, which we’ll cover in section 13.4.

13.3 Delivery Targets — Where Results Go

A cron job that runs perfectly but sends the result nowhere is like a tree falling in an empty forest. The whole point of automation is that results show up where you need them, without you having to go look for them.

The deliver parameter controls where Hermes sends the final response of the job. Let me show you before we dive into the details:

```
cronjob(
  action='create',
  name='server-health-alerts',
  schedule='30m',
  prompt='Check the health endpoints of production server at
https://prod.example.com/health. Report any issues.',
  deliver='slack'
)
```

When this job runs, Hermes will deliver its response directly to your Slack workspace. You'll see the health report pop up in Slack every 30 minutes, no manual checking required.

All Delivery Targets

Hermes supports over 15 delivery targets, covering the platforms you're already using:

Target	Description
origin	The original channel where the conversation started
local	Your local terminal
telegram	Your Telegram bot (default chat)
discord:#engineering	A specific Discord channel
slack	Your Slack workspace
whatsapp	WhatsApp
signal	Signal
matrix	Matrix
mattermost	Mattermost
homeassistant	Home Assistant
dingtalk	DingTalk
feishu	Feishu (Lark)

Target	Description
wecom	WeCom (WeChat Work)
email	Email
sms	SMS
bluebubbles	BlueBubbles (iMessage bridge)

That's a lot of options, and there's a good reason: different information belongs in different places.

Choosing the Right Target

origin is the simplest option. It delivers the result back to whatever channel you were using when you created the job. If you're chatting in Telegram and you create a cron job with deliver='origin', the results come back to that same Telegram chat. This is great for personal automation where you want everything in one place.

local delivers to your local terminal. This is useful during testing, or for jobs that produce output you'll pick up programmatically. The result just shows up in your shell session.

Platform targets like telegram, slack, discord, and the rest deliver directly to those platforms. This is where cron jobs really shine — you get notified in the tool you're already checking anyway.

Targeting Specific Chats and Topics

Some platforms support more granular targeting. Instead of just delivering to "Telegram," you can deliver to a specific Telegram topic:

```
deliver='telegram:-1001234567890:17585'
```

That breaks down as telegram:<chat_id>:<thread_id>. This lets you deliver to specific group chats or specific topics within a group.

For Discord, you can target a specific channel:

```
deliver='discord:#engineering'
```

This sends the result to the #engineering channel in your Discord server, which is exactly where server health alerts should go — not mixed into your DMs where they'll get lost.

The philosophy is simple: match the delivery target to the audience. Server alerts go to #engineering. Daily summaries go to your personal Telegram. Smart home notifications go to Home Assistant. Weekly reports go to email. Choose the channel where the information will actually be seen and acted upon.

13.4 Managing Jobs — List, Update, Pause, Resume, Remove

Creating cron jobs is exciting. Managing them over time is where the real discipline comes in. Jobs need updating, pausing, and sometimes removing. Let's walk through each operation.

List: See All Your Jobs

```
cronjob(action='list')
```

This returns a summary of every cron job you've created — including the job ID, name, schedule, status (active or paused), and delivery target. It's your dashboard for understanding what's running.

I check my job list once a week to make sure I'm not running stale jobs that I've forgotten about. Over time, it's easy to accumulate a graveyard of jobs that seemed like a good idea at the time but are now just burning through API calls for no reason. The list action is how you find those and clean them up.

Update: Modify an Existing Job

Let's say your morning news digest needs a schedule change. You originally set it for 9 AM, but now you want it at 7 AM because you're an early riser (good for you, by the way). You don't need to delete and recreate the job — just update it:

```
cronjob(
  action='update',
  job_id='abc123',
  schedule='0 7 * * *'
)
```

You can update any parameter: the schedule, the prompt, the delivery target, the skills, the model — basically anything you set during creation. The one thing you need is the job_id, which you get from the list action.

Important note about updating skills: if you pass skills=[] during an update, it clears all attached skills from the job. This is a deliberate design choice — an empty list means "no skills," not "keep the existing skills." If you want to keep the current skills, simply omit the skills parameter from your update call.

Pause and Resume: Temporary Stoppage

Sometimes you don't want to delete a job, you just want it to shut up for a while. Maybe you're on vacation. Maybe the thing the job monitors is under maintenance. Maybe it's the weekend and you don't want work notifications.

```
# Pause it
cronjob(
  action='pause',
  job_id='abc123'
)
```

The job stays configured but stops running. When you're ready to bring it back:

```
# Resume it
cronjob(
  action='resume',
  job_id='abc123'
)
```

The job picks up right where it left off on its original schedule. No data lost, no configuration changed. I use this all the time — I have a weekly report job that I pause during holidays and resume afterward.

Remove: Permanent Deletion

When you're truly done with a job, remove it:

```
cronjob(
  action='remove',
  job_id='abc123'
)
```

This is permanent. The job definition is deleted and won't run again. If you're not sure whether you'll need the job later, consider pausing it instead. There's no undo for remove.

Run: Manual Trigger for Testing

We covered this earlier, but it's worth emphasizing: always test a new job before trusting it to run on schedule.

```
cronjob(
  action='run',
  job_id='abc123'
)
```

The run action executes the job immediately, exactly as if its scheduled time had arrived. You see the output in your current session, and the delivery target also receives the result. This is your safety net — run the job manually, verify it works, and then let the schedule take over.

I've caught countless prompt issues this way. A prompt that's missing a crucial detail. A delivery target that's misspelled. A skill that's not quite right. Better to find out in a manual test than to discover it at 3 AM when the results never show up.

13.5 Skills and Scripts — Enhancing Cron Jobs

A vanilla cron job — one with just a prompt — can do a lot. But when you combine prompts with skills and scripts, you unlock a whole new level of automation. Let me show you what I mean.

Skills: Loading Specialized Capabilities

The skills parameter lets you specify which skills to load before the prompt runs. Skills are loaded in the order you list them, and then the prompt is executed. This means the AI agent running your cron job has access to all the tools and knowledge from those skills.

```
cronjob(
  action='create',
  name='weekly-security-scan',
  schedule='0 9 * * 1',
  prompt='Run a security scan on all production endpoints. Check
for SSL certificate expiry, open ports on common services, and any
changes to the DNS records since last week. Report any findings
```

```
that require attention.',
  skills=['security-audit', 'dns-monitor', 'ssl-checker'],
  deliver='discord:#security'
)
```

When this job fires, Hermes loads the security-audit skill first, then dns-monitor, then ssl-checker. After all three skills are loaded, it processes the prompt with all those capabilities available. The order matters if skills depend on each other — make sure foundational skills come before ones that build on them.

Why not just put skill instructions in the prompt? Because skills include tools, API connections, and specialized knowledge that can't be expressed in a simple text instruction. The skills parameter ensures those capabilities are properly installed and available before the task starts.

Scripts: Python for Data Collection

The script parameter points to a Python script that runs before the prompt. The script's standard output (stdout) gets injected into the prompt as additional context. This is incredibly powerful for data collection and change detection.

Scripts live in ~/.hermes/scripts/. When you specify a script, you give a path that resolves under this directory. So if you create a script at ~/.hermes/scripts/check_prices.py, you'd reference it as check_prices.py:

```
cronjob(
  action='create',
  name='price-monitor',
  schedule='every 2h',
  prompt='Based on the current prices in the data below, identify
any items that have dropped by more than 10% since the last check.
List the item name, old price, new price, and percentage drop.',
  script='check_prices.py',
  deliver='telegram'
)
```

Here's what happens when this job runs:

1. Hermes executes check_prices.py
2. The script's stdout (perhaps a JSON blob of current prices) gets captured
3. That output is injected into the prompt as context

4. Hermes processes the combined prompt and generates the response
5. The response is delivered to Telegram

This two-step pattern — script collects data, then prompt interprets it — is one of the most effective cron job architectures I've seen. The script handles programmatic tasks like API calls, web scraping, or data transformation. The prompt handles the interpretation, summarization, and communication.

Here's what a simple script might look like:

```
# ~/.hermes/scripts/check_prices.py
import requests
import json

response = requests.get('https://api.example.com/prices')
data = response.json()

# Format as readable context for the prompt
print("CURRENT PRICES:")
for item in data['items']:
    print(f"  {item['name']}: ${item['price']}")
    print(f"    Previous: ${item['previous_price']}")
    print(f"    Change: {item['change_percent']}%")
```

The print() output is what gets injected into the prompt. Keep your scripts focused and efficient — they're running automatically, so speed matters.

The Power Combination: Skills + Scripts + Prompts

The real magic happens when you combine all three:

```
cronjob(
  action='create',
  name='comprehensive-server-monitor',
  schedule='30m',
  prompt='Analyze the server metrics below. If any metric exceeds
its threshold, create a detailed incident report and post it to
the #incidents channel. For metrics within normal range, just
confirm the server is healthy.',
  skills=['server-monitor', 'incident-reporter'],
  script='collect_metrics.py',
  deliver='discord:#engineering'
)
```

When this fires: 1. collect_metrics.py gathers current server metrics (CPU, memory, disk, response times) 2. The script output becomes context data in the prompt 3. The server-monitor skill provides domain expertise for interpreting server metrics 4. The incident-reporter skill gives Hermes the ability to create and format incident reports 5. The prompt ties it all together, telling Hermes what to do with the information 6. The result delivers to Discord's #engineering channel

This is automation at a level that would have required a full DevOps team just a few years ago. Now it's a few lines of configuration.

