SKILLCRAFT: CAN LLM AGENTS LEARN TO USE TOOLS SKILLFULLY?

Anonymous authors
Paper under double-blind review

ABSTRACT

Real-world tool-using agents operate over long-horizon workflows with recurring structure and diverse demands, where effective behavior requires not only invoking atomic tools but also abstracting and reusing higher-level tool compositions. However, existing benchmarks mainly measure instance-level success under static tool sets, offering limited insight into agents' ability to acquire such reusable skills. We address this gap by introducing **SkillCraft**, a benchmark explicitly designed to stress-test agents' ability to form and reuse higher-level tool compositions, which we call *Skills*. SkillCraft features realistic, highly compositional tool-use scenarios with difficulty scaled along both quantitative and structural dimensions, designed to elicit skill abstraction and cross-task reuse. We further introduce a lightweight evaluation protocol that lets agents auto-compose atomic tools into executable Skills, cache and reuse them within and across tasks, thereby turning compositional skill acquisition into a directly observable, measurable behavior. Evaluating state-of-the-art agents on SkillCraft, we observe a clear capability gradient: stronger models exploit the same composition affordance to reduce token usage by up to 80%, while weaker ones fail to do so. Moreover, success rate positively correlates with skill execution ability, establishing compositional skill acquisition as a core, separately measurable capability of tool-using agents.

1 INTRODUCTION

> *"The intelligence of a system is a measure of its skill-acquisition efficiency over a scope of tasks, with respect to priors, experience, and generalization difficulty."*
> – François Chollet, *On the Measure of Intelligence*

Real-world tool-using language agents increasingly operate in long-horizon workflows with recurring substructures, such as repeated search–analyze–summarize patterns across documents, repositories, or web services. (Boisvert et al., 2024; Jimenez et al., 2024; Zhang et al., 2025) In cognitive science, such repetition is precisely what gives rise to *skill abstraction*: intelligence is characterized not by executing isolated actions, but by efficiently acquiring, reusing, and recomposing higher-level procedures from experience. In this view, effective behavior requires the ability to form *compositional skills*, which are reusable tool compositions that capture shared structure across tasks rather than repeatedly solving each instance from scratch with flat, atomic tool calls. This raises a fundamental question: *can an agent acquire and reuse such compositional tool skills that generalize across structurally similar tasks?*

Existing tool-using benchmarks (Zhou et al., 2023; Xu et al., 2024; Li et al., 2025) typically *fix* both the toolset and the model at deployment and adopt the paradigm: *Can the agent solve this task with the given tools?* As a result, they provide limited signal on whether agents can accumulate, abstract, and reuse compositional skills across tasks. To isolate and measure this missing capability, we introduce **SkillCraft**, a benchmark with standardized protocols specifically designed to elicit and evaluate reusable tool compositions (Skills) within and across tasks. Unlike existing benchmarks, **SkillCraft** embeds repeated substructures within a single task, requiring agents to identify and reuse tool compositions multiple times within a fixed budget.

We construct **SkillCraft** in a three-stage manner. First, we explore existing tool-using tasks such as Toolathlon (Li et al., 2025), TheAgentCompany (Xu et al., 2024), and WebArena (Zhou et al., 2023)

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

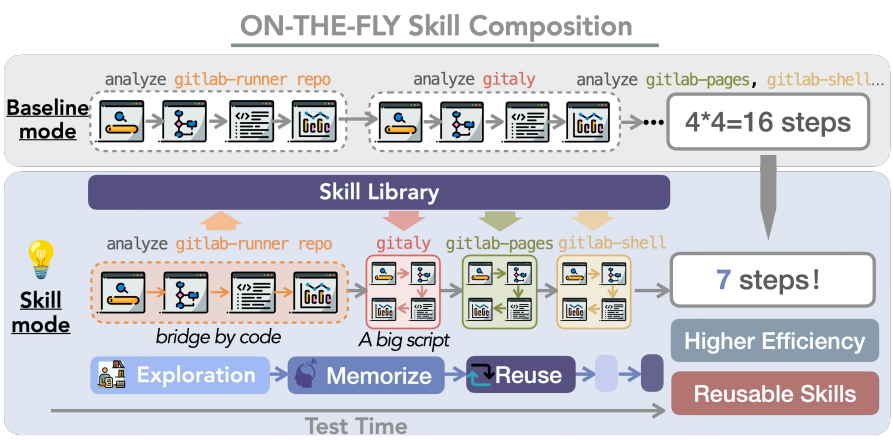

Figure 1: **On-the-fly Skill Composition.** Baseline mode requires $4 \times 4{=}16$ sequential tool calls, while Skill Mode abstracts recurring tool chains into reusable Skills, reducing the interaction to 7 steps.

to identify task design principles. Second, we construct seed tasks by both selecting and adapting high-quality tasks from existing benchmarks and carefully designing long-horizon tasks from scratch. Third, we scale task difficulty along two orthogonal dimensions to encourage tool composition and Skill abstraction. **Quantitative scaling** increases the number of entities involved in a task. For example, a task is extended from "analyze the commit history of repository A" to "analyze five repositories", encouraging the reuse of learned Skills. **Complexity scaling** links multiple subtasks into longer chains, increasing structural difficulty and enabling higher-level skill formation (e.g., fetching commits, identifying contributors, and correlating them). These settings reflect realistic long-horizon tool use, where reusable high-level compositions are essential for efficient and robust problem solving.

In addition, we introduce a protocol to evaluate agents' tool composition ability. We equip agents with a plug-and-play composition mechanism, termed **Skill Mode**, which enables them to (i) automatically discover and cache successful sequences of tool calls as reusable skills, and (ii) invoke these cached skills on new inputs when similar patterns arise. In practice, we achieve this by modifying the system prompt and registering a set of tools that allow agents to save and execute Skills in a plug-and-play manner. This creates test-time tool evolution: agents expand their action space through discovery and reuse *during the test time*, accumulating capabilities during solving tasks.

Using **SkillCraft**, we evaluate state-of-the-art models (e.g., Gemini-3-Pro, Claude-4.5-Sonnet, GPT-5.2) and observe a clear capability gradient: stronger models discover and reuse Skills reliably, cutting token usage by up to 80%, while weaker models fail to exploit the same opportunity. Moreover, efficiency gains correlate strongly with task success, indicating that compositional tool reuse is a model-level capability that SkillCraft isolates and measures.

We further conduct a fine-grained analysis of composition quality along two complementary dimensions: *depth* and *generalization*. We find that deeper, automatically generated hierarchies are often not a reliable scaling strategy—despite high per-skill execution rates, nesting amplifies error propagation and debugging overhead—whereas well-tested, shallow skill libraries remain more robust and cost-effective. In contrast, truly high-quality compositions exhibit strong transferability: skills learned at one difficulty level can be statically reused at other levels (and even across models) with consistently high execution success, improving both success and efficiency.

2 SkillCraft

2.1 What kinds of tasks can evaluate skill composition?

To faithfully evaluate skill composition, tasks must go beyond single-shot, low-branching problems: if a task is solvable with a few atomic calls, agents have no incentive to discover or reuse higher-level skills, and composition ability becomes indistinguishable from baseline tool use. We therefore seek

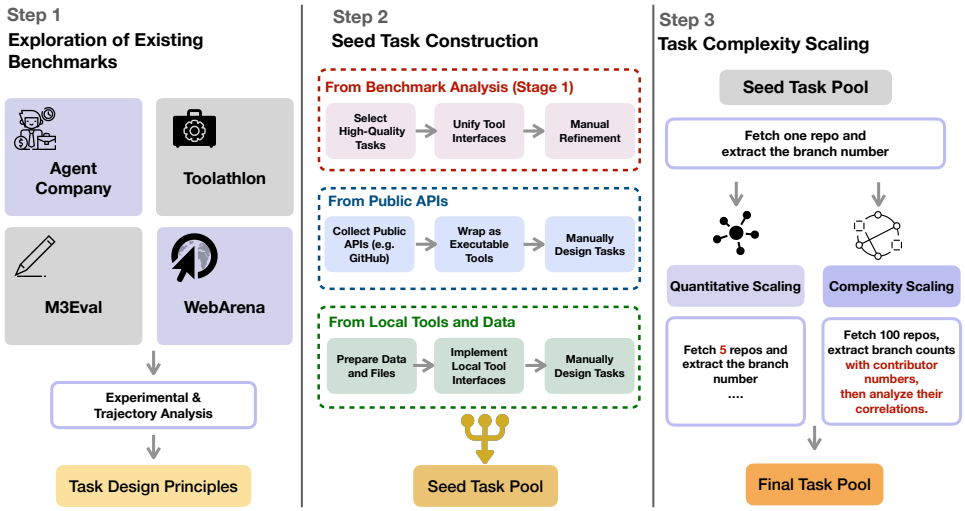

Figure 2: **Three-stage task construction pipeline for SkillCraft.** Stage 1 identifies task design principles from existing benchmarks; Stage 2 constructs seed tasks from three sources (Stage-1 adaptations, handcrafted web APIs, local data processing); Stage 3 scales seed tasks along **quantitative** and **complexity** axes into graduated difficulty levels.

tasks that resemble realistic workflows: long-horizon, structurally repetitive, and challenging enough that instance-by-instance solving is inefficient.

