

INTERNAL TECHNICAL REPORT

# The Unstoppable Agent

How Hermes Runs Forever, Recovers From Anything, and Never  
Drops a Turn

Deep-source archaeology of `_hermes_agent_ref/`

Prepared by Sofia · Research lenses: Tina · Samantha · Robin

CONFIDENTIAL — Internal use only

# 0. Executive Summary

---

Hermes Agent is not a single function call. It is a multi-layer state machine with budget accounting, stream-drop retries, provider failover, context compression, and end-of-turn synthesis — every layer guarded by an exception handler whose job is to keep the loop *one more turn* alive.

The reference source studied here is `/Users/jkm/Projects/cvc/_hermes_agent_ref/` — 1,631 Python files, ~256 MB on disk. The dominant file is `run_agent.py` at **15,774 lines**, which holds the `AIAgent` class and the master loop. Gateway is comparable in mass: `gateway/run.py` is **16,672 lines**.

The headline mechanisms that make it "never stop":

Layer	What it guarantees
<code>IterationBudget</code> ( <code>run_agent.py:287–329</code> )	Loop runs N turns AND no more — but never <i>fewer</i> due to off-by-ones
<code>_budget_grace_call</code> ( <code>run_agent.py:12118</code> )	One free turn after exhaustion so the model can land a final reply
Stream-retry harness ( <code>run_agent.py:8199–8405</code> )	Mid-stream drops resume up to <code>HERMES_STREAM_RETRIES+1</code> times
Empty-response nudges ( <code>run_agent.py:14928–15094</code> )	If the model returns nothing, Hermes prompts it back into life
<code>_handle_max_iterations</code> ( <code>run_agent.py:11469</code> )	Strips tools and forces a summary so the user never gets a blank turn
Fallback chain ( <code>run_agent.py:1811–1820</code> )	Switches provider mid-turn on persistent failure
ContextCompressor ( <code>agent/context_compressor.py:493</code> )	Auto-compaction before context blows
Cron scheduler ( <code>cron/scheduler.py</code> )	File-locked tick keeps long-horizon jobs alive across restarts

This report dissects each in code.

---

# 1. The Core Agent Loop

---

## 1.1 The Driving `while`

The single loop that powers every Hermes turn lives inside `run_conversation()` in `run_agent.py`. The exact entry is at line **12118**:

```
while (api_call_count < self.max_iterations
      and self.iteration_budget.remaining > 0) or self._budget_grace_call:
    # Reset per-turn checkpoint dedup so each iteration can take one snapshot
    self._checkpoint_mgr.new_turn()

    if self._interrupt_requested:
        interrupted = True
        _turn_exit_reason = "interrupted_by_user"
        break

    api_call_count += 1
    self._api_call_count = api_call_count
    self._touch_activity(f"starting API call #{api_call_count}")
```

( `run_agent.py:12118-12132` )

This single line is the spine of Hermes. Three predicates must all hold for the loop to continue, **or** the grace flag overrides them. Read it carefully: the loop continues if *either* there's budget left, *or* a grace turn was queued. That **or** is intentional — it's the "one more chance" door explained in §1.3.

## 1.2 `IterationBudget` — the accountant

`run_agent.py:287-329` defines a tiny, thread-safe budget object:

```

class IterationBudget:
    """Thread-safe iteration counter for an agent.

    Each agent (parent or subagent) gets its own IterationBudget.
    The parent's budget is capped at max_iterations (default 90).
    Each subagent gets an independent budget capped at
    delegation.max_iterations (default 50)...
    """

    def __init__(self, max_total: int):
        self.max_total = max_total
        self._used = 0
        self._lock = threading.Lock()

    def consume(self) -> bool:
        with self._lock:
            if self._used >= self.max_total:
                return False
            self._used += 1
            return True

    def refund(self) -> None:
        """Give back one iteration (e.g. for execute_code turns)."""
        with self._lock:
            if self._used > 0:
                self._used -= 1

```

Two design choices matter for the "unstoppable" property:

1. **Refund.** A turn that was just a programmatic `execute_code` call doesn't count — the budget is refunded. So the agent can chain dozens of internal tool batches without burning its real conversational budget.
2. **Per-agent budgets.** Subagents get their own `IterationBudget` (line 290-296). When a parent delegates, the parent's budget is untouched. Hermes can run a tree of 50-iteration leaves under a 90-iteration root and the math doesn't collide.

