SkillCraft: Can LLM Agents Learn to Use Tools Skillfully?

Shiqi Chen*
University of Oxford
Oxford, United Kingdom

Jingze Gai*
City University of Hong Kong
Hong Kong, China

Ruochen Zhou*
City University of Hong Kong
Hong Kong, China

Jinghan Zhang
Hong Kong University of Science and Technology
Hong Kong, China

Tongyao Zhu
National University of Singapore
Singapore, Singapore

Junlong Li
Hong Kong University of Science and Technology
Hong Kong, China

Kangrui Wang
Northwestern University
Evanston, IL, USA

Zihan Wang
Northwestern University
Evanston, IL, USA

Zhengyu Chen
Independent Researcher

Klara Kaleb
University of Oxford
Oxford, United Kingdom

Ning Miao
City University of Hong Kong
Hong Kong, China

Siyang Gao
City University of Hong Kong
Hong Kong, China

Cong Lu
Independent Researcher

Manling Li
Northwestern University
Evanston, IL, USA

Junxian He
Hong Kong University of Science and Technology
Hong Kong, China

Yee Whye Teh
University of Oxford
Oxford, United Kingdom

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 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 propose a lightweight evaluation protocol that enables agents to auto-compose atomic tools into executable Skills, cache and reuse them inside and across tasks, thereby improving efficiency while accumulating a persistent library of reusable skills. Evaluating state-of-the-art agents on SkillCraft, we observe substantial efficiency gains, with token usage reduced by up to 80% through skill saving and reuse. Moreover, success rate strongly correlates with tool composition ability at test time, underscoring compositional skill acquisition as a core capability.

Keywords: LLM Agents, Tool Composition, Skill Reuse, Agent Benchmark, Long-Horizon Tool Use, MCP

Code: https://github.com/shiqichen17/SkillCraft
Project page: https://skillcraft-website.github.io/page/
*Equal contribution.

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. [6, 14, 37] 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

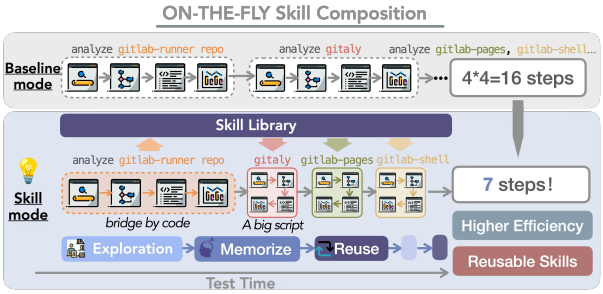


Figure 1. On-the-fly Skill Composition. Baseline mode requires $4 \times 4 = 16$ sequential tool calls, while Skill Mode abstracts recurring tool chains into reusable Skills, reducing the interaction to 7 steps.

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 [15, 32, 38] 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 [15], TheAgentCompany [32], and WebArena [38] 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 while solving tasks.

Using **SkillCraft**, we evaluate state-of-the-art models (e.g., Gemini 3 Pro, Claude 4.5 Sonnet, GPT-5.2) and find that *Skill Mode* substantially improves efficiency, reducing token usage by up to 80%. Moreover, efficiency gains from tool composition strongly correlate with task success, indicating that stronger models are better at discovering, reusing, and exploiting recurring tool-use patterns under the same composition mechanism.

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

Current tool-using benchmarks mainly test whether agents can solve a single task successfully with a fixed set of atomic tools (e.g., answering one real-time query with a search API). Such single-episode evaluations fail to reflect agents' tool composition ability. We therefore introduce **SkillCraft**, a long-horizon and compositional benchmark with repetitive structures that better reflects realistic settings and encourages the discovery and reuse of higher-level tool skills.

2.1 What kinds of tasks can evaluate skill composition?

We begin our exploration by asking: what kinds of tasks are required to faithfully evaluate an agent's ability to compose and reuse skills, rather than merely execute isolated tool calls? To evaluate skill composition, tasks must go beyond single-shot, low-branching problems. If a task can be solved efficiently with a few atomic tool calls, agents have little incentive to discover or reuse higher-level skills, and composition ability becomes indistinguishable. We therefore seek tasks that resemble realistic workflows: they are long-horizon, structurally repetitive, and sufficiently challenging that solving them instance-by-instance is inefficient, making reusable tool compositions genuinely beneficial.