13.6 Model Overrides — Stability and Cost Control

By default, cron jobs inherit whatever model your Hermes instance is configured to use. But there are times when you want a different model for a specific job. That's where the model parameter comes in:

```
cronjob(
  action='create',
  name='daily-summary',
  schedule='0 9 * * *',
  prompt='Summarize the top 5 news stories about artificial
intelligence from the past 24 hours.',
  deliver='telegram',
  model={'provider': 'openai-codex', 'model': 'gpt-4o-mini'}
)
```

The model parameter takes a dictionary with provider and model keys. This overrides the default model for this specific job.

Why Override the Model?

There are two main reasons: stability and cost.

Stability: If you're running a critical monitoring job that needs consistent output format, pinning a specific model ensures that a provider update doesn't suddenly change how your results look. Your weekly report job was working great with GPT-4o, and then the provider updates something and suddenly the format is different. Pinning the model prevents that surprise.

```
model={'provider': 'openai', 'model': 'gpt-4o'}
```

Cost control: Not every job needs the most powerful model. Simple formatting, basic checks, and routine summaries can often be handled by a smaller, cheaper model. If you're running a job every 30 minutes, the cost difference between models adds up fast:

```
# Routine health check — doesn't need GPT-4
cronjob(
  action='create',
  name='health-ping',
  schedule='30m',
  prompt='Check if https://example.com/health returns a 200 status
code. Respond with "OK" if healthy, or "ALERT: [details]" if
not.',
  deliver='slack',
  model={'provider': 'openai-codex', 'model': 'gpt-4o-mini'}
)

# Weekly strategic analysis — this one needs the best model
available
cronjob(
  action='create',
  name='weekly-analysis',
  schedule='0 9 * * 1',
  prompt='Analyze this week\'s performance metrics against our
quarterly goals. Identify gaps, recommend specific actions, and
project end-of-quarter outcomes.',
  model={'provider': 'openai-codex', 'model': 'gpt-4o'}
)
```

The first job runs 48 times a day. Using a smaller model for that frequency is a no-brainer. The second job runs once a week and needs deep analysis — worth the cost of a larger model.

When to Override and When to Inherit

Override the model when: - The job runs very frequently and the task is simple (use a cheaper model) - The job is critical and output format must be consistent (pin a specific model) - You want to experiment with a different model for one job without changing your global settings

Inherit the default when: - You're just getting started and don't need to optimize yet - The job uses skills that benefit from the most capable model available - Cost isn't a concern for infrequent jobs

My advice: start with the default model. Optimize later once you understand your actual usage patterns. Premature optimization is the root of all kinds of headaches.

13.7 My Cron Mistakes — Automation Gone Rogue

I've been using cron jobs with AI agents long enough to make every mistake in the book. Let me share my three most spectacular failures so you can avoid them.

The Every-5-Minute Twitter Bot That Got Rate-Limited

I thought I was being diligent. I had a cron job that checked a specific Twitter account for new posts and DM'd me a summary. Seemed reasonable, right? Except I set the schedule to 5m — every 5 minutes. That's 288 API calls per day. For a single account. That I checked roughly once a week.

Everything was fine for the first day. Then the rate limits kicked in. Twitter's API has limits, and I was blowing through them like they didn't exist. The job started failing. Not gracefully failing either — it was throwing errors, which triggered my error-alerting cron job (yes, I had one of those too), which was also running frequently, creating a cascade of notifications.

By the time I noticed, I had 200+ failed job notifications in my Telegram and a completely exhausted API quota.

The fix was simple: I changed the schedule to every 2h. That's 12 calls a day instead of 288. Same useful information, 96% fewer API calls.

The lesson: **think carefully about the minimum viable frequency for your job.** Just because you can check every 5 minutes doesn't mean you should. Most monitoring tasks are perfectly fine with 30-minute or hourly intervals. Start slow, then increase frequency only if you're actually missing things.

The Self-Referential Prompt

This one still makes me cringe. I had a great conversation with Hermes where we discussed analyzing my website's traffic. In that conversation, I defined what "normal" traffic looked like, which pages to monitor, and what thresholds constituted an alert. It was thorough. It was detailed. It was brilliant.

So I created a cron job with this prompt:

```
Check the website traffic based on what we discussed and flag
anything unusual.
```

See the problem? “Based on what we discussed.” The cron job runs in a fresh session. No conversation history. No memory of that wonderful chat. Hermes had absolutely no idea what “normal” traffic looked like, which pages to check, or what thresholds I cared about. The job ran, produced a generic and unhelpful response, and delivered it to my Slack channel every morning for three days before I noticed it was useless.

Three days of completely empty reports delivered like clockwork..Automation is great at automating failure, too.

The fix was writing a self-contained prompt that included all the context:

```
cronjob(
  action='create',
  name='traffic-monitor',
  schedule='0 8 * * *',
  prompt='Check the website at mysite.example.com. Visit /
analytics and extract: (1) total visits yesterday, (2) top 5 pages
by visit count, (3) bounce rate. Flag any metric that changes more
than 20% from these baselines: total visits > 500 (alert if below
400), bounce rate < 60% (alert if above 75%). Format as: HEALTH
REPORT header, then each metric with status (OK/ALERT), then
overall assessment.',
  deliver='slack'
)
```

That prompt is long, but it works. It contains everything Hermes needs, every time, with no assumptions about prior context.

The lesson: **every cron job prompt must be self-contained.** Write it as if the reader has never heard of you, your project, or your goals. Because they haven’t. The fresh session has no memory of your brilliant conversations.

The Cron Job That Created More Cron Jobs

This is my favorite one, in a “learning from disaster” kind of way. I wanted to be clever. I set up a cron job that monitored my server and, if it detected an issue, created a more frequent monitoring cron job to keep a closer eye on things until the issue resolved.

Sounds smart, right? Adaptive monitoring. Escalation based on severity.

Here's what actually happened: the initial job detected a minor issue (a brief CPU spike that resolved itself in 30 seconds). It created a new job to monitor every 5 minutes. That new job ran, detected another minor fluctuation (because 5-minute checks are noisy), and — you guessed it — created another job. Which detected another issue. Which created another job.

Within an hour, I had 47 cron jobs running, each one spawning new jobs faster than I could delete them. It was like a cancer of automation. My API costs skyrocketed. My delivery channels were flooded with alerts. And every alert said essentially the same thing: "CPU is at 32%, which is fine, but let me create a job to keep watching."

I had to manually list all jobs, identify the rogue ones, and remove them one by one. Then I added a very clear rule to all my cron prompts:

The lesson: **cron jobs should not create other cron jobs.** Period. Full stop. If you need adaptive monitoring, implement it within a single job's logic — check more frequently in the same prompt, adjust thresholds, escalate to a human — but do not have an automated system create new automated systems. It's a recipe for recursive chaos.

Here's my personal safety rule that I now add to every cron prompt:

```
SAFETY RULE: Do not create, modify, or schedule any new cron jobs.
If escalation is needed, report findings and let a human decide.
```

Simple, clear, and it prevents the exact scenario I described above.

Configuration: wrap_response

There's one configuration option worth knowing about for cron jobs. In your Hermes configuration file, you can set:

```
cron:
  wrap_response: true
```

When wrap_response is enabled, Hermes wraps the cron job's output in a clean, formatted container before delivering it. This ensures that the delivered message includes context about which job produced it,

when it ran, and other metadata. Without it, the delivery just contains the raw response, which can be confusing when you have multiple jobs sending to the same channel.

I keep this enabled. It's much easier to understand a message that says "Result from job 'morning-news-digest' (ran 2024-03-15 at 09:00:00)" than to get a raw block of text with no attribution.

Hands-On: Your First Scheduled Job

Time to get your hands dirty. We're going to create a cron job, test it, update it, and manage it through the full lifecycle. This exercise uses only the tools and parameters we've covered in this chapter.

TRY IT NOW

Step 1: Create a Daily Job

Let's create a simple daily job that checks a website and reports its status. We'll start with a manual test before committing to a schedule.

```
cronjob(
  action='create',
  name='daily-site-check',
  schedule='0 9 * * *',
  prompt='Visit https://example.com and verify the site is
accessible. Report: (1) HTTP status code, (2) approximate page
load time, (3) whether the page title contains "Example Domain".
Format as a brief status report with an overall HEALTHY/UNHEALTHY
assessment.',
  deliver='local'
)
```

Write down the job_id that Hermes returns — you'll need it for the next steps. Let's say it returns job_abc123.

Step 2: Test with a Manual Run

Don't wait until 9 AM to find out if your job works. Test it now:

```
cronjob(
  action='run',
  job_id='job_abc123'
)
```

Check the output. Does it make sense? Is the prompt self-contained enough that a fresh session understood what to do? If the output looks wrong, update the prompt (see Step 3). If it looks good, proceed.

