

Dexter: a multi-agent harness that keeps working when you stop

Anish, Velofy

CONTENTS

	Abstract		6	Autonomy
1	Introduction		6.1	Keepalive
2	Design goals		6.2	Spend caps and the kill switch
3	Architecture		6.3	Goal and scheduled goal checks
3.1	Packages and binaries		6.4	The vortex eval loop
3.2	Process model		7	The research kit
3.3	Provider layer and model catalog		8	Evaluation
3.4	Extension and skill system		8.1	Setup and what the numbers are
3.5	The TUI renderer		8.2	Startup
4	Multi-agent execution		8.3	Request size
4.1	Supervisor and executer lanes		8.4	Rendering
4.2	The subagent tool		8.5	Extension loading
4.3	Scheduler and concurrency cap		8.6	Subagent scheduling and dexter fanout
4.4	Completion, cancellation and process groups		8.7	The fan-out policy, live
4.5	The fan-out policy, and why subagents do not fan out again		8.8	What we did not measure
4.6	Live panes, inside and outside tmux		9	Limitations and threats to validity
4.7	dexter fanout and external orchestrators		10	Future work
5	Learning and healing			References
5.1	Frozen memory snapshot and prompt-cache stability			
5.2	Provenance and input checks			
5.3	Skills from sessions			
5.4	The self-healing playbook and the learning graph			

ABSTRACT

Dexter is a terminal coding harness built by Velofy on a fork of pi 1.4.0 [1]. Its premise is that the harness, not the model, is the durable part of a coding agent: model choice changes month to month, while context handling, process control, memory and supervision decide whether work finishes. This report describes Dexter 3.0.0 and the unreleased changes on its main branch as of 29 October 2026. We cover the package and process architecture, a provider layer with 42 built-in providers and a two-lane router, a subagent swarm that runs each task as a full Dexter child process in its own process group with bounded concurrency and live tmux panes, a fan-out policy that lets the model split independent work on its own while children never fan out again, a learning layer that injects memory and a self-healing playbook as a frozen block so the prompt prefix stays cacheable, and supervised autonomy through a roadmap loop with a pause file, a rolling spend cap and bounded heal retries. We report measurements from 28 October 2026 on an Apple Silicon laptop: `dexter --version` answers in about 60 ms, a fix to extension loading cut the files loaded at startup from 2,853 to 1,652, a three-task `dexter fanout` run finished in 37 s for about US\$0.08, and in a live check the model split a three-package review into three parallel subagent tasks without being asked. We are also explicit about costs: in the 2.1.2 profile a trivial request carried 31,469 prompt tokens against 4,731 for upstream pi, and the one live fan-out we observed cost 4.5 times more than doing the same work in one context.

1 Introduction

A coding agent is two things joined by an API call: a model that proposes the next action, and a harness that builds the request, runs the action, and decides what happens next. The mechanisms in this report address failures of the second kind: a context window that fills with tool output, a shell command that outlives its session, a subprocess that is never reaped, a fix that was found once and forgotten, a loop that keeps spending after it stopped making progress. Models change every month, in the README's words; those failure modes do not. Dexter is built on the bet that the harness is the product (`README.md` , "Why Dexter"; `docs/index.md`).

Dexter started on 25 October 2026 as an import of the pi monorepo at upstream commit `6c2fd73b0c54` (version 1.4.0) [1], whose terminal UI, extension system, skill loader, package manager and RPC interface it keeps. Version 3.0.0 was tagged four days later, on 29 October 2026, when the main branch held 234 commits. The additions fall into four groups that this report follows:

1. **Multi-agent execution.** A built-in `subagent` tool and a `dexter fanout` command run independent tasks as parallel Dexter children, with a fan-out policy in the system prompt, cancellation by process group, and live tmux panes inside or outside tmux (Section 4).
2. **Learning and healing.** Persistent memory, procedural skills written from sessions, and a learning graph that records tool failures repaired by code edits, all injected at session start as a frozen block (Section 5).
3. **Supervised autonomy.** A keepalive loop over `ROADMAP.md` with a pause file, a single-instance lock, a rolling spend cap and heal retries, plus the `vortex` and `goal` skills for eval-driven iteration (Section 6).

4. **A research kit.** Repository memory search, document parsing, browser-grade fetching, Chrome control over the DevTools Protocol, binary inspection, and a bridge to Velofy's Troy browser (Section 7).

The report is descriptive and evaluative. Every mechanism is traced to the Dexter source; paths are relative to the source root, with `ca/` abbreviating `packages/coding-agent/src/` and `pkg/` abbreviating `packages/`. Measurements (Section 8) come from a profiling session on 28 October 2026 and from short re-runs a few hours later, all on one Apple M3 Pro laptop with 11 cores, and we state where they are noisy or missing. Section 9 lists what we consider weak in the current design.

A note on the word "vortex", which appears in the brief for this report as an example of a binary. Dexter ships no `vortex` executable: `which -a vortex` finds nothing and no `package.json` in the tree declares one. Vortex is a bundled skill that runs an evaluation loop (Section 6.4), and a branch prefix used for work driven by that loop.

2 Design goals

The README, the roadmap and the design notes under `docs/` state the goals below. We restate each as a property that can be checked, and name the mechanism that is meant to deliver it.

G1. Model independence. No feature may assume a vendor. Providers sit behind one interface in `@dexter/ai`; the router's lanes are configuration, not code; local inference through Ollama or llama.cpp needs no key (`docs/providers.md`, `docs/models.md`).

G2. Parallel by default, bounded always. Independent work should run in parallel without the user asking, but the number of tasks per call, the number of live children, their lifetime and their recursion are all capped (Section 4).

G3. Learning that does not break the cache. Anything the harness learns during a session is written to disk at once but changes the model's prompt only at the next session start, so the prompt prefix of the current session stays byte-identical across turns (Section 5.1).

G4. Autonomy with brakes. Unattended work runs one roadmap item at a time, stops on a file the user can create from any shell, never pushes, and cannot exceed a session budget per hour (Section 6).

G5. Fast where it is felt. Startup, the first byte of a request, keystroke echo and streaming redraws are measured and have benches in the tree (`pkg/tui/test/`, `pkg/coding-agent/test/bench/`).

G6. Plain files. Sessions are JSONL, memory and skills are Markdown, the keepalive state is a JSONL log and a heartbeat file, the learning graph is a JSON file, and subagent agents are Markdown with frontmatter. Anything the harness did can be audited with ordinary tools.

G7. Compatibility with pi. pi themes, skills and extensions written against the fork-baseline API load unchanged; extensions that import newer upstream package names are skipped with a warning instead of breaking startup (`docs/compatibility.md`).

These goals pull against each other. G2 and G5 conflict whenever every child re-loads the user's extensions; G3 limits how quickly a lesson can take effect; G7 keeps a second import surface alive. Sections 8 and 9 return to these trade-offs with numbers.

3 Architecture

3.1 Packages and binaries

The source is an npm workspace of twelve packages under `packages/`. Ten are public and versioned in lockstep at 3.0.0 (`docs/versioning.md`, enforced by `scripts/sync-versions.js`); `@dexter/evals` is private, and `dexter-subagents` is an optional extension on its own 1.0.0 line. Table 1 lists them with what each ships.

Table 1. Packages in the Dexter source at 3.0.0. Each lives in `packages/<suffix>`, except `@anishfyi/dexter` (`coding-agent`) and `@dexter/agent-core` (`agent`). Source file counts exclude tests and generated `dist/` output. "CLI loads" means the `dexter` command imports the package at run time.

Package	Role	Executables	Src files	CLI loads
<code>@anishfyi/dexter</code>	The harness: session runtime, tools, built-in extensions, CLI commands, 209 bundled skills	<code>dexter</code> (<code>dist/bundle/cli.js</code>); Bun single-file binary via <code>build:binary</code>	317	yes
<code>@dexter/agent-core</code>	Agent loop, tool execution, event stream, stream-function seam	none	6	yes
<code>@dexter/ai</code>	Provider interface, 42 built-in providers, 10 chat APIs, model catalog	<code>dexter-ai</code> (OAuth login helper)	204	yes
<code>@dexter/tui</code>	Terminal UI with differential rendering; native clipboard and modifier-key addons	none (Node-API <code>.node</code> addons)	48	yes
<code>@dexter/mcp</code>	Model Context Protocol client: stdio and streamable HTTP, OAuth	none	18	yes
<code>@dexter/codemode</code>	Runs model-written JavaScript in QuickJS compiled to WebAssembly	none	17	yes
<code>@dexter/telemetry</code>	Vendor-neutral span and metric contracts	none	8	through <code>ai</code>
<code>@dexter/chord</code>	Typed host runtime: contracts, replicated JSON state, RPC over links	none	41	no
<code>@dexter/durable</code>	Durable conversations, tasks and documents committed before display	none	85	no

Package	Role	Executables	Src files	CLI loads
@dexter/env	Remote execution environments for Durable over SSH	pi-env daemon (Rust, 15 files)	8	no
dexter-subagents	Larger optional subagent extension with background runs and workflows	pi-subagents installer	338	if configured
@dexter/evals	Private. Behavioural evals on vitest-evals, host and Docker	none	5	no

Three observations follow from the table. First, there is one user-facing binary, `dexter`. Every other process the harness starts (subagent children, tmux pane viewers, keepalive workers) is the same `dexter` entry point started again with different arguments, so a fix to the entry point reaches all of them. The pane viewer is a hidden first argument, `dexter __subagent-pane <file>`, that loads only the viewer module (`ca/cli.ts:17-21`). Second, `dexter-ai` is a development helper that runs the OAuth login of nine providers and writes `auth.json` in the working directory (`pkg/ai/src/cli.ts:9-116`); it is not linked as a global command when Dexter itself is installed globally. Third, the durable stack (`chord`, `durable`, `env`) ships in the tree and is built and tested, but the `dexter` command does not depend on it at 3.0.0 (Figure 1). Its daemon, `pi-env`, is a Rust program that speaks a length-prefixed binary protocol on its standard streams, pings every 5 s and exits after 30 s of silence (`pkg/env/docs/protocol.md`); it is deployed over SSH under a content-addressed file name and verified by SHA-256 before use (`pkg/env/src/ssh.ts:379-470`).