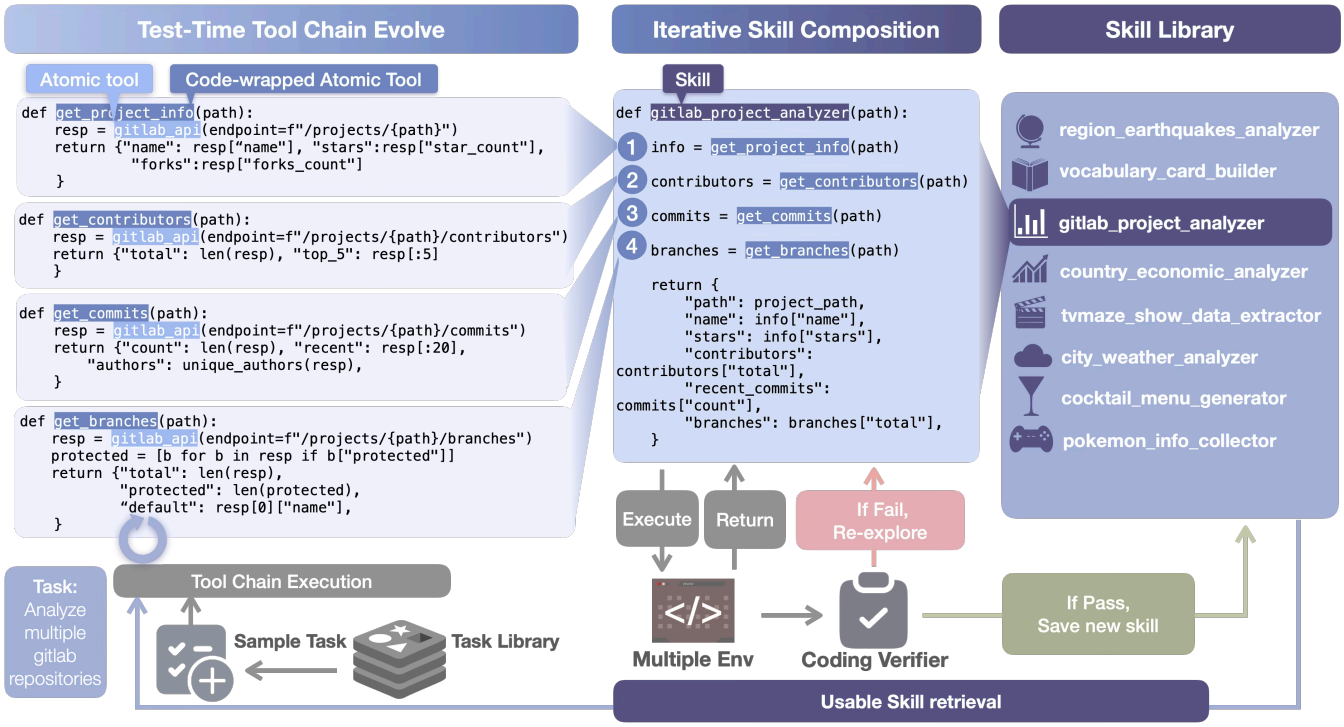


Figure 2. SkillCraft Protocol Pipeline Overview. The pipeline consists of three stages: **(1) Test-Time Tool-Chain Evolution:** The agent solves tasks from the Task Library by exploring and chaining atomic tools, forming executable tool sequences. **(2) Iterative Skill Composition:** Successful sequences are abstracted into candidate skills, executed and verified in a coding environment; failed executions trigger re-exploration, while validated skills are stored. **(3) Skill Library and Reuse:** A growing repository of verified, reusable skills that can be retrieved in later tasks to replace low-level tool exploration, enabling test-time skill accumulation and efficient composition.

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.

Importantly, these principles also mirror real-world tool-using scenarios, which are typically long-horizon and structurally repetitive, where similar sub-skills reoccur across tasks and the abstraction and reuse of higher-level skills are essential for efficient and robust problem solving.

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 [15], TheAgentCompany [32], WebArena [38] and M3ToolEval [29]. 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 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 4 shows coverage across domains and difficulty levels.

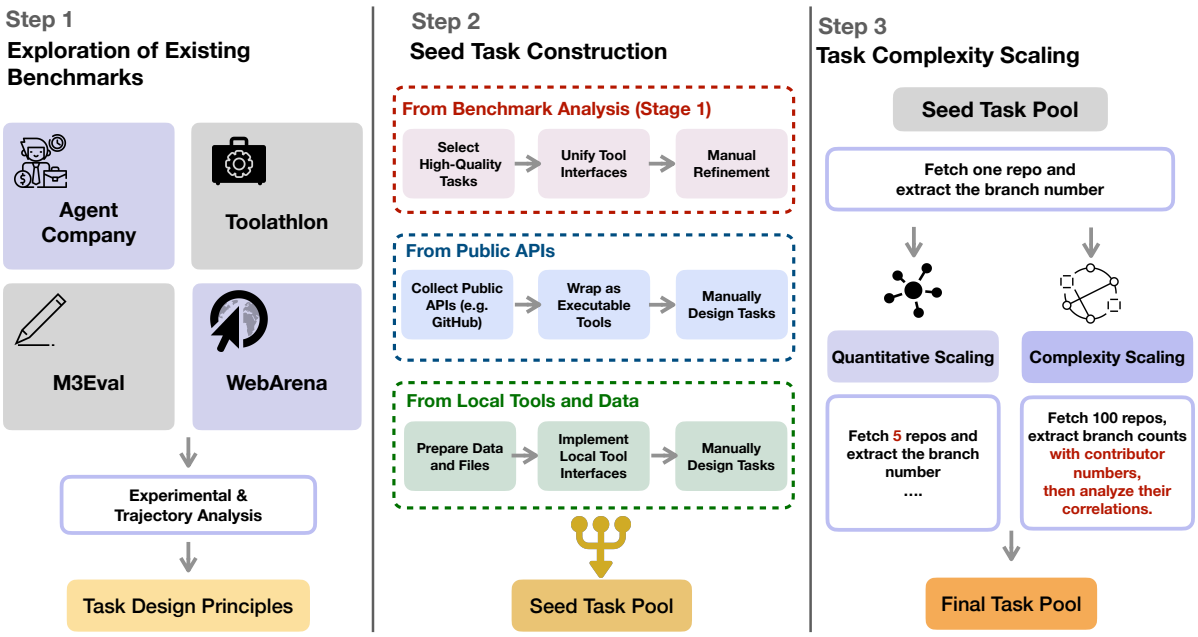


Figure 3. Three-stage task construction pipeline for SkillCraft. In **Stage 1**, we explore existing benchmarks through systematic experimentation to identify effective **task design principles**. In **Stage 2**, we construct seed tasks from three sources: (i) selected tasks from Stage 1 with **unified interfaces**, (ii) newly handcrafted **web API-based tasks**, and (iii) **local file and data processing tasks**. In **Stage 3**, we systematically scale the seed tasks via **quantitative scaling** (increasing subtask count) and **complexity scaling** (increasing tool calls per subtask), producing a task repository with **graduated difficulty levels**.

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**, we systematically scale seed tasks by increasing **entity number** and **subtask complexity**, producing **126 tasks** across 6 difficulty levels.

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) Scale subtask complexity (M: 3→4→5)	From Seed tasks Seed tasks × 6 levels	**126**

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 [1, 2, 8], 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

Difficulty	Tasks	Entity Num	Complexity	%
Easy	63	3	3	50.0%
Medium	42	4	4	33.3%
Hard	21	5	5	16.7%
Total	**126**	–	–	**100%**

Figure 4. Task distribution in SkillCraft. The chart shows 21 task families across 6 application domains. The table summarizes difficulty levels: **Entity Num** = number of target items (subtasks) per task; **Complexity** = tool calls required per entity. Each level is further sub-divided into variants (Easy: e1–e3, Medium: m1–m2, Hard: h1), yielding 6 sub-levels and 126 tasks in total.

