ORANGE BOOK

Hermes Agent
The Complete Guide

The first AI Agent that ships with reins built in
A practical guide to the Nous Research open-source framework

The Agent That Grows With You

Keywords: Self-improving Agent • Cross-session Memory • Skill System • MCP • Multi-platform

For: Developers and AI enthusiasts who want to build a personal AI Agent

Version: v260408

HuaShu

WeChat: 花叔 • Bilibili: AI进化论-花生

Based on Hermes Agent v0.7.0. AI tools evolve rapidly — some content may change with future versions. Refer to official docs for the latest.

Contents

TABLE OF CONTENTS

§01 Not Another Agent: From Harness to Hermes

Not Another Agent: From Harness to Hermes

The OpenClaw craze hasn't even died down, and here comes Hermes Agent. It's not "yet another Agent tool" — it's the first time the concept of Harness Engineering has been turned into a product.

Another one?

I get it. You're tired.

OpenClaw kicked off the lobster fever in late 2025. 26 million users. Every major tech company in China rushed to build their own lobster-like product. Your social feeds were probably flooded with "I'm raising a lobster" posts for weeks. And now, before the lobster craze has fully faded, something new pops up.

In February 2026, Nous Research released Hermes Agent. In under two months, **GitHub stars shot past 27,000**.

Your first reaction is probably: I still haven't figured out the lobster thing, and now this?

I spent a week tearing Hermes apart from top to bottom, and found that it's taking a completely different path from OpenClaw. **Hermes isn't another lobster. It's building something we've been talking about for a while but nobody had actually shipped as a product.**

What is Harness Engineering

If you've read the previous orange book *Harness Engineering*, feel free to skip this part. If not, here's the 30-second version.

In early 2026, a consensus emerged in the AI coding world: **the bottleneck isn't the model — it's the environment**. The LangChain team ran an experiment using the same model (GPT-5.2-Codex), only adjusting the surrounding "harness" configuration. Scores jumped from 52.8% to 66.5%, rankings leaped from Top 30 to Top 5. Not a single line of model code was changed.

Mitchell Hashimoto (creator of Terraform) was the first to name this: **Harness Engineering**. His approach was straightforward — every time the AI made a mistake, he'd add a rule so it would never make the same mistake again. The file was alive, always growing.

In that book, I broke the Harness down into five components. These five components are the key to understanding Hermes.

The five-component mapping

Harness Engineering is a methodology — it tells you "what kind of harness you should build for your AI." But methodologies have a problem: **execution is entirely manual**. You have to write CLAUDE.md yourself, configure hooks yourself, build a memory system yourself, design workflows yourself.

What Hermes did is: build all five components in.

Harness Component	Manual Implementation	Hermes Built-in System
Instruction Layer	Hand-write CLAUDE.md / AGENTS.md	Skill system (markdown skill files, auto-created + self-improving)
Constraint Layer	Configure hooks / linter / CI	Tool permissions + sandbox + toolset enabled on demand
Feedback Layer	Manual review / evaluator Agent	Self-improving Learning Loop (auto-retrospective after each task)
Memory Layer	Manually maintain knowledge base	Three-layer memory (session/persistent/Skill) + Honcho user modeling
Orchestration Layer	Build your own multi-Agent pipeline	Sub-Agent delegation + cron scheduling

Look at the left column versus the right. The left side is all manual — you'd need to be an experienced engineer to set it up. The right side is out of the box, **ready the moment you install it**.

This is the fundamental difference between Hermes and OpenClaw. OpenClaw gives you a configuration-as-behavior system — you write a SOUL.md, and it becomes what you want. Its memory system is capable (Daily Logs + MEMORY.md + semantic search) and its Skill ecosystem is massive, but Skills are primarily written and maintained by hand. Hermes has all five dimensions built in, and they run automatically.

> **Connecting the dots:** If you've used Claude Code's CLAUDE.md + hooks + memory, you've already been doing Harness Engineering manually. What Hermes does is turn that manual workflow into an automated system. From "you build the harness for the AI" to "the AI builds its own harness."

Why Nous Research built this

The team behind Hermes Agent isn't a big company — it's an open-source AI research lab.

Nous Research has been described as "the mysterious force in the open-source community." The key figure, Teknium, co-founded the lab and leads post-training. Early on they relied on Redmond AI for compute, and the team has always been small. Yet the Hermes model family they produced (from Hermes 3's 8B/70B/405B to Hermes 4's 14B/70B/405B) **reached frontier-level performance through post-training alone**. No pre-training from scratch, no massive compute budget required.

This philosophy carries over to Hermes Agent: **using open-source tools + any LLM API, even individuals can deploy AI Agents that rival commercial solutions**. MIT license, fully open source.

Their core principles are pretty clear. User control comes first. The model is steerable — you can adjust its behavior as needed, free from corporate content policies. They explicitly don't do censorship, stating the model is

"unencumbered by censorship, neutrally aligned." At the same time, they don't compromise on creativity, math, coding, or reasoning performance.

These principles shaped Hermes Agent's design philosophy: **it doesn't make decisions for you — you set the rules, it learns the rules, then it gets better and better**. Control stays with the user; the system handles the complexity of execution.

Not a replacement, but a progression

There's a common misconception: Hermes is here to replace Claude Code or OpenClaw. It's not. These three tools solve problems at different levels.

Claude Code does interactive coding. You sit at the terminal, going back and forth, collaborating in real time. It's your pair-programming partner.

OpenClaw does configuration-as-behavior. You write a SOUL.md, and it becomes what you want. Configuration is transparent, the ecosystem is mature, with 5,700+ community Skills on ClawHub.

Hermes does autonomous background work + self-improvement. You don't need to sit beside it. It runs on its own, learns on its own, evolves on its own. Online 24/7, reachable anytime through Telegram or Discord.

An interesting point: all three tools use the agentskills.io standard, so Skills are interoperable. A Skill you wrote in Claude Code works in Hermes too, and vice versa. They're not three parallel lines — more like **three roles with different jobs in the same ecosystem**.

> **核心建议**
> If you're just writing code, Claude Code is plenty. If you want a 24/7 background Agent that watches over your tasks and gets smarter on its own, that's when you should look at Hermes.

Ships with the harness built in

Back to the question we started with: why is this time different?

The lobster craze taught everyday users something: AI can be "something you raise," not just "something you open and use." OpenClaw's SOUL.md, memory system, and personalization let people experience for the first time what "my AI" really feels like.

But lobster owners also discovered a problem: **you have to build the entire harness yourself**. Writing SOUL.md, manually tweaking skills, periodically cleaning up memory. The lobster gets better with use, but only if you're willing to invest the time feeding it.

Hermes takes a different approach: **the harness comes welded into the factory settings, and it grows on its own.**

From your very first conversation, Hermes starts automatically writing to memory, extracting Skills, and optimizing workflows. The longer you use it, the deeper its understanding of you, the higher the quality of its work. You're not training it — it's training itself.

Hermes is the first Agent that ships with the harness built in. And the harness grows on its own.

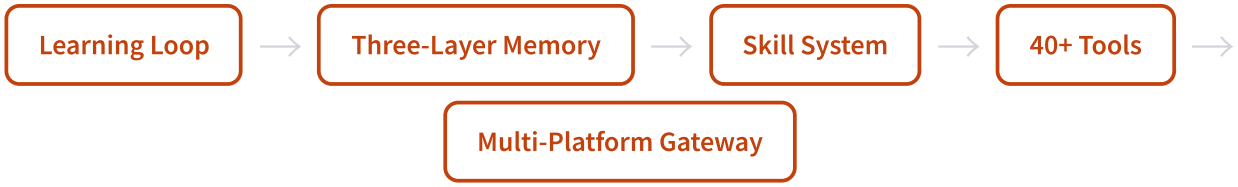
§02 Hermes Agent at a Glance: 60 Seconds

Hermes Agent at a Glance: 60 Seconds

One flow chart, one set of numbers, one comparison table. After these three, you'll know what Hermes is all about.

Architecture in one picture

Hermes Agent's architecture can be strung together in a single line:



From left to right, each module's job in one sentence:

The Learning Loop is Hermes's heart. After completing each task, it automatically does a retrospective: what should be remembered, what Skills should be extracted, do existing Skills need optimization. This loop runs continuously — you never have to trigger it manually.

Three-Layer Memory is Hermes's brain. Session memory remembers "what just happened," persistent memory remembers "who you are and what you like," and Skill memory remembers "how to do things." Each layer has its own job, stored in SQLite + FTS5 indexes, retrieved on demand rather than loaded all at once.

The Skill System is Hermes's skill library. Each Skill is a standalone markdown file, stored in the `~/.hermes/skills/` directory. There are three sources: bundled with the repo, created by the Agent itself, or installed from the community Hub. The key feature: **Skills aren't static — they self-improve through use**.

40+ Built-in Tools are Hermes's hands and feet. Five categories: execution (run code, manipulate files), information (search, scrape), media (images, video), memory (read/write storage), and coordination (sub-Agent delegation). Plus MCP integration, connecting to 6,000+ external applications.

Multi-Platform Gateway is Hermes's front door. Telegram, Discord, Slack, WhatsApp, Signal, CLI — 14 platforms supported. You can message it on Telegram, it processes in the background on your VPS, and conversations stay continuous across platforms.

Key numbers

Version v0.7.0, released April 3, 2026. A few numbers worth noting:

Metric	Data
GitHub stars	**27,000+** (two months after release)
First month growth	6,000+ stars
Built-in tools	40+
Supported platforms	14
MCP integrations	6,000+ apps
Sub-Agent concurrency	Up to 3
Minimum deployment cost	$5/month VPS
Memory usage	<500MB (without local LLM)
License	MIT (fully open source)

27,000+ stars in two months — that's fast. Keep in mind OpenClaw only reached its current scale thanks to the lobster craze's social virality. Hermes hit these numbers without any comparable viral effect, which tells you the developer community genuinely feels it solves a real problem.

Key differences from OpenClaw

After the lobster craze, the question everyone's asking is: what's actually different between Hermes and OpenClaw?

Dimension	Hermes Agent	OpenClaw
Core philosophy	Self-improving Learning Loop	Configuration-as-behavior (SOUL.md)
Memory	Three-layer self-improving (session/persistent/Skill)	Multi-layer memory (Daily Logs/MEMORY.md/semantic search), primarily manually maintained
Skill maintenance	Agent auto-creates + self-improves	Manually written and maintained
User modeling	Honcho dialectical modeling (12-layer identity inference)	Based on SOUL.md configuration
Multi-platform access	14-platform Gateway	50+ messaging platforms (Telegram/Discord/WhatsApp/Slack/Signal, etc.)
Ecosystem scale	40+ built-in tools + MCP 6,000+	ClawHub 5,700+ community Skills
Deployment	Self-hosted (from $5 VPS)	Official hosting / self-hosted
Skill interop	Both use agentskills.io standard	

The two biggest gaps are **learning ability and user modeling**. OpenClaw's Skills are primarily written and tuned manually — their evolution depends on the community and active user maintenance. The longer you use Hermes, the more precise its Skills become, the deeper its memory, the smoother its execution.

But OpenClaw has something Hermes can't match: **ecosystem maturity**. 5,700+ community Skills on ClawHub, with ready-made solutions for all kinds of scenarios. Hermes's community is still in its early stages. The network effects from 26 million users in the lobster craze gave OpenClaw a massive head start that technology alone can't bridge.

> **One-line distinction:** OpenClaw is a lobster you raise yourself. Hermes is a lobster that grows on its own. One depends on your careful feeding; the other learns from its own experience.

$5 gets you up and running

Cost is something a lot of people care about. The answer might surprise you.

Hermes itself is free, MIT open source. You only pay for LLM API calls. Deployment cost depends on what setup you choose:

Cheapest option: any $5/month VPS (Hetzner CX22 runs about $4/month; DigitalOcean, Vultr work too), Ubuntu 22.04. Without running a local LLM, memory usage stays under 500MB. Pair it with OpenRouter using Claude Haiku or DeepSeek, and API costs stay low too.

Even cheaper: Serverless. Use Daytona or Modal as your backend — the environment hibernates when idle, wakes up automatically when a message comes in. Between-session costs are practically zero.

Privacy-focused: Run Ollama on your VPS with a local 8B or 70B open-source model. API costs are zero, but you'll need a beefier VPS (16GB+ RAM recommended).

No matter which option, **a $5 VPS + Telegram Bot gives you a personal AI Agent running 24/7**. That cost-performance ratio beats subscription-based commercial Agent solutions by a mile.

> **核心建议**
>
> For comparison: Claude Code Pro subscription is $20/month, Max subscription is $200/month. With Hermes's $5 VPS + API cost setup, monthly expenses stay under $10 for most use cases. Of course, they're positioned differently so a direct price comparison isn't entirely fair — but the barrier to entry is undeniably much lower.

Who is this for

One last question: is this book for you?

If any of the following applies, keep reading:

You've used Claude Code or OpenClaw and want an Agent that can **run background tasks autonomously**. Not the kind where you sit and watch — the kind that keeps working while you sleep.

