FIELD GUIDE

THE OPERATOR'S GUIDE TO HERMES AGENT

Build an AI assistant that can act, remember, and improve

TONY SIMONS

BUILT AROUND HERMES AGENT BY
NOUS RESEARCH

The Operator's Guide to Hermes Agent

Build an AI assistant that can act, remember, and improve

Tony Simons

The Operator's Guide to Hermes Agent

Run your projects. Remember your context. Sleep while it works.

By Tony -- operator, builder, and someone who's spent way too long prompt-engineering

Introduction: Stop Using AI Like a Search Box

You've been using it wrong.

Not as an insult. As a diagnosis. Most people who talk about "using AI" are really talking about pasting prompts into a chat box and copying the output. They think they're doing something advanced. They're not. They're using a very expensive search engine with occasional hallucinations and a tendency to forget everything after 10 minutes. They leave with a draft that sounds plausible, paste it into their work, and hope it doesn't embarrass them.

I've been there. You probably have too.

For months I prompt-engineered like it mattered. Tried every jailbreak. Watched every productivity YouTuber tell me the secret was in the prompt structure. Rewrote my queries six times to get a worse answer than I would've gotten from Google in 30 seconds. And I kept doing it because the framing was so seductive. "Just ask better questions." Sure. Okay. That'll be $49.99 for the course.

The ceiling isn't you. It's the mental model.

Here's what I mean. The chatbot frame gets you somewhere. You can get answers, generate drafts, summarize docs, ask it to explain something you forgot from college. Fine. Useful in small doses. But you're still doing all the work. You're the one holding the context. You're the one remembering what you asked last week. You're the one deciding what's good and what's garbage. AI is just a really fast intern who never sleeps, has no memory of your last conversation, and will confidently tell you the wrong answer with perfect grammar.

There's a different way to use this stuff. Once you see it, you don't go back.

It's the difference between asking Google for driving directions versus having an executive assistant who calls you when traffic shifts, remembers your calendar, knows you prefer highway over side streets, and reschedules your whole day without being prompted. One is a lookup. One is a delegation.

This guide is about the second one.

I'm going to show you how I run Hermes Agent every single day. Not as a demo. Not as a screenshot in a sales deck. As an actual operator who has this running in production, has it scheduled with cron jobs, has it remembering things across sessions, has it doing the work between my active sessions while I'm at my day job or sleeping.

I set it up once. Now it runs.

That's the part most people miss when they try agents. They think it means sitting there prompting constantly, watching the AI think in real time, course-correcting every few turns. It doesn't. It means building something that operates on its own, that has memory, that can use tools, that can verify its own work. You set it up once, then it runs. Once you've got that

running, the agent becomes the operator. You're just overseeing it.

This isn't science fiction. It's not even that hard to set up. The rough part is the mental shift, not the tech. Once you get it, everything else follows.

We're going to cover what Hermes actually is, how the mental model works, the specific workflows I run every day, where people screw this up, and how to go deeper once you've got the basics down. We'll go section by section through the actual things I use, not the feature list. By the end you'll have a working setup and a mental model that transfers to anything else you run.

No fluff. No padding. No "in today's fast-paced digital landscape." No "leveraging AI synergies." No bullshit.

Just the operator frame, the actual setup, and the real stuff.

If you've been prompt-engineering for months and feeling the ceiling, this is for you. If you've tried agents before and they drifted, forgot things, or just stopped making sense after a few turns, this is for you. If you're technical enough to wire an API and you're ready to stop using AI like a search box, this is specifically for you.

If you don't know what an API key is, this guide will still mostly make sense. But you're probably not the target reader. Go build some stuff first, then come back.

Let's go.

Chapter 1: What Hermes Agent Actually Is

Here's the 90-second version so you can explain it to someone at a whiteboard.

Hermes Agent is an open-source AI agent framework by Nous Research. It runs in your terminal, in Telegram, Discord, Slack, and 15 other messaging platforms. It connects to almost any LLM provider you want -- OpenRouter, Anthropic, OpenAI, DeepSeek, MiniMax, and 15+ others. It uses tool calling to actually do things: run shell commands, search the web, read and write files, schedule tasks, spawn sub-agents, and remember things across sessions.

It's in the same category as Claude Code (Anthropic) and Codex (OpenAI). The difference is that Hermes is built to be your daily operator, not just a coding assistant. It can run 24/7. It can schedule work. It can live in your messaging apps. It can remember who you are and what you were working on last week.

Here's what that actually looks like in practice.

When you run Hermes, you're not using a web interface. You're running a process. That process connects to an LLM provider, gets a model, and then acts through tools. The tools are what make it an agent instead of a chatbot. Tools are how it interacts with the world: the file system, the terminal, the web, your calendar, external APIs, other agents.

You can talk to it in a terminal:

```
hermes chat -q "What's in my projects directory?"
```

Or you can message it on Telegram like you're texting a colleague:

```
/me What's the status of the build?
```

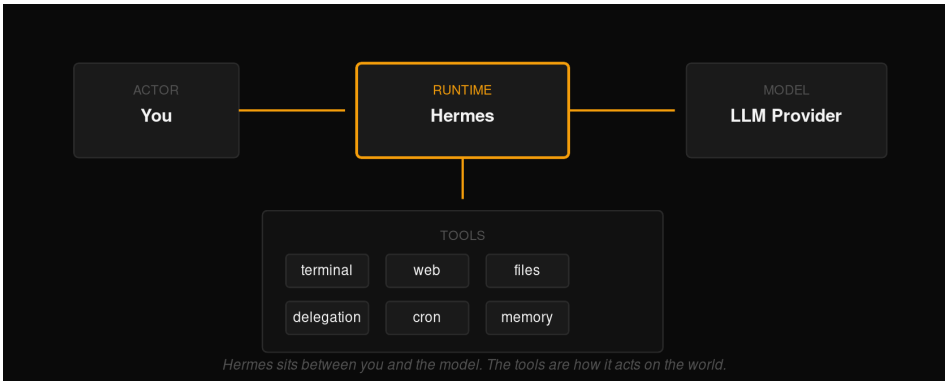
Same agent. Different surfaces.

The architecture, explained simply:

```
You -> Hermes (terminal or messaging app) -> LLM provider (your model)
```

```
                                                  |
                    Tools: terminal, web, files, delegation, cron,
memory
```

That's it. Hermes is the runtime. The model does the reasoning. The tools do the acting. You operate it.



Hermes sits between you and the model. The tools are how it acts on the world.

Not a chatbot. Not a prompt library. Infrastructure.

The best way to understand what Hermes is: it's the software layer between your LLM and the real world. Without it, your LLM can only output text. With Hermes, your LLM can take actions, remember things, schedule work, and coordinate other agents.

Think of it like this. If the LLM is the brain, Hermes is the nervous system and hands. It connects the brain to the environment.

Here's what this means for you in concrete terms:

Without Hermes, you paste a prompt, you get text back. You decide what to do with it. You do it yourself.

With Hermes, you give it a task. It reads your files, runs commands, schedules a follow-up, messages you when it's done, and remembers the outcome for next time. You're

supervising, not executing.

