

Skills as --help: CLI-Native Skill Exposure for Agents

Oct 20, 2026 · @Chris Kane

Author: Christopher Kane · Version 0.1 · Co-author: Claude (Anthropic)

Abstract

Skills should be exposed to agents the way `--help` is exposed to humans: as a bounded, structured, on-demand query against the tool itself, not as a pile of files the agent is free to `cat` and `grep` into its context window. Today a skill is a directory artifact (`SKILL.md` plus references, scripts, and examples) that the agent discovers by listing and consumes by reading. That design hands the agent a firehose and trusts it to sip.

This paper proposes that the CLI own the skill surface. A `skills` command family returns progressively deeper slices of a skill on request: a one-line summary, a short contract, a full procedure, a single reference file, a single example. Every response carries an explicit token cost and a continuation handle. Raw filesystem access to the skill tree is removed from the agent's reach. The result is a skill system where context consumption is a budgeted API call rather than an accident of `grep -r`.

The problem: skills are artifacts, and artifacts invite over-consumption

A skill today is a folder. The agent finds it through a listing injected into its prompt, then reads it with the same primitives it uses for source code: `cat`, `grep`, `find`, `Read`, `Glob`. Nothing in that path knows what a skill is, how much of it matters right now, or what it costs to load.

The failure modes are predictable and show up in every agent harness that ships skills this way:

- **Whole-file loads for one-line answers.** The agent needs the invocation syntax and reads a 400-line `SKILL.md` to get it. One question, 6,000 tokens.
- **Recursive discovery.** `grep -r "deploy" skills/` returns hits in 14 files. The agent reads 6 of them to find the relevant one. The other 5 are now permanent context.

- **Reference sprawl.** Skills link to `references/*.md`, `examples/*.py`, `scripts/*.sh`. Each link is a temptation. There is no signal distinguishing "load this now" from "exists if you need it."
- **Re-reading across turns.** With no handle to say "I already have section 3," the agent re-reads from the top after compaction or in a fresh subagent.
- **No cost feedback.** The agent has no idea a read cost 6,000 tokens until the context is gone. Humans see a scrollbar; agents see nothing.
- **Ambient instruction injection.** Everything the agent reads is treated as instruction. A reference file meant as background becomes a directive.

The root cause is architectural, not behavioral. We cannot prompt our way to disciplined reading when the only reading primitive is "give me the whole file." Prompting an agent to "read only what you need" is like telling a human to "only read the relevant man page" and then handing them the entire `/usr/share/man` tree. The tool has to enforce the discipline.

The analogy is exact. Before `--help`, Unix users read man pages: complete, authoritative, and far too long for the question at hand. `--help` won because it answered the question a user has at the moment of invocation, in under a screen, from the binary itself. Skills need the same thing, with the agent as the reader.

The `--help` analogy

`--help` is not documentation. It is a contract between a binary and a caller who has limited attention and a specific intent. Its properties are exactly the ones an agent needs from a skill:

Property of <code>--help</code>	What it does for a human	What the agent equivalent does
Served by the tool itself	No hunting for docs; <code>git --help</code> always works	No filesystem spelunking; <code>skills describe deploy</code> always works
Bounded by convention	Fits on one screen	Fits in a declared token budget
Layered	<code>--help</code> → <code>git commit --help</code> → man page	summary → contract → procedure → reference
Structured by habit	USAGE, OPTIONS, EXAMPLES in a known order	A fixed envelope the agent parses without thinking

Property of <code>--help</code>	What it does for a human	What the agent equivalent does
Reflects the installed version	Never stale relative to the binary	Never stale relative to the registered skill
Cheap to call repeatedly	No penalty for <code>--help</code> twice	Idempotent, cacheable, deduplicated across turns
Says what it does <i>not</i> cover	"See <code>man git-rebase</code> for details"	Explicit pointers to deeper layers with their cost

The critical insight is where `--help` lives: in the binary, behind an interface, not in a text file the user opens. A user cannot accidentally load all of `git`'s documentation by asking for `--help`. The interface makes over-consumption structurally impossible, not merely discouraged.

Skills today are the man-page era. Every skill is a complete document, and the agent is told to open it. The proposal is to move skills into the `--help` era: the CLI answers skill questions, and the agent never opens the file.

