Schema Discoverability, Not Locality, Drives MCP Cost Savings: A Controlled Decomposition

Anonymous Authors[1]

Abstract

Announcing an MCP tool in the system prompt cuts agent cost by 56% on Claude Sonnet 4.6 (N=30, cold cache); the default Claude Code deployment omits that announcement and captures only 19% of the gap (−10.9% vs. −56.3%). Same server, same execution, same cache regime; only the system-prompt addition differs. A pure-announcement variant (announcement text, no use-instruction) reaches −45.8%, 77% of the gap, attributing the rest to instruction priming. Practitioner benchmarks reporting 32−100× MCP savings (OnlyCLI, 2025; Speakeasy, 2025) measure the announced regime; users running default tooling do not. Prior benchmarks do not separate this from primitive type.

To isolate it, we decompose six agent memory primitives along three axes (schema availability, schema discoverability, execution locality) and run controlled contrasts. CLI-vs-script holds locality fixed and varies only schema availability; eager-vs-lazy MCP holds the server fixed and varies only announcement. Across three task families (file report, multi-file Python refactor, code Q&A) at N=30 with per-task UUID prefix-cache defeat, CLI and eager MCP overlap at −56% on report and within 9pp on the other families: locality contributes no detectable savings on report or refactor and at most ∼9pp on Code Q&A. A `UserPromptSubmit` hook reaches −80% by pre-executing the work entirely. Task quality is preserved across all primitives (oracle pass rates 1.00/1.00/0.99 on report/refactor/Code Q&A).

An exact additive re-parameterization (three axes plus a pre-execution scaling term that captures the hook ceiling) decomposes per-primitive means with moderately stable axis coefficients across families; schema discoverability alone accounts for ∼40% of baseline cost. Per-seed cost correlates with agent turn counts (Pearson r=0.92 / 0.88 / 0.71 on report / refactor / Code Q&A), consistent with the three axes acting through agent reasoning volume. The ordering replicates under warm cache on Claude Sonnet 4.6 and Opus 4.7, with one notable exception: Claude Haiku 4.5 discovers the lazy-MCP schema unaided (lazy collapses to −71%), so the discoverability axis is mediated by the model's tool-discovery behavior, not a fixed structural property. Benchmark and reference implementations released as supplementary material.

1. Introduction

Deploying LLM agents at scale makes token cost a first-order engineering concern. Two distinct lines of work attack the problem. The first compresses past behavior into in-context artifacts: Voyager's skill library (Wang et al., 2023), Agent Workflow Memory (Wang et al., 2025), Agentic Plan Caching (Zhao et al., 2025), MemGPT's paged context (Packer et al., 2025), and A-MEM's Zettelkasten graph (Xu et al., 2025). These artifacts live in the agent's attention budget. The second compiles recurring behaviors into runtime primitives that the agent never reads in full: hooks, permissions, MCP tools registered through the model context protocol (Anthropic, 2024), and progressive-disclosure skills as in Anthropic's Agent Skills system (Anthropic, 2025b). Practitioner benchmarks have reported large cost gaps between primitive types on this second axis. OnlyCLI (OnlyCLI, 2025) measures a 32× gap between CLI and MCP, Speakeasy (Speakeasy, 2025) reports up to 100× savings from dynamic schema retrieval over loading every tool's full schema upfront, and Hasan et al. (Hasan et al., 2026) study how schema description quality drives cost across 856 tools.

Three mechanisms hide inside the *primitive type* variable:

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

schema availability (whether a compact interface description is reachable without reading the implementation), discoverability (whether the agent is told that schema exists, or has to find it), and execution locality (inside or outside the agent's reasoning loop). Existing CLI-vs-MCP comparisons hold none of these fixed: a CLI is surfaced and runs in-process, while an MCP tool is surfaced but runs in a separate server, so the reported gap mixes all three. We disentangle them. We compare six primitive types (baseline, script, CLI, MCP eager, MCP lazy, hook) on three task families under per-task prefix-cache defeat, using within-subject paired tests. The CLI condition holds locality fixed (same in-process `Bash` as the bare script) so its difference from script isolates schema availability; the eager-vs-lazy MCP contrast holds the server fixed and varies only announcement, isolating discoverability; the hook condition layers on pre-execution. Beyond the CLI–MCP gap, scripts produce no practical savings (see §7); default lazy-MCP captures roughly one-fifth of the eager condition's savings on the same server; hooks bottom out where agent output reduces to a brief acknowledgement.

Contributions. The primitive-type variable splits along three axes: schema availability, schema discoverability, and execution locality. Two controlled contrasts isolate two of them. CLI-vs-script holds locality fixed (both run in the agent's in-process `Bash`), so its difference is pure availability. Eager-vs-lazy MCP holds the server fixed, so its difference is pure announcement. We measure all six primitives on three task families (file report, multi-file Python refactor, code Q&A) at $N{=}30$ under per-task UUID prefix-cache defeat, using paired Wilcoxon, BCa-bootstrap CIs, and Hedges' g_z. The resulting spectrum covers the script and hook endpoints prior practitioner benchmarks omit, plus the lazy-MCP condition that matches the Claude Code default. Benchmark and reference implementations ship with the paper.

2. Related Work

In-context memory. Reflexion (Shinn et al., 2023) and ExpeL (Zhao et al., 2024) append verbal reflections to future prompts. Generative Agents (Park et al., 2023) combine episodic logs with periodic reflection. AWM (Wang et al., 2025) induces reusable workflows from trajectories. A-MEM (Xu et al., 2025) organizes notes as a self-linking graph. These artifacts ride in context on every reuse, paying their token weight repeatedly. Our *script* condition instantiates the same regime.

Tool creation and discovery. LATM (Cai et al., 2024) and CRAFT (Yuan et al., 2024) have a strong model write Python tools that a cheaper model invokes; CREATOR (Qian et al., 2023) disentangles tool design from tool use. Toolformer (Schick et al., 2023) internalizes tool-use policy into weights, the opposite extreme from our runtime primitives. The MCP protocol (Anthropic, 2024) standardizes tool schemas; subsequent work on Tool Search (Anthropic, 2026) and Anthropic's Agent Skills (Anthropic, 2025b) both move toward progressive disclosure of schemas. Our CLI condition isolates this lever with the simplest possible mechanism (`--help`).

Practitioner benchmarks of the same axis. OnlyCLI (OnlyCLI, 2025) reports a $32\times$ token reduction when an agent calls a CLI's `--help` on demand instead of carrying an MCP server's full schema. ScaleKit's open-sourced 75-run within-subject benchmark (ScaleKit, 2025) reports CLI<MCP at $p{<}0.05$; Speakeasy (Speakeasy, 2025) reports $10{-}100\times$ wins via dynamic / lazy schema loading. Hasan et al. (Hasan et al., 2026) study schema-description quality as a cost lever across 856 tools, and MCP-Zero (Fei et al., 2025) reaches a 98% reduction via active tool discovery across 2,797 tools. Zechner (Zechner, 2025) reports a *counter*-polarity result on one Claude Code harness (MCP slightly cheaper than CLI), attributed to a Haiku-based command-injection scanner that taxes the `Bash` path. We share these works' motivation but extend their measurement: prior benchmarks omit the script, lazy-MCP, and hook endpoints, leave prefix-cache state uncontrolled, and confound schema availability with execution locality (CLI vs. MCP varies both at once). Our matched contrasts hold one variable at a time; in particular, the eager-vs-lazy MCP pair uses the *same* MCP server and varies only system-prompt announcement, isolating schema discoverability, a distinction the practitioner reports conflate. Speakeasy (Speakeasy, 2025) names lazy schema loading; we extend this with a same-server eager-vs-lazy controlled contrast in which only the system-prompt announcement varies. Table 1 summarizes the differences.

Spectra and surveys. Zhou et al. (2026) survey "harness engineering" as a unified framing. Zhang et al. (2026) propose a knowledge-abstraction spectrum (traces→rules). Our axis (interface schema availability) is orthogonal: any abstraction level in their spectrum can live at any of our primitive points.

Cost-oriented work. Zhao et al. (2025) report 50% cost reduction via plan-template caching; SWE-Effi (Hrubec, 2025) identifies "token snowballs" in agentic frameworks. Liu et al. (Liu et al., 2025) formalize a unified token+tool-call cost metric for budgeted agent scaling. MCPAgentBench (2025) release a 20,000-tool benchmark with a token-efficiency metric. We share the cost focus and add the primitive-type variable.

Table 1. How our measurement differs from prior practitioner benchmarks on the primitive-cost axis. "Endpoints" indicates whether the bare *script* and `UserPromptSubmit` *hook* primitives are measured. "Cache control" indicates whether prefix-cache state was held constant across primitives.