That's the shift. From "I ask AI things" to "I delegate to AI."

The install is one line. We'll get to that in the next chapter.

Chapter 2: The First 30 Minutes

This is where most guides lose people. They hand-wave through setup and then you're stuck at 2 AM wondering why the config isn't loading. We're not doing that here.

You will have Hermes installed, configured, and running a real task in 30 minutes. I'm going to tell you exactly what to type and exactly what can go wrong.

Install

One line. Run this in your terminal:

```
curl -fsSL https://raw.githubusercontent.com/NousResearch/hermes-agent/
    main/scripts/install.sh | bash
```

That's it. The install script handles Python dependency setup, path configuration, and the initial directory structure. It drops everything in `~/.hermes/`.

If you already have Hermes installed, run `hermes update` to pull the latest version. It auto-stashes any local config changes, pulls, and re-applies. If you have uncommitted changes in your config it handles the merge cleanly.

Setup wizard

After install, run:

```
hermes setup
```

This launches an interactive wizard with five steps:

1. Model/provider -- pick your LLM provider (OpenRouter, Anthropic, OpenAI, MiniMax, etc.) and enter your API key

2. Terminal backend -- pick local (default), Docker, SSH, or Modal

3. Gateway -- optionally configure messaging platforms (Telegram, Discord, etc.)

4. Tools -- choose which toolsets to enable (terminal, web, file, delegation, etc.)

5. Agent settings -- max turns, compression threshold, checkpoint settings

The wizard writes to `~/.hermes/config.yaml` and `~/.hermes/.env`. API keys go in `.env`, never in `config.yaml`.

The model you pick here is your default. You can override per-session with `--model` or change it interactively with `/model` during a session.

Health check

After setup, run:

```
hermes doctor
```

This checks:

- Python dependencies are installed
- Config file is valid YAML
- API key is present and credentials resolve
- Required directories exist
- Tool prerequisites (e.g., git for version control, curl for web calls)

If anything is broken, `hermes doctor --fix` attempts to auto-fix it. Run this before you start any real work.

Important: toolset changes require a new session. If you enable a new toolset with `hermes tools enable`, it won't be available until you start a fresh session. The toolset list is loaded at session start, not at runtime. If you add a toolset and wonder why nothing happened, `/new` first.

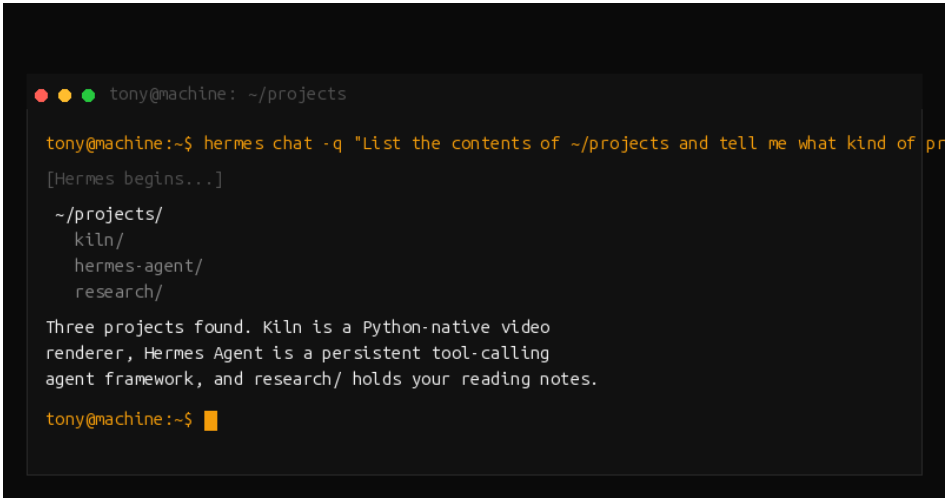
Important: config edits require restart. If you edit `config.yaml` directly (instead of using `hermes config set`), you need to exit and relaunch. Config is read once at startup and cached. A running session won't pick up file changes.

Your first real task

Let's run something that actually does something:

```
hermes chat -q "List the contents of ~/projects and tell me what kind of
    projects you see there"
```

You should get back a list of directories and a brief summary of what each project appears to be. Hermes will use the file tool to read your directory. This is your first real agent action -- it read the filesystem, reasoned about what it saw, and responded.



Step 1: Your first agent action. Hermes reads your filesystem and reasons about what it sees.

Now let's try something more useful. Start an interactive session:

```
hermes chat
```

At the prompt, type:

```
/skill
```

You'll see Hermes load its built-in skill system. Skills are how Hermes learns persistent procedures. The `/skill` command shows you what skills are available and lets you load them.

Right now you have the basic setup working. Here's the honest caveat: the first 30 minutes will probably include at least one API key error or provider timeout. That's normal. The providers all have rate limits and sometimes credentials take a few minutes to activate after you create them. If you get an auth error, wait 60 seconds and try again. If it persists, check your API key in `~/.hermes/.env`.

Also: the model you pick matters a lot for the experience. Cheap models work for simple tasks but drift faster on complex multi-step work. For your first session, use whatever you have budget for. You can always switch later.

What you just did

You installed a persistent AI operator in one command. You configured it with your preferred model. You ran a task that read your filesystem and responded intelligently. You saw the skill system load.

That's the foundation. Everything else in this guide builds on this.

Chapter 3: The Mental Model

This is the chapter that makes you feel smart. Once you understand these six concepts, every new Hermes feature becomes predictable. You stop learning features and start understanding a system.

The six concepts are: Tools, Skills, Memory, Sessions, Cron, and Gateway. Each one is a distinct capability. They don't overlap. Once you know what each one does, you'll never mix them up again.

Toolsets: What Hermes Can Do

A toolset is a group of capabilities that Hermes can use when it's thinking and acting. Think of toolsets as plugins -- each one unlocks a class of actions.

Toolsets are enabled via `hermes tools enable NAME`. The changes take effect on `/reset` (new session), not mid-conversation.

Here's what each toolset does:

`web`	Web search and content extraction
`terminal`	Run shell commands, manage processes
`file`	Read, write, search, and patch files
`browser`	Browser automation (local Chromium or Browserbase)
`code_execution`	Sandboxed Python execution
`vision`	Analyze images
`image_gen`	Generate images via AI
`tts`	Text-to-speech
`delegation`	Spawn sub-agents to handle parallel work

`cronjob`	Schedule tasks to run on a timeline
`memory`	Persistent cross-session facts
`session_search`	Search past conversation history
`skills`	Browse and load persistent procedures
`messaging`	Send messages across platforms

You don't need all of them. For most operators, `terminal`, `web`, `file`, `delegation`, `cronjob`, and `memory` cover 80% of what you'd ever do.

When to use: You need Hermes to actually do something. If you want it to search the web, you need `web`. If you want it to run tests, you need `terminal`. The toolset is the capability layer.

Skills: Saved Procedures

Skills are markdown files that teach Hermes your patterns. When you find yourself re-explaining the same context to Hermes in every session, that's when you write a skill.

A skill is a markdown file with YAML frontmatter. The frontmatter defines the skill name and triggers. The body is the procedure.

