

EvoSkill: Automated Skill Discovery for Multi-Agent Systems

Salaheddin Alzubi¹, Noah Provenzano², Jaydon Bingham²,
Christian Calvo², Weiyuan Chen², Tu Vu²

¹Sentient Labs ²Virginia Tech

salaheddin@sentient.xyz, {noahpro, jaydon, kuzoto, cweiyuan9, tuvuvu}@vt.edu

 <https://github.com/sentient-agi/EvoSkill>

Abstract

Coding agents are increasingly used as general-purpose problem solvers, but they often lack the domain expertise required for specialized tasks. *Agent skills* address this limitation through reusable modules that package instructions, scripts, and resources to augment agents with specialized capabilities. However, these skills are typically hand-crafted, which limits their scalability. *We ask whether agents can instead acquire such skills automatically from experience.* We introduce **EvoSkill**, a framework for automated skill acquisition. EvoSkill iteratively analyzes execution failures to identify capability gaps, proposes new skills or refinements to existing ones, and materializes them as structured skill folders. Candidate skills are selected based on held-out validation performance, while model parameters remain fixed. We evaluate EvoSkill on diverse benchmarks spanning financial-document reasoning, web search, multi-source reasoning, and coding (OfficeQA, SealQA, FRAMES, and LiveCodeBench), using models from the Claude, Gemini, Kimi, and gpt-oss families as executors. We show that EvoSkill can substantially improve agent performance across benchmarks, including gains of 15.7 points on SealQA and 11.4 points on OfficeQA. In a controlled comparison, skill evolution with EvoSkill substantially outperforms prompt evolution with GEPA. We further find evidence that evolved skills can transfer across settings: a search skill evolved on SealQA improves BrowseComp accuracy by 5.3 points without modification, while reuse across executor models improves performance in several model pairings, with outcomes depending on model and harness compatibility.

1 Introduction

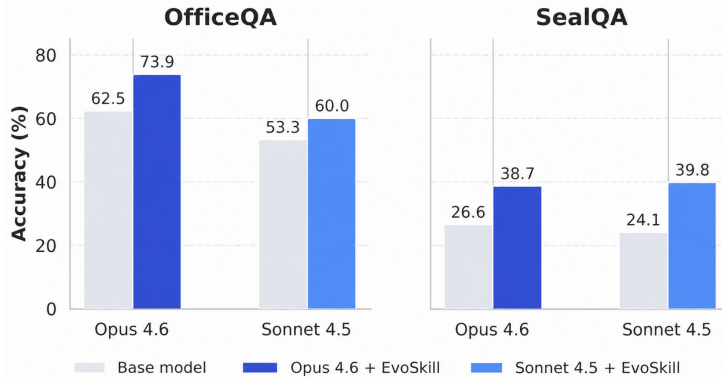


Figure 1: **EvoSkill substantially improves accuracy on OfficeQA and SealQA.** Skills are evolved by Claude Opus 4.6 and executed by the indicated model.

Coding agents such as Claude Code (Anthropic, 2025), Codex (OpenAI, 2025), OpenHands (Hevesi, 2024), and OpenCode (Anomaly, 2026) are increasingly used as general-purpose problem solvers across diverse domains. Their ability to write and execute code and interact with external tools allows them to tackle complex tasks within a common agent framework. However, general-purpose capabilities alone do not provide the specialized knowledge and workflows required to perform domain-specific tasks reliably.

Agent skills (Zhang et al., 2025) address this limitation through reusable modules (e.g., instructions, scripts, and resources) that agents can dynamically load to augment their capabilities for specialized tasks. However, agent skills are typically hand-crafted, which requires domain expertise and substantial manual effort, thus limiting their scalability as the number of target tasks grows. In this paper, we ask whether agents can instead acquire such skills automatically from experience. Existing evolutionary methods, such as AlphaEvolve (Novikov et al., 2025) and GEPA (Agrawal et al., 2026), provide a natural starting point. These methods iteratively improve agent artifacts such as prompts or code based on execution feedback. We explore whether this paradigm can instead be applied to the acquisition and refinement of reusable agent skills. This shift is particularly appealing: skills offer a modular abstraction that can package specialized knowledge and workflows and support selective loading and reuse across tasks and agents.

To this end, we introduce **EvoSkill**, a framework that iteratively acquires and refines agent skills from experience (Figure 2). At a high level, EvoSkill analyzes execution failures to identify capability gaps, then proposes new skills or refinements to existing ones that address these gaps. Each candidate skill is materialized as a structured folder that contains instructions and metadata, along with optional resources (e.g., scripts and templates). All candidate skills are evaluated on a held-out validation set, with successful ones retained for subsequent iterations. As this process repeats, the agent accumulates an increasingly effective set of skills, which expands its capabilities without modifying model parameters.

