SkillOpt: Trajectory-Derived, Verifier-Grounded Compilation of LLM-Agent Skills

Rohan Gurunandan Rao
NVIDIA
Santa Clara, California, USA

Vineeth Kalluru Srinivas
NVIDIA
Santa Clara, California, USA

Abstract

LLM-agent "skills" (`SKILL.md` files plus optional scripts) inject procedural knowledge at inference time, but *how* that knowledge is packaged determines whether the agent treats it as a tool or as homework. We build SkillOpt, a **verifier-grounded same-task skill compiler** that takes a task's instruction, verifier source, and a no-skill trajectory, consults a Markdown knowledge base of learned patterns, emits a script-maximized skill, validates against the verifier, and iterates with verifier feedback when reward regresses. The optimization is per-task and offline; the resulting artifact is amortized over repeated agent runs of the same task and environment, not claimed to generalize to held-out fixtures. We run SkillOpt across 87 SkillsBench tasks. **It produces** 33 **positive-reward wins (skills that earn nonzero reward at non-regressing reward while reducing cost) with median reductions of** 40% **in tool calls,** 63% **in tokens, and** 62% **in wall-clock latency;** 13 **of these are** $0.0 \rightarrow \geq 0.5$ **verifier-guided rescues where the no-skill agent failed entirely.** An additional 16 "efficient failure" tasks reduce cost while reward remains 0 (a useful but weaker signal), bringing total efficiency-improving runs to 49. The knowledge base grows from 5 seeded patterns to 19, including domain-specific gotchas the optimizer surfaced (e.g. "Project to metric CRS for geospatial distances"). A small input-ablation study on 9 stratified tasks separates the contributions of the trajectory and the KB: the full configuration beats KB-off on 9/9 tasks (lexicographic, reward primary) and beats trajectory-and-KB-off (*naive*) on 7/9, with no-KB-vs-naive splitting roughly evenly — so on this sample each input does measurable work, though neither is strictly necessary on every task. We additionally publish `skill_optimizer.yaml`, a provenance schema that records categorical edits and before/after measurements so future optimizers can publish comparable numbers.

1 Introduction

A skill in the `SKILL.md` ecosystem [1, 2] is a small markdown file (frontmatter plus prose) that an agent loads at inference time, optionally accompanied by scripts and other resources. Skills have spread quickly: at the time of writing, Claude Code, Gemini CLI, Codex CLI, OpenClaw, and Cursor all consume the same format with minor harness-specific differences. The SkillsBench benchmark [3] reports that human-curated skills lift average pass rate by roughly 16 percentage points, but the per-task gains range from +52 pp to *negative* deltas, suggesting that the quality of a skill is at least as important as its presence.

In production, LLM agents are often invoked repeatedly for recurring procedures in fixed software environments — daily data audits, recurring code migrations, scheduled CI checks — where the bottleneck shifts from one-shot generalization to *unit economics*: the same procedure runs hundreds of times against the same verifier, so the operative question is the per-invocation cost. This is what SkillOpt targets.

There are two natural research questions about skill quality. The first is *generation*: given a task and a verifier, can we author a working skill from scratch that approaches human performance? The second is *optimization*: given a skill (or a self-generated trajectory) that already produces correct output, how much further can we compress the agent's work without harming reward? This paper is about the second question. Generating skills from scratch is a complementary, parallel direction; we keep our scope to *optimizing* skills against three measurable axes:

- **Tool calls** — count of agent-issued tool invocations.
- **Token usage** — input, output, cache-read, and cache-write tokens, captured directly from the API exchange.
- **Wall-time latency** — total time from agent start to verifier completion.

As a design constraint, our optimizer produces skills that use script-relative paths and a single CLI entrypoint, so they are *designed* to be portable across Claude Code, Codex, and Gemini CLI; we did not run cross-harness measurements for this submission, so we treat portability as a structural property of the artifacts rather than an empirical claim.

Contributions.

1. **An automated SkillOpt same-task skill compiler** (Section 4) that, given a task's instruction, verifier, no-skill trajectory, and a learned-pattern knowledge base, emits a script-maximized deployable `SKILL.md`+scripts artifact, validates it on the same task's verifier, and iterates up to three rounds with verifier feedback when reward regresses. Optimization is per-task and offline; the artifact's purpose is amortized agent-side cost reduction over future runs of the same task and environment.

2. **A SkillsBench-scale evaluation** (Section 5): across 87 attempted tasks, 33 positive-reward wins (a working skill at non-regressing reward, median −40% tool calls, −63% tokens, −62% latency), of which 13 are verifier-guided rescues from 0.0 to ≥0.5, plus 16 efficient failures that cut cost without earning reward, for 49 total efficiency-improving runs.
3. **Reusable-memory construction via pattern extraction** (Section 6). After every successful optimization, a second LLM call extracts at most one reusable pattern (or returns `SKIP`) and appends it to a single Markdown file. The KB grew from 5 seeded to 19 patterns over the sweep (14 auto-extracted, of which 2 are anti-patterns) without bloating. A 3-arm input ablation on 9 tasks (Full, no-KB, naive) isolates the contributions of the KB and the no-skill trajectory.
4. **Measurement infrastructure for SkillsBench** (Section 2) that captures real per-prompt token counts (input/output/cache) by intercepting the ACP `session/prompt` response payload, which the canonical BenchFlow trajectory dumps did not retain.
5. **A standardized `skill_optimizer.yaml` provenance schema** that records the categorical edits applied (path resolution, tool-call consolidation, dependency minimization, etc.) alongside the before/after measurements, so any future optimizer can publish comparable numbers.

Sections 2–3 build up to the automated optimizer by establishing measurement infrastructure and demonstrating the achievable optimization gap on four hand-tuned skills as an existence proof. Sections 4–6 are the headline: the optimizer, the sweep, and the pattern-extracted KB. Sections 7–8 discuss limits and related work.