Example: you work on a Python project and you always want Hermes to know your testing setup. You write:

```
---
name: python-testing
description: Run tests for our Python project
---

# Python Testing Skill

Our project uses pytest with pytest-xdist for parallel execution.

## Running tests

To run all tests:
```

cd ~/projects/myapp && pytest -n auto

```
To run a specific test file:
```

pytest tests/test_api.py -v

```
## Common patterns

- Unit tests are in `tests/unit/`
- Integration tests are in `tests/integration/`
- Run the full suite before submitting a PR
```

Now when you run `/skill python-testing`, Hermes loads that context automatically. Skills live in `~/.hermes/skills/` (or `~/.hermes/profiles/<name>/skills/` if you're using profiles). You can publish skills to the Hermes skills registry with `hermes skills publish PATH`. You can also install skills other people have written with `hermes skills install <skill-name>`.

When to use: You keep re-explaining the same context. You want Hermes to know your project's coding standards, your deployment process, your preferred tools. Skills are your institutional knowledge, persisted.

Memory: Cross-Session Facts

Memory is Hermes's ability to remember things about you without being re-told. You can say "My main project is at ~/code/myapp and I use Poetry for dependency management" and Hermes will remember that in every future session.

The memory system uses a persistent store. By default it uses Hermes's built-in SQLite-backed memory. You can also configure Honcho, Mem0, Super Memory, or other providers.

Configure it with `hermes memory setup`. Check status with `hermes memory status`.

When to use: You want Hermes to know your name, your preferences, your projects, your setup. Memory is for facts that should persist across sessions, not for session-specific context.

Don't use memory for "I'm working on X today" -- that's a session-level concern. Use memory for "I always run X from directory Y."

Sessions: The Conversation Thread

Sessions are Hermes's conversation history. Each time you start a chat, you're in a session. Sessions are resumable, nameable, and searchable.

When you exit a session and come back, you can resume exactly where you left off:

```
hermes --resume <session-id>
# or
hermes --continue
```

The `--continue` flag resumes the most recent session by name or ID.

Sessions are stored in `~/.hermes/sessions/`. You can search past sessions with the `session_search` toolset:

```
hermes chat -q "What was I working on with the database migration last
    week?"
```

Sessions are the mechanism for continuity. Without sessions, every conversation starts from scratch. With sessions, you can pick up work from yesterday without re-explaining the context.

When to use: You want to continue work from a previous session. You want to search your conversation history for something you discussed. Sessions are your audit trail.

Cron: Scheduled Execution

Cron is Hermes's ability to run tasks on a schedule. You can set up a daily briefing that fires at 6:20 AM and sends you a summary in Telegram before you're out of bed.

Cron jobs are created with:

```
hermes cron create '0 9 * * *' "Run my daily content radar and send
    results to Telegram"
```

The schedule format is standard cron: minute, hour, day, month, weekday. You can also use shorthand: `'30m'`, `'every 2h'`, `'0 9 *'`.

List your cron jobs:

```
hermes cron list
```

Pause, resume, edit, or remove jobs with the matching subcommands.

Critical caveat: the built-in `deliver: telegram:chat_id` mechanism is unreliable for script-generated output. If your cron job runs a script that produces structured JSON, the built-in Telegram delivery can silently fail after the script succeeds. The audit log shows notification_sent: true but you never get the message. Always use direct curl to the Telegram Bot API from your scripts instead. More on this in Chapter 6.

Also: if you're setting up a cron job that you don't want running until you approve it, create the job then immediately pause it. Don't leave it enabled just because the first scheduled run is in the future.

When to use: You want something to happen automatically, with or without you present. Daily briefings, weekly reports, monitoring jobs, automated content pipelines. This is the "sleep while it works" part of the subtitle.

Gateway: Hermes in Your Messaging Apps

The Gateway is what lets Hermes live in Telegram, Discord, Slack, WhatsApp, Signal, and 15 other platforms. You're not limited to the terminal. You can message Hermes like a colleague.

Set up the Gateway:

```
hermes gateway setup
```

This walks you through platform configuration. For Telegram, you need a Bot token from @BotFather. For Discord, you need a bot token with Message Content Intent enabled. Each platform has its own setup steps.

Once configured:

```
hermes gateway start
```

Hermes will run as a background service. You message it on whatever platform you configured and it responds like a teammate.

Gateway logs are at `~/.hermes/logs/gateway.log`. If your bot goes silent, check the log first.

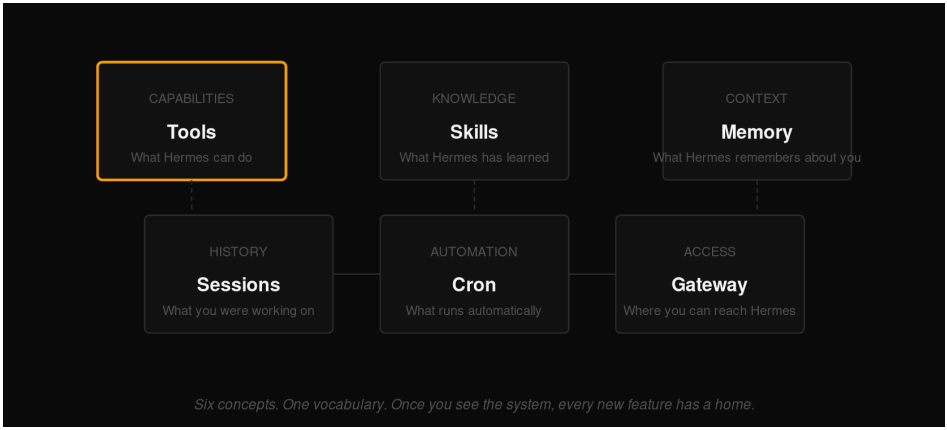
WSL2 caveat: If you're running Hermes in WSL2 and the gateway dies when you close your terminal, you need `systemd=true` in `/etc/wsl.conf`. Without it, WSL2 falls back to nohup mode and the gateway process dies when the session closes. If you're on SSH, run `sudo loginctl enable-linger $USER` so the gateway survives the SSH session ending.

When to use: You want to message Hermes from your phone while you're away from your desk. You want push notifications from automated jobs. You want to interact with your agent like it's a teammate, not a CLI tool.

How They All Fit Together

```
Tools: What Hermes can do (capabilities)
Skills: What Hermes has learned (knowledge)
Memory: What Hermes remembers about you (facts)
Sessions: What you were working on (history)
Cron: What runs automatically (scheduling)
Gateway: Where you can reach Hermes (access)
```

These six concepts are the vocabulary of operator-level AI usage. When you learn a new Hermes feature, you can immediately place it: it's a new tool, a new skill, a memory integration, a session feature, a cron option, or a gateway platform. The system becomes predictable.



Six concepts. One vocabulary. Once you see the system, every new feature has a home.

Chapter 4: Real Workflows I Run Every Day

This is the heart of the ebook. Four specific workflows I run every day. Exact prompts. Real outputs. Real timing. This is what it looks like when it actually works, not a demo.

Workflow A: Research Pipeline

The research pipeline is how I go from "I need to understand X" to a structured output file I can reference later, without sitting there manually searching and copy-pasting.