Step 3: Update the Schedule

Let's say you want the report at 8 AM instead of 9 AM, and you want to add a skill for more detailed monitoring:

```
cronjob(
  action='update',
  job_id='job_abc123',
  schedule='0 8 * * *'
)
```

You can also update the prompt if needed:

```
cronjob(
  action='update',
  job_id='job_abc123',
  prompt='Visit https://example.com and verify the site is
accessible. Report: (1) HTTP status code, (2) approximate page
load time, (3) whether the page title contains "Example Domain",
(4) check if the SSL certificate is valid and its expiry date.
Format as a brief status report with an overall HEALTHY/UNHEALTHY
assessment.'
)
```

Step 4: Pause the Job

Going on vacation? Pause the job so it doesn't keep running while you're away:

```
cronjob(
  action='pause',
  job_id='job_abc123'
)
```

Verify it's paused:

```
cronjob(action='list')
```

You should see the job's status as paused.

Step 5: Resume the Job

Back from vacation? Resume it:

```
cronjob(
  action='resume',
  job_id='job_abc123'
)
```

Step 6: Remove the Job (Optional)

When you're done experimenting, clean up:

```
cronjob(
  action='remove',
  job_id='job_abc123'
)
```

Congratulations — you've just taken Hermes through the full cron job lifecycle: create, test, update, pause, resume, and remove. These are the exact operations you'll use every day as you build out your automated workflows.

Quick Reference: Cron Job Actions

Action	Purpose	Required Parameters
create	Schedule a new job	name, schedule, prompt
list	See all jobs	None
update	Modify an existing job	job_id + any parameters to change
pause	Temporarily stop a job	job_id
resume	Restart a paused job	job_id
remove	Delete a job permanently	job_id
run	Manually trigger a job now	job_id

Cron CLI Commands

You can also manage cron jobs from the terminal using the hermes cron CLI:

```
hermes cron list       # Show all jobs
hermes cron create     # Create a job interactively (alias: hermes
cron add)
hermes cron edit       # Edit an existing job
hermes cron pause      # Pause a job
```

```
hermes cron resume     # Resume a paused job
hermes cron run        # Manually trigger a job
hermes cron status     # Show detailed status of a specific job
hermes cron tick       # Run all due jobs once and exit (useful
for testing)
hermes cron remove     # Delete a job (aliases: rm, delete)
```

The CLI is handy for quick checks and one-off management. For programmatic creation with all parameters, use the cronjob() tool from within a Hermes conversation.

Quick Reference: Schedule Formats

Format	Example	Meaning
Natural language	'30m'	Every 30 minutes
Natural language	'every 2h'	Every 2 hours
Cron syntax	'0 9 * * *'	Daily at 9:00 AM
Cron syntax	'0 9 * * 1'	Every Monday at 9:00 AM
Cron syntax	'*/15 * * * *'	Every 15 minutes
ISO timestamp	'2024-12-25T09:00:00'	One-shot: Dec 25, 2024 at 9 AM

Quick Reference: Delivery Targets

Target	Use Case
origin	Deliver back to the conversation where the job was created
local	Deliver to your local terminal (good for testing)
telegram	Send to your Telegram bot
telegram:-1001234567890:17585	Send to a specific Telegram group/topic
discord:#engineering	Send to a specific Discord channel
slack	Send to your Slack workspace
email	Send via email

Target	Use Case
sms	Send via SMS

Plus: whatsapp, signal, matrix, mattermost, homeassistant, dingtalk, feishu, wecom, bluebubbles

The cron job system turns Hermes from a tool you use into an assistant that works for you around the clock. Start simple — a daily check, a weekly summary, a single monitoring task. Test thoroughly with the run action. Keep your prompts self-contained. And never, ever let a cron job create another cron job.

In the next chapter, we'll shift from automation to something equally important: security. Approvals, secrets redaction, sandbox validation, PII protection — everything you need to run Hermes safely without losing sleep.

Chapter 14: Security — Keeping Hermes Safe

It was 2 AM on a Tuesday. I had been wrestling with a deployment script for three hours, and I just wanted it done. So I switched Hermes to auto mode — full autopilot, no approval needed for any command. Hermes cheerfully ran a cleanup script I had drafted, and within seconds, it wiped out my entire staging database. Not the test data. The staging database. The one with the demo accounts that my team was presenting to a client the next morning.

That single mistake cost me a night of rebuilding and a very uncomfortable conversation with my manager. It also taught me more about Hermes security than any documentation ever could.

In this chapter, I am going to walk you through every layer of Hermes' security system — the approval gates, the secret redaction, the sandbox guard, the network boundaries, and the privacy controls. By the end, you will know how to configure Hermes so it is both powerful and safe. And you will learn from my mistakes so you do not have to make them yourself.

14.1 Why Security Matters — Trust but Verify

Here is the fundamental tension of working with an AI agent: Hermes is incredibly useful precisely because it can do so much. It can read files, write files, run shell commands, browse the web, and interact with APIs. That power is what makes it valuable. It is also what makes it dangerous.

Think about it this way. If you give a person full access to your computer — root shell, network access, file system, everything — you would want some guardrails, right? You would not just hand over the keys and walk away. You would want to approve the scary actions. You would want secrets hidden. You would want a log of what happened.

Hermes is that person. It has shell access. It can execute commands. It can read and write any file you can. And while Hermes is designed to be helpful and cautious, it is an AI — it can misunderstand your intent, misinterpret a request, or simply make a wrong call. Security in Hermes is not about distrusting the AI. It is about building a system where mistakes are caught before they become disasters.

The Worst Case

Without any security configuration, here is what can go wrong:

- **Accidental deletion.** You ask Hermes to clean up old files, and it removes something critical.
- **Leaked secrets.** Your API key ends up in a conversation log that gets shared or stored.
- **Unauthorized access.** Hermes navigates to an internal company URL and exposes sensitive data.
- **Unreviewed actions.** A command runs automatically that you would have stopped if you had seen it first.

None of these are hypothetical. I have personally experienced two of them, and I have watched colleagues hit the others.

Defense in Depth

Hermes uses a defense-in-depth approach to security. That means multiple independent layers, each catching what the previous one might miss:

1. **Approval system** — The human gates dangerous actions.
2. **Secrets redaction** — Sensitive values are stripped before they enter logs.
3. **Tirith sandbox** — Commands are validated against safety rules before execution.
4. **Browser privacy** — Network boundaries prevent access to internal resources.
5. **Logging** — Everything is recorded so you can audit what happened.
6. **Session management** — Automatic resets prevent stale sessions from accumulating sensitive data.

No single layer is perfect. Together, they form a net that catches most mistakes before they cause real harm. Let me walk you through each one.

14.2 The Approval System — Your Safety Switch

Picture this: you are asking Hermes to reorganize your project directory. Hermes decides the best approach is to move everything into a new structure, which involves running mv commands on dozens of files. In manual mode, Hermes pauses before each move and shows you exactly what it plans to do. You review each one. You catch that it is about to move your .env file into a public directory. You say no. Disaster averted.

That is the approval system in action.

The Three Modes

Hermes offers three approval modes, each with a different balance between safety and speed:

```
approvals:
  mode: manual       # manual (default), suggest, auto
  timeout: 60         # seconds to wait for approval
```

Manual mode is the default, and for good reason. Every dangerous command — anything that modifies files, runs shell commands, or makes network requests — requires your explicit approval before Hermes executes it. Hermes will show you the command, explain what it does, and wait for you to say yes or no. This is the safest mode, and it is where you should start, especially when you are new to Hermes.

Suggest mode is the middle ground. Hermes will suggest actions and show you what it would do, but it will not execute them without your go-ahead. The difference from manual mode is subtle but important: in manual mode, Hermes asks for permission for each potentially dangerous step. In suggest mode, Hermes presents a plan and waits for you to approve the whole thing or request changes. It is faster for workflows where you trust the plan but still want oversight.

Auto mode is full autopilot. Hermes executes commands without asking for approval. It is faster, no question about it. But it is also riskier, because there is no human in the loop to catch mistakes. I will tell you more about when auto mode is appropriate — and when it absolutely is not — later in this chapter.

Changing Modes Mid-Session

You are not locked into one mode forever. You can switch between modes at any time, even mid-session. The command is straightforward:

```
hermes config set approvals.mode auto
```

Replace auto with manual or suggest as needed. This is incredibly useful when you are doing something like this:

- Start in manual mode while you explore a new task.
- Switch to auto mode for a batch of repetitive, safe operations (like renaming 50 files with a clear pattern).
- Switch back to manual mode when you hit sensitive operations.

I do this all the time. The key is to be intentional about when you go to auto. Never switch to auto mode just because you are tired of clicking approve. That is exactly how my staging database got wiped.

Approval Timeouts

The timeout: 60 setting tells Hermes how long to wait for your approval before giving up. If you do not respond within 60 seconds, Hermes cancels the pending action and lets you know.

This is a safety feature, not an inconvenience. Imagine you step away from your desk while Hermes is waiting for approval on a destructive command. Without a timeout, that command would sit in a pending state indefinitely — and if you accidentally pressed enter while scrolling back through your terminal, you might approve something you did not intend. The timeout ensures that unapproved actions do not hang around waiting to cause trouble.

If 60 seconds feels too short — maybe you are working on something and constantly getting timeouts — you can increase it:

```
approvals:
  timeout: 120         # give me two minutes to review
```

But be honest with yourself about why you need more time. If you are constantly running up against the timeout, it might mean you need to slow down and review more carefully, not that you need a longer window.