Work	Primitives compared	Cache control	Statistics	Endpoints
OnlyCLI (OnlyCLI, 2025)	CLI vs MCP	none	ratio (32×)	no
ScaleKit (ScaleKit, 2025)	CLI vs MCP	none	75-run, $p<0.05$	no
Speakeasy (Speakeasy, 2025)	full vs lazy-load MCP	none	ratio (10−100×)	no
Hasan et al. (Hasan et al., 2026)	856 tool descriptions	none	cross-tool quality regression	no
MCP-Zero (Fei et al., 2025)	2,797 tools (active discovery)	none	retrieval-style benchmark	no
Zechner (Zechner, 2025)	CLI vs MCP (one harness)	none	anecdotal	no
This work	**6 primitives (incl. script, lazy MCP, hook)**	**per-task UUID**	**paired Wilcoxon, d_z, BCa CI, Holm**	**both**

3. The Six-Primitive Spectrum

We operationalize six primitive types, ordered by increasing removal from the agent's reasoning loop:

1. **Baseline.** No primitive. Agent reasons from scratch each task.
2. **Script.** An agent-written executable file in `./scripts/`. Discovered via `ls`, read fully via `Read`, then invoked via `Bash`. No standardized interface.
3. **CLI.** A pre-built executable with `argparse` and a `--help` flag. Same invocation path as **script** (agent `ls` discovers, `Bash` invokes), but the agent can learn the interface from the short help output rather than reading the implementation.
4. **MCP (eager).** A stdio MCP server (Anthropic, 2024) registered via `--mcp-config`. The agent's system prompt is augmented with an explicit hint announcing the tool's name and signature. Execution happens in a separate process.
5. **MCP (lazy).** The same MCP server, registered the same way, but *without* the system-prompt announcement. The agent must discover the tool through the MCP protocol's `tools/list` on demand. This matches the default Claude Code deployment shape, where schemas are deferred (Anthropic, 2026).
6. **Hook.** A `UserPromptSubmit` hook (Anthropic, 2025) that parses the user prompt, pre-computes the result, writes the output file, and injects a short confirmation into the user message. Fires before the agent sees the prompt.

The six primitives differ on three axes: (i) whether a compact interface description is *available* (reachable without the agent reading the implementation), (ii) whether the agent is *told* that description exists, and (iii) whether execution happens inside or outside the agent's reasoning loop (Table 2).

Table 2. Three-axis characterization of the six primitives.

Primitive	Schema avail.	Surfaced	Exec. outside loop
Baseline	N/A	N/A	N/A
Script	No	No	No
CLI	`--help`	Yes	No
MCP (eager)	schema	Yes	Sep. proc.
MCP (lazy)	schema	No	Sep. proc.
Hook	N/A	N/A	Pre-prompt

4. Experiment

Task family. A compound "build report" family: for each of five file extensions (`.py`, `.js`, `.md`, `.rs`, `.go`), the agent produces a JSON summary of matching files under `./src/` containing: file count, total line count, total byte count, longest and shortest files, and a sorted file list with per-file statistics. Each task uses *different* natural-English phrasing to defeat lexical prefix caching. Fixture: a synthetic 6-file multi-language project with known line counts.

Cold cache per task. We inject a per-task UUID into the system prompt via `--append-system-prompt "[run=<uuid>-task=<uuid>]"`, defeating Anthropic's server-side prefix cache between tasks and between seeds. This isolates the primitive effect from caching and gives a conservative cold-cache lower bound on savings.

Workspace reset policy. Baseline, eager MCP, lazy MCP, and hook modes wipe the workspace between tasks (the primitive lives outside the workspace). Script and CLI modes preserve the workspace (the primitive lives in `./scripts/`).

Agent model. Claude Sonnet 4.6 throughout. Each trial invoked via `claude -p` with `--output-format json` and `--permission-mode bypassPermissions`.

Sample size and statistics. $N=30$ seeds per primitive on each task family (900 uncached inference runs per family: six primitives × thirty seeds × five tasks, each with a fresh UUID cache key). Each seed runs all five tasks in sequence within a mode. Primary metric:

`total_cost_usd` summed over the five tasks (cache-creation, cache-read, input, and output tokens priced per Anthropic's posted rates). We use within-subject paired tests: a one-sided paired Wilcoxon signed-rank test on per-seed primitive−baseline cost differences, paired Cohen's d_z, Hedges' g_z with the small-sample correction $J = 1 - 3/(4\,\mathrm{df} - 1)$, and a 10,000-iteration BCa-bootstrap 95% CI on the mean per-seed $\Delta\%$ (BCa is preferred over the percentile bootstrap for the heavier-tailed primitives in our sample, especially hook with sd/mean=54%). We report exact `scipy.stats.wilcoxon` p-values (one-sided). At N=30, α=0.05 one-sided, 80% power, the minimum detectable effect size is $d_z \approx 0.49$. Five primitive-vs-baseline tests are reported per family; we apply Holm–Bonferroni step-down at family-wise α=0.05. With observed p-values, Holm rejects the four non-null primitives (CLI, eager MCP, lazy MCP, hook) on both families and does not reject script on the report family. No runs were discarded.

Three task families. Beyond the report family described above, we replicate the full six-primitive comparison on two further families. (i) Multi-file Python refactor: rename a function across a codebase, move a class between modules with imports updated, convert positional args to keyword-only with caller fix-ups, extract an inline validation block into a helper, replace a magic number with an imported constant. Per task quality oracle = pytest passes + a per-task structural check (e.g. symbol absent at old name, present at new name). (ii) Code Q&A: five exploration questions over a synthetic codebase (callers of a symbol, transitive class dependencies, package entry point, call chain between two functions, Python 3.10+ language-feature detection). Per-task oracle = $\geq 80\%$ key-term match against the ground-truth answer key. All three families: same primitives, same UUID cache-defeat protocol, same N=30.

Quality oracle. Every task's JSON output is compared against known ground-truth line counts (`analyze.py::quality_check`, released with the benchmark). A trial counts as success only if all five files are correct.

5. Results

Table 3 presents per-primitive aggregate cost on the report family; Table 4 the same on the refactor family; Figure 1 visualizes all three families side by side (report, refactor, and Code Q&A).

Bare scripts produced no detectable cost reduction over baseline ($\Delta = -2.2\%$, one-sided paired Wilcoxon p=0.191, paired $d_z = -0.22$, 95% CI on $\Delta\%$ of $[-9.2, +5.0]$). The CI crosses zero, ruling out any practical effect at our power. Mechanism: the agent must `Read` the script body before invocation, and that read-cost roughly cancels the saved planning cost.

CLI uses the same execution path as the script (agent `ls` discovers, `Bash` invokes), plus a `--help` flag. It reaches -56.4% (CI=$[-60.0, -52.1]$). Eager MCP reaches -56.3% (CI=$[-59.7, -52.3]$). The two intervals overlap almost entirely: once a compact interface description is reachable without reading the implementation, where execution happens contributes essentially nothing to cost. All 30/30 seeds beat baseline for both.

Without the system-prompt hint that announces the tool, the agent must discover the MCP tool via the protocol's own `tools/list`; in practice the Claude Code harness defers schema loading and the agent often falls back to writing a Bash one-liner. Lazy MCP reaches only -10.9% (CI=$[-15.7, -6.2]$, $d_z = -0.80$), closer to the bare-script condition than to eager MCP. OnlyCLI's $32\times$ figure compares CLI to eager full-schema MCP, and Speakeasy's $100\times$ figure compares full-schema loading to dynamic retrieval that surfaces schemas on demand (OnlyCLI, 2025; Speakeasy, 2025); both implicitly assume the agent has access to the relevant schema. Our lazy condition tests what happens when that assumption fails by default, and finds the savings users see roughly $5\times$ smaller than the eager-condition figure.

At -80.5% (CI=$[-84.1, -76.4]$), the hook pre-executes the work before the agent reasons; agent output drops to 962 tokens versus 4,850 for baseline. The hook is a ceiling reference, not a peer on this axis: it replaces the agent's reasoning loop with a regex-driven pre-computation, so its Δ is what's left after taking the agent out. The bound applies only to tasks the hook can recognize from the user prompt; tasks needing mid-conversation context (§7) cannot benefit this much.