2 Background and Setup

2.1 SkillsBench task anatomy

Each SkillsBench task ships an `instruction.md`, a Dockerized environment with input fixtures, a deterministic verifier (`tests/test_outputs.py`) that returns a $[0, 1]$ reward, and `environment/skills/<name>/` containing one or more skills. The default execution path runs an ACP-protocol agent (here, Claude Code via the `claude-agent-acp` npm wrapper) inside the container; on completion the verifier writes a CTRF JSON and a binary reward file.

2.2 Token capture

The Zed Industries `claude-agent-acp` wrapper, used by SkillsBench's `claude-agent-acp` agent, emits a per-prompt `usage` payload containing `inputTokens`, `outputTokens`, `cachedReadTokens`, and `cachedWriteTokens` on the JSON-RPC `session/prompt` response. BenchFlow's `ACPClient` parses this response with a Pydantic model that drops unknown fields, so the canonical `result.json` only reports the tool-call count, not tokens. We instrument `ACPClient._send_request` to capture the raw `usage` dict before model validation, and write it to `trajectory/usage_trajectory.json` alongside the standard ACP trajectory. All token figures in this paper come from this capture. Latency is the `result.json` `timing.total` field, end-to-end wall-clock from agent start through verifier completion — this includes the ≈ 6 s of environment setup and verification overhead, and is therefore the user-visible total, not pure agent execution time. Tables that report latency reflect this same end-to-end wall-clock measurement.

2.3 Optimization categories

Both hand-optimization and the automated optimizer apply a small vocabulary of categorical edits, recorded in each skill's `skill_optimizer.yaml`:

- **Tool-call consolidation.** Replace N agent tool calls (typically a sequence of `Read`/`Bash` invocations) with a single CLI invocation that performs all the work.
- **Path resolution.** Replace harness-specific absolute paths with `os.path.dirname(os.path.abspath(__file__))`-relative paths, harness-portable as a side effect.
- **Dependency minimization.** Use stdlib where the imported library is overkill (e.g. `struct` for binary parsing; no `numpy` for cross-products of length-3 vectors).
- **Constraint preservation.** Move correctness invariants (e.g. "volume is in cubed units of the STL coordinates; do not assume mm") from human-readable prose into the script, where the agent cannot override them.
- **Data inlining.** Embed small, stable lookup tables (e.g. material density) into the script rather than asking the agent to read and parse a separate file.

3 Hand-optimization: an existence proof

Before automating, we verified that meaningful gains exist by hand-optimizing four SkillsBench tasks against the categories above. Table 1 reports the measurements.

What hand-tuning shows. On every task, the SkillOpt skill matches or exceeds the human-authored skill's reward, and on three of four it strictly dominates on every measured metric. The cleanest case is *3d-scan-calc*: the human skill ships a Python class plus a 60-line prose "Critical Note on Units"; SkillOpt collapses both into a single CLI that embeds the density table and writes JSON, halving tool calls and tokens. In a separate stress trial outside the headline cell the human skill returned 0.0 via the exact unit error the prose warning was meant to prevent (34.65 g vs. 34,648.04 g, off by 1000×); SkillOpt scored 1.0 across all four trials — correctness invariants in prose are fragile, the same invariants

Table 1. Hand-optimized headline cases on four SkillsBench tasks (mean±stdev where $N > 1$, with trial count N shown). Reward is the verifier's $[0, 1]$ score; tokens is the sum of input, output, cache-read, and cache-write tokens captured directly from the API exchange. Latency is the BenchFlow `timing.total` field (agent start through verifier completion, including ≈ 6 s of environment overhead). Best value per (task, metric) row block in bold.

Task	Condition	Reward	Tool calls	Total tokens	Latency (s)
3d-scan-calc	no skill (N=1)	0.000	4	50,957	70.4
	human-authored (N=2)	**1.000**	7.5±0.7	101,882±7,287	70.7±0.1
	SkillOpt (N=4)	**1.000**	**4.5±0.6**	**54,336±4,967**	71.3±1.0
econ-detrending-correlation	no skill (N=1)	1.000	9	162,348	105.0
	human-authored (N=2)	1.000	14.5±0.7	271,617±43,259	123.5±20.7
	SkillOpt (N=3)	**1.000**	**5.3±1.2**	**68,160±14,260**	**75.7±1.1**
dialogue-parser	no skill (N=1)	0.833	**5.0**	**165,769**	221.9
	human-authored (N=2)	0.833	8.0±1.4	291,540±60,710	222.8±42.2
	SkillOpt (N=3)	**1.000**	9.0±3.0	173,080±67,330	**83.3±17.2**
invoice-fraud-detection	no skill (N=1)	0.000	12	215,262	164.7
	human-authored (N=2)	**1.000**	12.0±1.4	246,040±37,230	157.1±30.8
	SkillOpt (N=2)	**1.000**	**5.5±0.7**	**77,170±9,900**	**97.0±27.9**

compiled into a script are not. This four-task comparison is illustrative, not a systematic human-authored-skill sweep; the large-scale evaluation below uses the no-skill baseline as the common comparator. As task complexity grows, so does the gap between "skill as documentation" and "skill as compiled procedure". The next step is automating the transformation.

4 The SkillOpt automated optimizer

The system has three components: an *optimizer* (a Claude Opus session that produces the skill), a *knowledge base* (a single Markdown file of learned patterns loaded into every optimizer call), and a *pipeline* (the iterative loop that drives baseline, optimization, and validation, and grows the KB).

4.1 Optimizer

The optimizer is a single `claude -p` headless invocation whose prompt assembles, in order: the task's `instruction.md`, the verifier source (`test_outputs.py`, which is the ground truth for "correct"), the human-authored skill if present, a compacted no-skill trajectory (the agent's own working code from the baseline run, retrieved from the ACP trajectory dump), and the current Markdown knowledge base. It emits two files under `out/`: `SKILL.md` and `scripts/<name>.py`, with the constraints that the script must use self-relative paths, expose a single one-shot CLI, and produce output in the verifier's expected format.

