Hermes Kanban

Multi-Agent Profile Collaboration for Hermes Agent

A durable, profile-aware work-queue architecture for coordinating heterogeneous agent workloads across research, operations, and engineering

Hermes & Teknium
Nous Research

April 25, 2026 | Design Spec, Revision 01

Abstract

This document specifies **Hermes Kanban**, a multi-agent collaboration layer for the Hermes Agent framework. It defines a durable, SQLite-backed task board shared across named agent *profiles*, allowing heterogeneous agents—researchers, editors, engineers, persistent digital-twin assistants—to collaborate asynchronously on long-lived work without the lifecycle fragility of in-process subagent swarms.

We compare the proposed architecture against three contemporary systems: Cline Kanban's worktree-per-task board, Paperclip's persistent-agent company orchestrator, and NanoClaw's failed in-process agent-team SDK. We further contextualize the design against Google's April 2026 Gemini Enterprise Agent Platform, which splits the agentic stack between a governance-oriented control plane and a velocity-oriented execution plane. Hermes Kanban deliberately stays at the execution plane, with governance expressible as user-space profiles and plugins rather than kernel features.

The specification covers: a minimal SQLite schema (tasks, links, comments, events), a dumb cron-driven dispatcher with atomic claims, eight canonical collaboration patterns, four user stories spanning research / scheduled operations / persistent digital twins / coding, a first-class orchestrator profile pattern (addressing the common "my orchestrator does the work itself" failure mode), an optional single-column multi-tenant extension that lets one specialist fleet serve many business contexts, and an implementation plan organized as eight independently shippable pull requests. The kernel requires no changes to `run_agent.py`, no new core tools, and no tool-schema expansion on any API call.

Status: DESIGN ONLY. No implementation accompanies this document.
Target audience: Hermes maintainers and contributors; design-review participants.

Contents

1 Motivation

1.1 The multi-agent gap in Hermes today

Hermes currently supports multi-agent workloads through a single primitive: `delegate_task`, a synchronous fork-and-join subagent call. The parent agent constructs a goal string and optional context, spawns a short-lived subagent in an isolated conversation, and blocks until the subagent returns a summary. This primitive is correct for its intended shape—short, self-contained reasoning subtasks whose result is immediately consumed by the parent—but it does not generalize to four increasingly common workload shapes observed in the Hermes user community:

1. **Research triage and synthesis.** Parallel specialist workers producing candidate findings, one or more reviewers selecting or merging, with the human able to correct course mid-flight.
2. **Scheduled recurring workflows.** Daily briefings, weekly reports, hourly inbox triage—all of which accumulate knowledge across runs and must survive individual failures.
3. **Digital-twin / persistent assistant roles.** Named, long-lived agent identities that build up memory of people, preferences, and context over weeks and months.
4. **End-to-end engineering pipelines.** Decompose, implement in parallel, review, iterate, ship—a loop that can span hours and must preserve contributor credit when opening PRs.

None of these shapes map cleanly onto `delegate_task`. Each of them requires: durable state across runs, visibility into work-in-progress, handoffs between differently-skilled agents, and the ability for humans (or peer agents) to interject at any point. A single primitive cannot satisfy both shapes without becoming so general that it satisfies neither.

1.2 What others have built, and what broke

Three contemporary systems illuminate the design space.

Cline Kanban [1] ships a local web application that treats each task as a card with an ephemeral git worktree and a CLI agent assignment. Cards link into dependency chains; when a parent lands in trash (its terminal *done* state), linked children auto-start. The system is agent-agnostic (Claude Code, Codex, Cline, Droid). It deliberately rejects server infrastructure: no account, no shared state between cards other than the board itself, no roles, no persistent agent identity. Its insight is that *the board plus git is enough coordination fabric.*

Paperclipai/Paperclip [3] ships a Node server and React UI that models agents as *employees of a company*: org charts, budgets, governance, goal-aligned task graphs, heartbeat-driven execution, per-agent API key rotation. Agents have persistent identity across runs and can be any runtime—OpenClaw, Claude Code, Codex, Cursor, bash, HTTP. Its insight is that *persistent agent identity plus atomic work checkout plus governance is the enterprise shape.* Its overreach is that most users do not need the enterprise shape; budgets and approval gates are separable concerns, not kernel primitives.

NanoClaw Agent Swarms [4] was positioned as the first personal AI assistant to support Claude Agent SDK *agent teams*—in-process subagents coordinating through the SDK's `query()` lifecycle. The feature is *fundamentally broken* in the only mode NanoClaw supports (non-interactive SDK in containers): subagents are silently terminated when the team-lead completes its turn, producing zero files despite appearing to succeed. The root cause is that the SDK's non-interactive `query()` forces an in-process backend whose subagent lifecycle is tied to the main query's end-turn event. NanoClaw's lesson is negative but sharp: *in-process subagent swarms are fragile to upstream SDK lifecycle decisions.* Coordination must happen at a layer the coordinating system controls.

1.3 Google Gemini Enterprise: control vs. execution

In April 2026, Google released the Gemini Enterprise Agent Platform and introduced subagents to the Gemini CLI [6, 5]. The Agent Designer is a no-code / low-code visual canvas with a chat pane and a Designer pane (Flow / Schedule / Preview tabs) for building multi-step agents. Subagents in the CLI are defined via Markdown-plus-YAML files, storable in a repository, enabling teams to standardize and version specialized agents. Users can invoke subagents explicitly via prompt syntax (`@agent-name`) or let the orchestrator route automatically.

VentureBeat [7] frames the significance: Google is positioning at the **control plane** (Kubernetes-style governance, identity, policy, audit), while AWS / Anthropic / OpenAI compete at the **execution plane** (harnesses, fast iteration, abstracted backend). Both are needed; they are different products.

Gemini Enterprise concept	**Hermes equivalent today**	**Gap, if any**
Agent Designer visual canvas	(CLI-first by design)	Dashboard plugin, eventually
Subagent as Markdown + YAML in a repo	Profiles (HERMES_HOME dirs)	**No portable file artifact yet**
Orchestrator-with-subagents runtime	`delegate_task` + (proposed) Kanban	Covered by this spec
Schedule tab	cron subsystem	Already exists
Tool connectors (Gmail / Drive / Jira)	Toolsets + MCP	Already exists
Explicit `@name` delegation	—	**Worth adopting; see §5**
Kubernetes-style control plane	—	User-space plugins; not kernel

Table 1: Gemini Enterprise feature map against Hermes. The two gaps worth closing are portable profile artifacts and `@mention` delegation syntax. Control-plane governance is explicitly *not* adopted as a kernel concern.