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

In this section, we introduce the evaluation protocol for SkillCraft. To assess models' composition and skill curation abilities, we employ a pipeline that enables models to compose existing tools into novel higher-level ones and reuse them both inside the current task and also across tasks. This evaluation protocol enables two core capabilities in a quantifiable process: (1) Composition: Models could abstract multi-step tool chains into reusable code-based Skills. (2) Reuse: Models retrieve and reuse the discovered Skills at test time, enabling graceful execution and accumulating efficiency gains over repeated interactions.

3.1 Four Minimal MCP Primitives

To support skill reuse with minimal system assumptions, we expose a lightweight MCP interface that allows an agent to store and reuse executable code-based Skills. In practice, we maintain a *Skill Library* (a cache of *verified* Skills and their metadata) and expose four lightweight MCP primitives as the only way to interact with this library. This interface intentionally covers only the operational actions required by SkillMode: *storage*, *retrieval*, *enumeration*, and *execution*. Specifically, the Skill Library is accessed through `save_skill` (persist a workflow), `get_skill` (retrieve code and metadata), `list_skills` (discover available skills), and `execute_skill` (run a skill as a higher-level tool). Together, these primitives define the evaluation boundary: whether a model attempts reuse, whether reuse succeeds, and whether failures are handled can all be directly observed through these API calls. Figure 8 illustrates how these primitives fit into the overall protocol.

3.2 Coding Verifier

We introduce a Coding Verifier that applies three-stage validation before any Skill enters the library. The stages are:

(a) Syntax Validation: Before accepting `save_skill`, we parse the Skill code and reject syntactically invalid submissions, returning error line numbers and context snippets to block fundamentally broken code.

(b) Runtime Error Reporting: When `execute_skill` fails, we return structured debugging information (e.g. exception messages, tracebacks, and input parameters), which enables models to distinguish syntax issues from tool invocation problems or parameter mismatches.

(c) Post-execution Quality Detection: To filter out useless Skills, we detect silent failures by checking output quality. For example, if over 50% of output fields contain *Unknown, None, or 0*, we flag the Skill as low-quality and reject it.

3.3 SkillCraft Protocol Pipeline

To capture how models discover, store, and reuse skills across episodes, the protocol makes explicit, at each step, whether

a previously learned skill can replace a sequence of atomic tool calls. The protocol proceeds as follows:

(1) **Reuse Attempt.** For new task, agent queries existing Skills by `list_skills` and attempts to invoke a matching one by `execute_skill` with task-specific parameters.

(2) **Exploration.** If no suitable Skill exists or execution fails, the agent solves the task with atomic tools and records the successful tool sequence.

(3) **Composition.** The successful sequence is abstracted into a parameterized candidate Skill, consolidating recurring subroutines and passing intermediate results through code variables rather than natural language.

(4) **Verification and Saving.** The candidate Skill is executed in a controlled *Coding Env* via a unified `call_tool()` interface and validated by a *Coding Verifier*. Only skills that pass execution and verification are stored in the *Skill Library* via `save_skill` for reliable future reuse.

4 Evaluation

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

Models. We benchmark a representative set of state-of-the-art models, including Kimi-K2-Thinking [26], DeepSeek-V3.2-EXP [16], DeepSeek-R1 [12], GLM-4.7 [36], Gemini 3 Pro [11], MiniMax-M2.1 [20], Claude 4.5 Sonnet [3], and GPT-5.2 [22].

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 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 **Reuse**, the average number of times each saved Skill is invoked.

For efficiency, we report per-task averages of **Avg Tokens** (total input + output tokens), **Avg Cost** (USD), **Avg Turns** (LLM interaction rounds), and **Avg Tool Calls**. 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. Overall, **Skill Mode yields consistent and substantial gains in both success and efficiency across models**. For every model, Skill Mode sharply reduces average token usage and cost, and typically decreases the number of tool calls as well. However, the average number of conversation turns (highlighted in red in Table 2 Avg Turns) can increase for some models, as Skill Mode adds extra decision and verification steps when selecting and executing cached skills. But these additional turns are typically lightweight, so overall tokens and cost still drop. For example, GPT-5.2 improves success from 109/126 (87%) to 114/126 (90%), and also cuts average tokens from 1.23M to 0.26M (-79%) and average cost from \$1.77 to \$0.43 (-75%). It suggests that once skills are discovered and cached, long-horizon tool-chain planning can be solved both more effectively and more efficiently through repeated reuse.

Moreover, **the magnitude of efficiency gains correlates positively with model capability**. Cross-metric correlation analysis shown in Figure 5 reveals two key patterns: (1) **skill execution rate correlates with task success** (r=0.65), indicating that skill composition ability is tightly coupled with coding ability (skill execution success rate measures how reliably generated skills can be executed, with higher rates indicating better coding quality); (2) **Efficiency savings correlate with baseline success** (e.g., $r = 0.53$ for *Turns Saved* and *success rate*), confirming that stronger models benefit more from skill reuse. Concretely, *closed-source* models such as Claude 4.5 Sonnet 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 either suffer from lower success rates (<90% overall and <60% on the hard set; see Table 6), as observed for Kimi, DeepSeek, and GLM models, or exhibit limited tool-composition gains. For example, MiniMax-M2.1 shows only modest savings (-11%), 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 correct skills and execute them reliably.

Moreover, our case studies 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. This supports the view that tool composition ability is a core metric of intelligence. Detailed examples are provided in the Appendix D.3.