Guided by this motivation, our benchmark design follows two principles. First, tasks should require *multi-step, multi-tool* reasoning, such that no single low-level tool call is sufficient and higher-level compositions provide a clear advantage. Second, tasks should exhibit *recurrent structure with rich entity interactions* across instances, so that a skill discovered in one context can be meaningfully reused in others. This allows us to measure not only whether agents can compose atomic tools, but also whether the composed skills are reusable and generalizable.



Level	Tasks	E×C	%
Easy	63	3 × 3	50.0%
Medium	42	4 × 4	33.3%
Hard	21	5 × 5	16.7%
Total	**126**	–	**100%**

Figure 3: SkillCraft task distribution: 21 task families across six domains and three difficulty levels. E×C denotes entities per task × tool calls per entity.

2.2 How to curate such tasks?

We construct the benchmark through a three-stage pipeline. (1) **Exploratory Phase.** We first sample a set of complex, multi-step tool-using tasks from multiple existing agent benchmarks such as Toolathlon (Li et al., 2025), TheAgentCompany (Xu et al., 2024), WebArena (Zhou et al., 2023) and M3ToolEval (Wang et al., 2024). Through systematic experimentation, we identify useful APIs&task types and gain key insights that guide our task design principles. (2) **Seed Task Creation.** We build the seed task pool from three sources: (i) a small set of high-quality tasks adapted from Stage 1 with reliable, stable, and rate-limit–robust APIs; (ii) a large collection of handcrafted web API tasks; and (iii) local file and data processing tasks based on custom datasets. Stage-1 tasks are converted to a unified MCP interface. For web APIs, we survey, test, and filter stable public endpoints (e.g., GitLab, Open-Meteo, TVMaze), wrap them as standardized local tools, and design tasks accordingly. For local tasks, we prepare datasets, implement standardized processing tools, and construct tasks

Table 1: **Task statistics across the three-stage construction pipeline.** In **Stage 1**, we explore **60+ tasks** from existing benchmarks to identify effective task design principles. In **Stage 2**, we construct **21 seed tasks** from three sources: adapted benchmark tasks, handcrafted web API-based tasks, and local processing tasks. In **Stage 3**, 21 seeds yield **126 tasks**: three Easy, two Medium, and one Hard instance per seed.

Stage	Description	Source	#Tasks
Stage 1: Exploratory Phase	Explore existing benchmarks to identify task design principles	Existing benchmarks like Toolathlon, WebArena, TextArena, M3Eval, etc.	60+
Stage 2: Seed Task Creation	(i) Select & adapt quality tasks from Stage 1	Existing benchmarks	5
	(ii) Handcraft web API-based tasks	GitLab, OpenMeteo, etc.	12
	(iii) Handcraft local processing tasks	Custom datasets & files	4
	Total Seed Tasks		**21**
Stage 3: Systematic Scaling	Scale entity number (N: 3→4→5)	From Seed tasks	**126**
	Scale subtask complexity (M: 3→4→5)	21 seed tasks × 6 instances	

on top of them. (3) **Systematic Scaling.** We expand seed tasks along two axes: (i) **quantitative scaling**, increasing the number of entities/subtasks, and (ii) **complexity scaling**, increasing tool calls per subtask. Combining the two yields multiple difficulty levels (e.g., 3×3, 4×4, 5×5), creating substantial headroom and encouraging discovery and reuse of higher-level compositional skills. Table 1 reports stage-wise statistics, and Fig 3 shows our coverage across application domains and difficulty levels.

2.3 How to Evaluate Tool Composition Ability?

Inspired by cognitive science, which views intelligence as the efficiency of acquiring and reusing skills under limited resources (Anderson, 1982; 1987; Chollet, 2019), we evaluate tool composition not only by task success but also by *efficiency*. In our specific agentic tool-use setting, we also question whether efficiency remains a reliable evaluation metric. As a first step, we analyze the baseline setting to establish a reference point. Our analysis of current models operating with only low-level (atomic) tools reveals two recurring inefficiency patterns: (1) **Redundant state passing:** Intermediate results are repeatedly serialized between consecutive tool calls, incurring substantial token overhead. (2) **Context window saturation:** Long sequences of tool calls and their outputs consume substantial context capacity, potentially causing the model to "forget" earlier information or lose track of the overall goal.

These observations expose a fundamental limitation: complex skills must be decomposed into sequences of atomic operations, each requiring explicit state passing and reasoning. A natural remedy is to **consolidate frequently co-occurring tool chains into a single executable unit**, which we term *Skills*. Code provides a natural medium for this consolidation, compactly representing data flow, control logic, and iteration.

Accordingly, our evaluation asks: given multi-step, multi-tool tasks, can models abstract recurring tool chains into reusable, code-based Skills? Does this abstraction improve efficiency and success, as measured by **token usage**, **tool call count**, and **interaction steps**? We answer these questions by evaluating models on SkillCraft.

3 SkillCraft Protocol

To make Skill Mode portable across models and to expose Skill formation as an observable, measurable behavior, we adopt a minimal MCP-style protocol. The protocol is designed to evaluate

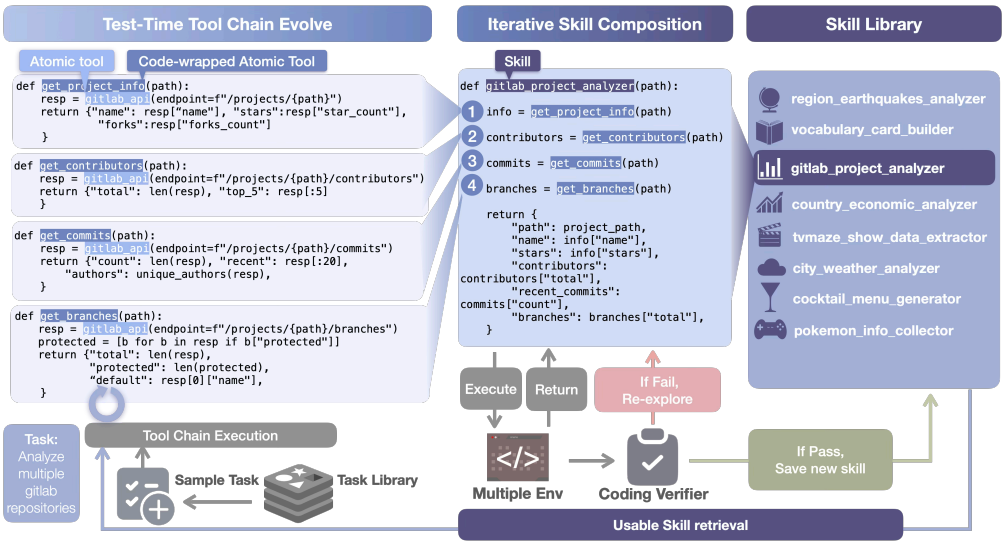

Figure 4: **SkillCraft Protocol Pipeline Overview.** The agent (1) chains atomic tools to solve tasks, (2) abstracts successful sequences into candidate skills verified in a coding environment before storage, and (3) reuses validated skills from a growing library across later tasks.

two complementary capabilities in a quantifiable process: *Composition*, where the agent abstracts multi-step tool chains into reusable, code-based Skills; and *Reuse*, where the agent retrieves and invokes those Skills at test time, enabling graceful execution and accumulating efficiency gains over repeated interactions. Concretely, we maintain a *Skill Library* (a cache of verified, reusable tool compositions) accessed through four primitives: `save_skill` (persist a workflow), `get_skill` (retrieve code and metadata), `list_skills` (enumerate available Skills), and `execute_skill` (invoke a Skill as a higher-level tool). Together these primitives cover the full Skill lifecycle and define the evaluation surface—whether reuse is attempted, succeeds, and how failures are handled are all directly observable as API calls. Per-primitive signatures and the tool diagram are deferred to Appendix B.1 (Figure 8).

A Skill enters the Library only after passing a lightweight *Coding Verifier* that runs three checks: (a) syntax validation, (b) runtime error reporting with structured tracebacks, and (c) post-execution quality detection (e.g., reject Skills whose outputs are mostly `None`/`Unknown`/`0`). This filters trivially broken or silently failing Skills before they pollute the library.

At task time, the agent (1) queries existing Skills via `list_skills` and attempts `execute_skill` with task-specific parameters; (2) on miss or failure, falls back to atomic tools and records the successful trajectory; (3) abstracts the trajectory into a parameterized candidate Skill, passing intermediate state through code variables rather than re-serializing through natural language; and (4) saves it via `save_skill` only after the verifier passes. Execution settings, storage format, and the rationale for why this design improves efficiency are detailed in Appendix B.

4 Evaluation

We evaluate agents on SkillCraft in a consistent and unified setting under the same task prompts, tool endpoints, and environment constraints. Here we introduce our settings.