Skills as harness harmonization

Heterogeneous pairings, a model running inside another provider's harness, score below matched pairings on agentic evals. The gap is mostly convention, not capability, and skills are the right layer to close it.

Why the gap exists

A harness and its native model co-evolve. The harness encodes the model's trained habits: how a tool call is shaped, how a tool result is read, whether to plan before acting, how an edit is expressed, how permission prompts are handled, how terse the output should be. Swap in another model and every one of those assumptions misses quietly. The model is not worse. It is playing someone else's instrument.

The human analogy, and where it stops

People specialize. Some code, some play guitar, and they cross over through learning and practice. Models have trained affinities the same way, and the affinity is visible in evals. The analogy holds for the diagnosis. It breaks on the remedy: a human who practices is changed; a skill does not change weights. A harmonization skill is sheet music plus a

technique note, not practice. That is the right tool at task time, because in-context adaptation is all a harness can do, but it sets the ceiling. Skills narrow the gap; only training closes it.

What a harmonization skill carries

A harness profile keyed by `(model, harness)`, delivered at L1 and never larger than a contract. Each row is a convention the matched pair takes for granted and the cross pair gets wrong.

Convention	Matched pair assumes	Cross-pair failure	Profile provides
Edit protocol	Native diff or search/replace shape	Whole-file rewrites, malformed patches	The exact edit shape, one example
Tool-result reading	Result framing the model was trained on	Treats result text as instruction, or ignores it	How results are framed, what is data
Plan-vs-act cadence	Model's default interleaving	Over-plans or thrashes	When this harness expects a plan
Permission flow	Model knows which calls prompt the user	Stalls on prompts or bypasses them	Which calls gate, how to proceed
Context hygiene	Model's compaction habits match the harness	Re-reads after compaction, loses state	Where state lives, how to resume state
Output register	Native verbosity matches the UI	Too chatty or too terse for the surface	Target length and format per surface
Skill consumption	Model knows this harness's <code>skills</code> verbs	Falls back to <code>cat</code> and <code>grep</code>	The verb set and the budget

Mechanism in this design

The resolver gains a `(model, harness)` dimension. `skills harmonize` returns the profile for the running pair, around 300 tokens, loaded once per session. `skills describe <name>` returns the skill's contract with the harness-specific overrides merged in, so a skill written for one harness still reads correctly in another. The registrar validates one profile per supported harness. The budget meter logs which pairs load which profiles, which is the eval signal for free.

The eval loop

Run the agentic suite four ways: matched and cross, each with and without the harmonization profile. The cross-with-profile delta is the value of the skill. Any pair the profile does not move is a capability gap, not a convention gap, and that list goes to training rather than to skill authors.

Design principles

Five principles govern the design. Each one closes a specific failure mode from the problem statement.

1. **The CLI is the only door.** Skills are registered with the harness and served through `skills` subcommands. The skill tree is not on the agent's readable path. `cat`, `grep`, `Read`, and `Glob` return nothing (or a redirect) for skill content. Closes: recursive discovery, whole-file loads.
2. **Progressive disclosure with declared depth.** Every skill has fixed layers: L0 one-line summary, L1 contract (when to use, inputs, outputs, cost), L2 procedure (the steps), L3 references (named, fetched individually), L4 examples (named, fetched individually). An agent asks for a layer by name. Nothing above the requested layer is returned. Closes: reference sprawl.
3. **Every response is priced.** The envelope carries `tokens_returned`, `tokens_remaining_at_this_layer`, and the estimated cost of each deeper layer. The agent decides with the number in front of it. Closes: no cost feedback.
4. **Bounded and paged by default.** No single call returns more than a hard cap (proposed: 1,500 tokens). Longer content is chunked with stable continuation handles. The agent must ask again to get more, and each ask is logged. Closes: re-reading, accidental large loads.

5. Content is data; the envelope is the instruction. Skill body text is delivered inside a typed field, not as bare prompt text. The harness tells the agent which fields are procedure and which are reference. Closes: ambient instruction injection.

A sixth, softer principle: **the skill author writes layers, not a document.** Authoring tools enforce the layer structure at registration time. A skill that cannot be expressed as L0 through L4 is a skill that was going to be read badly anyway.

Proposed command surface

One command family, seven verbs, each mapped to a disclosure layer. Output is always a structured envelope (JSON by default, with a `--format=text` escape for humans).