What it does: Web search -> extract key information -> write findings to a markdown file in my research directory.

The prompt I use:

```
Research the current state of GGUF quantization for local LLM inference.
Find: (1) what tools support GGUF (2) performance benchmarks vs full
    precision
(3) any recent updates in 2025-2026. Write a structured summary to
~/research/gguf-state-$(date +%Y%m%d).md with sources.
```

What Hermes does:

1. Activates the `web` toolset

2. Searches for GGUF quantization benchmarks and recent updates

3. Reads multiple sources

4. Synthesizes findings into a structured markdown file

5. Lists sources at the bottom

Output location: `~/research/gguf-state-YYYYMMDD.md`

Timing: Depends on how deep you want it to go. A surface-level scan is 2-3 minutes. A deep dive with multiple sources is 10-15 minutes. I usually start it and come back.

What makes this real: Hermes reads the actual pages, not just search snippets. It synthesizes across sources. You get a file you can reference, quote, and share. It's not just a list of links -- it's a doc with context.

```
[Research Pipeline]
web search -> page extraction -> synthesis -> file write
                                                |
                                    [Your research dir]
                                    gguf-state-20260504.md
```

Four steps. One command. A file you'll actually read later.

Workflow B: Repo Debugging

This is the one that saves me the most time on a per-incident basis. Something breaks in a project, I hand it to Hermes, and it either fixes it or tells me exactly what needs to change.

What it does: Reads the repo, runs diagnostics, identifies the bug, proposes or applies a fix.

The prompt I use:

```
There's a test failure in ~/projects/kiln. Run pytest and tell me what's
failing and why. If it's a simple fix, apply it and rerun to confirm.
Report back with what the problem was and what you changed.
```

What Hermes does:

1. Changes to the project directory
2. Runs pytest to capture the failure
3. Reads the failing test file
4. Reads the relevant source files
5. Identifies the root cause
6. Applies the fix
7. Re-runs the tests to confirm

One-shot vs interactive: For obvious bugs (wrong import, typo, missing argument), a one-shot works fine. For complex failures that require tracing through multiple files, I'll start an interactive session and walk Hermes through the stack trace step by step.

What makes this real: I hand it a broken repo and get a working one back. Or I get a clear explanation of why it can't be fixed automatically and what I need to do. Either way, I don't spend 45 minutes reading the same stack trace trying to find the off-by-one error.

```
[Repo Debugging]
```

```
cd ~/projects/kiln -> pytest -> read failing files -> identify root cause
-> apply fix -> verify
```

Workflow C: Daily Briefing via Cron

This is the "sleep while it works" workflow. Every morning at 6:20 AM, Hermes wakes up, runs a content radar, and sends me a briefing on Telegram before I'm out of bed.

The cron command:

```
hermes cron create '20 6 * * 1-5' "Daily content radar: scan for
    interesting
posts on AI agents, local models, and Hermes workflows. Format findings
    as a
brief briefing. Send to Telegram via curl."
```

The actual script (called by the cron job):

```
#!/bin/bash
# Daily content radar - runs at 6:20 AM CT Mon-Fri
# Sends briefing to Telegram

TELEGRAM_BOT_TOKEN=$(grep TELEGRAM_BOT_TOKEN ~/.hermes/.env | cut -d= -f2)
TELEGRAM_CHAT_ID=$(grep TELEGRAM_CHAT_ID ~/.hermes/.env | cut -d= -f2)

BRIEFING=$(hermes chat -q "Run a content radar: find 3-5 interesting
    posts
about AI agents, local model setups, or Hermes workflows from the past 24
    hours.
Format as a numbered briefing with links.")

curl -s -X POST "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/
    sendMessage" \
  -d "chat_id=${TELEGRAM_CHAT_ID}" \
  -d "text=${BRIEFING}" \
  -d "parse_mode=Markdown"
```

Timing: Fires at 6:20 AM CT every weekday. I'm usually awake by 6:45 and checking Telegram before I'm out of bed. The briefing is there waiting.

Why the direct curl pattern: The built-in `deliver: telegram:chat_id` mechanism in cron jobs silently fails for script-generated output. The script above succeeds every time because it's a direct call to the Telegram Bot API. This is documented in Chapter 6. Use the curl pattern.

What makes this real: I've set this up once. It runs every day without me. The briefing is formatted and in my Telegram

before I open my laptop. This is the difference between "I should check for interesting posts" and "I wake up to a briefing."

```
[Cron: 6:20 AM CT Mon-Fri]
Hermes wakes up -> content radar query -> Telegram curl -> your phone
```

Workflow D: Multi-Agent Kanban

This is the most powerful workflow and the one that took me longest to understand. Instead of running one agent on a complex task, I split the task across multiple specialist agents, run them in parallel, and synthesize the results.

The Kanban system in Hermes is how I orchestrate this.

How it works:

1. I create a parent task that describes the overall goal

2. I create child tasks for each parallel workstream and assign them to specialist profiles

3. I link the tasks so the parent waits for all children

4. I dispatch the board and let agents run

5. When children are done, the parent task picks up their outputs and synthesizes

Example: I want to write a comprehensive review of a technology. I create:

- Child task A: Research market landscape and competitors (assigned to researcher profile)
- Child task B: Technical deep-dive on architecture (assigned to engineer profile)
- Child task C: User sentiment and community discussion (assigned to social profile)

I link them to the parent. Dispatch. Each specialist runs in parallel. Their outputs feed into the parent synthesis task.

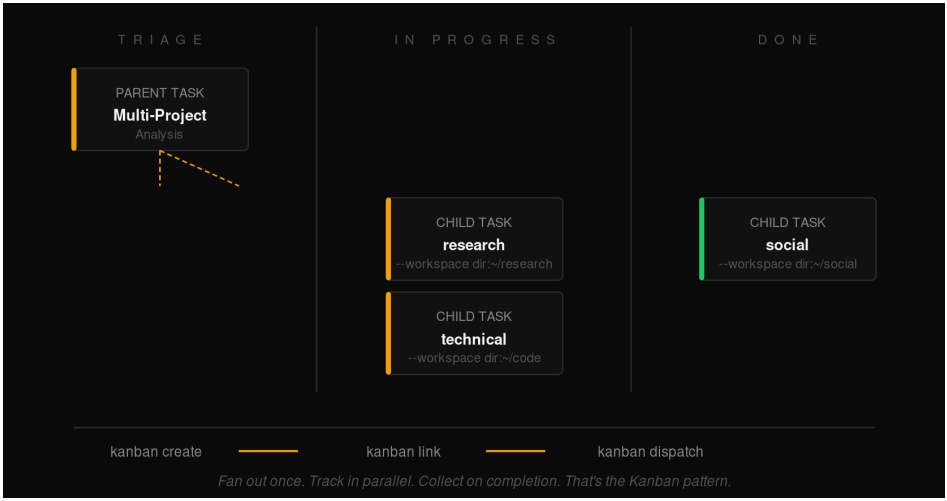
The exact commands:

```
# Create the parent task
hermes kanban create "Write comprehensive X review" --assignee writer

# Create child tasks for parallel workstreams
hermes kanban create "Research X market landscape" --assignee researcher -
    -parent <parent-id>
hermes kanban create "Technical analysis of X architecture" --assignee
    engineer --parent <parent-id>

# Link children to parent (ensures parent only runs after children
    complete)
hermes kanban link <parent-id> <child-a-id>
hermes kanban link <parent-id> <child-b-id>

# Dispatch the board
hermes kanban dispatch
```

What makes this real: Three agents working simultaneously. Each one does deep work in its specialty. The synthesis task has real inputs from real research. I didn't do the research -- I orchestrated the agents who did. The parallel work completes in a fraction of the time it would take sequentially.