All five reports per seed were correct across all runs; first-attempt success was 1.0 for every primitive. To check the structural-oracle itself for over-leniency on Code Q&A, we ran an independent LLM-judge audit (Claude Opus, blind to the structural oracle) over a $5\times6\times5$=150-cell sample: judge and oracle agree on 96.7% of cells. All four disagreements are on task-03, where the oracle accepts `__main__.py:__main__` but the canonical answer is `:main`. The leniency hits baseline and script only, never hook or lazy MCP. The audit catches false-positive leniency the judge can detect from the agent's response text, not failures invisible to that text. Dispersion grows down the spectrum (coefficient of variation, sd/mean: baseline 15.5%, script 10.5%, CLI 24.6%, MCP eager 18.9%, MCP lazy 9.4%, hook 54.3%), so d_z values across primitives are not directly comparable; the hook's larger $|d_z|$ mainly reflects smaller mean cost, not better separation. Output tokens: baseline 4,850; script 3,410; CLI 1,813; MCP eager 2,528;

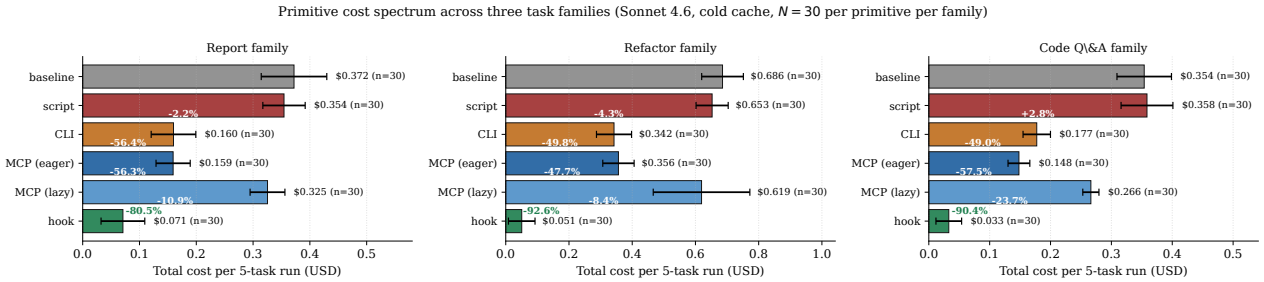

Figure 1. Mean cost per 5-task run by primitive across three task families (N=30; sd error bars; paired $\Delta\%$ vs. baseline annotated).

Table 3. Per-primitive cost on the report task family (N=30). Negative Δ is a cost reduction. The 95% CI is a 10,000-iteration bootstrap on the paired per-seed $\Delta\%$. Paired Wilcoxon p-values are one-sided; d_z is paired Cohen's d_z and g_z is its small-sample Hedges' correction.

Primitive	Mean ($)	sd ($)	$\Delta\%$	95% CI on $\Delta\%$	p	d_z	g_z
Baseline	0.372	0.058	—	—	—	—	—
Script	0.354	0.037	-2.2	$[-9.2, +5.0]$	0.191	-0.22	-0.21
CLI	0.160	0.039	$\mathbf{-56.4}$	$[-60.0, -52.1]$	**<0.001**	$\mathbf{-3.48}$	-3.39
MCP (eager)	0.159	0.030	$\mathbf{-56.3}$	$[-59.7, -52.3]$	**<0.001**	$\mathbf{-3.33}$	-3.24
MCP (lazy)	0.325	0.031	$\mathbf{-10.9}$	$[-15.7, -6.2]$	**<0.001**	-0.80	-0.78
Hook	0.071	0.039	$\mathbf{-80.5}$	$[-84.1, -76.4]$	**<0.001**	$\mathbf{-4.44}$	-4.32

MCP lazy 4,750; hook 962. The lazy-MCP output volume matches baseline because the agent reasons through the task without using the tool.

The qualitative spectrum is preserved on the refactor family (Table 4, Figure 1 middle panel). Script's effect becomes detectable but small ($\Delta = -4.3\%$); CLI and eager MCP overlap tightly at -50% / -48%; lazy MCP's CI crosses zero; hook *strengthens* to -92.6% as its regex-based intent parser pre-applies the refactor and the agent's reply collapses to a $\sim$25-token acknowledgement. Per-task oracle (pytest + structural check) passes at 1.0 across every primitive$\times$seed cell. The refactor lazy-MCP row deserves a note: $p<0.001$ alongside a CI crossing zero is not a contradiction. The Wilcoxon tests the sign of paired raw-cost differences (most seeds are negative, so p is small), while the bootstrap CI is on the mean of per-seed $\Delta\%$, which has heavier tails because of high cost-variance on hard refactor tasks (sd=0.153, roughly 3$\times$ the other primitives'). A handful of seeds with very high baseline cost contribute large positive $\Delta\%$ values that pull the percentage mean toward zero. The two statistics agree that lazy MCP usually saves; they disagree on how much.

A third family of five exploration questions (callers, class deps, entry point, call chain, Python 3.10+ feature detection) over a synthetic Python codebase preserves the ordering (Table 5, Fig. 1 right): hook -90.4%, eager MCP -57.5% and CLI -49.0% overlap, lazy MCP -23.7%. Two notes: script edges *positive* ($+2.8\%$, CI crosses zero) when each task needs fresh exploration; lazy MCP reaches -23.7% here vs. -10.9%/-8.4% on report/refactor. The turn counts pattern

Table 4. Refactor task family (N=30). Same statistics as Table 3.

Primitive	Mean	sd	$\Delta\%$	95% CI	p
Baseline	0.686	0.066	—	—	—
Script	0.653	0.051	-4.3	$[-7.4, -1.2]$	**0.001**
CLI	0.342	0.056	$\mathbf{-49.8}$	$[-52.7, -46.8]$	**<0.001**
MCP (eager)	0.357	0.050	$\mathbf{-47.7}$	$[-50.5, -44.7]$	**<0.001**
MCP (lazy)	0.619	0.153	-8.4	$[-16.6, +2.1]$	<0.001
Hook	0.050	0.042	$\mathbf{-92.6}$	$[-94.6, -90.3]$	**<0.001**

Table 5. Code Q&A task family (N=30). Same statistics as Table 3.

Primitive	Mean	sd	$\Delta\%$	95% CI	p
Baseline	0.354	0.045	—	—	—
Script	0.359	0.043	$+2.8$	$[-3.1, +9.1]$	0.708
CLI	0.177	0.022	$\mathbf{-49.0}$	$[-51.6, -43.2]$	**<0.001**
MCP (eager)	0.148	0.018	$\mathbf{-57.5}$	$[-59.6, -53.0]$	**<0.001**
MCP (lazy)	0.266	0.013	$\mathbf{-23.7}$	$[-27.0, -20.4]$	**<0.001**
Hook	0.033	0.021	$\mathbf{-90.4}$	$[-91.9, -84.6]$	**<0.001**

with the cost: on Q&A, lazy-MCP turns per task (4.2) match eager-MCP (4.1) and undershoot baseline (4.8); on report and refactor, lazy MCP turns track baseline (6.2 vs. 6.4, 10.8 vs. 10.1). Plausibly the agent discovers the lazy schema only on Q&A because tool-shaped verbs ("find," "trace," "detect") cue `tools/list` queries that report/refactor phrasings do not. We do not have direct logs to confirm. Oracle ($\geq$ 80% key-term match) passes at 99.3% overall.

6. Analysis

Our CLI result is the cleanest test of the schema-vs-locality question. CLI runs in the same agent-spawned `Bash` subprocess as the script primitive, yet saves $25\times$ more cost ($\Delta = -56.4\%$ vs. -2.2%). The only difference is what the agent must read to decide how to call the primitive: script requires reading the $\sim$50-line body; CLI requires only the 12-line `--help` output; eager MCP requires only the already-exposed schema.

This isolates schema availability as the relevant cause. The CLI and eager-MCP 95% bootstrap intervals on $\Delta\%$ overlap almost entirely ($[-60.0, -52.1]$ vs. $[-59.7, -52.3]$), so once a compact interface description is reachable, where execution happens contributes no measurable savings on this task. The hook condition gains a second factor by also eliminating the tool-call round trip, taking another $\sim 55\%$ off the eager MCP cost.

The lazy-MCP condition isolates the *discoverability* axis, which prior CLI-vs-MCP benchmarks could not separate from availability. Lazy MCP exposes the same schema as eager MCP; it just makes the agent ask for it. That extra step (or, more often, the agent's failure to ask, falling back to a Bash one-liner) costs roughly $5\times$ the savings: -10.9% vs. -56.3%. Practitioner benchmarks that report $32\times$ or $100\times$ savings (OnlyCLI, 2025; Speakeasy, 2025) measure the eager regime, where the schema is always announced; Claude Code's default is the lazy regime. A pure-announcement variant (App. B, Table 9) reaches -45.8%, showing that announcement-as-discovery alone accounts for 77% (-34.8pp) of the eager-vs-lazy gap; the use-instruction in our eager addition contributes the remaining 23% (-10.5pp).

Discoverability accounts for most of the script-to-eager-MCP gap; availability fills the script-to-CLI sub-gap, and pre-execution adds a separate factor on top.

Practitioner reports (OnlyCLI, 2025; Speakeasy, 2025; Hasan et al., 2026) report the same direction, as does the progressive-disclosure design of Anthropic's Agent Skills system (Anthropic, 2025b); the contributions over those works are listed in Section 1.

