SKILLFLOW: Benchmarking Lifelong Skill Discovery and Evolution for Autonomous Agents

Anonymous ACL submission

Abstract

As the capability frontier of autonomous agents continues to expand, they are increasingly able to complete specialized tasks through plug-and-play external skills. Yet current benchmarks mostly test whether models can use provided skills, leaving open whether they can discover skills from experience, repair them after failure, and maintain a coherent library over time. We introduce **SKILLFLOW**, a benchmark of 166 tasks across 20 families in which task construction within each family follows a Domain-Agnostic Execution Flow (DAEF) that defines an agent workflow framework, allowing these tasks to share a consistent workflow. Agents are evaluated under an Agentic Lifelong Learning protocol in which they begin without skills, solve tasks sequentially within each family, externalize lessons through trajectory- and rubric-driven skill patches, and carry the updated library forward. Experiments reveal a substantial capability gap. For Claude Opus 4.6, lifelong skill evolution improves task success from 62.65% to 71.08% (+8.43 points). However, high skill usage does not necessarily imply high utility: Kimi K2.5 gains only +0.60 points despite 66.87% skill usage, while Qwen-Coder-Next reaches only a 44.58% task completion rate and still regresses relative to the vanilla setting. SKILLFLOW contributes a structured testbed for this direction and an in-depth empirical analysis of skill discovery, patching, transfer, and their failure modes under lifelong evaluation.

1 Introduction

Frontier Large Language Models(LLMs) systems are increasingly deployed as autonomous command-line agents that solve realistic multi-step tasks in terminal environments. Claude Code (Anthropic, 2025), Gemini CLI (Google, 2025), Codex CLI (OpenAI, 2025), and Qwen-Coder (Qwen Team, 2025) are prominent examples. Agent skills have recently emerged as external packages of procedural knowledge that augment model capabilities for specialized tasks. These skills can encode usage scenarios, core guidance, and reusable code or documentation assets. As a result, many vendors are beginning to release agent versions with native skill support.

SkillsBench (Li et al., 2026) provides an early demonstration that skills can substantially assist models in solving tasks, while also suggesting that current models remain limited in generating skills autonomously. However, it does not examine whether an LLM can discover and summarize reusable skills from its own task-solving process and apply them to future tasks of the same type. SkillWeaver (Zheng et al., 2025), SkillRL (Xia et al., 2026), MemSkill (Zhang et al., 2026), and related systems (Zhou et al., 2026) also show that experience-derived skills can improve downstream performance. This leads to a central question: **can an autonomous agent extract reusable skills from its own experience, repair them after failures, and maintain an evolving skill library across a sequence of tasks?** To answer this question, we introduce **SKILLFLOW**, a benchmark of 166 runnable tasks across 20 task families for measuring lifelong skill discovery and evolution.

To construct the benchmark, we first analyze existing agent loops and find that mature agents (Huang et al., 2026b; Zeng et al., 2026; Fang et al., 2025) typically rely on stable AI workflows, which allow them to solve tasks of the same type in a consistent manner. Based on this observation, we define the Domain-Agnostic Execution Flow (DAEF), a workflow framework that preserves operational topology while abstracting away domain-specific entities and serves as a scaffold for benchmark construction. We use this structure to support controlled progression within each task family. Concretely, we select realistic tasks from GDPval (Patwardhan et al., 2025) and SkillsBench (Li et al., 2026) and extract a DAEF from each seed task. We then instantiate each DAEF through a dual-agent iterative pipeline using the

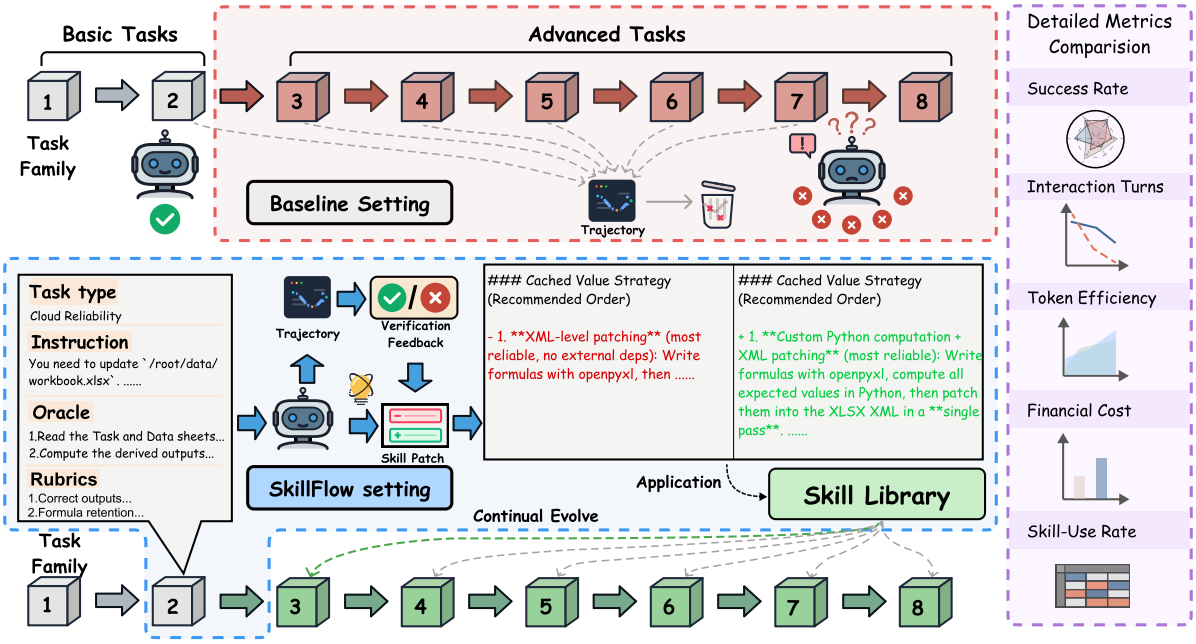

Figure 1: Conceptual Overview of SKILLFLOW. The figure contrasts conventional static-skill evaluation with our lifelong setting, in which agents externalize experience into reusable skill artifacts, revise them through patches, and transfer them across tasks that share a common DAEF.

original task as a reference instance, where one agent constructs task assets and the other reviews the generated tasks and provides timely feedback for revision. This process yields 8 to 9 tasks in the Harbor format (Harbor Framework Team, 2026) for each DAEF, which together form a task family. After human verification, the final benchmark contains 20 task families and 166 tasks. Evaluation then follows an **Agentic Lifelong Learning** protocol in which an agent begins without skills, solves tasks sequentially within each family, summarizes lessons from trajectories and rubric feedback, and updates the library through explicit *skill patches* after each task.

Experiments reveal a substantial **capability gap**. On Claude Opus 4.6, lifelong skill evolution improves task success from 62.65% to 71.08% (+8.43 points). Yet high usage does not imply high utility: Kimi K2.5 gains only +0.60 points despite a 66.87% skill usage rate, while Qwen-Coder-Next reaches only a 44.58% task completion rate and still regresses relative to the vanilla setting. We find that most current models fail to achieve stable self-evolution through iterative skill updates. Stronger models tend to consolidate prior experience into stable workflows and continue refining them over time, thereby realizing sustained capability gains. Weaker models struggle to integrate multiple experiences into a coherent procedure, which leads to cognitive overload in subsequent tasks.

Contributions. Our contributions are listed:

- We introduce **SKILLFLOW**, a benchmark of 166 runnable tasks across 20 task families for evaluating lifelong skill discovery and evolution in autonomous agents.
- We design **task families with a shared agent workflow framework** based on DAEF, enabling systematic evaluation of cross-task skill transfer learning.
- We specify an **Agentic Lifelong Learning** protocol in which agents begin without skills, evolve an external skill library through trajectory- and rubric-driven *skill patches*, and progressively learn and are evaluated under a fixed task order within each family.
- We identify and closely analyze **common failure patterns** in skill evolution, including fragmented skill growth, reinforcement of erroneous logic, and the gap between writing skills and reliably repairing them, revealing current models' limitations in producing compact, transferable procedural skills.

2 SKILLFLOW

2.1 Benchmark Overview

SKILLFLOW contains 20 workflow families and 166 tasks spanning five broad domains: *Finance &*

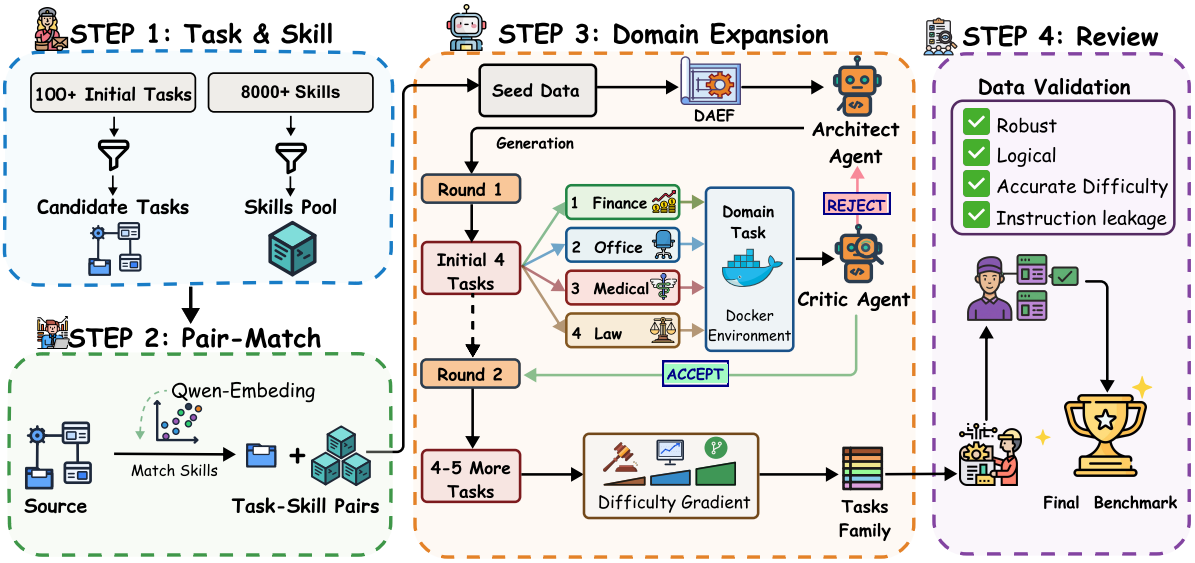

Figure 2: Task-construction pipeline of SKILLFLOW. Seed tasks and reference skills are collected and matched, a dual-agent loop expands each seed under a fixed DAEF into a task family with a difficulty gradient, and human review verifies robustness, logical soundness, and leakage risk before inclusion.

Economics, *Operations & Supply Chain*, *Healthcare & Life Sciences*, *Governance & Strategy*, and *Data & Document Intelligence*. Each benchmark instance is defined by a family-local task sequence, a fixed within-family difficulty order, and an associated verifier style. Family sizes range from 8 to 9 tasks. Overall, the benchmark encompasses a range of realistic workplace workflows, including spreadsheet-centric planning, OCR and PDF extraction, office-document editing, compliance analysis, and quantitative reasoning.

2.2 Domain-Agnostic Execution Flow

The central design principle of SKILLFLOW is a **Domain-Agnostic Execution Flow (DAEF)**: a workflow skeleton shared by a family of tasks after removing domain-specific entities, file names, and business semantics. Formally, we represent a task instance as a domain-grounded workflow graph

$$\mathcal{T} = (V, E, \lambda, \gamma), \tag{1}$$

where V is a set of executable sub-goals or operations, $E \subseteq V \times V$ is a set of precedence or dependency edges, $\lambda(v)$ assigns each node a domain-agnostic operation type, and $\gamma(v)$ provides task-specific grounding, such as concrete files, entities, fields, or business objects.

A DAEF is the abstract workflow graph obtained by removing task-specific grounding while preserving operation types and dependency structure:

$$\mathcal{F} = \phi(\mathcal{T}) = (V_F, E_F, \lambda_F). \tag{2}$$

Operationalizing DAEF in construction. We operationalize DAEF through a two-stage human annotation and standardization protocol that makes extraction explicit, standardized, and verifiable before task generalization.

Stage 1: meta-step extraction. Each seed task is independently annotated with a sequence of meta-step nodes containing 5–8 executable operations, dependency edges, and a short textual rationale. Node types are drawn from a controlled single-word vocabulary such as *read*, *retrieve*, *compute*, *detect*, and *output*; the full inventory is listed in Appendix A.6.

Stage 2: workflow construction. The extracted workflow representation is normalized by collapsing task-specific grounding, standardizing node labels, and retaining only the dependency structure and operation inventory stable across tasks. The resulting canonical DAEF also specifies the allowed categories of variation to control the difficulty gradient of subsequently instantiated tasks. We provide the detailed annotation rules and expert agreement screening procedure in Appendix A.6. A canonical DAEF is retained only if it remains internally coherent and supports controlled generalization to new tasks without introducing a new workflow; a detailed table of allowed variation types appears in Appendix A.7.

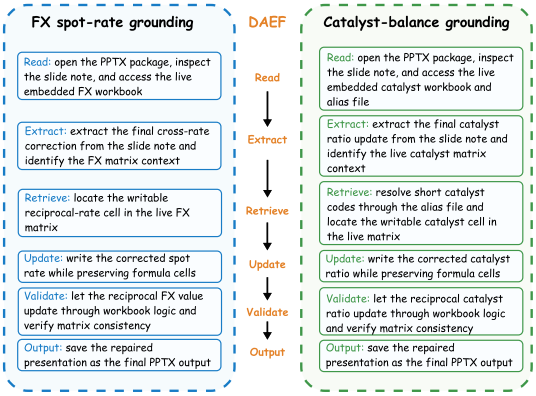

Figure 3: DAEF correspondence across domains. Distinct tasks can instantiate the same abstract workflow, enabling cross-domain skill transfer.