Caveat: always use `--workspace dir:/absolute/path` when creating tasks that write output. The default scratch workspace gets garbage collected when tasks are archived. If your agent writes output to scratch and the task is archived, the files disappear. Use `--workspace dir:/home/tony/projects/myproject` for any task that produces files you want to keep.

```
parent task
    |
    +-- child A (research)
    |    --workspace dir:/home/tony/research/
    +-- child B (technical)
    |    --workspace dir:/home/tony/code/analysis/
    +-- child C (social)
    |    --workspace dir:/home/tony/social/

    All run in parallel. Results feed parent on completion.
```



Fan out once. Track in parallel. Collect on completion. That's the Kanban pattern.

Chapter 5: The Operator Loop

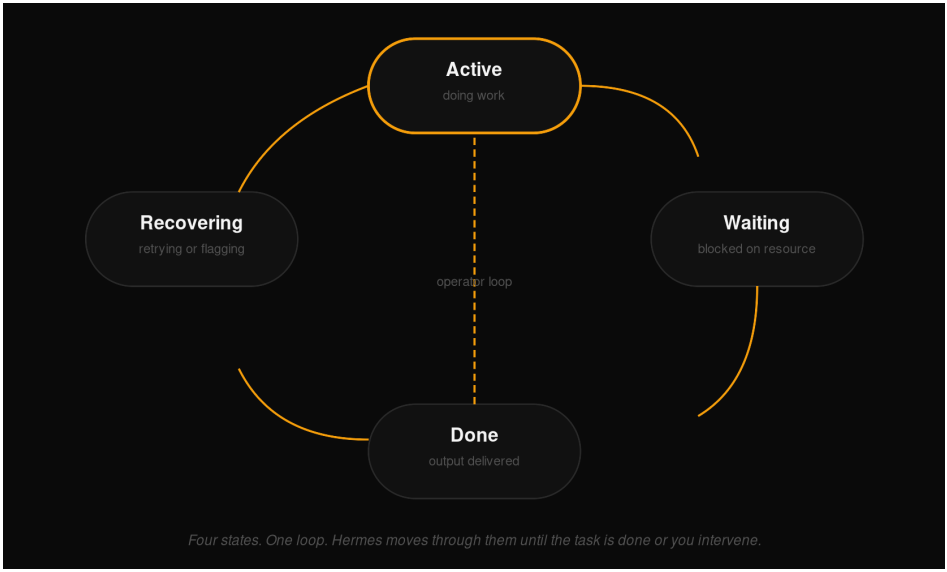
Now you know the concepts and you've seen the workflows. This chapter gives you the operational mental model for running Hermes continuously, not just in one-off sessions.

The operator doesn't babysit.

Operators set up loops. They verify outputs. They intervene only when something needs a human decision. The rest of the time, the agent runs.

Here's the four-state loop that governs every Hermes operator session:

```
Active -> Waiting -> Recovering -> Done -> (back to Active)
```



Four states. One loop. Hermes moves through them until the task is done or you intervene.

Active: Hermes is doing work. Running commands, searching, reading files, generating output. You can watch in real time or you can walk away. Either way, it's doing the thing.

Waiting: Hermes hit a blocking call: an API request to an LLM provider, a tool that's waiting for an external resource, a rate limit that's throttling progress. It resumes automatically when the resource is available. You don't need to do anything.

Recovering: Something failed. A command errored, a tool call returned an unexpected result, the context hit a limit. Hermes either attempts recovery autonomously (retrying, summarizing context, restarting a step) or it flags you and waits for instruction. This is the state where you decide whether to intervene.

Done: The task is complete. Output is delivered. Session is saved. You're done.

The loop applies to both individual sessions and longer-running operator setups. A one-shot query goes Active -> Waiting -> Done in 30 seconds. A complex multi-step project goes through Active/Waiting/Recovering cycles over hours.

Setting Up a Long-Running Operator Session

For tasks that take more than a few minutes, use an interactive session instead of a one-shot:

```
hermes chat
# then type your prompt at the interactive prompt
```

The interactive session keeps the context alive across multiple turns. You can hand off a complex task and come back to check progress.

For truly long-running tasks (hours), use background mode:

```
/background Run the full test suite for ~/projects/kiln and report
    results to ~/logs/test-results.md when complete
```

The background job runs independently. You get notified when it completes.

Context Management for Long Sessions

This is the part most operators skip and then regret.

Every LLM has a context window limit. As your conversation grows, Hermes feeds more history into the context. When you get close to the limit, the model starts losing track of older context -- it drifts, repeats itself, stops noticing things that were obvious three hours ago. It doesn't warn you explicitly. It just slowly gets worse.

Here's the strategy:

For sessions under 2 hours of active conversation: You can mostly ignore this. Run your session, get your output, done. The built-in compression triggers automatically when you get close to the limit. But don't count on it -- the auto-trigger is

conservative and you may lose important context right before a critical decision.

For sessions over 2 hours: Be deliberate. Use `/compress` manually before the auto-trigger kicks in, when you reach a natural stopping point. This summarizes the conversation history and replaces it with a compressed version. The original context is gone -- replaced by a summary. If you're working on something where the exact details of earlier steps matter, compress at a milestone, not at a crisis.

For multi-day projects: Don't try to keep one session alive for a week. Break work into chunks with explicit file-based continuity. At the end of each session, write a checkpoint file:

```
/background Write current progress and next steps to ~/checkpoint.md
```

Next session, read the checkpoint first, then continue. This gives you a human-readable audit trail that survives any session crash.

When to start fresh instead of compressing: If the session has gone so long that the summary would be 30+ lines, you're probably better off starting a new session and loading the relevant context via a skill or a paste. A compressed summary of 200 exchanges is still a summary -- you've lost resolution. Better to save the key facts to a file and start clean.

The operator instinct is to keep sessions alive as long as possible because continuity feels productive. Sometimes it is. More often, a fresh session with a good context file beats a compressed session with degraded context.

How to Verify Outputs Without Babysitting

The most important operator skill: knowing how to verify output without hovering.

File output: If your task writes to a file, check the file:

```
hermes chat -q "Check ~/logs/test-results.md and tell me if all tests
    passed"
```

Status checks: For ongoing work, periodic status queries:

```
hermes chat -q "What's the current status of the content pipeline? Any
    failures?"
```

Session resume: If you left an interactive session mid-task:

```
hermes --continue
```

Hermes resumes the session exactly where it left off, with full context of what was discussed and what was done.