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 introduce **Hierarchical Mode**, which enables hierarchical, tree-structured skill composition by allowing skills to invoke other skills during execution. Under the standard **SkillCraft** protocol (Skill Mode), only single-level composition is supported: each skill is defined as a composition

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 Stats		Avg Tokens			Avg Cost ($)			Avg Turns			Avg Tool Calls			Success Rate (Overall)		Success Rate (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	-42%	0.21	0.13	-39%	6.7	8.3	+24%	16.8	11.9	-29%	55/126 (44%)	56/126 (44%)	8/21 (38%)	7/21 (33%)
DeepSeek-V3.2-EXP	71%	4.8×	1.04M	0.53M	-49%	0.21	0.10	-51%	18.8	15.4	-18%	19.2	14.9	-23%	76/126 (60%)	87/126 (69%)	9/21 (42%)	15/21 (71%)
DeepSeek-R1	62%	3.4×	0.58M	0.41M	-30%	0.24	0.18	-24%	9.0	9.9	+10%	13.4	11.7	-12%	89/126 (71%)	101/126 (80%)	11/21 (52%)	15/21 (71%)
GLM-4.7	91%	3.7×	0.78M	0.48M	-39%	0.20	0.12	-41%	13.5	13.0	-4%	16.9	13.3	-21%	91/126 (72%)	108/126 (86%)	12/21 (57%)	15/21 (71%)
MiniMax-M2.1	100%	3.2×	0.42M	0.38M	-11%	0.04	0.04	-8%	5.5	5.2	-6%	16.6	15.4	-7%	117/126 (93%)	119/126 (94%)	18/21 (86%)	20/21 (95%)
Closed-Source Models																		
GPT-5.2	84%	3.8×	1.23M	0.26M	-79%	1.77	0.43	-75%	20.6	9.9	-52%	19.4	8.9	-54%	109/126 (87%)	114/126 (90%)	16/21 (76%)	17/21 (80%)
Gemini 3 Pro	93%	3.9×	0.66M	0.30M	-54%	0.59	0.30	-49%	10.5	11.9	+13%	16.0	9.5	-41%	108/126 (86%)	116/126 (92%)	16/21 (76%)	17/21 (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%	119/126 (94%)	121/126 (96%)	20/21 (95%)	20/21 (95%)

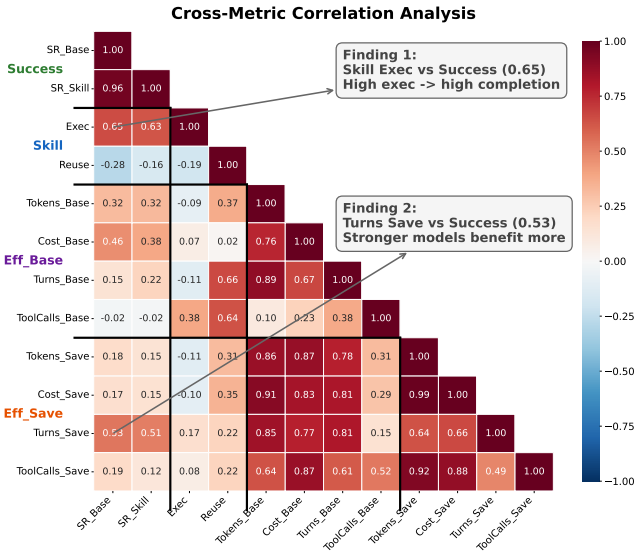


Figure 5. Cross-metric correlation heatmap. Metrics are grouped into four categories: Success, Skill, Eff_Base, and Eff_Save. Key findings: (1) Skill execution rate correlates with task success (r=0.65); (2) Stronger models achieve greater efficiency gains from skills (r=0.53).

of atomic tool calls and cannot invoke other skills. Hierarchical Mode lifts this restriction by enabling recursive skill invocation, permitting hierarchical composition up to a configurable nesting depth (*max_skills_nesting_depth=10* in our experiments). In theory, hierarchical composition enables reusable abstraction, yields multiplicative efficiency gains through nested skill reuse, and allows the agent to reason at higher levels rather than managing low-level details.

In practice, under **Hierarchical Mode**, when a skill is executed, the `call_tool` interface—responsible for dispatching executable actions during skill execution—can invoke not only atomic tools but also previously saved skills via `execute_pattern`. In contrast, under the standard **SkillCraft** protocol, `call_tool` is restricted to atomic tool invocations and cannot trigger other skills. This enables hierarchical/recursive skill invocation and yields a tree-structured execution graph, in which high-level skills orchestrate lower-level ones, as illustrated in Figure 6(a).

Table 3. Three-mode comparison across models. **Base**: No skill library. **Skill**: With skill library. **Hier**: Hierarchical mode with skill nesting. **N**: Number of successful tasks out of 126 total. **Ex**: Execution success rate (%). **Re**: Reuse factor (×). Δ: Relative change vs. Base (%).

Model	Mode	Success		Skill		Tokens		Cost ($)		Turns		Tools	
		N	%	Ex	Re	Val	Δ	Val	Δ	Val	Δ	Val	Δ
DeepSeek-V3.2	Base	76	60	–	–	1.04M	–	0.21	–	18.8	–	19.2	–
	Skill	87	69	71	4.8	0.53M	-49	0.10	-51	15.4	-18	14.9	-23
	Hier	92	73	75	3.0	0.68M	-36	0.10	-66	16.5	-13	15.4	-20
Claude-4.5	Base	119	94	–	–	1.36M	–	1.08	–	15.3	–	14.3	–
	Skill	121	96	81	3.4	0.40M	-71	0.28	-74	10.2	-33	9.2	-36
	Hier	121	96	99	3.8	0.63M	-54	0.44	-61	11.5	-26	10.5	-27
GPT-5.2	Base	109	87	–	–	1.23M	–	1.77	–	20.6	–	19.4	–
	Skill	114	90	84	3.8	0.26M	-79	0.43	-75	9.9	-52	8.9	-54
	Hier	100	79	95	3.6	0.60M	-51	0.81	-48	11.8	-42	10.8	-44

Practical Challenges. However, our experiments reveal that Hierarchical mode exhibits *lower overall success rates* compared to flat Skill mode. The primary reason is **error propagation through the skill hierarchy**. Figure 6