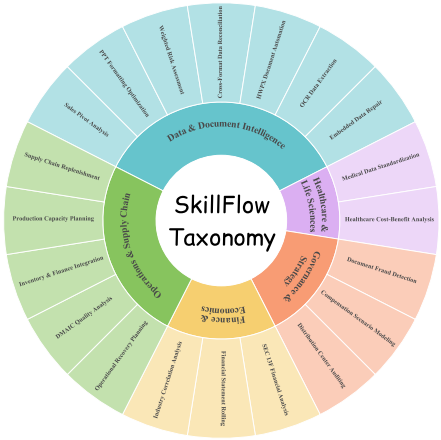

Figure 4: Relationship Between Benchmark Categories and DAEFs. The inner ring shows the five top-level benchmark categories, and the outer ring shows the corresponding DAEF names under each category.

2.3 Task Construction Protocol

Figure 2 summarizes the process used to build SKILLFLOW. We organize it into **four steps**.

Step 1: seed task collection and skill curation. We identify seed tasks from two benchmark suites, retaining high-quality examples that reflect realistic workplace scenarios without requiring external network services. We exclude tasks that are overly narrow, simple, or difficult to verify reliably. This yields 64 seed tasks: 18 from SkillsBench (Li et al., 2026) and 46 from GDPval (Patwardhan et al., 2025). Simultaneously, we gather over 8,000 open-source skills from public repositories (ComposioHQ, 2026; VoltAgent, 2026; Anthropic, 2026; SkillsMP, 2026), filtering them for safety, formatting, and domain relevance to obtain 2,318 skills for task construction.

Step 2: task–skill pair matching. To provide specialized support for subsequent task construction, we match each seed task with reference skills. Using Qwen3-embedding-4B (Zhang et al., 2025), we calculate semantic similarity between task and skill descriptions and retrieve 5–10 candidate skills for each seed, yielding 64 matched task–skill pairs.

Step 3: iterative task-family generalization. We then construct the family-level reference package used for domain expansion. For each selected seed, human annotators convert the task into a DAEF by extracting meta-steps and defining the workflow according to the previously described protocol. After deduplication and a generalization-difficulty check, we retain 30 DAEFs. The resulting seed-task reference package—including the original assets, matched skills, files, environment, and stable DAEF—enters an automated construction loop. We discard seeds at this stage if the resulting family fails to achieve execution stability or DAEF consistency within five revision iterations.

(a) Architect Agent generation. The first agent, built within Cursor (Anysphere, 2024) using GPT-5.3-Codex (OpenAI, 2026), serves as the Architect Agent for the new task family. Conditioned on the target DAEF, it proposes new task types and uses the seed task as a reference to construct task goals, create task assets, and define verification schemes for these tasks. With support from the matched skills, it then builds the corresponding Docker task environments and develops solver and verifier assets, producing four initial tasks in the Harbor format.

(b) Critic Agent review. A second agent, employing Claude Opus 4.6, serves as the Critic Agent and evaluates the four initial tasks. It verifies and evaluates these tasks in real Docker environments and provides detailed feedback after jointly assessing workflow consistency, the presence of a difficulty gradient, task solvability, verifier correctness, and environment reliability.

(c) Revision loop and expansion phase. The Architect Agent and the Critic Agent interact for multiple rounds until the Critic Agent approves the family. If the Critic Agent rejects the family more than five times, construction of that task family is abandoned. After passing inspection, the process enters a second round of task construction, which follows the same construction–verification loop as the first round and generates 4–5 new tasks. This

two-round process produces a task sequence of 8–9 tasks for each family and ultimately yields 20 task families.

Step 4: human review and family revision. Human reviewers inspect each candidate family along four dimensions: instruction leakage, logical soundness of the task, correctness of the final task environment, and reasonableness of the difficulty gradient. Families with issues in any dimension are returned for manual revision. In practice, thanks to the strict construction procedure in the preceding steps, none of the 20 task families exhibits fatal environment errors, and all 20 families are retained after manual revision.

After the final review and revision process, this pipeline yields 20 workflow families and 166 tasks.

2.4 Agentic Lifelong Learning Protocol

We formalize the lifelong learning protocol as a sequential process over a family of tasks. Let $\mathcal{F} = \{T_1, T_2, \ldots, T_n\}$ denote an ordered task family, where tasks are sorted by within-family difficulty. The agent maintains an updatable skill library $\mathcal{S}_t$ at each step t. The protocol proceeds as follows:

First-task execution and feedback. For the first task T_1, the agent does not use any skill and completes the task under its native agent harness, producing an execution trace τ_1. After task completion, the model receives a verifier-derived rubric r_1, which is a normalized textual description of missing or incorrect content.

Skill patch generation. After each task, the model learns from the resulting trajectory and verifier feedback and generates a skill patch conditioned on a fixed skill-patch prompt template g. This patch-generation step relies on the model's native capability rather than the surrounding agent harness. For the first task, the patch bootstraps the skill library from scratch:

$$\begin{aligned} \Delta_1 &= \text{Model}_g(\emptyset, \tau_1, r_1), \\ \mathcal{S}_1 &= \text{Apply}(\Delta_1, \emptyset). \end{aligned} \tag{3}$$

For all subsequent tasks $t > 1$, the agent executes T_t with the current library $\mathcal{S}_{t-1}$, obtains an execution trace τ_t and verifier-derived rubric r_t, and applies the same prompted generation process incrementally:

$$\begin{aligned} \Delta_t &= \text{Model}_g(\mathcal{S}_{t-1}, \tau_t, r_t), \\ \mathcal{S}_t &= \text{Apply}(\Delta_t, \mathcal{S}_{t-1}). \end{aligned} \tag{4}$$

The patch Δ_t may add, revise, or delete skills. This incremental update preserves the skill evolution history and avoids regenerating the entire library from scratch.

Family reset. This design follows the goal of SKILLFLOW to evaluate lifelong learning within a single class of agent tasks rather than the agent's ability to apply skills precisely across heterogeneous workflows. Interleaving tasks from different workflows in a single evaluation stream would introduce unnecessary system noise, including confounds from skill retrieval mechanisms.

Skill patch schema. We use a minimal auditable interface for skill evolution to avoid complex instruction-following effects that could compromise evaluation fairness. Skill patch generation is driven by the model's native capability and is decoupled from the agent harness used for task execution. In the equations above, g denotes the fixed prompt template that specifies how the model should generate skill patches; this template is provided in Appendix B.2. Each patch contains three fields: `summary`, `upsert_files`, and `delete_paths`. Together, these fields are sufficient to record the high-level lesson, create or overwrite `SKILL.md` files and helper scripts, and remove obsolete artifacts. The resulting skills follow the same standard skill structure described above. This file-level interface preserves patch history, supports update tracking over time, and makes failure modes such as uncontrolled skill growth directly inspectable when redundant or low-value artifacts accumulate in the library.

Skill use detection. Each task is completed under the planning policy of its native agent harness. Because these harnesses already support skills, the model can choose to use skills natively without any additional control mechanism. We only extract task-completion outcomes and execution-trace events associated with library files, such as skill reads or skill calls, to detect skill use and compute the skill reuse rate.

This protocol evaluates not only whether an agent can benefit from a static external memory, but also whether it can continuously maintain, repair, and improve a reusable skill repository over time.

2.5 Metrics

We evaluate agents using three metric groups:

- **Task success rate**: whether the final output satisfies the task verifier.
- **Efficiency**: runtime cost per task, reported through interaction turns, monetary cost, and output tokens.
- **Skill generation and reuse**: statistics that record both the average number of skills retained in the final family-local library and the rate at which previously stored skills are reused in later tasks.

3 Experiments

3.1 Experimental Setup

We evaluate SKILLFLOW under the **Agentic Lifelong Learning** protocol. Within each task family, an agent begins with an empty skill library, solves the first task without prior skills, and then updates the library iteratively through *skill patches* derived from execution trajectories and rubric feedback. To place each model in its strongest practical setting, we pair it with a matched agent wrapper and execution harness. We standardize only the interface for skill-patch generation by requiring a single-turn output format. This keeps the representation of generated skills consistent while preserving compatibility with the execution environment of each model.

The main experiments use Claude Code (Anthropic, 2025), Codex CLI (OpenAI, 2025), Qwen-Coder (Qwen Team, 2025), and Kimi-CLI (Moonshot AI, 2024) as evaluation harnesses. Across these harnesses, we evaluate 11 model variants: Claude Sonnet 4.5, Claude Opus 4.5, Claude Sonnet 4.6, Claude Opus 4.6, MiniMax M2.5, MiniMax M2.7, GPT 5.4, GPT 5.3 Codex, Qwen-Coder-Next, Qwen3-Coder-480B, and Kimi K2.5.

3.2 Observations

Table 1 shows that skill evolution helps some settings but not others. The clearest positive case is Claude Opus 4.6, which improves from 62.65% to 71.08% (+8.43 points, 104 to 118 completed tasks out of 166). Additional gains appear for MiniMax M2.5 (+6.63), Claude Sonnet 4.5 (+6.02), GPT 5.4 (+3.62), Claude Opus 4.5 (+2.41), and Kimi K2.5 (+0.60). By contrast, GPT 5.3 Codex drops by 6.02 points; Qwen-Coder-Next, Qwen3-Coder-480B, and MiniMax M2.7 each decline by 0.60 points; and Claude Sonnet 4.6 remains unchanged.

Figure 5 complements this pattern with an efficiency view: some settings move toward higher completion with comparable or lower cost, whereas others incur higher cost with limited or no gains. Figure 6 further shows that gains are spread across task-family groups rather than concentrated in a single category, although Finance & Economics more often shows negative transfer and Data & Document Intelligence more often shows positive transfer. Appendix Figure 13 provides the full family-level heatmap.

3.3 Main Results

To better understand model behavior under the Agentic Lifelong Learning protocol, we analyze the full set of experimental results from all 11 evaluated models and summarize the following findings.

Finding 1: Opus 4.6 comes closest to stable skill-based learning. Among the evaluated models, Opus 4.6 provides the clearest evidence that skill revision can improve the procedure stored in the library rather than merely accumulate successful traces. Table 1 and Figure 5 show the strongest overall improvement for this setting, with higher completion at lower cost. Trace analysis further shows cases in which repaired skills are reused after concrete failures, including spreadsheet workflows that require cached-value handling. We also conduct a control experiment that simply prepends the full prior interaction history as additional context. On Claude Opus 4.6, this setting reaches only 51.04%, below both vanilla and the full protocol (Appendix Table 2), which suggests that the gain is not explained by longer raw context alone. Within the scope of our benchmark, this is the clearest instance of stable library-level improvement.

Finding 2: Incorrect skills create systematic downstream drift. Once an incorrect skill is written into the library, later tasks often inherit the same flawed abstraction, turning a local error into a sequence-level pattern. This pattern likely reflects the difficulty of escaping a self-generated logic distribution, which suggests that, while external skills can amplify capability, they can also amplify error.

Finding 3: Unified high-utility skills outperform fragmented task-specific skills. The strongest libraries are usually organized around one or a few reusable skills that are revised repeatedly as new variants appear. In Table 1, stronger settings tend to end with smaller final skill inventories, whereas weaker settings often accumulate more skills with-

Table 1: Main Experimental Results. All values are benchmark-level averages. %comp. is task completion rate; Turns, Cost, and Out Tok. are per-task averages of interaction turns, USD cost, and output tokens (thousands). #Skills is the cumulative number of skills generated within a task family, and %use is the percentage of tasks that read or call at least one stored skill. In the Δ block, %comp. and Turns are absolute changes, whereas %Cost and %Out Tok. are relative percentage changes. Green indicates higher-is-better metrics, blue indicates lower-is-better metrics, and **bold** marks the best value in each column.

Agent	Model	vanilla				skills evolve						Δ			
		%comp.↑	Turns↓	Cost(USD)↓	Out Tok.(K)↓	%comp.↑	Turns↓	Cost↓	Out Tok.(K)↓	#Skills	%use	%comp.↑	Turns↓	%Cost↓	%Out Tok.↓
Claude Code	Claude Sonnet 4.5	49.4	25.04	0.293	1.07	55.42	24.88	0.246	0.85	2.55	72.89	+6.02	-0.16	-16.04	**-20.56**
	Claude Opus 4.5	58.43	18.83	0.571	1.5	60.84	18.31	0.384	1.4	1.5	60.84	+2.41	-0.52	**-32.87**	-6.67
	Claude Sonnet 4.6	56.63	17.48	0.168	1.23	56.63	17.42	0.245	1.59	2.55	53.01	+0.00	-0.06	+45.83	+29.27
	Claude Opus 4.6	**62.65**	17.34	0.665	3.00	**71.08**	19.00	0.615	2.39	1.05	45.78	**+8.43**	1.66	-7.52	-20.33
	MiniMax M2.5	28.31	35.22	**0.010**	**0.44**	34.94	34.01	**0.010**	**0.54**	2.50	32.53	+6.63	**-1.21**	0	+22.73
	MiniMax M2.7	37.35	25.44	0.012	0.5	36.75	27.42	0.017	0.96	4.6	51.2	-0.6	+1.98	+41.67	+92
Codex CLI	GPT 5.4	33.13	23.89	0.41	4.05	36.75	24.17	0.459	4.43	1.05	81.33	+3.62	+0.28	+11.95	+9.38
	GPT 5.3 Codex	52.41	17.74	0.492	6.8	46.39	17.14	0.434	6.82	1.1	84.94	-6.02	-0.6	-11.79	+0.29
Qwen Coder	Qwen-Coder-Next	45.18	18.64	0.103	9.74	44.58	19.91	0.113	10.69	5.45	12.05	-0.60	+1.27	+9.71	9.75
	Qwen3-Coder-480B	24.7	26.22	0.189	12.58	24.1	28.8	0.199	12.12	5.2	66.87	-0.6	+2.58	+5.29	-3.66
Kimi CLI	Kimi K2.5	55.42	**12.62**	0.103	7.31	56.02	**11.51**	0.104	7.10	1.50	66.87	+0.60	-1.11	+0.97	-2.87

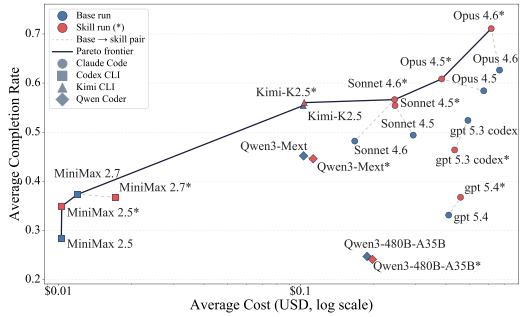

Figure 5: Completion–Cost Pareto Frontier. Each point is one evaluated agent-model setting under vanilla execution or lifelong skill evolution. Some settings shift toward higher completion with comparable or lower cost, while others gain little or regress despite additional spending.