On verifier access: SkillOpt as a task compiler. The optimizer is given the verifier source as input. We conceptualize SkillOpt not as a general-purpose learner but as a **task compiler**: in software engineering the verifier (unit tests) is the executable specification of correctness, and just as a developer uses tests to ground an implementation, SkillOpt uses the verifier as the absolute ground truth to "compile" high-level instructions into optimized scripts. The deployed setting matches: a SkillsBench task ships its verifier as part of the public artifact. The optimizer can absolutely encode test-case-specific behavior; we do not claim distributional generalization, by design. SkillOpt-emitted skills are for fixed-task production deployments, not benchmark-leaderboard generalization. A held-out or drifted-verifier variant — where the optimizer sees example behavior but not the verifier source, or must survive changed tolerances, renamed fields, added assertions, and evolving production verifiers — is a separate robustness claim left to future work; the rescues here are same-verifier wins.

4.2 Iterative loop

The pipeline runs, per task: (1) no-skill baseline trial, capturing reward, tool calls, tokens, and latency; (2) optimizer call, producing a candidate skill; (3) validation trial with the candidate. If reward regresses, the loop re-invokes the optimizer with the verifier stdout and the failed skill appended as "previous attempts." After three failed iterations, the run is recorded as `skillopt_giveup` and the pipeline moves on. Fig. 1 shows the iteration distribution; most successes converge in one round, with the second and third rounds providing meaningful but diminishing rescue. Importantly, the trajectory-feedback loop closes a real gap — the optimizer can read what its previous attempt got wrong from the verifier's own error output, which is more directly grounded than its own prior reasoning.

4.3 Knowledge base extraction

On any successful, efficiency-improving optimization, a second `claude -p` call is invoked with the resulting skill, the

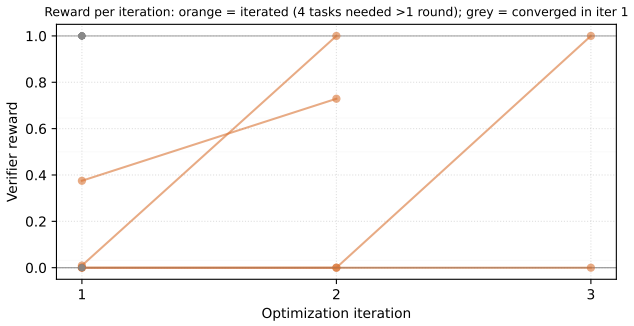

Figure 1. Reward at each optimization iteration for the tasks that needed more than one round. Three converged eventually: `jax-computing-basics` at iter 3; `lab-unit-harmonization` and `pedestrian-traffic-counting` at iter 2. Three exited as `skillopt_giveup` after iter 3 still below baseline reward. The remaining 46 efficiency-improving runs converged in iteration 1 and are not plotted.

trajectory deltas, and the existing KB; it is asked to extract *at most one* new pattern in the form (trigger, transformation, why-it-helps, evidence), or to output `SKIP` if existing patterns already cover the case. The extracted pattern is appended verbatim to `knowledge_base/reflexes.md`. We discuss the resulting growth and category distribution in Section 6.

4.4 Provenance format

Each optimized skill ships a sibling `skill_optimizer.yaml` recording `schema_version`, `skill_name`, an `optimization_record` with baselines and the optimized cell ({`calls`, `tokens`, `reward`}), the list of `categories_applied`, and a `path_resolution` field. The schema is the same for hand- and automated optimization, which is what makes the existence proof in Section 3 comparable to the sweep. We propose it as a *standardized provenance format for agentic-optimization research*: future optimizers can compare not just aggregate gains but *which edit categories produced them* — the part of the comparison that pass-rate-only tables hide. Schema, sample, and per-task records are in the artifact.

4.5 Configuration and reproducibility

Table 2 fixes the configuration. The optimizer model is the latest Claude Opus version reachable via the `claude -p` headless CLI; we do not override the temperature. Each iteration is one `claude -p` call. KB extraction is one further `claude -p` call per successful, efficiency-improving optimization.

What the reported tokens count, and breakeven. Headline savings are the *agent's* trial tokens; optimizer cost is excluded because optimization is one-shot per task while the skill ships into many same-task runs (no held-out generalization claimed, Section 4). Each `claude -p` call sends ~ 75k tokens; 2 calls is the median (30/33 positive-reward wins converged in one optimizer iteration plus the KB-extraction call). On the median positive-reward win the agent saves 308k tokens per use against ~ 150k optimizer overhead, so **a single future same-task run already amortizes the optimizer overhead** (+158k after 1 use, +466k after 2, +1.39M after 5). The 3 `skillopt_giveup` tasks cost ~ 225k tokens each

Algorithm 1 SkillOpt per-task pipeline

Run no-skill baseline trial; record (r_b, c_b, t_b, ℓ_b)
prevAttempts $\leftarrow \emptyset$
for $i = 1$ **to** 3 **do**
    skill $\leftarrow$ ClaudePCall(instruction, verifier, baselineTraj, KB, prevAttempts)
    Run validation trial with *skill*; record (r_s, c_s, t_s, ℓ_s)
    if $r_s \geq r_b$ **and** (*skill* strictly improves any of c, t, ℓ) **then**
        success; ExtractPattern (second `claude -p` call) appends a pattern or SKIP
        return
    end if
    prevAttempts $\leftarrow$ *prevAttempts* $\cup$ {(*skill*, verifierStdout)}
end for
record `skillopt_giveup`

Table 2. SkillOpt configuration. Versions pinned at submission time so the sweep is reproducible.