1.4 Design thesis

The architecture presented here synthesizes these lessons:

- Adopt Cline's **board + links + ephemeral workspaces** shape.
- Adopt Paperclip's **atomic claim** and **persistent agent identity**, but map the latter onto Hermes profiles rather than a new entity type.
- Reject NanoClaw's **in-process subagent swarm**; every worker is a full OS process, coordinating strictly through the durable board.
- Adopt Gemini's **portable profile artifacts** and **`@name` delegation syntax**.
- Reject Paperclip's and Gemini's **control-plane governance kernel**; make governance user-space.

The result is a minimum-footprint kernel—one SQLite file, one CLI subcommand, one skill, one cron job—upon which arbitrary collaboration shapes, roles, and policies can be expressed as profiles, skills, and plugins.

2 Related Systems: Comparative Analysis

Dimension	Cline Kanban	Paperclip	NanoClaw Swarms
Shape	Local board	Server + UI + "company" model	In-process SDK teams
Task granularity	One card = one worktree	Goal → project → issue	Subagent fork from team-lead
Agent identity	Anonymous per card	Persistent "employees"	Anonymous per spawn
Persistence	DB + git worktrees	Central server DB	Ephemeral, tied to query()
Dependencies	Linked cards auto-start children	First-class issue links + blockers	None explicitly
Governance	None (by design)	Budgets, approvals, audit log	None
Runtime	Agent-agnostic (Claude, Codex, Cline)	Runtime-agnostic (any heartbeat)	Claude Agent SDK only
Failure mode	Crashed worker = stale worktree	Crashed agent = orphan task (recoverable)	Silent subagent termination
Human-in-loop	Click lines, leave review comments	Approval gates, pause/resume	None
Coordination medium	Git + card state	DB + heartbeats + tickets	SDK message passing
Status	Research preview, active	Mature (58k stars)	**Broken** in the only mode it supports

Table 2: Feature comparison of contemporary multi-agent coordination systems.

The comparison is instructive. Cline Kanban is the simplest and most successful shape but is explicitly coding-centric (worktrees are mandatory, cards have no persistent identity across projects). Paperclip is the most enterprise-complete but overbuilds for non-enterprise users. NanoClaw is the cautionary tale: an architecturally clean *inbox-routing* design undermined by building on an upstream primitive (Claude Agent SDK non-interactive `query()`) whose lifecycle semantics do not support the feature.

Hermes Kanban targets the union of what works (board + links + workspaces + persistent identity) with the intersection of what is needed (no enterprise governance overhead; no fragile SDK swarms).

3 Architecture

3.1 Overview

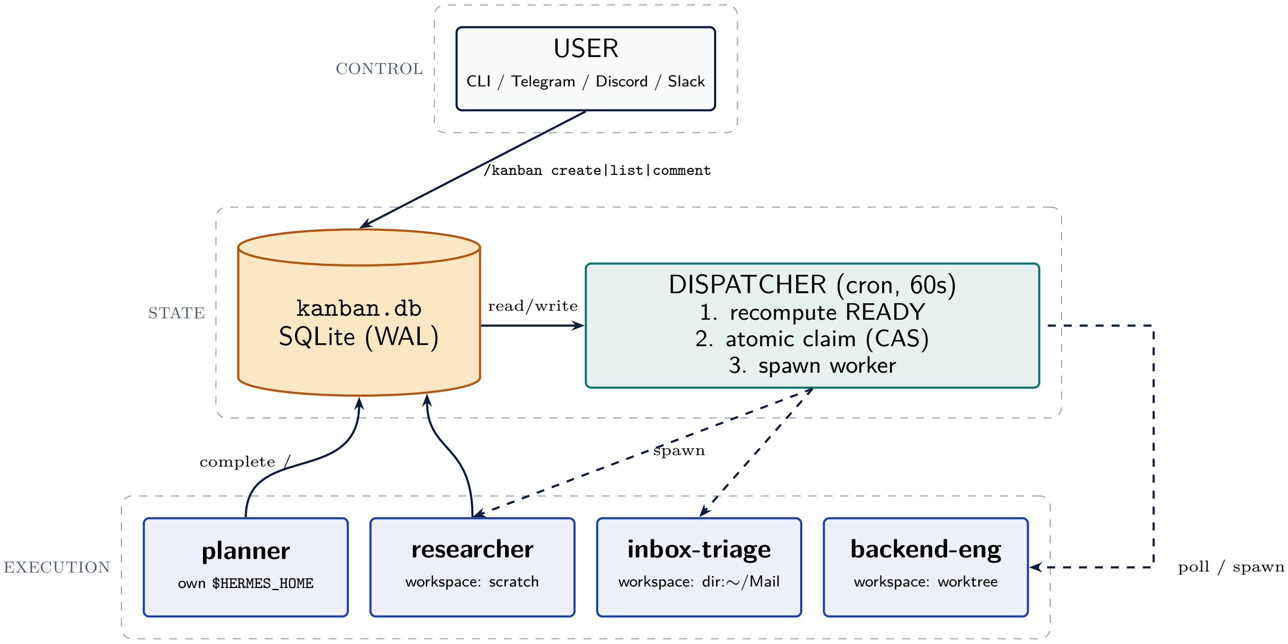

Figure 1: Three-plane architecture. The **control plane** is the user and gateway. The **state plane** is the board plus the dumb dispatcher. The **execution plane** is a pool of independent profile processes, each with isolated state. All coordination flows through the board; there is no direct inter-process communication between profiles.

3.2 Three-plane separation

Control plane. Users interact with the board via the `/kanban` slash command in the gateway (Telegram, Discord, Slack, etc.) or `hermes kanban` in the CLI. Both surfaces are derived from the same command definitions in `hermes_cli/commands.py`, so adding a command automatically propagates to all platforms.

State plane. The shared SQLite database at `~/.hermes/kanban.db` is the single source of truth. All profile processes read and write it. The dispatcher is a cron-triggered process that implements exactly three operations: (1) recompute the *ready* status of tasks whose parents have transitioned to *done*; (2) atomically claim ready tasks via a compare-and-swap SQL update; (3) spawn the assigned profile in the resolved workspace.

Execution plane. Each worker is a full Hermes process (`hermes -p <profile> chat -q ...`) with its own `HERMES_HOME`, its own memory, its own skills, and its own workspace. Workers never communicate directly; they only read and write the board.