14.3 Secrets Redaction — Hiding Your Keys

Let me show you a scenario. You are debugging an API integration, and you ask Hermes to show you the environment variables it is using. In your .env file, you have:

```
AWS_ACCESS_KEY_ID=AKIA3EXAMPLEKEY12345
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
DATABASE_URL=postgres://admin:SuperSecret123@db.example.com:5432/
production
```

With redact_secrets: true (the default), Hermes will show you something like:

```
AWS_ACCESS_KEY_ID=[REDACTED]
AWS_SECRET_ACCESS_KEY=[REDACTED]
DATABASE_URL=postgres://admin:[REDACTED]@db.example.com:5432/
production
```

Those secrets never appear in the conversation. They never get written to the chat log. They never show up if someone screenshots your session. The redaction happens automatically, before the text is displayed or stored.

How It Works

Hermes uses pattern detection to identify secrets. When redact_secrets is enabled, it scans outgoing text for known patterns of sensitive information:

```
security:
  redact_secrets: true          # automatically redact API keys,
passwords
```

The patterns it checks for include:

- **API keys** — Strings that match common API key formats (long alphanumeric strings with specific prefixes like sk-, AKIA, etc.)
- **AWS credentials** — Access key IDs and secret access keys matching AWS patterns
- **Database passwords** — Passwords embedded in connection strings like postgres://user:password@host
- **Bearer tokens** — Authorization headers and tokens
- **Generic secrets** — Values assigned to variable names that suggest they are secret (anything with PASSWORD, SECRET, TOKEN, KEY in the name)

What Gets Redacted vs. What Does Not

The pattern-based approach is powerful, but it has limits. Here is what you need to know:

Gets redacted: - Values of well-known environment variables like AWS_SECRET_ACCESS_KEY, DATABASE_PASSWORD - Strings matching API key formats - Passwords in database URLs and connection strings - Bearer tokens in HTTP headers

Might not get redacted: - A secret stored in a variable with an unusual name like MY_SPECIAL_VALUE - A password embedded in a JSON blob where the key name does not match expected patterns - Secrets passed as positional arguments where there is no variable name to clue in the detector

This is an important caveat. Secrets redaction is a safety net, not a guarantee. It catches the most common patterns, but a determined (or careless) user can still leak a secret if they store it in an unexpected format.

The best practice is simple: **use environment variables for API keys and never put secrets directly into your prompts or into Hermes' memory.** If a secret does not need to be in the conversation, do not put it there.

An Edge Case Worth Knowing

Here is one that trips people up. Suppose you have an environment variable called APP_CONFIG that contains a JSON string with embedded secrets:

```
{"api_key": "sk-abc123def456", "region": "us-east-1"}
```

Whether this gets redacted depends on whether the pattern detector recognizes sk-abc123def456 as an API key format. If the key follows a recognized pattern, the value will be redacted. If it does not — if it is just a random string that does not match any known key format — it might slip through.

The lesson: **always verify that your specific secrets are being redacted.** Do not assume. Test it.

14.4 Tirith — The Sandbox Guard

Imagine Hermes is about to run a command that looks perfectly fine on the surface: rm -rf /tmp/old_builds/. But what if there is a symlink in that directory pointing to your home folder? The command would follow the symlink and start deleting files you never intended to lose.

Now imagine that before Hermes runs any command, a separate program inspects it and says, "Wait, this command removes files recursively. Let me check the safety rules." That program is Tirith.

What Tirith Does

Tirith is a command validation sandbox. When it is enabled, Hermes sends every command to Tirith for approval before executing it. Tirith checks the command against a set of safety rules and returns a verdict: allow or deny.

```
security:
  tirith_enabled: true           # Tirith sandbox for command
validation
  tirith_path: tirith           # path to tirith binary
  tirith_timeout: 5             # seconds before Tirith gives up
  tirith_fail_open: true        # if Tirith fails, allow command
(safer default)
```

The tirith_path setting tells Hermes where to find the Tirith binary. If you have installed Tirith in a standard location, the default tirith should work. If you have installed it somewhere custom, point to it:

```
security:
  tirith_path: /usr/local/bin/tirith
```

Fail Open vs. Fail Closed

Here is the critical design decision in Tirith: what happens when Tirith cannot validate a command? Maybe Tirith crashes. Maybe it takes too long to respond. Maybe it is not installed correctly. In any of these cases, Hermes needs to decide: should the command be allowed or denied?

That is what tirith_fail_open controls.

tirith_fail_open: true (the default) — If Tirith cannot validate the command, Hermes allows it to run. The reasoning is that most of the time, Tirith failing is a configuration issue, not a sign of danger. You do not want a broken Tirith installation to paralyze your entire Hermes session. This is the safer *default* for usability, but it means that when Tirith is down, you lose that safety layer.

tirith_fail_open: false — If Tirith cannot validate the command, Hermes denies it. Nothing runs without Tirith's explicit approval. This is the more secure option, and it is what you should use in production or high-security environments. The tradeoff is that if Tirith has any issues, Hermes basically stops working until you fix it.

Which should you choose? It depends on your threat model:

- **Development, personal projects** — fail_open: true is fine. The approval system is your primary safety net, and Tirith is a bonus layer.
- **Production, shared environments, sensitive data** — fail_open: false is the right call. You want to ensure that no command ever runs without validation, even if that means occasionally being blocked by a Tirith issue.

The 5-Second Timeout

The tirith_timeout: 5 setting gives Tirith 5 seconds to respond. If Tirith takes longer than 5 seconds to validate a command, Hermes treats it as a validation failure and falls back to the fail_open behavior.

Why 5 seconds? It is a balance. Tirith validation should be fast — it is checking rules, not running the command. If it takes more than 5 seconds, something is probably wrong. But you also do not want the timeout so short that normal validation gets cut off on a slow machine.

If you are running on older hardware and notice that Tirith regularly times out on legitimate validation attempts, you can increase the timeout:

```
security:
  tirith_timeout: 10            # give Tirith more time on slow
machines
```

Just remember: a longer timeout means a longer delay between Hermes deciding to run a command and actually running it. For most workflows, 5 seconds is plenty.

14.5 Browser and Network Safety

Here is a scenario that happened to a colleague of mine. They were using Hermes to scrape documentation from a website. Hermes, being helpful, followed a link that pointed to an internal company wiki — wiki.internal.company.com. The internal wiki required authentication, but it was reachable from the machine Hermes was running on. Hermes accessed it, pulled down a page full of proprietary information, and displayed it in the conversation. That conversation log, with all the proprietary data, was now sitting in a file on disk.

This is the kind of thing that browser privacy settings are designed to prevent.

Blocking Private URLs

```
browser:
  allow_private_urls: false    # block private/internal URLs
```

When allow_private_urls is set to false (the default), Hermes will refuse to navigate to internal or private network addresses. This includes:

- Private IP ranges (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16)
- Localhost and loopback addresses (127.0.0.1, localhost)
- Internal domain names (.internal, .local, .internal.company.com)
- Link-local addresses

This is a critical safeguard. Your local machine can probably reach all sorts of internal services — databases, admin panels, internal APIs, wikis — that should never be accessed by an automated agent. By blocking private URLs, Hermes ensures it stays on the public internet and does not accidentally wander into your company's internal infrastructure.

If you specifically need Hermes to access an internal resource — for example, you are using it to test an internal API — you can enable private URLs:

```
browser:
  allow_private_urls: true     # allow internal access (use with
caution)
```

But do this deliberately and temporarily. Turn it on for the specific task, then turn it back off.

Website blocklist — For additional control over which websites Hermes can access, use the security.website_blocklist setting:

```
security:
  website_blocklist:
    enabled: false       # enable to block specific domains
    domains: []           # list of domains to block (e.g.,
["facebook.com", "twitter.com"])
    shared_files: []      # path to shared blocklist files
```

This is different from allow_private_urls — that blocks private network addresses, while website_blocklist lets you block specific public domains. Useful if you want to prevent Hermes from accessing social media, competitor sites, or any other specific domains.

Recording Browser Sessions

```
browser:
  record_sessions: false       # don't record browser sessions by
default
```

When record_sessions is false, Hermes does not save a record of what it did in the browser. No screenshots, no DOM snapshots, no navigation history is persisted. This is the privacy-friendly default — your browsing activity through Hermes is ephemeral.

There are times when you want recording enabled. If you are debugging a scraping workflow, for instance, having session recordings can be invaluable for figuring out what went wrong. But think carefully before turning it on, because those recordings capture everything — including any sensitive data that might appear on the pages Hermes visits.

Common Browser Security Scenarios

Here are a few real-world scenarios and how the browser settings handle them:

1. **Hermes tries to access your local database admin panel at http://localhost:8080/admin** — Blocked by allow_private_urls: false.
2. **You ask Hermes to check documentation on a public website** — Allowed. Public URLs are fine regardless of the private URL setting.
3. **Hermes follows a redirect from a public URL to an internal URL** — Blocked. Even if the initial URL was public, the redirect to an internal address gets caught.
4. **You need Hermes to test an internal staging API** — Temporarily set allow_private_urls: true, run your tests, then set it back to false.