Models We benchmark a representative set of state-of-the-art models, including Kimi-K2-Thinking (Team et al., 2025), DeepSeek-V3.2-EXP (Liu et al., 2025a), DeepSeek-R1 (Guo et al., 2025a), Gemini-3-Pro (Google DeepMind, 2025), Minimax-M2.1 (MiniMax, 2025), Claude-4.5-Sonnet (Anthropic, 2025), and GPT-5.2 (OpenAI, 2025).

Metrics We measure **Success Rate** using accuracy. For each task, we follow Toolathlon to define a human-expert, handcrafted evaluation rule for matching and scoring the outputs, counting a task

Table 2: Results (base vs skill mode) across models on 126 tasks. **Success Rate (Overall)**: task completion rate (score ≥ 90) for Baseline (no skills) and Skill (with skills) modes, plus **Success Rate (Hard)** for the hard subset only. **Skill Stats**: Exec = skill execution success rate; Reuse = average times each skill is invoked. **Efficiency metrics** (Tokens, Cost, Turns, Tools): per-task averages computed over tasks where *both* modes succeeded; each shows Base, Skill, and Diff values. **Diff**: percentage change (Skill − Baseline) / Baseline; negative values indicate improvement, positive values indicate degradation.

Model	Skill		Tokens			Cost ($)			Turns			Tool Calls			Overall (%)		Hard (%)	
	Exec	Reuse	Base	Skill	Diff	Base	Skill	Diff	Base	Skill	Diff	Base	Skill	Diff	Base	Skill	Base	Skill
Open-Source Models																		
Kimi-K2-Thinking	70%	3.4×	0.51M	0.30M	-41%	0.21	0.13	-38%	6.7	8.3	+24%	16.8	11.9	-29%	55(44%)	56(44%)	8(38%)	7(33%)
DeepSeek-V3.2-EXP	71%	4.8×	1.04M	0.53M	-49%	0.21	0.10	-52%	18.8	15.4	-18%	19.2	14.9	-22%	76(60%)	87(69%)	9(43%)	15(71%)
DeepSeek-R1	62%	3.4×	0.58M	0.41M	-29%	0.24	0.18	-25%	9.0	9.9	+10%	13.4	11.7	-13%	89(71%)	101(80%)	11(52%)	15(71%)
GLM-4.7	91%	3.7×	0.78M	0.48M	-38%	0.20	0.12	-40%	13.5	13.0	-4%	16.9	13.3	-21%	91(72%)	108(86%)	12(57%)	15(71%)
Minimax-M2.1	100%	3.2×	0.42M	0.38M	-10%	0.04	0.04	0%	5.5	5.2	-5%	16.6	15.4	-7%	117(93%)	119(94%)	18(86%)	20(95%)
Closed-Source Models																		
GPT-5.2	84%	3.8×	1.23M	0.26M	-79%	1.77	0.43	-76%	20.6	9.9	-52%	19.4	8.9	-54%	109(87%)	114(90%)	16(76%)	17(81%)
Gemini 3 Pro	93%	3.9×	0.66M	0.30M	-55%	0.59	0.30	-49%	10.5	11.9	+13%	16.0	9.5	-41%	108(86%)	116(92%)	16(76%)	17(81%)
Claude 4.5 Sonnet	81%	3.4×	1.36M	0.40M	-71%	1.08	0.28	-74%	15.3	10.2	-33%	14.3	9.2	-36%	119(94%)	121(96%)	20(95%)	20(95%)

as successful if its final score $\geq 90\%$. To measure Skill behavior beyond task completion, we report **Exec Rate**, the fraction of successful Skill executions among all Skill execution attempts, and **Reusing Rate**, the average number of times each saved Skill is invoked.

For efficiency metrics, we have **InTok/OutTok** (total input/output tokens) and **Turn Num** (LLM interaction rounds), and **Tool_Call Num** when applicable. For each consumption metric m, we compute **Diff** as $(m_{\text{skill}} - m_{\text{base}})/m_{\text{base}}$ (negative indicates savings). To ensure fair comparisons, efficiency metrics are averaged over the subset of tasks where both compared modes succeed.

Results Table 2 shows our main results. **Skill Mode improves success and reduces token use for every model**. Cost falls for all but MiniMax-M2.1 (0% at displayed precision), and tool calls usually fall too. Turns rise slightly for some models (red in Avg Turns) because of extra decision and verification steps, but these turns are typically lightweight. For example, GPT-5.2 improves success from 109/126 (87%) to 114/126 (90%), while cutting average tokens from 1.23M to 0.26M (-79%) and average cost from \$1.77 to \$0.43 (-76%).

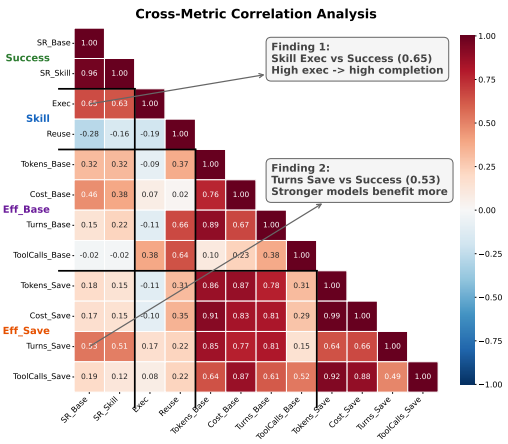

Figure 5: Cross-metric correlation heatmap. Metrics are grouped into four categories: Success, Skill, Eff_Base, and Eff_Save. Key findings: (1) Skill execution rate correlates with task success ($r = 0.65$); (2) baseline success correlates with turn savings from skills ($r = 0.53$).

Cross-metric analysis (Figure 5) shows that **skill execution rate correlates with task success** ($r = 0.65$), while **turn savings correlate with baseline success** ($r = 0.53$); correlations for token, cost, and tool-call savings are weaker. Concretely, *closed-source* models such as Claude Sonnet 4.5 and GPT-5.2—which start from high baseline success (94% and 87%)—achieve the largest token reductions (-71% and -79%). In contrast, *open-weight* models Kimi, DeepSeek, and GLM remain below 90% overall. MiniMax-M2.1 shows only modest savings (-10%), likely because it already solves many tasks efficiently without invoking skills. These findings suggest Skill Mode acts as a capability amplifier, benefiting models that can both synthesize and reliably execute correct skills.

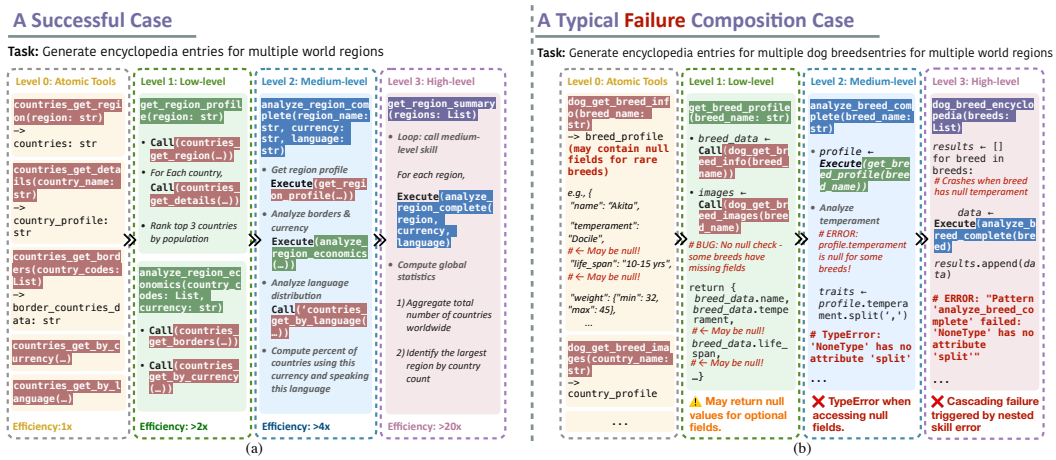

Figure 6: (a) Hierarchical skill composition in Hierarchical mode. A task organized as a depth-3 skill hierarchy, where atomic tools are encapsulated by low-level skills, composed into medium-level skills with additional processing, and orchestrated by a high-level skill. Efficiency gains compound across levels. (b) Error propagation in hierarchical skills. A null value returned by a low-level skill triggers a `TypeError` in the medium-level skill, which cascades into complete failure of the high-level skill. The tree structure amplifies the impact of edge-case bugs.

Our case studies further reveal clear differences in tool composition behavior across models. Stronger models compose tools flexibly, invoking and reusing skills only when beneficial, whereas weaker models tend to follow prompts more rigidly and over-apply composition even when it is unnecessary. Detailed examples are provided in Appendix D.2.

5 What is a good tool composition?

To understand what constitutes a *good* tool composition, we study tool composition along two key dimensions: **composition depth** and **generalization ability**. Specifically, we examine whether deeper, hierarchical compositions lead to better performance, and whether learned Skills can generalize across tasks, complexity levels, and models.

5.1 Is Deeper Composition Always Better?

We compare flat Skill Mode against **Hierarchical Mode**, a variant in which a Skill's `call_tool` can dispatch to either atomic tools *or* previously-saved Skills (via `execute_pattern`), yielding a tree-structured execution graph up to *max_skills_nesting_depth=10* (illustrated in Figure 6(a)). In theory this compounds reuse multiplicatively and lets the agent reason at higher abstraction levels.