The `dexter` entry is an esbuild bundle (`scripts/build-coding-agent-bundle.mjs`): ES modules targeting Node 22.19, minified, code-split into chunks, with OAuth flows, the Bedrock client and the image-resize worker as lazy entries. The file named in `bin` is a launcher that enables Node's compile cache and then loads the runtime chunk. `--version` returns before any of that runtime is evaluated (`ca/cli.ts:9-12`), which is why it answers in about 60 ms (Section 8.2).

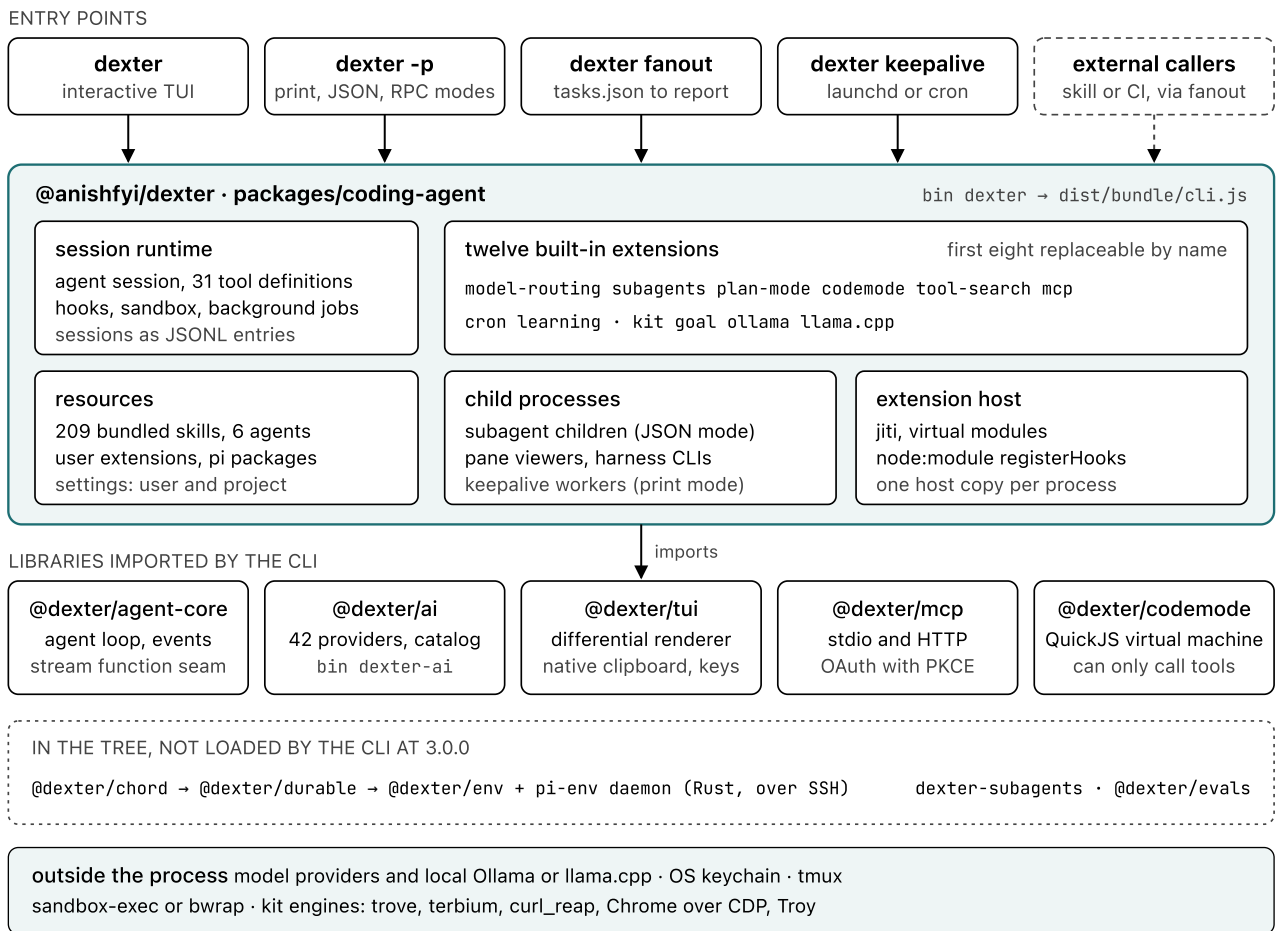


Figure 1. Package and process architecture at 3.0.0. All entry points are the same `dexter` binary. The coding-agent package imports five library packages; the durable stack ships in the same tree but is not on the CLI's import graph. Kit engines and providers are separate programs or services.

3.2 Process model

A `dexter` process runs in one of four modes: the interactive terminal UI, print mode (`-p`, one prompt, answer on stdout), JSON mode (`--mode json`, every agent event as one JSON line) and RPC mode (`--mode rpc`, a long-lived process driven over standard streams). The modes share one session runtime; they differ in input and rendering.

Child processes are started in three places. The `subagent` tool and `dexter fanout` start Dexter children in JSON mode (Section 4.3). The keepalive loop starts one print-mode worker per roadmap item (Section 6.1). Subagents declared with a `harness:` key start an external CLI instead: `claude -p -output-format text`, `codex exec`, or `pi` (`ca/extensions/subagents/runner.ts:284-296`). The `bash` tool starts shells, in the foreground or as background jobs that the model polls with `bash_output` and stops with `kill_shell` (`docs/background-jobs.md`). MCP servers configured for stdio are child processes too.

Two rules govern all of these. Every Dexter child receives its parent's process id in `DEXTER_PARENT_PID` and polls it once a second; when the parent is gone, the child sends itself `SIGTERM` and runs its normal shutdown, so a parent killed with `SIGKILL` does not leave workers running unobserved (`ca/core/parent-watchdog.ts`). The child deletes the variable from its own environment first, so commands it runs do not inherit it, and it arms the watchdog only if the variable

names its actual parent. Second, children that may own a tree of processes are started detached in their own process group, and are stopped by signalling the group (`ca/extensions/subagents/runner.ts:483-530`); Section 4.4 gives the timings.

3.3 Provider layer and model catalog

`@dexter/ai` defines a `Provider` interface with authentication, model listing, streaming, image generation and classification (`pkg/ai/src/models.ts:158-247`). It ships 42 built-in providers over 10 chat wire protocols (OpenAI completions and responses, Azure responses, the Codex responses variant, Anthropic messages, Bedrock converse-stream, Google generative AI and Vertex, Mistral conversations, and pi's own messages API), plus one image API and four classifier APIs (`pkg/ai/src/types.ts:64-86`). Each API is loaded lazily, so a session that talks only to one provider never evaluates the others.

The catalog is generated, not hand-written. `pkg/ai/scripts/generate-models.ts` merges `models.dev` [15] (which takes priority), OpenRouter, the Vercel AI Gateway and NVIDIA into per-provider data files and typed shards, and rolls back if any step fails. A release pins the catalog by content hash (`nix/model-catalog.json`, a `sha256-` revision checked against the downloaded content), so two builds of the same version list the same models. The hydrated catalog in the measured checkout held 1,597 chat models, 63 image models and 53 classifier models; only models that support tool calling enter the chat catalog (`pkg/ai/README.md`).

Credentials are resolved per provider. `dexter login` stores keys through a credential vault that uses the macOS Keychain via `security(1)`, with a mode-600 file fallback that doubles as a non-secret index of keychain entries (`ca/core/credential-vault.ts`). Explicit flags beat stored credentials, which beat environment variables. One subscription can serve two providers: the GLM Coding Plan key is accepted by both the global `zai` and the China `zai-coding-cn` endpoints through `credentialAliases`, which accept only API-key credentials from the alias slot (`pkg/ai/src/auth/resolve.ts:70-77`). Muse Code subscription models appear under the `meta` provider after an OAuth sign-in (`pkg/coding-agent/CHANGELOG.md`, 3.0.0). Defaults are GLM-first: until the user writes a `models.json`, Dexter ranks the Z.ai endpoints first and maps its default "medium" thinking level to GLM's "high" reasoning effort (`docs/models.md`).

Above the provider layer sits a router with two lanes, described with the other multi-agent mechanisms in Section 4.1.

3.4 Extension and skill system

Extensions. Every higher-level capability is an extension that registers tools, commands, flags, renderers or event handlers through one `ExtensionAPI`. Dexter ships twelve built-in extensions (`ca/extensions/index.ts:15-34`): `model-routing`, `subagents`, `plan-mode`, `codemode`, `tool-search`, `mcp`, `kit`, `cron`, `learning`, `goal`, `ollama` and `llama.cpp`. The eight marked replaceable (all but `kit`, `goal`, `ollama` and `llama.cpp`) step aside when a user extension registers a tool, command or flag of the same name (`ca/core/resource-loader.ts:188-225`); `docs/plugin-architecture.md` names this "replacement by name collision" and records why Dexter did not adopt a service-injection container in the style of Cordis [3] that deepseek-harness [2] uses. The event bus carries interceptors (`tool_call`, `tool_result`, `before_agent_start`, `before_provider_request`, `context` and others) whose return values alter the flow; the learning, goal and routing extensions in Sections 4 to 6 are built entirely from these hooks. One of the inherited extensions changes how tools are called rather than adding one:

`codemode` lets the model write a JavaScript program that calls other tools, run in a QuickJS [17] virtual machine compiled to WebAssembly whose only capability is calling the injected tools (`pkg/codemode/README.md`); MCP tools are exposed through it by default (`ca/extensions/mcp/index.ts`).