14.6 Privacy and Logging

Let's say you are using Hermes in a corporate environment. You are helping a customer troubleshoot an issue, and their name, email, and account number keep coming up in the conversation. You are confident that the conversation logs are secure on your machine, but what if they get shared? What if someone reviews the logs for quality assurance?

This is where privacy settings and logging controls come in.

PII Redaction

```
privacy:
  redact_pii: false        # redact personally identifiable
information
```

By default, PII redaction is **off**. This might surprise you, but the reasoning is straightforward: PII detection is imprecise. It can flag legitimate data as PII and redact things you need to see, or it can miss PII that does not match its patterns. Keeping it off by default means you get full visibility, and you can enable it when you know you are dealing with sensitive personal data.

When you enable it:

```
privacy:
  redact_pii: true         # redact personally identifiable
information
```

Hermes will attempt to detect and redact common categories of PII: names, email addresses, phone numbers, Social Security numbers, credit card numbers, and similar identifiers. Like secrets redaction, this is pattern-based, and it is not perfect. But it adds another layer of protection when you are working with real people's data.

If you are in healthcare, finance, or any regulated industry, you should strongly consider enabling PII redaction. It is not a complete compliance solution, but it reduces the chance of sensitive data persisting in logs.

Logging Levels

```
logging:
  level: INFO              # DEBUG, INFO, WARNING, ERROR, CRITICAL
```

```
  max_size_mb: 5           # max log file size before rotation
  backup_count: 3          # number of rotated log files to keep
```

The level setting controls how much detail Hermes records. The levels, from most to least verbose, are: DEBUG, INFO, WARNING, ERROR, CRITICAL.

- **DEBUG** — Everything. Every command, every API call, every internal decision. Useful for troubleshooting but produces enormous logs.
- **INFO** — Normal operations. Commands executed, approvals granted, files read. This is the default and a good balance.
- **WARNING** — Things that seem wrong but are not fatal. Approvals timed out, Tirith validation failed, suspicious commands detected.
- **ERROR** — Things that broke. Commands that failed, API errors, file access problems.
- **CRITICAL** — System-level failures. Hermes could not start, configuration is broken.

For day-to-day work, INFO is the right level. Switch to DEBUG only when you are actively troubleshooting a problem, and switch back when you are done. Debug logs are tempting because they have everything, but that "everything" includes a lot of noise — and potentially a lot of sensitive data.

Log Rotation

The max_size_mb: 5 and backup_count: 3 settings control log rotation. When a log file reaches 5 MB, Hermes rotates it — the current log becomes a backup, and a fresh log file starts. It keeps up to 3 backup logs. That means your total log storage is at most about 20 MB (5 MB current + 3 × 5 MB backups).

If you need more history for auditing or debugging, you can increase these:

```
logging:
  max_size_mb: 10          # larger log files before rotation
  backup_count: 5          # keep more backups
```

But remember: larger logs mean more data stored on disk, and potentially more sensitive information sitting in files. Size your logs for your actual needs, not "just in case."

Session Reset

```
session_reset:
  mode: both              # 'time', 'idle', or 'both'
  idle_minutes: 1440       # 24 hours of inactivity
  at_hour: 4                # reset at 4 AM local time
```

Session reset is an often-overlooked security feature. It automatically clears conversation state, reducing the amount of sensitive data that persists over time.

The mode setting determines what triggers a reset:

- **time** — Reset at a specific time every day (controlled by at_hour).
- **idle** — Reset after a period of inactivity (controlled by idle_minutes).
- **both** — Reset on whichever condition is met first. This is the default and the most thorough option.

With the default configuration, Hermes will reset your session at 4 AM every day, or after 24 hours of inactivity — whichever comes first. This means sensitive data in your conversation history does not stick around forever.

The 4 AM time is chosen because it is typically off-peak. If you work unusual hours, adjust it to a time when you are unlikely to be in the middle of something:

```
session_reset:
  at_hour: 2                # reset at 2 AM instead
```

And if you find that 24 hours of idle time is too long for your security needs, shorten it:

```
session_reset:
  idle_minutes: 60          # reset after 1 hour of inactivity
```

Checkpoints

```
checkpoints:
  enabled: true         # save conversation state snapshots
  max_snapshots: 50     # keep last 50 snapshots
```

Checkpoints are related to session reset. When enabled, Hermes periodically saves snapshots of your conversation state. This is useful for recovery — if something goes wrong, you can roll back to a previous checkpoint.

The max_snapshots: 50 setting limits how many snapshots are kept. Older snapshots are automatically deleted. This prevents checkpoint storage from growing without bound, and it limits the amount of sensitive conversation data sitting on disk.

If you are in a high-security environment and want to minimize stored conversation data, you can reduce this number or disable checkpoints entirely:

```
checkpoints:
  enabled: false        # no saved conversation snapshots
```

But be aware: disabling checkpoints means you lose the ability to roll back to a previous state. It is a tradeoff between audit capability and data minimization.

14.7 Command Allowlist — Restricting What Hermes Can Run

There's one more safety feature worth knowing about: command_allowlist. By default, this is empty, meaning Hermes can run any command it decides is necessary. But in some environments, you want tighter control.

```
command_allowlist:
  - git
  - npm
  - python3
  - docker
  - ls
  - cat
  - grep
```

When configured, Hermes will only run commands that match entries in this list. If Hermes wants to run curl but curl isn't on the list, the command is blocked. This is a blunt instrument — it can break workflows where Hermes needs unexpected commands — but for high-security environments or public-facing bots, it provides an additional guardrail.

Most users don't need this. The approval system already gives you control over dangerous commands. The allowlist is for cases where you want absolute certainty about what can and cannot run, regardless of what the LLM decides.

14.8 Human Delay — Appearing Natural

This section is about a different kind of safety — social safety. When Hermes operates in chat platforms like Slack, Discord, or Microsoft Teams, the way it responds can affect how people perceive and interact with it.

Why Human-Like Timing Matters

Imagine you are in a Slack channel with a colleague, and you send them a complex question. If they reply with a detailed, thoughtful answer in 0.1 seconds, you know something is off. Humans do not respond that fast. When Hermes responds instantly, it can feel jarring or suspicious to the humans in the channel. It can also trigger anti-bot detection in some platforms.

Human delay settings make Hermes' timing feel more natural:

```
human_delay:
  mode: off              # off, typing, reading
  min_ms: 800
  max_ms: 2500
```

The Three Modes

off — No delay. Hermes responds as fast as it can. This is the default because it is the most efficient and because human delay is only relevant in chat platform contexts. If you are using Hermes in a terminal or API context, there is no reason to slow it down.

typing — Hermes simulates the delay of a human typing a response. After generating a response, Hermes waits a random amount of time (between min_ms and max_ms) before sending it. During this delay, the chat platform may show a "typing" indicator, making it look like someone is composing a message.

reading — Hermes simulates the delay of a human reading an incoming message before responding. When it receives a message, it waits briefly before processing it, mimicking the time a human would spend reading and thinking.

Configuring the Delay Range

The min_ms: 800 and max_ms: 2500 settings define the range of the random delay. Each response gets a random delay within this range. The default range of 800ms to 2500ms (0.8 to 2.5 seconds) feels reasonably natural for most chat interactions.

If you want even more natural variation:

```
human_delay:
  mode: typing
  min_ms: 500
  max_ms: 3500
```

Or if you want it to be subtle:

```
human_delay:
  mode: typing
  min_ms: 1000
  max_ms: 1500
```

When to Use and When to Skip

Use human delay when: - Hermes is operating in a shared chat platform where other humans are present. - You want to avoid triggering anti-bot detection. - The social dynamics of the conversation benefit from more natural pacing.

Skip human delay when: - You are using Hermes in a terminal or API context with no other humans. - Speed is a priority and there are no social considerations. - You are running automated tasks or scripts through Hermes.

Human delay is a small thing, but it can make a surprising difference in how smoothly Hermes integrates into your team’s communication flow.

14.9 My Security Mistakes — Lessons from the Vault

I promised you some stories about things I got wrong. Here they are, in painful detail, so you can learn from my failures without repeating them.

The Auto-Mode Disaster

I already told you about this one at the start of the chapter, but let me give you the full picture. I was working on a deployment script that cleaned up old build artifacts. The script was supposed to delete files in /tmp/old_builds/. I had tested it carefully in manual mode, step by step, and it worked perfectly. So I switched to auto mode to "save time" on the final run.

What I did not realize was that the script had a variable substitution issue. In my test environment, the build directory path was set correctly. In the staging environment, an environment variable was empty, and the rm -rf command ended up running with a path that resolved much more broadly than I intended. Auto mode did not give me a chance to review the actual command before it executed. Within seconds, the staging database was gone.

The lesson: Auto mode is for well-tested, repetitive operations where you have already verified the safety of each action. Never switch to auto mode the first time you run something. Never switch to auto mode because you are tired or impatient. If the actions are important enough that you would want to review them, they are important enough to stay in manual mode.

The API Key That Leaked Into Logs

This one was embarrassing. I was debugging a connection to a third-party API, and I asked Hermes to show me the request headers it was sending. Hermes helpfully printed out the full headers, including the Authorization: Bearer sk-proj-abc123... header with my actual API key.