You know about Harness Engineering and you're curious to see **what it looks like when the methodology becomes a product**.

You want to deploy a private AI Agent on your own VPS, where **your data never leaves your server**.

You're simply curious about what an open-source project that hit 27,000+ stars in two months actually got right.

In the following chapters, we'll start with the Learning Loop and peel back Hermes's core mechanisms layer by layer.

§03 The Learning Loop: Self-Harnessing Agent

The Learning Loop: Self-Harnessing Agent

The most surprising thing about Hermes Agent isn't what it can do — it's that it changes. The more you use it, the better it gets. This isn't marketing speak. It's an observable, verifiable closed-loop mechanism.

Starting with a real scenario

Say it's your first time asking Hermes to write a Python script. You tell it: write me a scraper that grabs titles and summaries from a certain website.

It'll produce a working script. But the style might not be what you prefer, variable naming might not match your conventions, error handling might not be the way you'd do it. Perfectly normal — it doesn't know you yet.

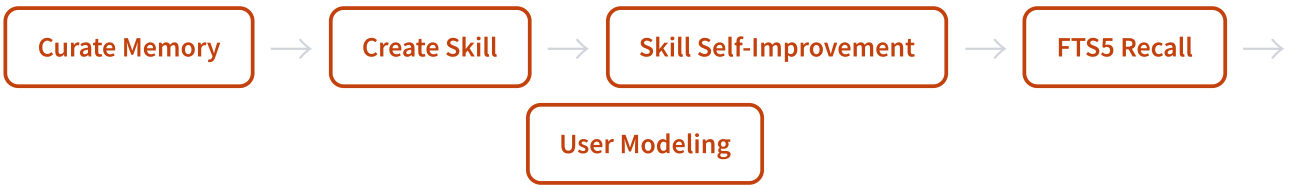
By the tenth time, everything's different. It knows you prefer httpx over requests. It knows you like writing error logs to a file instead of printing to the terminal. It knows your project structure is typically organized by module under a src/ directory. It knows you hate long functions.

Nobody taught it any of this. It figured it out on its own.

That's what the Learning Loop does.

Five steps, one closed loop

Hermes's Learning Loop has five steps. Individually, none of them are complicated, but strung together they form a continuous improvement flywheel.



Looks like five independent features? They actually have causal relationships. Memory provides the raw material for Skill creation. Skills accumulate feedback during use, triggering self-improvement. FTS5 enables precise recall of historical experience. User modeling assembles these fragments into a complete picture.

Let's break them down one by one.

Step one: Memory curation

After each conversation, Hermes actively decides what's worth remembering. **Actively decides — not passively stores.**

Traditional conversation memory is brute-force: shove the entire chat history into the context window. The more you chat, the longer the context, until it overflows. It's like a person trying to hold every experience they've ever had in their mind at once. Normal people can't do that.

Hermes works more like a person writing a diary. After each conversation, it looks back: what was this about? Any new discoveries? Did the user express any preferences? Then it writes what's worth keeping into a SQLite database, with FTS5 full-text indexing.

The system also has a periodic nudge mechanism that reminds the Agent to review recent interactions. Kind of like a journaling app sending you a notification: anything worth recording today?

Step two: Autonomous Skill creation

When Hermes finishes a reasonably complex task, it asks itself a question: **will this solution be useful again in the future?**

If the answer is yes, it distills the solution into a Skill file and saves it to `~/.hermes/skills/`. This Skill is a markdown file containing the task description, execution steps, and things to watch out for.

Here's an example: you ask Hermes to clean a CSV file and import it into a database. After it's done, it might create a Skill called `csv-to-database.md`, recording common data-cleaning steps, your preferred database connection method, and the field validation rules you typically need.

Next time you say "import this CSV for me," it doesn't start from scratch. It loads that Skill and follows the approach you've already validated.

Step three: Skill self-improvement

Creating a Skill isn't the end. Every time it's used and you provide feedback, Hermes takes that feedback and modifies the Skill itself.

Say you tell it "this import script should check if the table exists first." Hermes doesn't just add the check this one time — **it goes back and edits the Skill file**, writing that rule in. Next time the Skill is used, the check is included by default.

This process is a lot like continuous improvement in software development. Every time a production issue happens, you don't just patch it — you update the documentation and standards too, preventing the same class of problem from recurring.

> **Key distinction:** Traditional AI tools have memory that's an accumulation of conversation logs. Hermes's memory is a distillation of experience. One is a video tape, the other is a notebook. Video tapes keep getting longer until they overflow; a notebook you can use indefinitely.

Step four: FTS5 cross-session recall

Remembering all this stuff is one thing. The real trick is finding the right piece at the right time.

Hermes uses SQLite's FTS5 extension for full-text indexing. Before each new conversation, it searches historical memory based on the current topic and loads only the relevant parts into context. Not all history — just what's needed.

This matters more than you'd think. Most AI tools either don't remember what you said last time or dump everything in and slow to a crawl. Hermes's approach: ask a database question, it searches database-related memories; ask a frontend question, it searches frontend memories. Like a well-organized note system with a table of contents and index — you look up what you need.

FTS5 has another benefit: it's purely local. Your memory data doesn't need to be uploaded to any server — it's all in a local SQLite file. When you move machines, just copy the `~/.hermes/` directory.

Step five: User modeling

The final step is Honcho user modeling, an optional external integration. What it does goes beyond remembering what you said: **it infers what kind of person you are.**

After each conversation, Honcho analyzes the exchange and derives your preferences, habits, and goals. These derivations aren't just records of what you said — they're deeper patterns generalized from your behavior.

For example, you never explicitly said "I prefer concise code style," but Honcho inferred it by analyzing the patterns in how you modify code across multiple sessions. Next time it generates code, it defaults to the concise approach.

We'll dig deeper into Honcho's 12-layer identity modeling in the next chapter.

Mitchell Hashimoto's philosophy, automated

If you read the Harness Engineering orange book, you might remember Mitchell Hashimoto's approach. When using Claude Code, he had a habit: **every time the Agent made a mistake, he'd add a rule to CLAUDE.md.**

"Don't use the any type in this project."

"Put test files in the __tests__ directory, not in src."

"Write commit messages in English, starting with a verb."

One rule at a time. After a few weeks, CLAUDE.md became an incredibly detailed project spec. The Agent went from a clueless newcomer to a veteran who knew every unwritten rule of the project. Mitchell said it felt like training a new team member.

What Hermes does is essentially the same thing, but automated.

You don't need to manually write CLAUDE.md. You don't need to summarize rules yourself after every mistake. Hermes observes on its own, summarizes on its own, writes Skills on its own, and applies those rules on its own the next time around.

Dimension	Mitchell's Way (Manual)	Hermes's Way (Automated)
Rule source	Human spots a problem, writes it down	Agent extracts from its own feedback
Storage location	CLAUDE.md (single file)	Multiple Skill files + memory database
Improvement trigger	Only when the human remembers to add a rule	Automatic evaluation after every use
Cross-project portability	Manually copy CLAUDE.md	Skills are global, shared across all projects
Improvement speed	Depends on how diligent the human is	Continuous and automatic — never gets lazy

Of course, automated doesn't mean perfect. Mitchell's hand-written rules tend to be more precise, because humans have a clearer understanding of their own needs. Hermes's auto-generated rules may need tweaking, may misjudge. But here's the thing: **it drops the barrier to zero.** Not everyone has Mitchell's patience to maintain a finely-tuned rule file. Hermes lets people who don't want to mess with configuration still enjoy the "gets better with use" experience.

The flywheel acceleration effect

None of the five steps are individually novel. Memory, Skills, retrieval, user profiling — the AI field has seen all of these before.

Hermes's innovation is wiring them into a closed loop. Memory feeds Skill creation. Skill usage generates new memories. New memories trigger Skill improvement. Improved Skills produce better results. Better results make user modeling more accurate. More accurate profiling makes the next round of memory curation more targeted.

This is a positive feedback loop. The more you use it, the stronger every step gets — and they get stronger simultaneously. Like Amazon's flywheel: more users bring more data, more data brings better recommendations, better recommendations bring more users.

The difference is that Hermes's flywheel spins for a single user. It doesn't need data from millions of users to improve — just your own usage history. Use it for three to five days, and you'll notice a clear difference.

> **核心建议**
> The Learning Loop's effectiveness is directly tied to how often you use it. If you only use it once or twice a week, improvement will be slow. But if you treat it as your daily work partner, using it every day, the flywheel spins remarkably fast. That's why Hermes is particularly well-suited for deploying on a $5 VPS running 24/7 — let it keep accumulating.

What this means

Back to the title: the Agent builds its own harness.

In Harness Engineering, the harness is human-made. You write CLAUDE.md, configure hooks, design feedback mechanisms. All of that requires ongoing human investment.

Hermes's approach: **let the Agent weave its own harness while it runs.** When it veers off course, it self-corrects and remembers the lesson. When it finds a good method, it distills it into a Skill for future reuse. When it encounters a new user, it builds its own understanding model.

This doesn't mean humans are out of the picture. You can manually edit Skill files anytime, delete inappropriate memories, adjust the user profile. But by default, the system is self-driven.

Next chapter, we look at the most critical infrastructure in this loop: the three-layer memory system. If the Learning Loop is the engine, memory is the fuel.

§04 Three-Layer Memory: From Goldfish to Old Friend

Three-Layer Memory: From Goldfish to Old Friend

Most AI chat tools have the memory of a goldfish — whatever was said last round is forgotten by the next. Hermes aims to be an old friend: one who remembers what you said, knows what kind of person you are, and has learned how you like to get things done.

Why memory is the hardest problem

You might think AI memory is just saving chat logs. Store them, load them next time, done.

Not that simple. An active user chats thousands of words a day with their AI. That's tens of thousands per month. Cram it all into the context window and either it won't fit, or the model gets sluggish from information overload. And most of a chat log is noise and repetition — the actually valuable information might be just 10%.

Good memory isn't about storing more — it's about finding what matters.

Hermes solves this with a three-layer architecture, each layer handling a different type of memory.

Layer one: Session memory

Session memory answers the question: **what happened?**

Every conversation's content, tool calls, and return results are written to a SQLite database with FTS5 full-text search indexing. This is episodic memory, analogous to the hippocampus in the human brain.

The key design decision is on-demand retrieval rather than loading everything. When a new conversation starts, Hermes doesn't stuff all past conversation history into context. It searches relevant historical fragments using FTS5 based on the current topic, loading only what's needed.

The benefits of this approach:

Approach	Load Everything	On-Demand Retrieval (Hermes)
Context usage	Grows linearly with conversation volume	Essentially constant
Retrieval precision	Everything's there, but nothing's findable	Precise keyword matching
Long-term viability	Hits the wall after a few days	Works for months, even years
Response speed	Gets slower over time	Stays essentially the same

FTS5 is SQLite's full-text search extension — no extra database installation needed. All data lives in local SQLite files, **no network dependency, no privacy concerns**. Your conversation memory never leaves your machine.

Layer two: Persistent memory

Persistent memory answers the question: **who are you?**

This layer doesn't store conversation content — it stores durable state distilled from conversations. Things like your coding preferences, project structure habits, commonly used toolchain, work schedule patterns. These persist across sessions and don't disappear when you start a new conversation.

Technically, persistent memory is also stored in SQLite, managed through the `memory` tool. The entire solution is file-level: no external servers, no cloud sync, data lives in the `~/.hermes/` directory.

This means you can:

- Back up `~/.hermes/` to a USB drive and continue on a different machine
- When deploying with Docker, mount the `/opt/data` directory to the host to persist state
- Sync across devices using cloud storage (real-time SQLite sync isn't recommended, but periodic copying works fine)

Portability is an underrated feature. Many AI tools lock your memory in the cloud — switch tools and you start from zero. Hermes's memory is your own files, portable however you want.

Layer three: Skill memory

Skill memory answers the question: **how to do things?**

The first two layers remember what happened and who you are. The third layer remembers methodologies and operating procedures. Each Skill is a markdown file in `~/.hermes/skills/`, human-readable and editable.

These three layers correspond to three types of memory in cognitive science:

Learning to ride a bicycle is all three layers working together: you remember falling last time (episodic), you know to keep your center of gravity low (semantic), and your body automatically balances (procedural). Hermes handles tasks the same way: it remembers how you edited code last time, knows your preferences, and has a proven execution plan at hand.

Three layers in action: You say "help me deploy this project." Hermes first searches session memory with FTS5, finding the port conflict you hit during your last deployment (episodic). Then it checks persistent memory, knowing you use Alibaba Cloud ECS with Nginx reverse proxy (semantic). Finally it loads the deployment-checklist Skill and follows the steps you've already validated (procedural). Each layer doing its job.

Honcho: Knows you better than you know yourself

Beyond the three local memory layers, Hermes has an optional add-on: the Honcho user modeling system, developed by Plastic Labs.

What Honcho does goes a step deeper than remembering what you said. It infers your characteristics as a person. The official term is dialectical user modeling, covering 12 identity layers.

What do these 12 layers actually mean? Let's use a scenario to illustrate.