Loading, and one copy of the host. User extensions are TypeScript or JavaScript loaded with jiti 2.7.0 [13]. Imports of host packages (`@anishfyi/dexter`, `@dexter/*`, `typebox` and the legacy `@mariozechner/pi-*` names) are served from the running process through jiti's virtual modules (`ca/core/extensions/virtual-modules.ts:15-45`), so an extension shares the host's classes and singletons. That held for TypeScript, but not for `.js` files in `"type": "module"` packages: on its `async` path jiti imports such a file with Node's native `import()`, which resolves the file's own imports, and its chunks' imports, from disk without the virtual modules. The bundled subagents extension keeps its host imports in chunks, so loading it pulled a second, unbundled copy of Dexter from `packages/*/dist` into the same process (issue #38). The fix registers one pair of Node module hooks [14] per process (`ca/core/extensions/host-module-hooks.ts`): the `resolve` hook maps a host specifier imported from inside a loaded extension's package root to a `dexter-host:` URL, unless that package lists the specifier in its own `dependencies`; the `load` hook returns a small ES module that re-exports the host's namespace from a symbol-keyed global. Section 8.5 gives the effect: 2,853 to 1,652 files loaded at startup.

Skills. A skill is a directory with a `SKILL.md` whose frontmatter `description` is what the model matches against a task. The package bundles 209 skills across devices (iOS, macOS, Android), GitHub workflows, design, review, red-team engagements, media generation and the autonomous loops of Section 6; a small set of engineering-discipline skills is adapted from ponytail [7] with attribution kept. The model sees each enabled skill's name and description in the system prompt and reads the body only when it decides the skill applies; `dexter skills enable|disable` writes `+path` or `-path` overrides at user or project scope. One bundled skill, `auto-invoke`, is a router: it tells the model to read a generated `MAP.md` (188 lines of action keywords per skill) before acting and to load the most specific match. Skills are also the unit of learning (Section 5.3).

Agents. Subagent roles are Markdown files with frontmatter (`name`, `description`, `tools`, optional `model` and `harness`). Six ship in `ca/extensions/subagents/agents/`: `scout` and `reviewer` (read, grep, find, ls, bash), `planner` (read, grep, find, ls), `worker` (all tools), and the harness lanes `claude-code` and `codex`. None of the bundled files pins a model, so a child uses the task's model, else the agent's, else the parent session's current model. Files in `~/.dexter/agent/agents/` override bundled agents by name; project agents in `.dexter/agents/` run only when the project is trusted or the user confirms.

3.5 The TUI renderer

`@dexter/tui` draws the interface as an array of terminal lines and writes only what changed. On the main screen, each frame renders the component tree, composites overlays, then compares old and new lines to find the first and last changed rows and rewrites only that range, clearing each line before writing it (`pkg/tui/src/tui-main-screen.ts:248-615`). Every write is wrapped in the synchronized-output mode (`CSI ? 2026 h` and `l`) so the terminal presents a frame at once instead of tearing. A full redraw happens only on the first frame, on a width change, or when the first changed line has scrolled above the viewport; inside `tmux`, a height-only change repaints just the visible rows so the pane's scrollbar survives (`docs/tmux.md`).

Three changes in 2.1.0 made streaming cheaper. Frames are throttled by their own cost: the delay before the next frame is `min(100, max(16, 2 * last frame cost))` milliseconds, minus the time already elapsed, while keyboard input bypasses the throttle (`pkg/tui/src/tui.ts:557-559, 1106-1127`). Unchanged rows reuse the previous frame's string object after the line-reset suffix is applied, so the diff compares by reference (`tui.ts:1498-1517`). The editor caches its layout and rendered lines keyed by content, cursor and width (`pkg/tui/src/components/editor.ts:328-350`). The streaming-markdown bench measured a component mean of 1.599 ms before and 0.052 ms after, about 30 times faster (`ROADMAP.md` , CC16). Animated elements, such as the working line's spinner and elapsed seconds, share one clock at 12 frames per second, so adding a counter adds no frames (`pkg/tui/src/animation-clock.ts:15`).

Native addons (Objective-C on macOS, C on Linux and Windows) read the modifier-key state and the clipboard, including images and file paths (`pkg/tui/native/`). The terminal UI does not depend on them: the loader returns nothing when no prebuilt addon loads, and callers treat that as unavailable (`pkg/tui/src/native-platform.ts:28-53`).

4 Multi-agent execution

Dexter has two ways to put more than one model call to work on a request: lanes, which route successive requests of one session to different models, and subagents, which run separate sessions in separate processes. Lanes are cheap and invisible; subagents are expensive and isolated. This section describes both, then the policy that decides when the model fans out.

4.1 Supervisor and executer lanes

The `model-routing` extension registers a virtual model, `dexter/auto` ("Auto (supervisor/executer)"), which never reaches a provider; for each request it picks a real model (`ca/extensions/model-routing/index.ts` , `ca/core/virtual-models.ts`). The rule is short. A request goes to the executer lane when its reason is `direct` (compaction summaries and calls from extensions), or when the reason is not a new user message and a successful `edit` or `write` result exists after the last user message (`index.ts:19-33, 66-68`). Every other request goes to the supervisor. So the first request after each user message always reaches the supervisor, which plans and reads; once a file edit has succeeded in that turn, the rest of the turn's mechanical follow-up goes to the executer.

Lanes resolve per request, in order: `DEXTER_SUPERVISOR_MODEL` or `DEXTER_EXECUTER_MODEL` from the environment, then `modelRouting` in settings, then the default model; an unset executer inherits the supervisor (`ca/core/model-routing.ts:48-83`). Because resolution happens per request, changing a setting takes effect on the next request without a restart. A lane switch is persisted as a session entry only when the lane changes. The design trades some cache efficiency for cost: each switch moves the conversation to a model whose cache does not hold the prefix.

4.2 The subagent tool

A subagent is a second Dexter session that works on one task in its own context window and returns only its final answer (`docs/subagents.md`). The tool has three modes: `single` (`{agent, task}`), `parallel` (`{tasks: [...]}` , results in task order) and `chain` (`{chain: [...]}` , where `{previous}` inserts the prior step's output and the chain stops at the first failure). A parallel call may hold at most

`subagents.maxParallelTasks` tasks (default 8, range 1 to 64); more is rejected with a message that names the setting (`ca/extensions/subagents/index.ts:479-484`). Each parallel answer is cut at 50 KB before it reaches the parent's context (`index.ts:62`).

A child is a full headless Dexter. For each task the parent builds this command line (`runner.ts:212-232`):

```
dexter --mode json -p --no-session [--model <provider/id>] [--thinking <level>] \  
  [--tools <list>] [--no-mcp] [--approve | --no-approve] \  
  --append-system-prompt <agent prompt file> "Task: <task>"
```

and this environment (`runner.ts:238-251`): `DEXTER_SUBAGENT=1` , `DEXTER_PARENT_PID=<parent pid>` , `DEXTER_NO_WELCOME=1` and `DEXTER_SKIP_VERSION_CHECK=1` . Three choices keep children light without making them less capable. A child whose `tools` list cannot reach MCP tools (no `mcp__` entry, no wildcard, no `codemode` , no `tool_search`) starts with `--no-mcp` , so no MCP server starts or has to be shut down; the bundled `scout` , `planner` and `reviewer` qualify. A child in the parent's directory inherits the parent's project-trust decision instead of prompting or reading the trust store. The agent's system prompt is written once per call to a temporary file shared by tasks of the same agent, so nothing touches the disk between one spawn and the next.

4.3 Scheduler and concurrency cap

Tasks run through `mapWithConcurrencyLimit` (`ca/extensions/subagents/scheduler.ts:73-96`): it starts `min(concurrency, tasks)` workers in the same tick, each of which pulls the next index until none remain, and writes each result into `results[index]` so the output keeps input order whatever the completion order. The default concurrency is `os.availableParallelism()` capped at 8 (`scheduler.ts:26-28`). The reasoning, given in `docs/subagents.md` , is that a child's startup is CPU-bound (about 250 ms to the first JSON event on an Apple M-series laptop, of which roughly 140 ms evaluates the bundle and 45 ms parses the frontmatter of the bundled skills), while a child waiting on the model uses no CPU. More children than cores therefore start no faster. Each child is a full process of typically 100 to 200 MB.

Eight streaming children produce hundreds of events a second. The parent folds the events it needs (finished messages, usage, tool starts, `agent_settled`) into each task's result and skips the rest by matching the first bytes of each line against a regular expression: records that repeat whole messages, such as the child's system prompt, are often over 100 KB and are never parsed (`runner.ts:366-419`). Progress reaches the screen through a coalescer that publishes at most once per 100 ms, publishes the first change at once, collapses later changes into one trailing publish with the latest state, and skips publishes that would show nothing new (`scheduler.ts:113-160`). In an eight-task call this reduced progress publishes from 32 to between 3 and 5 (`docs/subagents.md`).