3.3 The critical invariant: no in-process subagent swarms

Every coordinating worker is an operating-system process under the user's control, not a subagent inside the Claude Agent SDK's `query()` lifecycle. This is the lesson NanoClaw teaches in the negative. When a worker exits—whether cleanly, by crash, or by SIGKILL during a host reboot—its claim lock expires and the next dispatcher tick reclaims the task. There is no lifecycle we do not own.

4 Data Model

4.1 SQLite schema

```
CREATE TABLE tasks (
    id             TEXT PRIMARY KEY,          -- e.g. "t_9f2a"
    title          TEXT NOT NULL,
    body           TEXT,                       -- optional opening post
    assignee       TEXT,                       -- profile name; nullable =
        unassigned
    status         TEXT NOT NULL,              --
        todo|ready|running|blocked|done|archived
    priority       INTEGER DEFAULT 0,
    created_by     TEXT,                       -- profile or "user"
    created_at     INTEGER NOT NULL,
    started_at     INTEGER,
    completed_at   INTEGER,
    workspace_kind TEXT NOT NULL DEFAULT 'scratch',  -- scratch|worktree|dir
    workspace_path TEXT,                        -- resolved at claim time
    claim_lock     TEXT,                        -- host+pid of claimer; NULL
        when free
    claim_expires  INTEGER                      -- unix ts; stale-claim recovery
);

CREATE TABLE task_links (
    parent_id  TEXT NOT NULL,
    child_id   TEXT NOT NULL,
    PRIMARY KEY (parent_id, child_id),
    FOREIGN KEY (parent_id) REFERENCES tasks(id),
    FOREIGN KEY (child_id)  REFERENCES tasks(id)
);

CREATE TABLE task_comments (
    id         INTEGER PRIMARY KEY AUTOINCREMENT,
    task_id    TEXT NOT NULL,
    author     TEXT NOT NULL,                   -- profile name or "user"
    body       TEXT NOT NULL,
    created_at INTEGER NOT NULL
);

CREATE TABLE task_events (
    id         INTEGER PRIMARY KEY AUTOINCREMENT,
    task_id    TEXT NOT NULL,
    kind       TEXT NOT NULL,                   --
        status_change|claim|release|error
    payload    TEXT,                            -- JSON
    created_at INTEGER NOT NULL
);

CREATE INDEX idx_tasks_assignee_status ON tasks(assignee, status);
CREATE INDEX idx_links_child            ON task_links(child_id);
CREATE INDEX idx_comments_task          ON task_comments(task_id, created_at);
```

Listing 1: Full schema for `kanban.db`.

The schema is intentionally minimal. There are exactly four tables; there are exactly three indexes; there is no JSON column masquerading as a schema (only `task_events.payload`, which is opaque diagnostics).

4.2 Status semantics

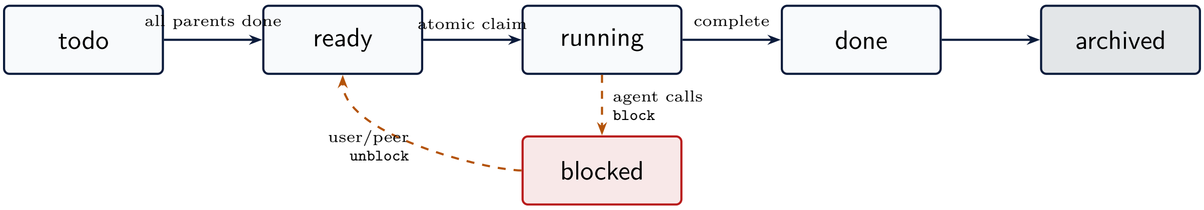

Figure 2: Task state machine. Solid arrows are the happy path. Dashed amber arrows are the human-in-the-loop recovery path. The dispatcher owns *ready→running*; workers own *running→done|blocked*; humans or peer agents own *blocked→ready*.

Status	**Owner**	**Meaning**
`todo`	creator	Task created, one or more parents not yet `done`.
`ready`	dispatcher	All parents done; eligible for atomic claim.
`running`	worker	Claimed by a profile process; worker is executing.
`blocked`	worker	Worker requires peer or human input to proceed.
`done`	worker	Completion result written; triggers children re-evaluation.
`archived`	user	Removed from default views; workspace may be GC'd.

Table 3: Six status values and their owners. Only one role may transition each status; this separation eliminates write contention.

4.3 Workspace kinds

A critical design decision for non-coding workloads: the `workspace_kind` field decouples the collaboration primitive from git worktrees. Three values are supported in v1:

Kind	**Resolution at claim**	**Use**
`scratch`	Fresh tmp dir under `~/.hermes/kanban/workspaces/<id>/`	Research tasks, one-shot analysis, anything ephemeral.
`dir:<p>`	Existing shared path `<p>` bind-mounted into worker	Long-running journals (Obsidian vaults, mail ops dirs).
`worktree`	`git worktree add .worktrees/<id>/` on the current branch	Coding tasks with isolated commits.

Table 4: Workspace kinds. Default is `scratch` because most non-coding workloads do not need git.

5 Collaboration Patterns

Seven reusable patterns fall out of the base primitives (tasks + links + comments + assignee + workspace). Documenting them as first-class idioms prevents users from having to discover them by accident.

5.1 P1 — Fan-out

One role, N siblings, no dependencies between them. "Research these five angles in parallel." Create five tasks with the same assignee; no links. The dispatcher claims all five atomically and spawns five concurrent worker processes in five separate workspaces. Parallelism through OS processes, not in-process subagents.

5.2 P2 — Pipeline

Role-specialized chain. "Scout → editor → writer." Create tasks with different assignees; link them in order. Each stage's completion result feeds the next stage's parent-result context.

5.3 P3 — Voting / Quorum (fan-in)

N workers race or vote; an aggregator picks. Create N sibling tasks with the same body but different assignees; plus one aggregator task linked from all N. When the N complete, the aggregator sees all N results in its parent-results context and decides. Zero new primitives required.

5.4 P4 — Long-running journal

Same profile + same shared dir workspace + recurring tasks. "Daily briefing appends to the vault." The profile's persistent memory plus the shared directory make the work cumulative across runs. The board serves as the audit timeline.

5.5 P5 — Human-in-the-loop triage

Block → question → unblock. Any worker may `block` a task and post a `comment` with its question. The user (or a peer profile) replies via `comment` and `unblock`. The dispatcher respawns the worker, which reads the full comment thread as part of its task prompt. No prompt surgery or context fragility.