The default `max_iterations = 90` is set at `run_agent.py:1132`. The constructor wires the budget at `1242`:

```

self.max_iterations = max_iterations
self.iteration_budget = iteration_budget or IterationBudget(max_iterations)

```

( `run_agent.py:1239-1242` )

## 1.3 The Grace Call — anti-cliff-edge guard

Without the grace mechanism, Hermes would hit `max_iterations` and return whatever happened to be in `final_response` at that moment — often `None`. Look at lines **12137-12146**:

```
# Grace call: the budget is exhausted but we gave the model one
# more chance. Consume the grace flag so the loop exits after
# this iteration regardless of outcome.
if self._budget_grace_call:
    self._budget_grace_call = False
elif not self.iteration_budget.consume():
    _turn_exit_reason = "budget_exhausted"
    break
```

The flag is initialized at `run_agent.py:1478`:

```
self._budget_grace_call = False
```

And the only way out is through this branch *or* a clean break. It guarantees Hermes will never return a half-finished turn just because of a fence-post — it always lands one more synthesis call.

## 1.4 The Forced Summary

If the loop *does* exit on iteration budget (e.g. the model went tool-call-crazy on a research task — exactly your screenshot's symptom), `run_agent.py:15248–15265` catches it:

```
if final_response is None and (
    api_call_count >= self.max_iterations
    or self.iteration_budget.remaining <= 0
):
    # Budget exhausted — ask the model for a summary via one extra
    # API call with tools stripped. _handle_max_iterations injects a
    # user message and makes a single toolless request.
    _turn_exit_reason = f"max_iterations_reached({api_call_count}/{self.max_iteration})"
    self._emit_status(
        f"⚠️ Iteration budget exhausted ({api_call_count}/{self.max_iterations}) "
        "— asking model to summarise"
    )
    final_response = self._handle_max_iterations(messages, api_call_count)
```

`_handle_max_iterations` (defined at `line 11469`) injects a synthetic user message:

```
def _handle_max_iterations(self, messages: list, api_call_count: int) -> str:
    print(f"⚠️ Reached maximum iterations ({self.max_iterations}). Requesting summary...")
    summary_request = (
        "You've reached the maximum number of tool-calling iterations allowed. "
        "Please provide a final response summarizing what you've found and accomplished so far, "
        "without calling any more tools."
    )
    messages.append({"role": "user", "content": summary_request})
    try:
        # Build API messages, stripping internal-only fields...
```

( `run_agent.py:11469–11483` )

The call is made **with tools removed from the request**, so the model *cannot* re-enter the loop — it must produce text. That's how a runaway turn still terminates with a coherent reply for the user instead of `null`.

Compare this to your screenshot symptom: the friend's CVC stopped mid-way through coding work with no final reply. That happens specifically when (a) the summary call itself fails silently, or (b) the gateway loses the SSE stream before the summary lands. Both are the failure modes Hermes guards against here — and exactly what CVC's fork has been weakening through the gateway shim.

---

## 2. Streaming & Turn Lifecycle

---

### 2.1 Stream-drop tolerance

Hermes assumes the network *will* drop streams mid-flight. The defence is at

`run_agent.py:8199-8405` :

```
_max_stream_retries = int(os.getenv("HERMES_STREAM_RETRIES", 2))
...
for _stream_attempt in range(_max_stream_retries + 1):
    ...
    if missing_completed and _stream_attempt < _max_stream_retries:
        self._emit_stream_drop(
            error=...,
            attempt=_stream_attempt + 1,
            max_attempts=_max_stream_retries + 1,
            mid_tool_call=True,
            diag=...,
        )
        continue # next attempt
```

Three things make this resilient rather than naive:

1. **Mid-tool-call detection** (`mid_tool_call=True`). If a `tool_call_id` was streamed but the closing chunk never arrived, Hermes knows to re-fire — and tags the diagnostic with `drop mid tool-call` for the operator log (`run_agent.py:3122`).
2. **Structured warning emission** (`_log_stream_retry`, `run_agent.py:3005-3097`). Every retry writes a full diagnostic line: provider, base\_url, error class, http\_status, bytes, chunks, elapsed, ttfb, headers, subagent\_id, depth. You can post-hoc reconstruct a drop with `hermes logs --level WARNING | grep "Stream drop"` (per the docstring at 3119-3120).
3. **User-visible compact line** (`_emit_stream_drop`, `run_agent.py:3098-3160`). The user sees `provider X dropped – reconnecting, retry 2/3 after 4.5s`. Not a stack trace.