Command	Layer	Returns	Cap (tokens)
<code>skills list [--tag T] [--match Q]</code>	L0	name, one-line summary, tags, L1 cost per skill	40 per skill, 600 total
<code>skills describe <name></code>	L1	when to use, when not to, inputs, outputs, side effects, layer costs	300
<code>skills show <name> [--step N]</code>	L2	the procedure, or one step of it	1,500
<code>skills ref <name> <ref-id></code>	L3	one named reference, chunked	1,500 per chunk
<code>skills example <name> <ex-id></code>	L4	one named example	800
<code>skills search <query> [--layer L]</code>	any	ranked hits with layer + chunk id, never content	400
<code>skills run <name> [args]</code>	n/a	executes the skill's script if it has one; returns result envelope	1,000

skills list

```
{
  "skills": [
    {"name": "deploy", "summary": "Ship a service to an environment with
rollback", "tags": ["ops"], "l1_tokens": 210},
    {"name": "pdf-extract", "summary": "Pull text and tables out of PDFs",
"tags": ["files"], "l1_tokens": 180}
  ],
  "count": 2, "truncated": false, "tokens_returned": 85
}
```

skills describe deploy

```
{
  "name": "deploy", "version": "3.2.0",
  "use_when": "User asks to ship, release, roll out, or roll back a
service.",
  "not_when": "Local dev servers; use `dev-server` instead.",
  "inputs": {"service": "string", "env": "staging|prod"},
  "outputs": {"release_id": "string"},
  "side_effects": ["network", "writes to release ledger"],
  "layers": {"procedure": {"steps": 6, "tokens": 920},
    "refs": {"rollback-policy": 1400, "env-matrix": 600},
    "examples": {"prod-hotfix": 350}},
  "tokens_returned": 210
}
```

skills show deploy --step 3

```
{
  "name": "deploy", "step": 3, "of": 6,
  "title": "Verify health before cutover",
  "body": "Call `skills run deploy --check <release_id>`. Proceed only on
`healthy: true`.",
  "refs_cited": ["rollback-policy"],
  "next": "skills show deploy --step 4",
  "tokens_returned": 60
}
```

```
skills search "blue green"
```

```
{
  "hits": [
    {"skill": "deploy", "layer": "ref", "id": "rollback-policy", "chunk": 2,
    "score": 0.91, "tokens": 480}
  ],
  "tokens_returned": 40
}
```

Search returns pointers, never bodies. The agent sees the cost of the hit and makes a second, deliberate call to fetch it. That two-step is the whole point: it turns `grep -r` into a quote-then-buy transaction.

Enforcing token discipline

The guarantees have to be mechanical. An agent will take the cheapest path its tools allow, so the tools must make the cheap path the correct one. Six mechanisms, in order of importance:

1. Skill content is unreachable by generic file tools

The skill store lives outside the agent's sandbox root, or inside it under a path the file tools refuse. `Read skills/deploy/SKILL.md` returns:

```
{"error": "skill_content", "hint": "skills describe deploy"}
```

`grep`, `find`, `Glob` behave the same way. This is a hard gate, not a prompt instruction. It removes the single largest source of blowout at zero prompt cost. Humans who need the raw files use `skills export <name>` from their own shell.

2. Hard output caps per verb

Each verb has a ceiling (see the command table). The CLI truncates at the cap and returns `truncated: true` plus a `next` handle. There is no `--all`, `--full`, or `--no-limit` flag exposed to agents. If a skill author needs more than 1,500 tokens for one step, the registration tool rejects the skill until the step is split.

3. Continuation handles, not offsets

Chunks are addressed by stable ids (`rollback-policy#2`), not byte offsets or line numbers. Handles survive skill updates within a minor version, so a subagent or a post-compaction turn can resume without re-reading from chunk 1. The harness can also record which handles a session has already fetched and return `already_in_context: true` instead of the body on a repeat request.

4. Priced envelopes

Every response ends with `tokens_returned` and every pointer carries an estimated `tokens`. Estimates come from the same tokenizer the harness uses for context accounting, computed at registration time and cached. The agent never has to guess what a call will cost.

5. Session budget with soft and hard limits

The harness tracks cumulative skill tokens per session. At a soft limit (proposed: 8,000) responses include `budget_warning: true`. At a hard limit (proposed: 20,000) `ref` and `show` calls return pointers only until the agent explicitly acknowledges with `--budget-override`, which is logged. The numbers are tunable; the existence of a ceiling is not.