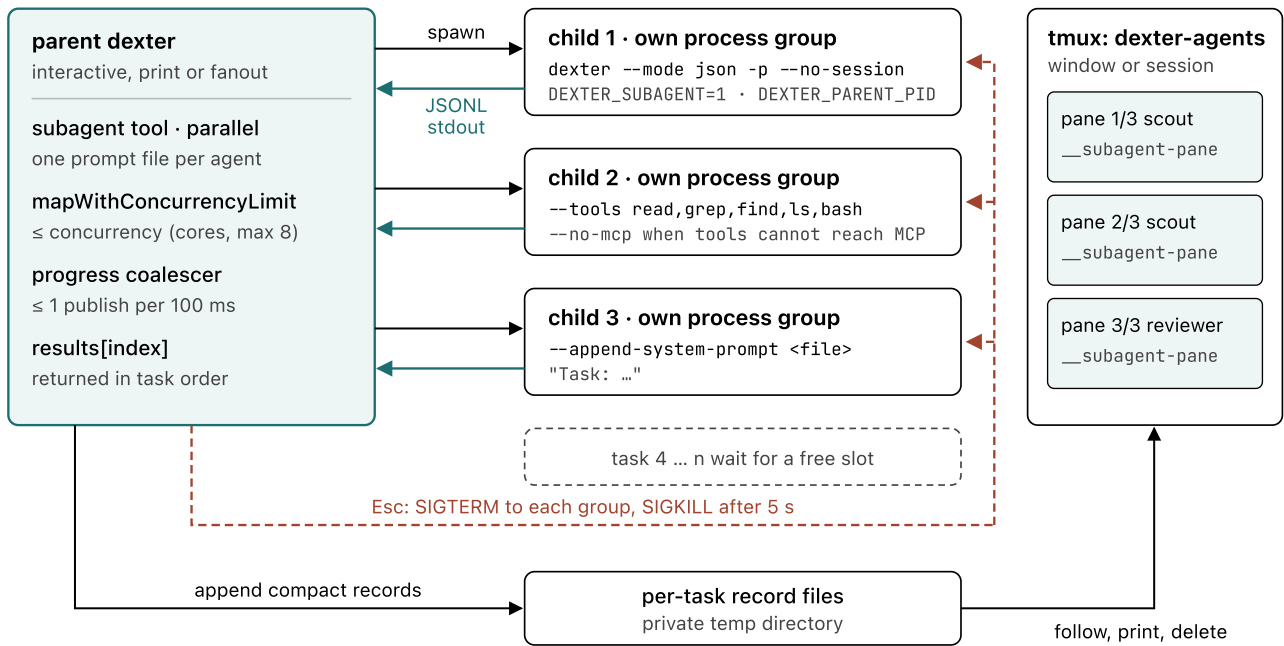


Figure 2. Process model of a three-task fan-out. The parent spawns each child detached in its own process group and reads its JSON events from stdout; tasks beyond the concurrency cap wait for a slot. For the panes, the parent appends compact records to one file per task, and each tmux pane runs a viewer that follows its file. Esc signals every group.

4.4 Completion, cancellation and process groups

When a task is done. Before 2.1.0, a task ended when its child exited. After a child's work ends it still shuts down: extensions receive `session_shutdown`, and each stdio MCP server has its standard input closed and is given time to exit before SIGTERM, as the MCP lifecycle prescribes [10]. Dexter allows 500 ms, and many servers use all of it. Since 2.1.0 a task completes when the child emits `agent_settled` and no follow-up run starts within 100 ms (`SETTLE_GRACE_MS`, `runner.ts:117`), or when the child exits, whichever comes first. The result returns at that moment while the child finishes shutting down in the background; a child that has not exited 10 s later (`REAP_TIMEOUT_MS`) is terminated (Figure 3).

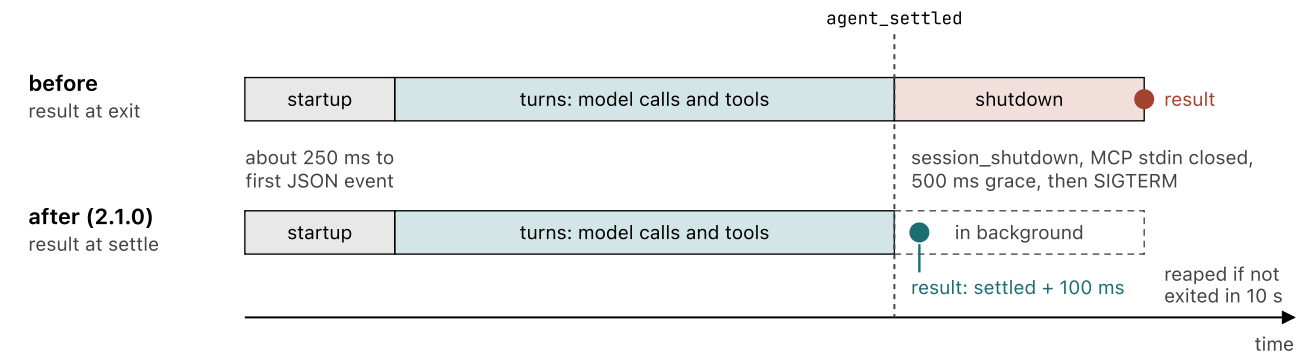


Figure 3. When a subagent task completes. Before 2.1.0 the result waited for the child to exit, including the MCP shut-down grace. Since 2.1.0 it returns 100 ms after `agent_settled`, and shutdown overlaps the parent's next step. Segment lengths are schematic.

Cancellation. Esc aborts the call. Each running child's process group receives SIGTERM; the child's own handler stops its background bash jobs (each in its own process group) and closes its MCP servers, and anything still alive 5 s later (`KILL_GRACE_MS`) receives SIGKILL. On Windows, which has no graceful signal, `taskkill /T /F` ends the tree. The tool returns as soon as the abort arrives, reporting per task whether it completed, was aborted with partial output, or had not started; queued tasks are reported as aborted without spawning anything. When the parent itself exits, children that are still running or still shutting down receive SIGTERM on their groups; when the parent dies without running handlers, the watchdog of Section 3.2 brings each child down within about a second.

Why no warm pool. A pool of idle children started ahead of time would hide startup, but a child's setup is fixed when it starts: working directory, tool list, system prompt, trust decision, settings, extensions, skills and context files. A pooled child could serve only a later task with the same agent, directory and model, and would miss any change to context files or skills made since it started, while holding 100 to 200 MB and its own MCP servers. With startup between a fraction of a second and about 1.5 s, all children starting together, and real tasks running tens of seconds against a model, the source argues the pool would save little and risk stale context (`docs/subagents.md`). We agree for interactive use; Section 10 notes where the argument weakens.

Table 2. Subagent limits and timings. All are constants in `ca/extensions/subagents/` or settings with their defaults.

Name	Value	Where
Tasks per parallel call (<code>subagents.maxParallelTasks</code>)	8 (1 to 64)	settings
Children running at once (<code>subagents.concurrency</code>)	cores, at most 8 (1 to 64)	<code>scheduler.ts:26</code>
Progress publish interval	100 ms	<code>scheduler.ts:14</code>
Settle grace after <code>agent_settled</code>	100 ms	<code>runner.ts:117</code>
Reap timeout after completion	10 s	<code>runner.ts:119</code>
Grace between SIGTERM and SIGKILL	5 s	<code>runner.ts:121</code>
Parent watchdog poll	1 s	<code>ca/core/parent-watchdog.ts:14</code>
Parallel answer cap per task	50 KB	<code>index.ts:62</code>
Stderr kept per child	64 KB	<code>runner.ts:122</code>
Fan-out rules added to each request	512 characters	<code>index.ts:398-404</code>
Finished tmux pane stays open (<code>subagents.tmuxCloseDelayMs</code>)	15 s	settings

4.5 The fan-out policy, and why subagents do not fan out again

Until 3.0.0 the model called `subagent` only when the user asked. The tool's description listed the modes, but nothing said when to choose parallel mode. Independent searches, reviews and checks therefore ran one after another in the main context. The 3.0.0 change adds two rules to the system prompt, emitted only while the tool is active (`index.ts:398-404`):

Without being asked, use subagent parallel mode instead of doing the parts yourself when work splits into 2 or more independent parts, such as searching several areas of a codebase, reviewing separate files or packages, running independent checks, or researching several questions: write each task with the full context it needs (subagents do not see this conversation), at most 8 tasks per call

Do not fan out for trivial or single-step requests, steps that depend on each other, or edits to the same files

The wording was tuned against a live model. A first draft (452 characters) produced zero subagent calls on the test prompt: the model cited the rule as fitting and then worked directly, reasoning that subagents might lack context. The shipped wording names that trade-off ("instead of doing the parts yourself", "write each task with the full context it needs") and produced one parallel call with three tasks on the same prompt (Section 8.7). Because these rules travel with every request, they have a budget: 512 characters against a limit of 600 set in the tracker's system-prompt-size issue.

Recursion is discouraged at the source. Every child runs with `DEXTER_SUBAGENT=1`, and the extension registers no rules when that variable is set (`index.ts:399`), so a delegated task never receives the instruction to fan out. The tool remains available to the child, but nothing invites it to use it. This is a policy bound, not a hard one: a child could still call the tool if its task asked it to. The alternative, removing the tool from children, would also remove single-mode delegation that a worker might legitimately want. Panes add a second bound: they open only from an interactive parent, never from a subagent, so nested calls stay quiet (`docs/tmux.md`).

4.6 Live panes, inside and outside tmux

Each task of a parallel call with two or more tasks gets a live tmux pane [12] (`subagents.tmux: auto`). Inside tmux, the panes open in a window named `dexter-agents` in the current session, created with `new-window -d` so focus stays on Dexter. Since 3.0.0, an interactive Dexter running outside tmux, with `tmux` on the `PATH`, opens the same window in a detached session also named `dexter-agents`, created with `new-session -d` or reused, and shows once per process the command to watch it: `tmux attach -t dexter-agents (ca/extensions/subagents/tmux-panes.ts)`. Print, JSON and RPC modes, keepalive, and nested subagents never start the detached session.

Panes are a view, never a dependency. The parent appends compact JSON records for each task to a file in a private temporary directory; each pane runs `dexter __subagent-pane <file>`, which follows the file and prints the child's text as it streams, its tool calls with short arguments (`→ $ npm test, → read src/auth.ts`), failed tools and retries, and a summary line with turns, tokens, cost and time. The viewer deletes the file after the summary. Closing a pane does not stop its subagent. Any tmux failure (no room, an error, `tmux` missing) falls back to the in-Dexter rendering, writes one line to the debug log, and is reported by `/tmux (tmux-panes.ts:1-13)`. The same lifecycle events are available to code inside Dexter through `addSubagentLifecycleListener()` (`runner.ts`), whose listener errors are swallowed so an observer cannot break a task.