The three axes track decision volume. The three axes look like distinct knobs, but in our data they all move together with how many decisions the agent makes. Agent turn counts (number of inference rounds per 5-task seed) correlate with cost across all six primitives and all three families: Pearson r=0.92 on report, r=0.88 on refactor, r=0.71 on Code Q&A (Figure 2). The Q&A correlation is weaker because hook turns there cluster tightly at exactly 5 regardless of cost variance, compressing one of the two endpoints; the linear relationship still holds over a $\sim 10\times$ cost range across primitives. Turn count is not itself a cost component—a single turn can emit few or many tokens—so the correlation is not a price-identity artefact. Output tokens correlate similarly but are partly mechanical, since output enters the cost formula directly; we read the turn-count correlation as the cleaner signal.

The information content of each task's actual answer is at most a few hundred tokens; the bulk of the agent's work is reasoning about *how to act*. Each axis removes a different chunk of that reasoning load: the schema axes shrink what the agent must read to locate the right tool, and pre-execution removes the act-decision step entirely.

An additive re-parameterization (the three axes plus a separate pre-execution scaling term δ that captures the hook ceiling) fitted independently per family yields discoverability $\sim$40%, availability $\sim$12%, locality $\sim$0% on report and refactor (and $+8.5\%$ on Q&A), and δ $\sim$88%. With four free coefficients and four non-trivial primitives the system is exactly determined within a family, so this is a re-parameterization rather than a fitted model; the actual evidence is cross-family stability of the three axis coefficients. We defer the formal model, bootstrap CIs, and a falsifiable prediction for MCP Tool Search to Appendix A.

The qualitative ordering replicates under warm cache and on Claude Opus 4.7. On Claude Haiku 4.5 the ordering holds for every primitive *except* lazy MCP, which collapses to -71% because Haiku proactively issues `tools/list` unaided. We discuss the generality implications in §7; full robustness numbers are in Appendix B.

7. Limitations

Script effect bounded but not zero. At N=30 our minimum detectable effect is $d_z \approx 0.49$. Script's 95% CI on $\Delta\%$ is $[-9.2, +5.0]$ on report (null) and $[-7.4, -1.2]$ on refactor (small but detectable). TOST at $\pm 10\%$ margin rules out a comparable-to-CLI effect on both families ($\max p < 0.05$); at $\pm 5\%$ it does not. A linear mixed-effects model (task fixed, seed random) attenuates the refactor script effect to p=0.14 vs. 0.001 via Wilcoxon, suggesting per-seed aggregation may inflate it. Script's effect, if any, is small.

Cold-cache protocol; one MCP-lazy variant. Per-task UUID injection defeats prefix caching: a worst-case regime. Warm-cache (App. B) preserves the ordering but compresses absolute reductions; lazy MCP collapses to a null and CLI/eager MCP shrink to $\sim -40\%$. Read cold-cache numbers as ranks, not expected savings. Our lazy MCP is the most common shape (stdio server, no system-prompt announcement); MCP Tool Search (Anthropic, 2026) surfaces schemas via active retrieval and may close some of the lazy→eager gap.

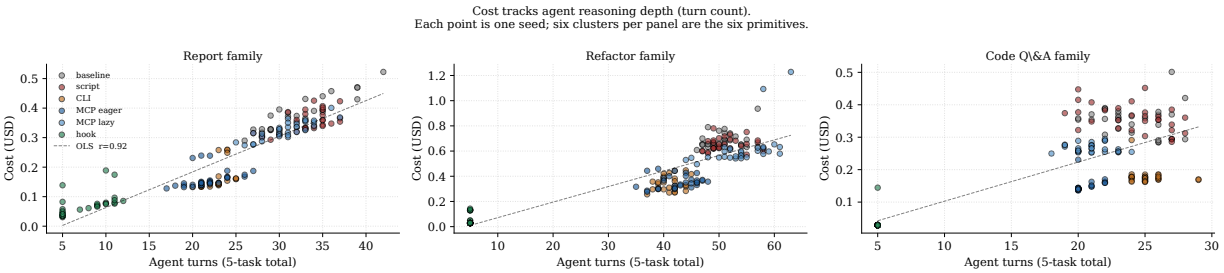

Figure 2. Per-seed cost vs. agent turn count across the three task families. Each point is one seed-primitive cell; six clusters per panel (one per primitive). Turns are an inference-round count, not a cost component.

Workspace-state asymmetry. Script and CLI preserve the workspace between tasks; baseline, MCP, and hook wipe. A *cli-wipe* arm (N=30) that wipes the workspace and re-installs the CLI on each reset matched cli at \$0.161 vs. \$0.160 (paired Wilcoxon p=0.27). Workspace asymmetry is not a meaningful confound on CLI's advantage.

Generality. Three task families (report, refactor, Code Q&A); the spectrum ordering and the CLI≈eager-MCP overlap hold on all three. Other shapes (long-horizon debugging, free-form navigation, multi-turn clarification) may differ. Hooks work only when the task is recognizable from the prompt; mid-conversation hooks have different cost profiles not measured here. We measure Anthropic Sonnet/Haiku/Opus through Claude Code; we have not replicated on MiniMax M2.5, OpenHands, or Cursor CLI.

Discoverability is model-dependent. The cross-model replication (App. B, Table 8) shows lazy MCP collapsing from -5% on Sonnet 4.6 and -12% on Opus 4.7 to -71% on Haiku 4.5: Haiku proactively issues `tools/list` where Sonnet and Opus do not. The discoverability axis therefore reflects model tool-discovery behavior layered on the spectrum, not a fixed structural property of the primitives. Other model families may sit anywhere on this behavior continuum; the eager→lazy gap we report is specific to the deployed model's discovery policy. We treat the ∼45pp announcement effect on Sonnet as a worst-case bound on typical Claude Code installations rather than a universal constant.

Quality-oracle blind spot. The independent LLM-judge audit in §5 catches false-positive leniency the judge can detect from the agent's response *text*. Failures invisible to the response text—e.g. silently truncated outputs that nevertheless parse, or correct-shape answers with wrong content the judge happens to endorse—are not bounded by this audit. We mitigate via a structural oracle (line counts, pytest, key-term match) but the oracle+judge combination still reflects what is checkable from logs, not ground-truth task completion in the strict sense.

Cost model is descriptive, not predictive. The 4-coefficient decomposition in App. A is exactly determined within each family (four free parameters, four non-trivial primitives that vary on the axes), so it is a re-parameterization of the per-primitive means rather than a model with predictive power. Cross-family stability of three of the four coefficients is the empirical evidence that the axes pick out genuine factors. The single falsifiable prediction we offer—retrieval-rate-weighted $\Delta\%(h)$ for MCP Tool Search—is not yet directly testable on a shipped harness. Treat the decomposition as accounting, not theory.

8. Conclusion

Announce a tool, or lose 45 percentage points. The default Claude Code deployment does not announce MCP schemas in the system prompt; on the same server, that default captures only 19% of the savings the eager-announced condition delivers (-10.9% vs. -56.3% on report). A pure-announcement variant of the eager addition, controlling for instruction priming, recovers most of the gap (-45.8%, 77% of the eager-vs-lazy savings; App. B). Compiling a skill into a script saves nothing on its own; locality saves at most a few percentage points once a schema is announced; only a pre-prompt hook reaches the cost floor, by replacing the agent's reasoning loop. The three axes track agent turn count across all primitives and families (r=0.71–0.92), suggesting a single underlying quantity (decision volume), not proving it.

Anthropic's MCP Tool Search (Anthropic, 2026) is the closest near-term falsification target. It is operationally lazy MCP plus active retrieval; on our spectrum its cost should be a retrieval-rate-weighted mix of the lazy and eager endpoints. Using report-family values (-10.9% lazy, -56.3% eager), retrieval hit rate h predicts $\Delta\% \approx -10.9 - 45.4\,h$. At h=0.5 this is $\approx -33.6\%$; a Tool Search measurement above -25% at high h would falsify the discoverability-as-mechanism account.

LLM / Agent Usage Disclosure

The scientific claims of this paper are the author's: the three-axis decomposition (schema availability, discoverability, execution locality), the addition of refactor and Code Q&A as task families, the lazy-MCP condition as a counterpoint to practitioner benchmarks, and the cost-model formulation. A Claude Opus coding agent assisted with implementation (Docker setup, MCP servers, `UserPromptSubmit` hooks, fixtures, quality oracles, statistics and cost-model scripts, figure generators) and prose drafting across review-and-revise cycles. All numerical results were produced by the instrumented experiment and re-verified by the author against raw JSON logs. Source code for all primitive implementations, benchmark harness, prompts, and cost-model fitter is released with the submission.

References

Cai, T., Wang, X., Ma, T., Chen, X., and Zhou, D. Large Language Models as Tool Makers. *ICLR*, 2024.

Yuan, L., Chen, Y., et al. CRAFT: Customizing LLMs by Creating and Retrieving from Specialized Toolsets. *ICLR*, 2024.