5.6 P6 — `@mention` delegation

Inline cross-profile invocation from prose. Inspired by Google Gemini CLI's subagent invocation syntax [6]. When a message (user-authored in a gateway, or agent-authored in a kanban comment) contains the pattern `@<profile-name>`, the system interprets it as an implicit `hermes kanban create --assignee <profile-name>` with the surrounding text as the task body. A cheap parser, a large UX gain, and it makes the board feel conversational in chat surfaces without sacrificing any of its durability properties.

5.7 P7 — Thread-scoped workspace

Workspace pinned to a conversation thread. A pattern contributed by community feedback (user psbd, Nous Discord, April 25 2026). `/kanban here` in a threaded messaging platform creates a task whose `workspace_kind=dir:<path>` is derived from the thread's associated directory. Pins workspace and thread together so that conversations stay coherent across sessions, and multiple profiles operating in the same thread share a working directory without interleaving git worktrees.

5.8 P8 — Fleet farming

One specialist profile, N parallel tasks, one workspace directory per "subject." Concrete example from community feedback (user neo2k8, Nous Discord): 50 Instagram accounts managed by a single `insta-manager` profile, each task assigned `--workspace dir:~/insta/acct-<N>/`. Cron creates recurring tasks per account. Parallelism limited only by host resources. Failure recovery via stale-claim reclaim. Per-subject audit trail in the task event log. RPA-shaped workflows without RPA tooling.

Pattern coverage test

Every user story in §9 can be expressed using one or more of these eight patterns. No user story requires a primitive beyond the base schema. If a future story cannot be expressed in these patterns, it is signal that either the pattern library needs a new entry, *or* the proposed feature is not actually about collaboration and belongs elsewhere.

6 The Orchestrator Profile

A recurring frustration from early Hermes users surfaced during the Nous Discord design discussion (user *sudo_relax*, April 26, 2026):

> *"Even when I create separate profiles like Researcher, Writer, QA, the orchestrator often starts doing the work itself instead of cleanly routing it to the right specialist. The orchestrator should be the control room, not the worker."*

This is *not* a kanban architectural gap. It is a profile configuration gap that kanban makes clean to solve. The fix is a first-class **orchestrator profile** template built from three properties.

6.1 Three properties of a well-behaved orchestrator

1. Disabled execution toolsets. The orchestrator profile ships with `toolsets: [kanban, gateway, memory]` and explicitly disables `terminal`, `file`, `web`, `browser`, and `code`. It *literally cannot* do implementation work; its only tools are "create/link/assign tasks" and "read the board." When the model is tempted to "just fix this quickly," it fails the attempt — forcing delegation.

2. Orchestrator skill with prescriptive system message. A `kanban-orchestrator` skill whose system message is short, prescriptive, and anti-temptation:

- "You are a dispatcher, not a worker."
- "For any concrete task, create a kanban task and assign it to the appropriate specialist profile. Do not attempt to execute it."
- "Your job is to decompose, route, and summarize — not to research, write, or code."
- "If no specialist fits, ask the user which profile to create. Do not default to doing it yourself."

3. Standard specialist roster convention. The orchestrator skill documents a canonical starter roster (`researcher`, `writer`, `analyst`, `backend-eng`, `reviewer`, `ops`) with one-line role descriptions. Users fork and edit for their own fleets. Nothing enforces this convention; it is soft guidance to keep decomposition consistent across users.

6.2 Shipping implication: installable profile templates

Distribute the orchestrator and the specialist roster as installable profile templates:

```
$ hermes profile install orchestrator      # the router
$ hermes profile install researcher        # specialists
$ hermes profile install writer
$ hermes profile install analyst
$ hermes profile install reviewer
$ hermes profile install ops
# ... and now the fleet is on the board, ready to work.
```

Listing 2: Installing a starter fleet.

This is the "company template" idea from Paperclip, minus the ceremony: Hermes provides the org chart as a set of installable profiles, and users customize from there. It also closes the Gemini-subagent-as-file gap: these templates are the portable YAML / Markdown artifacts Google's ecosystem standardized on.

6.3 Why this is not kernel work

The dispatcher does not know or care that a profile is called "orchestrator." The board has no role field. Everything is user-space convention expressed through two primitives we already have: **profiles** (toolset restrictions) and **skills** (behavioral guidance). A user who prefers a different pattern (orchestrator that can also do quick fixes; multiple domain-specific orchestrators in parallel; no orchestrator at all) gets it by editing their own profile — no kernel change, no schema migration, no PR.

7 Multi-Tenant Context

A second theme from the Nous Discord design discussion (users *sudo_relax* and *Aiz*, April 26, 2026): users want *one specialist fleet that can operate in multiple tenant contexts* (Business A, Business B, personal) without mixing data or requiring duplicate profiles per tenant.

> *"Specialists are good at their role; they shouldn't need to be recreated for every business. The orchestrator should know which business context to give them and keep boundaries clean."*

7.1 The tenant primitive

Exactly one optional column on the `tasks` table:

```
ALTER TABLE tasks ADD COLUMN tenant TEXT;  -- nullable; defaults to NULL
CREATE INDEX idx_tasks_tenant ON tasks(tenant);
```

And one optional flag on `create`:

```
hermes kanban create "monthly report" --assignee researcher \
    --tenant business-a \
    --workspace dir:~/tenants/business-a/data/
```

That is the entire schema change. Everything else falls out of it.

7.2 What "tenant" actually means

A tenant is a **namespace**, not a new entity type. Concretely:

Scoping axis	How tenant achieves it
Workspace	Task's `workspace_path` lives under the tenant's directory: `~/tenants/<tenant>/`. Filesystem gives data isolation for free.
Memory	Worker receives `HERMES_TENANT` env var; profiles namespace memory entries by prefix.
Board view	`hermes kanban list --tenant <tenant>` filters; per-tenant `/kanban` pinned to a thread via P7.
Audit	Tenant is stamped on every `task_events` row via the task's `tenant` field. One-query export of a tenant's complete history.

Table 5: Four scoping axes, one column.

7.3 What stays shared across tenants

- The **profile identity itself.** One `researcher` profile exists; it is invoked with different tenant contexts. The profile's skills, model, and tool allowlist are tenant-agnostic.
- The **dispatcher.** One cron job, one board, all tenants.
- The **orchestrator.** Same orchestrator instance routes tasks for all tenants; it reads the tenant off the incoming task and propagates it to children.