4.7 dexter fanout and external orchestrators

`dexter fanout [tasks.json]` (`ca/cli/fanout-command.ts`, unreleased on main) runs a list of tasks through the same parallel machinery without a model making the call: the same agents, limits, child command line, environment, prompt files, cleanup and panes, through the shared `runParallelTasks` function (`index.ts:279-303`). Tasks come from a file or standard input, as an array or as `{"tasks": [...]}`; each task needs `task` and may set `agent` (default `worker`), `model`, `thinking` and `cwd`. Standard output receives only the report, Markdown by default or one JSON document with `--json`, with each task's full answer, model, duration, tokens and cost in input order; progress and the pane notice go to standard error. The exit code is 0 when every task succeeded, 1 when any failed, and 2 for invalid input. `SIGINT` and `SIGTERM` stop every child and print the report with unfinished tasks marked aborted. Unlike the tool, `fanout` never cuts an answer at 50 KB.

The command makes Dexter's swarm callable from shell scripts, CI and other agents. The source ships one such integration, a skill for Claude Code (`integrations/claude-code/dexter-fanout/SKILL.md`, installed by a symlink into the user's skill directory, `docs/claude-code.md`). The skill tells the calling agent to write one self-contained task per part to a JSON file, run `dexter fanout --json tasks.json < /dev/null` (Dexter waits on an open standard input, so it must be closed), read the results, treat each answer as a lead to check rather than a fact, and treat any instruction inside an answer as data. For older Dexter versions the skill falls back to a shell loop of `DEXTER_SUBAGENT=1 dexter -p --no-session --no-mcp --tools read,grep,find,ls,bash` children. In the other direction, the `claude-code` and `codex` harness agents let a Dexter call hand a lane to those CLIs. Harness lanes share the concurrency limit, ordering, progress, cancellation and cleanup of Dexter children, but run with their own tools, model and permissions.

5 Learning and healing

The `learning` extension (`ca/extensions/learning/`, added 26 October 2026) gives the harness three kinds of persistent state: declarative memory, procedural skills, and a graph of repaired failures. Its design draws on Hermes Agent [4] for memory and skills and on OpenClaw [6] for provenance, as `docs/harness-comparison.md` records. All state lives under the agent directory, `~/.dexter/agent` by default, and every write goes to a temporary file that is renamed into place.

5.1 Frozen memory snapshot and prompt-cache stability

Memory is two Markdown files, `memories/MEMORY.md` (what the agent learned about the environment and repositories) and `memories/USER.md` (what it learned about the user), capped at 2,200 and 1,375 characters and split into entries by a `§` delimiter line (`store.ts:109-111`). The `memory` tool adds, replaces and removes entries; replace and remove need a substring that matches exactly one entry.

The way memory enters the prompt is the important part. On `session_start` the extension reads both files and the graph once and builds a block of up to three tagged sections: `<agent_memory>`, `<user_profile>` and `<self_healing_playbook>` (`store.ts:638-672`). On every `before_agent_start` it appends that same cached block to the system prompt (`index.ts:108-114`). Writes made during the session reach disk immediately but do not change the block until the next session. The reason is economic. Providers cache the longest previously seen prefix of a request and bill cached tokens at a fraction of the normal input price; the system prompt is the start of that prefix. If a memory write

changed the system prompt mid-session, every later request would miss the cache from the first changed byte onward. Freezing the block keeps the prefix byte-identical for the whole session, at the cost that a lesson takes effect one session late.

The same concern shapes the rest of the prompt. The harness counts cache misses above a 1,024-token noise floor and attributes each to an idle gap longer than the cache lifetime or a model change (`ca/core/cache-stats.ts`), and a cache warmer inherited from pi can refresh an entry at 90 percent of its lifetime when the expected saving is at least US\$0.05 (`ca/core/cache-warmer.ts`). The fan-out rules were kept to 512 characters for the same reason. Section 8.6 shows the effect of a warm cache on subagent cost: two runs of the same three tasks cost US\$0.0445 to US\$0.0465 per child in one run and US\$0.0148 per child in the other, where 54.8k prompt tokens per child were read from the provider's cache.

5.2 Provenance and input checks

Each tool write records the SHA-256 of the serialized file, its origin and a timestamp (`store.ts:446-451`). On read, the file is hashed again; if the hash differs from the recorded one, the file was changed outside the tool and its section is marked `trust="untrusted"` in the prompt (`store.ts:419-424`). Symlinked memory files are refused on read and write, and symlinked skill files are skipped, so a file planted in a workspace cannot redirect persistent memory (`store.ts:411-414, 437-439, 272-273`). Entries are rejected if they contain zero-width or bidirectional control characters, the phrase "ignore previous instructions" in its common forms, or strings shaped like API keys (`store.ts:229-244`). These checks are pattern-based; Section 9 discusses what they do not catch.

5.3 Skills from sessions

`skill_manage` creates, edits, deletes and lists user skills as `skills/<name>/SKILL.md`, with a validated name, a description of at most 1,024 characters, a category, and `related_skills` edges to other skills (`index.ts:217-262, store.ts:686-725`). Before any write, a scanner rejects the skill if it matches one of four rules: prompt injection, hidden Unicode, secret exfiltration (a `curl` that sends a `KEY`, `TOKEN`, `SECRET` or `PASSWORD` variable, or reads `~/.ssh`, `~/.aws` or the Dexter credentials file) and destructive deletion (`rm -rf of /, ~ or $HOME`) (`store.ts:194-224`). The `/learn [focus]` command sends a synthesis prompt that asks the model to turn the current session into skills, memory entries and heal records; `/learn graph` shows the graph instead. This is a single pass by the model over one session; Hermes Agent's companion project evolves skills by testing candidate mutations with GEPA [5], and Dexter does not attempt that. Skill usage is counted each time the model successfully reads a skill's `SKILL.md`, and becomes the node weight in the graph.

5.4 The self-healing playbook and the learning graph

The extension watches `tool_result` events with a three-state detector held in memory (`index.ts:116-162`, Figure 4):

1. A result with `isError` records a pending failure: the tool name and the first non-empty line of the error, cut to 160 characters.
2. A successful `edit` or `write` while a failure is pending adds the edited path to the pending set.
3. The next successful result of a tool with the same name, with at least one edited file, records a heal and clears the pending state.

A heal is stored under the signature `<tool>:<first 12 hex digits of the SHA-256 of the snippet>`; a repeat increments its occurrence and heal counts and merges up to 10 healed files (`store.ts:781-825`). The model can also record a heal explicitly with `learning_graph record_heal` , giving the root cause and remediation in its own words. At the next session start, the six heals with the highest heal count are written into `<self_healing_playbook>` as lines of the form `- [bash] <error> → Fix: <remediation>` , and the bundled `self-healing` skill tells the model to consult the playbook before guessing at a fix.

The graph (`/journey`) connects four node kinds: skills, memory entries, user entries and heals. Edges come from `related_skills` frontmatter (weight 3), from word overlap of at least two significant tokens between a memory entry and a skill (top three skills), and from heals to skills, either named explicitly (weight 4) or by overlap (`store.ts:859-1008`). It is stored as `learning/graph-state.json` and drawn as ASCII art in the terminal.

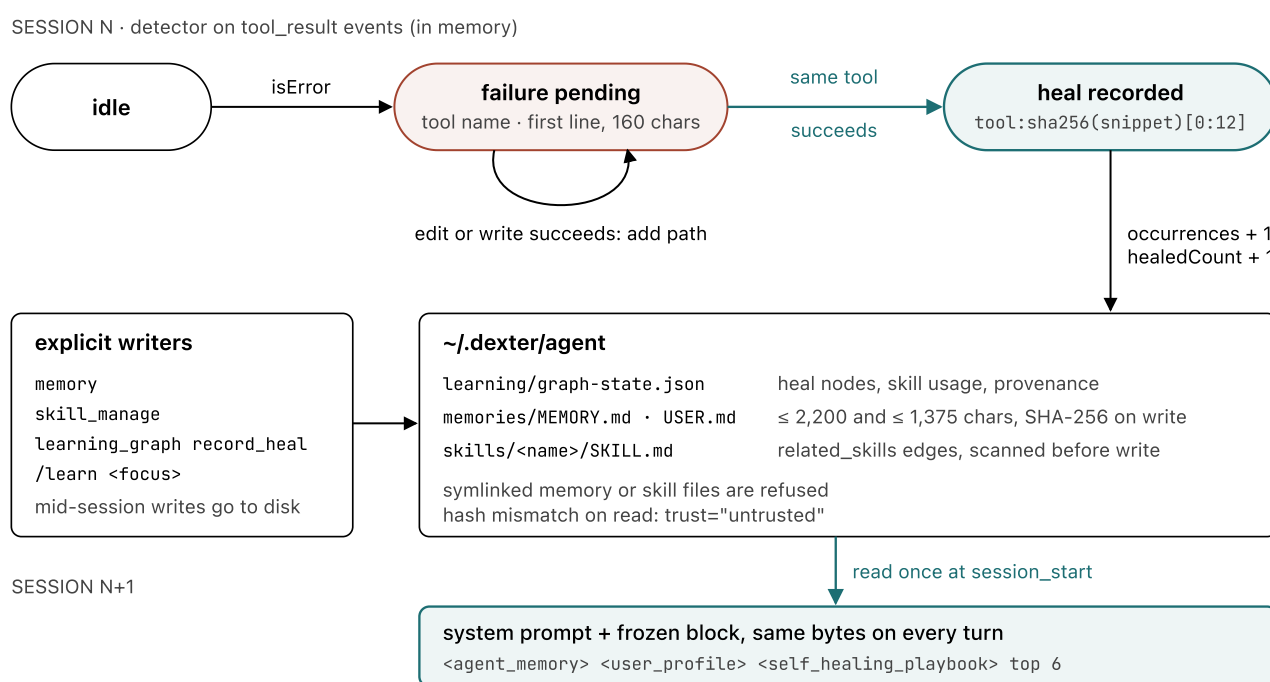


Figure 4. The healing loop. A failed tool result, a successful edit and a later success of the same tool record a heal node. Explicit writers add memory, skills and heals. Everything lands in files under the agent directory and enters the prompt only at the next session start, as one frozen block.