Qian, C., Han, C., Fung, Y., Qin, Y., Liu, Z., and Ji, H. CREATOR: Tool Creation for Disentangling Abstract and Concrete Reasoning. *EMNLP Findings*, 2023.

Schick, T., Dwivedi-Yu, J., et al. Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*, 2023.

Packer, C., et al. MemGPT: Towards LLMs as Operating Systems. *COLM*, 2025.

Shinn, N., et al. Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS*, 2023.

Zhao, A., et al. ExpeL: LLM Agents Are Experiential Learners. *AAAI*, 2024.

Park, J. S., et al. Generative Agents: Interactive Simulacra of Human Behavior. *UIST*, 2023.

Wang, G., et al. Voyager: An Open-Ended Embodied Agent with Large Language Models. *NeurIPS*, 2023.

Wang, Z. Z., Mao, J., Fried, D., and Neubig, G. Agent Workflow Memory. *COLM*, 2025.

Xu, W., et al. A-MEM: Agentic Memory for LLM Agents. *NeurIPS*, 2025.

Zhao, Q., et al. Agentic Plan Caching. *NeurIPS*, 2025.

Zhou, C., Zhang, W., et al. Externalization in LLM Agents: A Unified Review. Preprint, 2026.

Zhang, X., et al. Experience Compression Spectrum. Preprint, 2026.

Hasan, M. M., Li, H., Rajbahadur, G. K., Adams, B., and Hassan, A. E. MCP Tool Descriptions Are Smelly: An Empirical Study of Schema Quality and Cost. Preprint, 2026.

Hrubec, N. Reducing Token Usage of Software Engineering Agents. TU Wien thesis, 2025.

Anthropic. Model Context Protocol Specification. https://modelcontextprotocol.io, 2024.

Anthropic. Claude Code Hooks Documentation. https://code.claude.com/docs/hooks, 2025.

Anthropic. Equipping Agents for the Real World with Agent Skills. Engineering blog, 2025.

Anthropic. MCP Tool Search: Progressive Disclosure for Claude Code. Engineering note, 2026.

OnlyCLI Team. The MCP Token Trap: a CLI vs MCP Cost Benchmark. Practitioner benchmark, 2025.

ScaleKit Team. MCP vs CLI: When to Use Each. Practitioner benchmark, 2025.

Speakeasy Team. How We Reduced MCP Token Usage by 100× with Dynamic Toolsets. Practitioner post, 2025.

Zechner, M. MCP vs CLI: Benchmarking Tools for Coding Agents. https://mariozechner.at/posts/2025-08-15-mcp-vs-cli/, 2025.

Fei, X., et al. MCP-Zero: Active Tool Discovery for Autonomous LLM Agents. Preprint, 2025.

Liu, T., et al. Budget-Aware Tool-Use Enables Effective Agent Scaling. Preprint, 2025.

MCPAgentBench Team. MCPAgentBench: Evaluating LLM Agents on 20,000 MCP Tools. Preprint, 2025.

Appendix

Supplementary material; not counted toward the main-paper page limit. Appendix sections are lettered A, B, C.

A. Cost model fit

The qualitative argument in the main text suggests an additive decomposition. We make it concrete: write the percent cost reduction over baseline as a sum of per-axis contributions,

$$\Delta\%(p) \;=\; -\alpha A(p) - \beta B(p) - \gamma C(p) - \delta D(p) \qquad (1)$$

where $A(p) \in \{0, 1\}$ is whether a compact schema is available, B is whether the agent reliably surfaces that schema without active discovery (satisfied when the schema is announced in the system prompt as in MCP eager, or trivially listable as in CLI's `ls`+`--help`; not satisfied when the agent must actively call `tools/list`, as in MCP lazy), C is whether execution happens outside the agent's reasoning loop, and D is whether the primitive pre-executes the work. B is only meaningful when A=1. Each primitive's axis encoding is given in Table 2.

The model has four free coefficients and four non-trivial primitives that vary on the axes (CLI, eager MCP, lazy MCP, hook), so we can solve for $(\alpha, \beta, \gamma, \delta)$ exactly per family. This is a re-parameterization, not a model fit: the four coefficients are an alternate basis for the four per-primitive means. The bootstrap CIs we report are on those underlying means; they do not test additivity itself, which would require an out-of-sample primitive (e.g., a hypothetical lazy CLI, or hook-without-pre-execution) we did not measure. Cross-family stability of three of the four coefficients (Table 6) is the actual evidence; agreement across families suggests the axes pick out genuine factors rather than basis artefacts.

Table 6. Cost-model coefficients (% of baseline saved per axis) fitted independently per task family. 95% bootstrap CIs (10,000 per-seed resamples) reported in text. α=schema available without low-cost surfacing; β=schema surfaced (via `--help` or system-prompt announcement); γ=execution outside loop; δ=pre-execution.

Family	α	β	γ	δ
Report	11.0	45.3	−0.1	80.5
Refactor	10.5	39.3	−2.2	92.6
Code Q&A	15.2	33.8	+8.5	90.4

Tightest 95% bootstrap CIs come from β (e.g. report $[41.6, 49.1]$) and δ ($[76.4, 84.0]$); α has wider intervals ($[6.1, 16.1]$, $[0.2, 18.8]$, $[11.1, 19.3]$); γ CIs are near-zero on report and refactor ($[-2.7, +2.6]$, $[-4.3, +0.0]$) but clearly non-zero on Code Q&A ($[+7.1, +10.0]$). The four axes contribute additively with moderate cross-family drift: α varies $\sim$5pp (10.5–15.2), β $\sim$12pp (33.8–45.3), δ $\sim$12pp on the 80–93% range, and γ shifts from $\sim$0 on report/refactor to $+8.5\%$ on Code Q&A. β is the largest non-pre-execution contributor: making the schema reliably visible to the agent (`--help`-style or system-prompt announcement, indistinguishable on report and refactor), holding everything else fixed, accounts for $\sim$ 40% of baseline cost on average. Schema availability without that surfacing (α) gives only $\sim$12%. γ is null on report and refactor but non-trivial on Code Q&A, where MCP execution outside the agent loop saves an additional $\sim$8.5pp over equivalent in-process CLI invocation; we attribute this to Q&A prompts being tool-shaped enough that a pre-baked MCP tool sidesteps Bash one-liner construction the agent would otherwise attempt. The δ coefficient absorbs everything pre-execution removes, including the schema and locality effects bundled inside the hook mechanism, so the hook-as-a-primitive comparison is partly mechanical.

Falsifiable prediction for Tool Search. Anthropic's MCP Tool Search (Anthropic, 2026) is operationally a lazy MCP with active retrieval: schema is available, locality is outside, announcement is replaced by an `mcp_search` tool. Its $\Delta\%$ should be a retrieval-rate-weighted mix of our lazy and eager endpoints. Using report-family values (-10.9% lazy, -56.3% eager), retrieval hit rate h predicts

$$\Delta\%(h) \;\approx\; -10.9 - 45.4\,h.$$

At h=0.5 this is $\approx -33.6\%$; at h=1.0 it recovers the eager endpoint at -56.3%; at h=0 it collapses to lazy. A Tool Search measurement above -25% at high h, or below -15% at low h, would falsify the discoverability-as-mechanism account. The current Claude Code build does not yet expose an explicit `mcp_search` affordance, so our lazy condition is the closest direct measurement we can offer; it sits at the h=0 endpoint.

B. Robustness checks

Warm-cache replication. Our main protocol injects a per-task UUID into the system prompt to defeat Anthropic's prefix cache. To check that the spectrum is not an artifact of that worst-case cache regime, we re-ran the report family at N=8 with UUID injection disabled (Table 7). The qualitative ordering is preserved; absolute reductions compress as warm cache absorbs some of the overhead the primitives remove. Lazy MCP collapses to a null ($\Delta\% = +0.4$). Script's effect *strengthens* ($\Delta\% = -19.4$ vs. cold -2.2%): a script the agent wrote on task 1 is in the read cache for tasks 2–5, so the warm regime rewards reuse. CLI and eager MCP remain dominant at -38.2% and -46.4%; the hook still reaches the floor at -66.9%.

Table 7. Warm-cache report family (N=8). UUID injection disabled.

Primitive	Mean ($)	sd ($)	Δ%
Baseline	0.296	0.080	—
Script	0.238	0.034	−19.4
CLI	0.183	0.044	**−38.2**
MCP (eager)	0.159	0.032	**−46.4**
MCP (lazy)	0.297	0.043	+0.4
Hook	0.098	0.027	**−66.9**

Cross-model replication. We re-ran the report family at N=8 on Claude Haiku 4.5 and Claude Opus 4.7 (Table 8). The dominant cost-floor primitive (hook, −78% to −88%) and the dominant mid-spectrum band (CLI and eager MCP, −46% to −79%) hold on all three models. Haiku is uniquely effective at finding the lazy-MCP schema on its own, collapsing the lazy MCP penalty from −5% on Sonnet to −71% on Haiku; Opus shows the inverse pattern, with lazy MCP at −12% resembling Sonnet's. Script's effect grows with model capability (−7% Sonnet, −19% Haiku, −25% Opus). The discoverability axis is mediated by the model's tool-discovery behavior, not a fixed structural property of the spectrum.