6. Typed delivery

The body of a procedure step arrives in a `body` field inside a JSON envelope. The harness renders it to the model with a frame that names it as skill procedure content. Reference material arrives in a `reference` field framed as background. The model is never handed skill text as unframed prompt, which is how reference files become accidental directives today.

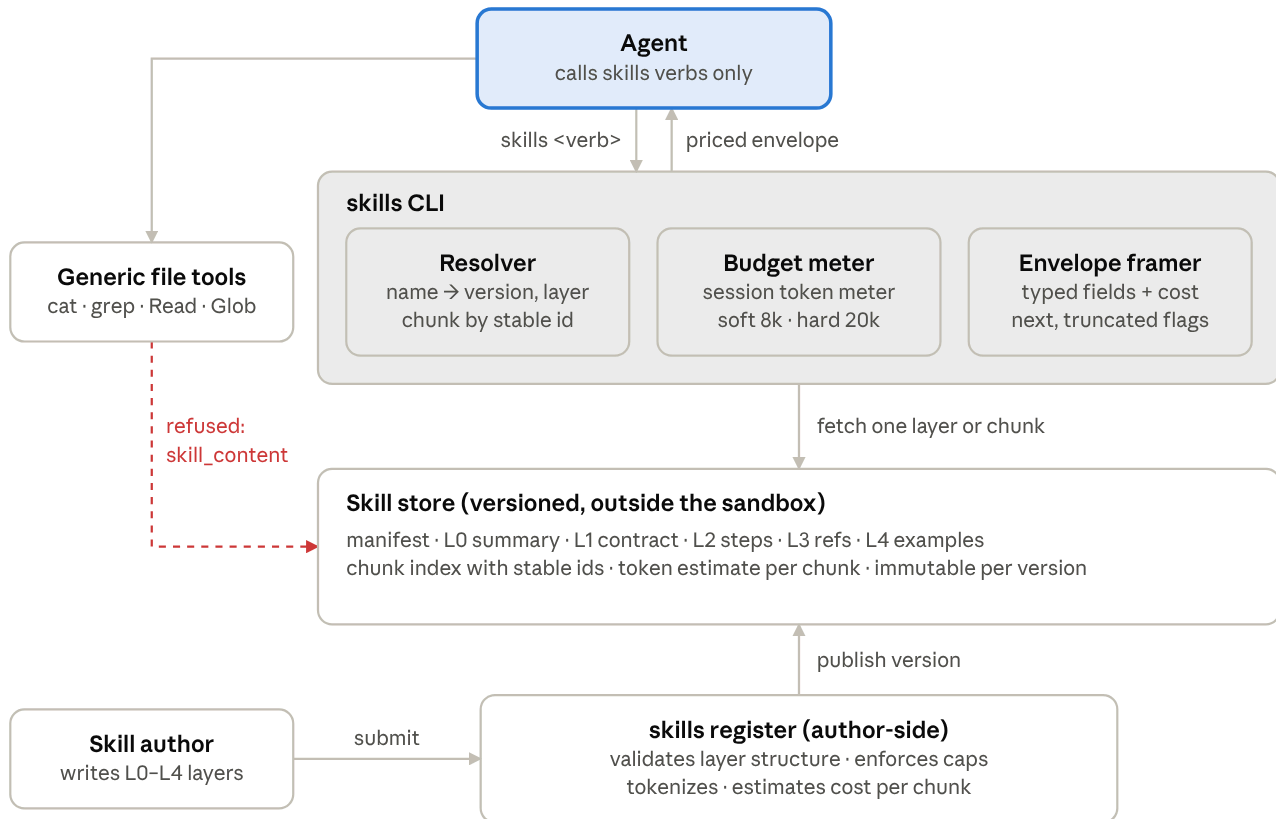
What this does not do

It does not stop an agent from calling `skills show` on every step of every skill. It makes that expensive in visible, budgeted, logged ways, which is enough: agents respond to cost signals when the signals are present. The goal is to make over-consumption a deliberate choice rather than the default outcome of `grep -r`.

Reference architecture

Four parts: the agent, the `skills` CLI, the skill store, and the registrar. The agent talks only to the CLI. Generic file tools are refused at the store boundary. Authors publish through the registrar, which is the only writer.