7.4 What is deliberately *not* here

- **Access control per tenant.** v1 assumes all tenants are owned by the same user. A future plugin can add per-tenant credentials or approval gates; not a kernel concern.

- **Cross-tenant task dependencies.** Links cannot cross tenants in v1. Fold cross-tenant work into a neutral parent tenant if needed.
- **Tenant-scoped profiles.** A profile belongs to the user, not to a tenant. If `business-a-researcher` and `business-b-researcher` should genuinely differ (not just by context), create two profiles. Tenants scope data, not profile definitions.

7.5 The “small company serving multiple clients” ask, solved

One `researcher` profile. One `writer`. One `reviewer`. Invoked with `--tenant business-a` they work inside Business A’s data. Invoked with `--tenant business-b` they work inside Business B’s data. The orchestrator propagates the tenant to children. Boundaries are enforced by workspace path and memory namespace, not by yet another new entity type. This is the OS-for-agent-fleets framing that has been building in the community discussion; it is achieved with a single nullable column and a convention.

8 Worked Example: 50-Account Social Media Fleet

A concrete demonstration of pattern P8 (Fleet farming), using the social media automation example raised in the Nous Discord discussion (user *neo2k8*):

> *"My use case would be Instagram marketing automation…creation of profiles, posting of videos and pictures etc. When I could control a farm of 50 agents all simultaneously managing their accounts while behaving like a human (from Instagram's point of view) I would be really happy."*

8.1 Setup

```
# One specialist profile, installed once
$ hermes profile install insta-manager
# ... or define your own: tools limited to browser + file + schedule

# Per-account workspaces on disk
$ for i in $(seq 1 50); do
    mkdir -p ~/insta/acct-$i/{assets,drafts,logs}
  done

# Cron template: generate daily work for each account at jittered times
$ cat ~/.hermes/cron/insta-fleet.yaml
schedule: "0 */4 * * *"
command: hermes kanban dispatch-fleet insta-manager ~/insta
```

8.2 Per-tick task generation

The `dispatch-fleet` helper (user-space, not kernel) enumerates accounts and creates one task per account per tick:

```
# Generated by dispatch-fleet
$ for acct in acct-1 acct-2 ... acct-50; do
    hermes kanban create "engage acct-$acct" \
      --assignee insta-manager \
      --workspace dir:~/insta/$acct/ \
      --tenant $acct
  done
# -> 50 tasks, all ready, all claimable in parallel
```

8.3 The dispatcher does what it does

```
# dispatcher tick at :00
$ hermes kanban dispatch
# -> 50 atomic claims succeed, 50 insta-manager processes spawn

# Each worker sees:
#   HERMES_TENANT=acct-17
#   workspace: ~/insta/acct-17/
#   task body: "engage acct-17"
#   profile memory: account-17-specific (namespaced by tenant)

# Worker executes with human-like pacing (skill enforces jitter)
# Logs outcomes to ~/insta/acct-17/logs/YYYY-MM-DD.jsonl
# Writes completion result to kanban

$ hermes kanban complete T-xxx --result "posted 2, liked 18, followed 1"
```

8.4 Why this is clean

- **No fleet runtime.** No farm-specific code paths. This is just pattern P1 (fan-out) + P4 (journal) + tenants + cron.
- **Natural failure isolation.** Account-17 hitting an Instagram captcha blocks one task; the other 49 proceed. Next tick reclaims the stale one.
- **Per-account audit.** Thirty days of engagement on account-17 is `SELECT * FROM task_events WHERE tenant = 'acct-17' ORDER BY created_at`.
- **Horizontal scale.** Adding a 51st account is `mkdir` and one line added to the enumerator. No profile duplication, no new dispatch logic.

RPA tools solve this problem with dedicated runtimes, schedulers, and account-state databases. Hermes Kanban solves it with a board, a tenant column, and a filesystem convention. The absence of dedicated infrastructure is the feature.

9 User Stories

Each story names the profiles involved, the workspace kind, and the specific question “why not use `delegate_task`?” so that the boundary between the two primitives is explicit.

9.1 Story 1: Research triage and synthesis (non-coding)

User intent

“Fire a research question at my assistant. Have it split the work across specialist profiles. Let me correct course when it hits ambiguity. End with a synthesized answer I can act on.”

Profiles: `planner`, `researcher` (×N), `analyst`, `writer`.
Workspace: `scratch`.
Patterns used: P1 (fan-out researchers), P2 (analyst → writer pipeline), P3 (analyst as fan-in aggregator), P5 (blocked researcher asks user a question).
Why not `delegate_task`? Five reasons, any one of which is sufficient: (i) mid-flight human-in-the-loop correction; (ii) durable journal across weeks; (iii) multi-role handoff that `delegate_task` cannot express without nested blocking; (iv) parallel + named-winner aggregation; (v) visibility into partial progress before the work completes.

```
$ /kanban create "what killed Gemini 2 adoption?" --assignee planner
  -> T1

# planner decomposes
$ (planner) hermes kanban create "angle: cost"        --assignee researcher
    --parent T1
$ (planner) hermes kanban create "angle: latency"     --assignee researcher
    --parent T1
$ (planner) hermes kanban create "angle: tool-quality"--assignee researcher
    --parent T1
$ (planner) hermes kanban create "angle: licensing"   --assignee researcher
    --parent T1
$ (planner) hermes kanban create "synthesize"         --assignee analyst
$ (planner) hermes kanban link T2 T6 && link T3 T6 && link T4 T6 && link T5 T6
$ (planner) hermes kanban create "write brief"        --assignee writer
$ (planner) hermes kanban link T6 T7
$ (planner) hermes kanban complete T1

# dispatcher fans out T2..T5 in parallel
$ hermes -p researcher chat -q "work T2" &
$ hermes -p researcher chat -q "work T3" &   # hits gated source
$ hermes -p researcher chat -q "work T4" &
$ hermes -p researcher chat -q "work T5" &

# T3 blocks
$ (researcher) hermes kanban comment T3 "source gated; alternative?"
$ (researcher) hermes kanban block T3
$ /kanban comment T3 "use arxiv-sanity mirror"
$ /kanban unblock T3

# cascade into analyst and writer
$ hermes kanban complete T2 / T3 / T4 / T5    ->  T6 ready
$ hermes -p analyst chat -q "work T6"         ->  ranked + deduped
$ hermes -p writer  chat -q "work T7"         ->  brief delivered
```

Listing 3: Research triage worked example.