Figure 6: Domain-Grouped Completion Gains from Skill Evolution. Rows correspond to evaluated agent-model settings, while columns are organized into coarse benchmark groups. Abbreviations: FE = Finance & Economics; OS = Operations & Supply Chain; HLS = Healthcare & Life Sciences; GS = Governance & Strategy; DDI = Data & Document Intelligence.

out comparable gains. This pattern suggests that consolidation is generally more valuable than proliferation.

Finding 4: Qwen and part of MiniMax mainly fail through skill inflation. For Qwen and some MiniMax settings, the main problem is often not skill absence but uncontrolled growth of overlapping skills. Appendix Figure 9 shows that the accumulated number of stored skills in these settings increases almost monotonically with task index, which suggests that they tend to summarize nearly every task into an additional skill. Yet Table 1 still shows weak or negative benchmark-level gains, and Figure 6 does not show correspondingly broad improvement across task families. In these cases, the dominant failure mode is better described as fragmentation through skill inflation than as skill scarcity.

Finding 5: Codex is relatively strong at consolidating variants into one evolving core skill. Codex shows a clearer tendency than most models to absorb nearby task variants into a shared evolving skill rather than proliferate many narrowly specialized entries. Appendix Figure 10 shows that Codex-based libraries remain relatively compact in file count compared with many other settings. However, Table 1 also shows that this compactness does not translate into stronger task completion: compared with Opus settings, which also keep their libraries relatively small, Codex does not achieve comparable end-to-end gains under the full harness.

Finding 6: The key model gap lies in repairing bad skills, not in writing skills. Most models can write some skill after completing or attempting a task, so raw skill generation is not the main source of variation. The larger difference is whether a model can recognize an incorrect skill, revise it, and obtain better behavior on later tasks. Across the table, the frontier plot, and the traces, positive transfer aligns more closely with effective skill

repair than with skill volume alone.

Summary. These findings show that iterative skill evolution depends strongly on model capability. Across settings, compact, well-revised skills are more effective than fragmented, isolated entries, while incorrect early skills can induce persistent negative transfer and degrade later performance. More broadly, the practical value of skill evolution depends less on writing more skills than on maintaining a small, repairable library of high-utility procedures. This property may also ease retrieval of relevant skills when extending such systems to more open-ended environments.

4 Related Work

4.1 Execution Environments and Benchmarks

A growing line of work evaluates agent capabilities in controlled environments with tool use and multi-step reasoning (Patil et al., 2025; Liu et al., 2023; Trivedi et al., 2024). Some benchmarks further focus on reliability aspects such as error correction in tool calling (Huang et al., 2025), as well as the inefficiency patterns that emerge during complex tool-integrated reasoning (Su et al., 2026). Other work studies whether explicit skill usage improves performance in realistic software engineering settings (Han et al., 2026b). Recent coding-agent benchmarks emphasize realistic, long-horizon tasks under shared Harbor-based execution setups for reproducibility and comparability (Merrill et al., 2026; Xia et al., 2024; Deng et al., 2025; Chou et al., 2025; Chen et al., 2026).

4.2 Skills as Procedural Knowledge for Agents

Recent studies treat **skills** as reusable procedural knowledge bridging models and workflows, including large-scale skill management, skill-aware benchmarking, and trajectory distillation into reusable skills (Liang et al., 2026; PinchBench Team, 2026; Ni et al., 2026). However, these works mainly focus on infrastructure or downstream performance, with limited evaluation of skill derivation and cross-task transfer.

4.3 Automatic Skill Discovery and Evolution

Another line of work explores automatic skill discovery and evolution from interaction, including distilling interaction patterns, refining skills through feedback and failures (Zheng et al., 2025; Yang et al., 2026; Alzubi et al., 2026; Zhang et al., 2026; Wei et al., 2025; Huang et al., 2026a), and self-improving models via self-generated supervision (Han et al., 2026a). Some approaches further model skills as persistent and improvable structures or leverage long interaction traces for accumulation (Xu et al., 2023; Zhou et al., 2026; Xia et al., 2026; Cao et al., 2026). These methods demonstrate performance gains from skill accumulation, but focus more on optimization than on evaluating skill abstraction, revision, and transfer.

5 Conclusion

We introduce SKILLFLOW, a benchmark evaluating lifelong skill discovery and evolution in autonomous agents. It contributes a dual-agent construction pipeline, an explicit sequential evaluation protocol, and a task set organized around Domain-Agnostic Execution Flow to test skill transfer at the workflow level instead of superficial lexical overlap. Experiments show that lifelong skill evolution yields selective rather than universal gains: stronger agent–model stacks convert externalized experience into compact reusable procedures, whereas weaker ones often exhibit a creation–reuse coordination gap, fragmented skill libraries, and unstable reinforcement under feedback. This positions SKILLFLOW as both a benchmark release and an initial empirical characterization of skill evolution under one concrete external-memory mechanism.

More broadly, as autonomous agents become increasingly general-purpose, continuous skill evolution may provide a practical mechanism for acquiring and consolidating knowledge across domains. In this sense, enabling agents to revise skills from experience appears important for building more robust and adaptive lifelong learning systems.

Limitations

This work studies lifelong skill discovery and evolution under one concrete external-memory setting: family-local skill libraries updated through explicit skill patches. While this design isolates the mechanism of interest and supports controlled evaluation, it does not cover all forms of agent adaptation, such as latent parameter updates, long-context accumulation, or richer retrieval systems. In addition, tasks are organized into fixed within-family sequences and families are reset independently, so the current protocol does not yet measure open-ended transfer across heterogeneous workflows over very long horizons. Finally, although SKILLFLOW spans diverse workplace-style domains and uses real exe-

cution environments, it still reflects the coverage, verifier design, and agent-harness choices of the current benchmark release. Extending the benchmark with broader task distributions, more agent frameworks, and longer cross-family curricula remains important future work.

References

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. 2026. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*.

Anthropic. 2025. Claude Code: an agentic coding tool. https://github.com/anthropics/claude-code.

Anthropic. 2026. anthropics/skills: Public repository for agent skills. https://github.com/anthropics/skills. GitHub repository. Accessed: 2026-04-13.

Anysphere. 2024. Cursor: Ai code editor. https://cursor.sh.

Weili Cao, Xunjian Yin, Bhuwan Dhingra, and Shuyan Zhou. 2026. Coding agents are effective long-context processors. *arXiv preprint arXiv:2603.20432*.

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, and 1 others. 2026. Skillcraft: Can llm agents learn to use tools skillfully? *arXiv preprint arXiv:2603.00718*.

Jason Chou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, and 1 others. 2025. Autocodebench: Large language models are automatic code benchmark generators. *arXiv preprint arXiv:2508.09101*.

ComposioHQ. 2026. awesome-claude-skills: A curated list of awesome claude skills, resources, and tools. https://github.com/ComposioHQ/awesome-claude-skills. GitHub repository. Accessed: 2026-04-13.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Zhen Fang, Zhuoyang Liu, Jiaming Liu, Hao Chen, Yu Zeng, Shiting Huang, Zehui Chen, Lin Chen, Shanghang Zhang, and Feng Zhao. 2025. Dualvla: Building a generalizable embodied agent via partial decoupling of reasoning and action. *arXiv preprint arXiv:2511.22134*.

Google. 2025. Gemini CLI: An open-source AI agent that brings the power of Gemini directly into your terminal. https://github.com/google-gemini/gemini-cli.

Ruiyan Han, Zhen Fang, XinYu Sun, Yuchen Ma, Ziheng Wang, Yu Zeng, Zehui Chen, Lin Chen, Wenxuan Huang, Wei-Jie Xu, and 1 others. 2026a. Unicorn: Towards self-improving unified multimodal models through self-generated supervision. *arXiv preprint arXiv:2601.03193*.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. 2026b. Swe-skills-bench: Do agent skills actually help in real-world software engineering? *arXiv preprint arXiv:2603.15401*.

Harbor Framework Team. 2026. Harbor: A framework for evaluating and optimizing agents and models in container environments. https://github.com/harbor-framework/harbor. GitHub repository. Accessed: 2026-04-16.

Shiting Huang, Zhen Fang, Zehui Chen, Siyu Yuan, Junjie Ye, Yu Zeng, Lin Chen, Qi Mao, and Feng Zhao. 2025. CRITICTOOL: Evaluating self-critique capabilities of large language models in tool-calling error scenarios. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Suzhou, China. Association for Computational Linguistics.

Shiting Huang, Zecheng Li, Yu Zeng, Qingnan Ren, Zhen Fang, Qisheng Su, Kou Shi, Lin Chen, Zehui Chen, and Feng Zhao. 2026a. Internalizing meta-experience into memory for guided reinforcement learning in large language models. *arXiv preprint arXiv:2602.10224*.

Wenxuan Huang, Yu Zeng, Qiuchen Wang, Zhen Fang, Shaosheng Cao, Zheng Chu, Qingyu Yin, Shuang Chen, Zhenfei Yin, Lin Chen, and 1 others. 2026b. Vision-deepresearch: Incentivizing deepresearch capability in multimodal large language models. *arXiv preprint arXiv:2601.22060*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, and 1 others. 2026. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, and 1 others. 2026. Skillnet: Create, evaluate, and connect ai skills. *arXiv preprint arXiv:2603.04448*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Moonshot AI. 2024. Kimi cli. https://github.com/MoonshotAI/kimi-cli. GitHub repository, commit abc123, Accessed: 2026-04-02.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Xiaoxi Jiang, and Guanjun Jiang. 2026. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*.

OpenAI. 2025. Codex CLI: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex.

OpenAI. 2026. Introducing gpt-5.3-codex. https://openai.com/index/introducing-gpt-5-3-codex/.

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

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simón Posada Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, and 1 others. 2025. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. *arXiv preprint arXiv:2510.04374*.

PinchBench Team. 2026. Pinchbench: Skill-based benchmark for llm agents. https://github.com/pinchbench.

Qwen Team. 2025. Qwen-code. https://github.com/QwenLM/qwen-code. GitHub repository, Accessed: 2026-04-02.

SkillsMP. 2026. Skillsmp: Skills marketplace. https://skillsmp.com/. Website. Accessed: 2026-04-13.

Qisheng Su, Shiting Huang, Zhen Fang, Ziyan Chen, Zehui Chen, and Feng Zhao. 2026. Beyond accuracy: Unveiling inefficiency patterns in tool-integrated reasoning. *Preprint*, arXiv:2604.05404.

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. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

VoltAgent. 2026. awesome-openclaw-skills: The awesome collection of openclaw skills. https://github.com/VoltAgent/awesome-openclaw-skills. GitHub repository. Accessed: 2026-04-13.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, and 1 others. 2025. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, and 1 others. 2026. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*.

Mengda Xu, Zhenjia Xu, Cheng Chi, Manuela Veloso, and Shuran Song. 2023. Xskill: Cross embodiment skill discovery. In *Conference on robot learning*, pages 3536–3555. PMLR.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, and 1 others. 2026. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*.

Yu Zeng, Wenxuan Huang, Zhen Fang, Shuang Chen, Yufan Shen, Yishuo Cai, Xiaoman Wang, Zhenfei Yin, Lin Chen, Zehui Chen, and 1 others. 2026. Vision-deepresearch benchmark: Rethinking visual and textual search for multimodal large language models. *arXiv preprint arXiv:2602.02185*.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. 2026. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, and 1 others. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and 1 others. 2025. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, and 1 others. 2026. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*.

A Benchmark Details

A.1 Benchmark Statistics

The final benchmark contains 20 task families and 166 tasks organized under five broad domains: *Finance & Economics*, *Operations & Supply Chain*, *Healthcare & Life Sciences*, *Governance & Strategy*, and *Data & Document Intelligence*. These five domains cover 20 workflow families in total, with family sizes ranging from 8 to 9 tasks. During construction, more than 200 candidate tasks are generated. Thirty-three of them are filtered out before final inclusion because of environment failures, ambiguity, invalid difficulty ordering, or workflow inconsistency.

A.2 Workflow Families

This section lists the 20 workflow families by their *workflow-definition names* rather than internal source identifiers. The grouping is derived from the curated benchmark spreadsheet and uses only the formal family names reported in the main paper.

- **Finance & Economics**: *Industry Correlation Analysis*, *Financial Statement Rolling*, and *SEC 13F Financial Analysis*.
- **Operations & Supply Chain**: *Supply Chain Replenishment*, *Production Capacity Planning*, *Inventory & Finance Integration*, *DMAIC Quality Analysis*, and *Operational Recovery Planning*.
- **Healthcare & Life Sciences**: *Healthcare Cost-Benefit Analysis* and *Medical Data Standardization*.
- **Governance & Strategy**: *Distribution Center Auditing*, *Compensation Scenario Modeling*, and *Document Fraud Detection*.
- **Data & Document Intelligence**: *Embedded Data Repair*, *OCR Data Extraction*, *HWPX Document Automation*, *Cross-Format Data Reconciliation*, *Weighted Risk Assessment*, *PPT Formatting Optimization*, and *Sales Pivot Analysis*.