When to Intervene

This is where operator discipline is tested. Here's the honest decision tree:

Intervene when:

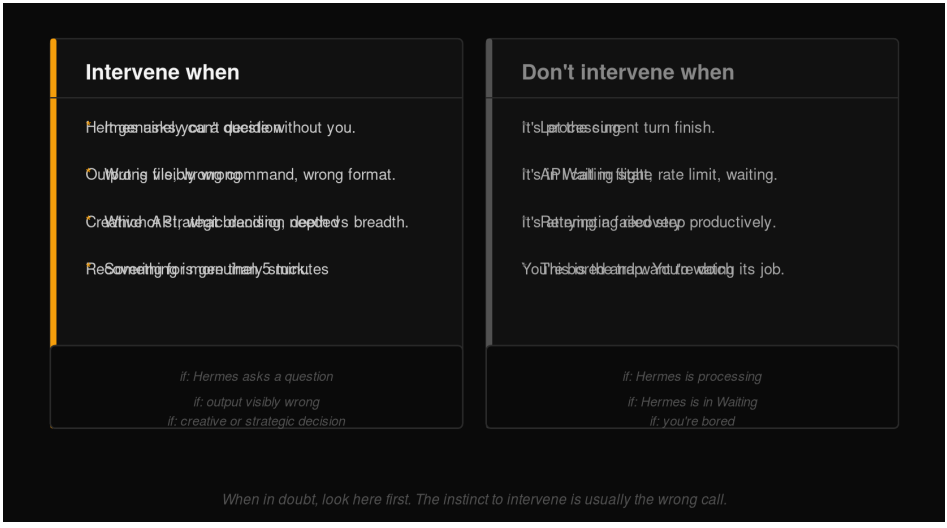
- Hermes asks you a question. It genuinely can't decide without you. When it asks "should I continue with option A or B?" that's real -- it's not posturing.
- Output is visibly wrong and it can't self-correct. Wrong file written, wrong command run, wrong format produced. You'll know it when you see it.
- You've hit a creative or strategic decision. Which API to use, what the branding should say, whether to prioritize depth over breadth. These are human decisions.
- The task has been in Recovering state for more than 5 minutes without resolving. Something is stuck.

Don't intervene when:

- It's processing. Let the current turn finish.
- It's in Waiting state. An API call is in flight, a rate limit is throttling it, a tool is waiting for an external resource. It will resume. You waiting at the screen doesn't help.

- It's attempting recovery and making progress. Hermes is retrying a failed step, trying an alternative approach. Give it one good attempt before stepping in.
- You're bored and want to see what's happening. This is the trap. You're not supervising a toddler. If you're watching every turn, you're doing the work, not the agent.

The instinct to hover is hard to break. Fight it. Hermes is doing the work -- your job is to operate it, not perform alongside it.



When in doubt, look here first. The instinct to intervene is usually the wrong call.

Here's the test: if you're reading the output as it's being generated rather than waiting for a notification that it's done, you're probably babysitting. Set it up, verify the output, let it run.

Chapter 6: Where People Screw This Up

This is the chapter that gets bookmarked and sent to your cofounder. The honest one. The stuff that actually breaks in production, why it breaks, and how to fix it.

I'm not going to dress this up. If you hit these, it's not you -- it's the setup. And they're all fixable.

1. Context Overflow

What it looks like: Hermes starts drifting. The responses get less precise. It starts repeating itself or missing obvious context from earlier in the session. Eventually it just stops making sense.

Why it happens: Every LLM has a context window limit. As your conversation grows, Hermes is feeding more history into the context. When you get close to the limit, the model starts losing track of older context. It doesn't tell you this explicitly -- it just starts drifting.

How to fix it: Use `/compress` to manually trigger context compression. This summarizes the conversation history and frees up context space. You can also start a fresh session with `/new` and load the relevant context via a skill. For long-running projects, consider breaking work into smaller sessions with explicit file-based continuity (Hermes writes progress to a file, next session reads it).

How to prevent it: Don't let sessions run for more than a few hours of active conversation. If you're working on a complex project, use file-based checkpoints: `/background "Write current progress to ~/checkpoint.md"` every hour or so.

2. Toolset Mismatch

What it looks like: You enabled a toolset with `hermes tools enable web`, then asked Hermes to search the web, and it said it couldn't do that.

Why it happens: Toolsets are loaded at session start. Enabling a toolset mid-session doesn't make it available until you start a new session. This trips up everyone at least once.

How to fix it: Run `/new` to start a fresh session. The newly enabled toolsets will be active.

How to prevent it: Get in the habit of running `/new` any time you change toolsets. Yes, even if you think you won't need them yet. It's faster than debugging why a tool isn't responding.

3. Config Drift

What it looks like: You edited `config.yaml` directly, launched Hermes, and it seemed to ignore your changes. Or it crashed on startup.

Why it happens: Hermes reads `config.yaml` once at startup and caches it. Edits made while Hermes is running are not picked up. Edits made to a running session's config are similarly ignored until you restart.

How to fix it: Exit and relaunch Hermes. Or for gateway: `/restart`. The config is re-read from disk on startup.

How to prevent it: Use `hermes config set section.key value` instead of editing the YAML directly. This is the safe path. If you must edit the YAML, restart afterward.

4. Cron Delivery Lies

What it looks like: Your cron job runs successfully. The script completes. You check the logs and it says `notification_sent: true`. But you never got the Telegram message.

Why it happens: The built-in `deliver: telegram:chat_id` mechanism in cron jobs has a known bug: it silently fails when

the script produces structured output. The cron agent thinks it sent the message, but the Telegram API call actually errored out after the script finished.

How to fix it: Don't use the built-in `deliver:` mechanism for Telegram (or any platform) when your script generates its own output. Instead, use direct curl in your script. Read the bot token from `~/.hermes/.env`:

```
TELEGRAM_BOT_TOKEN=$(grep TELEGRAM_BOT_TOKEN ~/.hermes/.env | cut -d= -f2)
curl -s -X POST "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/
    sendMessage" \
  -d "chat_id=${TELEGRAM_CHAT_ID}" \
  -d "text=${BRIEFING}"
```

This pattern never fails silently. If the curl fails, you see it in the script exit code.

How to prevent it: Always use direct API calls from your scripts for delivery. Treat the built-in `deliver:` as unreliable for structured output use cases.

5. Scratch Workspace Data Loss

What it looks like: You ran a Kanban task, the agent wrote output to the scratch workspace, the task completed successfully, and now you can't find the files.

Why it happens: The default Kanban workspace is a scratch directory at `~/.hermes/kanban/workspaces/<task_id>/`. When a task is archived, this directory is garbage collected. Files you thought were saved are gone.

How to fix it: You can't recover them -- they're gone. Recreate the output or rerun the task with the correct workspace.

How to prevent it: When creating tasks that produce output you want to keep, always use `--workspace dir:/absolute/path`:

```
hermes kanban create "Analyze project architecture" \
  --assignee engineer \
  --workspace dir:/home/tony/projects/myapp
```

This writes directly into your project directory. The output persists after the task is archived.

6. Delegation Model Mismatch

What it looks like: You run a task that uses `delegate_task` to spawn sub-agents. All the sub-agents fail immediately with an error like "model not supported" or "provider cannot serve this model."