9.2 Story 2: Scheduled recurring workflow (non-coding)

User intent

"Every weekday at 9am give me my AI-funding brief. Pull sources, dedupe, rank by relevance to me, post to Telegram. If it fails I want to see why without re-running from scratch."

Profiles: `scout`, `editor`, `writer`.
Workspace: `dir:~/Obsidian/AI-Funding/`.
Patterns used: P2 (pipeline), P4 (long-running journal).
Why not `delegate_task`? Scheduled, resumable across days, auditable as a thirty-task timeline in SQLite, operates on a long-lived vault. `delegate_task` is ephemeral by design; this is the opposite.

9.3 Story 3: Digital-twin / persistent assistant role (non-coding)

User intent

"I want a named 'inbox-triage' assistant that processes my email every hour, builds up knowledge of who matters to me, and gets better over time. Its own memory, its own voice."

Profiles: `inbox-triage` (a single durable identity).
Workspace: `dir:~/Mail-Ops/`.
Patterns used: P4 (journal), P5 (blocks on contract needing legal review), P6 (escalates to `@legal` profile).
Why not `delegate_task`? There is no parent agent. The profile *is* the agent. It exists across runs with its own memory, voice, and accumulating skills.
Why not just a cron job? Cron gives scheduling but no handoff primitive. When `inbox-triage` needs a lawyer, the board routes the task; a cron job alone does not.

9.4 Story 4: Coding pipeline (code-shaped)

User intent

"Ship a feature end-to-end: decompose, implement in parallel worktrees, review, iterate, open PR."

Profiles: `planner`, `backend-eng` (×N), `frontend-eng`, `reviewer`.
Workspace: `worktree` (Cline Kanban pattern).
Patterns used: P1 (parallel engineers), P2 (pipeline to reviewer), P5 (ambiguity handoff), P2 (review feedback loop via follow-up task).
Why not `delegate_task`? Long-running (minutes to hours); parallel worktrees; human or peer may intervene at any point; review cycle potentially iterates multiple times; each phase has a different specialist profile with different skills.

9.5 What all four stories share

- Multiple distinct profiles, each with its own `HERMES_HOME`, memory, skills.
- Dependencies between tasks (sequential or fan-in).
- Humans may inspect, comment, unblock, or reassign at any point.
- Durable history: the board is the audit trail.

- Workspace type varies across stories (`scratch`, `dir:<path>`, `worktree`) but the coordination primitive is identical.

If v1 does not support all four stories cleanly, we built the wrong thing.

10 Kanban vs. `delegate_task`

These primitives look similar from ten feet. They are fundamentally different, and the boundary between them is the central design question of this specification.

Dimension	**`delegate_task`**	**Kanban**
Shape	RPC call (fork → join)	Durable message queue + state machine
Parent lifecycle	Blocks until child returns	Fire-and-forget; parent done after create
Child lifecycle	Ephemeral; dies after returning summary	Ephemeral *per run*; task persists across runs
Context passing	`goal` + `context` strings at call time	Task title + body + comment thread + parent results, read from board at spawn
Identity	Anonymous subagent, no persistent self	Named profile; own `HERMES_HOME`, memory, skills, history
Resumability	None; failed = failed; parent retries from scratch	`blocked` → `unblock` → re-run; crash → reclaim in 15 min
Human interposition	None; runs headless	Comment, block, unblock, reassign at any time
Agents per task	One subagent per call	N agents over the task's life (retry, review, follow-up)
Dependency graph	Manual, in parent's head	First-class `task_links`; dispatcher enforces order
Audit trail	Lost on parent's context compression	Durable SQLite rows forever
Coordination shape	Strictly hierarchical (caller → callee)	Peer; any profile reads/writes any task
Inter-profile comms	None	Board-mediated, durable, asynchronous

Table 6: Twelve dimensions along which `delegate_task` and Kanban differ.

One-sentence distinction. `delegate_task` is a function call; Kanban is a durable work queue where every handoff is a row any profile (or human) can read and edit.

When to use which.

- **Use `delegate_task`** for short, self-contained reasoning subtasks the parent agent wants an answer to before continuing: "research X and summarize," "review this diff and flag issues," "run these three checks in parallel." Seconds-to-minutes. No human in the loop. Result goes back into parent's context.
- **Use Kanban** for work that crosses agent boundaries; needs to survive restarts; might need human input; might be picked up by a different role (engineer writes, reviewer reviews, engineer fixes); or needs to be discoverable after the fact.

They coexist. A kanban worker may still call `delegate_task` internally for reasoning within its own run; this is expected. The single test is:

> *Does this handoff need to outlive a single API loop and be visible to others?* If yes, board. If no, delegate.

11 Assignment Semantics

Base rule. A task has exactly one `assignee` field, which is a profile name. Everything else falls out of this single field.

Who creates tasks. Anyone: the user via `/kanban create`; any profile via `hermes kanban create` from the terminal; the `router` profile (user-space, optional) if the user sets one up. No creator role is privileged.

The "planner" profile is a convention, not a structural role. It is simply "a profile whose skills and system prompt lean toward decomposition." For simple goals, skip it and create child tasks directly.

Worker's view of a task. When the dispatcher spawns a worker, the task prompt it sees contains, in order:

1. Task title (mandatory).
2. Task body (optional opening post; often empty).
3. Every comment on the task, chronologically, with author names (`user`, `planner`, `backend-eng`, etc.).
4. Completion results of every parent task.
5. The worker profile's normal skills and memory (unchanged).

That is the complete context. No hidden channel. No out-of-band state. If it is not visible on `hermes kanban show <id>`, the worker cannot see it either.

What is deliberately *not* supported in v1.

- **Two assignees on one task.** Causes claim races; muddies accountability. Use two linked sibling tasks instead.
- **Round-robin pools.** Could be added later; v1 is explicit assignment.
- **Auto-assignment ("any idle profile claims it").** Not in v1. Add `--open` queue mode later if the need is real.
- **Broadcast ("fan this task out to every profile").** Not a primitive. If N copies are desired, create N tasks.