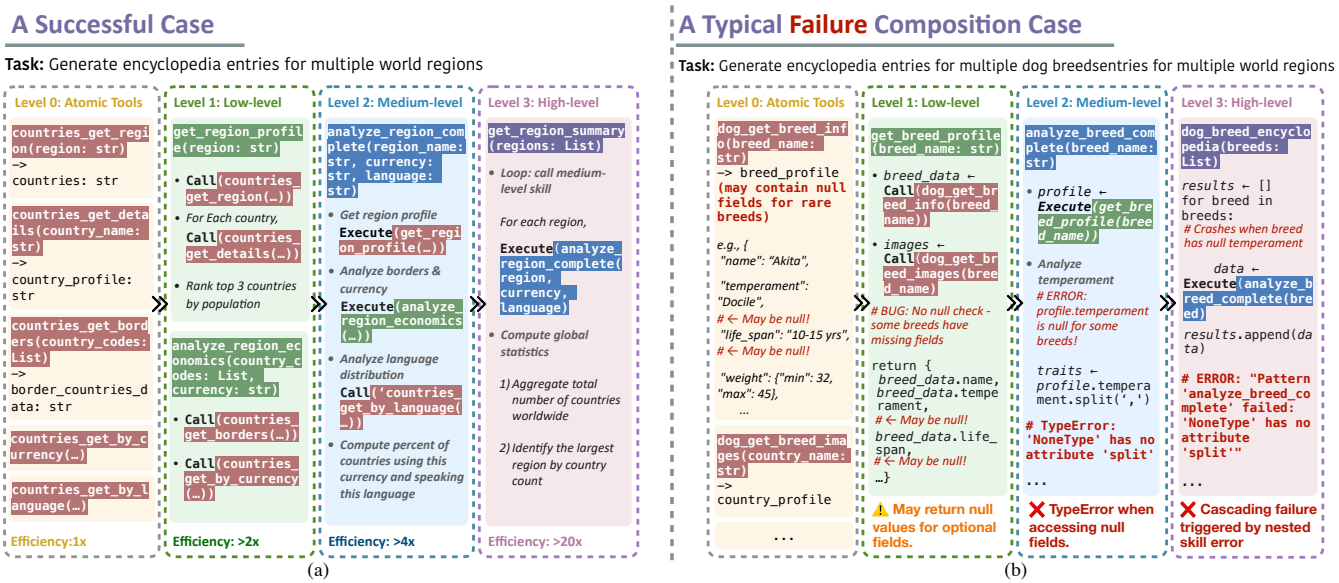


Figure 6. (a) Hierarchical skill composition in Hierarchical mode. A task organized as a depth-3 skill hierarchy, where atomic tools are encapsulated by low-level skills, composed into medium-level skills with additional processing, and orchestrated by a high-level skill. Efficiency gains compound across 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.

(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 (`dog_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.

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

A key property of a useful composition is its ability to generalize across problem difficulty. If a Skill captures reusable procedural structure rather than instance-specific solutions, it should transfer from simpler to more complex tasks (and vice versa) within the same task family. We therefore evaluate whether Skills learned at one difficulty level can be effectively reused at other difficulty levels.

We implement **Cross-Task Mode** using a two-phase static transfer approach. In Phase 1 (Skill Creation), an agent solves tasks at the *source* difficulty level in standard Skill mode, creating and caching Skills in its workspace. In Phase 2 (Skill Transfer), the runner: (1) copies the pre-computed Skill cache to the workspace for *target* difficulty tasks, (2) generates a `cross_task_skills_summary` and injects into the system prompt, providing the agent with a structured description of available Skills including signatures, parameters, and execution history, and (3) executes the agent on target tasks with full access to the inherited Skills.

We evaluate three transfer directions: **Easy→Hard** (Skills from e1–e3 tasks transferred to h1 tasks), **Hard→Easy** (Skills from hard tasks applied to easy tasks), and **Hard→Hard**

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. **Skill Exec**: Skill execution success rate. Efficiency metrics computed over tasks where both modes succeeded. **Claude** is Claude 4.5 Sonnet and **Gemini** is Gemini 3 Pro. Avg Tokens are in millions.

Model	Setting	Success Rate		Skill	Avg Tokens			Avg Cost ($)		
		Base	Skill	Exec	Base	Skill	Diff	Base	Skill	Diff
Claude	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	-45%
	H→H	20/21 (95%)	20/21 (95%)	98%	1.96	0.47	-76%	1.46	0.43	-71%
Gemini	E→H	16/21 (76%)	19/21 (90%)	99%	1.33	0.78	-41%	1.26	0.76	-39%
	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%

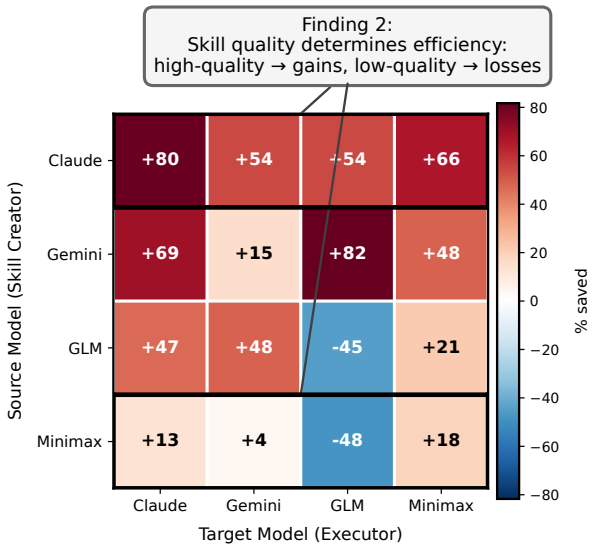
(Skills from one set of hard tasks applied to different hard tasks within the same family). This static transfer approach isolates the generalization capability of Skills by preventing any modification or accumulation during Phase 2 execution.

Empirical Results. Table 4 studies cross-task transfer across difficulty. For Claude 4.5 Sonnet, Easy→Hard tasks raise success from 95% to 100% and cut tokens from 1.92M to 1.56M, and Hard→Hard keeps success at 95% while dropping tokens from 1.96M to 0.47M. For Gemini 3 Pro, transfer also improves both success and efficiency. Easy→Hard increases success from 76% to 90% and reduces tokens from 1.33M to 0.78M. Hard→Hard increases success from 76% to 100% and reduces tokens from 1.30M to 0.75M. Notably, transferred Skills execute with consistently high *Exec* (typically 97–100%), suggesting that a pre-computed Skill cache learned at one level can be reused across other levels with strong cross-task generalization.

5.3 Cross-Model Skill Generalization

To investigate whether skills created by one model can benefit other models, we conduct a cross-model static reuse experiment on 8 hard-difficulty tasks. Four models (Claude, Gemini, GLM, and MiniMax) each create skills during their initial task execution, and these skills are then provided to all four models for execution in static-reuse mode, resulting in a total of 16 cross-model combinations (including 4 self-reuse baselines). In static-reuse mode, agents can invoke pre-loaded skills via `execute_skill` but cannot create new skills, ensuring that performance differences reflect skill quality and cross-model compatibility rather than on-the-fly adaptation. Figure 7 presents the results as two heatmaps: task success rate and token saving percentage.