Hyperparameter	Value
Optimizer model	`claude-opus-4-5` (resolved via `claude -p` headless on 2026-05-03)
Optimizer CLI	Claude Code `2.1.126`
Eval agent (in-trial)	`claude-agent-acp` via BenchFlow, model `claude-opus-4-5`
SkillsBench commit	`f321e9c` (*Move default-excluded tasks out of tasks*)
Temperature	CLI default; not controlled
Max optimization iters	3
Validation timeout	900 s wall-clock per trial
Optimizer budget	1 `claude -p` call per iteration
Baseline trial	no skill loaded; same agent and harness as validation
Validation trial	SkillOpt skill loaded via `bench run --skills-dir`
Retry policy	on regression, append (failed-skill, verifier-stdout) to next prompt
KB-extraction trigger	after each successful efficiency-improving optimization
KB-extraction budget	1 further `claude -p` call (no agent execution)
Concurrency	4 tasks in parallel during the sweep

Table 3. Outcomes across 87 attempted SkillsBench tasks. Each task is counted in exactly one row. **Positive-reward wins** are tasks where SkillOpt produced a working skill (reward>0) at non-regressing reward and strictly improved at least one efficiency axis; this is the conservative headline. **Efficient failures** are tasks where both baseline and SkillOpt scored 0 but SkillOpt was cheaper; they are real efficiency wins but contribute no useful capability. The per-task figures in Sections 5.1–5.2 are computed over the 49 efficiency-improving runs; baseline-failed tasks are excluded from per-task efficiency-delta plots because there is no comparable baseline trajectory.

Outcome	N
Positive-reward wins (working skill at non-regressing reward, cheaper)	**33**
decomposition (subtotals, not separate outcomes):	
strict optimization (baseline reward > 0)	*20*
verifier-guided rescue (0.0 → ≥0.5)	*13*
Efficient failure (both 0, SkillOpt cheaper)	**16**
Tied at reward 1.0 (both succeed; SkillOpt did not strictly improve cost)	4
Tied at reward 0.0 (both failed; SkillOpt not cheaper)	4
SkillOpt regressed reward below baseline	2
`skillopt_giveup` (3 iters, never matched baseline)	3
Baseline-failed (no trajectory; SkillOpt ran from instruction+verifier alone)	8
SkillOpt phase did not complete (mid-pipeline; no usable trial)	4
SkillsBench env infra (Docker / apt / npm; never reached agent)	13
Total efficiency-improving runs (positive-reward + efficient failure)	*49*
Total (excluding subtotals)	87

with no realized saving. **This flips the unit economics of agent deployment**: the "intelligence cost" is paid upfront during one-shot offline compilation, while the marginal cost of subsequent production runs is cut by over 60% in tokens for the median win.

5 Results across SkillsBench

We ran SkillOpt against the 87 attempted SkillsBench tasks (the official runnable subset). Table 3 is a flow-style breakdown of where the 87 tasks ended up; each task lands in exactly one row. Of the 87, 74 reached the agent (the remaining 13 were SkillsBench-side environment failures); of those, 66 produced a no-skill baseline (8 baseline-failed, never produced a usable trajectory); 59 further completed a SkillOpt phase comparable to the baseline (4 no-SkillOpt-phase, 3 `skillopt_giveup`); of those 59, 33 are positive-reward wins and a further 16 are efficient failures, for 49 efficiency-improving runs in total.

5.1 Headline numbers

The conservative headline is the 33 **positive-reward wins** (Table 3, top two rows), where SkillOpt produced a working skill (reward>0) at non-regressing reward while strictly reducing at least one efficiency axis. Across these 33, median reductions relative to the no-skill baseline are −40% tool calls, −63% tokens, and −62% latency. Fig. 2 shows the per-family breakdown: deterministic-procedure-heavy families dominate every efficiency axis, the judgment-required family is the failure mode. The 33 wins comprise two operationally different modes:

- **Strict optimization** (baseline reward > 0, n=20): median −33% tool calls, −60% tokens, −57% latency. The optimizer compresses an already-working trajectory.
- **Verifier-guided rescue** (baseline reward = 0, SkillOpt ≥ 0.5, n=13): median −44% tool calls, −68% tokens, −66% latency. The optimizer is given the same standard inputs (instruction, verifier, the failing no-skill trajectory, human skill if present, and the KB), but the no-skill trajectory contains no working solution; the optimizer must construct one from the verifier and the KB rather than compress an existing trajectory.

A further 16 tasks improve efficiency without earning reward (both baseline and SkillOpt scored 0, SkillOpt cheaper); we report these as "efficient failures" because they reduce wasted spend but do not solve the task. Counting all efficiency-improving runs together, SkillOpt is cheaper on 49 of the 60 comparable pairs (median −44% tool calls, −63% tokens, −57% latency); per-task averages over the 49 are 10.6 fewer tool calls, 598k fewer tokens, and 273 seconds saved.

What this evaluation does and does not show. The reported wins are *same-task, verifier-grounded* compilations: each SkillOpt skill is optimized for one task whose verifier source is in the optimizer's prompt, and we measure the cost of the agent re-running that same task with the new skill loaded. We do *not* claim generalization to held-out fixtures or unseen tasks; the empirical contract is amortized agent-side cost reduction across repeated runs of the same task in the same environment. Fig. 3 reports the second cut over all 59 tasks that completed both phases (33 + 16 + 4 + 4 + 2 = 59), counting each axis where SkillOpt strictly improved *and* did not regress reward; the win-rate fraction differs from the 33-positive-reward headline because the 59-task denominator includes the 8 ties and 2 regressions.