12 Dispatcher Design

The dispatcher is deliberately dumb. Its only operations are:

1. **Recompute `ready`.** For each task in `todo`, if all parent links resolve to tasks in `done`, transition the task to `ready`.
2. **Atomic claim.** For each `ready` task with `claim_lock IS NULL` and `assignee IS NOT NULL`, issue a compare-and-swap update setting `status='running'`, `claim_lock=<host>:<pid>`, and `claim_expires=now+900`.
3. **Spawn worker.** For each successful claim, resolve the workspace (create scratch dir / ensure worktree / validate shared dir) and execute `hermes -p <assignee> -w <workspace> chat -q "work kanban task <id>"`.
4. **Stale claim recovery.** For each `running` task with `claim_expires < now`, reset `status='ready'` and `claim_lock=NULL`.

There is no smart routing. There are no priorities beyond the `priority` ORDER BY tiebreaker. There is no backpressure beyond "the dispatcher runs every 60 seconds." These are all user-space concerns, and any of them may be added as a profile-level behavior (for example, a `router` profile that reassigns unassigned tasks) without touching the dispatcher itself.

12.1 Concurrency correctness

The dispatcher may run concurrently with itself (two cron ticks overlap) and with any number of manual `hermes kanban claim` invocations. Correctness rests on SQLite's BEGIN IMMEDIATE transaction semantics combined with the row-level CAS pattern:

```sql
BEGIN IMMEDIATE;
UPDATE tasks
  SET status = 'running',
      claim_lock = ?,
      claim_expires = ?,
      started_at = ?
  WHERE id = ?
    AND status = 'ready'
    AND claim_lock IS NULL;
-- examine changes(); 1 = won the claim, 0 = lost the race
COMMIT;
```

Because `WHERE status='ready' AND claim_lock IS NULL` is re-evaluated inside the transaction and SQLite serializes writers via its WAL lock, at most one claimer can win any given task. Losers simply observe zero affected rows and move on.

13 CLI and Gateway Surface

13.1 Command surface

Command	Purpose
`hermes kanban init`	Create DB if missing.
`hermes kanban create "<title>" [--body ...] [--assignee P] [--workspace K]`	Create a task. `K` is `scratch` (default), `worktree`, or `dir:<path>`.
`hermes kanban list [--mine] [--assignee P] [--status S] [--json]`	List tasks with filters.
`hermes kanban show <id>`	Show task detail, comments, events.
`hermes kanban assign <id> <profile>`	Assign or reassign.
`hermes kanban link <parent> <child>`	Add a dependency link.
`hermes kanban unlink <parent> <child>`	Remove a dependency link.
`hermes kanban claim <id>`	Manual atomic claim; prints resolved workspace.
`hermes kanban comment <id> "<text>"`	Append a comment.
`hermes kanban complete <id> [--result "..."]`	Mark done; triggers child re-evaluation.
`hermes kanban block <id> "<reason>"`	Mark blocked; surfaces to creator via gateway.
`hermes kanban unblock <id>`	Return to `ready`.
`hermes kanban archive <id>`	Remove from default views; GC

Profile portability (addressing the Gemini-subagent-as-file pattern):

```
$ hermes profile export ops-reviewer > ops-reviewer.yaml
$ git add ops-reviewer.yaml && git commit -m "versioned ops-reviewer profile"
$ # teammate:
$ hermes profile install ./ops-reviewer.yaml
```

Listing 4: Portable profile artifacts (separate PR from kanban v1; listed here for completeness).

13.2 Gateway integration

One new slash command, `/kanban`, is added to the central `COMMAND_REGISTRY` in `hermes_cli/commands.py`. This single registration propagates automatically to Telegram, Discord, Slack menus, CLI autocomplete, and `/help` output, per the existing platform-agnostic command mechanism.

The running-agent guard (`gateway/run.py`) must permit `/kanban list` and `/kanban show` while an agent is running, but reject mutations. `/kanban unblock` must always bypass the guard, because unblocking is often the only action that frees a stuck agent.

14 Scope Boundaries

The kernel stays small on purpose. The following features have all been proposed or requested, and they all belong in user-space profiles or plugins, not in the kernel:

Proposed feature	**User-space realization**
“Smart routing” / auto-assignment	A `router` profile whose job is to scan unassigned tasks and reassign them. Not a dispatcher feature.
“Org chart / hierarchy”	Profile naming convention + skills. Not a schema change.
“Budgets per agent”	A plugin that wraps `spawn` to enforce limits. Not a task field.
“Fleet management dashboards”	A dashboard plugin that reads the board. Not in the CLI.
“Approval gates”	Reuse `tools/approval.py`. Not a new status.
“Governance control plane” (Gemini)	Combination of router profile + budget plugin + audit-export plugin. Not kernel.

Table 8: Out-of-kernel features and their user-space realizations.

This is NanoClaw’s lesson in reverse: their kernel was small, and their coordination feature broke because it relied on upstream primitives (Claude Agent SDK in-process subagents) whose lifecycle semantics did not support the use case. Our kernel stays small, and ornaments go to user-space profiles and plugins where they cannot take down the coordination fabric.

15 Implementation Plan

Eight pull requests, each independently shippable and reviewable. Each PR ends with a merged, working slice of functionality.

#	PR	What lands
1	Scaffolding	`hermes_cli/kanban.py`, `kanban_db.py`, DESIGN.md, skill skeleton, `CommandDef("kanban",...)`. No dispatcher.
2	Single-profile path	`create`, `list`, `show`, `claim`, `complete`, `comment`. No deps, no workspace autocreate.
3	Dependency resolution	`link`, `unlink`, `todo` → `ready` recomputation on `done`.
4	Atomic claim + workspace	CAS claim, `ensure_workspace(kind)`, stale-claim recovery.
5	Dispatcher	`hermes kanban dispatch` one-shot pass; cron template.
6	Gateway slash command	`/kanban` handler in `gateway/run.py`, running-agent guard logic.
7	Worker skill	`kanban-worker` skill teaching a profile how to handle a spawn.
8	Portable profile artifacts	`hermes profile export/install` (separate track, unblocks Gemini-parity).
9	Dashboard plugin (opt.)	Read-only board view in the web dashboard. Deferred.

Table 9: Implementation roadmap.

15.1 Files touched

New files.

- `hermes_cli/kanban.py` — CLI subcommand and DB access layer.
- `hermes_cli/kanban_db.py` — schema, migration, CAS helpers.
- `skills/kanban-worker/SKILL.md` — worker-side instructions.
- `tests/hermes_cli/test_kanban.py` — schema, claim atomicity, dependency resolution.
- `tests/hermes_cli/test_kanban_dispatch.py` — dispatcher behavior.
- `website/docs/user-guide/features/kanban.md` — user documentation.
- `cron/templates/kanban_dispatch.yaml` — default cron template.