Practical Challenges. In aggregate, Hierarchical mode completes fewer tasks than flat Skill mode (313 vs. 322 across three models), despite a gain for DeepSeek and a tie for Claude. The primary reason is **error propagation through the skill hierarchy**. Figure 6(b) illustrates a typical failure pattern: a low-level skill (`get_breed_profile`) returns data with null fields for edge cases, which propagates upward and causes a `TypeError` in the medium-level skill (`analyze_breed_complete`), ultimately cascading into complete failure of the high-level skill (`compile_breed_encyclopedia`).

To illustrate cascading failures arising from implementation details, we identify three underlying micro-level factors: (1) compounding failures, where a skill at depth d depends on its entire dependency subtree and success rate degrades rapidly with nesting; (2) latent bugs, where early-created skills may harbor edge-case errors that only manifest upon reuse, contaminating all higher-level skills built upon them; and (3) debugging overhead, where diagnosing nested failures requires tracing through dependencies—a cost that often exceeds simply re-executing with flat tool calls.

Table 3: Three-mode comparison across models. **Base**: No skill library. **Skill**: With skill library. **Hier**: Hierarchical mode with skill nesting. **Exec**: Skill execution success rate. **Reuse**: Average reuse factor per skill. **Diff**: Relative change vs. Base.

Model	Mode	Success Rate	Skill		Tokens		Cost ($)		Turns		Tools	
			Exec	Reuse	Val	Diff	Val	Diff	Val	Diff	Val	Diff
DeepSeek-V3.2-EXP	Base	76/126 (60%)	–	–	1.04M	–	0.21	–	18.8	–	19.2	–
	Skill	87/126 (69%)	71%	4.8×	0.53M	-49%	0.10	-52%	15.4	-18%	14.9	-22%
	Hier	92/126 (73%)	75%	3.0×	0.68M	-35%	0.10	-52%	16.5	-12%	15.4	-20%
Claude 4.5 Sonnet	Base	119/126 (94%)	–	–	1.36M	–	1.08	–	15.3	–	14.3	–
	Skill	121/126 (96%)	81%	3.4×	0.40M	-71%	0.28	-74%	10.2	-33%	9.2	-36%
	Hier	121/126 (96%)	99%	3.8×	0.63M	-54%	0.44	-59%	11.5	-25%	10.5	-27%
GPT-5.2	Base	109/126 (87%)	–	–	1.23M	–	1.77	–	20.6	–	19.4	–
	Skill	114/126 (90%)	84%	3.8×	0.26M	-79%	0.43	-76%	9.9	-52%	8.9	-54%
	Hier	100/126 (79%)	95%	3.6×	0.60M	-51%	0.81	-54%	11.8	-43%	10.8	-44%

Table 4: Cross-task skill generalization results. **E→H**: Skills learned from easy tasks (e1–e3) transferred to hard tasks (h1). **H→E**: Skills from hard tasks applied to easy tasks. **H→H**: Skills from hard tasks reapplied to the same hard tasks. **Base**: Baseline without skill transfer. **Skill**: With cross-task skill transfer. **Exec**: Skill execution success rate. Efficiency metrics computed over tasks where both modes succeeded. Avg Tokens are in millions.

Model	Setting	Success Rate		Skill	Avg Tokens			Avg Cost ($)		
		Base	Skill	Exec	Base	Skill	Diff	Base	Skill	Diff
Claude 4.5 Sonnet	E→H	20/21 (95%)	21/21 (100%)	100%	1.92	1.56	-19%	1.41	1.07	-24%
	H→E	60/63 (95%)	60/63 (95%)	97%	1.06	0.69	-35%	0.81	0.44	-46%
	H→H	20/21 (95%)	20/21 (95%)	98%	1.96	0.47	-76%	1.46	0.43	-71%
Gemini 3 Pro	E→H	16/21 (76%)	19/21 (90%)	99%	1.33	0.78	-41%	1.26	0.76	-40%
	H→E	55/63 (87%)	60/63 (95%)	100%	0.55	0.36	-35%	0.46	0.30	-35%
	H→H	16/21 (76%)	21/21 (100%)	99%	1.30	0.75	-42%	1.23	0.67	-46%

Empirical Results. Table 3 compares Base, flat Skill, and hierarchical composition. Overall, deeper composition is *not* a consistently beneficial scaling strategy. For a strong model (GPT-5.2), moving from flat Skill to Hierarchy reduces end-to-end success from 90% to 79%, while also weakening token savings (0.26M vs. 0.60M). Even when success does not change (e.g., Claude-4.5-Sonnet remains at 96% in both modes), Hierarchy can still be less efficient (0.40M vs. 0.63M). Notably, Hierarchy often achieves high *Exec* rates (e.g., 95–99%), yet this does not translate into higher task success. Together, these results suggest that *shallow, well-tested skill libraries* are currently more reliable and cost-effective than *deep, automatically generated hierarchies*; realizing the latter likely requires much stronger systematic error handling and compositional verification.

5.2 Cross-task Generalization

We test whether Skills learned at one difficulty level transfer to other levels using **Cross-Task Mode**: Phase 1 creates Skills via standard Skill mode at a source level; Phase 2 copies the cached Skills to target-level tasks (with a `cross_task_skills_summary` injected into the system prompt) and executes *without* allowing further skill modification, isolating pure transfer capability. We evaluate three directions: Easy→Hard, Hard→Easy, and Hard→Hard (within-difficulty within-family).

Empirical Results. Table 4 shows token savings in all three directions for both models. Claude's Hard→Hard preserves 95% success while cutting tokens from 1.96M to 0.47M (-76%); Gemini's Hard→Hard raises success from 76% to 100% while cutting tokens from 1.30M to 0.75M (-42%). Transferred Skills also execute reliably (*Exec* 97–100%), indicating a pre-computed Skill cache reuses well across difficulty levels. All three directions (E→H, H→E, H→H) save tokens, but by different margins, suggesting well-abstracted Skills capture stable substructure independent of entity count or chain depth at creation time.

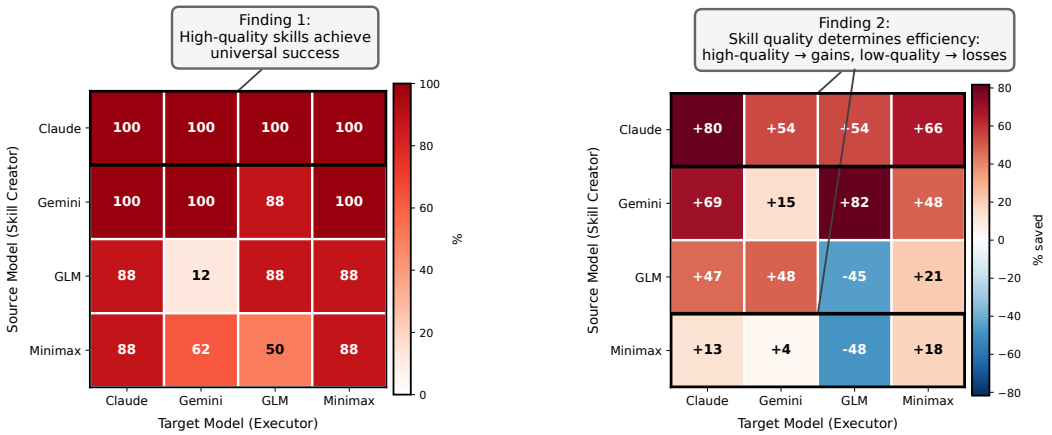

Figure 7: Cross-model skill reuse: success rate (left) and token savings (right) for each creator–executor pair. Blue denotes increased cost and red reduced cost.

5.3 Cross-Model Skill Generalization

A more pointed question is whether Skills produced by one model are useful to *other* models. We conduct a cross-model static-reuse experiment on 8 hard-difficulty tasks across four models (Claude-4.5-Sonnet, Gemini-3-Pro, GLM-4.7, Minimax-M2.1). Each model first creates Skills during its own initial execution; the resulting four Skill caches are then provided to all four models as executors in static-reuse mode (skills can only be invoked, not modified), yielding 16 creator×executor combinations. Figure 7 reports the two heatmaps—success rate and token saving for every cell (i, j); we summarize the qualitative pattern below.

Finding 1: High-quality skills achieve universal cross-model success. Claude-created skills achieve **100% success** across all four executor models, including Gemini, GLM, and Minimax. Well-abstracted skills with clear parameter interfaces transfer effectively across architectures, suggesting that "Skill quality" is a meaningful, model-independent property.

Finding 2: Skill creator quality dictates whether reuse helps or hurts. Token savings vary dramatically by *creator*: Claude-created skills yield substantial savings (up to 80%) on most executor pairs, whereas Minimax-created skills swing from -48% to $+18\%$—i.e., poorly-designed skills can *increase* cost regardless of executor. Strikingly, self-reuse is not always best: Claude executing Gemini's skills reaches 69.2% saving, far above Gemini's own 14.8% self-reuse—executor capability can compensate for moderate skill quality, but cannot rescue fundamentally flawed designs. In practice, multi-model skill libraries should therefore be curated from high-capability creators, since poorly-authored skills propagate cost penalties across all executors.