Say you've been asking Hermes to write Python scripts every day for three straight weeks. During this time, Honcho might infer:

- **Technical level**: You're not a complete beginner, but not an expert either. You can read code but struggle to write it from scratch
- **Work rhythm**: You're usually active from 9-11 PM, likely a personal project after work
- **Communication style**: You prefer seeing results first and asking about principles later; you don't like long explanations
- **Goal inference**: You're probably working on a data analysis project, since recent tasks all revolve around data processing
- **Emotional patterns**: You get a bit frustrated when code throws errors; concise, direct answers work better in those moments
- **Preference contradictions**: You say you want comprehensive comments, but when you actually review code you never read them

Notice that last one. **Honcho catches inconsistencies between what you say and what you do.** Your stated preferences and your revealed preferences may differ — dialectical modeling pays attention to both.

These inferences get injected into subsequent conversation prompts as invisible context. You don't see this information, but you feel Hermes becoming more attuned to you.

Compared to Claude Code's auto-memory

Claude Code also has a memory system: CLAUDE.md files and auto-memory. How do they compare with Hermes's memory?

Dimension	Claude Code	Hermes Agent
Memory format	CLAUDE.md + auto-memory text files	SQLite database + FTS5 index + Skill files
Write method	CLAUDE.md manually written, auto-memory semi-automatic	Fully automatic, human can override anytime
Retrieval method	CLAUDE.md loaded in full at startup	On-demand FTS5 full-text search
Memory granularity	Project-level (one CLAUDE.md per project)	Both global-level and project-level
User modeling	None (user writes their own preferences)	Honcho auto-infers user profile
Procedural memory	Instructions in CLAUDE.md	Standalone Skill files, self-improving
Cross-project sharing	~/.claude/CLAUDE.md (global instruction file)	All memory is inherently global
Storage limit	CLAUDE.md recommended at a few KB	SQLite's theoretical limit is very high

The two have different design philosophies. Claude Code's CLAUDE.md follows a human-writes, AI-executes model — the upside is full human control, the downside is it requires ongoing maintenance. Hermes follows an AI-writes, human-reviews model — the upside is low barrier and high automation, the downside is auto-generated content may not always be accurate.

Which is better depends on the use case. If you're a heavy Claude Code user who's spent weeks carefully crafting your CLAUDE.md, your hand-woven harness may be more precise than what Hermes auto-generates. But if you don't want to spend time maintaining config files, Hermes's fully automated approach is genuinely easier.

Memory isn't a silver bullet

After all the good stuff, let's also talk about limitations.

What should be remembered:

- User preferences and habits (code style, tool choices, communication style)
- Project context (architecture decisions, tech stack, file structure)
- Validated solutions (Skills)
- Recurring patterns (like how to handle certain classes of errors)

What shouldn't be remembered:

- One-off task details (write me a birthday greeting — no need to remember that)
- Outdated information (an API version number from three months ago that's probably changed)
- Wrong inferences (Hermes may misjudge your preferences — these should be cleaned up)

- Sensitive information (passwords, keys, personal identity info shouldn't enter the memory store)

> **注意**
>
> Hermes's memory system currently has no automatic expiration mechanism. With long-term use, the memory database will keep growing. It's a good idea to periodically check the size of the `~/.hermes/` directory and clean up outdated Skill files. This is a known area for improvement.

There's also the problem of memory pollution. If Hermes remembers incorrect information from early conversations, that error can persist and affect later behavior. For instance, if it mistakenly remembers you prefer Python 2, subsequent generated code might carry Python 2 syntax.

So periodic memory audits are worth doing. Check which Skills are in `~/.hermes/skills/`, delete ones that don't fit. Review persistent memory, correct wrong inferences. Like tidying up a notebook — flipping through it occasionally, you'll find plenty that needs updating.

From storage to understanding

Back to the title: from goldfish to old friend.

A goldfish's problem isn't that it has no eyes — it's that it has no memory. Every time it sees you is like the first time. Most AI tools are like this: open a new conversation, and everything starts from scratch.

An old friend is different. An old friend knows your temper, your habits, knows that when you say you don't care, you actually do. An old friend doesn't need you to explain the backstory every time, because the backstory is already there.

Hermes's three-layer memory plus Honcho user modeling is walking this path. Session memory provides the raw material, persistent memory provides the profile, Skill memory provides the methodology, and Honcho assembles these fragments into a complete understanding of you.

The longer you use it, the deeper the understanding. This isn't a slogan — it's a mathematical inevitability of three-layer memory plus a closed-loop learning system.

Next chapter, we look at another core mechanism in Hermes: the Skill system. If memory is "knowing," Skills are "being able to do."

§05 The Skill System: Capabilities That Evolve on Their Own

Skills: Self-Evolving Capabilities

OpenClaw's Skills require you to write and maintain them by hand. Hermes' Skills grow on their own and get better over time. This difference defines two entirely different user experiences.

What Exactly Is a Skill

In Hermes, each Skill is a standalone markdown file stored in the `~/.hermes/skills/` directory. It captures the agent's procedural memory for how to do something.

Think of it this way: you're training a new colleague to write weekly reports. The first time, you walk them through every step. The second time, they still ask a few questions. By the third time, they've got it. **A Skill is that "after the third time" state** — the agent solidifies the method into a reusable document.

Skills come from three sources:

Source	Description	Scale
Bundled Skills	Pre-built capabilities that ship with the install, covering common scenarios like MLOps, GitHub workflows, and research	40+
Agent-Created	After completing complex tasks, the agent automatically distills solutions into Skills	Grows with usage
Skills Hub	Community-contributed skill packs, installable with one click	Continuously growing

These three sources aren't equal. Bundled Skills are the starting point, Skills Hub is the accelerator, but **agent-created Skills are Hermes' real killer feature**.

agentskills.io: A Universal Language for Skills

Hermes' Skills aren't a walled garden. They follow the agentskills.io standard, which is already supported by 30+ tools including Claude Code, Cursor, Copilot, Codex CLI, and Gemini CLI.

Skills you wrote for Claude Code work directly in Hermes. And vice versa.

This is different from the App Store model. App Stores mean one ecosystem per platform, forcing developers to build for each one. agentskills.io is more like a USB port — one Skill plugs in anywhere and just works.

For people already using Claude Code, your accumulated Skill assets aren't locked into any single tool. Switch to Hermes, or use both simultaneously — Skills migrate seamlessly.

Self-Improvement: Skills Get Better With Use

This is the biggest difference between Hermes and every other agent Skill system out there.

Traditional Skills require manual maintenance. You write a code review Skill, and it follows your steps exactly. Discover that a certain step doesn't work well in practice? You have to go in and fix it manually. That's primarily how OpenClaw's 5,700+ community Skills operate.

Hermes' Skills are alive. They run inside a learning loop, automatically optimizing based on real feedback.

Here's the mechanism:

1. **Execute the Skill**
   The agent follows the steps recorded in the Skill to complete the task

2. **Collect Feedback**
   The user's reactions (satisfied / unsatisfied / corrections) get logged into session memory

3. **Update the Skill**
   The agent analyzes feedback and automatically modifies the relevant steps in the Skill file

4. **Next Execution Uses the New Version**
   The improved Skill takes effect automatically in subsequent tasks

Sounds idealistic? It is, to some extent — results depend on the LLM's capabilities and feedback quality. But the direction is right: **let the agent learn from experience instead of waiting for someone to maintain it.**

> **Comparison with Mitchell Hashimoto:** Mitchell said that when using Claude Code, he "adds a rule to CLAUDE.md every time it makes a mistake." Hermes automates that process. You don't have to add rules yourself — the agent observes, summarizes, and writes them into Skills. The trade-off is you give up some control over the rules.

Key Differences from OpenClaw Skills

OpenClaw's ClawHub has 5,700+ community Skills, far outnumbering Hermes. But the design philosophies are completely different:

Dimension	OpenClaw Skills	Hermes Skills
Creation	Manually written SOUL.md	Agent-created + manually written
Maintenance	Manual updates	Auto-evolution + manual intervention
Personalization	Generic templates, fork and customize	Grows organically from your usage habits
Interoperability	agentskills.io standard	agentskills.io standard (interoperable)
Ecosystem Size	5,700+ (large)	40+ bundled + community (growing)

OpenClaw's strength is scale and transparency. One look at the SOUL.md tells you exactly what a Skill does — every step is something you wrote, so you know what's going on.

Hermes' strength is adaptability. The same "write code" Skill, after three weeks of use by a Python developer and a Rust developer, will have evolved into two completely different versions. Not a generic template — a custom fit.

The two aren't mutually exclusive. The agentskills.io standard lets Skills move between them. You can absolutely install a Skill from ClawHub into Hermes, then let Hermes continuously improve it through use.

In Practice: Letting Hermes Create a Skill on Its Own

Enough about mechanisms — let's look at a concrete example.

Say every morning you need Hermes to sort through yesterday's GitHub notifications, summarizing important PRs and Issues. The first few times, you have to spell out the request each time:

> **核心建议**
> "Go through my GitHub notifications from yesterday, sort by importance, separate PRs and Issues, and ignore automated bot notifications."

After the third or fourth time, Hermes does something in the background: it distills this recurring task pattern into a Skill file. You'll find a new markdown file in `~/.hermes/skills/`, looking something like this:

```
# GitHub Daily Digest

## Trigger Conditions
User mentions "GitHub notifications", "daily summary", etc.

## Steps
1. Call GitHub MCP to fetch notifications from the past 24 hours
2. Filter out automated notifications from bot accounts
3. Group by type (PR / Issue / Discussion)
4. Sort by importance (mention > review request > other)
5. Present as a concise list

## User Preferences
- Only titles and status needed, no detailed content
- PRs and Issues displayed separately
```

From this point on, you just say "check GitHub" and Hermes knows what to do.

What's even more interesting is what happens next. One day you say "include Discussions this time too." Hermes doesn't just add them for this one request — it updates the rules in the Skill file. **Next time, it'll include Discussions even without you asking.**

That's what "self-evolution" actually means in practice. No mysterious AI breakthrough — just an automated loop of "user corrects -> rules update -> next execution applies the change."

> **注意**
> Skill self-improvement requires your feedback to be clear enough. If you just feel "something's off" but don't say what specifically, the agent can't improve accurately. Good feedback = good evolution direction.

§06 40+ Tools & MCP: Connect Everything

40+ Tools & MCP: Connect Everything

No matter how smart an agent is, it can't get real work done without tools. Hermes ships with 40+ built-in tools covering everything from running code to sending messages. MCP extends its reach to 6,000+ external applications.

Five Categories at a Glance

Hermes' tools fall into five categories. You don't need to memorize them all — just know these capabilities exist and look them up when you need them.

Category	Core Tools	What They Do
Execution	terminal, code_execution, file	Run commands, execute code (sandboxed), read/write files
Information	web, browser, session_search	Web search, browser automation, search conversation history
Media	vision, image_gen, tts	Understand images, generate images, text-to-speech
Memory	memory, skills, todo, cronjob	Operate the memory layer, manage Skills, task planning, scheduled jobs
Coordination	delegation, moa, clarify	Delegate to sub-agents, multi-model reasoning, ask user for clarification

A few worth calling out individually:

session_search is a fairly unique Hermes capability. It uses FTS5 full-text indexing to search conversation history, paired with LLM summarization, letting the agent quickly recall "that approach we discussed last week." Most agents don't have this — every conversation starts from scratch.

moa (Multi-model Orchestrated Answering) lets Hermes call multiple LLMs simultaneously, synthesizing their responses into a final answer. Useful for scenarios requiring high reliability, like fact-checking or technical decisions.

cronjob defines scheduled tasks using natural language. Say "check my GitHub notifications every morning at 9am" and it creates a timed trigger. No cron expressions, no scheduler configuration needed.

Toolsets: Not Everything On, but On-Demand

Having all 40+ tools enabled at once doesn't make sense. An agent helping you write code doesn't need Home Assistant permissions, and a calendar-managing agent has no use for code_execution.

Hermes solves this with the Toolset mechanism. Tools are grouped by function and enabled or disabled in config.yaml as needed:

```
# config.yaml example
toolsets:
  - web          # web search
  - terminal     # terminal commands
  - file         # file operations
  - skills       # Skill management
  - delegation   # sub-agent delegation
  # - homeassistant  # comment out what you don't need
  # - rl            # reinforcement learning, most people won't need this
```

This isn't just a feature toggle. **Fewer enabled tools means the agent stays more focused, responds faster, and consumes fewer tokens.** If all you need is a file-organizing assistant, enabling just the file and memory Toolsets is enough.

Toolsets also serve as security boundaries. The constraint layer mentioned in S03 is implemented at the tool level through Toolsets. You get precise control over what the agent can and can't touch.

MCP: A Unified Interface to 6,000+ Apps

The 40+ built-in tools cover general scenarios. But everyone's toolchain is different — you might use Jira for project management, Notion for docs, Slack for communication. How do you get Hermes to work with these?

MCP (Model Context Protocol).