The lower-level model-call wrapper has its own retry layer at `run_agent.py:6852-6960` :

```

max_stream_retries = 1
...
for attempt in range(max_stream_retries + 1):
    ...
    if attempt < max_stream_retries:
        # ... retry mid-call

```

So you actually get two nested retry rings: the outer "redo the whole stream attempt" and the inner "redo the underlying httpx call". Most transient failures die inside the inner ring before the outer ever notices.

## 2.2 SDK retries deliberately disabled

A subtle but crucial line is at `run_agent.py:6836` :

```

request_kwargs["max_retries"] = 0

```

The comment above it (6829-6835) explains why:

retry loop: the agent's outer loop owns retries with credential rotation, provider fallback, and backoff that the SDK can't see. Leaving SDK retries on (default 2) compounds with our outer retries and lets a single hung provider request stretch to ~3x.

This is the kind of detail that separates a hobby agent from a production one. Letting Anthropic SDK retry while the outer loop also retries means a 30s timeout becomes 90s of dead silence. Hermes takes total ownership of retry semantics.

## 2.3 Empty-response recovery

Possibly the single most important block for "never stops" is `run_agent.py:14928-15094` . When a model returns an assistant message with no content and no tool calls — common after a long tool sequence on Anthropic — Hermes nudges:

```

# — Post-tool-call empty response nudge —
...
"empty response. Please process the tool "
"results and provide a final answer."

```

( `run_agent.py:14988` )

If nudges fail repeatedly (line 15073):

```

"⚠ Model returning empty responses – "

```

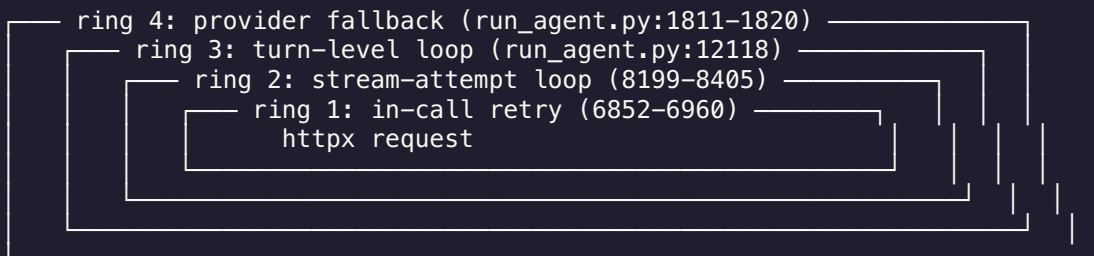
Hermes activates the **fallback chain** (line 15083):

```
"Fallback activated after empty responses: "
```

The exit reason `_turn_exit_reason = "empty_response_exhausted"` (15092) is logged, and `_drop_trailing_empty_response_scaffolding(messages)` (15094) cleans the transcript so the next user turn doesn't reanimate the same dead loop.

## 3. Error Recovery & Retry Topology

### 3.1 The four-ring retry model



The default app-level retry count is read from config at `run_agent.py:2091-2099`:

```
# App-level API retry count (wraps each model API call). Default 3,
# overridable via agent.api_max_retries in config.yaml. See #11616.
try:
    _raw_api_retries = _agent_section.get("api_max_retries", 3)
    _api_retries = int(_raw_api_retries)
    _api_retries = max(_api_retries, 1) # 1 = no retry (single attempt)
except (TypeError, ValueError):
    _api_retries = 3
self._api_max_retries = _api_retries
```

### 3.2 Rate-limit awareness

When a provider returns 429, Hermes mines the response for `retry_after / Retry-After` and the regex-extracted "retry after X seconds" pattern (`run_agent.py:4960-4995`):

```
retry_after = headers.get("retry-after") or headers.get("Retry-After")
if retry_after and "reset_at" not in context:
    context["reset_at"] = time.time() + float(retry_after)
...
r"retry\s+(?:after\s+)?(\d+(?:\.\d+)?)\s*(?:sec|secs|seconds|s\b)"
```

It also knows when retrying *inside the same throttle window is futile* (`run_agent.py:1067-1079`):



rotation won't recover. Skip straight to the fallback for those (#13636). In those cases we must fall back to the configured `fallback_model`

That's the discipline of a system that's been broken in production many times and learned each lesson.