Table 8. Cross-model report family (N=8 each). Δ% is paired to baseline within each model.

Primitive	Sonnet 4.6	Haiku 4.5	Opus 4.7
Baseline ($)	0.36	0.36	1.06
Script	−6.7	−18.7	−25.0
CLI	**−50.5**	−45.9	**−61.6**
MCP (eager)	**−54.5**	**−78.6**	**−51.7**
MCP (lazy)	−5.1	**−70.6**	−12.0
Hook	**−77.3**	**−81.7**	**−87.7**

Pure-announcement variant: separating announcement from instruction priming. The eager-MCP system-prompt addition used in the main results (§C.1) contains both an announcement ("tool X is available") and a use-instruction ("do NOT write Python scripts"). To separate the two we ran a fifth condition, *pure-announcement MCP*, identical to eager MCP except that the system-prompt addition is reduced to a single sentence: `The MCP tool mcp__build-report__build_report(root, ext, output) is available.` The same server, locality, cache regime, and N=30 seed protocol apply (Table 9).

Table 9. Pure-announcement MCP variant (N=30, report family). Pure mean cost $0.198 (sd $0.025, d_z vs. baseline =−2.94). 95% CI is 10,000-iter bootstrap on per-seed Δ%; Wilcoxon p-values one-sided paired.

Comparison	Δ%	95% CI	p
Pure − baseline	**−45.8**	[−49.1, −42.4]	$< \mathbf{10^{-9}}$
Pure − lazy MCP	**−38.8**	[−41.5, −35.6]	$< \mathbf{10^{-9}}$
Eager − pure	−10.5	[−12.7, −8.5]	$< \mathbf{10^{-9}}$
Decomposition of eager−lazy gap (−45.3pp):			
announcement (pure−lazy)	−34.8pp	77%	
priming (eager−pure)	−10.5pp	23%	

The pure-announcement condition reaches Δ%= − 45.8 (CI=[−49.1, −42.4], d_z= − 2.94), much closer to eager MCP (−56.3) than to lazy MCP (−10.9). The eager-vs-lazy gap of −45.3pp partitions: announcement-as-discovery accounts for −34.8pp (77%); the brief use-instruction adds another −10.5pp (23%, paired Wilcoxon $p<10^{-9}$). The discoverability-as-mechanism account from §1 is upheld as the dominant cause; instruction priming is a real but minor secondary effect.

This also serves as an out-of-sample check on the additivity claim in §A: if the 4-axis model fitted on {baseline, script, CLI, eager, lazy, hook} were truly additive in the original axes, it would predict pure-MCP at the eager-MCP value (−56.3) since pure-MCP shares the same axis encoding as eager (A=B=C=1, D=0). The observed 10.5pp gap means the eager-MCP coefficient β was conflating schema surfacing with use-instruction priming. Refitting β on pure-MCP gives β_{schema}=34.8 and a new fifth axis $\epsilon_{\text{instruction}}$=10.5.

C. Prompts and primitive specifications

This appendix gives the verbatim text used in each experimental condition: §C.1 the `--append-system-prompt` additions per primitive, §C.2 the CLI `--help` outputs the agent reads, §C.3 the report-family CLI source, §C.4 the report-family MCP tool schema, §C.5 the three `UserPromptSubmit` hook intent parsers, and §C.6 all 15 task prompts.

C.1. System-prompt additions per primitive

Lazy MCP attaches the same MCP server as eager MCP but adds no system-prompt text. The eager-vs-lazy contrast reduces to a single `--append-system-prompt` flag. All primitives share one nonce addition (`[run=...-task=<UUID>]`) to defeat prefix caching; that nonce is omitted below.

Eager MCP, report family.

```
An MCP tool mcp__build-report__build_report(root, ext, output) is available. Use this tool directly to produce
any report required by the user; do NOT write bash or Python scripts, and do NOT invoke Bash with wc/awk/find
for counting. Call the MCP tool with absolute paths and return when it succeeds.
```

Eager MCP, refactor family.

```
An MCP server 'refactor' is registered with tools: refactor_rename, refactor_move_class, refactor_keyword_only,
refactor_extract_block, refactor_inline_const. Use these tools directly to perform the refactor described in the
 user prompt; do NOT write Python scripts or use Bash sed for the refactor. Use file paths relative to the
workspace root (e.g. 'src/utils.py'). After applying the refactor with the MCP tool, run pytest tests/ to
confirm and you are done.
```

Eager MCP, Code Q&A family.

```
An MCP server 'codeqa' is registered with five tools: find_callers, class_deps, entry_point, trace_call_chain,
detect_python310_features. Use these tools directly to answer the user's question; do NOT write Python or use
Bash for the analysis. After the tool returns, write the answer to ./answer-0N.txt as the user asked.
```

Hook (all three families). The hook system-prompt addition has the same shape across families and tells the agent to trust the `[precomputed-by-hook]` block that the `UserPromptSubmit` hook injects (§C.5). Report-family text:

```
A UserPromptSubmit hook may pre-execute build-report tasks and inject a [precomputed-by-hook] block into the
user's message. When you see such a block: trust it completely, do NOT call any tools, do NOT write scripts, do
NOT verify, do NOT echo the data back. Respond with a single short line like 'Done.' and nothing else.
```

CLI (all three families). The CLI system-prompt addition tells the agent that a pre-built CLI exists at `./scripts/<name>` and to discover its interface via `--help` rather than reading the source. Report-family text:

```
A pre-built CLI tool exists at ./scripts/build_report. To learn its interface, run './scripts/build_report --
help' (the output is short). Then call it directly. Do NOT Read the source file -- the --help output is
authoritative and much cheaper.
```

Baseline, script, lazy MCP. No system-prompt additions other than the per-task UUID nonce.

Note on announcement vs. instruction. The eager-MCP additions above include both an announcement ("tool X is available") and a use-instruction ("do NOT write Python scripts"). A pure-announcement variant (announcement only, no use-instruction) is reported in App. B, Table 9: announcement-as-discovery alone accounts for 77% of the eager-vs-lazy gap (-34.8pp out of -45.3pp); the use-instruction contributes the remaining 23% (-10.5pp). The discoverability claim is upheld as the dominant cause; instruction priming is a minor secondary effect.

C.2. CLI `--help` outputs

The CLI condition's cost claim hinges on how compact the `--help` output is relative to the script body (§C.3). Each output below is what `./scripts/<name> --help` prints in the agent's `Bash` subprocess.

`build_report --help` (report).

```
usage: build_report [-h] --root ROOT --ext EXT --output OUTPUT

Build a summary JSON report for files matching an extension.

options:
  -h, --help       show this help message and exit
  --root ROOT      Directory to walk (recursive).
  --ext EXT        File extension (with or without leading dot).
  --output OUTPUT  Output JSON file path (pretty-printed).

Example:
  build_report --root ./src --ext py --output ./report-py.json
```

`refactor --help` (refactor).

```
usage: refactor [-h]
                {rename,move-class,keyword-only,extract-block,inline-const} ...

refactor -- small multi-file refactor CLI.

positional arguments:
  {rename,move-class,keyword-only,extract-block,inline-const}
    rename              rename a symbol everywhere it appears (whole-word)
    move-class          move a class definition to another file and fix imports
    keyword-only        add * marker to make selected args keyword-only
    extract-block       extract a contiguous line range into a helper function
    inline-const        replace a literal with a named constant in a constants module

options:
  -h, --help            show this help message and exit

Run `refactor <subcommand> --help` for details on each operation.
```

`code_query --help` (Code Q&A).

```
usage: code_query [-h]
                  {find-callers,class-deps,entry-point,trace,feature-detect} ...

code_query -- small Code-Q&A CLI.

positional arguments:
  {find-callers,class-deps,entry-point,trace,feature-detect}
    find-callers        files calling a named symbol
    class-deps          direct deps of a class within a root
    entry-point         find package entry point
    trace               BFS call chain between two functions
    feature-detect      detect Python 3.10+ features in a file

options:
  -h, --help            show this help message and exit
```

C.3. Script and CLI tool source (report family)