MCP is an open protocol that defines a communication standard between AI agents and external tools. Hermes supports connecting to any MCP Server via stdio or HTTP. The MCP ecosystem currently covers 6,000+ applications — GitHub, Slack, Jira, Google Drive, databases, you name it.

Integration is straightforward — add a config block to config.yaml:

```
# Connect to GitHub MCP Server
mcp_servers:
  github:
    command: npx
    args: ["-y", "@modelcontextprotocol/server-github"]
    env:
      GITHUB_PERSONAL_ACCESS_TOKEN: ${GITHUB_TOKEN}
```

Once configured, Hermes can use GitHub's capabilities: create Issues, review PRs, check repo status. No code to write, no custom tools to build — **MCP Servers are plug-and-play capability extensions**.

Hermes also supports per-server tool filtering. An MCP Server might expose 20 tools, but you might only want the agent to use 3 of them. Tool filtering gives you precise control over the capabilities exposed to the agent.

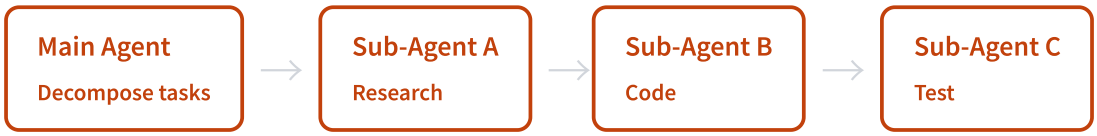
Sub-Agent Delegation: Three Horses Running at Once

Delegation is Hermes' most powerful coordination tool. It can spawn sub-agent instances, distributing tasks for parallel execution.

Independent context. Each sub-agent has its own conversation context, isolated from the others. The main agent passes in necessary background when assigning tasks, and each sub-agent works in its own space.

Restricted toolsets. You can specify which tools each sub-agent can use. The research one only needs web and browser; the coding one only needs terminal and file. This is both an efficiency optimization and a security measure.

Max 3 concurrent. This limit is intentional. Three concurrent sub-agents already cover most parallel scenarios (research, coding, and testing simultaneously), and more would be too hard to coordinate effectively.



If you've used Claude Code's multi-instance parallelism, this will feel familiar. The difference is that Claude Code's multiple instances are manually opened by you, with no coordination between them. **Hermes' sub-agent delegation is the agent autonomously deciding when to distribute tasks and how to consolidate results.**

> **Real-world feel:** Sub-agents work best for "do several unrelated things and then combine the results." For example, ask Hermes to write a technical blog post and it might dispatch one sub-agent to research the latest materials, another to analyze competitor articles, and a third to organize code examples — then the main agent integrates all three into a first draft.

Tool Permissions & Sandboxing: The Constraint Layer in Action

The previous chapters covered the learning loop, memory, and Skills — all mechanisms for making the agent more capable. But the more capable an agent gets, the more constraints matter.

Hermes implements three layers of constraints at the tool level:

Toolset control. Only tools enabled in config.yaml can be called by the agent — the coarsest-grained switch.

code_execution sandbox. Code runs in an isolated environment, separate from your system. Even if the agent executes problematic code, it won't affect your filesystem.

Sub-agent restricted toolsets. When delegating to a sub-agent, you can specify a subset of tools it can use. A sub-agent responsible for searching doesn't need — and shouldn't have — terminal access.

If you've read the Harness Engineering orange book, you'll recognize that **this is exactly how the constraint layer (hooks/linter) is implemented in Hermes**. Same principle: give the agent enough capability to complete tasks, but don't hand over unnecessary permissions.

> **核心建议**
>
> For security-sensitive scenarios (like running on production servers), enable only the necessary Toolsets and use MCP's per-server filtering to further restrict the tools exposed to the agent. Better to have the agent ask "I need XX permission" than to leave everything open by default.

These three layers of constraints together form a pragmatic security model. It doesn't chase theoretically perfect isolation — instead, it finds a balance between usability and security. The more trust you give the agent, the more it can do. But even with default settings unchanged, Hermes won't perform dangerous operations without your knowledge.

§07 Installation & Configuration: Three Approaches

Installation & Configuration

From zero to running in as little as 5 minutes. This section covers three installation methods — from local development to a 24/7 server. Pick the one that fits.

Option 1: Local Install (5 Minutes to Get Started)

Local installation is the most straightforward, perfect for people who want to try it before committing to running it long-term. The only prerequisite is having git on your machine.

1. **Run the One-Line Installer**

   Open your terminal and paste this:

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

   The installer automatically handles Python, Node.js, and all dependencies. Works on macOS, Linux, and WSL2.

2. **Configure the LLM Backend**

   After installation, edit the config file:

   ```
   # Config file location
   ~/.hermes/config.yaml
   ```

   Enter your model API key (we'll cover how to choose a model shortly).

3. **Launch Hermes**

   ```
   hermes
   ```

   That's it — one word. If you see the welcome message, you're good to go.

> **核心建议**
>
> **If you manage Python with uv**, you can also clone the repo and install via `uv pip install -e ".[all]"`. Same result — use whichever you prefer.

Option 2: Docker (Clean Isolation)

Don't want to install a bunch of dependencies on your machine? Docker is the cleanest choice.

```
docker pull nousresearch/hermes-agent:latest
docker run -v ~/.hermes:/opt/data nousresearch/hermes-agent:latest
```

Key flag: `-v ~/.hermes:/opt/data` maps the container's data volume to your host machine. All of Hermes' state (memory, Skills, config) lives in `/opt/data` — one single directory. Delete and rebuild the container, your data survives.

This is a nice design choice. Unlike some tools that scatter state across various paths, **everything in Hermes lives under `~/.hermes/`**. When it's time to migrate, just pack up that directory.

Option 3: $5 VPS for 24/7 Uptime

If you want Hermes always online, independent of whether your computer is running, **a $5/month VPS is all you need**.

Recommended setups:

VPS Provider	Monthly Cost	Notes
Hetzner CX22	~$4/mo	Best value, European nodes
DigitalOcean Droplet	$5/mo	Singapore/US West nodes
Vultr	$5/mo	Tokyo node, low latency

Pick Ubuntu 22.04 LTS, SSH in, run the install script — identical to local installation. **If you're not running local models, memory usage stays under 500MB**, so a $5 box handles it with room to spare.

Pair it with the Telegram Gateway (covered in S09), and you can message Hermes from your phone anytime while it responds from the VPS. The price of a coffee gets you a 24-hour AI assistant.

> **Serverless option**: Hermes also supports Daytona and Modal as serverless backends. The environment hibernates when idle and wakes automatically on incoming messages, driving inter-session costs toward zero. Great for light usage when you still want to stay reachable. Set `terminal: daytona` or `terminal: modal` in config.yaml.

config.yaml Deep Dive

Regardless of how you install, all core configuration lives in one file: `~/.hermes/config.yaml`.

A minimal working config looks like this:

```
# ~/.hermes/config.yaml
model:
  provider: openrouter        # Model provider
  api_key: sk-or-xxxxx        # Your API key
  model: anthropic/claude-sonnet-4  # Model to use

terminal: local               # Terminal backend (local/docker/ssh/daytona/modal)

gateway:                      # Messaging gateway (optional, details in §09)
  telegram:
    token: YOUR_BOT_TOKEN
  discord:
    token: YOUR_BOT_TOKEN
```

Not many fields — let's walk through each one.

provider and model

Hermes supports a wide range of model sources:

Provider	Recommended Models	Best For
OpenRouter	Claude Sonnet 4 / GPT-4o	200+ models available, flexible switching
Nous Portal	Hermes 3 series	Officially recommended, deeply integrated with the agent
OpenAI	GPT-4o / o3	Direct OpenAI API connection
z.ai / Zhipu	GLM-5	China-friendly option
Ollama	Hermes 3 8B/70B	Fully offline, privacy first

> 注意
>
> **Heads up:** As of April 2026, Anthropic banned third-party tools from accessing Claude through Pro/Max subscriptions. This affects Hermes, OpenClaw, and all other Agent frameworks. You can still use Claude via API keys (pay-as-you-go), but it costs significantly more. Consider OpenRouter or Nous Portal's Hermes 3 series as primary options.

My suggestion: start with OpenRouter so you can switch models freely and get a feel for them. Once you've settled on your go-to model, connect directly to that provider's API to save the middleman fee.

terminal

Six backends determine where Hermes executes code:

- **local**: Runs directly on your machine, simplest option
- **docker**: Runs inside a container, isolated and secure
- **ssh**: Connects to a remote server
- **daytona** / **modal**: Serverless, spins up on demand
- **singularity**: For HPC cluster environments

Most people should just go with local. If you're worried about the security of an agent executing code on your machine, Docker is a solid middle ground.

Common Troubleshooting

> 注意
>
> **Install script hanging?** Check your network connection. The script needs to download dependencies from GitHub and PyPI. If you're in a region with restricted access, you may need to set up a proxy or use mirror sources.

> 注意
>
> **hermes command not found?** The install script adds the command to your PATH, but if your shell config is unusual (e.g., you use fish), you may need to manually run `source ~/.bashrc` or reopen your terminal.

> 核心建议
>
> **Want to verify the install?** Run `hermes --version` — if you see a version number, you're set. The latest version is v0.7.0.

> 注意
>
> **Docker container starts but nothing happens?** Make sure `~/.hermes/config.yaml` exists and has model info configured. The container reads the config mapped in from the host. If config.yaml doesn't exist, Hermes will guide you through creating one on startup.

> 核心建议
>
> **Security note for VPS deployments**: If running on a VPS, consider setting `terminal: docker` so code executes inside a container rather than directly on the host filesystem. A $5 VPS has plenty of power to run containers.

That's it for configuration. Hermes' design philosophy is to keep things minimal — **one config.yaml handles everything**. No scattered environment variables, no layered config files. For an agent this feature-rich, keeping config this simple is impressively restrained.

Next up, let's start talking. Hermes doesn't need you to configure everything before you can use it — install, add your API key, launch, and you're ready for your first conversation.

§08 Your First Conversation: Letting Hermes Get to Know You

First Conversation

Installed and launched — now what? This section walks you through the first conversation. The focus isn't what you say, but what Hermes does behind the scenes.

Starting from a Blank Slate

Type `hermes` in your terminal and you'll see a clean chat interface. No onboarding flow, no setup wizard — just a cursor waiting for you to talk.

Say anything:

```
Hey, I'm HuaShu. I run an AI-focused WeChat blog. I've been using Claude Code a lot and want to try o
```

Hermes will reply normally. But the interesting stuff is happening where you can't see it.

Memory Is Quietly Being Written

After the first exchange, take a look at the `~/.hermes/` directory:

```
~/.hermes/
├── config.yaml # Your configuration
├── state.db # SQLite database (conversation history + FTS5 index)
├── skills/ # Skills directory (still empty)
│ └── bundled/ # Built-in Skills
└── memories/ # Persistent memory (MEMORY.md + USER.md)
```

state.db already has content. That self-introduction you just typed, along with Hermes' reply, has been written into the SQLite database with a FTS5 full-text index. Next time you start Hermes, it won't start from zero — it can search and find this conversation.

This is different from ChatGPT's approach of "appearing to have memory but actually reloading all history every time." **Hermes retrieves on demand, only pulling up history when it's relevant.** Even after months of accumulated conversations, the database won't slow down.

Keep Talking, Trigger Deeper Memory

Chat for a few more rounds. Tell it about your work habits, for example:

```
I'm on macOS, my main editor is Cursor. I write articles in Markdown and prefer angle quotes over sta
```

Hermes will write this into its persistent memory layer. Mapping to the three-layer memory from S04: the conversation content is session memory (what happened), while your preferences and habits are persistent memory (who you are).

You don't need a special command to say "remember this." Hermes judges on its own which information is worth persisting. If you've used Claude Code's auto-memory, the experience feels familiar. But Hermes is more aggressive — it proactively strategizes what to remember.

Triggering the First Skill Creation

Give Hermes a slightly more complex task:

```
Convert this Markdown into WeChat-blog-compatible HTML, preserving bold and code block styling.
```

The first time, Hermes will figure it out as it goes. It might invoke the terminal to run a script, or generate the HTML directly in the conversation.

After it's done, here's where it gets interesting. Check `~/.hermes/skills/` again:

```
~/.hermes/skills/
├── bundled/ # Built-in Skills
└── markdown-to-wechat.md # This is new!
```

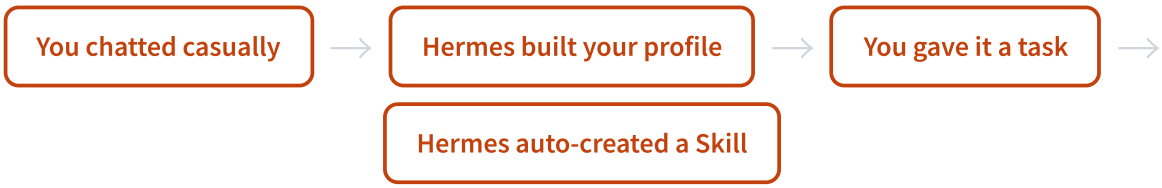
Hermes **automatically distilled the solution into a Skill**. Open that markdown file and you'll see it records the input format, conversion rules, and output requirements. Next time you ask for something similar, it'll invoke this Skill directly instead of figuring it out from scratch.