I did not notice at first. The key was just sitting there in the conversation, and the conversation was being logged. The log file made it into a git commit. By the time I realized what had happened, the key had been in the repository for two days, and I had to rotate it on the provider's dashboard.

Here is the thing: redact_secrets was enabled. I had it set to true. So why was the key not redacted? Because the API key format I was using did not match the pattern detector's expectations. It was a newer key format from the provider that the redaction patterns had not been updated to catch.

The lesson: Secrets redaction is a net, not a wall. It catches many common patterns, but it does not catch everything. **Test your specific secrets** to make sure they are being redacted. And more fundamentally, never ask Hermes to display raw headers, environment variables, or configuration files containing secrets if you can avoid it. Use environment variables, and let the application pull them in at runtime rather than surfacing them in your conversation.

The Private URL That Got Accessed

My colleague's story from section 14.5 could just as easily have been mine. I was using Hermes to analyze competitor documentation, and it followed a link from a public blog post to what turned out to be an internal wiki of the competitor's company. Their wiki was accidentally exposed to the public internet — no authentication required. Hermes pulled down a page of internal product roadmap data and displayed it in our conversation.

allow_private_urls would not have helped here, because the URL was not technically private — it was on the public internet, just not intended to be public. The browser privacy settings protect against accessing your own internal network, not against accidentally stumbling onto someone else's accidentally exposed resources.

The lesson: Browser privacy settings are essential, but they are not a complete solution for data sensitivity. Be thoughtful about what you ask Hermes to browse. If you are asking it to scrape or follow links broadly, consider the possibility that it will encounter data that was not meant for you.

The Sum of All Lessons

If there is one theme running through all these stories, it is this: **security features are only as good as the person using them.** Hermes gives you powerful tools — approval gates, secret redaction, sandboxed execution, privacy controls — but none of them are automatic guarantees. They are layers in a defense-in-depth strategy, and every layer has gaps.

The key security principles that I now follow religiously:

1. **Start with manual approval mode.** Only switch to auto when you have verified each action individually first.
2. **Enable secrets redaction** (it is on by default, but verify it is working for your specific secrets).
3. **Keep Tirith enabled with fail-open** for development, fail-closed for production.
4. **Block private URLs by default.** Enable them only when you have a specific need and disable them immediately after.
5. **Review logs regularly.** Set aside time each week to check what Hermes has been doing.
6. **Never store secrets in prompts or memory.** Use environment variables for API keys and rotate them regularly. Environment variables don't protect stale credentials.

I cannot promise that following these principles will keep you perfectly safe. I can promise that ignoring them will eventually cause you problems.

Hands-On: Configure Security Settings

Time to get your hands dirty. This exercise walks you through setting up and verifying Hermes' key security configurations. You will set the approval mode, test secrets redaction, check your logs, and verify Tirith is working.

Step 1: Check Your Current Security Configuration

Start by seeing where you stand. Run:

```
hermes config show security
```

This will display your current security settings. Make note of what is enabled and what is not. The defaults should look like:

```
security:
  redact_secrets: true
  tirith_enabled: true
  tirith_path: tirith
```

```
  tirith_timeout: 5
  tirith_fail_open: true
```

Step 2: Set Your Approval Mode

If you are just starting with Hermes, make sure you are in manual mode:

```
hermes config set approvals.mode manual
```

Verify it took effect:

```
hermes config show approvals.mode
```

You should see manual. Practice a simple task — ask Hermes to list files in a directory, for example — and notice how it asks for approval before executing the command. Get comfortable with the approval flow before you consider changing modes.

Step 3: Verify Secrets Redaction

This is important. Create a test environment variable with a dummy API key:

```
export TEST_API_KEY="sk-test-1234567890abcdef"
```

Now ask Hermes to display environment variables that start with TEST_:

```
Show me the value of the TEST_API_KEY environment variable
```

Check the response. With redact_secrets: true, the key value should appear as [REDACTED] or similar. If it shows the full key, something is wrong with your redaction configuration.

Also check the conversation log after this interaction:

```
hermes logs tail
```

Search the log output for your test key. It should not appear in plain text anywhere.

Once you have verified redaction is working, unset the test variable:

```
unset TEST_API_KEY
```

Step 4: Check Your Logs

Review your current log configuration:

```
hermes config show logging
```

You should see something like:

```
logging:
  level: INFO
  max_size_mb: 5
  backup_count: 3
```

Now look at the actual logs:

```
hermes logs tail
```

Scan through recent entries. Look for any unexpected entries — commands you do not remember running, access to URLs you did not request, or any data that looks like it should have been redacted. This kind of regular log review is one of the best security habits you can develop.

Step 5: Test Tirith

Check that Tirith is running and configured:

```
hermes config show security.tirith_enabled
```

It should return true. Now try asking Hermes to run a command that should be blocked or flagged by Tirith. For example:

```
Run: rm -rf /tmp/test_directory
```

Depending on your Tirith configuration and safety rules, Tirith may flag this command as potentially dangerous. Pay attention to what happens — does Tirith validate the command? Does it flag it? How long does validation take?

If you want to see what happens when Tirith is unavailable, temporarily disable it:

```
hermes config set security.tirith_enabled false
```

Run the same command again and notice the difference in behavior. Then re-enable Tirith:

```
hermes config set security.tirith_enabled true
```

Step 6: Review Browser Privacy

Check your browser settings:

```
hermes config show browser
```

Make sure allow_private_urls is false and record_sessions is false. If either has been changed, reset them:

```
hermes config set browser.allow_private_urls false
hermes config set browser.record_sessions false
```

Step 7: Document Your Baseline

After completing all the steps above, record your security baseline. Write down your current configuration or save it to a file:

```
hermes config show security > hermes-security-baseline.yml
hermes config show approvals >> hermes-security-baseline.yml
hermes config show browser >> hermes-security-baseline.yml
hermes config show logging >> hermes-security-baseline.yml
hermes config show privacy >> hermes-security-baseline.yml
```

This baseline is your reference point. If something goes wrong in the future, or if you suspect a setting has been changed, you can compare against this baseline. Keep this file somewhere safe and revisit it periodically.

Try It Now

Before you close this chapter, do one more thing. Open your Hermes configuration file and do a full security audit. Ask yourself:

1. Is my approval mode set to manual? If not, why not?
2. Is secrets redaction enabled? Have I tested it with my actual API key formats?
3. Is Tirith enabled? What is my fail-open setting, and is it appropriate for my environment?
4. Are private URLs blocked? Are browser sessions unrecorded?
5. What is my logging level? Am I capturing enough information without capturing too much?
6. Is PII redaction appropriate for the data I am handling?

7. What is my session reset schedule? Is sensitive data lingering too long?

8. Is my command_allowlist configured? Do I need to restrict which commands Hermes can run?

If you cannot answer yes or give a clear reason for every setting, you have work to do. Do it now, before something goes wrong. Security is not a feature you configure once and forget. It is a practice you maintain.

I learned that the hard way. You do not have to.

In the final chapter, we'll step back and look at the bigger picture — where to go from here, how to keep learning, and how the Hermes community can help you push further.

Chapter 15: Beyond the Basics — Your Hermes Journey Continues

You made it.

Fourteen chapters ago, you might not have known what an AI agent was, let alone how to configure one, give it memory, teach it skills, and send it off to do real work on your behalf. Now here you are — with a working Hermes setup, a bag of tools, and (I hope) a growing sense of what this thing can really do.

But here's the thing about learning Hermes: finishing the book isn't finishing the journey. It's the opposite. Everything up to now has been runway. This chapter is the takeoff.

I want to do something different here. Instead of walking through features one by one, I want to show you what a real Hermes workflow looks like — all the pieces working together, in context, over time. Then I want to point you at the horizons you haven't explored yet, give you a practical path for building your personal toolkit, connect you to the community that will help you grow, and send you off with everything you need.

Let's get to it.

15.1 A Week with Hermes — A Complete Workflow

Books teach tools one at a time. Real life doesn't work that way. You don't use memory on Tuesday and the browser on Thursday in some perfectly sequenced curriculum. You use everything together, in a messy, organic way that builds on itself day after day.

So let me show you what a real week with Hermes looks like — not the textbook version, but the way it actually unfolds when you're living with this tool day to day.

Day 1: First Setup, First Conversation

You install Hermes. You run through the initial setup. You open your config.yaml for the first time and — okay, 50+ configuration keys is a lot. You don't need most of them on day one. You set your model, your API key, and maybe your name in user.md. That's it.

Then you have your first conversation. Maybe you ask it something simple. Maybe you ask it to explain something about your own project. Maybe you just poke at it to see what happens. That's fine. That's day one. You're learning the chat — interactive mode, the back-and-forth rhythm, the way Hermes responds when you give it a clear instruction versus a vague one.

The key insight on day one: Hermes is a conversation partner that actually remembers what you said three messages ago. Use that. Build up context before asking for the big thing.

Day 2: Memory and Skills — Building Your Knowledge Base

Day two is where it starts to get personal. You write your USER.md — your preferences, your coding style, the things you care about. You notice that Hermes starts remembering things across conversations because of MEMORY.md. It's no longer a blank slate every time you say hello.