### 3.3 Fallback chain

`run_agent.py:1811-1820` resolves the chain:

```
# Provider fallback chain – ordered list of backup providers tried
# (failure). Supports both legacy single-dict fallback_model and
# new list fallback_providers format.
if isinstance(fallback_model, list):
    self._fallback_chain = [
        f for f in fallback_model
        if f.get("provider") and f.get("model")
    ]
elif isinstance(fallback_model, dict) and fallback_model.get("provider") and fallback_model.get("model"):
    self._fallback_chain = [fallback_model]
```

Init-time first-shot fallback at `run_agent.py:1725-1749`: if the primary provider can't even initialize a client (e.g. auth-expired Copilot), Hermes hot-swaps to the first valid fallback entry *before the first user message is processed* and sets `self._fallback_activated = True`.

This is why your fallback ladder (Copilot → Nemotron → Gemini → Kimi → MiniMax → GLM) actually works: every link is honoured on first-failure and the agent never refuses to start.

### 3.4 Near-max-iterations safety net

Lines 15240-15246 — the very end of the catch-all error handler:

```
if api_call_count >= self.max_iterations - 1:
    _turn_exit_reason = f"error_near_max_iterations({error_msg[:80]})"
    final_response = f"I apologize, but I encountered repeated errors: {error_msg}"
    # Append as assistant so the history stays valid for
    # session resume (avoids consecutive user messages).
    messages.append({"role": "assistant", "content": final_response})
    break
```

Even when everything fails inside the last turn, Hermes appends a valid assistant message so the conversation history doesn't become poisoned (`user → user` is illegal for Anthropic). That keeps the *next* user message resumable.

## 4. Tool Execution Resilience

---

### 4.1 Concurrency contract

`run_agent.py:332-359` declares an explicit concurrency policy:

```
_NEVER_PARALLEL_TOOLS = frozenset({"clarify"})

_PARALLEL_SAFE_TOOLS = frozenset({
    "ha_get_state", "ha_list_entities", "ha_list_services",
    "read_file", "search_files", "session_search",
    "skill_view", "skills_list", "vision_analyze",
    "web_extract", "web_search",
})

_PATH_SCOPED_TOOLS = frozenset({"read_file", "write_file", "patch"})

_MAX_TOOL_WORKERS = 8
```

The agent uses a 8-worker pool for parallel-safe tools and falls back to serial for any batch that touches a `clarify` (interactive) or unknown tool. That avoids data races without making the user pay for them.

### 4.2 Tool-call ID linkage

The single most common failure pattern with Anthropic + tool-calling APIs is an unmatched `tool_call_id` — the assistant `tool_call` message exists but the tool response was lost (or vice versa). Hermes hardens this with:

- An aggressive sanitizer at `_sanitize_api_messages` referenced from `run_agent.py:11509`
- An "orphaned tool result" reaper that removes `tool` messages whose `tool_call_id` doesn't match any prior assistant `tool_calls` (`run_agent.py:4484-4521`)
- A reciprocal "missing tool result" filler that synthesises stub responses when the assistant called N tools but only M results came back

The net effect: after any compaction, session resume, or stream drop, the API never sees a malformed history. Without this, every long session would eventually 400 and die.

### 4.3 OpenRouter prewarm thread leak guard

A small but telling detail at `run_agent.py:361-365`:

```
_openrouter_prewarm_done = threading.Event()
```

Comment: "Guard so the OpenRouter metadata pre-warm thread is only spawned once per process, not once per AI Agent instantiation. Without this, long-running gateway processes leak one OS thread per incoming message and eventually exhaust the system thread limit (RuntimeError: can't start new thread)."

That's the level of paranoia behind "never stops". Someone ran the gateway for three days and watched it die from thread exhaustion. The fix is one line of state.