This is the learning loop from S03 in action. You didn't teach it how to do things — it crystallized what it did into a reusable capability on its own.

> **Skill self-improvement**: If you're not happy with the result, tell it what needs adjusting. Hermes won't just fix the current output — it'll update that Skill file too. Next time, the improved version runs automatically.

You Don't Need to Configure — Just Use It

Let's recap what just happened:



Through this entire process, you didn't write a single line of config, didn't edit a single file, didn't set a single rule. That's a completely different experience from Claude Code asking you to hand-write CLAUDE.md, or OpenClaw requiring you to configure yaml.

Of course, you absolutely can edit Skill files manually (covered in S10), but you really don't need to when starting out. Just use it — Hermes will grow into the shape that fits you.

This is also what makes Hermes most distinctive. Other agent tools need you to figure out what you want, how to configure it, and how to constrain it upfront. Hermes flips this around: **you start using it first, and it forms its own structure through the process of being used.**

Next section — let's take Hermes beyond the terminal and onto your phone.

§09 Multi-Platform Access: Reach It Anywhere

Multi-Platform Access

Hermes doesn't just live in the terminal. Set up a Telegram Bot and reach it from your phone anytime. Add Discord and Slack, and your team can use it too. The key: all platforms share the same brain.

Gateway: One Process, All Platforms

Hermes' multi-platform access is powered by the Messaging Gateway module. Instead of writing separate code for each platform, it uses **a single unified gateway process** that listens to all configured platforms simultaneously.

The architecture looks like this:

Underneath the Gateway sits the same Hermes Agent instance, the same memory, the same set of Skills. A message from Telegram and a command from the CLI are indistinguishable to Hermes.

Telegram Bot Setup (The Recommended Entry Point)

Why Telegram? It's **the simplest to set up and the best mobile experience**. Creating a Bot requires no approval process — you get a Token instantly.

1. **Create a Telegram Bot**

   Find `@BotFather` in Telegram, send `/newbot`, and follow the prompts to name it. BotFather will give you a Token that looks like this:

   ```
   123456789:ABCdefGhIJKlmNoPQRsTUVwxyz
   ```

2. **Add It to config.yaml**

   ```
   # ~/.hermes/config.yaml
   gateway:
     telegram:
       token: 123456789:ABCdefGhIJKlmNoPQRsTUVwxyz
   ```

3 Launch Hermes

```
hermes
```

Hermes automatically connects to the Telegram Gateway on startup. Send your Bot a message in Telegram and it'll respond.

Three steps, under two minutes total. If Hermes is running on a VPS (the $5 setup from S07), this gives you **a 24-hour online, always-reachable, persistent-memory personal AI assistant**. $5/month plus model API costs.

> **核心建议**
>
> **Telegram also supports voice messages.** Send a voice memo and Hermes will automatically transcribe it to text before processing. Think of something during your commute? Just say it — no typing needed.

Discord and Slack Integration

The process is similar to Telegram, with differences mainly in how you obtain the Token.

Discord

Go to the Discord Developer Portal, create an Application, and grab the Token from the Bot page:

```
gateway:
  discord:
    token: YOUR_DISCORD_BOT_TOKEN
```

Invite the Bot to your Server and it can respond in channels. **Great for team collaboration** — have Hermes help review code in the dev channel, or summarize data in the operations channel.

Slack

```
gateway:
  slack:
    token: xoxb-YOUR-SLACK-BOT-TOKEN
```

You'll need to create an App in the Slack App management page and install it to your Workspace. Permission setup is a bit more involved than Telegram, but **it's the more enterprise-appropriate option** that IT departments are comfortable with.

> More platforms available: Hermes also supports WhatsApp, Signal, Email, SMS (Twilio), Home Assistant, Mattermost, Matrix, DingTalk, Feishu/Lark, WeCom, and Open WebUI — 14 platforms in total. Full list on the official docs' Messaging page. Most just need the corresponding Token added to config.yaml.

Cross-Platform Conversation Continuity

This is the most practical feature of Hermes' multi-platform design.

Say you're on your morning commute and message Hermes via Telegram:

```
Research Hermes Agent deployment options and put together a doc for me.
```

Hermes starts working, storing the research results in memory. You get to the office and open your terminal:

```
How's that research coming? Show me what you've got.
```

Hermes knows exactly what you're talking about. **It doesn't distinguish which platform a message came from** — all platforms share the same Agent instance and the same memory. What you said on Telegram can be continued in the CLI. What was discussed in a Discord channel can be referenced from Slack.

This is nothing like having different ChatGPT windows on different devices where you have to re-explain the context every time. **Hermes has one brain, no matter which door you walk through.**

A Practical Deployment Setup

Pulling together the previous sections, a typical deployment looks like this:

```
$5 VPS (Ubuntu 22.04)
├──── Hermes Agent Core
├──── Messaging Gateway
|  ├──── Telegram Bot (reachable from your phone)
|  ├──── Discord Bot (team collaboration)
|  └──── Slack App (enterprise use)
├──── ~/.hermes/
|  ├──── state.db (all conversation history)
|  ├──── skills/ (capabilities that accumulate automatically)
|  └──── config.yaml (one file, everything configured)
└──── Model calls → OpenRouter API
```

Total cost: VPS $5/month + model API fees (light usage runs $2-5/month). Less than a fancy cup of coffee per month for an AI assistant with memory, real capabilities, and 24/7 availability.

> **核心建议**
>
> **On model API costs**: If you want to cut costs further, you can run open-source models on the VPS via Ollama (like Hermes 3 13B). A $5 VPS might not have enough RAM for large models, but a $10-15/month VPS can handle 13B, and after that model calls are completely free.

Automated Scheduling

Beyond passively responding to messages, the Gateway also supports cron scheduling. You can have Hermes automatically send a news briefing every morning at 8am, or automatically summarize the week's GitHub commits every Friday afternoon.

Scheduled task results get pushed through the Gateway to whatever platform you specify. **It doesn't just wait for you to talk — it can work proactively.**

At this point, you have a working Hermes setup: installed locally or on a VPS, model configured, reachable from your phone. Next up: how to manually create and optimize Skills (S10), and how to integrate more tools via MCP (S11), to make this assistant even more powerful.

§10 Custom Skills: Teaching Hermes New Tricks

Custom Skills: Teaching Hermes New Tricks

Hermes's learning loop creates Skills automatically, but you can also teach it manually. This section covers how to write Skills, install them from the community, and port your Claude Code Skills over.

What Exactly Is a Skill

In Hermes, **a Skill is just a markdown file**. No framework to learn, no API to call -- just text that tells Hermes what to do in a specific scenario.

The idea is the same as CLAUDE.md: **define behavior in natural language.** The difference is that CLAUDE.md applies globally, while Skills activate on demand. Ask Hermes to write your weekly report, and it loads the "weekly report" Skill; ask it to do a code review, and it loads a different one.

Each Skill lives in its own folder under `~/.hermes/skills/`, with a `SKILL.md` file as the entry point. A Skill folder can also contain `references/`, `templates/`, and `scripts/` subdirectories for supporting files.

Creating a Skill by Hand

Let's write a Skill that's actually useful: making Hermes enforce a consistent Git commit message format.

Create a folder called `git-commit-style/` in `~/.hermes/skills/`, and inside it create a `SKILL.md` file:

```
---
name: git-commit-style
description: Enforce a consistent Git commit message format
version: "1.0.0"
---

# Git Commit Style

## Trigger
Activate when the user asks me to commit code, write a commit message, or review commit history.

## Rules

### Commit Message Format
- First line: type(scope): summary (50 chars max)
- Blank line
- Body: explain WHY, not WHAT

### Type Enum
- feat: new feature
- fix: bug fix
- refactor: restructure (no behavior change)
- docs: documentation
- test: tests
- chore: build/toolchain

### Constraints
- Body in plain language, types in English
- Don't write "modified XX file" -- that's noise
- One commit, one thing

## Example
```
feat(auth): add WeChat QR code login

Previously users could only log in with a phone number, which meant WeChat users
had to bind a phone first. Now they scan a QR code and they're in -- unbound users
get an account created automatically.
```
```

Save it, and the next time you say "commit these changes," Hermes follows this format. No extra setup needed.

> 核心建议
>
> **Skill triggering is automatic.** You don't need to say "use XX Skill" -- Hermes matches the most relevant Skill to your request on its own. Under the hood, it uses FTS5 full-text search plus semantic understanding.

Anatomy of a Good Skill

After writing dozens of Skills, I've found they share a common structure:

Section	Purpose	Required?
Title	Lets Hermes quickly identify what the Skill does	Yes
Trigger	When to activate this Skill	Strongly recommended
Rules	Concrete steps, constraints, formats	Yes
Example	A complete input-to-output example	Strongly recommended
Don'ts	Explicit boundaries to prevent drift	Optional

The more specific your trigger, the better the hit rate. "When the user mentions code" is too vague; "When the user asks me to commit code, write a commit message, or review commit history" is much better.

Installing Community Skills from the Skills Hub

Hermes has a built-in Skills Hub where community developers share ready-made Skills. Installing them is straightforward:

1. **Browse Available Skills**

   Just ask Hermes in conversation: "What community Skills are available?" It'll list popular Skills from the Hub, grouped by category. You can also narrow it down: "Any Python-related Skills?"

2. **Install**

   Found one you like? Tell Hermes "Install XX Skill." It downloads the Skill folder to `~/.hermes/skills/` and it takes effect immediately. No restart needed.

3. **Customize**

   An installed Skill is just a folder with a SKILL.md file -- open it and edit away. Community Skills are a starting point; making them yours is the goal.

Hermes ships with 40+ bundled Skills covering MLOps, GitHub workflows, research assistance, and more. They're built in -- no extra installation required.

Debugging Skills

You wrote a Skill -- how do you know it's actually firing?

Just ask. Say "What Skills do you have loaded right now?" and Hermes tells you which Skills are active. If the one you expected isn't there, your trigger is probably too narrow.

Check the logs. Log files record which Skills matched each request, why they were selected, and why others were skipped. Logs live in `~/.hermes/logs/`.

Test incrementally. Don't jump straight to edge cases. Start with the simplest request to confirm the trigger and basic behavior, then try boundary conditions.

> **注意**
>
> Skills can conflict. If two Skills have overlapping triggers, Hermes picks the one with the higher match score -- but the result may not be what you expect. If behavior seems off, check for Skill conflicts first.

Hands-On: Porting a Claude Code Skill to Hermes

Here's a very practical need: you've built up a collection of great Skills in Claude Code and don't want to start from scratch after switching to Hermes. **The agentskills.io standard makes Skills portable across Agents**, so migration cost is low.

Let me walk through a real example. I had a "WeChat blog proofreading" Skill in Claude Code -- its core logic runs three proofreading passes (facts, style, details). The SKILL.md looked roughly like this:

```
---
name: proofreading
description: Three-pass proofreading for articles
version: "1.0.0"
---

# Article Proofreading

## Trigger
Activate when the user mentions "proofread," "reduce AI tone," "too AI-like," or "polish."

## Proofreading Flow
### Pass 1: Fact Check
- Verify all data, dates, product names
- Flag anything uncertain

### Pass 2: Style Check
- Remove AI-favorite phrases (firstly/secondly/in conclusion)
- Break up AI sentence patterns
- Replace formal vocabulary with conversational language

### Pass 3: Detail Polish
- Keep sentences to 15-25 words
- Keep paragraphs to 3-5 lines on a phone screen
- Bold ~10 key sentences for scanability
```

Copy this file to `~/.hermes/skills/proofreading/SKILL.md`, and Hermes can use it right away. No format changes, no API adapters -- everyone uses markdown with the same semantic structure.

If your Skill references tools specific to Claude Code (like a particular MCP Server), you'll need to swap those for Hermes equivalents. But **the core logic, triggers, and rules are all portable**.

Why agentskills.io matters: Skills are no longer locked to a single Agent. Skills you built in Claude Code, Cursor, or Gemini CLI work in Hermes. And vice versa. You can switch Agents without worrying about migration cost.

Next up: MCP integration. Skills teach Hermes how to do things; MCP lets it connect to external tools. Put them together, and the range of scenarios you can cover gets very wide.

§11 MCP Integration: Connecting Your Tool Stack

MCP Integration: Connecting Your Tool Stack

Hermes's 40+ built-in tools are already quite capable, but real-world workflows go far beyond that. MCP lets Hermes plug into GitHub, databases, Slack, Jira, and thousands of other external services -- without writing a single line of adapter code.

What MCP Is and Why It Matters

MCP stands for Model Context Protocol, an open standard proposed by Anthropic in late 2024. Think of it as **the USB port of the AI tool world**: as long as an MCP Server implements this protocol, any MCP-compatible Agent can call the tools it provides.