The CLI is the only door between the agent and skill content



skill exposure architecture · agent, CLI, store, registrar

The agent reaches skill content through one path, the `skills` CLI, which resolves, meters and frames every response; `cat` and `grep` are refused at the store, and only the registrar writes to it.

Skill package layout

A skill is still a directory on the author's side. The registrar compiles it into the store's layered form.

```

deploy/
  manifest.yaml      # name, version, summary (L0), contract (L1), layer
map
  procedure.md       # L2: one `## Step N` heading per step, each ≤ 1,500
tokens
  refs/
    rollback-policy.md # L3: chunked at registration
    env-matrix.md
  examples/
    prod-hotfix.md     # L4
  scripts/
    check.sh           # exposed via `skills run`, never read by the agent

```

Resolver

Maps `(name, layer, id, chunk)` to a store entry. Pins to the highest registered version compatible with the session's lock, so a session never sees a skill change mid-run. Returns `already_in_context` when the harness has already delivered that chunk this session.

Budget meter

Counts delivered tokens per session and per skill. Emits warnings and enforces the hard limit. Logs every fetch with layer, chunk, and cost so skill authors can see which layers agents actually use.

Envelope framer

Wraps the body in the typed envelope, attaches `tokens_returned`, `next`, and layer cost estimates. The harness renders the envelope to the model with a fixed frame: procedure content is marked as procedure, reference content as reference.

Registrar

Author-side command. Rejects a skill that is missing L0 or L1, has a step over the cap, or references a file not under `refs/` or `examples/`. Tokenizes every chunk with the harness tokenizer and stores the estimates. Writes are append-only by version.

Comparison with existing approaches

Three ways skills reach agents today, against the proposal. CLI skill commands are the only option that bounds consumption at the tool boundary rather than in the prompt.

Concern	SKILL.md on disk	MCP tool per skill	CLI skill commands (proposed)
Discovery cost	Listing injected into prompt; grows with skill count	Tool schema per skill in every request; grows faster	<code>skills list</code> on demand, 40 tokens per skill, paged
Consumption unit	Whole file	Whole tool result	One layer or chunk, capped
Cost visibility	None	None	<code>tokens_returned</code> on every response
Over-read prevention	Prompt instruction only	Prompt instruction only	File tools refused; caps enforced
Resume after compaction	Re-read from top	Re-call tool	Stable chunk handle, <code>already_in_context</code>
Injection surface	Every file is bare prompt	Tool result is bare prompt	Typed envelope with framed fields
Author constraints	None	Schema only	Layer structure and caps validated at registration
Works offline / in CI	Yes	Needs server	Yes, local store
Human readable	Yes	No	<code>--format=text</code> , <code>skills export</code>

MCP deserves a note. It solves distribution and execution well, and `skills run` should delegate to MCP where a skill wraps a remote tool. It does not solve disclosure: an MCP tool's description is all-or-nothing in the schema, and its result is unbounded. The two compose: MCP for execution, CLI skill commands for knowledge.

Migration path

Existing `SKILL.md` skills convert mechanically; the hard gate comes last so nothing breaks on day one.

1. **Shim the CLI over existing files.** `skills` commands read the current `SKILL.md` directories as-is. L0 = frontmatter description, L1 = synthesized from the first section, L2 = the body split on `##` headings, L3 = everything under `references/`, L4 = everything under `examples/`. No author changes required.
2. **Add the budget meter in observe mode.** Log every skill fetch with cost. No limits yet. Two weeks of data shows which skills are over-read and which layers are never touched.
3. **Ship the registrar.** Authors run `skills register` to validate and tokenize. Skills that fail (a step over the cap, missing L1) get a report, not a block. Fix the worst offenders.
4. **Enforce caps and soft budget.** Truncation and `budget_warning` go live. Agents that were over-reading now see pointers instead of bodies.
5. **Close the filesystem door.** Move the store out of the sandbox root; file tools return the redirect. This is the step that delivers the guarantee, and it is safe only after steps 1 through 4.
6. **Retire the prompt-injected skill listing.** Replace the static block of skill names in the system prompt with one line: `run skills list to see what is available`. This is the largest single token saving for harnesses with many skills.

Each step is independently reversible. Step 5 is the only one with a blast radius, and it is a one-line config change to undo.