DAEF Annotation Rules. The annotation process follows five rules. First, each task is annotated with 5–8 meta-step nodes. Second, node labels must be selected only from the controlled vocabulary in Table 5, and annotators are not allowed to invent new labels. Third, dependency edges are directed from prerequisite operations to subsequent operations, and cycles are not permitted. Fourth, when two operations can be executed in parallel, both are connected to the same downstream node. Fifth, after annotation, annotators must provide a brief operational rationale in one sentence explaining why each node label is selected.

DAEF Agreement Screening. To verify that a seed task admits a stable workflow abstraction, two expert annotators independently produce DAEF annotations for each candidate task in the final screening pool, which yields 30 retained tasks after screening. The annotators then score the other annotator's graph against their own judgment, checking agreement on the meta-step decomposition, node labels, and dependency structure. A candidate passes only when both annotators judge the two annotations to be mutually consistent. Otherwise, we treat the seed task as lacking a stable execution workflow and discard it from benchmark construction.

A.3 Human Verification Checklist

Human reviewers inspect each candidate family with this checklist:

- **Solvability**: a correct solution path exists under the provided environment and assets.
- **Logical consistency**: instructions, assets, and expected outputs do not contradict one another.
- **Environment closure**: the task can be completed with the provided files, tools, and runtime.
- **Difficulty ordering**: later tasks in a family are not easier than earlier tasks under the intended curriculum.
- **No skill leakage**: instructions do not reveal which exact skill should be used.

For family acceptance, reviewers additionally assess whether the candidate tasks satisfy the DAEF membership rule:

- **Shared operation inventory**: the core operation types are consistent across tasks.
- **Shared dependency topology**: these operations follow the same dependency pattern rather than merely similar names.

- **Grounding variation only**: differences arise from domain entities, file formats, noise, scale, or instruction phrasing rather than from a new workflow core.

- **Transfer plausibility**: a procedure learned on earlier tasks should plausibly transfer to later tasks without requiring a new workflow.

A.4 Comparison

Table 3 compares SKILLFLOW with representative benchmarks from the perspective of skill lifecycle evaluation. We focus on seven properties: **Skill Eval** (whether the benchmark explicitly evaluates the impact of skills), **Self-Gen** (support for self-generated skills), **Revision** (iterative updates or patching of skills), **Lifelong** (sequential accumulation and reuse of skills across tasks), **Transfer** (evaluation of cross-scenario procedural transfer), **Traj-Grounded** (whether skills are extracted from interaction trajectories), and **Usage Eval** (whether the benchmark evaluates the alignment between skill utility and actual usage).

A.5 Workflow Details

Table 4 provides a compact view of the 20 workflow families. Internal source identifiers are omitted. Instead, the table reports the workflow definition, seed-benchmark lineage, a short task summary, and the corresponding DAEF description. The **Source** column indicates whether the family is derived from a seed task selected from SkillsBench or GDPval.

A.6 DAEF Node Vocabulary

Table 5 lists the controlled node labels used when annotating seed-task workflow graphs. The vocabulary is intentionally compact and coarse-grained so that it captures transferable procedural roles rather than low-level commands or domain-specific business concepts.

A.7 Allowed Surface Variation Types

Table 6 summarizes the variation dimensions that annotators may allow within a family while preserving a shared DAEF. These dimensions are intended to alter surface realization and difficulty without introducing a new workflow core.

B Skill Evolution Details

B.1 Skill Patch Format

Each skill patch in the current framework is represented as a JSON object with three top-level fields:

- `summary`: a natural-language description of the newly extracted lesson or repair.

- `upsert_files`: a mapping from file paths to new or updated file contents.

- `delete_paths`: a list of obsolete files to remove from the skill library.

This schema is used as a minimal auditable interface for file-level updates rather than as an optimal or exhaustive formalism for skill evolution. Its purpose is to make patch history inspectable, support update tracking over time, and make failure modes such as uncontrolled skill growth directly visible when redundant or low-value content accumulates.

B.2 Skill Patch Generation Prompt Template

This subsection records the prompt template used for skill patch generation. For readability, we separate the system-level instruction and the user-level prompt template into two boxed figures, referenced as Figure 7 and Figure 8.

B.3 Skill Library Growth and Composition

Figure 9 and Figure 10 provide two complementary views of how the external skill library evolves during sequential evaluation. The first figure tracks the cumulative number of stored skills as tasks progress, while the second summarizes the file-type composition of the resulting skill libraries.

C Experiments and Analysis Details

C.1 Metric Notes

The main text emphasizes aggregate task completion, efficiency statistics (turns, monetary cost, and output tokens), and statistics of skill generation and reuse because these signals are consistently available across the systems reported in Table 1. In Table 1, Final #Skills denotes the cumulative number of skills generated while completing a task family, and %use denotes the percentage of tasks in which the agent reads or calls at least one previ-

ously stored skill.

C.2 Historical-Trajectory Context Control

To test whether the gain of the full protocol can be explained by access to longer raw histories alone, we run a single-model control on Claude Opus 4.6 in which the agent receives the full prior interaction history as additional context, but does not externalize that experience into a skill library. Table 2 compares this setting against the vanilla baseline and the full protocol. In this comparison, the history-context setting reaches 47.41% completion, which is 15.24 points below vanilla and 23.67 points below the full protocol. While this control does not isolate every factor and is limited to one model, it suggests that simply appending raw historical trajectories is not sufficient to recover the benefit of the full protocol; the observed gain is more consistent with structured externalization into reusable skills than with longer context alone.

Additional Efficiency Pareto Views. Figure 11 and Figure 12 provide complementary Pareto views of task completion against interaction turns and output tokens.

Full Skill-Gain Heatmap. Figure 13 provides a family-level complement to the coarse domain-grouped heatmap shown in the main text. It resolves the benchmark into individual workflow families and gives a more granular view of where skill evolution yields large gains, negligible changes, or regressions.

Setting	Completion (%)	Δ vs. Vanilla
Vanilla	62.65	–
History context	51.04	-11.61
Skill-library protocol	71.08	+8.43

Table 2: Single-model control on Claude Opus 4.6. The history-context setting prepends the full prior interaction history as additional context, without externalizing the experience into skills.

Table 3: Comparison between SKILLFLOW and representative benchmarks from the perspective of skill lifecycle evaluation.

Benchmark	Skill Eval	Self-Gen	Revision	Lifelong	Transfer	Traj-Grounded	Usage Eval
Terminal-Bench (Merrill et al., 2026)	✗	✗	✗	✗	✗	✗	✗
SWE-bench (Jimenez et al., 2023)	✗	✗	✗	✗	✗	✗	✗
SkillsBench (Li et al., 2026)	✓	✓	✗	✗	✗	✗	✗
PinchBench (PinchBench Team, 2026)	✗	✗	✗	✗	✗	✗	✗
SWE-Skills-Bench (Han et al., 2026b)	✓	✗	✗	✗	✗	✗	✗
SKILLFLOW (Ours)	✓	✓	✓	✓	✓	✓	✓

Table 4: Workflow Families and Seed-Benchmark Lineage

Workflow definition	**Source**	**Task summary**	**DAEF**
Industry Correlation Analysis	SkillsBench	Detrend revenue series for paired industries, compute Pearson correlations, and judge cyclical association.	read source series → extract target fields → normalize and align time series → compute annualized and deflated values → compute filtered correlation signals → compare cyclical association → output scalar result.
Financial Statement Rolling	GDPval	Update monthly financial workbooks with reserve merges, accrual rolling, deferred revenue, liabilities, warranties, project cost, rebates, and commissions.	read multi-source ledgers → extract target entries → filter duplicate and invalid rows → align override and adjustment items → compute rolling balances → compare against control totals → update and output summary workbook.
SEC 13F Financial Analysis	SkillsBench	Analyze quarterly SEC 13F filings for cross-period reconciliation, fund categorization, holding changes, issuer ownership, and manager comparison.	read multi-period filings → extract entity records → normalize identifiers → align cross-period holdings → compute target aggregates → compare across periods → output structured analysis.
Supply Chain Replenishment	GDPval	Produce replenishment plans across categories such as bakery, beauty, frozen meals, household routes, pet care, produce, and snacks.	read inventory and transit data → extract target SKUs → filter duplicate and invalid rows → align master data with arrivals → compute coverage and shortage timing → compare against replenishment thresholds → output detail and action sheets.
Production Capacity Planning	GDPval	Build and vary a manufacturing catch-up capacity plan under demand, backlog, overtime, and buffer constraints.	read demand and parameters → retrieve initial state → compute period-level demand and backlog updates → compare candidate resource actions → compute overtime and buffer outcomes → output planning workbook.
Inventory & Finance Integration	GDPval	Integrate replenishment planning with financial rolling scenarios across mixed business contexts.	read multi-source data → extract and normalize keys and time windows → align inventory and finance records → compute rolling deltas and coverage → compare against exception rules → output structured results.

Workflow definition	Source	Task summary	DAEF
DMAIC Quality Analysis	GDPval	Perform DMAIC Analyze-stage diagnostics across logistics, hospitals, SOC alerts, university IT, DevOps, and field service settings.	read process data → extract analysis window → filter valid samples → compute grouped statistics → compare trend and significance signals → detect anomalies → output structured diagnostics.
Operational Recovery Planning	GDPval	Create recovery plans under multi-scenario operational disruptions such as harvesting, data centers, warehouses, manufacturing, radiology, and returns.	read baselines and constraints → extract planning horizon → compute candidate recovery plans → compare capacity and calendar outcomes → detect backlog gaps → output plan sheets and summaries.
Healthcare Cost-Benefit Analysis	GDPval	Compare healthcare supply-chain scenarios under different cycle and batch settings to assess cost-benefit trade-offs.	read inputs and mappings → extract and normalize aliases and keys → filter valid records and exclusions → align coverage with adjustments → compute revenue cost and margin scenarios → compare trade-off thresholds → output structured conclusions.
Medical Data Standardization	SkillsBench	Standardize medical laboratory data across units, precision rules, and formatting conventions.	read raw tables and templates → extract valid rows → align with target schema → normalize numeric formats and units → detect anomalies or missing values → validate required precision → output standardized file.
Distribution Center Auditing	GDPval	Audit distribution-center records for inventory discrepancies, outbound lists, receiving anomalies, return handling, SLA compliance, labor, and trailer detention.	read source data → extract audit fields → compute row-wise rule checks → detect violations and anomalies → compare totals across dimensions → output audit artifacts.
Compensation Scenario Modeling	GDPval	Build and refresh multi-year compensation models for organizations such as orchestras, universities, property firms, airlines, and construction teams.	read assumptions and master data → retrieve model structure and references → align base records across sheets → compute period-wise compensation values → update and compare scenario indicators → output completed model.
Document Fraud Detection	SkillsBench	Review commercial documents across clinics, trials, maintenance logs, travel, and cold-chain records to detect inconsistencies and suspicious items.	read source documents → extract candidate records and references → normalize names and identifiers → align cross-document evidence → compare claimed facts with authorizations → detect anomaly types → output structured flags.

Workflow definition	Source	Task summary	DAEF
Embedded Data Repair	SkillsBench	Repair or refresh embedded spreadsheet objects inside PowerPoint files for domains such as chemistry, catalysts, or foreign exchange.	read document container → extract embedded tables and target cells → retrieve label mappings → update cell values and formulas → validate dependencies → output repaired document.
OCR Data Extraction	SkillsBench	Extract structured records from scanned images such as legal settlements, measurements, orders, invoices, fuel slips, and medicine labels.	read scanned images → extract OCR fields and target records → normalize dates amounts and codes → align with template rows → validate supplemented references → output structured results.
HWPX Document Automation	SkillsBench	Fill Korean HWPX templates for clinics, announcements, inventory, proposals, renewals, audits, vendor directories, and training feedback.	read templates and data sources → extract placeholders and values → normalize transformed values and enumerations → update filled content and structure → validate document integrity → output completed file.
Cross-Format Data Reconciliation	SkillsBench	Compare archived PDF snapshots with current Excel data to find added, deleted, or modified records across domains.	read archived snapshot and current spreadsheet → extract primary keys and records → normalize and align across formats → compare additions deletions and modifications → detect field-level differences → output structured results.
Weighted Risk Assessment	SkillsBench	Implement weighted risk formulas in Excel for scenarios such as API SLA, campus budgeting, factories, hospitals, energy, cloud reliability, and ports.	read workbook and lookup tables → retrieve input and output ranges → update formulas with target cells → compute weighted indicators and risk scores → validate workbook consistency → output result workbook.
PPT Formatting Optimization	SkillsBench	Normalize image-title formatting in PowerPoint decks by unifying fonts, positions, and index pages.	read presentation objects → extract target titles and tags → normalize text font and layout attributes → update positions and alignment → validate summary-page consistency → output formatted document.
Sales Pivot Analysis	SkillsBench	Build pivot-style analysis from PDF catalogs and Excel transaction data for sales, budgets, payroll, registration, inventory, libraries, quality, and grades.	read multi-source data → extract and normalize fields and dimensions → align supplementary joins → compute pivot indicators → compare grouped summaries → output report workbook.