For Hermes, MCP means **no need to build a custom tool for every external service**. Want to connect to GitHub? Install a GitHub MCP Server. Need to query a database? Install a database MCP Server. The community already has thousands of ready-made Servers.

Two Connection Modes: stdio and HTTP

Hermes supports two ways to connect to MCP Servers, depending on where the Server runs.

Mode	Server Location	Best For	Performance
stdio	Local subprocess	Local tools, file system, databases	Fast, no network overhead
HTTP (StreamableHTTP)	Remote server	Cloud services, shared team Servers	Depends on network

stdio is enough for most cases. The MCP Server runs as a Hermes subprocess, communicating over stdin/stdout -- fast and simple to set up. HTTP is for when the Server needs to be independently deployed or shared across multiple Agents.

Configuration goes in the `mcp_servers` section of `config.yaml`:

```
# stdio mode
mcp_servers:
  github:
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-github"]
    env:
      GITHUB_PERSONAL_ACCESS_TOKEN: "ghp_xxxxx"

# HTTP mode
mcp_servers:
  remote-tools:
    url: "https://your-server.com/mcp"
    transport: "streamable-http"
```

Hands-On: Connecting GitHub MCP

GitHub is one of the most common MCP integrations. Once connected, Hermes can create Issues, open PRs, review code, and manage project boards directly.

1. **Create a GitHub Token**

   Go to GitHub Settings -> Developer settings -> Personal access tokens and generate a token. At minimum, check `repo` and `read:org`. If you want to manage Issues and PRs, enable write access for `issues` and `pull_requests` too.

2. **Configure config.yaml**

   Add the GitHub MCP Server to `config.yaml`:

   ```
   mcp_servers:
     github:
       command: "npx"
       args: ["-y", "@modelcontextprotocol/server-github"]
       env:
         GITHUB_PERSONAL_ACCESS_TOKEN: "ghp_your_token_here"
   ```

   Pro tip: store the token in an environment variable instead of hardcoding it. You can reference it with `${GITHUB_TOKEN}`.

3. **Restart Hermes and Verify**

   After restarting, ask Hermes "List my GitHub repos" or "Show recent Issues in the XX repo." If it returns the right info, you're connected.

4 **Daily Use**

Once connected, you can operate GitHub with natural language. For example:

```
"Create an Issue in the alchaincyf/my-app repo titled 'Fix login page redirect bug'"
"Look at this PR's changes and do a code review"
"What new Issues were opened this past week? Group them by label"
```

The GitHub MCP provides tools for repo management, Issue operations, PR reviews, code search, branch management, and more. **You don't need to remember tool names** -- just describe what you need in plain language and Hermes picks the right tool.

Hands-On: Connecting a Database

Databases are another high-frequency use case. Once connected, Hermes can query data, generate reports, and analyze trends -- no hand-written SQL required.

Using PostgreSQL as an example:

```
mcp_servers:
  postgres:
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-postgres"]
    env:
      POSTGRES_CONNECTION_STRING: "postgresql://user:pass@localhost:5432/mydb"
```

Once configured, you can ask Hermes "How many users signed up this month?" or "Show me the daily order revenue trend for the last 30 days" -- it generates SQL, runs it, and returns the results.

> **注意**
> Database MCP has read-write access by default. If you only want Hermes to query without modifying data, connect with a read-only database account. This is especially important for production databases.

SQLite and MySQL have their own MCP Servers too. Configuration is nearly identical -- just swap the Server package name and connection string.

Per-Server Tool Filtering

Once you connect multiple MCP Servers, the available tool list grows fast. A single GitHub Server exposes a dozen-plus tools; add databases, file systems, and Slack, and you could have fifty to a hundred.

Too many tools degrades Agent decision quality. Matching accuracy with 100 tools is lower than with 20. And some tools you don't want the Agent touching at all.

Hermes supports per-server tool filtering -- specify in the config which tools each Server should expose:

```
mcp_servers:
  github:
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-github"]
    env:
      GITHUB_PERSONAL_ACCESS_TOKEN: "ghp_xxxxx"
    allowed_tools:
      - "list_issues"
      - "create_issue"
      - "get_pull_request"
      - "create_pull_request_review"
```

This way, even though the GitHub MCP Server offers high-privilege tools like deleting repos or changing settings, Hermes can't use them. **The principle of least privilege matters more in the Agent era than ever before.**

When to Use MCP vs. Native Tools

Hermes has 40+ built-in tools, and MCP opens up thousands more. How do you choose?

推荐

Use native tools for:
Terminal commands, file operations, web search, image generation, memory management, sub-Agent delegation. These built-in tools are deeply optimized and tightly integrated with the learning loop and memory system -- **fast response, predictable behavior**.

不推荐

Use MCP for:
GitHub, databases, Slack, Jira, Google Drive, and other external services. These require specific API protocols, making MCP the right choice. You could use the terminal to run CLIs instead, but **MCP provides structured input/output, which means higher accuracy**.

A simple rule of thumb: **if Hermes already has the capability built in, use built-in; if you need to interact with an external service, use MCP.**

Some scenarios work either way -- like Git operations. The terminal tool can run git commands directly, and the GitHub MCP can operate on repos too. **The difference**: terminal works locally, great for everyday commits and pushes in your current repo; GitHub MCP works through the API, better for cross-repo management, batch Issue operations, and PR reviews that need platform-level capabilities.

Practical advice: Don't connect a dozen MCP Servers on day one. Start with one or two you use most (GitHub, database), get comfortable, then add more. Every additional Server expands the tool selection space and lengthens the decision path.

MCP + Skills: The Combo

MCP solves "what can I connect to," Skills solve "how to use it." They work better together.

Example: you connect the GitHub MCP and create a "Code Review" Skill. The Skill defines review criteria (naming conventions, error handling, test coverage), while MCP provides the ability to read PR diffs. Combined, Hermes can automatically review code against your standards.

Another example: the database MCP lets Hermes run SQL, and a "Weekly Report" Skill defines the report format and key metrics. Pair that with a Friday afternoon cron job, and Hermes automatically queries the data, generates the report, and posts it to Slack. **MCP, Skills, and native tools working together -- that's where the real power is.**

At this point, you know all of Hermes's core capabilities. The next Part gets into real-world scenarios, showing what these capabilities look like when combined.

§12 Personal Knowledge Assistant: The Power of Cross-Session Memory

Personal Knowledge Assistant: The Power of Cross-Session Memory

How big is the gap between an AI assistant that actually remembers things and one you have to re-introduce yourself to every day? Let's find out with a project that spans three weeks.

Three Weeks of Research, Starting from Scratch Every Time

Say you're researching a new domain. You're an indie developer trying to figure out AI Agent deployment options for 2026: local, cloud, Serverless -- what are the gotchas for each.

Week one, you ask three or four questions, covering Docker deployment memory usage, VPS pricing, Daytona's free tier limitations. This information ends up scattered across different conversations.

Week two, you want to dig deeper into Serverless. You open ChatGPT or Claude, and what's the first thing you do? **Re-explain what you're working on.**

"I'm researching AI Agent deployment options. Last week I looked at Docker and VPS, now I want to understand Serverless. I found that Daytona has a free tier but with limitations..."

Every new conversation costs 3-5 minutes of context-setting. This isn't an AI capability problem -- **it's an architecture problem**: traditional AI has no cross-session memory, so every conversation starts from a blank slate.

What Hermes Remembers

Same scenario, but with Hermes.

After week one's conversations, Hermes's three memory layers have recorded different things:

Memory Layer	What It Records	Purpose
Session memory (SQLite + FTS5)	What you asked, what it looked up, raw conversation text	Precise retrieval when details are needed
Persistent memory	"User is researching AI Agent deployment, ruled out option X, prefers low cost"	Auto-loads context for the next conversation
Skill memory	"Research tasks: list dimensions first -> dig into each -> summarize per round"	Methodology reuse

Week two, you open Hermes and just say "Let's continue with the Serverless options." No need to re-explain anything -- persistent memory already knows what you're doing. **It might even proactively remind you: "Last**

week you mentioned Daytona's free tier had limitations -- want to check if the policy has been updated?"

This isn't magic; it's FTS5 full-text search at work. Hermes doesn't stuff all of last week's conversations into the context -- that would waste tokens. Instead, it retrieves the most relevant historical snippets based on your current question.

Retrieval vs. Full-Context Loading

This design choice is worth unpacking.

Many people assume "memory" means cramming all conversation history into the prompt. Claude Code's auto-memory does work that way: key info gets written into MEMORY.md, which is fully loaded at startup. Fine for coding tasks.

But the knowledge assistant use case is different. Three weeks of research conversations could be tens of thousands of words. Loading everything causes two problems: **token costs explode**, and **information overload actually degrades answer quality**. Large models have uneven attention distribution across very long contexts, and key information gets drowned out.

Hermes's approach: **persistent memory stores summaries (a few hundred words), and when details are needed, FTS5 searches the raw conversations, injecting only the most relevant snippets into context.** It's like carrying a one-page cheat sheet and going back to the filing cabinet only when you need specifics.

推荐

Hermes approach: Persistent memory (summaries) + on-demand retrieval (FTS5). Token consumption stays controlled, information stays precise.

不推荐

Full-context approach: Stuff all history into the prompt. Works short-term, but after three weeks the prompt is maxed out and costs double.

Honcho: It Knows You Better Than You Know Yourself

If you enable Honcho user modeling, the depth of memory goes up another level.

Honcho doesn't just record what you said -- **it infers things you didn't say**. For instance, if you consistently pick the cheapest option across three research sessions, Honcho infers "this user is cost-sensitive." Next time it recommends something, pricing info gets surfaced first.

This dialectical modeling continuously derives insights about you -- from technical skill level and preference patterns to communication habits -- building a representation that deepens over time. **The longer you use it, the more accurately it understands you.**

HuaShu's experience: After two weeks, Hermes started automatically giving me shorter, punchier replies -- because it noticed I tend to want conclusions rather than lengthy analysis. This adaptation is gradual; you don't have to explicitly configure anything.

The Experience Gap vs. Traditional AI

Here's an analogy to sum it up: **Traditional AI is like a hotel front desk -- different person every day, and you re-introduce yourself each time**. Hermes is like your personal assistant who's known you for three months, knows you drink black coffee, hates writing weekly reports, and what project you're researching right now.

Specifically for the knowledge assistant use case, the gap shows up in three ways:

1. **Zero Startup Cost**

   No context-setting every time. Say "continue" and you continue.

2. **Research Has Continuity**

   Options you ruled out last week don't get re-recommended. Information you already confirmed doesn't get re-verified. The whole research effort feels like a single thread instead of a pile of disconnected dots.

3. **Methodology Compounds**

   The approach Hermes develops while helping you with your first research project (list dimensions, dig into each, summarize per round) gets saved as a Skill. **Your second research project reuses it automatically -- no re-teaching needed.**

> **核心建议**
>
> Cross-session memory isn't a silver bullet. If the task is one-off -- like "translate this paragraph" -- Hermes's memory system offers zero advantage. **The value of memory scales with task duration.** The longer and more context-heavy the task, the bigger the efficiency gain from memory.

§13 Dev Automation: From Code Review to Deployment

Dev Automation: From Code Review to Deployment

Claude Code writes the code, Hermes watches the pipeline. These two tools aren't either-or -- they each own a different part of the workflow.

A Developer's Morning

9 AM, you open your laptop. Three Telegram messages pop up. Not from a coworker -- **from Hermes**:

"Last night at 11:17 PM, a PR was merged into main -- 387 new lines of code. I reviewed it and found two issues: the auth module's token expiration logic doesn't handle an edge case, and test coverage dropped from 82% to 76%. Full report saved to the project Skill."

"At 2:40 AM, the CI pipeline ran a regression test suite. 3 cases failed. 2 were introduced by yesterday's PR, 1 is a known flaky test."

"Today's daily standup notes are drafted, based on yesterday's 4 commits and 2 PRs. Needs your sign-off before sending."

This isn't hypothetical. **Hermes's cron scheduling + GitHub MCP + memory system lets it keep working while you sleep.**

Automated Code Review

Traditional code review has two pain points: **latency** -- you open a PR and wait for a reviewer to find time; and **inconsistency** -- different reviewers focus on different things, one checks architecture while another checks formatting.

Setting up automated code review with Hermes is straightforward:

1. **Connect GitHub MCP**
   Configure the GitHub MCP Server in config.yaml, and Hermes can read your repo's PRs, diffs, and issues.

2. **Set Up cron Scheduling**
   Tell Hermes in plain language: "Check main branch for new PRs every 6 hours and do a code review." It creates the cron job automatically.

3. **Define Review Standards**
   Write your coding standards as a Skill. For example: "Functions must be under 50 lines," "Error handling must use custom types," "All API endpoints must have tests." Hermes reviews every PR against these standards.