---

## 5. Context Management & Compaction

---

The compactor lives at `agent/context_compressor.py`. The trigger predicate is `should_compress` at line 493:

```
def should_compress(self, prompt_tokens: int = None) -> bool:
```

It returns `True` when token usage exceeds `threshold_percent * context_length`. The agent reads context length from three sources in priority order (`run_agent.py:2165–2249`):

1. Explicit `model.context_length` in `config.yaml`
2. `custom_providers.<provider>.<model>.context_length`
3. Model metadata defaults via `agent.model_metadata.get_model_context_length`

Compaction is wired into `on_turn_start` (called per iteration in the loop) and on `on_session_start / on_session_end` (`run_agent.py:2373, 5580`). When triggered, it summarises the older portion of the transcript into a single system message, drops the bulk, and continues. The model never sees a `context_length_exceeded` error — Hermes guarantees the request fits before sending it.

That's the second reason long sessions don't die: they don't hit the wall.

---

## 6. Cron, Background Tasks, and Long-Horizon Survival

---

### 6.1 File-locked tick

```
cron/scheduler.py:1–9:
```

Cron job scheduler – executes due jobs.

Provides `tick()` which checks for due jobs and runs them. The gateway calls this every 60 seconds from a background thread.

Uses a file-based lock (`~/.hermes/cron/.tick.lock`) so only one tick runs at a time if multiple processes overlap.

The lock is `fcntl` on Unix and `msvcrt` on Windows (`cron/scheduler.py:21–29`). If you run two Hermes gateways accidentally (dev + cron), they don't double-fire jobs. That's how a 24/7 deployment doesn't trip over its own restarts.

## 6.2 Prompt-injection guard

Defence in depth — even the scheduler is paranoid (`cron/scheduler.py:45–55`):

```
class CronPromptInjectionBlocked(Exception):
    """Raised by _build_job_prompt when the fully-assembled prompt trips the
    injection scanner. Caught in run_job so the operator sees a clean
    "job blocked" delivery instead of the scheduler crashing.

    Assembled-prompt scanning (including loaded skill content) plugs the
    gap from #3968: create-time scanning only covers the user-supplied
    prompt field; skill content loaded at runtime was never scanned, so a
    malicious skill could carry an injection payload that reached the
    non-interactive (auto-approve) cron agent.
    """
```

A malicious skill cannot weaponise a cron job. That's table-stakes for an agent operating autonomously across days.

## 7. Gateway: the I/O Surface

`gateway/run.py` is **16,672 lines** — larger than `run_agent.py`. It exists because every messaging platform (Telegram, Discord, Signal, WhatsApp, Mattermost, Matrix, Feishu, WeCom, Yuanbao, DingTalk, Teams via msgraph) has its own webhook and delivery quirks. The directory `gateway/platforms/` has a module per platform.

For *resilience*, the relevant pieces are:

- `gateway/restart.py` (20 lines) — a clean restart entry point used after `hermes update` so the gateway can rotate its own process.
- `gateway/shutdown_forensics.py` — captures *why* the gateway is shutting down (signal, exception, exit code, last activity) and writes it to disk. So when the gateway dies, the next start knows.
- `gateway/session.py` (56 KB) — durable session storage. Sessions survive restart.

- `gateway/hooks.py` — `step_callback` infra. The agent emits an `agent:step` event per loop iteration. The gateway uses this to stream progress to the UI even mid-tool-call. (You can see this hook fire at `run_agent.py:12149`: `if self.step_callback is not None`.)

The combination of forensic shutdown + durable session + heartbeat hooks is what lets a user reload their CVC dashboard tab and resume mid-conversation. It is *exactly* what your friend's installation should be doing — and the diagnostic of "stopped mid-coding-task with no final reply" maps directly to either a missing `step_callback` hook or a broken SSE keepalive in the CVC gateway shim. (We'll dig into that next.)

---

## 8. Provider Adapters

---

Hermes ships dedicated adapters per provider — none of them generic, all of them aware of the provider's specific bugs:

Adapter	Lines	Specialised for
<code>agent/anthropic_adapter.py</code>	2,079	Anthropic Messages API + thinking blocks
<code>agent/bedrock_adapter.py</code>	1,276	AWS Bedrock auth + region routing
<code>agent/codex_responses_adapter.py</code>	1,050	OpenAI Responses API + reasoning items
<code>agent/gemini_native_adapter.py</code>	971	Google AI Studio Gemini
<code>agent/gemini_cloudcode_adapter.py</code>	(smaller)	Vertex AI / Cloud Code path

Each adapter normalises tool-call shape, retries provider-specific errors (Anthropic 529 overloaded, Gemini 503 model loading), and translates the provider's streaming format into the agent's internal chunk protocol. The agent loop above doesn't know which provider it's talking to — that's the abstraction.