Table 5: Controlled DAEF Node Vocabulary

Node label	Operational meaning	Typical examples in released workflows
read	Load or inspect source files, tables, documents, images, or structured records	Read ledgers, filings, scanned images, templates, workbooks, PPTX containers, and PDF snapshots
extract	Identify and isolate target fields, records, entities, or patterns from complex or unstructured sources	Extract OCR fields from scanned images, extract candidate records from documents, extract embedded tables from containers
retrieve	Locate or select target states, ranges, placeholders, or action candidates from available context	Retrieve initial state, retrieve target entries, retrieve audit fields, retrieve target cells, retrieve placeholders, retrieve input/output ranges
normalize	Standardize schema, keys, units, formatting, aliases, or transformed values before downstream reasoning	Normalize entity identifiers, numeric formats, lab units, dates, codes, font attributes, and transformed enumerations
filter	Retain valid records and remove irrelevant, duplicate, malformed, or excluded cases	Filter duplicate ledger rows, filter excluded healthcare cases, filter invalid samples, filter malformed source entries
align	Join, reconcile, or map multiple sources under a shared key, structure, or layout	Align time series, align cross-period holdings, align inventory with arrivals, align finance records, align label mappings, align records across formats
compute	Derive intermediate or final quantities through formulas, aggregation, scoring, iterative updates, or planning operations	Compute rolling balances, compute coverage windows, compute backlog updates, compute revenue and margin scenarios, compute weighted risk scores, compute pivot indicators
compare	Contrast values, scenarios, plans, grouped summaries, or cross-period states against a threshold or reference target	Compare cyclical association, compare control totals, compare replenishment thresholds, compare candidate resource actions, compare trade-off thresholds, compare grouped summaries, compare record changes
detect	Identify anomalies, violations, exceptions, gaps, or field-level changes that require action	Detect suspicious claims, detect audit violations, detect missing values, detect backlog gaps, detect anomaly types, detect field-level differences
update	Modify, fill, repair, or refresh existing artifacts such as cells, documents, or configurations	Update embedded cell values, update workbook formulas, update document formatting, update template fields, update configuration values
validate	Check whether outputs satisfy structural, numeric, dependency, formatting, or integrity constraints before release	Validate required precision, validate workbook consistency, validate formulas and dependencies, validate cache integrity, validate reference supplements, validate summary-page consistency
output	Write the final artifact in the required structure, file format, workbook, document, or report layout	Output scalar results, output structured diagnostics, output linked summary workbooks, output audit artifacts, output repaired documents, output formatted presentations

Table 6: Allowed Surface Variation Types Within a DAEF Family

Variation type	What may change	Representative examples in the release
Domain entities	Business objects, named fields, organizational units, or scenario semantics	Vendor vs. patient vs. warehouse records; holdings vs. inventory vs. clinical measurements
File format	Surface container or artifact type while preserving the same procedural core	XLSX vs. CSV vs. JSON inputs; PPTX embedded spreadsheets vs. standalone workbooks; OCR image vs. PDF table
Data scale	Number of rows, time periods, scenarios, or cross-file joins	More SKUs, longer time horizons, larger audit logs, additional quarterly filings
Noise and quality	Missing values, alias ambiguity, OCR noise, inconsistent formatting, or extra distractors	Misspelled headers, low-quality scans, inconsistent units, partial records, cluttered slides
Constraint regime	Thresholds, policy rules, reconciliation targets, or validation tolerances	Different SLA cutoffs, coverage rules, accounting constraints, exception thresholds
Instruction surface	Wording, ordering, or presentation of the task request without changing the workflow core	Domain-specific phrasing, reformulated reporting goals, alternate output wording
Difficulty level	The amount of bookkeeping, exception handling, or multi-step reasoning required	Additional scenario branches, more exception rules, harder reconciliation chains, denser formatting checks

System Prompt Template

A skill is a reusable capability package, not a task-specific note. Improve the library so a future agent can solve similar tasks faster, more reliably, and with fewer failed attempts.

Hard structural rules (treat these as requirements, not suggestions):

- A new skill should normally live in its own capability-named directory, for example `fill-pdf-forms/` or `api-debugging/`.

- Every skill directory must contain `SKILL.md` at its root.

- `SKILL.md` must begin at the first character of the file with YAML frontmatter in exactly this shape:

```
---
name: <skill-name>
description: <what the skill does and when to use it>
---
```

- The frontmatter may contain exactly two keys: `name` and `description`. Do not add any other metadata keys.

- After the closing `---`, write normal Markdown instructions. Prefer starting the body with `# <Readable Title>`.

- Put executable helpers only in `scripts/`.

- Put long-form documentation, schemas, API notes, and detailed examples only in `references/`.

- Put copyable templates or non-context assets only in `assets/`.

- Do not create empty placeholder files or directories.

- Do not create README, CHANGELOG, INSTALLATION_GUIDE, QUICK_REFERENCE, or any process notes.

- If you add files under `scripts/` or `references/`, `SKILL.md` must explicitly tell a future agent when to run or read them.

- Keep references one hop away from `SKILL.md`; avoid deep navigation or nested indirection.

- If an existing skill already covers the capability, update that skill instead of creating a parallel duplicate.

Design principles:

- Be concise. Add only information that is non-obvious, reusable, and worth the context cost.

- Generalize from the trace, but do not merely restate it. Infer the reusable workflow, decision points, validation steps, tool patterns, anti-patterns, and troubleshooting notes that should change a future agent's behavior.

- Prefer verifier evidence, failed tests, and concrete execution results over the agent's self-report when they conflict.

- Prefer minimal edits to the existing library over broad rewrites.

- Use progressive disclosure. Keep `SKILL.md` focused and short; move detailed material into `references/` or `scripts/` only when it improves reuse.

- Avoid duplication across `SKILL.md` and `references/`.

- Use `scripts/` for deterministic, fragile, or repeatedly rediscovered code patterns.

- Use `references/` for detailed schemas, API notes, long examples, or variant-specific details.

- Use `assets/` only for files that the agent should copy or use directly in outputs.

- The `description` field must explain both what the skill does and when to use it, including trigger contexts.

- The `SKILL.md` body should be imperative, operational, and easy to scan. Prefer workflows, decision rules, and concise examples over long prose.

- Keep `SKILL.md` under 500 lines when possible.

When interpreting the trace:

- Identify where the agent's reasoning or chosen strategy was wrong, incomplete, or too brittle.

- Treat repeated failures, failed tests, verifier mismatches, and dead-end tool choices as signals for what future agents should avoid.

- If the agent succeeded after trial and error, capture the final working pattern, the discarded bad paths, and the key decision rule that separates them.

- If the agent failed, capture the missing knowledge, validation steps, troubleshooting workflow, and the most plausible alternative approach or escalation path that should be tried earlier next time.

```
- Prefer decision rules such as "if X pattern appears, do Y instead of Z" over vague advice.

- If the trace does not justify a meaningful reusable change, return an empty patch and explain why.

Output requirements:

- Return exactly one JSON object with keys: `summary`, `upsert_files`, `delete_paths`.

- `upsert_files` must map relative file paths to full file contents.

- `delete_paths` must only include paths that should truly be removed as obsolete.

- Do not wrap the JSON in commentary.
```

Figure 7: System prompt template used for skill patch generation.

USER_PROMPT_TEMPLATE

```
# Shared skill evolution task
You are updating the shared skill library rooted at the current shared-skills directory.
All file paths in the patch must be relative to that root.

## Required skill layout
When creating a new skill, use this default structure unless an existing skill for the same capability already exists:

skill-name/
|-- SKILL.md
|-- scripts/      # optional executable helpers
|-- references/   # optional reference docs loaded when needed
`-- assets/       # optional templates or files used in final outputs

`SKILL.md` must begin exactly like this, with nothing before the first `---`:

---
name: skill-name
description: Explain what the skill does and when to use it. Include trigger scenarios, file types, or task patterns.
---

# Readable Title

Only `name` and `description` are allowed in the frontmatter.

## Existing skill library
### Tree
{tree_json}

### Existing files
{files_block}

## Trial summary
- Task name: {task_name}
- Task source: {task_source}
- Verifier passed: {verifier_passed}
- Reward: {reward}
- Exception: {exception_info}
- Failed tests: {failed_tests}

## Final agent message
{final_message}

## Compacted execution trace
{trajectory_json}

## Internal workflow to follow
1. Derive the reusable capability from this trace.
2. Compare the agent's apparent plan against verifier outcomes, failed tests, and concrete tool results. Explicitly
identify any wrong assumptions, brittle choices, or dead-end strategies.
3. Identify the strongest trigger phrases or task types this skill should support.
4. Extract the minimal reusable workflow, validation steps, failure-prevention guidance, and anti-patterns to avoid.
5. If the current approach failed or was brittle, infer the next-best direction, fallback, or escalation path that a
future agent should try earlier, even if that exact fix was not fully executed in the trace.
6. Convert those lessons into reusable decision rules, such as when to switch tools, inspect lower-level formats, add
verification earlier, or abandon a high-level API.
7. Decide whether the knowledge belongs in an existing `SKILL.md`, a new skill directory, `references/`, `scripts/`, or
 `assets/`.
8. Keep the patch small, high-signal, and generalized.

## Skill authoring checklist
- Do not encode task-specific filled values, IDs, or one-off outputs unless they belong in a reusable template.
- Prefer one skill directory per capability.
```

```
- If creating a new skill, create `skill-name/SKILL.md` instead of writing a bare `SKILL.md` at the library root.
- Put executable code in `skill-name/scripts/...`.
- Put long documentation or variant-specific details in `skill-name/references/...`.
- Put templates or output assets in `skill-name/assets/...`.
- If you add `scripts/` or `references/`, update `SKILL.md` so a future agent knows when to use them.
- In `SKILL.md`, put "when to use" guidance in `description`, not as a separate metadata field.
- In the body, provide:
  - a short overview of the workflow
  - clear sequential steps or a decision tree when useful
  - validation or verification steps when reusable
  - concise troubleshooting notes for the important failure modes seen in the trace
  - explicit anti-patterns or "do not rely on this when..." guidance when the trace shows a tempting but wrong path
  - a fallback or alternative direction when the trace suggests a better next attempt for future runs
  - explicit pointers to `scripts/` or `references/` if you add them
- Use imperative style.
- Keep `SKILL.md` concise. Move long or variant-specific content into `references/`.
- Only add scripts when deterministic code is genuinely reusable and brittle enough to deserve bundling.
- If the current library already contains a relevant skill, update it instead of creating a parallel duplicate.

## Output contract
Return exactly one JSON object matching this shape:

{
  "summary": "why this patch helps future runs",
  "upsert_files": {
    "skill-name/SKILL.md": "---\nname: skill-name\ndescription: what it does and when to use it\n---\n\n# Readable
Title\n...",
    "skill-name/scripts/example.py": "#!/usr/bin/env python3\n...",
    "skill-name/references/details.md": "# Details\n..."
  },
  "delete_paths": []
}

Return no prose before or after the JSON object.
```

Figure 8: User prompt template used for skill patch generation.

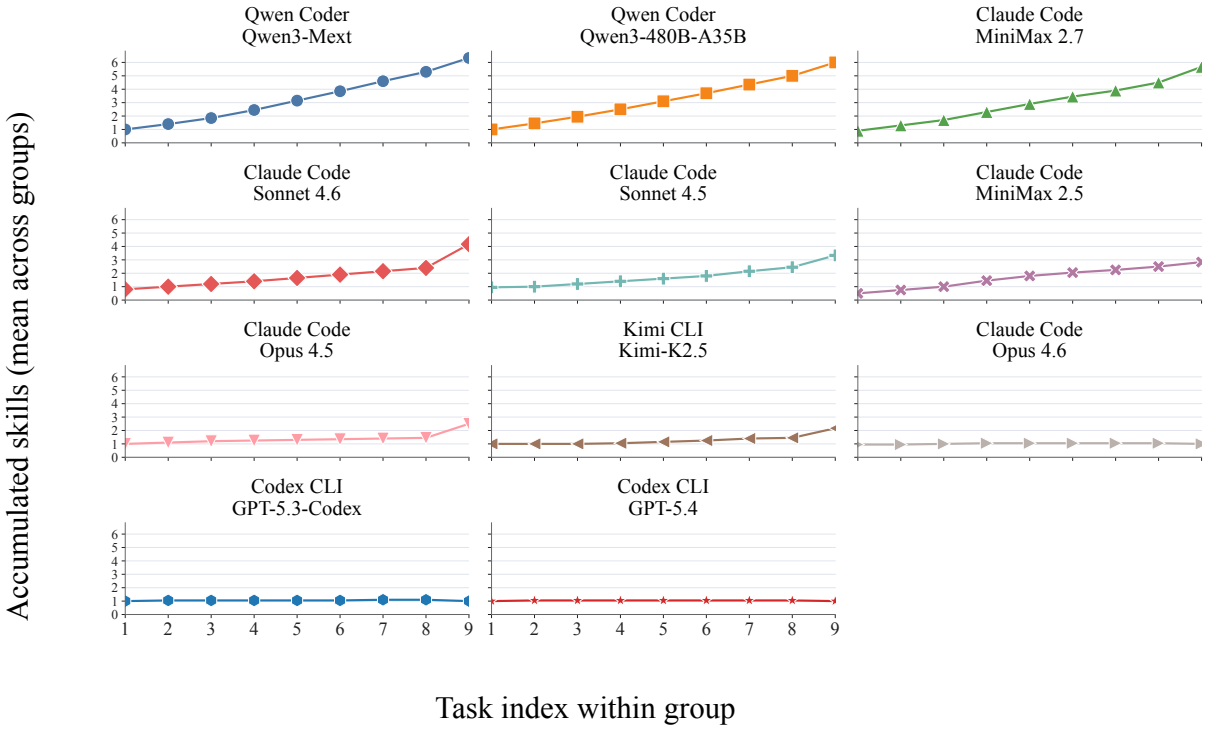

Figure 9: Skill Count Growth by Task. The figure shows how the cumulative number of stored skills changes as the agent proceeds through sequential tasks, which helps reveal whether different models consolidate experience into a compact library or continue to expand the library over time.