Step three is the key. Review standards are a Skill, and **they evolve automatically based on your feedback**. You flag an issue Hermes missed, and next time it watches for that pattern. **Traditional lint rules are static; Hermes's review standards are alive.**

Test Generation and Execution

There's a fundamental difference between how Hermes does testing and how Claude Code does it: with Claude Code, you say "write tests for this function," it writes them, and you verify. **Hermes discovers which functions lack tests on its own, writes the tests, runs them, fixes failures, and delivers a report.**

This capability comes from combining several tools:

Tool	Role
file	Scans the codebase to find modules without tests
code_execution	Runs tests in a sandbox
terminal	Generates coverage reports
memory	Remembers which modules have been reviewed and which tests are flaky

Pair it with cron for weekly coverage checks. Every Monday morning it runs an automatic scan, and if coverage drops below the threshold, it sends a notification.

Automated Daily and Weekly Reports

This feature sounds simple, but it's addictive once you start using it.

Hermes pulls the day's commit history, PR statuses, and issue changes through GitHub MCP, combines that with conversation memory from your discussions, and **generates a daily report so accurate you can approve it without even thinking back through your day**.

Weekly reports are even more interesting. Thanks to cross-session memory, Hermes can see the full arc of the week: what bug you fixed Monday, why you changed the architecture plan Wednesday, why Friday's PR got rejected. **It's not just a list of commits -- it's a narrative with cause and effect.**

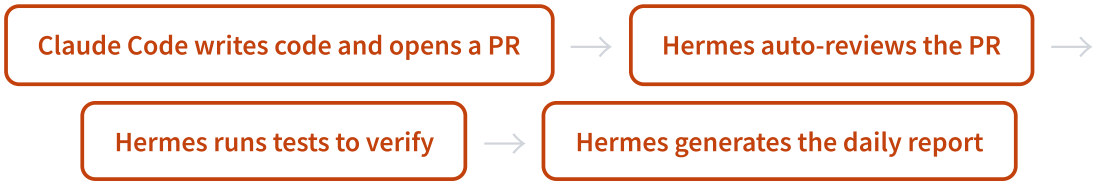
> **Pro tip:** When Hermes generates the daily report, have it also update a project Skill that logs the week's technical decisions. When it writes the weekly report, it can reference those decisions directly instead of guessing what you were thinking from commit messages.

How Claude Code and Hermes Divide the Work

These two tools aren't competing. They're good at completely different things:

Dimension	Claude Code	Hermes Agent
Interaction mode	You're right there, real-time conversation	Runs in the background, reports on schedule
Strengths	Writing code, refactoring, debugging	Monitoring, auditing, summarizing, scheduling
Time horizon	Completed within a single session	Runs continuously across days and weeks
Trigger	You initiate it	cron or event-driven

In one sentence: Claude Code is the craftsman, Hermes is the butler. The craftsman builds things; the butler makes sure everything stays on track. You wouldn't ask the butler to lay bricks, and you wouldn't ask the craftsman to do the night watch.



Once this pipeline is running, your focus shifts from "write code + review code + run tests + write reports" to "write code + confirm results." **Everything in between is automated.**

> 注意
>
> **Hermes's code review is a supplement, not a replacement.** It's great at catching pattern-based issues (inconsistent naming, missing tests, overlooked edge cases), but architectural decisions still need a human eye. Don't skip manual review just because you have automation -- at minimum, keep a human in the loop for core modules.

§14 Content Creation: From Research to Final Draft

Content Creation: From Research to Final Draft

Using AI to write articles isn't new. Having AI remember your writing style, research habits, even reader feedback, and continuously evolve into your personal editor — that's how Hermes does content.

The Life Cycle of an Article

I've written over 100 WeChat blog articles with Claude Code. The workflow is well-established: give it a topic, it searches, drafts, proofreads, and generates images. The whole process takes about 2 to 3 hours.

But one problem never went away: **every time I started a new session, it forgot the lessons from the last article.**

Last week, I told it to stop using a certain cliched phrase. It complied. This week, writing a new article, the same phrase was back. I added a banned-words list to CLAUDE.md, which helped — but CLAUDE.md is a static file. It doesn't update itself.

This is exactly where Hermes does content creation differently.

Ongoing Content Projects

Say you're working on a content series — five consecutive articles about AI Agents. The traditional way, each article is independent: you re-explain the reader persona, re-define the style, re-state which topics were already covered.

With Hermes, the series works completely differently:

After the first article is done, Hermes's memory system records several things: the series positioning, target audience, your editing preferences (like how you broke all its long sentences into short ones), and which technical concepts were already explained in article one.

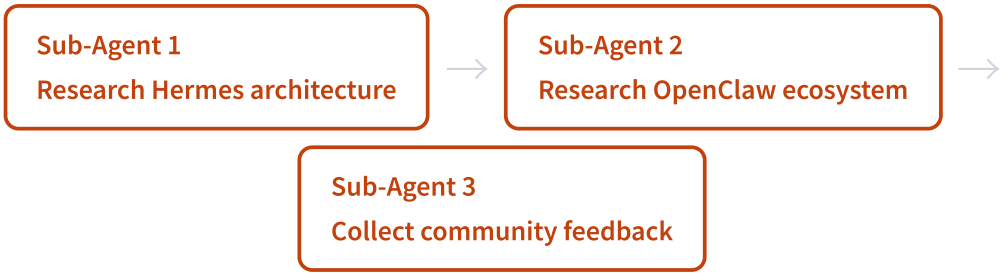
When you start the second article, you just say "write the next one in this series, topic is XXX." **It knows what style to use, which concepts need no re-explanation, and what you were unhappy about last time.**

By the fifth article, its understanding of your writing preferences is remarkably precise. Not because you configured anything — **because it learned from your feedback on its own**.

Parallel Research with Sub-Agents

For a deep technical article, research often takes more time than writing. The traditional approach is linear: search topic A, organize it, search topic B, organize that, then topic C.

Hermes's delegate_task tool makes this parallel. Say you're writing an AI Agent comparison piece — you can dispatch three sub-agents simultaneously:



Three sub-agents working simultaneously, each returning their research results. The main agent consolidates everything into structured research material. **Research that used to take over an hour? Done in 20 minutes.**

Each sub-agent can be assigned a different toolset. The one researching architecture needs web search and browser; the one collecting community feedback might only need web search. **Restricted toolsets aren't just a safety design — they're an efficiency design** — sub-agents don't get distracted by having too many tools.

Skills That Accumulate Writing Style

This is the most valuable part of how Hermes handles content creation.

Traditional AI writing tools control style through prompts. You write in the prompt: "conversational tone, sentences under 20 words, avoid AI-typical phrases." You have to write this every time, or maintain a very long system prompt.

Hermes stores style rules as a Skill. Initially, this Skill might have just a few simple rules: don't use cliched summary phrases, keep paragraphs to 3-5 lines, use casual words like "I think" or "actually."

Here's the key: this Skill self-improves.

When you proofread a draft and make edits, Hermes observes and learns. If you change a stiff phrasing to something more natural three times in a row, it adds a rule to the Skill: "avoid formal verb constructions." If you delete a forced inspirational ending it wrote, it learns "don't force an uplifting conclusion."

A month later, this writing style Skill has accumulated dozens of rules, all from your real feedback. **It becomes your personal editorial handbook — and it maintains itself automatically.**

> **核心建议**
>
> Hermes's Skill self-improvement isn't a black box. Every Skill update is visible as a diff in the `~/.hermes/skills/` directory. If a rule drifts in the wrong direction, you can manually correct it, and Hermes incorporates your correction into its learning.

How Is This Different from Writing with Claude Code

I'm using both tools for content creation simultaneously. The difference is clear:

Dimension	Claude Code	Hermes Agent
Best for	Standalone articles, one-off tasks	Content series, ongoing projects
Style control	CLAUDE.md + manual maintenance	Skills that auto-accumulate and evolve
Research efficiency	Linear search	Parallel research via sub-agents
Context continuity	Relies on auto-memory, limited capacity	Three-layer memory, on-demand retrieval
Learning ability	Doesn't learn; rules are manually written	Learns automatically from your feedback

This isn't to say Claude Code is worse. For a single article, Claude Code's interactive experience is smoother — you see edits in real time and give feedback on the fly. **Hermes's advantage is in the long game.** Write two articles a week for three months, and by article ten Hermes is dramatically better than article one. Claude Code's article ten performs about the same as article one.

> **HuaShu's approach:** I use Claude Code for one-off sponsored articles, because the interaction is fast and brand feedback can be incorporated instantly. For content series and personal columns, I use Hermes, letting its writing Skill grow over time. The two complement each other — no conflict.

§15 Multi-Agent Orchestration: Running Three Horses at Once

Multi-Agent Orchestration: Running Three Horses at Once

When one agent isn't enough, put three to work in parallel. delegate_task is Hermes's most powerful tool — and the easiest to misuse.

Why You Need Multiple Agents

A single agent's ceiling is determined by its context window and toolset. When a task is complex enough, a solo agent hits two walls.

Context explosion. One agent handling research, coding, and testing at the same time — all that information crammed into a single context, interfering with each other. Web page content from research eats up tokens, leaving insufficient room for code reasoning.

Time bottleneck. Three tasks running sequentially at 30 minutes each means 90 minutes total. Run them in parallel, and total time equals the slowest one.

Hermes's delegate_task tool exists to solve both problems. It can launch up to 3 sub-agents simultaneously, each with its own independent context and toolset.

delegate_task in Detail

delegate_task isn't just "spawn a thread." It does several critical things.

Feature	Description
Independent context	Each sub-agent has its own conversation history, preventing pollution of the main agent's context
Restricted toolset	You specify which toolsets a sub-agent can use. Additionally, delegate_task, clarify, memory, send_message, and execute_code are always blocked for sub-agents
Isolated terminal sessions	Each sub-agent has its own terminal, no interference
Max 3 concurrent	Hard-coded limit to prevent resource exhaustion
Result relay	When a sub-agent finishes, results are returned to the main agent for consolidation

The 3-concurrent limit is a deliberate design choice. Nous Research found in testing that beyond 3 sub-agents, the main agent's consolidation quality drops sharply. **It's not a compute limitation — it's an attention dispersion**

problem when LLMs try to integrate too many independent information sources.

A Real Example: Competitive Analysis Report

Say you need to write a competitive analysis of AI coding tools, covering three products: Claude Code, Cursor, and Hermes Agent. The traditional approach is researching them one by one, then manually integrating everything.

With delegate_task, the workflow becomes:

Zooming into the parallel execution:

1. **Main agent defines the task template**
   "Research [product name] along these dimensions: positioning, core features, technical architecture, pricing, community size, pros and cons. Output as a markdown table."

2. **Dispatch three sub-agents**
   Sub-agent A researches Claude Code, sub-agent B researches Cursor, sub-agent C researches Hermes. **Each sub-agent gets only web and browser — two tools**, no file or terminal needed.

3. **Main agent consolidates**
   After all three sub-agents return their research, the main agent integrates everything into a comparison report, adding its own judgment and recommendations.

Three research tasks in parallel — total time goes from "A+B+C" to "max(A, B, C)." In practice, **a competitive analysis that normally takes 40 minutes is done in 15**.

Security Design of Restricted Toolsets

The restricted toolset in delegate_task isn't just an efficiency feature — it's a security mechanism.

Imagine this scenario: you send a sub-agent to search for a code snippet online, and it finds one with malicious injection. If that sub-agent also has terminal permissions, it might execute the code. But if it only has web permissions, the search result comes back as text for the main agent to review.

That's the value of restricted toolsets: **the principle of least privilege, applied at the agent level.**

推荐

Good practice: Research sub-agents get only web+browser. Coding sub-agents get only terminal+file+code_execution. Consolidation sub-agents get no external tools at all — they only process text.

不推荐

Bad practice: Every sub-agent gets the full toolset. Convenient, sure, but you lose the isolation that keeps things safe.

Relationship to Anthropic's Three-Agent Architecture

Anthropic's agent design guide proposed a classic three-agent architecture: Planner for planning, Generator for execution, Evaluator for assessment. This pattern has proven effective in many scenarios.

Hermes's delegate_task shares similarities with this architecture, but has key differences:

Dimension	Anthropic Three-Agent	Hermes delegate_task
Role assignment	Fixed roles (plan/execute/evaluate)	Task-driven, flexible roles
Communication	Chain (plan → execute → evaluate)	Star topology (main agent ↔ sub-agents)
Parallelism	Typically sequential	Up to 3 concurrent
Memory	No built-in memory	Main agent maintains full memory

Hermes's model is more flexible. You can use delegate_task to implement Anthropic's three-agent architecture — one sub-agent for planning, one for execution, one for evaluation. But you can also have three sub-agents doing the same type of parallel task (like researching three products simultaneously). The architecture isn't fixed — it depends on how you decompose the task.

Anthropic's three-agent architecture is a mental framework, telling you "complex tasks can be split into planning, execution, and evaluation." Hermes's delegate_task is an execution tool that turns that framework into reality. **One governs design; the other governs implementation.**

核心建议

Multi-agent orchestration is easy to over-engineer. Not every task needs to be split into sub-agents. If a task fits comfortably within a single agent's context window, splitting actually adds communication overhead and consolidation errors. **Only reach for delegate_task when context is insufficient or you need parallel speedup.**

Rule of thumb: If you find yourself writing lengthy consolidation instructions for the main agent to integrate sub-agent results, the task decomposition is probably wrong. Good decomposition makes consolidation simple: each sub-agent's output should be self-contained, uniformly formatted, and directly composable.