6 Conclusion

We introduced **SkillCraft**, a 126-task benchmark with recurring substructures, and **Skill Mode**, which exposes compositional skill acquisition by letting agents compose, cache, and reuse tool sequences. Across state-of-the-art models, reuse cuts token use by up to 80% while maintaining or improving success, with larger gains from stronger models. Cross-task skill execution exceeds 95%; cross-model success varies by skill creator, while hierarchies remain vulnerable to accumulated errors. These results establish skill composition as a key capability for efficient, robust long-horizon tool use.

At the same time, our results distinguish the benefits of reuse from the reliability of the skills being reused: transfer can preserve high execution rates, but differences in skill quality and errors in nested compositions can limit end-to-end success. This makes skill creation, validation, and maintenance central parts of tool-use systems rather than one-off preprocessing steps. SkillCraft provides a controlled setting for examining these trade-offs across recurring task structures and difficulty levels.

AI USE STATEMENT

Generative AI tools were used for text polishing and adaptation to the ICLR 2027 conference template. The authors reviewed all changes and take responsibility for the final content.

REPRODUCIBILITY STATEMENT

The appendix documents the benchmark construction, execution budgets, model configurations, prompts, task sources, evaluation protocol, and additional results. Code and benchmark artifacts are available at the anonymous repository: `https://anonymous.4open.science/r/skillcraft-anon-2026-F810/README.md`.

REFERENCES

John R Anderson. Acquisition of cognitive skill. *Psychological review*, 89(4):369, 1982.

John R Anderson. Skill acquisition: Compilation of weak-method problem situations. *Psychological review*, 94(2):192, 1987.

Anthropic. Claude sonnet 4.5 system card, October 2025. URL `https://www.anthropic.com/claude-sonnet-4-5-system-card`.

Anthropic. Agent skills, 2026. URL `https://agentskills.io/home`. Accessed: 2026-05-25.

Chaithanya Bandi, Ben Hertzberg, Geobio Boo, Tejas Polakam, Jeff Da, Sami Hassaan, Manasi Sharma, Andrew Park, Ernesto Hernandez, Dan Rambado, et al. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers. *arXiv preprint arXiv:2602.00933*, 2026.

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks, 2024. URL `https://arxiv.org/abs/2407.05291`.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, et al. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*, 2025.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, et al. Are: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*, 2025.

Xuanqi Gao, Siyi Xie, Juan Zhai, Shiqing Ma, and Chao Shen. Mcp-radar: A multi-dimensional benchmark for evaluating tool use capabilities in large language models. *arXiv preprint arXiv:2505.16700*, 2025.

Google DeepMind. Gemini 3 pro: Model card. Technical report, Google DeepMind, November 2025. URL `https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf`. Published / Model Release: November 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. Mcp-agentbench: Evaluating real-world language agent performance with mcp-mediated tools. *arXiv preprint arXiv:2509.09734*, 2025b.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, et al. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. *arXiv preprint arXiv:2510.25726*, 2025.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Zhiwei Liu, Jielin Qiu, Shiyu Wang, Jianguo Zhang, Zuxin Liu, Roshan Ram, Haolin Chen, Weiran Yao, Shelby Heinecke, Silvio Savarese, et al. Mcpeval: Automatic mcp-based deep evaluation for ai agent models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 373–402, 2025b.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

MiniMax. Minimax m2.1: Significantly enhanced multi-language programming, built for real-world complex tasks, December 2025. URL https://www.minimax.io/news/minimax-m21.

Guozhao Mo, Wenliang Zhong, Jiawei Chen, Xuanang Chen, Yaojie Lu, Hongyu Lin, Ben He, Xianpei Han, and Le Sun. Livemcpbench: Can agents navigate an ocean of mcp tools? *arXiv preprint arXiv:2508.01780*, 2025.

OpenAI. Update to gpt-5 system card: Gpt-5.2. Technical report, OpenAI, December 2025. URL https://cdn.openai.com/pdf/3a4153c8-c748-4b71-8e31-aecbde944f8d/oai_5_2_system-card.pdf.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive tools. *arXiv preprint arXiv:2305.15334*, 2023.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

Guanzhi Wang, Yuhuai Qin, Vineet Kosaraju, Dimitar Lee, Fei Zhang, Percy Liang, Jiajun Chen, Zhenzhong Chen, Ilija Ilievski, et al. Voyager: An open-ended embodied agent in minecraft powered by large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2024. URL https://arxiv.org/abs/2412.14161.

Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, et al. Mcpworld: A unified benchmarking testbed for api, gui, and hybrid computer use agents. *arXiv preprint arXiv:2506.07672*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023.

A Related Work

Tool-use benchmarks mainly differ in the realism of tool executability and in whether tasks require long-horizon composition. In controlled settings, BFCL (Patil et al., 2025) reduces tool use to structured function-parameter prediction, while τ-Bench and ACEBench emphasize multi-turn interaction and correct tool selection under reproducible environments (Yao et al., 2024; Chen et al., 2025). Gorilla and AgentBench broaden tool and domain coverage (Patil et al., 2023; Liu et al., 2023), but primarily evaluate API selection, such that short tool-call chains often suffice.

More realistic benchmarks execute tools in richer environments. AppWorld supports application-level state transitions (Trivedi et al., 2024), and MCP-based suites such as MCPWorld, MCP-RADAR, MCPEval, MCP-AgentBench, LiveMCPBench, and MCPAtlas standardize tool integration across servers (Yan et al., 2025; Gao et al., 2025; Liu et al., 2025b; Guo et al., 2025b; Mo et al., 2025; Bandi et al., 2026), though tasks often remain single-application with simplified initial states. WebArena, OSWorld, SWE-Bench, and TheAgentCompany emphasize long-horizon execution and error recovery in web, desktop, and code workflows (Zhou et al., 2023; Xie et al., 2024; Jimenez et al., 2024; Xu et al., 2024), while GAIA, ARE, and BrowseComp focus on everyday tasks and web-based information seeking (Mialon et al., 2023; Froger et al., 2025; Wei et al., 2025). Tool Decathlon (Toolathlon) further consolidates real tools, fuzzy instructions, execution verification, and cross-application workflows (Li et al., 2025).

On the pipeline side, most tool-using agents follow the “reasoning–acting–observing" loop introduced by ReAct (Yao et al., 2023), where planning and state tracking are repeated at every tool call. CodeAct (Wang et al., 2024) shifts the action space to executable code to express control flow and multi-tool orchestration, but it still regenerates code per task and does not accumulate reusable procedures. Voyager (Wang et al., 2023) and Ghost in the Minecraft (Zhu et al., 2023) show that agents can grow a code skill library through exploration, yet the resulting skills are tied to game rules and state spaces. CREATOR (Qian et al., 2023) abstracts reusable components from patterns but provides limited evidence of robust cross-task generalization in realistic tool ecosystems. Anthropic Skills (Anthropic, 2026) packages workflows as explicit skill modules, but these modules are typically authored and configured by humans rather than induced from execution. In contrast, SkillCraft enables autonomous reuse with a minimal MCP protocol that compiles successful tool sequences into verified executable skills.

B Skill Mode: System Details

B.1 Four primitive tools enabling Skill Mode

We illustrate the detailed design and functionality of the four primitive tools that together enable the proposed Skill Mode in Figure 8.

B.2 Why Skill Mode improves efficiency

Figure 9 illustrates why Skill Mode improves efficiency through two complementary mechanisms. In normal tool use, raw tool outputs (e.g., full webpages or verbose API responses) are repeatedly injected into the context, bloating the prompt with extraneous information and incurring repeated argument-passing costs as the output of one tool becomes the input of the next via the agent. Skill Mode instead extracts and caches only the minimal, task-relevant fields, enabling direct tool-to-tool chaining and allowing intermediate results to be passed once rather than re-serialized at every step. Moreover, by reusing previously discovered tool sequences as atomic skills, the agent amortizes planning and reasoning cost over repeated executions, avoiding the need to reconstruct the same multi-step workflow from scratch.

B.3 Implementation details

This section provides additional implementation details that complement the methodology described in the main text.

Tool 1: save_skill

Description: Save a reusable workflow as executable skill.

Input:

```
skill_name - Unique identifier
script_code - Python script to execute
parameters - List of variable names
description - Human-readable summary
```

Output:

```
"Skill '<name>' saved successfully."
```

When to Use:

When identifying repetitive workflows. External tools can be called inside the script.

Tool 2: execute_skill

Description: Run a saved skill with new arguments.

Input:

```
skill_name - Name of saved skill
args - Dict of arguments
```

Output:

```
{"status": "success", "result": <data>}
{"status": "failed", "result": <error>}
```

When to Use:

For batch tasks with repetitive logic. Replaces sequential calls with skill executions.

Tool 3: list_skills

Description: List all skills saved in current session.

Input:

(No parameters required)

Output:

```
Skill 1: <name> - <description>
Skill 2: <name> - <description>
...
```

When to Use:

Check available skills before creating new or reusing existing ones.