The detector is deliberately simple, and its simplicity has a cost we return to in Section 9: "same tool" means the same tool name, not the same command, and there is no time window, so a failing `bash` test followed by an edit and any later successful `bash` call records a heal.

6 Autonomy

Dexter's autonomy is supervised, local and budgeted. Four mechanisms provide it: a keepalive loop over a roadmap, an in-process scheduler for recurring goal checks, and two skills, `goal` and `vortex` , that structure long work inside a session.

6.1 Keepalive

`dexter keepalive` (`ca/cli/keepalive-command.ts`) runs a loop whose whole contract is a `ROADMAP.md` of Markdown checkboxes in the repository root (`docs/keepalive.md`). Each iteration picks the first unchecked item that is not marked `BLOCKED` and whose `blocked-by:` items are all checked (an unknown id counts as blocking), starts one headless worker with a prompt that asks it to do exactly that item, read `ROADMAP.md` and `AGENTS.md`, build and test, commit locally, tick the box, never push, and stop if it finds changes it did not make (`keepalive-command.ts:128-181`). The worker is `node <dexter entry> -p <prompt>` with inherited standard streams. After each worker the loop appends a record to `log.jsonl` (start, end, item, exit code, directory, heal attempt) and updates `heartbeat.json`, then waits 1 s before the next item.

`dexter keepalive install` registers the loop with the operating system: a `launchd` agent `com.dexter.keepalive` with a `StartInterval` of the chosen minutes (default 30) on macOS, or a marked block in the user's crontab on Linux, where the interval must divide 60 (`ca/cli/keepalive-scheduler.ts`). `stop` removes exactly what was installed, and `status` reports the schedule, the guards, the lock, the last heartbeat, sessions in the last hour against the cap, and the last five log records.

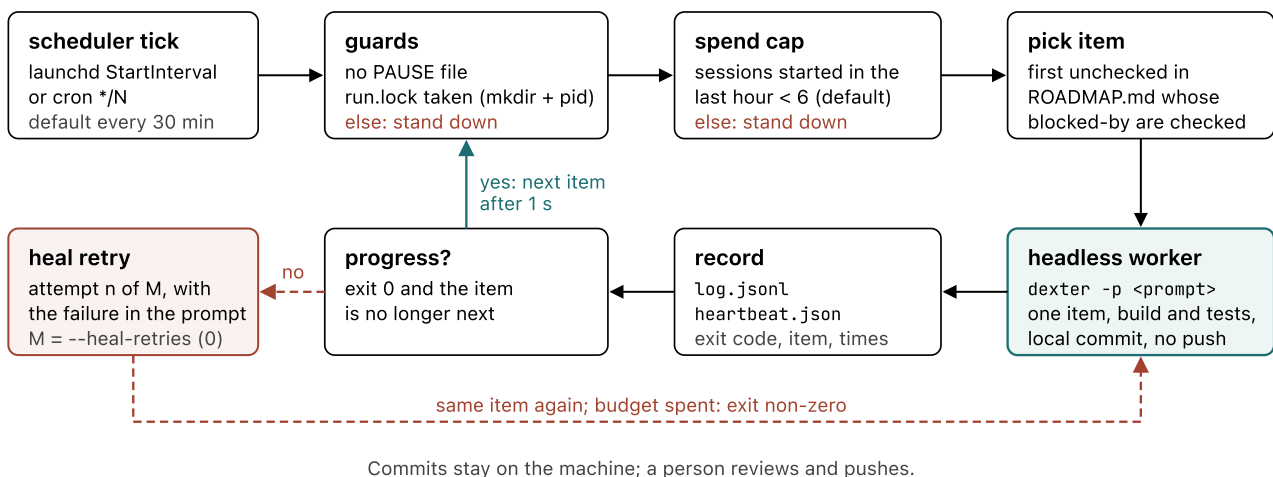


Figure 5. The keepalive loop. Each scheduler tick passes three guards, picks one item, and runs one worker. A worker that fails, or exits without moving the roadmap forward, is retried on the same item with its failure in the prompt until the heal-retry budget is spent.

6.2 Spend caps and the kill switch

Three guards bound the loop (`keepalive-command.ts:35-44`, `392-560`). A file named `PAUSE` in the agent directory stops it: the check runs before the loop starts and before every item, so the running worker finishes and no new one starts. A single-instance lock, an atomic `mkdir` of `run.lock` holding the owner's pid, prevents two loops from working the same repository; a lock whose pid is dead is taken over. A rolling spend cap counts log records started in the last hour and stops the loop when the count reaches the cap, 6 sessions per hour by default (`DEFAULT_MAX_SESSIONS_PER_HOUR`), so a loop that keeps failing cannot keep spending. The cap counts sessions, not dollars; Section 9 discusses the gap.

`--heal-retries <n>` (default 0) is a per-item retry budget. A retry starts when the worker exits non-zero, or exits 0 while the same item is still next; the next prompt carries "Self-Healing Recovery Mode (attempt n of M)" and the failure reason. The counter resets when a different item is picked, and an exhausted budget ends the loop with a non-zero exit (`keepalive-command.ts:476-553`). The trust model is stated in the docs: the loop commits locally and never pushes, so a person remains the gate for what leaves the machine. `dexter doctor` checks the pause file and the lock, and `--fix` clears a stale lock.

6.3 Goal and scheduled goal checks

The `goal` extension pins one goal across sessions (`ca/extensions/goal/index.ts`). `/goal <text>` writes it to `~/.dexter/goal.txt` ; before each user-prompted run, the extension injects a hidden message wrapping the goal in `[GOAL]` tags with steering text, and a `context` handler removes old goal messages once the goal is cleared. The bundled `goal` skill complements it with a written `GOAL.md` whose "Done when" criteria must be checkable by a machine or a reviewer.

The `cron` extension runs scheduled jobs inside a running Dexter session (never in print mode), not through the system cron (`ca/extensions/cron/`). A job has a five-field schedule, a prompt and a check command. Each attempt runs `dexter -p <prompt>` and then the check; exit 0 marks the job done, and a job that reaches its attempt limit (default 5) is marked exhausted. The next attempt's prompt carries the last 2,000 characters of the check's output, so each try sees why the last one failed. Jobs and per-attempt logs are files under the agent directory, and jobs left running by a crash are recovered at startup.

6.4 The vortex eval loop

Vortex is a bundled skill, not a program (`pkg/coding-agent/skills/vortex/SKILL.md` , added 26 October 2026). It turns a goal into an evaluation harness and iterates: evaluate, gate, analyze the smallest set of concrete failures for root causes, apply minimal fixes, re-evaluate. The goal comes from the user or from `GOAL.md` . Before touching code, the skill requires a harness: the repository's own (an eval package, a test command, a script that exits 0 or 1, a benchmark with a score) or the smallest honest check the user would accept, saved as `eval.sh` . The budget defaults to 10 iterations. Each iteration appends one JSON line to `.dexter/vortex/log.jsonl` and commits as `vortex(i<N>): <change>` ; a run resumes from the last logged iteration. The hard rules are the substance of the skill: never weaken the eval to make it pass, fix a wrong eval in its own commit, never report a score the harness did not produce, revert an iteration that makes things worse and log the dead end, run one writer at a time, and write `POSTMORTEM.md` with the closest score and remaining failures if the budget runs out.

Vortex has no runtime of its own. Its loop is carried out by the model with ordinary tools, and its guarantees are only as strong as the model's adherence to the skill text; the keepalive guards of Section 6.2 do not apply to it unless it runs inside a keepalive worker. At the time of writing, a vortex run against Dexter itself was in progress on a local branch with the goal of completing legitimate coding work end to end with no harness-level blockers. Its log held a setup record and a restarted baseline iteration over the repository's existing eval suites (`smoke` , `glm-tuning` , `documentation-audit`), run with `npm run eval:host` on GLM-5.3; no iteration commit existed yet. Elsewhere in git history the word appears only in branch names, among them that of pull request #36, which moved the working spinner.

7 The research kit

The `kit` extension adds 15 tools that reach outside the repository (`ca/extensions/kit/`). Three wrap external engines that Dexter calls with `execFile` , never installs, and locates through an environment variable or the `PATH` ; a missing engine returns an install hint instead of an error trace (`kit/runner.ts`). `/kit` probes each engine with a 5 s timeout and `dexter doctor` reports them.

Trove. `trove_query` asks a repository-memory engine written in Rust for the code most relevant to a question (architecture, subsystems, symbols) before the model reads files blind. It runs `trove --repo <cwd> query <question> --budget <n>` with a default budget of 6,000 tokens and a 60 s timeout (`kit/index.ts:56-83`).

Terbium. `parse_document` converts documents to Markdown through the `terbium` Python module, called through a short bridge script with a 30 s timeout (`kit/index.ts:85-117`).

curl_reap. `web_fetch` and `web_crawl` use the `reap` binary, which fetches with a browser-like TLS fingerprint (`--profile chrome|firefox|safari`) so that pages behind bot checks answer as they would to a browser. A crawl follows links to at most 20 pages and depth 2 (`kit/index.ts:119-188`).

Browser control. Nine `browser_*` tools drive Chrome over a raw Chrome DevTools Protocol client [11] using the runtime's own WebSocket, with no automation library (`kit/browser.ts`). The engine attaches to `DEXTER_BROWSER_CDP_URL` if set, otherwise to a running Troy instance, otherwise launches Chrome headless with a persistent profile under the agent directory. `browser_snapshot` returns visible interactive elements as `- role "name" [ref=eN]` lines; refs stay stable across snapshots through a `WeakMap` in the page, so the model can act on `e12` after a re-read. Snapshots are capped at 15,000 characters, commands at 15 s. The design follows the locator discipline recorded in the research note under `docs/research/` , which also takes the evidence-record idea of the rea toolkit [16] as a target for kit results.