§16 Hermes vs OpenClaw vs Claude Code: Not a Choice

Not a Choice — A Combination

These three tools aren't three roads. They're three horses. The question isn't which one to ride — it's figuring out which one hauls cargo, which one covers distance, and which one guards the house.

Three Species

Over the past year, the AI Agent ecosystem has spawned too many tools to count. But the three genuinely worth a serious look, in my opinion, are Claude Code, OpenClaw, and Hermes Agent.

Not because they're the most hyped, but because they represent three fundamentally different design philosophies.

Claude Code is an interactive coding tool. You sit at the terminal, give it requirements, and it writes code, runs tests, commits to git. You're present the whole time, like pair programming with a very capable engineer. The core value is real-time code productivity.

OpenClaw is a "configuration-as-behavior" framework. You define the agent's personality, knowledge, and skills entirely through SOUL.md and Skill files. The config files determine what the agent is. The core value is predictability, auditability, and reproducibility.

Hermes Agent is an autonomous background engine. You deploy it on a server, and it runs 24/7 — remembering, creating Skills, improving itself. The core value is autonomy and self-improvement.

Dimension	Claude Code	OpenClaw	Hermes Agent
Core philosophy	Interactive coding	Configuration as behavior	Autonomous background + self-improvement
Your role	Sitting at the terminal directing	Writing config files to define behavior	Deploy and check in occasionally
Memory mechanism	CLAUDE.md + auto-memory	Multi-layer memory (SOUL.md + Daily Logs + semantic search), transparent and controllable	Three-layer self-improving memory
Skill source	Manually installed	ClawHub 5,700+	Agent-created + community Hub
Run mode	On-demand	On-demand	24/7 background
Deployment	Local CLI (subscription)	Local CLI (free + API costs)	$5 VPS / Docker / Serverless

See the distinction? These three tools aren't even solving the same problem.

Which Tool for Which Scenario

I've used Claude Code for half a year, OpenClaw for several months, and I've been tinkering with Hermes recently. My takeaway: picking a tool isn't about which one is more powerful — it's about which interaction model fits your scenario.

Scenario	Recommended Tool	Why
Building new features, refactoring code	Claude Code	Needs real-time feedback and human judgment
Setting up standardized agents for a team	OpenClaw	SOUL.md is transparent, auditable, reproducible
24/7 code review	Hermes	Cron scheduling + GitHub MCP, runs unattended
Personal knowledge assistant	Hermes	Three-layer memory accumulates across sessions, gets smarter over time
Building a customer support / community bot	Hermes	Native 12+ platform Gateway, multi-channel
Rapid product idea validation	Claude Code	Fast to start, fast to iterate, real-time course correction
Enterprise scenarios needing high control	OpenClaw	Transparent config, predictable behavior
Long-term content creation project	Hermes + Claude Code	Hermes for ongoing research and accumulation, Claude Code for writing

That last row matters. Many scenarios can't be handled by a single tool. In a long-term content project, Hermes handles daily automated information gathering and memory accumulation, while Claude Code sits down and actually writes the piece. Each owns a different leg of the relay.

Convergence or Divergence

Here's an interesting pattern: these three tools are learning from each other's strengths.

Claude Code added auto-memory, moving toward Hermes-style persistent memory. OpenClaw's ClawHub has 5,700+ community Skills; Hermes is building its own Skill Hub. Hermes supports the agentskills.io standard, meaning it can directly use Skills from the Claude Code ecosystem.

Looks like convergence. But I think the underlying divergence is actually widening.

Claude Code is fundamentally about real-time conversation between human and AI. No matter how much memory and automation it adds, the fact that you're sitting there watching it work won't change. Anthropic's business model dictates this: subscription-based, charged by your usage time.

Hermes is fundamentally about AI running autonomously in the background. No matter how many interactive interfaces it adds, its core value is that it keeps working when you're not there. The MIT open-source + self-hosted model dictates this.

OpenClaw sits in the middle. It doesn't emphasize real-time interaction like Claude Code, nor autonomous operation like Hermes. Its unique value is "transparent and controllable." SOUL.md lets you see at a glance exactly what an agent will and won't do. For enterprise compliance scenarios, this property is irreplaceable.

agentskills.io: Why Skill Interoperability Matters

In early 2026, the agentskills.io standard started gaining adoption across multiple tools. Currently 16+ tools support it, including Claude Code, Cursor, OpenAI Codex, Gemini CLI, and Hermes.

What does this mean?

It means a Skill you wrote for Claude Code can be directly used by Hermes. A Skill auto-created inside Hermes can be brought into Claude Code. Skills are no longer tied to a specific tool — they become portable capability units.

The long-term impact of this might be bigger than any individual tool. Because it's saying: **no matter which horse you ride, the saddle is universal.**

The time you invest in writing Skills won't be wasted if you switch tools. Your Skill library is your own asset, not a platform's appendage.

OpenClaw's ClawHub has 5,700+ Skills. If those Skills can be called directly by Hermes through the agentskills.io standard, Hermes's capability boundary expands instantly. Conversely, Skills that Hermes auto-creates and improves can feed back into the broader ecosystem.

Not a Choice — A Combination

Having written three orange books (Claude Code, Harness Engineering, OpenClaw), my biggest takeaway about this space is: **the winner won't be any single tool — it'll be the people who know how to combine tools.**

My own workflow is already a combination. Claude Code handles everything that needs me present: writing articles, writing code, making product decisions. It's my "day shift."

Hermes (or similar autonomous agents) handles what doesn't need me present: monitoring repos, running scheduled research, maintaining knowledge bases. It's my "night shift."

OpenClaw's SOUL.md and Skill system give me a standardized configuration language. Whether Claude Code or Hermes is running underneath, the behavioral constraints are written the same way.

> **HuaShu's take:** Don't "choose" between these three tools. Ask yourself three questions: Which tasks need me watching? Which tasks can run in the background? Which scenarios need transparent auditability? The answers naturally sort the tools into their respective positions.

Competition among agent tools won't converge to a single winner. Just like you don't use a hammer to turn a screw — interactive coding, configuration management, and autonomous operation are three distinct work modes that will coexist long-term.

The truly interesting question isn't "which is better" but "how do we make them collaborate." agentskills.io is already paving that road.

§17 The Boundaries of Self-Improving Agents: How Far Can They Go

The Boundaries of Self-Improving Agents

Hermes's most exciting capability is also its most unsettling one.

Revisiting That Diagram

In the final chapter of the Harness Engineering orange book, I wrote about the diagram Kief Morris drew. Three layers: in the loop, on the loop, out of the loop.

In the loop: reviewing every line of the agent's output. On the loop: not checking outputs, just holding the reins. Out of the loop: you say what you want, the agent handles everything.

My conclusion then was that on the loop is probably the best balance. You're not duplicating effort, but you're still there.

Hermes Agent pushes this discussion to a new place.

Its learning cycle is automatic. It creates Skills on its own, improves Skills on its own, decides what to remember on its own. After you deploy it, it keeps getting stronger. This isn't on the loop anymore. You don't even have to adjust the reins — the reins are growing by themselves.

Is this progress or risk?

Can Skill Self-Improvement Go Out of Control

Let's start with the technical layer. Hermes's Skill self-improvement has several constraints.

Skill files are readable markdown. Not black-box neural network weights — they're text you can open and read. What it changed, you can see in the diff.

Memory data is local. Built on SQLite + FTS5, data sits on your local disk. You can inspect and delete it directly. There's no "the agent secretly learned something you don't know about" situation.

Tool permissions are sandboxed. The agent can't arbitrarily acquire new system permissions; the toolset must be explicitly configured.

Technically speaking, Hermes's self-improvement is controlled and auditable. You can see what it changed, roll it back, delete it.

But technically controlled doesn't mean practically controlled.

The problem is on the human side.

Are you really going to check which Skills the agent modified every day? Are you going to audit its memory database? Probably not. The whole appeal of deploying Hermes is "not having to babysit it." If you had to review its self-improvement results daily, how would that be different from maintaining Skills manually?

This contradiction is fundamental: the value of an autonomous agent lies in not having to watch it, but safety requires you to watch it.

Kief Morris's insight holds true again here: **the difference between in the loop and on the loop becomes most obvious when you're unhappy with the result.** But if the agent self-improved a Skill and you never noticed anything wrong, when would you ever catch it?

Nous Research's Choice

The founding team at Nous Research made a clear choice on this issue: user control first.

They've explicitly stated that models should be "steerable" — users can adjust behavior as needed, unconstrained by corporate content policies.

This isn't just talk. Hermes Agent's MIT license means you own the entire source code. You can audit every step of the learning cycle's logic, modify the thresholds, frequency, and scope of self-improvement, or turn off automatic Skill creation entirely.

Compared to closed-source agents, this transparency gives you a floor: in the worst case, you can see what all the code is doing.

But let me say something a bit uncomfortable.

"You can see the code" and "you've actually read the code" are two very different things. The vast majority of users won't read source code. The vast majority of people deploying Hermes will use default settings. The MIT license gives you the right to audit, but it doesn't guarantee you'll exercise that right.

Open Source vs Closed Source: Different Shapes of Trust

This leads to a bigger question: when it comes to self-improving agents, is open source or closed source more trustworthy?

The intuitive answer is open source. Transparent code, community audits, MIT license.

But reality is more nuanced than intuition.

Claude Code is closed source. You don't know what Anthropic's agent internals look like. But Anthropic has a clear commercial incentive to keep agent behavior predictable: if an agent goes rogue and damages a user's codebase, they lose subscribers. Business pressure is a form of constraint.

Hermes is open source. You can see all the code. But if the agent's self-improvement causes a problem, Nous Research has no commercial obligation to fix it for you. The other side of the MIT license: you bear the consequences.

Two shapes of trust: one is "I trust your business incentives," the other is "I trust my own ability to audit."

For people with technical chops, open source is clearly better — you control everything. For people who don't want to touch code and just want to use the tool, a closed-source commercial service might actually be safer, because someone else is on the hook.

Where Is the Ceiling of Self-Improvement

Back to the core question: how far can self-improving agents go?

My assessment: **the ceiling isn't technical — it's the feedback signal.**

Hermes's self-improvement loop relies on a key assumption: it can judge whether its own improvements are good or bad. It modifies a Skill, the next task goes better — that's positive feedback. But who defines "better"?

If you're there giving feedback, the loop works. You say "that's wrong," it adjusts. This is supervised improvement.

If you're not there, the agent can only use its own evaluation criteria. It thinks the response was faster, more accurate. But "fast" and "accurate" don't equal "correct." Some errors require domain knowledge to catch. The agent doesn't know what it doesn't know.

In the Harness Engineering book, I wrote: Mitchell Hashimoto could write excellent harness for Ghostty because he understood every detail of terminal emulators. A self-improving agent doesn't have Mitchell's domain knowledge. It can optimize execution efficiency, but it can't judge whether the direction is right.

Self-improvement makes agents run faster in a known direction. But the direction itself still needs a human to set.

Questions for You

At this point, I'm not going to wrap things up with a neat conclusion. Because these questions don't have definitive answers, and the answers will keep shifting as technology evolves.

But I want to lay out a few questions worth thinking about over time.

How much autonomous self-improvement are you comfortable with?

Rewriting Skill files? Sure. Auto-creating new Skills? Probably fine. Modifying its own core configuration? Maybe not. Modifying its own learning loop logic? Absolutely not. Where do you draw the line?

Who audits the results of self-improvement?

You do it yourself — how often? The community audits — do you trust the community's judgment? Nobody audits — are you okay with that risk?

Do self-improving agents need a "forgetting" mechanism?

Humans forget, and that's not a bug — it's a feature. Outdated experience fading away is what keeps it from polluting current judgment. Agent memory only grows; patterns learned three months ago might be obsolete. Who tells the agent "this one should be forgotten"?

What does Kief Morris's concern look like in the Hermes context?

Morris worried: if junior developers never touch code details, who will design the harness in the future? In the Hermes context, the question becomes: if the agent designs its own reins, who judges whether the reins are designed correctly?

Maybe the answer is: you always need a human in the loop at some level. Not reviewing every line of code, but understanding what the system is doing and why it's doing it.

Maybe the answer is: we don't know yet.

> **HuaShu's take:** Self-improving agents are the most exciting direction in this space, but their ceiling is determined by the degree of human involvement. A fully hands-off self-improving agent will win on efficiency but lose on direction. The sweet spot might be: let the agent self-improve on the "how," while you own the "what" and the "don't." That's not being lazy — it's a different kind of on the loop.

Hermes Agent: The Complete Guide

Orange Book Series

HuaShu · AI进化论-花生

A practical guide to the Nous Research open-source AI Agent framework

From installation to multi-Agent orchestration

All Orange Books → huasheng.ai

Bilibili · WeChat: 花叔 · X/Twitter · YouTube · Xiaohongshu · Website

Created by HuaShu · v260408 · April 2026