(a) Success Rate



(b) Token Saving

Figure 7. Cross-model skill reuse heatmaps. Each cell (i, j) shows the result when model j executes skills created by model i. Bold borders highlight key findings. Token saving uses a diverging colormap: blue = increased cost, red = reduced cost.

Finding 1: High-quality skills achieve universal success. The first row of Figure 7a demonstrates that Claude-created skills achieve 100% success rate across all four target models, including when executed by Gemini, GLM, and MiniMax. This universally high success rate indicates that well-abstracted skills with clear parameter interfaces transfer effectively across different executor models, regardless of their architectural differences.

Finding 2: Skill quality determines efficiency gain or loss. Figure 7b reveals a stark contrast in computational efficiency based on skill creator quality. Claude-created skills (first row) yield consistently high token savings of 54–80% across all executors, demonstrating that high-quality skills provide universal efficiency benefits. In contrast, MiniMax-created skills (bottom row) result in token savings ranging from −48% to +18%, meaning poorly designed skills often *increase* rather than decrease computational cost. Notably, self-reuse (diagonal) does not always outperform cross-model reuse: Claude achieves 69% saving with Gemini's skills, substantially exceeding Gemini's own 15% self-reuse—indicating that executor capability can compensate for moderate skill quality, but cannot salvage fundamentally flawed skill designs.

Implications. These findings demonstrate that *skill creator quality matters more than executor capability*: investing in high-quality skill creation from capable models yields transferable efficiency benefits across the entire model ecosystem, while poorly designed skills can harm performance regardless of which model executes them. This suggests that multi-agent systems should prioritize skill libraries curated from high-capability models rather than allowing arbitrary skill contributions from all participants.

6 Conclusion

We introduced **SkillCraft**, a benchmark containing 126 tasks with recurring substructures, and Skill Mode, a protocol enabling agents to auto compose, cache, and reuse tool sequences. This framework allows us to measure whether agents can acquire compositional skills rather than merely execute isolated tool calls. Evaluating state-of-the-art models reveals three key findings. First, skill reuse reduces token usage by up to 80% while maintaining or improving success rates. Second, efficiency gains strongly correlate with model intelligence. Finally, skills generalize well across tasks and models (>95% execution), though hierarchical compositions are less reliable due to error accumulation across nested steps, highlighting compositional skill acquisition as a crucial capability for robust long-horizon tool use.

References

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

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

[3] Anthropic. 2025. *Claude Sonnet 4.5 System Card.* https://www.anthropic.com/claude-sonnet-4-5-system-card

[4] Anthropic. 2026. *Agent Skills.* https://agentskills.io/home Accessed: 2026-05-25.

[5] Chaithanya Bandi, Ben Hertzberg, Geobio Boo, Tejas Polakam, Jeff Da, Sami Hassaan, Manasi Sharma, Andrew Park, Ernesto Hernandez, Dan Rambado, et al. [n. d.]. MCP-Atlas: A Large-Scale Benchmark for Tool-Use Competency with Real MCP Servers. ([n. d.]).

[6] Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. 2024. WorkArena++: Towards Compositional Planning and Reasoning-based Common Knowledge Work Tasks. arXiv:2407.05291 [cs.AI] https://arxiv.org/abs/2407.05291

[7] Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, et al. 2025. ACEBench: Who Wins the Match Point in Tool Usage? *arXiv preprint arXiv:2501.12851* (2025).

[8] François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547* (2019).

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

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

[11] Google DeepMind. 2025. *Gemini 3 Pro: Model Card.* Technical Report. Google DeepMind. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf Published / Model Release: November 2025.

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

[13] Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. 2025. MCP-AgentBench: Evaluating Real-World Language Agent Performance with MCP-Mediated Tools. *arXiv preprint arXiv:2509.09734* (2025).

[14] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations.* https://openreview.net/forum?id=VTF8yNQM66

[15] Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, et al. 2025. The Tool Decathlon: Benchmarking Language Agents for Diverse, Realistic, and Long-Horizon Task Execution. *arXiv preprint arXiv:2510.25726* (2025).

[16] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556* (2025).

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

[18] Zhiwei Liu, Jielin Qiu, Shiyu Wang, Jianguo Zhang, Zuxin Liu, Roshan Ram, Haolin Chen, Weiran Yao, Shelby Heinecke, Silvio Savarese, et al. 2025. 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.* 373–402.

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

[20] MiniMax. 2025. *MiniMax M2.1: Significantly Enhanced Multi-Language Programming, Built for Real-World Complex Tasks.* https://www.minimax.io/news/minimax-m21

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

arXiv:2508.01780 (2025).

[22] OpenAI. 2025. *Update to GPT-5 System Card: GPT-5.2*. Technical Report. OpenAI. https://cdn.openai.com/pdf/3a4153c8-c748-4b71-8e31-aecbde944f8d/oai_5_2_system-card.pdf

[23] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. [n. d.]. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Forty-second International Conference on Machine Learning*.

[24] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive Tools. *arXiv preprint arXiv:2305.15334* (2023).

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

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

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

[28] Guanzhi Wang, Yuhuai Qin, Vineet Kosaraju, Dimitar Lee, Fei Zhang, Percy Liang, Jiajun Chen, Zhenzhong Chen, Ilija Ilievski, et al. 2023. Voyager: An Open-Ended Embodied Agent in Minecraft Powered by Large Language Models. *arXiv preprint arXiv:2305.16291* (2023).

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

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

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

[32] 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. 2024. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. arXiv:2412.14161 [cs.CL] https://arxiv.org/abs/2412.14161

[33] Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, et al. 2025. MCP-World: A Unified Benchmarking Testbed for API, GUI, and Hybrid Computer Use Agents. *arXiv preprint arXiv:2506.07672* (2025).

[34] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* (2024).

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

[36] Z.ai. 2025. *GLM-4.7*. https://z.ai/blog/glm-4.7 Blog post.

[37] 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. 2025. SWE-bench Goes Live! *arXiv preprint arXiv:2505.23419* (2025).