We evaluate EvoSkill across diverse benchmarks: OfficeQA (Singhvi et al., 2025), SealQA (Pham et al., 2026), FRAMES (Krishna et al., 2025), and LiveCodeBench (Jain et al., 2025), which span financial-document reasoning, web search, multi-source reasoning, and coding, respectively. We use models from the Claude, Gemini, Kimi, and gpt-oss families as executors. We show that EvoSkill can yield substantial performance gains, including improvements of 16.4 points with Kimi K2.5 on FRAMES, 15.7 points with Claude Sonnet 4.5 on SealQA, and 11.4 points with Claude Opus 4.6 on OfficeQA. Notably, EvoSkill can discover useful skills from very little data (12 examples on OfficeQA yield a 2.8-point gain), while combining skills from independent runs yields further gains (7.5 points), which suggests that different runs can discover complementary capabilities. In a controlled comparison, we find that evolving skills with EvoSkill can be substantially more effective than optimizing prompts with GEPA. To study whether evolved skills can be reused beyond the settings in which they are acquired, we further explore reuse across datasets and executor models. We find evidence of zero-shot cross-dataset transfer: a search skill evolved on SealQA improves BrowseComp (Wei et al., 2025) accuracy by 5.3 points without modification. We also find that skills evolved from one executor’s experience can benefit other executors in several model pairings, with outcomes depending on model and harness compatibility.

Our contributions are as follows:

1. We introduce EvoSkill, a framework that automatically acquires and refines structured agent skills from execution experience without modifying model parameters.
2. We show that evolving skills can substantially improve agent performance across diverse tasks and executor models, and that skill evolution can be more effective than prompt optimization in a controlled comparison.
3. We study the reuse of evolved skills, and show evidence of complementary skills across independent runs and reuse across datasets and executor models.

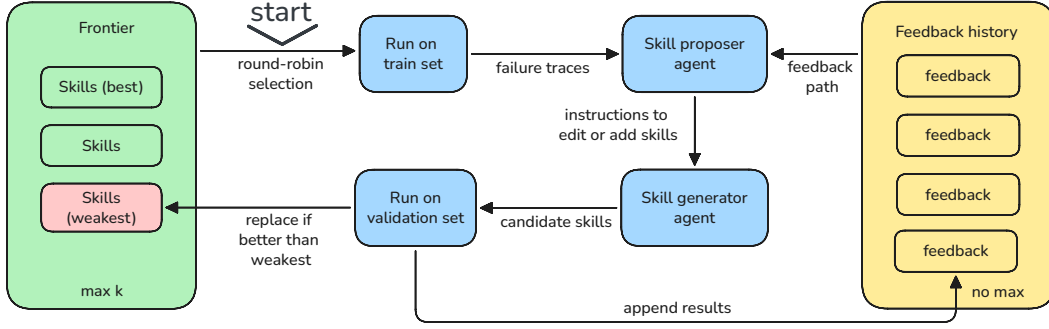


Figure 2: **Overview of the EvoSkill loop.** EvoSkill analyzes training failures to generate candidate skills, evaluates them on held-out data, and retains successful candidates for subsequent iterations.

2 Methodology

EvoSkill iteratively acquires and refines agent skills from execution experience using textual feedback descent (Lee et al., 2025). Given training examples where the current agent fails, EvoSkill analyzes the failures, proposes new skills or refinements to existing ones, and evaluates the resulting candidates on held-out validation data. We assume a coding agent harness that supports skill folders, such as Claude Code, Codex, OpenHands, or OpenCode. Model parameters remain fixed throughout the process.

2.1 Framework Overview

EvoSkill uses three agents with distinct roles (see Figure 2):

1. **Executor Agent (A):** Executes training examples using the current set of skills, starting with an empty skills directory.
2. **Skill Proposer Agent (P):** Analyzes the Executor’s traces, predicted answers, and ground-truth answers to diagnose failures and identify capability gaps. Based on this analysis, the Proposer recommends either a new skill or a refinement to an existing one. Ground-truth answers are used only for failure diagnosis and are not included in the generated skills.
3. **Skill Generator Agent (S):** Converts the Proposer’s recommendation into one or more new or refined skills in the skills directory. Skills can contain three types of content that become available to the agent at different stages: metadata for discovery, instructions that are loaded when a skill is triggered, and optional resources and code that are accessed as needed. The Skill Generator is provided with a fixed meta-skill that contains guidelines for creating well-structured skills.

All three agents have read access to the base agent repository, while only the Skill Generator can modify the skills directory. The Skill Proposer also has access to a cumulative *feedback history*, which records previous proposals, their validation scores, and selection outcomes. This history helps the Proposer avoid redundant proposals, learn from previous mistakes, and refine approaches that showed promise.

2.2 EvoSkill Loop

We define an agent program p as a fixed system prompt paired with its current set of skills. EvoSkill maintains a *frontier of agent programs* G with capacity k , which is initialized with the base program that contains no skills. The frontier evolves through the iterative process shown in Figure 2 and Algorithm 1 in Appendix B.