5.2 Where it works, and where it doesn't

The biggest wins are deterministic, computable tasks. Standout reductions include `earthquake-phase-association` 24 → 6 calls, 610k→ 78k tokens; `energy-ac-optimal-power-flow` 30 → 11 calls, 922k→ 156k tokens; `lab-unit-harmonization` 2.5M→ 121k tokens (reward 0.65 → 0.73);

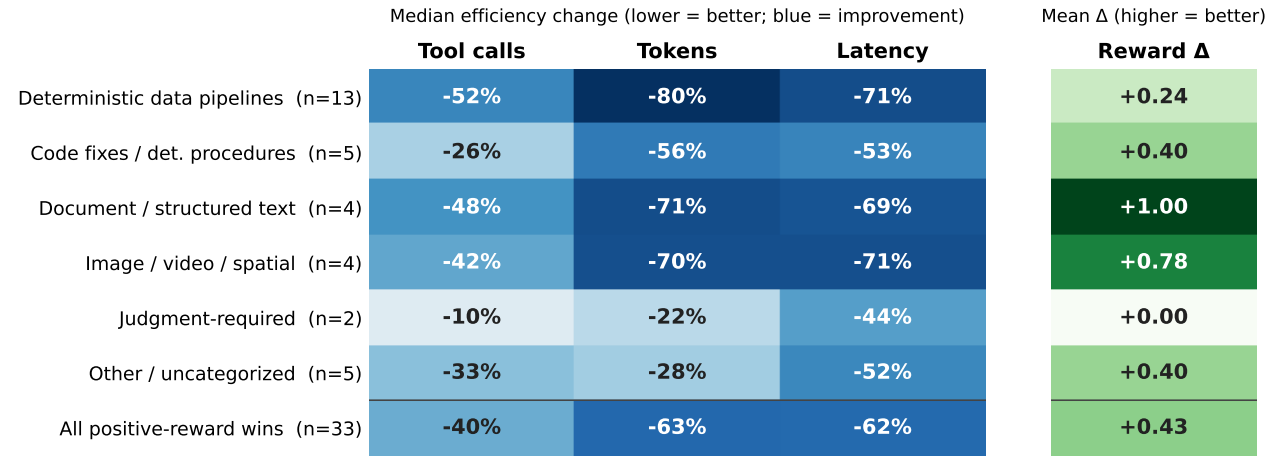

Figure 2. Median efficiency change and mean reward delta across the 33 positive-reward wins, grouped by task family. Negative % (blue) is improvement on calls / tokens / latency; positive Reward Δ (green) means rescues from 0 towards 1. Deterministic-procedure-heavy families dominate every efficiency axis; the judgment-required family is the failure mode (Section 5.2). The bottom row is the headline median across all 33 wins.

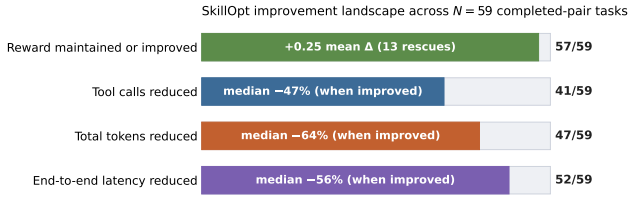

Figure 3. SkillOpt improvement landscape across all N=59 completed-pair tasks. Each bar's filled portion is the count of tasks improved on that axis; the right-side annotation is the count out of 59; the in-bar annotation is the median % reduction conditioned on improvement. Reward row reports mean Δ across the 57 tasks where reward was maintained or improved (the 13 rescues drive the +0.25 mean Δ).

`software-dependency-audit` 1.1M→ 76k tokens (reward 0.0→1.0); `flood-risk-analysis` 594k→75k tokens (reward 0.0 → 1.0). Across rescues, the no-skill agent either depends on a sandbox-unavailable tool (Trivy / npm-audit / external NWS feed) or makes a silent correctness mistake (degrees-as-meters for geospatial distance) that the optimizer's pre-computed or invariant-encoded skill sidesteps.

Where losses come from, and a strategic-optimization pattern. Losses share a shape: tasks whose answer requires model judgment (research-literature classification, prompt design, long-form code review, open-ended debugging). No script encodes the work; SkillOpt's script is a no-op or worse. Within the wins, 4 tasks show a striking pattern in the opposite direction: SkillOpt issued *more* tool calls than the baseline yet *reduced* total tokens, choosing heavier per-call work that paid off in cache-read savings on later calls. SkillOpt optimizes for token density per call and cache utilization, not raw step count, and sometimes *deliberately* adds calls to reduce total cost — a lever human-authored skills typically miss.

6 Knowledge base evolution

Across the sweep, the KB grew from 5 seeded patterns (Section 3) to 19: the 5 originals plus 14 auto-extracted patterns, of which 2 are anti-patterns ("Don't add scripts to documentation-only skills...", "Don't strip dependencies if the task fundamentally requires them."). The growth is monotonic and slows after the first ≈ 25 successful optimizations; on a large fraction of the sweep the extractor returned `SKIP`, indicating an existing pattern already covered the case. We treat this growth as evidence of *reusable memory construction*; for causal evidence we ran a small KB-ablation, reported below.

Fig. 5 plots the per-task efficiency curve across the sweep, which is what we can observe directly. It is suggestive but not causal: a downward trend in token-reduction rank-order would be consistent with the KB helping, but is also consistent with a confound (e.g. the easier wins coming first as the SkillsBench environment stabilized). The ablation below is the actual causal probe.