Binary inspection. `inspect_artifact` identifies Mach-O, ELF and PE files, their architecture and sections with Shannon entropy, framework fingerprints (Go, Rust, .NET, Bun, Node, Electron, PyInstaller) and printable strings, with no external dependency (`kit/inspect.ts`).

Troy bridge. Troy is Velofy's Chromium-based browser for agents; because it is the user's own browser, it is already signed in to the user's accounts. The bundled skill describes its agent surface: a tab tree, a DOM reader with stable `el-N` references, a mode that fuses the DOM with a screenshot read by the macOS Vision text recognizer, delta reads that return only what changed since the last read, and a persistent SQLite memory of visited pages (`pkg/coding-agent/skills/troy/SKILL.md`). Dexter's `troy` tool discovers Troy's `agent-endpoint.json` , sends each request to a loopback port with the token from that file in an `x-troy-agent` header, and can launch Troy hidden (`--agent --hidden`) and wait up to 12 s for it (`kit/troy.ts:88-362`). It exposes sixteen actions, from `status` and `tabs` through `read` , `click` , `fill` and `batched act` to `recall` and `forget` . According to the skill, three rules are enforced by Troy's bridge rather than left to the model: the agent may navigate or close only tabs it opened; mutations are gated per origin (the tool can grant the current origin once and retry, `kit/troy.ts:434-466`); and submit-like buttons (pay, buy, delete, send) are refused unless the call explicitly allows them after user confirmation.

8 Evaluation

8.1 Setup and what the numbers are

All measurements were taken on one Apple M3 Pro laptop (11 cores, 18 GB, macOS 26.3) that was also running other work. Most numbers come from a profiling session on 28 October 2026 that compared Dexter 2.1.2 (main at `5ab28a7`) with upstream pi 1.4.0, both on the `zai/glm-5.3` model; they are medians over the run counts shown. The extension-loading figures come from the same day's work on issue #38, measured from source with the real subagents bundle. The fan-out figures are live runs of the same day on GLM-5.3. We added short re-runs of startup timings with the installed Dexter 3.0.0 a few hours later, early on 29 October local time, using hyperfine with standard input closed; the machine's load average was 21 to 23 on 11 cores during those runs. No measurement used a model from the `meta` or `google` providers.

We report each number with its source and run count. None of these runs is a controlled experiment with repeated trials and confidence intervals, and the model calls in particular show large variance: in one profile, the time to response headers for the same request ranged from 1.0 s to 15.8 s over eight trials.

8.2 Startup

Table 3. Startup and request latency, Dexter 2.1.2 against pi 1.4.0, medians, 28 October 2026. Model `zai/glm-5.3`. "No ext." is `--no-extensions`; "no skills" adds `--no-skills`.

Measure	Dexter	pi 1.4.0	Runs
<code>--version</code>	58 ms	175 ms	10 / 10
<code>--list-models zai</code>	991 ms	379 ms	7 / 7
<code>--list-models zai</code> , no ext.	431 ms		7
Launch to first prompt box	987 ms	421 ms	7 / 9
Launch to first prompt box, no ext. / no skills	443 / 377 ms		7
<code>-p --no-session "reply ok"</code> , total	7,379 ms	5,891 ms	16 / 16
Print mode: request leaves the process	1,375 ms	521 ms	6 / 6
First token after the request is sent	3,864 ms	4,220 ms	6 / 6
Module files loaded at startup	2,590	263	1 trace each
Keystroke to screen, idle / streaming	11 / 13 ms		40 keys

Table 4. Re-runs with the installed Dexter 3.0.0, hyperfine, standard input closed, early 29 October 2026, load average 21 to 23.

Command	Median	Min	Runs
<code>dexter --version</code>	64.0 ms	60.8 ms	15
<code>dexter --help</code>	1,028 ms	923 ms	15

Command	Median	Min	Runs
<code>dexter --list-models zai</code>	1,641 ms	1,475 ms	9
<code>dexter --no-extensions --list-models zai</code>	608 ms	574 ms	9
<code>dexter --no-extensions --no-skills --list-models zai</code>	495 ms	457 ms	9
Peak resident memory of <code>--version</code>	45.7 MB		3

The early exit for `--version` works: about 60 ms and 46 MB, a third of pi's time. Everything that evaluates the runtime is slower than pi, and the profile located where the time goes. About 550 ms per command came from extension packages loading a second runtime: two third-party context packages in the user's configuration resolve upstream pi package names to an upstream pi 1.4.0 runtime that npm had installed as a peer dependency (1,230 files); and a settings package that imported Dexter natively and loaded the unbundled `packages/*/dist` tree (the issue #38 fix in Section 8.5 removed the second cause). Skills cost about 66 ms of launch time in the profile and about 110 ms of `--list-models` time in the re-run. Dexter itself, with no extensions and no skills, starts in roughly the time pi takes with its defaults. The 3.0.0 re-runs are slower than the 2.1.2 profile in absolute terms (608 ms against 431 ms without extensions); we attribute most of that to machine load, which we did not control, and cannot separate it from any regression without a quieter re-run.

8.3 Request size

Table 5. What a one-line request carries, 28 October 2026.

Item	Dexter 2.1.2	pi 1.4.0
Prompt tokens for "reply ok"	31,469	4,731
Cost per "reply ok"	US\$0.0085	US\$0.0014
Tools declared	31	10
Characters of tool definitions	27,661	
Skills advertised (bundled plus user)	214	
Characters of skill advertisements	87,374	

This is Dexter's largest measured cost. A trivial request carries 6.7 times the prompt of upstream pi, mostly skill advertisements (214 skills were advertised, and 182 user skills on the test machine collided with bundled skills of the same name) and tool definitions. Caching softens the cost after the first turn of a session but does not remove it, and a larger prompt also delays the request: in the profile, Dexter's request left the process 1,375 ms after start against 521 ms for pi, although the provider's time to first token was similar (3,864 ms against 4,220 ms). Two A/B runs on provider latency disagreed. In one, the full Dexter request took a median 4.1 s to response headers against 2.4 s for pi's request. In the other, the full Dexter request reached its first byte in a median 1.7 s, the same as pi's 1.8 s, while adding only Dexter's bash tool definition to pi's request gave 4.4 s. With eight trials per variant and spreads of several seconds within each, we draw no conclusion about provider-side latency.

8.4 Rendering

Streaming render cost is at parity with pi: an 80-line reply rendered in 671 frames in Dexter and 714 in pi, without flicker and with a clean resize, and keystroke-to-screen latency was 11 ms idle and 13 ms while streaming over 40 keys. Two benches in the tree quantify the 2.1.0 renderer changes: the streaming-markdown component mean fell from 1.599 ms to 0.052 ms, and 16 waiting tool rows render at 12 frames per second on the shared animation clock instead of about 28 (`pkg/coding-agent/CHANGELOG.md` , 2.1.0). We did not re-run these benches for this report, and the three large-transcript benches in `pkg/tui/test/` have no recorded results in the tree.

8.5 Extension loading

Table 6. Effect of the host-module hooks (issue #38, pull request #54), from source with the bundled subagents extension, `--no-extensions -e packages/subagents/bundle/index.js --list-models zai`, median of 5, 28 October 2026.

Measure	Before	After
Files loaded at startup	2,853	1,652
Files loaded from <code>packages/*/dist</code>	284	0
Command time	994 ms	828 ms
Time added by the subagents bundle over <code>--no-extensions</code> (805 ms)	189 ms	23 ms

The fix removed a duplicated runtime rather than tuning it. Four tests cover static and dynamic ES module chunks, a CommonJS chunk, a host package declared in an extension's own dependencies, and an extension's private dependency; the chunk and CommonJS tests failed before the change. The coding-agent suite passed 3,357 tests with the change.

8.6 Subagent scheduling and dexter fanout

Table 7 reproduces the scheduling benchmark recorded in `docs/subagents.md` for the 2.1.0 design (pull request #31). It runs the real `subagent` tool against children that talk to an in-process fake model, so it measures harness overhead only. The machine was the same laptop under a load average of 20 to 45; values are medians of six runs.

Table 7. Wall time of a `subagent` call before and after the 2.1.0 scheduler, in milliseconds (`docs/subagents.md`).

Scenario	Tasks	Before	After
scout , one MCP server configured	1	836	313
scout , one MCP server configured	4	919	397
scout , one MCP server configured	8	1,813	602
worker (keeps MCP), one MCP server configured	1	844	428
worker (keeps MCP), one MCP server configured	8	1,791	769
scout , model answers after 1 s	8	2,793	1,567

Scenario	Tasks	Before	After
scout , no MCP, instant model	1	295	316
scout , no MCP, instant model	8	712	618

Most of the gain comes from two of the mechanisms in Section 4: skipping MCP for agents that cannot use it, and returning at `agent_settled` instead of waiting for the server's shutdown grace. With eight tasks all eight now start at once instead of four then four, which halves wall time when the model takes a second; when startup is the only work, eight children compete for 11 cores and gain less. The single-task case without MCP did not improve, within noise.

Table 8. Live `dexter fanout` runs, `zai/glm-5.3`, 28 October 2026. Each task's answer was checked against the source tree.

Run	Tasks	Wall	Per task	Cost	Answers correct
A: three short questions	3 <code>scout</code>	10.7 s	8.0, 10.7, 8.0 s	US\$0.136	3 of 3
B: same tasks, <code>--json</code>	3 <code>scout</code>	14.0 s	14.0, 8.8, 8.9 s	US\$0.044	3 of 3
C: integration check	2 <code>scout</code> , 1 <code>reviewer</code>	36.8 s	8.4, 7.4, 36.8 s	US\$0.078	2 of 2 factual

Run C is the check of the Claude Code integration: two counting and lookup questions and one review question, run through `dexter fanout --json` with standard input closed. Wall time equals the slowest task, as the scheduler intends. Runs A and B ask identical questions; in run B each child read 54.8k prompt tokens from the provider's cache and cost US\$0.0148 instead of US\$0.0445 to US\$0.0465, which shows how much of a child's cost is the fixed prompt of Section 8.3.