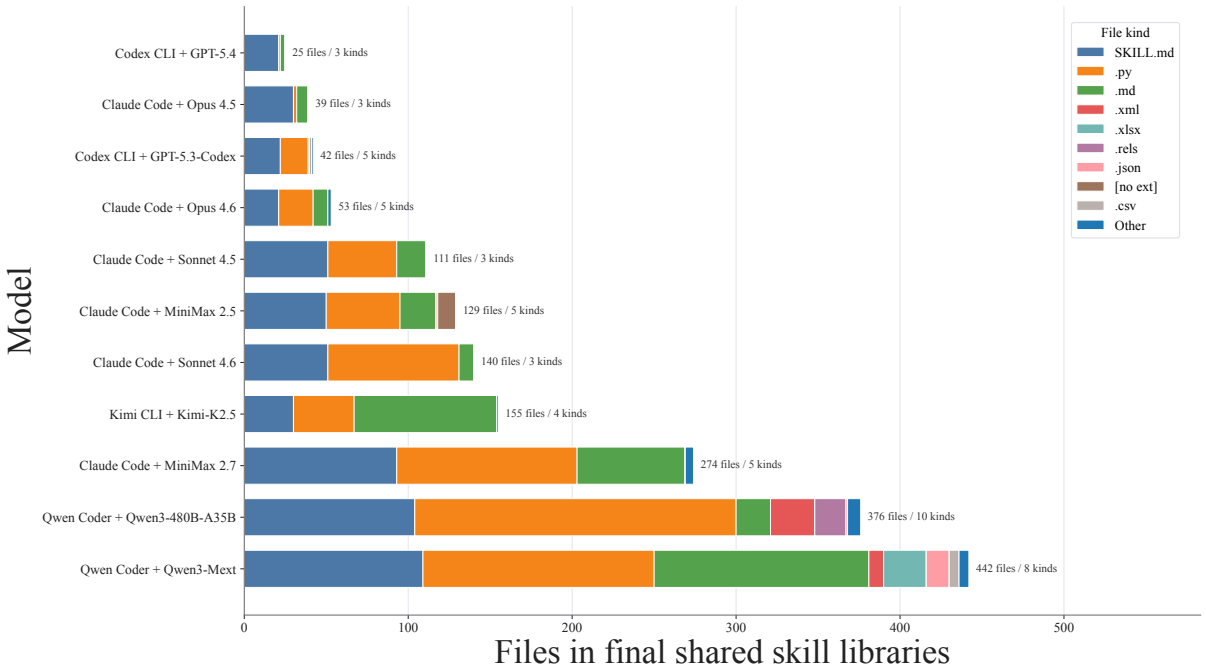

Figure 10: Skill File-Kind Composition. The figure summarizes the composition of file kinds contained in the evolved skill libraries, which helps characterize whether different models tend to store compact procedural summaries, auxiliary files, or more fragmented collections of artifacts.

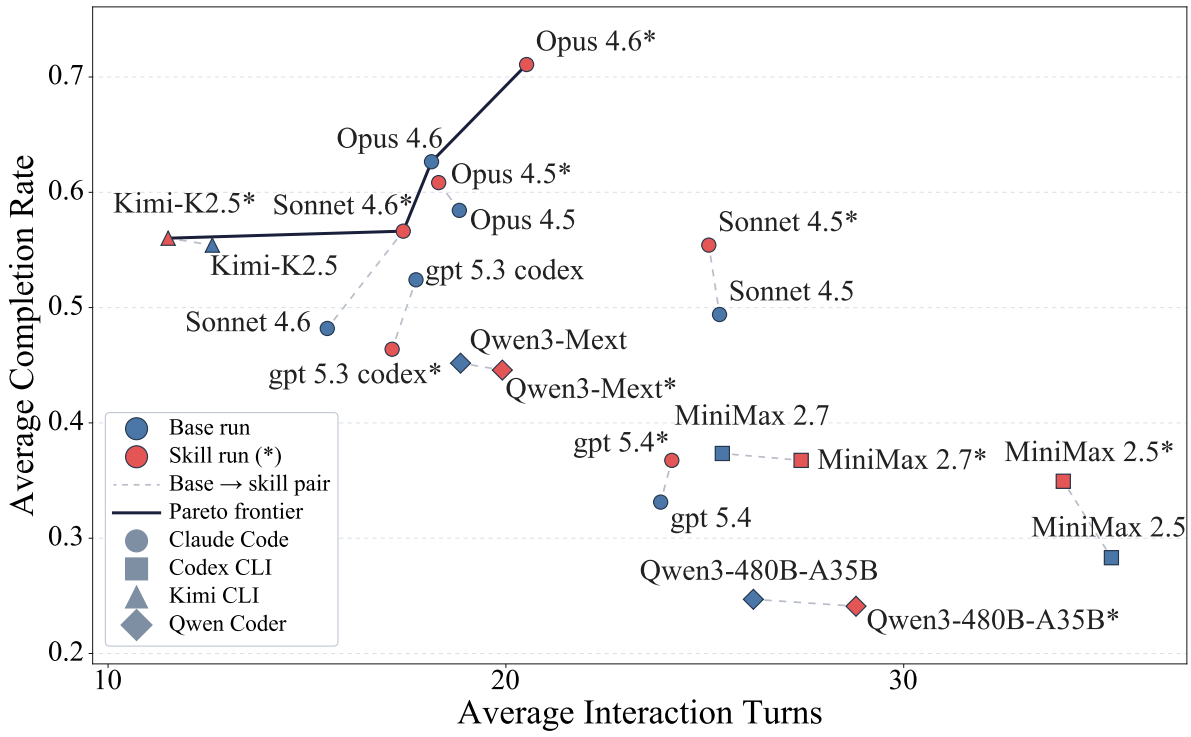

Figure 11: Completion–Interaction Turns Pareto Frontier. This view complements the main-text cost frontier by showing whether gains from skill evolution come with longer or shorter interaction sequences under the same evaluation setting.

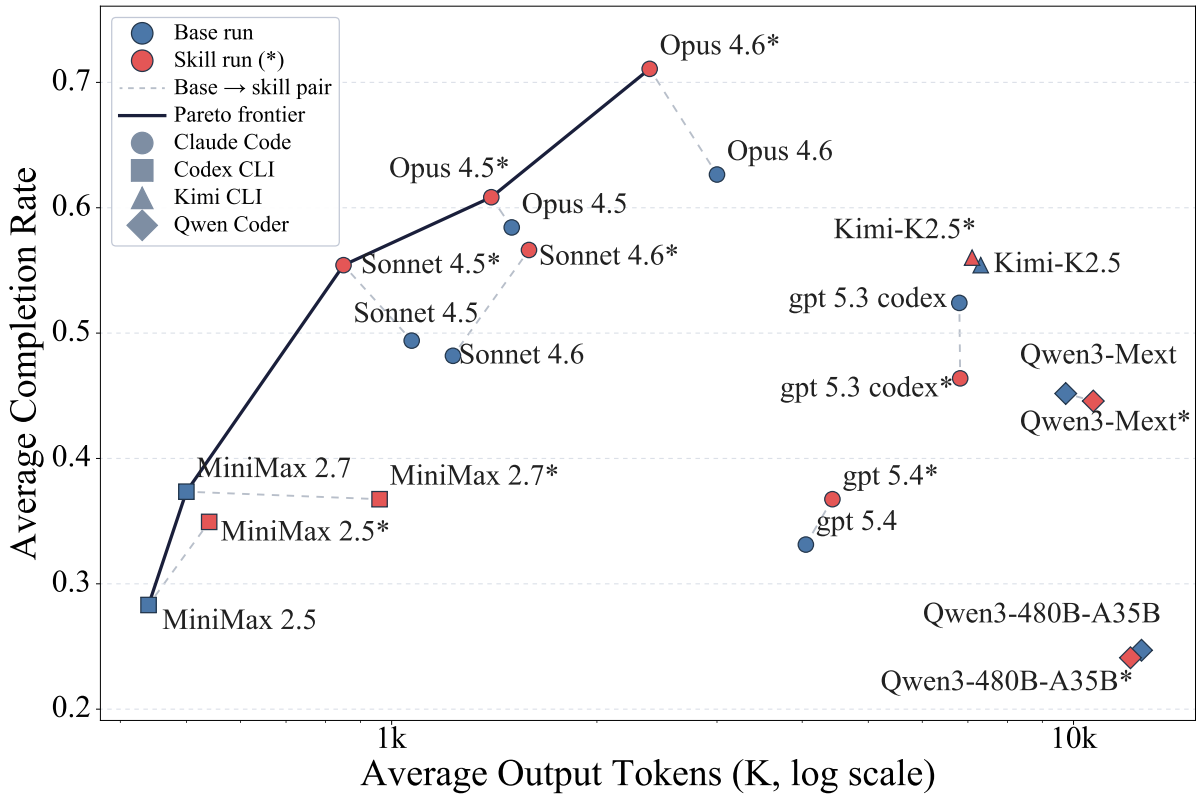

Figure 12: Completion–Output Tokens Pareto Frontier. This figure highlights the efficiency trade-off between task success and generated output volume, which is useful for diagnosing verbosity-driven gains or regressions across wrappers.

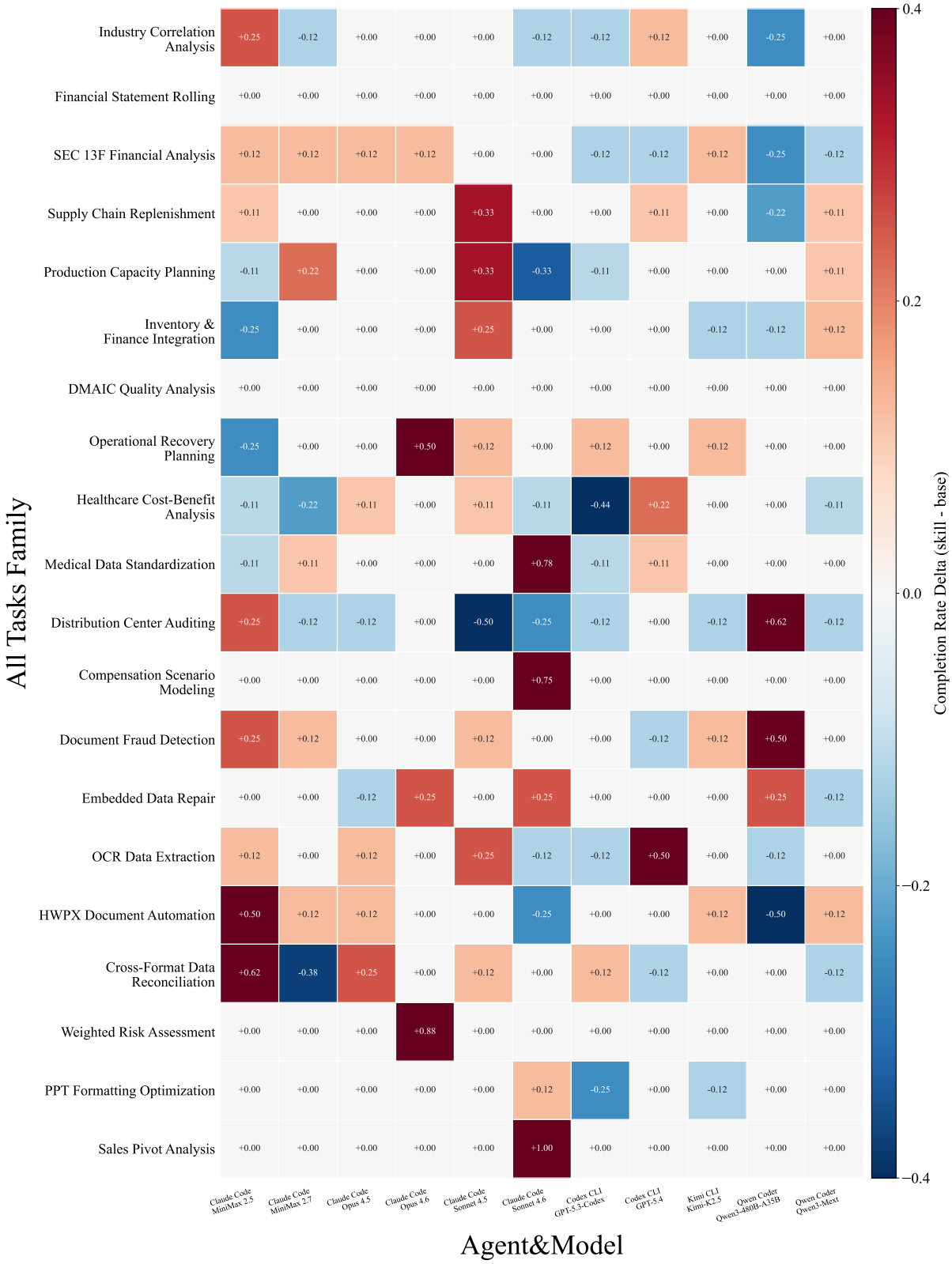

Figure 13: Family-Level Skill-Gain Heatmap. Compared with the coarse domain-grouped heatmap in the main text, this appendix figure resolves individual workflow families so that localized gains, flat regions, and regressions can be inspected in detail.

D Full Trajectory

D.1 Skill Evolution Patch

Patch on task:weighted-hospital-bedflow-calc

summary:

Updates excel-formula-tasks skill with a robust XML-level cached value patching script and workflow guidance. The trace revealed that: (1) the formulas library doesn't reliably cache values in openpyxl-readable format, (2) LibreOffice is often unavailable, (3) XML-level patching is the reliable fallback but must be done carefully in a single pass to avoid corrupting formula cells, and (4) patching must handle both numeric and string formula results. The new script automates the compute-and-patch workflow that the agent discovered through trial and error.

upsert_files["excel-formula-tasks/SKILL.md"]:

```
---
name: excel-formula-tasks
description: Populate Excel workbooks with formulas using openpyxl. Use when a task requires
writing INDEX/MATCH lookups, derived calculations, summary statistics, or weighted aggregations
into .xlsx files. Triggered by tasks involving highlighted/yellow cells to fill, multi-sheet
workbooks with source data, and formula-only requirements.
---

# Excel Formula Tasks with openpyxl

## Critical First Step: Read the Test File

Before writing any formulas, **always read the test/verifier file** (e.g., `test_output.py`) to
understand:
- Whether the verifier checks formula strings or computed values
- Expected cell addresses and value tolerances
- Whether it uses `data_only=True` (expects cached values) or checks formula text
- Any structural requirements (sheet names, cell formats, etc.)