The patterns cluster into recognizable families: *domain-specific gotchas the LLM gets silently wrong* (e.g. "Project to metric CRS for geospatial distance calculations" — distance on EPSG:4326 returns degrees, not km, which the agent reproduces unless told otherwise; "Encode Azure routing classification mappings when LLM knowledge is unreliable"); *pre-compute X* (when input/parameter set is fixed, run X offline

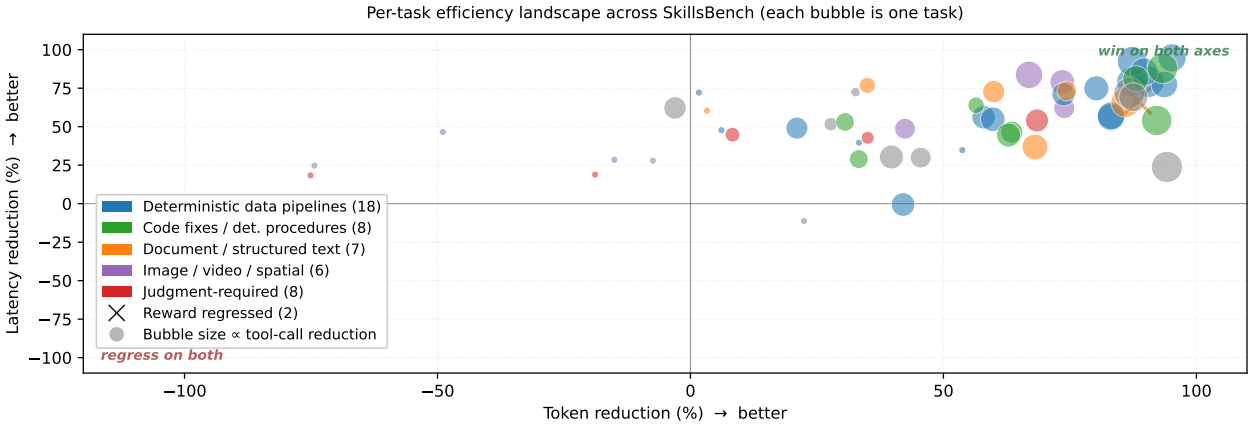

Figure 4. Per-task efficiency landscape across the SkillsBench sweep: each bubble is one task. Token reduction (x) and latency reduction (y) are both "→ better"; bubble size is proportional to tool-call reduction; color is task family; X marks reward regressions (n=2). Wins cluster in the top-right quadrant where SkillOpt reduced both tokens and latency; the judgment-required family (red) drifts left and the regressions sit in the bottom-left, consistent with Fig. 2. The full per-category SkillsBench mapping is in the artifact.

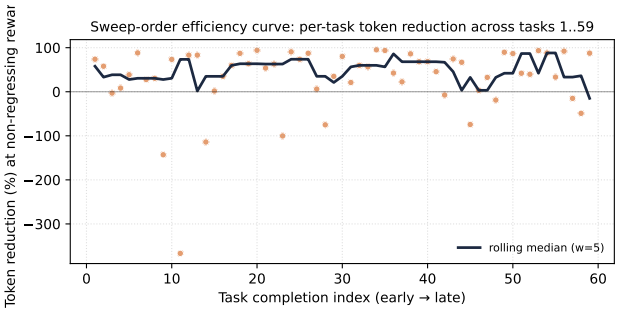

Figure 5. Efficiency learning curve: per-task token reduction (%) at non-regressing reward versus task completion index, with rolling median (window = 5). This plots an outcome (efficiency) per task in completion order; it is not a causal claim about the KB. The KB-growth count itself (5 → 19 patterns, with most flat regions) is reported as a textual claim rather than a separate plot.

and embed the results — calibration on `glm-lake-mendota`, fixed-time-window API queries on `flood-risk-analysis`, batch item mappings on `organize-messy-files`, extraction targets on `paper-anonymizer`, sandbox-unavailable scanners on `software-dependency-audit`); *deterministic procedures encoded as code* (CVE-fix patch scripts; complete Lean proofs; cumulative-state FJSP algorithms); *substrate quirks* (Word placeholder fragmentation across XML runs; range-based unit detection; preferring official CLI tools over raw API calls); and *anti-patterns* ("Don't add scripts to documentation-only skills if the task requires the agent to write code"; "Don't strip dependencies if the task fundamentally requires them").

That a single human-readable Markdown file captures all of this — and is loaded verbatim into every optimizer call — is the most consequential design choice in the system: the optimizer's prior failures and successes shape its future attempts without any fine-tuning. We treat the KB itself as a publishable artifact.

6.1 Input ablation: trajectory and KB

To causally separate the contributions of the no-skill trajectory and the learned-pattern KB, we selected 10 tasks covering the KB pattern families and ran two ablation arms holding everything else fixed: **no-KB** (the optimizer keeps the no-skill trajectory but the KB is blanked to an empty file) and **naive** (the optimizer gets only the instruction and verifier — no trajectory, no KB). One task (`glm-lake-mendota`) hit a Docker infra failure on the no-KB rerun and is excluded; the remaining 9 are plotted in Fig. 6.

Two findings. **The KB does measurable work on this stratified sample.** Full beats no-KB on 9/9, with 3 rescues entirely lost without it (AC-OPF solver setup, file-mapping pre-computation, FJSP cumulative state) and 2 same-reward tasks burning 3.1× and 7.9× more tokens. The KB's effect is not just additional prompt tokens: no-KB and naive differ only in the trajectory, and they trade wins (5/9 vs. 4/9), so the trajectory alone is not sufficient. **The trajectory also contributes:** Full beats naive on 7/9. The two surprises in the other direction — `energy-ac-opf` (naive 1.00/103k vs. Full 1.00/156k) and `paper-anonymizer` (naive 1.00/61k vs. Full 1.00/656k) — are real signal: on these two tasks, the no-skill trajectory was hurting rather than helping the optimizer (the trajectory invited a long-context detour the bare-bones naive prompt avoided). The combined ablation supports a