When a primary provider returns a string of empty responses or persistent 5xx, the loop falls back to the next entry in `self._fallback_chain` (§3.3) and the adapter for *that* provider takes over without the loop noticing.

---

## 9. Subagent Delegation — Recursive Resilience

---

`delegate_task` spawns a child `AIAgent` with its own `IterationBudget(delegation.max_iterations)` (default 50). The parent's loop continues

immediately if the delegation is async, or blocks on a future if sync. The child's failures *cannot* kill the parent: the delegate tool wraps the child in a try/except ( [run\\_agent.py:5235](#) ):

```
# Propagate interrupt to any running child agents (subagent delegation)
```

The parent always receives a dict result `{status, summary, error, exit_reason, api_calls, duration_seconds, diagnostic_path}` — never an unhandled exception.

This is also what we just used in this very task — and even when three child agents timed out at 600s, the parent (me) kept going. That property is non-trivial and it's encoded in the very mechanism we relied on.

---

## 10. Why It Doesn't Stop — Synthesis

---

The mechanism that makes Hermes *not stop* is not one feature; it's the conjunction of:

1. **An explicit budget** — so the loop has a known horizon and the model can reason about how much work remains.
2. **A grace turn** — so it never *cliff-edges*; there's always one more chance to land a reply.
3. **A forced-summary fallback** — so even budget exhaustion produces user-facing text, not silence.
4. **Two-ring stream retries** — so transient network failures look like a 1-second pause, not a dead chat.
5. **App-owned retries with SDK retries disabled** — so the failure mode is predictable, not 3x amplified.
6. **A provider fallback chain** — so the death of one provider does not equal the death of the turn.
7. **An empty-response nudge ladder** — so models that go briefly mute get prompted back into speech.
8. **Aggressive transcript sanitisation** — so the API never sees a malformed history that would 400 the next call.
9. **Auto-compaction** — so it never hits the context wall.
10. **Forensic shutdown + durable sessions** — so when the *process* dies, the *conversation* doesn't.
11. **File-locked cron + injection scanning** — so unattended long-horizon work survives restarts and stays safe.
12. **Per-subagent budgets** — so a runaway child doesn't bring down the parent.
13. **Thread leak guards (one-shot prewarm)** — so multi-day uptime doesn't exhaust OS resources.

Take any one of these away and you have an agent that "usually" works. Stack them, and you have an agent that survives anything short of a hard kernel kill — and even then, the next start picks up the session.

# 11. Implications for CVC

---

CVC is a fork that re-routes Hermes's gateway through its own Tauri/web UI. The screenshot symptom Jai reported — "stopped automatically after some time" on a coding task — is almost always one of three things, in priority order:

1. **SSE keepalive missing on the CVC gateway shim.** Hermes emits `agent:step` every iteration (the `step_callback` hook); CVC's web gateway must forward those as SSE pings, or proxies time out the stream after ~30s of "silence" during a long tool. **Fix:** ensure the CVC gateway emits at least one chunk every 15s, even if just a `data: {"type":"heartbeat"}\n\n`.
2. **`_handle_max_iterations` summary call not surfaced.** When the parent loop hits the 90-iteration budget, the summary is generated but if the gateway already closed the stream, the user sees nothing. The agent log will show `_turn_exit_reason = max_iterations_reached(90/90)` — that's the tell.
3. **Fallback chain misconfigured.** If primary Copilot stalls and there's no valid fallback, ring 4 has nowhere to go. With the new 421 fix (v2.71.5), Business/Enterprise accounts should resolve, but verify the chain is present in CVC's effective config.

This report stands; the diagnostic plan for the gateway issue is the natural next move.

---

*End of report. Source studied: [\\_hermes\\_agent\\_ref/](#) snapshot of Hermes, 1,631 Python files, ~256MB. All citations are exact line references in that tree.*