Tool 4: get_skill

Description: Retrieve full source code of a saved skill.

Input:

```
skill_name - Name of target skill
```

Output:

```
{"script_code": "...",
  "parameters": ["path"],
  "version": 1}
```

When to Use:

Inspect or debug a skill before executing it on new data.

Figure 8: Pseudo-code for the four primitive tools that enable Skill Mode.

Execution Configuration. To ensure reproducibility and prevent resource exhaustion, we impose several execution limits on each task. Each task is allocated a maximum of 150 conversation turns (or 300 steps in single-turn mode) and a 60-minute timeout. We enforce cumulative token limits of 1M input tokens and 150K output tokens per task, with individual requests capped at 150K input tokens. Tasks exceeding these limits are terminated and evaluated based on partial completion. For generation, all models use temperature=0.0 and top_p=1.0 to ensure deterministic outputs. We set `tool_choice="auto"` to allow models to decide when to invoke tools autonomously.

Skill Storage and Execution. Skills are persisted as JSON entries in a `skill_cache.json` file within each task's workspace. Each skill entry contains: (1) `script_code`—executable Python code that invokes tools via a `call_tool(name, **kwargs)` interface, (2) `parameters`—a list of input parameter names, (3) `description`—natural language documentation, and (4) `execution_stats`—runtime statistics tracking successful and failed executions.

Evaluation Protocol. We employ a partial-credit scoring system where each task defines multiple weighted evaluation criteria. Typical criteria include: output file existence (10 points), JSON validity

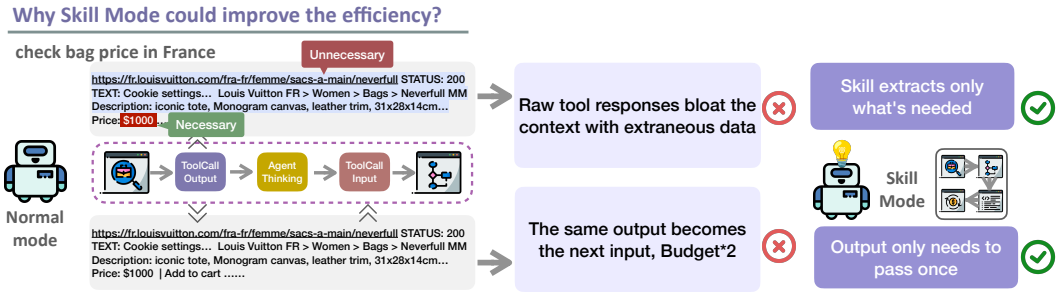

Figure 9: Skill Mode improves efficiency through two mechanisms. First, it reduces argument passing overhead by enabling direct tool chaining (Tool A → Tool B → Tool C) rather than shuttling intermediate outputs through the agent (Tool A → Agent → Tool B → Agent → Tool C). Second, it amortizes planning cost by allowing agents to reuse previously discovered tool sequences, eliminating the need to reason about recurring multi-step patterns from scratch.

(10 points), data completeness (30 points), and field-level accuracy (50 points). A task is considered *successful* if it achieves ≥90% of the maximum score. Efficiency metrics (tokens, cost, turns, tool calls) are computed only over tasks where *both* baseline and skill modes succeed, ensuring fair comparison. All API costs are tracked via the OpenRouter billing API.

C SkillCraft: Benchmark Construction Details

C.1 Task API Sources

We present the complete list of API sources used in SkillCraft benchmark in Table 5. Our 21 task families span six application domains—from entertainment and gaming to science and development—covering a diverse range of real-world API interaction patterns. All APIs are publicly available REST endpoints that require structured multi-step interactions, making them ideal candidates for evaluating skill composition and reuse. For each task family, we implement 5–7 tool functions wrapping distinct API endpoints; difficulty levels (Easy/Medium/Hard) control the number of subtasks (3/4/5) and thus total API calls required per task. Most of these APIs are sourced from existing community-maintained projects, while the Local DNA Analysis task uses a custom implementation for bioinformatics operations.

D Additional Analyses

D.1 Direct execution mode

We further investigate the efficiency impact of script parameterization by implementing **Direct Exec Mode**, an alternative approach that trades generalization capability for execution efficiency.

In our Skill mode, agents create parameterized skills through a two-step process: first `save_skill` to store a reusable script with parameter placeholders, then `execute_skill` to invoke it with specific arguments. This design enables skill reuse across similar tasks but introduces overhead from parameter abstraction and the save-then-execute workflow.

Direct Exec Mode takes a fundamentally different approach. Instead of creating generalizable skills, agents write **single-use scripts** with all values **hardcoded directly** into the code. The agent uses `exec_script` to execute these scripts immediately, after which they are discarded. This eliminates both the abstraction overhead of designing reusable interfaces and the two-step save-execute workflow.

Table 6 compares Base, Skill, and Direct Exec on a 48-task subset. For Claude 4.5 Sonnet, Direct Exec largely preserves success at 96% while cutting tokens from 1.72M to 0.16M, and it reduces turns from 15.7 to 5.8 with tool calls from 14.7 to 4.8. Skill mode is less aggressive at 0.34M tokens and it drops success to 90%. For GPT-5.2, Direct Exec achieves the largest savings from 1.18M to 0.06M tokens and reduces turns from 24.5 to 4.5, but success falls from 94% to 85%, while Skill

Table 5: Complete list of API sources used in SkillCraft. The benchmark comprises 21 task families across 6 domains (Entertainment & Gaming, Reference & Knowledge, Education & Society, Developer & Web, Science & Environment, Food & Lifestyle), matching the categorization in Figure 3. **Tools**: number of distinct API-wrapping functions per task. Each task family includes six instances across three difficulty levels (three Easy, two Medium, one Hard); Easy, Medium, and Hard tasks contain 3, 4, and 5 subtasks, respectively, yielding $21 \times 6 = 126$ tasks. All APIs except Local DNA Analysis are publicly available REST endpoints.

Task Family	Domain	Tools	Source
Cat Facts Collector	Reference	5	https://catfact.ninja
Cocktail Menu Generator	Food	5	https://thecocktaildb.com
Countries Encyclopedia	Reference	5	https://restcountries.com
D&D Campaign Builder	Gaming	6	https://dnd5eapi.co
D&D Monster Compendium	Gaming	6	https://dnd5eapi.co
Dog Breeds Encyclopedia	Reference	5	https://dog.ceo/api
GitLab Deep Analysis	Developer	6	https://gitlab.com/api/v4
Jikan Anime Analysis	Entertainment	5	https://api.jikan.moe
JSONPlaceholder Analyzer	Developer	7	https://jsonplaceholder.typicode.com
Local DNA Analysis	Science	5	Custom Implementation
Name Demographics	Society	5	https://genderize.io
Open-Meteo Weather	Science	5	https://open-meteo.com
PokéAPI Pokédex	Gaming	5	https://pokeapi.co
Random User Database	Society	5	https://randomuser.me
Recipe Cookbook Builder	Food	6	https://themealdb.com
Rick & Morty Explorer	Entertainment	5	https://rickandmortyapi.com
TVMaze Series Analyzer	Entertainment	5	https://api.tvmaze.com
University Directory	Education	5	http://universities.hipolabs.com
USGS Earthquake Monitor	Science	6	https://earthquake.usgs.gov
Vocabulary Builder	Reference	5	https://dictionaryapi.dev
World Bank Snapshot	Education	5	https://api.worldbank.org

Table 6: Three-mode comparison (Base, Skill, Direct Exec) on 48-task subset. **Base**: No skill library. **Skill**: With skill library from previous runs. **Direct Exec**: Skills are directly executed without agent intervention. Efficiency metrics are computed over tasks where both Base and the respective mode succeeded.

Model	Mode	Success Rate		Skill Stats		Avg Tokens		Avg Cost ($)		Avg Turns		Avg Tool Calls	
		Succ	Rate	Exec	Reuse	Val	Diff	Val	Diff	Val	Diff	Val	Diff
Claude 4.5 Sonnet	Base	47/48	98%	–	–	1.72M	–	1.73	–	15.7	–	14.7	–
	Skill	43/48	90%	99%	3.7×	0.34M	-80%	0.22	-87%	10.5	-33%	9.5	-35%
	Direct Exec	46/48	96%	68%	3.1×	0.16M	-91%	0.17	-90%	5.8	-63%	4.8	-67%
GPT-5.2	Base	45/48	94%	–	–	1.18M	–	1.52	–	24.5	–	23.1	–
	Skill	43/48	90%	97%	3.5×	0.26M	-78%	0.39	-74%	8.9	-64%	7.9	-66%
	Direct Exec	41/48	85%	68%	3.1×	0.06M	-95%	0.14	-91%	4.5	-82%	3.5	-85%

keeps 90% at 0.26M tokens. Direct Exec also has lower Exec at 68% versus 97% to 99% in Skill mode, matching the fact that removing the agent loop removes recovery and adaptation. These results show Direct Exec as the efficiency upper bound when Skills transfer cleanly as standalone programs. This advantage stems from two factors: (1) **reduced cognitive load**—the agent need not design generalizable parameter interfaces or anticipate future reuse scenarios; and (2) **simplified execution**—hardcoded values eliminate potential parameter binding errors that can occur in parameterized skill execution.