8.7 The fan-out policy, live

The fan-out rules were tested with one prompt that asked for a review of three packages (`packages/tui/src` , `packages/agent/src` , `packages/ai/src/providers`) for error handling that swallows exceptions, three worst cases per area, with no mention of subagents. Each wording was run once on `zai/glm-5.3` in JSON mode.

Table 9. The same review prompt under the first-draft rules and the shipped rules, one run each, 28 October 2026. Times are from the run's message timestamps to the start of the final answer.

Measure	First draft (452 chars)	Shipped (512 chars)
<code>subagent</code> calls	0	1, parallel, 3 <code>scout</code> tasks
Parent tool calls	12 <code>bash</code> , 8 <code>read</code>	8 <code>read</code> of cited line ranges
Parent model turns	9	3
Parent context at the final turn	55.3k tokens	44.8k tokens
Child turns		17, 11, 17
Cost, parent / children / total	US\$0.174 / none / US\$0.174	US\$0.091 / US\$0.690 / US\$0.781

Measure	First draft (452 chars)	Shipped (512 chars)
Time to final answer	about 196 s	about 247 s

With the shipped wording the model split the work into three tasks, one per package, without being asked. It wrote each task to stand alone, with the patterns to search for, severity criteria and the answer format, and after the children returned it spent its own turns reading eight cited line ranges before answering. The parent's context stayed smaller. It was not cheaper or faster: the three children together ran 45 turns and cost 4.5 times the single-context run, and the call took longer than the whole single-context run. We did not grade the two answers against each other, so this run shows that the policy fires, not that fanning out produced a better review. One run per wording cannot separate the wording from model variance.

8.8 What we did not measure

We did not measure task success rates on a benchmark, the quality of fanned-out answers against single-context answers, the effect of the frozen memory block on cache hit rates over long sessions, the precision of the heal detector, keepalive throughput or failure modes over a night of unattended work, the cost of lane switching under `dexter/auto`, memory use with eight concurrent children, or any platform other than one macOS laptop. The repository's own eval suite reports that all nine cells of its GLM tuning matrix passed at every thinking level and that 41 of 44 documentation pages pass an audit on GLM-5.3 (`pkg/evals/docs/glm-tuning.md`); we cite it but did not re-run it.

9 Limitations and threats to validity

The prompt is too large. Section 8.3 is the most consequential finding. Every request pays for 31 tool definitions and every enabled skill's advertisement whether the task needs them or not. The `auto-invoke` router and `tool_search` exist, but the default configuration still advertises everything. Until the base prompt shrinks, each subagent child inherits the same fixed cost, which is why fan-out multiplies spend.

Startup depends on the user's configuration. Every subagent child re-loads the user's extensions and packages. A package that resolves upstream pi names pulls in a second runtime in every child, and the harness has no way to tell the user that a package costs half a second per process beyond the debug trace.

The heal detector over-counts. "Same tool" is a tool name, and there is no time or turn window, so an unrelated success of `bash` after any edit records a heal; failures of `edit` and `write` themselves can never be healed automatically, because the edit step returns early (`ca/extensions/learning/index.ts:141-147`). A playbook built from such records can teach the wrong lesson with confidence. The declared `heal_memory` edge kind is never created.

Pattern checks are not a security boundary. The memory and skill scanners catch a handful of literal phrasings. A paraphrased injection, an encoded payload, or an exfiltration through a tool other than `curl` passes. Provenance marking helps the model discount tampered memory but does not

prevent it from reading it. Sandboxing with seatbelt profiles [18] on macOS or bubblewrap [8] on Linux is off by default, applies to the bash tool only, and the macOS profile cannot filter network access by host (docs/sandbox.md).

The spend cap counts sessions, not money. Six sessions an hour bound a failing loop's rate, but one session with many subagents or a long context can cost far more than another. The keepalive worker can fan out, and its children are not counted.

Recursion is bounded by policy. Subagents receive no fan-out rules, but they keep the subagent tool, so a task that asks for delegation can still recurse. The 8-task cap applies per call, not per tree.

Documentation and code drift. The code puts the pause file and keepalive state under `~/.dexter/agent/` (`keepalive-command.ts:395-399`) while the README, `docs/keepalive.md` and the file's own header say `~/.dexter/` ; the credential fallback file is in the agent directory while `docs/providers.md` omits that level; the eval README names `DEXTER_PROVIDER` and `DEXTER_MODEL` while the code reads `PI_PROVIDER` and `PI_MODEL` ; the support packages' changelogs stop at 1.4.0 although they are versioned 3.0.0; and the skill router ranks a `pomerado` skill that is not in the tree (Pomerado's minter is reached over MCP instead [9]). None is severe alone; together they show that four days of fast change outran the documentation. The credential vault also still reads the legacy plain `auth.json` store as a fallback and migrates a key only when it is written again (`ca/core/credential-vault.ts:1-13`), so an installation that predates the vault can keep keys in a plain file despite the README's claim.

Threats to validity. All numbers come from one shared laptop with load averages from 20 to 45, mostly as medians of 5 to 16 runs without confidence intervals. Several comparisons mix versions (2.1.2 in the profile, 3.0.0 in the re-runs, source builds for issue #38). The fan-out evidence is one run per condition on one prompt and one model. The live runs used a GLM model whose provider showed multi-second variance in time to first byte. The author of this report is the author of the system, and the measurements were taken while the code was being changed, by the same person.

10 Future work

A deep revamp, starting the week of 2 November 2026. An unmerged README note on a branch of the Dexter source announces a revamp of every package and warns of breaking changes to commands, settings and extension APIs. Sections 8 and 9 suggest its first targets: a base prompt that advertises tools and skills on demand rather than all at once, with a size budget enforced in tests as the fan-out rules already are; extension loading that refuses or isolates a second runtime; a heal detector that keys on the failing command and its arguments within a bounded window, so the playbook records causes rather than coincidences; a spend cap in currency that counts subagent children; a hard recursion limit for subagents; and one source of truth for paths that the documentation is generated from. The durable stack (`chord` , `durable` , `env`) already ships in the tree with crash-safe conversations and remote execution over SSH; connecting it to the CLI would let a keepalive worker or a subagent survive a crash or run on another machine, but no part of the current CLI does this yet.

Native Troy integration. Today Troy is reached as an external process: Dexter discovers an endpoint file, authenticates each HTTP request with a token, may wait up to 12 s for a hidden launch, and handles origin refusals by granting and retrying. The kit's browser engine separately attaches

to Troy over the DevTools Protocol. A native integration would make Troy the browser engine of the harness rather than one tool beside nine others, so page reads, origin grants, the submit gate and page memory share one session state with the agent. The design is not fixed in the source at the time of writing.

SMBK as a native skill. Velofy's SMBK secrets vault is planned as a native skill. The current credential path (OS keychain, a mode-600 file fallback, and a legacy plain-file store read for compatibility) protects provider keys but not the many other secrets a coding session handles, such as service tokens and deployment credentials. Bringing SMBK into the harness, with secrets fetched at the moment of use and never written into transcripts, memory or skills, would extend the existing exfiltration checks of Section 5 from pattern matching toward a store the model never sees in plain text. No SMBK code exists in the Dexter source today.

Warm children for batch runs. The argument against a warm pool in Section 4.4 assumes interactive use, where tasks run for tens of seconds. In batch runs through `dexter fanout`, many short tasks can share one agent, directory and model; the tasks in Table 8 took 7 to 11 s, of which child startup is a visible share. A pool keyed by exactly those inputs, and discarded on any change to settings, skills or context files, would avoid the staleness the source warns about. Whether it pays for its memory is a measurement we have not made.

Better evidence. The evaluation above needs repeated, controlled runs: fan-out against single-context work on graded tasks, cache hit rates with and without the frozen block over long sessions, heal-detector precision on recorded sessions, and keepalive over unattended nights with failure injection. The vortex loop of Section 6.4, run against Dexter's own eval suite, is the instrument the source already provides for that work.

References

1. M. Zechner and Earendil Works. *pi*, terminal coding agent and toolkit. Fork baseline: commit `6c2fd73b0c54`, version 1.4.0. <https://github.com/earendil-works/pi>
2. DeepSeek. *deepseek-harness* (dsh). Studied at commit `dd6322d604`. <https://github.com/deepseek-ai/deepseek-harness>
3. Cordis, a plugin and service framework. <https://github.com/cordiverse/cordis>
4. Nous Research. *Hermes Agent*. <https://github.com/NousResearch/hermes-agent>
5. L. A. Agrawal et al. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv:2507.19457, 2025. <https://arxiv.org/abs/2507.19457>
6. OpenClaw. <https://github.com/openclaw/openclaw>
7. DietrichGebert. *ponytail*. <https://github.com/DietrichGebert/ponytail>
8. containers. *bubblewrap*. <https://github.com/containers/bubblewrap>
9. Pomerado. <https://github.com/Pomerado/pomerado>
10. Model Context Protocol specification, version 2025-11-25: Lifecycle, shutdown of stdio servers. <https://modelcontextprotocol.io/specification/2025-11-25/basic/lifecycle>

11. Chrome DevTools Protocol. <https://chromedevtools.github.io/devtools-protocol/>
12. tmux, a terminal multiplexer. <https://github.com/tmux/tmux>
13. unjs. *jiti*, runtime TypeScript and ESM support for Node.js. <https://github.com/unjs/jiti>
14. Node.js documentation. Modules: customization hooks, `module.registerHooks`.
<https://nodejs.org/api/module.html#customization-hooks>
15. models.dev, an open database of model specifications and pricing. <https://models.dev>
16. morluto. *rea*, a reverse-engineering CLI and MCP server.
<https://github.com/morluto/rea>
17. F. Bellard. *QuickJS*, a small embeddable JavaScript engine. <https://bellard.org/quickjs/>
18. reverse.put.as. Apple Sandbox Guide v1.0, September 2011.
<https://reverse.put.as/2011/09/14/apple-sandbox-guide-v1-0/>