Files touched.

- `hermes_cli/commands.py` — add `CommandDef("kanban", ...)`.
- `hermes_cli/main.py` — register `kanban` argparse subparser.
- `gateway/run.py` — slash command handler and guard.
- `cli.py` — `process_command()` dispatch for `/kanban`.
- `website/sidebars.ts` — add kanban doc.
- `website/docs/reference/cli-commands.md` — add subcommand.

Files not touched.

- `run_agent.py` — zero changes. No new core tool. No schema bloat.
- `model_tools.py` / `toolsets.py` — zero changes.

- `agent/` — zero changes.

16 Risks, Tradeoffs, Open Questions

16.1 Risks

1. **SQLite cross-process contention.** Multiple profiles writing simultaneously. Mitigation: WAL mode, `BEGIN IMMEDIATE` for claim transactions, short-held locks, no long reads inside write transactions. Proven pattern from `hermes_state.py`.

2. **Stale workspace buildup.** Tasks abandoned mid-run leave workspaces on disk. Mitigation: `hermes kanban gc` periodic cleanup, trigger on `archive`.

3. **Profile misconfiguration at spawn.** Cron launches a profile that lacks the necessary skills. Mitigation: on `assign`, verify the profile exists; warn if the `kanban-worker` skill is disabled.

4. **Cron scheduler drift on laptop sleep/wake.** Missed ticks. Mitigation: every `hermes kanban` CLI invocation runs a cheap "mini dispatch" to recompute ready-states. Acceptable — tasks that wait a few extra minutes are fine at this scale.

5. **Runaway automation chains.** Profile A completes task 1, which auto-launches task 2 on profile B, which auto-completes and launches task 3, with no human in the loop. Mitigation: optional `--require-approval` flag on create; assigned approver profile must `unblock` before transition to `ready`. Reuses existing approval infrastructure.

16.2 Tradeoffs

- **Polling over events.** Cron poll is simpler than a real event bus. Cost: up to 60 seconds of latency between dependency-clear and worker launch. Acceptable for this scale.

- **CLI-first, dashboard-later.** Dashboard is better UX but not required for correctness. Shipping it with v1 doubles review surface; defer.

- **Profiles are agents, not roles.** A profile is the identity (config + skills + memory). There is no separate role model. Simpler. But "changing an agent's personality" is `hermes profile edit`, not runtime config.

16.3 Open questions

1. Should Kanban be exposed as a **toolset** (agent-callable as a tool) or purely as a **CLI + skill**? The ecosystem prefers the latter: zero schema bloat, skill teaches the flow. Recommend CLI + skill. Revisit if agents routinely need structured kanban I/O.

2. Should unassigned `ready` tasks be claimable by any profile (open queue), or require explicit assignment? Recommend explicit; add `--open` flag later.

3. Should `blocked` tasks auto-escalate to a human via the gateway? Recommend yes: `block` writes a gateway message to the task creator.

4. Do we need `hermes kanban export/import` for board portability (a "save-this-workflow-as-template" feature)? Defer; not v1.

5. What is the relationship between kanban tasks and cron jobs? Overlapping (a recurring cron is similar to a recurring task). Recommend keeping them separate in v1; cron is time-triggered, kanban is dependency-triggered. Revisit if the overlap becomes painful.

17 Conclusion

Hermes Kanban adopts Cline Kanban's board shape, borrows Paperclip's atomic-claim and persistent-identity primitives (mapped onto existing Hermes profiles), explicitly rejects NanoClaw's fragile in-process subagent swarms, and inherits two good ideas from Google's April 2026 Gemini Enterprise release (portable profile artifacts, `@mention` delegation syntax). The result is a minimum-footprint kernel—one SQLite file, one CLI subcommand, one skill, one cron job, zero changes to `run_agent.py`—upon which four distinct workload shapes (research, scheduled operations, digital twins, engineering) can be expressed through seven reusable collaboration patterns.

The boundary against `delegate_task` is explicit: delegate is a function call, Kanban is a durable work queue where every handoff is a row any profile (or human) can read and edit. The two primitives coexist; a kanban worker may call `delegate_task` for internal reasoning within its run. The single test is whether a handoff needs to outlive a single API loop and be visible to others.

The governance debate playing out between Google (control-plane) and AWS / Anthropic / OpenAI (execution-plane) is resolved here in favor of execution-plane simplicity. Governance, budgets, smart routing, fleet dashboards, and approval gates are all expressible as user-space profiles and plugins, not as kernel primitives. This preserves the Hermes philosophy of small, debuggable cores and pushes complexity to the layer where it can be iterated on without destabilizing the coordination fabric.

Next step

A DESIGN.md-only scaffolding PR, tracking this specification, is the proposed starting point. No code changes; only a committed design artifact, a skill skeleton, and a `CommandDef` stub, so the direction can be reviewed and iterated on in the repository before any implementation work begins.

References

[1] Cline Bot Inc. *Kanban: Launch a local web app that runs CLI agents in parallel.* GitHub repository, 2026. `https://github.com/cline/kanban`

[2] Sidd Sant. *Announcing Cline Kanban: a CLI-agnostic app for multi-agent orchestration.* Cline Blog, March 26, 2026. `https://cline.bot/blog/announcing-kanban`

[3] Paperclip AI. *Paperclip: Open-source orchestration for zero-human companies.* GitHub repository, 2026. `https://github.com/paperclipai/paperclip`

[4] Fufu-0101 (reporter). *Agent Swarms: in-process subagents silently terminate when main agent completes turn (SDK/container mode).* qwibitai/nanoclaw GitHub Issue #684, opened March 4, 2026. `https://github.com/qwibitai/nanoclaw/issues/684`

[5] Google Cloud. *Agent Designer overview.* Gemini Enterprise documentation, April 23, 2026. `https://docs.cloud.google.com/gemini/enterprise/docs/agent-designer`

[6] Robert Krzaczyński. *Subagents in Gemini CLI Enable Task Delegation and Parallel Agent Workflows.* InfoQ, April 20, 2026. `https://www.infoq.com/news/2026/04/subagents-gemini-cli/`

[7] Emilia David. *Google and AWS split the AI agent stack between control and execution.* VentureBeat, April 22, 2026. `https://venturebeat.com/orchestration/google-and-aws-split-the-ai-agent-stack-between-control-and-execution`

— End of specification —

Hermes Kanban, Revision 01, April 25, 2026