These results suggest that the generalization capability of Skills incurs a non-trivial overhead. When tasks are isolated and patterns are unlikely to be reused, Direct Exec Mode provides a more efficient alternative.

D.2 TRAJECTORY ANALYSIS

Traj A: Claude 4.5 Sonnet *cat-facts-collector/e2*

System
You are an AI assistant with access to tools. Use save_skill to create reusable workflows and execute_skill to run them.

User
Collect information for 3 cat breeds (Persian, Siamese, Maine Coon) using breed_profile, breed_facts, and breed_encyclopedia APIs. Save results to a JSON file.

Step #1 Agent
I'll process each breed systematically using direct API calls.
call *catfacts_breed_profile* *Persian*

{"origin": "Persia", "temperament": "Gentle", "life_span": "12-17"...}

Step #2 Agent
call *catfacts_breed_facts* *Persian*

{"fact_1": "Persians are known for their long fur"...}

Step #3 Agent
call *catfacts_breed_encyclopedia* *Persian*

{"description": "The Persian cat is a long-haired breed..."...}

Step #4 Agent
Persian complete. Continuing with Siamese...

Steps #5–7: Same 3 API calls for Siamese and Maine Coon

Step #8 Agent
call *write_file* *cat_breeds_report.json*

File written successfully.

Step #9 Agent
call *claim_done* Task completed successfully.

Statistics — Pass

Total Steps: 9 **Total Tokens:** 762,052

✓ No skill created — Direct API calls sufficient.

Traj B: DeepSeek-V3.2 *cat-facts-collector/e2*

System

You are an AI assistant with access to tools. Use save_skill to create reusable workflows and execute_skill to run them.

User

Collect information for 3 cat breeds (Persian, Siamese, Maine Coon) using breed_profile, breed_facts, and breed_encyclopedia APIs. Save results to a JSON file.

Step #1 Agent

call *list_directory* *workspace*

[workspace contents]

Step #2 Agent

call *save_skill* *process_cat_breed*

Skill saved successfully.

Step #3 Agent

call *execute_skill*

skill_name: *process_cat_breed* param: *Persian*

Skill executed.

▷ Warning: missing breed_facts, breed_encyclopedia fields

Step #4 Agent

call *execute_skill*

skill_name: *process_cat_breed* param: *Siamese*

Skill executed.

▷ Warning: missing breed_facts, breed_encyclopedia fields

Step #5 Agent

call *execute_skill*

skill_name: *process_cat_breed* param: *Maine Coon*

Skill executed.
▷ Warning: missing breed_facts, breed_encyclopedia fields

Step #6 Agent
call *write_file* *partial_results.json*

File written.

Step #7 Agent
call *file_write_json_chunk* *persian_data*

Chunk written.

Steps #8–14: More repair operations (write_json_chunk ×4, read_file, file_append)

Step #15 Agent
call *claim_done* Task completed.

Statistics — Pass

Total Steps: 15 **Total Tokens:** 1,507,830

× Skill created unnecessarily — 8 extra repair calls needed.

Traj C: Claude 4.5 Sonnet *cocktail-menu-generator/h1*

System
You are an AI assistant with access to tools. Use save_skill to create reusable workflows and execute_skill to run them.

User
Create a cocktail menu for 5 cocktails (Margarita, Mojito, Old Fashioned, Martini, Negroni) using 5 APIs each. Save to JSON.

Step #1 Agent
25 calls total. I'll create a reusable skill.
call *save_skill* *process_cocktail_complete*

Skill saved successfully.

Step #2 Agent
Now processing all 5 cocktails...
call *execute_skill*
skill_name: *process_cocktail_complete* param: *Margarita*

{search, details, ingredient, category, glass} → all fields complete

Steps #3–6: execute_skill for Mojito, Old Fashioned, Martini, Negroni → all success

Step #7 Agent
call *write_file* *cocktail_menu.json*

File written successfully.

Step #8 Agent
call *claim_done* Cocktail menu saved. Task completed.

Statistics — Pass

Total Steps: 8 **Total Tokens:** 213,222

✓ 1 save → 5 executes — Skill correct on first try.

Traj D: DeepSeek-V3.2 *cocktail-menu-generator/h1*

System

You are an AI assistant with access to tools. Use save_skill to create reusable workflows and execute_skill to run them.

User

Create a cocktail menu for 5 cocktails (Margarita, Mojito, Old Fashioned, Martini, Negroni) using 5 APIs each. Save to JSON.

Step #1 Agent
call *save_skill* *process_cocktail*

Skill save failed.
▷ SyntaxError: unexpected token '}' at line 8

Step #2 Agent
Fix syntax error and retry...
call *save_skill* *process_cocktail_v2*

Skill saved.

Step #3 Agent
call *execute_skill*
skill_name: *process_cocktail_v2* param: *Margarita*

Execution failed.
▷ SyntaxError: 'return' is invalid outside function (line 12)

Step #4 Agent
Skill failed. Fallback to direct API...
call *cocktail_search* *Margarita*

{"id": "11007", "name": "Margarita"...}

Step #5 Agent
call *cocktail_details* *11007*

{"ingredients": ["Tequila", "Lime"...]...}

Step #6 Agent
call *cocktail_by_ingredient* *Tequila*

{"cocktails": [...]...}

Step #7 Agent
call *cocktail_by_ingredient* *Lime*

{"cocktails": [...]...}

Step #8 Agent
call *save_skill* *process_cocktail_v3*

Skill saved.

Steps #9–18: More execute_skill calls + manual cleanup

Step #19 Agent
call *claim_done* Task completed after multiple retries.

Statistics — Fail

Total Steps: 19 **Total Tokens:** 1,141,166

× 3 saves + fallback — Skill quality poor, task failed.

We present representative trajectories from our experiments to illustrate the qualitative differences in how models approach skill creation and reuse. The above shows four trajectories: two from an easy task (`cat-facts-collector/e2`) and two from a hard task (`cocktail-menu-generator/h1`), comparing Claude 4.5 Sonnet and DeepSeek-V3.2.

Behavioral Divergence. A fundamental distinction emerges in how models decide *whether* to create skills. Claude exhibits efficiency-maximizing behavior: it autonomously evaluates whether the abstraction overhead is justified before committing to skill creation. In Trajectory A, Claude identifies that the easy task (9 API calls for 3 cat breeds) does not warrant skill abstraction and proceeds with direct calls, completing in 9 steps. In Trajectory C, facing a harder task (25 API calls for 5 cocktails), Claude creates a single skill that executes correctly 5 times with zero errors. In contrast, DeepSeek

follows the system prompt more literally, attempting skill creation regardless of task complexity. In Trajectory B, it creates `process_cat_breed` for the same easy task despite minimal reuse benefit, and in Trajectory D, it persists through three failed skill creation attempts before abandoning the approach entirely.

Skill Creation Failures. DeepSeek's skill creation attempts reveal systematic issues. In Trajectory B, the created skill `process_cat_breed` is incomplete—its output schema omits `breed_facts` and `breed_encyclopedia` fields, requiring 8 additional repair operations. In Trajectory D, DeepSeek attempts skill creation three times (`process_cocktail`, `process_cocktail_v2`, `process_cocktail_v3`), each failing with syntax errors such as "unexpected token" and "return is invalid outside function." These errors indicate that DeepSeek treats skill creation as template expansion rather than program synthesis.

Skill Execution Failures. Even when skills are successfully saved, execution failures reveal deeper issues. In Trajectory B, all three `execute_skill` calls produce incomplete results with warnings about missing fields. The skill's internal logic failed to properly chain the three required API calls. In Trajectory D, the `execute_skill` call fails immediately with a runtime error, forcing the agent to fall back to manual API calls and ultimately failing the task.

Implications. These findings suggest that effective tool composition requires not just the *ability* to create and execute skills, but the *judgment* to know when abstraction is beneficial. The 5.4× token savings achieved by Claude in the hard task (213K vs. 1.14M tokens) compared to DeepSeek demonstrates that understanding-driven skill use leads to both higher success rates and greater efficiency.

E Prompt Templates

This section presents the prompt templates used in our experiments, including the system prompt for skill-enabled modes and representative task prompts across different difficulty levels.

E.1 System Prompt for Skill Reuse

In skill mode, agents receive an augmented system prompt that introduces the skill abstraction mechanism. The prompt provides: (1) available skill tools (`save_skill` and `execute_skill`); (2) guidelines for when to create skills; (3) script authoring rules; and (4) a concrete example demonstrating the skill creation and execution workflow.

The key design principle is *minimal intervention*: rather than prescribing when agents should use skills, we provide the capability and let agents autonomously decide based on task structure. This enables fair comparison between skill-enabled and baseline modes, as the core task instructions remain identical.

System Prompt: Skill Reuse Mode

Skill Tools: You have access to skill cache tools to save and execute reusable scripts:

- `save_skill` — Save an executable script as a reusable skill
- `execute_skill` — Execute a saved skill with different arguments

When to Use: For repetitive operations (processing multiple items, files, etc.), create a skill to encapsulate the workflow, then execute it for all items. You can create skills based on tool schemas without calling the tool first—especially efficient when tools return large data.