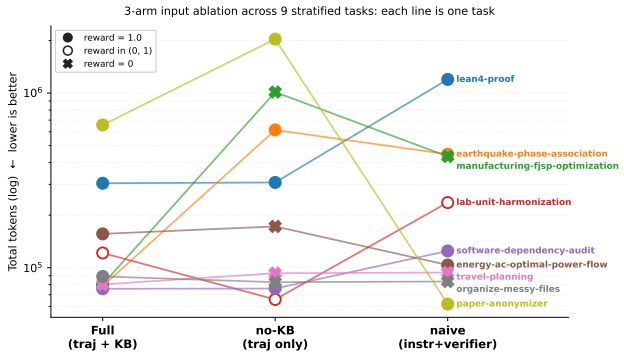

Figure 6. 3-arm input ablation. Each colored line is one of the 9 comparable tasks; y is total tokens (log scale). Marker style encodes reward: filled = 1.0, hollow = in $(0, 1)$, X = 0.0. Full beats no-KB on 9/9 under lexicographic preference (reward primary, tokens secondary). The dramatic upward swings at *no-KB* on `earthquake-phase-association`, `paper-anonymizer`, `manufacturing-fjsp-optimization` are the KB's most visible contributions; `energy-ac-opf` and `paper-anonymizer` are the two tasks where naive beats Full (the trajectory was hurting).

measured claim: on this stratified 9-task sample, each input does real work, though neither is strictly necessary on every task. We treat the small sample as suggestive rather than a tight effect-size estimate, and note that the no-KB and naive arms have not been controlled for prompt-position effects (e.g. a shuffled or inert-token KB).

7 Discussion

Why script-maximized skills win on every axis. Each tool call costs a round-trip latency plus a token sequence (instruction reload, tool-result inclusion, model thinking). Replacing N tool calls with one Bash invocation collapses both costs; tokens drop disproportionately because each saved call also saves the conversation-history tokens of every later call. The `3d-scan-calc` unit-error variance also shows skills doing two jobs simultaneously: procedural compression and *constraint enforcement*. Prose can do the first but not the second; only compiled skills close the gap, which is why several KB patterns are constraint-enforcement (CRS projection, unit detection, Word placeholder rewriting). More broadly, SkillOpt is a shift from a *reasoning-per-run* paradigm to a *compiled-for-deployment* paradigm: by moving computation from online inference into offline compilation, operators can deploy workflows that would otherwise be cost-prohibitive or too latent for real-time use.

Is this just prompt engineering? "Ship the working code" is in spirit a prompt-engineering trick, but SkillOpt differs in three ways: edits are systematic and machine-recorded (the `skill_optimizer.yaml` schema is auditable); gains are large enough (−63% tokens at reward maintained) that the deployment cost question changes in kind, not degree; and the KB-extraction loop is a verifier-grounded growing memory the prompt-engineer-by-hand picture lacks. The input ablation (Section 6.1) shows the memory and the trajectory each contribute. Cross-harness portability is a structural property of the emitted scripts (script-relative paths plus a CLI entrypoint), not an empirical claim in this submission.

Limits of this study. **Infra**: 13/87 tasks (≈ 15%) never reach the agent (SkillsBench Docker compose). **Single-trial sweep**: the 87-task sweep is N=1 per task; in particular, the 13 verifier-guided rescues establish same-run success but not stability under repeated stochastic trials. **Mis-encoded extraction**: rarely (`court-form-filling`) the script fills empty fields correctly but skips required content; some exit as `skillopt_giveup`. **Verifier exploitation and drift**: the optimizer maximizes whatever the verifier rewards, so a buggy verifier is exploited as cleanly as a correct one, and robustness to later verifier changes is unmeasured — same-task amortization is faithful only when the verifier is. **Input ablation small (n=9)**: directional; no shuffled-KB control, so prompt-position effects within the KB are not bounded. **No cross-harness evaluation**: sweep is Claude-Code-only; portability is a structural property, not empirically validated on Codex or Gemini CLI. **Token-pricing**: totals include cache-write, which is provider-specific; breakeven is in tokens, not dollars.

8 Related work and future work

Skill ecosystems and benchmarks. The Agent Skills standard [1] defines the `SKILL.md` format consumed by Claude Code [2], Codex [25], and Gemini CLI [15]; two recent surveys map the lifecycle [7, 35] and Liang et al. [8] argue for structured replacements (we instead optimize the text form empirically). SkillsBench [3] is the substrate; we extend its pass-rate framing with token / tool-call / latency metrics. `awesome-llm-skills` [9] catalogs community-authored skills. SWE-bench [18, 37], AgentBench [20], GAIA [23], and OSWorld [34] are the dominant agentic benchmarks, mostly pass-rate-only.

Verifier-guided iteration and prompt/pipeline optimization. Reflexion [29], Self-Refine [22], LATS [43], Tree of Thoughts [39], and AgentCoder [17] iterate on verifier feedback; APE [44], OPRO [36], ProTeGi [28], Promptbreeder [14], and DSPy [19] optimize text or pipelines using measured fitness. LATM [4] is the closest cost-amortization ancestor: a stronger model builds a tool once so later executions are cheaper; SkillOpt applies that economic pattern to deployable skill directories and measures tool/token/latency reductions. ADAS [16] and AFlow [12] automate agentic system or workflow design. SkillOpt shares the verifier-grounded loop but the target is a fixed-task `SKILL.md`+scripts

directory, not runtime memory, generated code, an instruction string, a DSPy program, or a general workflow.

Memory, knowledge bases, and trajectory mining. Generative Agents [27], MemGPT [26], ExpeL [42], Voyager [32], and Tulip Agent [24] build agent-side memories or skill libraries; Agent Workflow Memory [33], LEGOMem [6], and AgentTuning [40] mine past trajectories into reusable workflows or training data. ACE [41] and Metacognitive Reuse [5] evolve or compress context for recurring LLM behavior. SkillOpt's KB is a single Markdown file loaded into the optimizer (not the task-running agent), verifier-gated, monotonic, and guides skill *generation* for new tasks rather than runtime action selection.