And then you discover skills. You drop a SKILL.md file into ~/.hermes/skills/ and suddenly Hermes knows how to do something specific — really knows it, with instructions and context and the benefit of your

experience baked in. Maybe it's a skill for writing commit messages the way your team likes them. Maybe it's a skill for reviewing pull requests with a particular checklist in mind. Whatever it is, it's yours.

By the end of day two, Hermes isn't just a chatbot anymore. It's starting to feel like it knows you.

Day 3: Terminal and Files — Real Automation

Day three is when things get real. You ask Hermes to run a command in the terminal, and it does. You ask it to read a file, edit a specific function, and run the tests again. It does. You ask it to do something that takes a while — you set it to background mode, go work on something else, and check back when it's done. Process management. Foreground for quick things, background for long things, PTY mode when you need something interactive.

And the file tools — read, write, search, patch. You're not just talking about code anymore. You're changing it. Searching across your project for that function name you can never remember. Patching a configuration file without opening an editor. Asking Hermes to refactor something and watching it actually do it, not just suggest it.

This is the day it clicks for most people. This is the day Hermes stops being a fancy search engine and starts being a coworker.

Day 4: Browser Power — Research and Web Tasks

Day four, you need to research something. Maybe you're evaluating a library, checking the latest API documentation, or comparing competing solutions. Hermes has ten browser tools for web navigation and interaction. It can visit pages, follow links, extract information, fill in forms, and interact with web applications.

You watch Hermes navigate a documentation site, find the relevant section, extract the key details, and summarize them for you — all while you were getting coffee. (Okay, you were probably watching, but you could have been getting coffee.)

The browser tools turn Hermes from something that only knows what's in your local files into something that can access the entire web. That's a transformative shift.

Day 5: Delegation — Parallel Workflows

Day five, you realize you have more than one thing to do. Of course you do — you always have more than one thing to do. Hermes can handle that. Delegation lets you spin up subagents — single tasks handed off to focused workers, or batch delegation where Hermes takes a whole list of tasks and parallelizes them.

Need to review five pull requests, check the build status in three repositories, and update documentation across a project? That's not one long serial task anymore. That's five, three, and however many sub-tasks, all running in parallel. Delegation is how you stop working at human speed and start working at agent speed.

Day 6: Cron Jobs — Scheduled Automation

Day six changes your relationship with Hermes fundamentally. Cron jobs let you schedule tasks that run autonomously, on a schedule, without you even being there. Daily standup summaries. Weekly report generation. Nightly monitoring checks. Hourly sync operations.

You set up your first cron job and then you go to bed. In the morning, the report is waiting for you. Hermes did the work while you slept. That's not a metaphor. That's literally what happened.

Cron is the point where Hermes goes from "tool I use" to "system that works for me." There is a real, tangible difference between those two things. You'll feel it.

Day 7: Channels — Connecting Everywhere

Day seven, you connect Hermes to your life. Channels give you seven (or more) ways to reach Hermes: CLI, Telegram, Discord, WhatsApp, Slack, Signal, HomeAssistant. You're not limited to your terminal anymore. You can message Hermes from your phone on the train. You can have it pipe alerts into your team's Slack. You can control your smart home through HomeAssistant.

By the end of day seven, Hermes is everywhere you are. It's in your terminal, in your chat apps, in your home, in your scheduled tasks, in your memory, in your skills. It's not a single interaction point. It's an ecosystem you've built around your actual workflow.

And that's the point. Seven days. You went from "what is this thing?" to "I can't imagine working without it." Not because each individual tool is mind-blowing on its own — though some of them are — but because the tools compound. Memory makes skills better. Skills make delegation better. Delegation makes cron better. Cron makes channels better. Each piece amplifies the others.

That's the workflow. That's what a week with Hermes actually looks like.

15.2 Advanced Horizons — What's Next

You've covered the core. But Hermes has capabilities we haven't touched in this book — features that go beyond the basics, into territory that's more specialized, more powerful, and honestly, more fun. Let me point you at the horizons you can explore next.

MCP Integration: Connecting to External Services

Model Context Protocol (MCP) is how Hermes talks to the broader world of tools and services. Hermes ships with a native MCP client, and the mcporter CLI makes it straightforward to connect to MCP-compatible servers — databases, APIs, internal company services, anything that speaks the protocol.

What this means in practice: your Hermes instance isn't limited to the tools it ships with. Any service that exposes an MCP interface can become another tool in Hermes's toolkit. Your company's internal API? Connect it via MCP. A specialized database you query daily? MCP. A weather service, a stock ticker, a project management tool — if it has an MCP server, Hermes can use it.

This is how you go from a general-purpose AI agent to one that's deeply integrated with your specific workflow and infrastructure. MCP is the bridge.

Smart Model Routing: Cost Optimization at Scale

Here's a secret that catches people off guard: you don't need your most expensive, most capable model for every single task. Summarizing a document doesn't require the same reasoning power as debugging a race condition. Hermes knows this.

Smart model routing lets you configure which model handles which type of task. Push cheap, simple tasks to cheap models. Save your powerful (and expensive) model for the jobs that actually need it. Over time — especially at scale, especially with cron jobs running daily — this can make a meaningful difference in cost without sacrificing quality.

It's one of those features that sounds minor until you see your API bill at the end of the month.

Voice Features: TTS and STT

Hermes can speak and listen. Text-to-speech (TTS) support comes with four providers, so you can choose the voice quality and cost that works for you. Speech-to-text (STT) support comes with three providers. And there's voice recording capability built in.

Use cases? Imagine asking Hermes to read you your morning briefing while you're making coffee. Imagine dictating a complex task instead of typing it. Imagine Hermes responding to voice commands through your home automation setup. The voice features turn Hermes from a text-based tool into a multimodal one.

This isn't science fiction. It's configuration.

Obsidian Vault as Extended Memory

If you use Obsidian for note-taking, Hermes can integrate directly with your vault. This means your notes, your knowledge graph, your daily journals — all of it becomes accessible context that Hermes can draw on.

Think about what that means in combination with MEMORY.md. Your built-in memory handles the short-term, conversational stuff. Your Obsidian vault handles the long-term, archival stuff. Together, they give Hermes a depth of context that no single memory system could match.

If you're an Obsidian user, this integration alone is worth the setup time.

Platform Toolsets: Running on Different Platforms

Hermes isn't just a CLI tool. It runs across multiple platform interfaces, each with its own toolset optimized for that environment — CLI, Telegram, Discord, WhatsApp, Slack, Signal, and more. The tools you

use in the terminal aren't identical to the tools you use through Telegram, and that's by design — each platform has different constraints and different affordances.

What this means: you can choose the right platform for the right moment. Quick question from your phone? Telegram or WhatsApp. Team collaboration? Slack or Discord. Deep work session? CLI. Home automation? HomeAssistant. The platform toolsets make sure Hermes works well wherever you are, not just where it was originally designed to run.

Compression: Managing Context at Scale

When your conversations get long — and with Hermes, they will — context window management becomes a real concern. Compression is Hermes's way of handling this: intelligently summarizing and compressing earlier context so that the most relevant information stays available without running out of room.

You won't need to think about this most of the time. It happens automatically. But knowing it's there means you can trust long-running sessions and complex delegated tasks without worrying that Hermes will "forget" something important because it ran out of context space.

Auxiliary Models: The Specialist Team

Hermes can enlist up to eight auxiliary sub-models for specialized tasks: vision processing, web extraction, compression, and more. These aren't separate products or separate installations. They're part of the same system, called in when needed, handling the tasks they're best at.

It's like having a team of specialists on call. The main model handles the general reasoning and orchestration. The auxiliary models handle the edge cases that require specific expertise. You don't manage them explicitly — Hermes figures out when to call them in. But knowing they exist helps you trust the system with a wider range of tasks.

Code Execution with hermes_tools

For programmatic access, the hermes_tools Python library lets you execute code directly within Hermes workflows. This bridges the gap between “AI agent” and “development environment” in a way that feels natural. You’re not limited to what the built-in tools can do — if you can write Python, you can extend Hermes’s capabilities on the fly.

This is particularly powerful for data processing tasks, custom transformations, and anything that requires logic beyond what prompt engineering alone can deliver.

15.3 Building Your Personal Toolkit

I want to tell you a story about how I messed up with Hermes.

When I first started using it seriously, I tried to set up everything at once. I configured all the channels in one afternoon. I wrote five skills on day one. I set up three cron jobs before I’d even finished exploring what cron could do. I was so excited about the potential that I tried to realize all of it immediately.

The result was a mess. My skills were half-baked. One of my cron jobs had a configuration error and ran every five minutes instead of every five hours, and I didn’t notice for two days. My Telegram channel was connected but my user.md was basically empty, so Hermes didn’t know anything about me when I messaged it from my phone. Nothing was terrible, but nothing was good either. I’d built a house by putting up all the walls at once without making sure the foundation was solid.

I had to tear most of it down and start over. This time, I went slow. I used Hermes for a week with nothing but basic chat. Then I added memory. Then I added one skill — one that I’d tested thoroughly and actually needed. Then one cron job. Then the CLI channel. Each addition was deliberate, tested, and integrated into how I was already working.

That slower approach took longer, but every piece actually worked. Every piece made the next piece better. The foundation was solid because I built it that way.