Open questions and risks

The design has real costs. These are the ones that could sink it, with the current best answer for each.

- **Latency per layer.** A skill that needs L1, L2 and two refs is four round trips where it used to be one `cat`. Mitigation: `skills describe --with-procedure` bundles L1 and L2 when their combined cost is under the cap. Keep the bundle list short; every bundle flag is a step back toward whole-file reads.
- **Agents that cannot discover what they do not know to ask for.** Progressive disclosure assumes the agent asks the right next question. A skill whose critical caveat sits in L3 will be missed by an agent that stops at L2. Mitigation: L1 is required to list `must_read`

refs, and `skills show` includes them in `refs_cited` on the relevant step. Authors own this; the registrar can lint for refs never cited.

- **Token estimates drift across models.** The registrar tokenizes with one tokenizer; the agent may run on another. Mitigation: store estimates per tokenizer family, fall back to the closest, and label the field `tokens_est`. Being 10% off is fine; being silent is not.
- **Search quality.** `skills search` is only as good as its index. A bad index pushes agents back toward reading everything. Mitigation: index L0 and L1 with embeddings, L2 and L3 with BM25 over chunk titles and first lines, and return `score` so the agent can judge.
- **Author friction.** Forcing layers and caps makes authoring harder than writing one Markdown file. This is the point, but adoption will suffer if the registrar is hostile. Mitigation: `skills register --fix` auto-splits steps at the cap and drafts L1 from the body, then asks the author to confirm.
- **Harness lock-in.** The CLI is harness-specific unless the envelope and verb set are standardized. Open question: whether to propose the verb set and envelope as a spec that any harness can implement, so a skill registered once works everywhere.
- **Does denying `grep` actually hold?** An agent with shell access can `find / -name SKILL.md`. Mitigation: the store is a content-addressed blob store, not a file tree, mounted nowhere in the sandbox. The CLI reaches it over a socket. There is nothing to `find`.

The largest unknown is behavioral, not technical: whether agents trained on `cat -and-grep` will use `skills describe` when it exists. Early evidence from harnesses with `--help`-style tool introspection suggests yes, but only when the generic path is closed. If both paths stay open, agents take the familiar one.

Appendix: manifest schema and response envelope

Minimum viable versions of both. Fields marked optional can be omitted in v1.

manifest.yaml

```
name: deploy
version: 3.2.0
summary: Ship a service to an environment with rollback # L0, ≤ 20 words
tags: [ops, release]
contract: # L1
  use_when: User asks to ship, release, roll out, or roll back a service.
```

```

not_when: Local dev servers; use dev-server instead.
inputs:
  service: string
  env: {enum: [staging, prod]}
outputs:
  release_id: string
  side_effects: [network, writes release ledger]
  must_read: [rollback-policy] # refs the agent
must fetch before running
procedure: procedure.md # L2, split on "##
Step N"
refs: # L3
  rollback-policy: refs/rollback-policy.md
  env-matrix: refs/env-matrix.md
examples: # L4
  prod-hotfix: examples/prod-hotfix.md
run: # optional
  entry: scripts/check.sh
  args: [--check]
caps: # optional
overrides, never above harness max
  step_tokens: 1200

```

Response envelope

Every `skills` verb returns this shape. Unused fields are omitted, not nulled.

```
{
  "skill": "deploy",
  "version": "3.2.0",
  "layer": "ref",
  "id": "rollback-policy",
  "chunk": 2,
  "of": 3,
  "kind": "reference",
  "body": "...",
  "refs_cited": [],
  "next": "skills ref deploy rollback-policy --chunk 3",
  "truncated": false,
  "already_in_context": false,
  "tokens_returned": 480,
  "tokens_est_next": 410,
  "budget": {"session_used": 3120, "soft": 8000, "hard": 20000, "warning":
false}
}
```

`kind` is one of `summary`, `contract`, `procedure`, `reference`, `example`, `result`, `pointer`. The harness uses it to choose the frame when rendering `body` to the model. `pointer` is what every verb returns once the hard budget is hit.

Error shape

```
{"error": "skill_content", "hint": "skills describe deploy"}
{"error": "not_found", "hint": "skills search \"blue green\""}
{"error": "budget_exceeded", "hint": "pass --budget-override to continue;
this is logged"}
```

Every error carries a `hint` that is a runnable command. The agent should never have to guess the recovery.