In the CLI condition, the file below ships at `./scripts/build_report` in the workspace before task 1. In the script condition no script is pre-shipped; the agent writes a similar helper itself in task 1 and reuses it across tasks 2–5 ("∼50-line body" in the main text refers to typical agent-written instances; the pre-built reference shown here is 75 lines).

```python
#!/usr/bin/env python3
"""build_report -- summarize files by extension under a directory.

Writes a JSON report with file_count, line_count, byte_count, longest,
shortest, and a sorted files list.
"""
from __future__ import annotations

import argparse
import json
import os
```

```python
import sys


def build_report(root: str, ext: str, output: str) -> None:
    if not ext.startswith("."):
        ext = "." + ext
    files = []
    for dirpath, _, filenames in os.walk(root):
        for fname in filenames:
            if fname.endswith(ext):
                path = os.path.join(dirpath, fname)
                size = os.path.getsize(path)
                with open(path, "rb") as fh:
                    lines = sum(1 for _ in fh)
                files.append({"path": path, "lines": lines,
                              "bytes": size})

    if not files:
        report = {"file_count": 0, "line_count": 0, "byte_count": 0,
                  "longest": None, "shortest": None, "files": []}
    else:
        files_sorted = sorted(files,
                              key=lambda f: f["lines"], reverse=True)
        report = {
            "file_count": len(files),
            "line_count": sum(f["lines"] for f in files),
            "byte_count": sum(f["bytes"] for f in files),
            "longest": {"path": files_sorted[0]["path"],
                        "lines": files_sorted[0]["lines"]},
            "shortest": {"path": files_sorted[-1]["path"],
                         "lines": files_sorted[-1]["lines"]},
            "files": files_sorted,
        }

    with open(output, "w") as fh:
        json.dump(report, fh, indent=2)


def main() -> int:
    parser = argparse.ArgumentParser(
        prog="build_report",
        description="Build a summary JSON report for files "
                    "matching an extension.",
        epilog=(
            "Example:\n"
            "  build_report --root ./src --ext py "
            "--output ./report-py.json\n"
        ),
        formatter_class=argparse.RawDescriptionHelpFormatter,
    )
    parser.add_argument("--root", required=True,
                        help="Directory to walk (recursive).")
    parser.add_argument("--ext", required=True,
                        help="File extension (with or without "
                             "leading dot).")
    parser.add_argument("--output", required=True,
                        help="Output JSON file path "
                             "(pretty-printed).")
    args = parser.parse_args()

    try:
        build_report(args.root, args.ext, args.output)
    except Exception as exc:
        print(f"error: {exc}", file=sys.stderr)
        return 1

    print(f"wrote {args.output}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

Refactor and Code Q&A CLI sources are larger (∼200 and ∼100 lines respectively) and follow the same structure: argparse sub-commands dispatched to operation helpers in `refactor_ops.py` / `code_query_ops.py`. Full sources are in supplementary material.

C.4. MCP tool schema (report family)

The dictionary below is what `tools/list` returns over JSON-RPC. In the eager condition the agent is told this tool exists by the system-prompt addition (§C.1); in the lazy condition the same schema is reachable only by an explicit `tools/list` call the agent has to issue itself.

```
TOOL_SCHEMA = {
    "name": "build_report",
    "description": (
        "Build a summary JSON report for files "
        "matching an extension under a directory. The "
        "report contains: file_count, line_count, "
        "byte_count, longest (path + lines), shortest "
        "(path + lines), and files (sorted array of "
        "{path, lines, bytes}). Writes pretty-printed "
        "JSON to output path."
    ),
    "inputSchema": {
        "type": "object",
        "properties": {
            "root": {
                "type": "string",
                "description": "Directory to walk."},
            "ext": {
                "type": "string",
                "description": ("Extension including or "
                                "excluding dot (e.g. "
                                "'.py' or 'py').")},
            "output": {
                "type": "string",
                "description": "Output JSON file path."},
        },
        "required": ["root", "ext", "output"],
    },
}
```

Refactor and Code Q&A MCP servers expose 5 tools each, with analogous schemas; full sources in supplementary material.

C.5. Hook intent parsers

The `UserPromptSubmit` hook receives a JSON envelope on stdin containing the user prompt and emits text injected as additional context. The central object is a `parse_intent` function that classifies the prompt by keyword/regex and dispatches to a pre-built operation. If parsing fails the hook is a no-op and the agent runs normally. The three parsers cover all 5 tasks per family.

Report family.

```
def parse_intent(prompt: str) -> dict | None:
    """Return {ext, output} if it's a build-report task."""
    ext_match = re.search(
        r"`?\.?(py|js|md|rs|go)`?\s+"
        r"(?:files?|sources?|extension)",
        prompt, re.I)
    if not ext_match:
        ext_match = re.search(r"\.(py|js|md|rs|go)\b", prompt)
    if not ext_match:
        return None
    ext = ext_match.group(1).lower()

    out_match = re.search(
        r"`?\.?/?(report-[a-z]+\.json|counts-[a-z]+\.txt)`?",
        prompt)
    if not out_match:
        return None
    output = out_match.group(1)
    return {"ext": ext, "output": output}
```

Refactor family.

```
def parse_intent(prompt: str) -> tuple[str, dict] | None:
    """Return (kind, params) if recognized, else None."""
    p = prompt.strip()
```

```
    # task 1: rename
    if re.search(r"\brename\b", p, re.I):
        quoted = re.findall(r"`(\w+)`", p)
        m_to = re.search(r"\bto\s+`(\w+)`", p)
        if m_to and quoted:
            new_name = m_to.group(1)
            old_name = next(
                (q for q in quoted if q != new_name), None)
            if old_name:
                return "rename", {"old": old_name,
                                  "new": new_name}

    # task 2: move-class
    if (re.search(r"\bmove\b", p, re.I)
            and re.search(r"\bclass\b", p, re.I)):
        cls_m = re.search(r"class\s+`?(\w+)`?", p, re.I)
        if not cls_m:
            cls_m = re.search(r"`(\w+)`", p)
        from_m = re.search(
            r"\bfrom\s+`?([^`\s,]+\.py)`?", p, re.I)
        to_m = re.search(
            r"\b(?:to|into)\s+`?([^`\s,]+\.py)`?", p, re.I)
        if cls_m and from_m and to_m:
            from_stem = Path(from_m.group(1)).stem
            to_stem = Path(to_m.group(1)).stem
            return "move_class", {
                "cls": cls_m.group(1),
                "from_file": f"src/{from_stem}.py",
                "to_file": f"src/{to_stem}.py",
            }

    # task 3: keyword-only
    if (re.search(r"keyword[-\s]only", p, re.I)
            or re.search(r"\(\s*\w+\s*,\s*\*\s*,", p)):
        sig_m = re.search(
            r"`?(\w+)\s*\(\s*(\w+)\s*,\s*([^)]+)\)`?", p)
        if sig_m:
            func = sig_m.group(1)
            other_args = [
                a.strip()
                for a in sig_m.group(3).split(",")
                if a.strip() and a.strip() != "*"]
            file_m = re.search(
                r"`?\.?/?src/?(\w+)\.py`?", p)
            if file_m:
                return "keyword_only", {
                    "file": f"src/{file_m.group(1)}.py",
                    "func": func, "args": other_args}

    # task 4: extract block
    if (re.search(r"\bextract\b", p, re.I)
            and re.search(r"_validate_batch", p)):
        return "extract_block", {
            "file": "src/processor.py",
            "name": "_validate_batch", "args": "batch"}

    # task 5: inline constant
    if (re.search(r"DEFAULT_TIMEOUT", p)
            and re.search(r"\b30\b", p)):
        return "inline_const", {
            "name": "DEFAULT_TIMEOUT", "value": "30",
            "constants_file": "src/constants.py",
            "context": r"timeout"}

    return None
```

Code Q&A family.

```
def parse_intent(prompt: str):
    p = prompt.strip()
    pl = p.lower()
    # Task 01: callers of a symbol
    if (re.search(r"\b(call|calls|invokes|call sites)\b", pl)
            and re.search(r"`(\w+)`", p)):
        m = (re.search(r"\bfunction\s+`(\w+)`", p)
             or re.search(r"`(\w+)`", p))
        if m and "answer-01" in pl:
```

```
            return ("callers", {"symbol": m.group(1)})
    # Task 02: class dependencies
    if (re.search(r"\b(depend|dependency|dependencies)\b", pl)
            and re.search(r"\bclass\s+`(\w+)`", p)):
        m = re.search(r"\bclass\s+`(\w+)`", p)
        if m and "answer-02" in pl:
            return ("class_deps", {"cls": m.group(1)})
    # Task 03: entry point
    if (re.search(r"\bentry[-\s]?point\b", pl)
            and "answer-03" in pl):
        return ("entry_point", {})
    # Task 04: call chain
    if (re.search(r"\b(call chain|chain|trace)\b", pl)
            and "answer-04" in pl):
        m_from = re.search(r"\bfrom\s+`(\w+)`", p)
        m_to = re.search(r"\bto(?:ward[s]?)?\s+`(\w+)`", p)
        if m_from and m_to:
            return ("trace", {"from_fn": m_from.group(1),
                              "to_fn": m_to.group(1)})
    # Task 05: Python 3.10+ features
    if (re.search(r"python\s*3\.10\+?", pl)
            and "answer-05" in pl):
        m = re.search(r"`?\.?/?src/?(\w+)\.py`?", p)
        if m:
            return ("features",
                    {"file": f"src/{m.group(1)}.py"})
    return None
```

C.6. Task prompts

The 15 task prompts below are the agent's input in each task slot, verbatim. Each seed runs all 5 tasks of one family in sequence within a single primitive condition.

Report family, task 01.