So here’s the path I recommend — the progressive mastery path:

Start with the Basics

Use Hermes in interactive mode for at least a few days before you add anything. Get comfortable with the conversation. Learn how to give clear instructions. Learn what Hermes is good at and where it needs more guidance. Build your intuition for the tool before you start extending it.

Then Add Memory

Write a thorough USER.md. You don't need to write a novel, but put in the things that actually matter: your coding style, your project context, your preferences, the things you find yourself repeating in every conversation. Then let MEMORY.md do its job — it will accumulate context over time without you having to manage it explicitly.

Then Create Your First Skill

If you haven't made a SKILL.md file yet, make one. A good first skill is something you do repeatedly that has a clear structure. Code review is a classic. Project scaffolding. Meeting notes formatting. The skill doesn't need to be complex — it needs to be something you actually use. Drop it in ~/.hermes/skills/ and test it with a few conversations before you move on.

Then Set Up Your First Cron Job

Pick something that genuinely needs to happen on a schedule. A daily summary of a repository. A weekly report. A monitoring check. Configure it carefully, test it manually first, verify the output, and then — only then — set it to run automatically. Watch it for a few cycles. Make sure it's doing what you expect. Trust is earned, and that's doubly true for autonomous tasks.

Then Connect Your First Channel

Pick the channel you'll actually use the most. For me, that was the CLI — I live in the terminal. For you, it might be Telegram so you can reach Hermes from your phone, or Slack so it's part of your team workflow. Connect one, test it thoroughly, and make sure your memory and skills are solid before you add more. A channel connection works best when Hermes already knows who you are and how you work.

Then Keep Building

After that, add more as you need them. Delegate when you have parallel tasks. Try MCP when you want to connect to an external service. Explore voice when you want to interact without typing. Add the browser tools to your research workflow. Each addition should feel like a natural extension of what you're already doing, not a bolted-on feature.

The progressive path looks like this: chat, then memory, then skills, then automation, then channels, then advanced features. Each step makes sense on its own and enables the next one. That's not accidental. That's how the system is designed to grow with you.

Don't try to do it all at once. I did, and I wasted two days undoing my own enthusiasm. Build deliberately. Test each addition. Integrate it into your real workflow. Then move to the next thing.

15.4 The Community — You're Not Alone

One of the things that surprised me most about Hermes is the community around it. This isn't a niche tool used by a handful of enthusiasts. It's a movement. And that matters for you, because it means there are people out there who have been where you are, solved the problems you're running into, and built things you can learn from.

The GitHub Repository

Run hermes to find the source code, issues, and discussions. If you hit a bug, this is where you report it. If you have a question, this is where you ask it. If you want to understand how something works under the hood, this is where you read the code.

Browse the issues sometime. Not just the open ones — the closed ones too. You'll find answers to questions you didn't even know you had. You'll see how other people configured their setups, what problems they ran into, and how they solved them. It's like a living FAQ written by thousands of people over months of real-world use.

the Skills Hub: The Skills Marketplace

the Skills Hub is where skills go to be shared. It's a marketplace — not in the transactional sense, but in the community square sense. People build skills that solve their particular problems, and then they share them so others don't have to solve the same problems from scratch.

This is one of the most practical resources available to you. Before you write a skill from scratch, search the Skills Hub. Someone may have already written exactly what you need. And when you do write something useful, share it. That's how the ecosystem grows. Every skill contributed makes Hermes more useful for everyone.

The skills on the Skills Hub are also a learning resource. When you're trying to figure out how to structure a SKILL.md file or how to make a skill more robust, reading other people's skills is invaluable. You see patterns, techniques, and approaches that no documentation can fully capture. Real code from real use cases — that's how you learn.

Learning from Others

Beyond the official channels, there's a broader community of Hermes users sharing their experiences. Blog posts, tutorial videos, configuration guides, "how I set up Hermes for my team" writeups. These real-world perspectives are gold. They show you not just what the features are, but how people actually use them — the messy, practical, not-in-the-docs reality of daily work with an AI agent.

When someone posts about how they automated their entire development pipeline with Hermes cron jobs, or how they use delegation to manage a team of agents reviewing code, or how they connected Hermes to their company's internal tools via MCP — that's knowledge you can adapt to your own situation. You don't have to copy it exactly. You just have to understand the pattern and apply it to your context.

Contributing Your Own Skills

The community isn't a one-way street. You'll start as a consumer — finding skills on the Skills Hub, reading issues, learning from others. But at some point, you'll have something to share. A skill you wrote that solved a problem nobody else had tackled. A configuration pattern that made your workflow smoother. A workaround for a gotcha that wasted your afternoon.

When that moment comes, share it. Write it up clearly, put it on the Skills Hub or post it in the discussions, and let other people benefit from what you learned. This isn't altruism (though it is that too). It's how you become part of the ecosystem. The people who contribute are the people who learn the most, because explaining something forces you to understand it deeply.

The Hermes community is large, active, and welcoming. You don't need permission to participate. You just need something to share — and after reading this book, you have plenty.

15.5 Goodbye and Good Luck

Let me recap where you've been.

You started with nothing but curiosity. You installed Hermes, configured it, and had your first conversation. You learned the chat modes — interactive, one-shot, resume — and how each one fits a different kind of work. You gave Hermes memory through MEMORY.md and USER.md, and watched it become something that knows you across sessions.

You created skills — SKILL.md files that encode your expertise and make Hermes better at the things you care about. You used the file tools — read, write, search, patch — to let Hermes actually change your code, not just talk about it. You opened the terminal and ran commands, managed processes, and discovered that background mode means you don't have to wait.

You sent Hermes to the web with ten browser tools, extracting information and navigating sites without you. You delegated tasks to subagents — single and batch — and felt the speed of parallel work. You set up cron jobs and went to sleep while Hermes kept working. You connected channels across multiple platforms and reached Hermes from anywhere.

And through all of that, you configured the system — 50+ keys in config.yaml, tuned and tweaked to your needs — while trusting the security model: approvals, Tirith, secrets redaction, and privacy controls that keep your data safe.

That's a lot. Take a moment to appreciate how far you've come.

Now let me tell you what I really believe about Hermes.

Hermes is a tool amplifier, not a replacement. It does not replace your judgment, your creativity, or your expertise. What it does is amplify them. When you know what you want and you tell Hermes clearly, it can execute faster, search broader, and persist longer than you can alone. But the knowing what you want — that's still you. The judgment about whether the result is good enough — that's still you. The creative insight that reframes the problem — that's still you.

Hermes makes you more effective, not less necessary. The people who get the most out of this tool are the people who bring the most to it — clear thinking, domain expertise, and the willingness to iterate until it's right.

Start small. I know I've said this throughout the book, but I'm saying it one more time because it's the single most important piece of advice I can give. Don't try to automate your entire workflow on day one. Don't try to configure every channel, write every skill, set up every cron job in a single weekend. Start with one thing. Get it working. Understand it. Trust it. Then add the next thing.

Iterate. Your first skill will not be perfect. Your first cron job might not work exactly right. Your first channel connection might feel awkward. That's normal. Fix it. Improve it. Try again. The gap between "first attempt" and "actually useful" is almost always a few iterations, and those iterations are where the real learning happens.

Build confidence. Each small success makes the next attempt easier. Each working cron job makes you more willing to set up another. Each shared skill makes you more inclined to write the next one. Confidence isn't something you're born with — it's something you build, one working feature at a time.

And look at the future. AI agents are going to get more capable. The models will get smarter, the tools will get richer, the integrations will get deeper. What you've learned in this book isn't a snapshot of a static technology — it's a foundation for something that will keep evolving. The patterns you've internalized — how to give clear instructions, how to build skills, how to manage autonomous tasks, how to think about security and cost — those patterns will serve you regardless of what the models look like in two years or five.

The tools will change. The principles won't.

I want to close with something personal. Writing this book has been a journey of its own. I've been using AI agents for a while now, but sitting down to explain every feature, every configuration option,

every gotcha — that forced me to understand them more deeply than I ever had before. There were moments when I realized that a feature I'd been using casually had implications I'd never considered. There were moments when I had to go back to the documentation, back to the source code, back to the community, because my understanding wasn't good enough to explain clearly.

And there was one very specific moment — I won't say which chapter — where I wrote an explanation, tested it, found out I was wrong, and had to rewrite the whole section. That was humbling. But it was also exactly the point. If I'm asking you to iterate and learn from mistakes, I'd better be willing to do the same.

This book is better because of those corrections. Your Hermes setup will be better because of your corrections, too. Don't fear them. They're not failures. They're the process.

So here you are. At the end of the book, at the beginning of your real Hermes journey. You have the knowledge. You have the tools. You have the community. You have a path forward — progressive, deliberate, one solid step at a time.

Go forth and build something amazing.

I mean that literally. Build something. Don't just read about Hermes — use it. Don't just configure it — create with it. Don't just connect it — make it part of how you work, every day, in ways that matter to you.

The best Hermes setup isn't the one with the most channels or the most skills or the most cron jobs. The best Hermes setup is the one that makes your actual, real, daily work better. The one that saves you time on tasks you hate. The one that handles the routine so you can focus on the creative. The one that works so seamlessly you almost forget it's there — until you try to work without it and realize how much you've come to rely on it.

That's the destination. And now you have the map.

Good luck. You're going to do great things.

— The journey doesn't end here. It starts here.