[38] 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. 2023. WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854* (2023).

[39] Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 2023. 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 [23] reduces tool use to structured function-parameter prediction, while τ-Bench and ACEBench emphasize multi-turn interaction and correct tool selection under reproducible environments [7, 34]. Gorilla and AgentBench broaden tool and domain coverage [17, 24], 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 [27], and MCP-based suites such as MCPWorld, MCP-RADAR, MCPEval, MCP-AgentBench, LiveMCPBench, and MCPAtlas standardize tool integration across servers [5, 10, 13, 18, 21, 33], 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 [14, 31, 32, 38], while GAIA, ARE, and BrowseComp focus on everyday tasks and web-based information seeking [9, 19, 30]. Tool Decathlon (Toolathlon) further consolidates real tools, fuzzy instructions, execution verification, and cross-application workflows [15].

On the pipeline side, most tool-using agents follow the "reasoning–acting–observing" loop introduced by ReAct [35], where planning and state tracking are repeated at every tool call. CodeAct [29] 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 [28] and Ghost in the Minecraft [39] show that agents can grow a code skill library through exploration, yet the resulting skills are tied to game rules and state spaces. CREATOR [25] abstracts reusable components from patterns but provides limited evidence of robust cross-task generalization in realistic tool ecosystems. Anthropic Skills [4] 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.

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 (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

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.

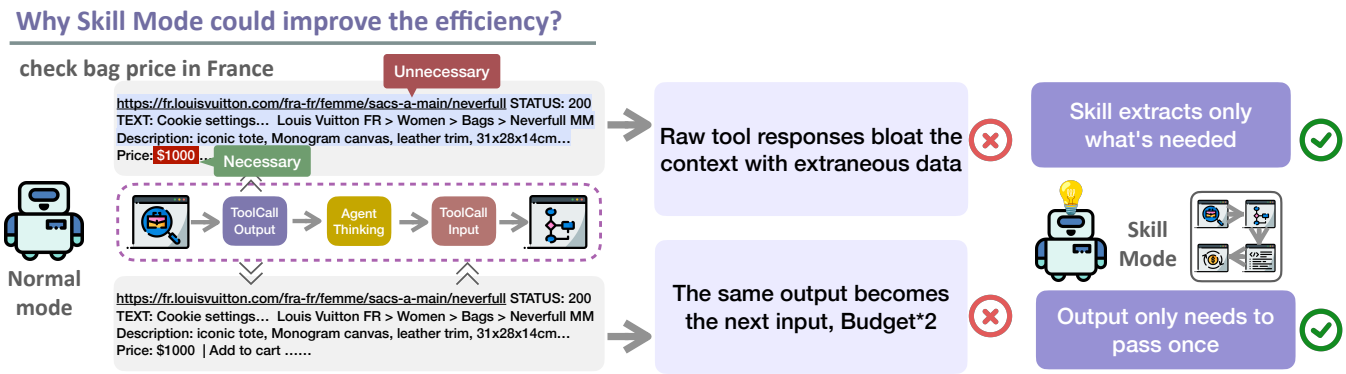


Figure 9. Skill Mode improves efficiency through two mechanisms. First, it reduces argument passing overhead by enabling direct tool chaining (Tool A → Tool B → Tool C) rather than shuttling intermediate outputs through the agent (Tool A → Agent → Tool B → Agent → Tool C). Second, it amortizes planning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

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

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 4. **Tools**: number of distinct API-wrapping functions per task. Each task family includes 3 difficulty-scaled variants (Easy: 3 subtasks, Medium: 4 subtasks, Hard: 5 subtasks), totaling 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	Developer	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

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 Results by task difficulty

Table 6 presents a detailed breakdown of our experimental results across three difficulty levels: Easy (tasks e1–e3), Medium (tasks m1–m2), and Hard (task h1). We identify several noteworthy patterns that provide deeper insights into the behavior and benefits of skill reuse.

Skill Reuse Frequency Increases with Task Complexity. Across all models, the average skill reuse count shows a consistent upward trend with task difficulty. For Easy tasks, skills are invoked 2.3–3.0× on average, while Hard tasks see 3.0–4.9× reuse. This pattern reflects the compositional nature of our benchmark: harder tasks require more repeated API compositions, which naturally leads to more opportunities for skill reuse. Notably, GLM-4.7 achieves the highest reuse rate (4.9×) on Hard tasks, demonstrating effective skill generalization across complex scenarios.

Efficiency Gains are More Pronounced on Harder Tasks. Token savings exhibit a clear correlation with task difficulty. For frontier models like Claude 4.5 Sonnet and GPT-5.2, token reduction on Hard tasks reaches 77–78%, compared to 62–79% on Easy tasks. Similarly, tool call reduction is most dramatic on Hard tasks: Gemini 3 Pro achieves a 70% reduction on Hard versus 29% on Easy, while GPT-5.2 shows 68% versus 38%. This suggests that skill reuse provides greater benefits when tasks involve more complex, multi-step API orchestrations—precisely the scenarios where manual tool composition becomes most costly.

Success Rate Improvements Favor Challenging Tasks. For models with moderate baseline performance, skill reuse disproportionately improves success rates on Hard tasks. DeepSeek-V3.2-EXP shows a remarkable +29 percentage

Table 6. Results breakdown by difficulty level (Easy: e1–e3, Medium: m1–m2, Hard: h1). **Success Rate**: task completion rate (score ≥ 90) for Baseline and Skill modes. **Skill Stats**: Exec = skill execution success rate; Reuse = average times each skill is invoked. **Efficiency metrics**: per-task averages computed over tasks where *both* modes succeeded. **Diff**: percentage change; negative = improvement, positive = degradation.