```
Build a summary report for all `.py` files under `./src/`. The report must include:

1. `file_count` -- total number of matching files
2. `line_count` -- cumulative line count across all matching files
3. `byte_count` -- total byte size summed across all matching files
4. `longest` -- object with `path` and `lines` for the file with the most lines
5. `shortest` -- object with `path` and `lines` for the file with the fewest lines
6. `files` -- array of `{path, lines, bytes}` objects, sorted by lines descending

Write the result as a JSON object (pretty-printed, 2-space indent) to `./report-py.json`.

You'll have four more similar report-building tasks for different extensions. The workspace persists, so a
reusable helper in `./scripts/` is encouraged. No narration needed when done.
```

Report family, task 02.

```
I need a summary report for the JavaScript sources under `./src/` (files ending in `.js`). Please give me:

- Total count of matching files (`file_count`)
- Their combined line count (`line_count`)
- Their combined byte size (`byte_count`)
- The largest by lines (`longest`, an object with `path` and `lines`)
- The smallest by lines (`shortest`, same shape)
- A list (`files`) of every matching file with `path`, `lines`, `bytes` -- sorted by lines, biggest first

Save to `./report-js.json` as pretty-printed JSON (2-space indent).

Check `./scripts/` for a helper from earlier work before rolling your own.
```

Report family, task 03.

```
For every Markdown file (`.md`) under `./src/`, compile statistics into `./report-md.json`. I want:

- `file_count`: how many .md files there are
- `line_count`: total lines across them all
- `byte_count`: total bytes across them all
- `longest`: `{path, lines}` of the one with the most lines
```

```
- `shortest`: `{path, lines}` of the one with the fewest
- `files`: detailed list (`path`, `lines`, `bytes` per entry), sorted with the longest first

Output should be pretty-printed JSON (2 spaces).

A reusable helper from a prior task is preferred -- look in `./scripts/` first.
```

Report family, task 04.

```
Generate a stats report for the Rust sources (`.rs`) inside `./src/`. Produce a JSON file at `./report-rs.json`
with these fields:

  file_count     -- integer
  line_count     -- integer (total lines)
  byte_count     -- integer (total bytes)
  longest        -- {path, lines}
  shortest       -- {path, lines}
  files          -- array of {path, lines, bytes}, sorted by lines desc

Pretty-print with 2-space indentation. Use any existing helper from `./scripts/` if one's available.
```

Report family, task 05.

```
Last one: the Go files in `./src/` (that's `.go`). I need a comprehensive report at `./report-go.json` covering:

- file count
- aggregate line count
- aggregate byte count
- the file with the most lines (as `{path, lines}`, field name `longest`)
- the file with the fewest lines (same shape, field name `shortest`)
- the full list of files with path / lines / bytes, sorted from longest to shortest (field name `files`)

JSON, pretty-printed (2-space indent).

A helper in `./scripts/` from earlier tasks is preferred over ad-hoc shell.
```

Refactor family, task 01.

```
In `./src/`, the function `parse_config` (currently in `config_loader.py`) is misnamed -- it doesn't only parse,
it also validates. Rename it to `load_config` everywhere it's defined or referenced. After your edit, every call
 site must use the new name and `pytest tests/` must still pass.

Don't introduce new imports, don't add new tests, and don't touch unrelated symbols. The rename is purely a name
 change.

A reusable helper in `./scripts/` is encouraged for this and the four similar refactor tasks that follow. No
narration when done.
```

Refactor family, task 02.

```
The class `RetryPolicy` currently lives in `./src/utils.py`, but it conceptually belongs in `./src/policies.py` (
which already exists and holds related policy classes). Please move it: cut the class definition from `utils.py`,
 paste it into `policies.py`, and update every import statement in `./src/` and `./tests/` so they import `
RetryPolicy` from `policies` instead of `utils`.

Preserve the class's body exactly. Don't reorder imports beyond what's needed. `pytest tests/` must still pass.

Look in `./scripts/` first for any helper from prior tasks before writing your own.
```

Refactor family, task 03.

```
The function `fetch(url, retries, timeout)` defined in `./src/fetcher.py` currently takes `retries` and `timeout`
 as positional arguments. We want to make those keyword-only -- change the signature to `fetch(url, *, retries,
timeout)` so that callers must spell them out.

Then update every call site in `./src/` and `./tests/` so it passes `retries=...` and `timeout=...` by keyword.
Don't change default values, don't reorder, and don't add docstrings or type hints that weren't there.

`pytest tests/` must still pass after your edit. Reuse anything in `./scripts/` if applicable.
```

Refactor family, task 04.

```
In `./src/processor.py`, the function `process_batch` contains an inline validation block (a sequence of `if not
 ... raise ValueError(...)` checks at the top, before any other logic). Extract that block into a new private
helper `_validate_batch(batch)` defined immediately above `process_batch` in the same file, and replace the
inline checks with a single call to `_validate_batch(batch)`.

The helper should raise the same `ValueError`s with the same messages on the same conditions. Don't change the
public signature of `process_batch`. Don't add type hints that weren't there.

`pytest tests/` must still pass. Helpers in `./scripts/` may be reused.
```

Refactor family, task 05.

```
Throughout `./src/`, the magic number `30` is used in three places to mean "default request timeout in seconds".
 Replace all three occurrences with a module-level constant `DEFAULT_TIMEOUT = 30` defined at the top of `./src/
constants.py` (the file already exists and holds related constants).

Each call site that previously used the literal `30` for that purpose must import `DEFAULT_TIMEOUT` from `
constants` and use it. Don't replace `30`s that mean something else (e.g. a retry count, an array length). `
pytest tests/` must still pass.

Use any `./scripts/` helper from earlier tasks if it fits.
```

Code Q&A family, task 01.

```
List every file under `./src/` that calls (i.e., invokes) the function `parse_config`. The function is defined
in one file but may be called from others.

Write the result to `./answer-01.txt` as one path per line, relative to the project root (e.g. `src/main.py`),
sorted alphabetically. No header, no commentary, no trailing blank line.

Only include files that actually contain a call site like `parse_config(...)` -- do not include the file that
defines `parse_config` itself unless it also recursively calls it.
```

Code Q&A family, task 02.

```
Find every direct dependency of the class `RetryPolicy` defined in `./src/utils.py`. A "direct dependency" is
any class or function name that appears in `RetryPolicy`'s definition (in its decorators, base classes, type
hints, default-value expressions, or method bodies) and is also defined or imported within `./src/`.

Write the result to `./answer-02.txt` as one identifier per line, sorted alphabetically. Only include
identifiers that resolve to symbols defined under `./src/` -- exclude builtins, stdlib, third-party. No
commentary.
```

Code Q&A family, task 03.

```
Identify the package's command-line entry point -- i.e., the function or module that runs when a user does `
python -m thispackage`. Look for the conventional `__main__.py`, or for a function registered as a script entry
point, or for an `if __name__ == "__main__":` block in a top-level module.

Write the result to `./answer-03.txt` as a single line: the path (relative to project root) of the file that
contains the entry point, followed by a colon, followed by the name of the entry-point function (or `__main__`
if it's a module-level block). For example: `src/cli.py:main` or `src/__main__.py:__main__`. No commentary.
```

Code Q&A family, task 04.

```
Trace the call chain from `parse_config` (defined in `./src/config_loader.py`) toward `process_batch` (defined in
 `./src/processor.py`). A valid chain is a sequence of function names where each function in the sequence
directly calls the next, starting from `parse_config` and ending at `process_batch`.

Multiple intermediate hops may exist; find one valid call chain.

Write the result to `./answer-04.txt` as one function name per line in order of call, starting with `
parse_config` and ending with `process_batch`. For example:

  parse_config
  load_session
  process_batch
```

```
If no chain exists, write `none` on a single line. No commentary.
```

Code Q&A family, task 05.

```
Identify which Python 3.10+ language features are used in `./src/fetcher.py`. Look for: structural-pattern-
matching (`match ... case`), the walrus operator `:=`, parenthesized context managers (`with (a, b):`), or new
union syntax (`int | str` outside a type alias).

Write the result to `./answer-05.txt` as one feature name per line, choosing from this fixed vocabulary (any
subset, sorted alphabetically):

- `match-case`
- `parenthesized-context-managers`
- `union-syntax`
- `walrus`

If `fetcher.py` uses none of these features, write `none` on a single line. No commentary.
```