Why it happens: The `delegation.model` in `config.yaml` specifies a model that your current provider can't serve. For example, if your provider is OpenAI and `delegation.model` is set to a DeepSeek model name, it fails. The `CLI_CONFIG` dict is also cached at import time, so file edits may not be picked up without restarting.

How to fix it: Check `~/.hermes/config.yaml` and make sure `delegation.model` matches what your provider supports. If you've changed providers or API keys, update this field. Then restart Hermes to clear the cached config.

How to prevent it: After changing your main model or provider, check the delegation config. Keep them in sync.

7. WSL2 Gateway Death

What it looks like: You set up the Gateway to run on Telegram. It works fine while your terminal is open. The moment you close the terminal or shut down your PC, Hermes stops responding to messages.

Why it happens: WSL2 doesn't run systemd by default. Without systemd, Hermes Gateway falls back to nohup mode, which dies when the WSL session closes.

How to fix it: Add `systemd=true` to `/etc/wsl.conf`:

```
[boot]
systemd=true
```

Then restart WSL (`wsl --shutdown` from PowerShell, then reopen). The gateway will now run as a proper systemd service and survive session closure.

How to prevent it: Set up `/etc/wsl.conf` with systemd before configuring the gateway. This is a one-time fix.

8. Profile Name Ghosting

What it looks like: You assign a Kanban task to a profile with `hermes kanban assign`. The command returns success. The task doesn't get picked up. The dispatcher keeps showing `crashed=N` with no reclaims.

Why it happens: `hermes kanban assign` can silently fail if the profile name doesn't exactly match an existing profile. The command returns exit code 0 but the assignment never actually updates. Hermes spawns workers for the old (nonexistent) profile name, they crash instantly, and the dispatcher retries indefinitely.

How to fix it: Always verify after assigning:

```
hermes kanban show <task-id>
```

Check that the `assignee` field shows the correct profile name. If it didn't take, create a fresh task with the correct profile name and archive the broken one.

How to prevent it: Before assigning, verify the profile exists:

```
hermes profile list
```

Use the exact name from this list.

Chapter 7: The Next Layer

You have the foundation. You know the mental model. You've run the workflows. Now what?

This chapter is about the layers that compound. The stuff you hit once you've got the basics running and you start asking "what else can I do with this?"

Multi-Agent Orchestration

The Kanban system is your orchestration layer. Once you're comfortable creating tasks and linking them, you can scale to complex multi-agent pipelines. But the real leverage is when you stop thinking about Kanban as a todo list and start thinking about it as a workflow engine.

Here's the pattern that actually works: decompose the goal into specialist roles, run them in parallel, synthesize the outputs. Not a linear chain where each step waits for the previous one. Parallel fans-out where every specialist works simultaneously, then the synthesis task collects results and produces the final deliverable.

The critical constraint most people miss: each Kanban task needs its own workspace. If you create child tasks that all write to the same directory or the default scratch workspace, their outputs collide. Use `--workspace dir:/absolute/path` for every task that produces output. The parent synthesis task reads from those directories. This is not a nice-to-have -- without it, you'll lose work when the scratch workspace gets garbage collected.

The other constraint: Kanban tasks need to be small enough to complete in one agent run. If a task requires 200 turns and 5 agent interactions, it's too big. Split it. A task that times out mid-run crashes and gets re-queued by the dispatcher, which can create duplicate work or corrupted state.

Hermes Vault

When you're running multiple agents across multiple projects, credential management stops being theoretical. Which agent

has access to which API keys? How do you rotate a key without breaking running setups? How do you audit what each agent accessed when?

Vault is the answer. It's a credential broker that lets you provision credentials to agents without hardcoding them in config files or environment variables. Agents request credentials at runtime; Vault provides them with audit logging. You rotate a key once in Vault and every agent that depends on it picks up the new credential on its next request.

If you're running production workloads with Hermes, Vault is the piece that makes it operationally sound. If you're running experimental or hobbyist setups, you can ignore it for now and hardcode keys in your `.env` file. You'll know when you need the upgrade.

Custom Skills Development

The skills system is how you teach Hermes your patterns. Most people stop at "save a testing procedure" and think they're done. They're not. The real leverage is encoding decision-making patterns, not just procedures.

Think about the difference. A procedure skill says "here's how to run tests." A decision-making skill says "here's how to decide what to test first, what a passing result looks like for our codebase, and what to do when a flaky test appears." The second one is worth 10 of the first.

Example from my setup: I have a code review skill that knows my project's architecture, understands the team's coding standards, and runs a specific review checklist every time a PR is opened. It catches the mechanical stuff -- missing tests, wrong import paths, insecure direct SQL concatenation. I read the output and make the actual judgment calls. The skill did the first pass. I did the human part.

Skills compound. Each one you write makes the next session faster. After a few months of building skills, Hermes knows your projects almost as well as you do. Not as well -- you still know more -- but enough that you stop re-explaining context every session.

MCP Server Integrations

The Model Context Protocol (MCP) is how you connect Hermes to external systems. Not just for connecting Hermes to other tools -- though that's the obvious use case. The interesting part is connecting to data sources that shape how Hermes operates.

Run Hermes as an MCP server:

```
hermes mcp serve
```

Connect external tools or other MCP servers to it:

```
hermes mcp add github --command "npx @modelcontextprotocol/server-github"
hermes mcp add filesystem --command "npx @modelcontextprotocol/server-
    filesystem"
```

The `hermes mcp add` command registers an MCP server by name with either a `--url` or a `--command`. The `--command` form runs a local MCP server process. The `--url` form connects to a remote MCP endpoint.

The combination of MCP servers plus Hermes's native toolsets gives you an agent that can interact with your database, your GitHub repos, your Notion workspace, your Slack channels -- anything with an API. You can build skills that wire these together into domain-specific workflows that would be hard to replicate with generic prompting.

One gotcha: MCP server connections are configured per-profile. If you're running multiple profiles for different contexts, you'll need to configure MCP servers separately in each. This is by design -- a researcher profile shouldn't automatically have access to your production database MCP server.

Profile Isolation

Profiles let you run multiple independent Hermes instances with isolated configs, sessions, skills, and memory. This is the feature that turns Hermes from a single personal assistant into a team.

Use profiles for:

- Work vs personal separation. Different `.env` files, different session histories, different memory stores.
- Task-type isolation. A researcher profile with web tools and memory configured for research context. A writer profile with a voice and style baked in.
- Provider separation. If you want to use a cheap model for routine tasks and a premium model for complex reasoning, use separate profiles rather than switching manually.
- Experimental isolation. Test new skills or config changes in an isolated profile before rolling them into your main setup.

```
hermes profile create researcher
hermes profile create writer
hermes profile create engineer
```

Each profile is a fully independent Hermes instance at `~/.hermes/profiles/<name>/`. They don't share memory unless you explicitly configure cross-profile memory sharing. They don't share sessions. They don't share skills unless you install the same skill in each profile.

The isolation is real and enforced. This is a feature, not a limitation. When you're in the writer profile, you can't accidentally interfere with the researcher session. When you're testing a new skill in the experimental profile, you can't break your main setup.