```python
import os
for root, dirs, files in os.walk('/root/'):
    for f in files:
        if 'test' in f.lower():
            print(os.path.join(root, f))
```

Formula vs. Value Problem

This is the #1 failure mode. openpyxl writes formula strings but does NOT evaluate them. If
the verifier loads with `data_only=True`, all formula cells return `None`.

Decision Rule
- If verifier uses `data_only=True` or checks numeric values $\rightarrow$ you **must** also write
cached values
- If verifier checks formula strings $\rightarrow$ write formulas normally
- If no test file exists $\rightarrow$ **always write cached values** as a safety measure

Cached Value Strategy (Recommended Order)

1. **XML-level patching** (most reliable, no external deps): Write formulas with openpyxl, then
use `scripts/patch_cached_values.py` to compute values in Python and inject them into the XLSX
XML. **Do this in a single pass** --- multiple XML patches can corrupt formula cells.
2. **LibreOffice recalc**: If available, use `scripts/libreoffice_recalc.sh`. Check with `which
libreoffice`.
3. **`formulas` library**: `pip install formulas --break-system-packages`. Note: often fails to
produce openpyxl-readable cached values. **Do not rely on this as primary approach.**

XML Patching Workflow

```bash
# After saving workbook with openpyxl:
python3 scripts/patch_cached_values.py /root/output/result.xlsx
```

See `scripts/patch_cached_values.py` for the implementation. Key points:
- Compute each formula's expected value in Python
- Patch the `<v>` element inside each `<c>` cell in the worksheet XML
- Handle both numeric values (`t` attribute absent or `"n"`) and string values (`t="str"`)
- Do all patching in **one pass** to avoid corrupting the ZIP structure
- Patch pre-existing formulas too (e.g., `=B12` reference formulas)

Workflow

1. **Read test file** to understand verification method
2. **Examine workbook structure** --- both sheets, all rows/columns, identify:
- Yellow/highlighted cells (targets) --- check fill colors
   - Header rows with labels/years
   - Source data ranges
```

```
   - Series code columns for lookups
   - Pre-existing formulas (e.g., `=B12` in label cells)
3. **Map the layout precisely** --- off-by-one row/column errors are common
4. **Write formulas** with correct absolute/relative references:
- `$D12` --- absolute column, relative row
- `H$10` --- relative column, absolute row
- `$H$21:$L$38` --- fully absolute range
5. **Save with openpyxl** (preserves formatting)
6. **Patch cached values** using XML-level approach (single pass)
7. **Verify output** --- run `scripts/verify_formulas.py` to confirm all formula cells have cached
values
8. **Run the test suite** before declaring done:
   ```bash
   cd /root && python -m pytest test_output.py -v 2>&1 | head -80
   ```

Common Formula Patterns

INDEX/MATCH Lookup
```python
formula =
f'=INDEX(Data!$H$21:$L$38,MATCH($D{row},Data!$D$21:$D$38,0),MATCH({col}$10,Data!$H$4:$L$4,0))'
```

Net Gap / Derived Percentage
```python
formula = f'=({col}{success_row}-{col}{fail_row})/{col}{capacity_row}*100'
```

Summary Statistics Block
```python
stat_formulas = {
    'MIN': f'=MIN({col}{start}:{col}{end})',
    'MAX': f'=MAX({col}{start}:{col}{end})',
    'MEDIAN': f'=MEDIAN({col}{start}:{col}{end})',
    'AVERAGE': f'=AVERAGE({col}{start}:{col}{end})',
    'P25': f'=PERCENTILE({col}{start}:{col}{end},0.25)',
    'P75': f'=PERCENTILE({col}{start}:{col}{end},0.75)',
}
```

Weighted Mean with SUMPRODUCT
```python
formula =
f'=SUMPRODUCT({col}{val_start}:{col}{val_end},{col}{wt_start}:{col}{wt_end})/SUM({col}{wt_start}:
{col}{wt_end})'
```

Anti-Patterns

- **Never skip reading the test file.** The verifier determines whether you need formulas, values,
or both.
- **Never assume formulas will be auto-evaluated.** openpyxl does not compute; the verifier may
expect numeric results.
- **Never declare success without running `pytest`.** Self-verification with Python math is
necessary but not sufficient.
- **Don't hardcode data values in formulas.** Use cell references so the workbook remains dynamic.
- **Don't modify cells outside the target range.** Preserve all existing formatting, values, and
structure.
- **Don't patch XML multiple times.** Each patch reopens and rewrites the ZIP; multiple passes can
corrupt formula cells by converting them to plain values. Do all patching in one pass.
- **Don't rely on `formulas` library for caching.** It often doesn't produce values readable by
`data_only=True`.
- **When no test file exists, still cache values.** The verifier likely uses `data_only=True`.

Preserving Formatting

```python
# openpyxl preserves formatting when you only set .value
wb = openpyxl.load_workbook('input.xlsx')  # NOT data_only=True for writing
ws = wb['Task']
ws['H12'] = '=INDEX(...)'  # Preserves existing fill, font, borders
wb.save('output.xlsx')
```

Troubleshooting

Symptom	Likely Cause	Fix
All formula cells show None in test	Verifier uses `data_only=True`	Patch cached values via
XML
#REF! errors	Wrong range boundaries	Double-check row/column numbers in Data sheet
#N/A from MATCH	Series code mismatch	Print actual D-column values and compare
Wrong values	Off-by-one in row mapping	Map rows carefully to source
Test expects different stat order	MIN/MAX/MEDIAN row assignment wrong	Read row labels from
```

```
the workbook, don't assume |
| Formulas disappeared after XML patch | Multiple XML patches corrupted cells | Redo from openpyxl
save, patch once |
| `formulas` library doesn't cache | Known limitation | Use XML patching instead |
| LibreOffice not available | Not installed in container | Use XML patching instead |
| `KeyError: sharedStrings.xml` | File doesn't exist in this XLSX | List ZIP contents first,
handle missing files |
```

upsert_files["excel-formula-tasks/scripts/patch_cached_values.py"]:

```
#!/usr/bin/env python3
"""Patch cached values into Excel formula cells at the XML level.

Usage: python patch_cached_values.py <xlsx_path> [--data-sheet DATA_SHEET]

This script:
1. Reads the workbook to find all formula cells
2. Computes expected values using Python (simulating INDEX/MATCH, arithmetic, stats)
3. Patches the <v> elements in the worksheet XML so data_only=True returns values

IMPORTANT: Run this ONCE after saving with openpyxl. Multiple runs may corrupt formulas.
"""
import sys
import re
import zipfile
import xml.etree.ElementTree as ET
import shutil
import os
import openpyxl
import statistics

NS = 'http://schemas.openxmlformats.org/spreadsheetml/2006/main'


def col_letter_to_index(letter):
    """Convert column letter(s) to 0-based index."""
    result = 0
    for c in letter.upper():
        result = result * 26 + (ord(c) - ord('A') + 1)
    return result - 1


def parse_cell_ref(ref):
    """Parse 'H12' into (col_letter, row_number)."""
    m = re.match(r'([A-Z]+)(\d+)', ref)
    if m:
        return m.group(1), int(m.group(2))
    return None, None


def get_cached_value(ws_data, formula, ws_task, cell_ref):
    """Attempt to compute the expected value of a formula.

    This handles common patterns:
    - Simple cell references: =B12, =C12
    - INDEX/MATCH lookups
    - Arithmetic: =(A1-B1)/C1*100
    - Statistics: MIN, MAX, MEDIAN, AVERAGE, PERCENTILE
    - SUMPRODUCT/SUM

    Returns (value, is_string) tuple.
    For complex or unrecognized formulas, returns (None, False).
    """
    if not formula or not formula.startswith('='):
        return None, False

    f = formula[1:]  # strip leading =

    # Simple same-sheet cell reference: =B12
    simple_ref = re.match(r'^([A-Z]+)(\d+)$', f)
    if simple_ref:
        col, row = simple_ref.group(1), int(simple_ref.group(2))
        val = ws_task[f'{col}{row}'].value
        is_str = isinstance(val, str) and not val.startswith('=')
        return val, is_str

    return None, False


def patch_xlsx_cached_values(xlsx_path, computed_values):
    """Patch cached values into XLSX XML.

    computed_values: dict of {cell_ref: (value, is_string)}
    where is_string indicates if the value should be stored as type="str"
    """
```

```
    tmp_path = xlsx_path + '.tmp'

    with zipfile.ZipFile(xlsx_path, 'r') as zin:
        # Read sheet1 XML
        sheet_xml = zin.read('xl/worksheets/sheet1.xml')

        ET.register_namespace('', NS)
        # Preserve all other namespaces
        namespaces = {}
        for event, elem in ET.iterparse(xlsx_path.replace('.xlsx', '_ns_dummy.xml') if False else
                                        __import__('io').BytesIO(sheet_xml), events=['start-ns']):
            prefix, uri = elem
            if prefix:
                ET.register_namespace(prefix, uri)
                namespaces[prefix] = uri

        tree = ET.parse(__import__('io').BytesIO(sheet_xml))
        root = tree.getroot()

        for row_elem in root.iter(f'{{{NS}}}row'):
            for cell_elem in row_elem.iter(f'{{{NS}}}c'):
                ref = cell_elem.get('r')
                if ref in computed_values:
                    value, is_string = computed_values[ref]
                    if value is None:
                        continue

                    # Find or create <v> element
                    v_elem = cell_elem.find(f'{{{NS}}}v')
                    if v_elem is None:
                        # Insert <v> after <f> if present
                        f_elem = cell_elem.find(f'{{{NS}}}f')
                        v_elem = ET.SubElement(cell_elem, f'{{{NS}}}v')

                    if is_string:
                        cell_elem.set('t', 'str')
                        v_elem.text = str(value)
                    else:
                        # Remove string type if present
                        if cell_elem.get('t') == 's':
                            cell_elem.set('t', 'n')
                        v_elem.text = str(value)

        # Write modified XLSX
        modified_xml = ET.tostring(root, xml_declaration=True, encoding='UTF-8')

        with zipfile.ZipFile(tmp_path, 'w', zipfile.ZIP_DEFLATED) as zout:
            for item in zin.infolist():
                if item.filename == 'xl/worksheets/sheet1.xml':
                    zout.writestr(item, modified_xml)
                else:
                    zout.writestr(item, zin.read(item.filename))

    shutil.move(tmp_path, xlsx_path)


def main():
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} <xlsx_path>")
        sys.exit(1)

    xlsx_path = sys.argv[1]

    # Load workbook to read formulas and compute values
    wb = openpyxl.load_workbook(xlsx_path)
    ws = wb.active or wb.worksheets[0]

    computed = {}
    for row in ws.iter_rows():
        for cell in row:
            if isinstance(cell.value, str) and cell.value.startswith('='):
                val, is_str = get_cached_value(None, cell.value, ws, cell.coordinate)
                if val is not None:
                    computed[cell.coordinate] = (val, is_str)

    if computed:
        patch_xlsx_cached_values(xlsx_path, computed)
        print(f"Patched {len(computed)} cached values")
    else:
        print("No simple formulas to patch. Use task-specific computation logic.")
        print("See SKILL.md for the XML patching pattern to adapt.")


if __name__ == '__main__':
    main()
```

upsert_files["excel-formula-tasks/references/xml-caching-pattern.md"]:

```
# XML-Level Cached Value Patching Pattern

When openpyxl writes formulas, the resulting XLSX has `<c>` elements with `<f>` (formula) but no
`<v>` (cached value). Verifiers using `data_only=True` see `None`.

## The Pattern

```python
import zipfile
import xml.etree.ElementTree as ET
import shutil
import io

NS = 'http://schemas.openxmlformats.org/spreadsheetml/2006/main'

def patch_cached_values(xlsx_path, cell_values):
    """
    cell_values: dict mapping cell ref (e.g., 'H12') to (value, is_string)
    """
    tmp_path = xlsx_path + '.tmp'

    with zipfile.ZipFile(xlsx_path, 'r') as zin:
        sheet_xml = zin.read('xl/worksheets/sheet1.xml')
        ET.register_namespace('', NS)

        # Register all namespaces from the XML
        for event, elem in ET.iterparse(io.BytesIO(sheet_xml), events=['start-ns']):
            prefix, uri = elem
            if prefix:
                ET.register_namespace(prefix, uri)

        tree = ET.parse(io.BytesIO(sheet_xml))
        root = tree.getroot()

        for row_elem in root.iter(f'{{{NS}}}row'):
            for cell_elem in row_elem.iter(f'{{{NS}}}c'):
                ref = cell_elem.get('r')
                if ref in cell_values:
                    value, is_string = cell_values[ref]
                    if value is None:
                        continue

                    v_elem = cell_elem.find(f'{{{NS}}}v')
                    if v_elem is None:
                        v_elem = ET.SubElement(cell_elem, f'{{{NS}}}v')

                    if is_string:
                        cell_elem.set('t', 'str')
                        v_elem.text = str(value)
                    else:
                        if cell_elem.get('t') in ('s',):
                            cell_elem.set('t', 'n')
                        v_elem.text = str(value)

        modified_xml = ET.tostring(root, xml_declaration=True, encoding='UTF-8')

        with zipfile.ZipFile(tmp_path, 'w', zipfile.ZIP_DEFLATED) as zout:
            for item in zin.infolist():
                if item.filename == 'xl/worksheets/sheet1.xml':
                    zout.writestr(item, modified_xml)
                else:
                    zout.writestr(item, zin.read(item.filename))

    shutil.move(tmp_path, xlsx_path)
```

Critical Rules

1. **Single pass only.** Each patch reopens the ZIP. Multiple patches can corrupt formula cells by
losing `<f>` elements.
2. **Compute ALL values in Python first**, then patch all at once.
3. **Handle string vs numeric types.** String formula results (e.g., `=B12` referencing text) need
`t="str"`. Numeric results should not have `t="s"`.
4. **List ZIP contents first** if you need `sharedStrings.xml` --- it may not exist.
5. **Sheet filename:** Usually `xl/worksheets/sheet1.xml` for the first sheet. Check the
workbook's `xl/_rels/workbook.xml.rels` if unsure.

Computing Formula Values in Python

For INDEX/MATCH lookups, build a lookup dict from the Data sheet:

```python
# Build lookup: (series_code, year) -> value
```
```

1039

```
lookup = {}
for row in range(data_start, data_end + 1):
    code = ws_data[f'D{row}'].value
    for col in ['H', 'I', 'J', 'K', 'L']:
        year = ws_data[f'{col}{header_row}'].value
        lookup[(code, year)] = ws_data[f'{col}{row}'].value
```

For statistics, use Python's `statistics` module or numpy:

```python
import statistics
vals = [computed[f'{col}{r}'] for r in range(start, end+1)]
results = {
    'MIN': min(vals),
    'MAX': max(vals),
    'MEDIAN': statistics.median(vals),
    'AVERAGE': statistics.mean(vals),
}
# PERCENTILE requires numpy or manual interpolation
import numpy as np
results['P25'] = float(np.percentile(vals, 25))
results['P75'] = float(np.percentile(vals, 75))
```

For SUMPRODUCT:
```python
result = sum(a*b for a,b in zip(values, weights)) / sum(weights)
```

delete_paths:

```
[]
```

D.2 Weighted Hospital Bedflow Trajectory

User

[1] You need to update `/root/data/workbook.xlsx`. The workbook already contains the sheets `Task` and `Data`; work only inside these existing sheets.
Step 1: Populate the yellow cells in `H12:L17`, `H19:L24`, and `H26:L31` on sheet `Task` with spreadsheet formulas. Use two inputs for every lookup: the series code in column `D` of the current row and the year in row `10`. The source records are on sheet `Data` in rows `21:38`. Use one of these lookup patterns: `VLOOKUP` with `MATCH`, `HLOOKUP` with `MATCH`, `XLOOKUP` with `MATCH`, or `INDEX` with `MATCH`.
Step 2: In `H35:L40`, calculate `Net patient flow` for the six hospitals with this formula: `(Patient Admissions - Patient Discharges) / Effective Bed Capacity * 100`
Then calculate the column-wise minimum, maximum, median, simple mean, 25th percentile, and 75th percentile in `H42:L47`.
Step 3: In `H50:L50`, calculate the weighted mean for `Metro Hospital Network (MHN)` with `SUMPRODUCT`, using the Step 2 percentages as values and the `Effective Bed Capacity` block in `H26:L31` as weights.
Keep the existing workbook formatting unchanged. Do not add sheets, macros, VBA, external links, or helper tabs. Save the finished workbook to `/root/output/result.xlsx`.

Assistant

[2] *Reasoning:* Let me start by reading the Excel workbook to understand its structure before writing any formulas.

User

[3] *Skill reminder injected by environment.* Base directory for this skill: `/etc/claude-code/.claude/skills/excel-formula-tasks`.
The reminder emphasizes two things: read the verifier/test file first, and be careful that `openpyxl` writes formulas but does not evaluate them, so a verifier using `data_only=True` may see `None` unless cached values are present.

Assistant

[4] *Tool Calls (1):*
```
Skill("excel-formula-tasks", args=<full task instruction>)
```

Tool Call/Response

[5] *Tool Response:*
```
Launching skill: excel-formula-tasks
[metadata] {"success": true, "commandName": "excel-formula-tasks"}
```

Assistant

[6] *Reasoning:* Let me start by examining the workbook structure and any test files to understand what's expected.

Assistant

[7] *Tool Calls (1):*
```
TodoWrite(
Read test file and examine workbook structure [in progress],
Write lookup formulas in H12:L17, H19:L24, H26:L31 [pending],
Write Net patient flow formulas in H35:L40 and stats in H42:L47 [pending],
Write weighted mean formulas in H50:L50 [pending],
Run tests and verify output [pending]
```

```
)
```

Tool Call/Response

[8] *Tool Response:*
`Todos have been modified successfully.`

Assistant

[9] *Tool Calls (2):*
`Bash: ls -la /root/ /root/data/ /root/output/`
`Bash: find /root -name '*.py' -o -name '*.xlsx' | head -30`

Tool Call/Response

[10] *Tool Response Summary:*
The agent confirms the presence of:

- the input workbook at `/root/data/workbook.xlsx`,
- an empty output directory `/root/output/`,
- and a verifier script at `/root/.claude/skills/excel-formula-tasks/scripts/verify_formulas.py`.

Assistant

[11] *Tool Calls (1):*
`Read(/root/.claude/skills/excel-formula-tasks/scripts/verify_formulas.py)`

Tool Call/Response

[12] *Tool Response Summary:*
The verifier loads the workbook twice:

- once normally to inspect formulas,
- once with `data_only=True` to inspect cached values.

It warns whenever a formula cell has no cached value. This confirms that the agent must preserve both formulas and cached values in the final workbook.

Assistant

[13] Now let me examine the workbook structure in detail.

Assistant

[14] *Tool Calls (2):*
`Bash: inspect Task sheet`
`Bash: inspect Data sheet`

Tool Call/Response

[15] *Tool Response Summary:*
On sheet `Task`, the agent identifies:

- lookup targets in `H12:L17`, `H19:L24`, and `H26:L31`,
- derived metric cells in `H35:L40`,
- summary statistic cells in `H42:L47`,

- weighted mean cells in `H50:L50`,
- and year headers in `H10:L10` corresponding to 2020–2024.

On sheet `Data`, rows `21:38` contain the source records, while row `4` contains the year headers.

Assistant

[16] Now I have a clear picture. Let me also check if there's a test file.

Assistant

[17] *Tool Calls (1):*

```
Bash: find /root -name 'test_output*' -o -name 'test_*.py'
```

Tool Call/Response

[18] *Tool Response Summary:*
No task-specific `test_output.py` file is found. The agent therefore proceeds using the verifier script plus direct workbook inspection.

Assistant

[19] *Tool Calls (1):*

```
Bash: write formulas into workbook and save /root/output/result.xlsx
```

Tool Call/Response

[20] *Tool Response Summary:*
The agent writes formulas as follows:

- **Step 1 lookup:**

```
=INDEX(Data!$H$21:$L$38,
       MATCH($Drow,Data!$D$21:$D$38,0),
       MATCH(H$10,Data!$H$4:$L$4,0))
```

- **Step 2 net patient flow:**

```
=(H12-H19)/H26*100
```

- **Step 2 statistics:** `MIN`, `MAX`, `MEDIAN`, `AVERAGE`, and `PERCENTILE(...,0.25/0.75)`
- **Step 3 weighted mean:**

```
=SUMPRODUCT(H35:H40,H26:H31)/SUM(H26:H31)
```

The workbook is saved to `/root/output/result.xlsx`.

Assistant

[21] *Tool Calls (1):*

```
TodoWrite(
Read test file and examine workbook structure [completed],
Write all formulas and save workbook [completed],
Run verification and validate output [in progress]
)
```

Tool Call/Response

[22] *Tool Response:*
`Todos have been modified successfully.`

Assistant

[23] Now let me verify the output thoroughly.

Assistant

[24] *Tool Calls (1):*
`Bash: open /root/output/result.xlsx and inspect formulas/values`

Tool Call/Response

[25] *Tool Response Summary:*
The agent checks formula placement and sees that formulas exist in the expected regions. It then continues with deeper validation of cached values and workbook internals.

Assistant

[26] *Tool Calls (several):*
```
Bash: inspect XLSX archive internals
Bash: list XML files inside result.xlsx
Bash: patch workbook XML to preserve cached values
Bash: re-open workbook and inspect selected cells
```

Tool Call/Response

[27] *Tool Response Summary:*
During this stage, the agent discovers that the workbook has inline strings and no `sharedStrings.xml`. It attempts XML-level patching to keep cached values while preserving formulas, but the intermediate result is fragile.

Assistant

[28] No shared strings file. The strings are inline. Let me fix the `B35:C40` cells with a simpler approach.

Assistant

[29] *Tool Calls (1):*
`Bash: patch affected cells and save workbook again`

Tool Call/Response

[30] *Tool Response Summary:*
The agent repairs display cells such as `B35:C40` and continues checking whether formulas and cached values both survive the patching procedure.

Assistant

[31] *Tool Calls (1):*
`Bash: perform more detailed verification on Task sheet formula cells`

Tool Call/Response

[32] *Tool Response Summary:*
The verification reveals a problem: some Step 2 and Step 3 cells are no longer stored as formulas after XML patching. The agent recognizes that the patching process has partially corrupted the intended structure.

Assistant

[33] It seems the XML patching converted the Step 2/3 formula cells to just cached values (the formulas got dropped). The issue is that the XML patching process may have corrupted the formula cells. Let me redo the entire process more carefully.

Assistant

[34] *Tool Calls (1):*
`Bash: restart from original workbook and rebuild everything from scratch`

Tool Call/Response

[35] *Tool Response Summary:*
The agent reloads `/root/data/workbook.xlsx` and reconstructs the full solution pipeline:

- build a lookup dictionary from `Data!D21:L38`,
- rewrite all Step 1 formulas with `INDEX+MATCH`,
- rewrite all Step 2 formulas for net patient flow,
- rewrite all Step 2 statistics formulas,
- rewrite all Step 3 weighted mean formulas,
- then save a fresh `/root/output/result.xlsx`.

Assistant

[36] *Rebuilt Step 1 formula template:*

```
=INDEX(Data!$H$21:$L$38,
MATCH($D12,Data!$D$21:$D$38,0),
MATCH(H$10,Data!$H$4:$L$4,0))
```

Rebuilt Step 2 formula template:

```
=(H12-H19)/H26*100
```

Rebuilt Step 3 formula template:

```
=SUMPRODUCT(H35:H40,H26:H31)/SUM(H26:H31)
```

Assistant

[37] *Tool Calls (1):*
`Bash: compute expected cached values in Python and perform a single XML patch pass`

Tool Call/Response

[38] *Tool Response Summary:*
The agent computes formula results directly in Python, including lookup results, net patient flow percentages, summary statistics, and the weighted mean. It then performs one careful XML patch pass so the workbook keeps both formula expressions and cached values.

Assistant

[39] Let me inspect the saved workbook one more time.

Assistant

[40] *Tool Calls (1):*
`Bash: verify formulas and cached values in the rebuilt workbook`

Tool Call/Response

[41] *Tool Response Summary:*
The rebuilt workbook now preserves both formula text and cached results. Sample checks confirm that cells such as `H12`, `H35`, and `H50` have formulas when loaded normally and numeric cached values when loaded with `data_only=True`.

Assistant

[42] *Tool Calls (1):*
`Bash: run verifier script on /root/output/result.xlsx`

Tool Call/Response

[43] *Tool Response Summary:*
The verifier reports success: formula cells are present, cached values are available, and no missing-value warnings remain.

Assistant

[44] Great — the workbook now has both formulas and cached values, and the verification passed.

Assistant

[45] *Tool Calls (1):*

```
TodoWrite(
Read test file and examine workbook structure [completed],
Write all formulas and save workbook [completed],
Run verification and validate output [completed]
)
```

Tool Call/Response

[46] *Tool Response:*
`Todos have been modified successfully.`

Assistant

[47] I have updated the workbook and verified that the result preserves both formulas and cached values. The final file is saved at `/root/output/result.xlsx`.

D.3 FAILURE TAXONOMY EXAMPLES

D.3.1 Verifier Toolchain Mismatch

Failure Mode: Verifier / toolchain incompatibility
Task: `api-sla-at-risk-calc`
Other Failure Modes: Missing recalculation and No or Incomplete verification.
Where it went wrong:

```
[Verifier]
subprocess.CalledProcessError: Command
['ssconvert', '/root/output/result.xlsx',
 '/tmp/.../recalculated.xlsx', '--recalculate']
returned non-zero exit status 1.
stderr = "Unknown option --recalculate"

[Agent]
Workbook saved successfully.
Row 50: =SUMPRODUCT(H35:H40,H26:H31)/SUM(H26:H31)
```

D.3.2 Missing Cached Values

Failure Mode: Missing recalculation / missing cached formula values
Task: `weighted-hospital-bedflow-calc`
Other Failure Modes: No or Incomplete verification and Premature success declaration.
Where it went wrong:

```
[Agent]
Step 1: Lookup formulas written
Step 2a: Net patient flow formulas written
Step 2b: Statistics formulas written
Step 3: Weighted mean formulas written
Workbook saved to /root/output/result.xlsx

[Verifier]
H12: expected 612.5, got None
...
H35: expected 4.499999999999997, got None
```

D.3.3 Incomplete Verification

Failure Mode: No or Incomplete verification
Task: `weighted-port-throughput-calc`
Other Failure Modes: Verifier / toolchain incompatibility and Missing recalculation.
Where it went wrong:

```
[Agent]
H35 (ALP 2019): (302.5 - 395.4) / 502.0 * 100 = -18.5060
H36 (BET 2019): (111.9 - 169.1) / 203.6 * 100 = -28.0943
H38 (DEL 2019): (413.2 - 284.6) / 568.7 * 100 = 22.6130
All formulas reference correct rows and will compute correctly in Excel.

[Verifier]
Unknown option --recalculate
Command '['ssconvert', '/root/output/result.xlsx', ...,
'--recalculate']' returned non-zero exit status 1.
```