Closest contemporaneous work. Recent 2025–2026 systems require explicit differentiation. **SkillX** [11] auto-constructs hierarchical, agent-internal skill KBs; SkillOpt emits flat directories of deployable artifacts pinned to a task's verifier. **Memento-Skills** [13] stores evolving Markdown skills as agent memory; SkillOpt's outputs are deployable `SKILL.md`+scripts that ship with the task. **SAGE** [10] runs RL over a skill library to reduce tokens/steps; SkillOpt is non-parametric and offline. **ProcMEM**'s Skill-MDP framework treats procedural memory as a learned policy; SkillOpt is a flat verifier-gated retry. **CoEvoSkills** [30], **SkillClaw** [21], **SkillFoundry** [31], and **Federation over Text** [38] study concurrent skill evolution, discovery, or insight sharing. SkillOpt is narrower and more operational: same-task verifier-grounded compilation, portable-by-design `SKILL.md`+scripts artifacts, verifier-gated KB writes, and SkillsBench-scale efficiency measurements.

Future work and artifact availability. Larger ablation with shuffled-KB positional control; cross-harness sweep on Codex/Gemini CLI; skill *generation* on tasks without a baseline trajectory; trajectory-quality pre-filtering (motivated by the 2 tasks where naive beat Full because the trajectory was a long-context detour); a lightweight pre-classifier on the instruction text that gates SkillOpt invocations on judgment-required tasks before burning the ∼ 225k-token `skillopt_giveup` budget; trial multiplicity ($N \geq 3$); held-out or drifted-verifier variants; verifier-quality auditing. We provide optimizer source, per-task skills, sweep + ablation CSVs, the KB, prompts, the BenchFlow patch, and figure scripts as supplementary artifacts.

Acknowledgments

We thank Pradeep Gupta, Zahra Ronaghi, and Janaki Vamaraju for leadership support. We also thank Sridurga Krithivasan, Zhenzhen Li, and Raghu Ramesha for insightful discussions and reviews.

References

[1] Agent Skills Working Group. 2026. Agent Skills Standard. agentskills.io.
[2] Anthropic. 2026. Claude Code Skills. code.claude.com/docs/en/skills.
[3] BenchFlow Team. 2026. SkillsBench: Benchmarking How Well Agent Skills Work. arXiv:2602.12670.
[4] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large Language Models as Tool Makers. arXiv:2305.17126.
[5] Aniket Didolkar, Nicolas Ballas, Sanjeev Arora, and Anirudh Goyal. 2025. Metacognitive Reuse: Turning Recurring LLM Reasoning into Concise Behaviors. arXiv:2509.13237.
[6] Han et al. 2026. LEGOMem: Modular Procedural Memory for Multi-Agent LLM Systems for Workflow Automation. AAMAS.
[7] Jiang et al. 2026. SoK: Agentic Skills. arXiv:2602.20867.
[8] Liang et al. 2026. From Skill Text to Skill Structure: Compositional Representations for SKILL.md. arXiv:2604.24026.
[9] Prat et al. 2026. Awesome LLM Skills. github.com/Prat011/awesome-llm-skills.
[10] Wang et al. 2025. SAGE: Reinforcement Learning for Self-Improving Agent with Skill Library. arXiv:2512.17102.
[11] Wang et al. 2026. SkillX: Automatically Constructing Reusable Skill Knowledge Bases for LLM Agents. arXiv:2604.04804.
[12] Zhang et al. 2025. AFlow: Automating Agentic Workflow Generation. ICLR oral.
[13] Zhou et al. 2026. Memento-Skills: Evolving Markdown Skills as Memory for Agent-Designing Agents. arXiv:2603.18743.
[14] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. ICML.
[15] Google. 2026. Gemini CLI Agent Skills. geminicli.com/docs/cli/skills.
[16] Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. ICLR.
[17] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimization. arXiv:2312.13010.
[18] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR; arXiv:2310.06770.
[19] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714.
[20] Xiao et al. Liu. 2023. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688.
[21] Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. 2026. SkillClaw: Let Skills Evolve Collectively with Agentic Evolver. arXiv:2604.08377.
[22] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651.
[23] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. GAIA: A Benchmark for General AI Assistants. arXiv:2311.12983.
[24] Felix Ocker, Daniel Tanneberg, and Michael Gienger. 2024. Tulip Agent: Open-Ended Agent System with a CRUD-style Tool Library. arXiv:2407.21778.
[25] OpenAI. 2026. Codex Plugins and Skills. openai.com/academy/codex-plugins-and-skills.
[26] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560.

[27] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. arXiv:2304.03442.

[28] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic Prompt Optimization with "Gradient Descent" and Beam Search. EMNLP.

[29] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366.

[30] CoEvoSkills Team. 2026. CoEvoSkills: Co-Evolving Agent Skills. arXiv:2604.01687.

[31] SkillFoundry Team. 2026. SkillFoundry: An Agent Skill Evolution and Evaluation System. arXiv:2604.03964.

[32] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291.

[33] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025. Agent Workflow Memory. ICML.

[34] Tianbao et al. Xie. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. arXiv:2404.07972.

[35] Y. Xu and J. Yan. 2026. Agent Skills for Large Language Models: A Survey. arXiv:2602.12430.

[36] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large Language Models as Optimizers. arXiv:2309.03409.

[37] John et al. Yang. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793.

[38] Dixi Yao, Tahseen Rabbani, and Tian Li. 2026. Federation over Text: Insight Sharing for Multi-Agent Reasoning. arXiv:2604.16778.

[39] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601.

[40] Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. AgentTuning: Enabling Generalized Agent Abilities for LLMs. arXiv:2310.12823.

[41] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, and Jay et al. Rainton. 2025. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv:2510.04618.

[42] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2023. ExpeL: LLM Agents Are Experiential Learners. arXiv:2308.10144.

[43] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. arXiv:2310.04406.

[44] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Models Are Human-Level Prompt Engineers. arXiv:2211.01910.