Model	Diff.	Success Rate		Skill Stats		Avg Tokens			Avg Cost ($)			Avg Turns			Avg Tool Calls		
		Base	Skill	Exec	Reuse	Base	Skill	Diff	Base	Skill	Diff	Base	Skill	Diff	Base	Skill	Diff
Kimi-K2-Thinking	Easy	30/63 (48%)	32/63 (51%)	81%	2.6×	427K	293K	-31%	0.17	0.12	-29%	9.0	13.7	+53%	12.2	10.6	-13%
	Medium	17/42 (40%)	17/42 (40%)	76%	2.9×	576K	335K	-42%	0.23	0.14	-39%	8.1	15.1	+87%	19.5	12.5	-36%
	Hard	8/21 (38%)	7/21 (33%)	66%	4.8×	622K	285K	-54%	0.25	0.13	-50%	10.4	16.4	+58%	27.8	18.4	-34%
DeepSeek-V3.2-EXP	Easy	42/63 (66%)	47/63 (74%)	94%	2.5×	943K	512K	-46%	0.27	0.22	-18%	28.2	25.7	-9%	15.4	13.2	-14%
	Medium	25/42 (59%)	25/42 (59%)	89%	3.2×	1.34M	556K	-59%	0.22	0.08	-63%	36.2	30.4	-16%	23.4	17.2	-26%
	Hard	9/21 (42%)	15/21 (71%)	88%	4.4×	844K	547K	-35%	0.22	0.07	-69%	21.4	33.1	+55%	28.7	17.6	-39%
DeepSeek-R1	Easy	50/63 (79%)	55/63 (87%)	68%	2.9×	498K	470K	-6%	0.21	0.20	-3%	14.0	16.8	+20%	11.1	11.0	-1%
	Medium	28/42 (66%)	31/42 (74%)	77%	3.5×	631K	241K	-62%	0.26	0.13	-50%	13.6	14.9	+10%	15.8	10.7	-32%
	Hard	11/21 (52%)	15/21 (71%)	68%	4.2×	855K	421K	-51%	0.34	0.17	-49%	17.1	16.9	-1%	16.7	16.8	+1%
GLM-4.7	Easy	52/63 (82%)	57/63 (90%)	90%	2.9×	661K	428K	-35%	0.18	0.12	-33%	12.2	11.4	-6%	13.6	11.8	-13%
	Medium	27/42 (64%)	36/42 (85%)	89%	4.0×	874K	514K	-41%	0.22	0.10	-54%	13.7	15.3	+11%	19.0	15.1	-21%
	Hard	12/21 (57%)	15/21 (71%)	94%	4.9×	1.17M	648K	-45%	0.28	0.16	-43%	19.6	15.1	-23%	28.2	16.5	-41%
Gemini 3 Pro	Easy	55/63 (87%)	59/63 (93%)	95%	2.3×	534K	300K	-44%	0.46	0.30	-35%	15.4	19.6	+27%	12.8	9.2	-29%
	Medium	37/42 (88%)	40/42 (95%)	92%	2.7×	730K	323K	-56%	0.68	0.31	-55%	16.6	18.4	+11%	19.5	10.6	-46%
	Hard	16/21 (76%)	17/21 (81%)	96%	3.3×	970K	227K	-77%	0.90	0.27	-70%	22.2	20.3	-8%	28.9	8.7	-70%
MiniMax-M2.1	Easy	59/63 (94%)	58/63 (96%)	100%	3.0×	379K	363K	-4%	0.04	0.03	-13%	7.7	7.4	-4%	12.1	11.4	-5%
	Medium	40/42 (95%)	41/42 (98%)	84%	3.6×	468K	380K	-19%	0.05	0.05	-8%	7.9	7.1	-11%	18.6	16.8	-10%
	Hard	18/21 (86%)	20/21 (95%)	100%	3.0×	479K	409K	-15%	0.06	0.05	-4%	8.2	7.2	-12%	26.8	24.8	-8%
Claude 4.5 Sonnet	Easy	60/63 (95%)	60/63 (95%)	99%	3.0×	1.06M	399K	-62%	0.81	0.25	-69%	19.2	17.4	-9%	11.4	9.0	-21%
	Medium	39/42 (92%)	41/42 (98%)	100%	3.7×	1.54M	369K	-76%	1.32	0.27	-80%	22.0	17.7	-19%	15.7	9.0	-43%
	Hard	20/21 (95%)	20/21 (95%)	98%	4.7×	1.96M	440K	-77%	1.46	0.40	-72%	25.5	19.9	-22%	20.3	10.2	-50%
GPT-5.2	Easy	59/63 (94%)	60/63 (95%)	91%	3.0×	939K	196K	-79%	1.38	0.30	-78%	22.3	15.1	-32%	12.5	7.7	-38%
	Medium	34/42 (81%)	37/42 (88%)	95%	4.2×	1.44M	314K	-78%	2.10	0.48	-77%	26.9	17.0	-37%	21.0	8.9	-58%
	Hard	16/21 (76%)	17/21 (80%)	90%	4.3×	1.86M	405K	-78%	2.72	0.61	-77%	31.4	20.5	-35%	41.3	13.4	-68%

point improvement on Hard tasks (from 42% to 71%) compared to only +8 points on Easy tasks. Similarly, DeepSeek-R1 improves by +19 points on Hard versus +8 points on Easy. This indicates that skills learned from easier variants effectively transfer to help models overcome challenges they would otherwise fail, validating the cross-difficulty generalization capability of our skill framework.

High-Capacity Models Benefit from Efficiency, Not Accuracy. Frontier models (Claude, GPT-5.2) already achieve ≥94% success rates on Easy tasks in baseline mode, leaving little room for accuracy improvement. However, they show the largest efficiency gains: Claude achieves 72% average token reduction, and GPT-5.2 achieves 78%. In contrast, MiniMax-M2.1, which exhibits highly efficient baseline behavior (only 379K–479K tokens per task), shows modest 4–19% token savings. This suggests that skill reuse is most valuable for models whose baseline execution involves verbose, sequential API interactions.

Skill Execution Remains Robust Across Difficulties. Skill execution success rates remain consistently high (66–100%) across all difficulty levels for most models, indicating that skills created during easier tasks transfer reliably to harder contexts. The lowest execution rates appear in Kimi-K2-Thinking (66% on Hard) and DeepSeek-R1 (68% on Easy/Hard), both of which employ extended reasoning that may occasionally conflict with deterministic skill execution patterns.

D.2 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

Table 7. 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	-36%
	Direct Exec	46/48	96%	68%	3.1×	0.16M	-90%	0.17	-89%	5.8	-64%	4.8	-68%
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	-78%	3.5	-81%

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 7 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 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.3 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 (15 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 5.3× 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* (≥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