Script Rules:

1. Use `call_tool()` for ALL tool calls: `call_tool('tool_name', arg1=val1, ...)`

2. `call_tool()` returns DIRECT result—use it directly without `.get("result")` wrapper
3. MUST set `result` variable—this is what gets returned from `execute_skill`
4. Modules available: `re`, `json`, `os` are pre-imported
5. No recursion: Cannot call skill tools within skills

Example:

```
save_skill({
  "skill_name": "analyze_project",
  "script_code": ...
})

execute_skill({
    "skill_name": "analyze_project"
    "args": {"path": "org/repo1"}}
```

Best Practices:

- **Token Efficiency:** Extract only fields needed for final output
- **Maximize ROI:** Create skill early, execute for ALL items (beneficial when $N \geq 3$)
- **Fallback:** If skill fails 2–3 times, process items directly

E.2 Task Prompt Examples

Task prompts describe the objective, required outputs, and available domain-specific tools. We present three representative examples from our scaled task suite, spanning easy (E), medium (M), and hard (H) difficulty levels. The scaling follows a systematic pattern: easy tasks involve $3 \times 3 = 9$ API calls, medium tasks involve $4 \times 4 = 16$ calls, and hard tasks involve $5 \times 5 = 25$ calls.

Each prompt specifies:

- **Objective**: The data collection or analysis goal
- **Output format**: JSON schema for structured results
- **Available tools**: Domain-specific APIs (prefixes removed for clarity)
- **Scale**: Number of subtasks and API calls per subtask

Note that skill-related tools (`save_skill`, `execute_skill`) are *not* mentioned in task prompts—they are injected via the system prompt only in skill-enabled modes. This ensures that baseline (Normal) mode and skill-enabled modes receive identical task instructions.

cat-facts-collector/e1 [Easy]

Prompt: Create encyclopedia entries for **3 cat breeds** (Persian, Siamese, Maine Coon) using 3 API endpoints per breed. For each breed, collect: (1) *Breed Profile* — basic info and characteristics; (2) *Country Relatives* — breeds from same country; (3) *Coat Family* — breeds with similar coat. Compile a summary with statistics across all breeds and save results to cat_encyclopedia.json.

Available tools:

- breed_profile(breed_name)
  Get breed info and characteristics
- breed_relatives(country)
  List breeds from same country
- breed_coat_family(coat_type)
  List breeds with similar coat

- write_file(path, content)
  Save JSON output
- claim_done(status)
  Signal task completion

Scale: 3 subtasks × 3 API calls = 9 total calls

cocktail-menu-generator/m1 [Medium]

Prompt: Create a cocktail menu for **4 classic cocktails** (Margarita, Mojito, Old Fashioned, Martini) using 4 API endpoints per cocktail. For each cocktail, collect: (1) *Search* — find cocktail by name; (2) *Details* — full recipe and instructions; (3) *By Ingredient* — list cocktails using main ingredient; (4) *By Category* — list cocktails in same category. Calculate complexity rating (Easy/Medium/Complex based on ingredient count) and estimated prep time. Save results to cocktail_menu.json.

Available tools:

- search(name)
  Search cocktail by name
- details(id)
  Get full recipe and instructions
- by_ingredient(ingredient)
  List cocktails with ingredient
- by_category(category)
  List cocktails in category
- write_file(path, content)
  Save JSON output
- claim_done(status)
  Signal task completion

Scale: 4 subtasks × 4 API calls = 16 total calls

gitlab-deep-analysis/h1 [Hard]

Prompt: Perform a comprehensive analysis of **5 GitLab repositories** (gitlab-runner, gitaly, gitlab-pages, gitlab-shell, cli). For each project, collect: (1) *Project Info* — stars, forks, description; (2) *Contributors* — top 5 by commit count; (3) *Recent Commits* — last 20 commits with authors; (4) *Branches* — all branches with protection status; (5) *Issues* — open count and recent titles. Calculate activity score (0–100) based on commits (40%), contributors (30%), issues (20%), branches (10%). Determine health status: *healthy* ($\geq$70), *moderate* (40–70), *inactive* ($<$40). Save results to gitlab_analysis_results.json.

Available tools:

- get_project_info(project_path)
  Get project details (stars, forks, description)
- get_contributors(project_path)
  Get contributor list
- get_commits(project_path, limit)
  Get commit history
- get_branches(project_path)
  Get branch information

- get_issues(project_path)
  Get issue list
- write_file(path, content)
  Save JSON output
- claim_done(status)
  Signal task completion

Scale: 5 subtasks × 5 API calls = 25 total calls

F LIMITATIONS

While SkillCraft offers a controlled and reproducible environment for evaluating tool composition and skill reuse, several limitations remain.

Tool ecosystem coverage. Our 21 task families span six application domains and rely on stable, publicly available REST APIs wrapped through MCP. This design favors reproducibility and rate-limit robustness, but it inevitably narrows the surface area compared to fully open agentic settings such as web browsing, GUI manipulation, or real software engineering workflows. Skills discovered on SkillCraft are well-suited to API-driven, structured pipelines, and our findings on cross-task and cross-model transfer should be extrapolated to less structured environments only with caution.

Single-language skill substrate. All Skills are expressed as Python scripts that invoke tools through a unified `call_tool()` interface. This choice keeps the verifier simple and decouples Skill semantics from agent dialect. However, agents that natively reason in other action representations (e.g., shell pipelines, declarative DSLs, or multi-modal tool grammars) may benefit from skill substrates we do not currently support. Extending the verifier to additional execution backends is a natural next step.

Verifier conservatism. The Coding Verifier rejects skills based on syntax errors, runtime failures, and silent-failure heuristics (e.g., outputs dominated by `None`/`Unknown`/`0`). These checks effectively block obviously broken skills, but they cannot certify semantic correctness for every input distribution. A skill may pass verification on the saving instance and still degrade on edge cases encountered during reuse, which is part of why hierarchical compositions amplify failures.

Static cross-model and cross-task transfer. Our cross-task and cross-model experiments deliberately use a static transfer protocol: skills are copied from a source workspace to a target workspace, without modification, and the agent cannot create new skills during the transfer phase. This isolates transferability as a property of the saved skills, but it also forecloses scenarios where mild on-the-fly repair could unlock more reuse. We view dynamic, repair-aware transfer as an important follow-up.

Closed-source model variability. Several of the strongest models we evaluate (Claude 4.5 Sonnet, GPT-5.2, Gemini 3 Pro) are closed-source and continually updated by their providers. The absolute numbers reported here are tied to the model snapshots and pricing in effect at the time of submission; the qualitative findings (e.g., the correlation between baseline capability and skill-mode efficiency gains) are expected to be more durable, but should be re-validated when these models change.

G BROADER IMPACT

SkillCraft studies whether language agents can autonomously discover, abstract, and reuse compositional skills built from atomic tools. We discuss the broader impact of releasing this benchmark and protocol.

Positive impact. By providing a standardized protocol with explicit storage, retrieval, enumeration, and execution primitives, SkillCraft makes skill composition a first-class, observable capability rather

than an incidental property of long trajectories. This supports more rigorous comparisons of agent designs and lowers the cost of reproducing tool-using behavior across labs. The substantial efficiency gains we measure (up to $\sim 80\%$ token reduction in Skill Mode) also point to a path toward cheaper, lower-latency, and more energy-efficient agent deployments—particularly important as tool-using agents are increasingly used at scale.

Risks of more capable tool-using agents. Better tool composition makes agents more capable, which has dual-use implications. A model that reliably abstracts and reuses tool chains can be applied to beneficial automation (data analysis, scientific computing, accessibility) but could also be misused to scale undesirable behavior such as spam generation, automated scraping in violation of service terms, or social engineering. SkillCraft itself does not target adversarial scenarios, but practitioners building on top of these capabilities should pair them with robust authentication, rate-limiting, and abuse-detection mechanisms when deploying agents against external services.

Tool and data provenance. All APIs used in SkillCraft are public REST endpoints that we identify, test, and wrap as standardized MCP tools. We respect the published terms of service and rate limits of each provider, and we encourage downstream users of the benchmark to do the same. The local file/data tasks use either custom datasets we author or publicly available data; no personally identifying information is collected or distributed by the benchmark.

Model and provider dependence. Our evaluation includes both open-weight and closed-source models. Reported costs and tokens depend on provider pricing and tokenizers at the time of submission. We document the API endpoints, configuration (temperature, token caps, turn caps), and partial-credit scoring rule so that subsequent users can re-run the benchmark on different models or providers without ambiguity.

Reproducibility and scientific rigor. We release the task suite, evaluation harness, and the four-primitive Skill MCP interface so that researchers can re-create the experimental conditions used in this work. Because the conclusions about cross-model and cross-task transfer rely on standardized reuse semantics, we encourage the community to extend the benchmark with additional task families and to report results using the same execution constraints (turn budgets, token caps, partial-credit threshold). This way, reported gains attributable to "skill reuse" remain interpretable across follow-up work.