Where to Go From Here

The path isn't linear. Some people start with the Gateway because they want Hermes in Telegram. Some start with Cron because they want automation. Some start with Skills because they want Hermes to remember their context.

The order doesn't matter. What matters is that you start. Pick one workflow from Chapter 4 and run it today. Get it working. Then add the next one.

The mental model from Chapter 3 is your anchor. Once you have that solid, everything else slots into place predictably.

You're not learning features. You're understanding a system. And once you understand the system, the features don't surprise you anymore -- they just confirm what you already knew.

Appendix: Quickstart Reference

Everything you need to get running in 30 minutes and keep running.

Install and Setup

```
# Install
curl -fsSL https://raw.githubusercontent.com/NousResearch/hermes-agent/
    main/scripts/install.sh | bash

# Configure
hermes setup

# Health check
hermes doctor

# Update (if already installed)
hermes update
```

First-Run Checklist

- [ ] API key in `~/.hermes/.env` (not in `config.yaml`)
- [ ] Provider configured via `hermes model` or `hermes setup`

- [ ] Basic toolsets enabled: `terminal`, `web`, `file`
- [ ] `/new` run after any toolset change
- [ ] First test query run successfully

Toolset Rules

Rule 1: Toolset changes require `/new`. Always.

Rule 2: Config changes require restart. Always.

Rule 3: Enable only the toolsets you need. Fewer toolsets means a cleaner tool schema and less confusion about what Hermes can do.

Session Commands

```
hermes chat -q "one-shot query"
hermes chat              # interactive session
hermes --continue        # resume most recent session
hermes --resume <id>     # resume by session ID
```

Skill Commands

```
hermes skills list       # show available skills
/skill <name>            # load a skill in session
hermes skills install <name>   # install from registry
hermes skills publish <path>   # publish to registry
```

Cron Commands

```
hermes cron list                   # list all jobs
hermes cron create '20 6 * * 1-5' "Daily briefing"   # create job
hermes cron pause <id>             # pause a job
hermes cron resume <id>            # resume a job
hermes cron remove <id>            # delete a job
```

Reliable Telegram Delivery (Cron Script Pattern)

```
#!/bin/bash
TELEGRAM_BOT_TOKEN=$(grep TELEGRAM_BOT_TOKEN ~/.hermes/.env | cut -d= -f2)
TELEGRAM_CHAT_ID=$(grep TELEGRAM_CHAT_ID ~/.hermes/.env | cut -d= -f2)

RESULT=$(hermes chat -q "Your query here")

curl -s -X POST "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/
    sendMessage" \
  -d "chat_id=${TELEGRAM_CHAT_ID}" \
```

```
    -d "text=${RESULT}" \
    -d "parse_mode=Markdown"
```

Never use `deliver: telegram:chat_id` for script-generated output.

Gateway Commands

```
hermes gateway setup      # configure platforms
hermes gateway start      # start background service
hermes gateway stop       # stop service
hermes gateway restart    # restart service
```

Profile Commands

```
hermes profile list              # list profiles
hermes profile create <name>     # create new profile
hermes profile use <name>        # set default profile
hermes profile show <name>       # show profile details
```

Emergency Recovery

Hermes won't start:

```
hermes doctor --fix
# check ~/.hermes/logs/ for error logs
```

Tool not working after enabling:

```
/hermes new   # start fresh session
```

Config change not taking effect:

```
# Exit Hermes completely, then relaunch
hermes chat
```

Cron job not firing:

```
hermes cron status   # check scheduler is running
hermes cron list     # check job exists and is not paused
```

Gateway not responding on Telegram:

```
tail -f ~/.hermes/logs/gateway.log
# Check bot token is correct in ~/.hermes/.env
# Check WSL2 /etc/wsl.conf has systemd=true
```

Update cache stale (shows wrong "behind" count):

```
rm -f ~/.hermes/.update_check
```

Scratch workspace files gone after task archive:

- This can't be recovered. Always use `--workspace dir:/absolute/path` for persistent output tasks.

Final Note

You made it.

If you've been reading straight through, you now have the mental model, the workflows, the failure modes, and the reference material to actually run Hermes as an operator.

The thing I'd tell you that isn't in any chapter: the setup friction is real and it's worth it. The first workflow takes longer than the fifth. By the time you've got three workflows running, you'll start seeing the next five without me telling you what they are.

Start with one. The research pipeline is a good first one because it's self-contained and the output is a file you can verify immediately. Get that working, then add the daily briefing, then the repo debugging.

The operator frame is the hard part. Everything else follows from that.

Now go run your projects. Remember your context. Sleep while it works.

If this helped you, find me on X (@tonysimons_). I post about Hermes workflows, local AI setups, and the operator frame.

Built with Hermes Agent. No fluff. No course upsell. Just the real setup.

Changelog

Changes from v1 -> v2 (based on voice-growth-review.md and technical-review.md):

Blocking corrections applied

- Ch4 Workflow D Kanban diagram: Fixed garbled `--workspace dir:~/research/---` syntax to current CLI format, `--workspace dir:/absolute/path`
- Ch5 `/background` command: Fixed `/hermes background` to correct in-session form `/background`
- Ch5 `/skill` command: Fixed `/hermes skill <name>` to correct in-session form `/skill <name>` (Appendix)
- Workflow C timing: Fixed 6:45 AM self-contradiction in Ch3 Cron conceptual description and Ch4 Workflow C opening line; all references now consistently say 6:20 AM (cron expression is source of truth). Personal wake time "I'm usually awake by 6:45" in Timing note is preserved (that's Tony's schedule, not a scheduling discrepancy)
- WSL2 caveat: Removed "in Windows" from `/etc/wsl.conf` phrasing -- it is a Linux path inside WSL, not a Windows file

Expansion and rewrite

- Ch5 (The Operator Loop): Expanded significantly (+30-40%) per reviewer request. Added "Context Management for Long Sessions" section with specific guidance on when to compress, when to start fresh, and file-based checkpoint strategy for multi-day projects. Added decision tree in "When to Intervene" section. Added concrete intervention vs. non-intervention criteria.
- Ch7 (The Next Layer): Fully rewritten. Replaced feature-list voice with operator-specific context. Added specific gotchas (Kanban workspace isolation, task sizing limits), Vault characterization adjusted to acknowledge it is

production-grade and that hardcoded `.env` is fine for non-production. Added MCP `--url` form to command examples. Strengthened closing line.

Non-blocking notes (not changed, for reviewer awareness)

- "15+ LLM providers" and "15+ messaging platforms" numbers in Ch1: left as-is; conservative estimate that doesn't overstate. SKILL.md names 16 platforms total.
- Hermes Vault description: characterized per Tony's authorized knowledge; not directly sourced in research docs.
- Chapter 6 (failure modes): shipped as strongest chapter per both reviewers; verified commands against SKILL.md.
- Chapter 1 (Claude Code / Codex comparison): comparison class claim is general characterization, not a specific sourced number.

Growth CTA

- Added "find me on X (@tonysimons_)" note to Final Note -- addresses reviewer gap about ebook readers who arrive via non-X channels.