At each iteration t , EvoSkill selects a *parent program* p from G in round-robin order, which ensures that each frontier member is explored before any is revisited. The Executor A evaluates p on a batch of training examples. Training examples cycle without replacement

until all have been used, after which the cycle restarts. Responses are scored against ground-truth answers using a task-specific scoring function, and examples that score below a fixed threshold form the *failure set* F . If F is empty, EvoSkill proceeds to the next iteration.

The Proposer P uses the failures in F and the feedback history H to produce a *textual proposal* π that specifies a new skill or a refinement to an existing one. The Skill Generator S uses π to modify the skills in the parent program p , which produces a *candidate program* $\tilde{p}$.

EvoSkill evaluates the candidate $\tilde{p}$ on the held-out *validation set* V using the same task-specific scoring function. If its validation score exceeds that of the weakest program in G , the candidate replaces that program; otherwise, it is discarded. Regardless of the outcome, EvoSkill appends the proposal, validation score, and selection outcome to H , which provides feedback for subsequent iterations.

After T iterations, EvoSkill returns the highest-scoring program in G .

2.3 Implementation Details

2.3.1 Data Setup

Given a benchmark dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ and a task-specific scoring function, we first use an LLM to assign each example to one of K categories. These categories help preserve coverage of different example types when constructing the data splits and sampling examples during evolution. Depending on the experiment, we use either the full benchmark or a category-stratified sample.

We then partition the data into three disjoint subsets: a *training set* used for failure detection during evolution, a *validation set* used to score candidate programs for frontier selection, and a *held-out test set* used only for final evaluation. For benchmarks without dedicated training and validation splits, we construct the training and validation sets from the benchmark evaluation set and reserve the remaining examples as the held-out test set.

Training examples are grouped by category to support category-aware sampling during evolution. Appendix D provides additional details on dataset construction and splitting.

2.3.2 Environment Setup

EvoSkill operates within a Git repository with a fixed codebase and system prompt. Each agent program is represented by a separate branch that inherits from its parent and differs only in its skills. EvoSkill also stores metadata for each program, including its parent program and validation score. This setup keeps skill changes separate across programs while keeping branches lightweight. Appendix E provides additional details on the repository configuration.

3 Experiments

We evaluate EvoSkill along three dimensions. First, we test whether evolved skills improve agent performance across tasks and executor models. Second, we study properties of the skill evolution process, including how much evolution data is needed, whether skills discovered across independent runs can be combined, and how skill evolution compares with prompt evolution. Finally, we explore whether evolved skills can be reused across datasets and executor models.

3.1 Experimental Setup

Benchmarks. We evaluate EvoSkill across diverse benchmarks spanning financial-document reasoning, web search, multi-source reasoning, and coding:

OfficeQA (Singhvi et al., 2025) evaluates grounded reasoning over approximately 89,000 pages of U.S. Treasury Bulletins. Its 246 questions require agents to retrieve and aggregate

information from complex documents, including tables and figures, and perform analytical reasoning to produce precise answers.

SealQA (Pham et al., 2026) evaluates search-augmented reasoning on fact-seeking questions for which web search returns conflicting, noisy, or unhelpful information. We use its main Seal-0 split, which contains 111 questions.

FRAMES (Krishna et al., 2025) evaluates factuality, retrieval, and reasoning in end-to-end retrieval-augmented generation. Its questions require agents to retrieve and integrate information across multiple sources. We focus on questions that require at least four source articles, which comprise 27.4% of the benchmark.

LiveCodeBench (Jain et al., 2025) evaluates coding capabilities using problems collected from programming competitions. We use its code-generation setting, in which agents generate solutions that are evaluated against test cases.

BrowseComp (Wei et al., 2025) is used for the cross-dataset transfer experiment, where we test whether a search skill evolved on SealQA can be reused on a different web-browsing benchmark without modification.

Models and harnesses. We use Claude Opus 4.6, a frontier model with native support for the Agent Skills format, as the Skill Proposer and Skill Generator throughout our experiments. Across model comparisons, we keep the Skill Proposer and Skill Generator fixed and vary the Executor Agent. We consider executor models from the Claude, Gemini, Kimi, and gpt-oss families; when no Executor Agent is specified, we use Claude Opus 4.6. Claude models run with Claude Code, while other models run with OpenCode.

Evaluation setup. We construct training, validation, and held-out test sets following Section 2.3.1 and report final performance on the held-out test set. We run EvoSkill for at most 20 iterations and stop early after five consecutive iterations without improvement.

To manage computational cost, our experiments vary in scope according to their purpose. For broad comparisons across Executor models, we cap each benchmark at 120 examples and apply the category-stratified sampling procedure described in Section 2.3.1. Targeted analyses use configurations suited to their specific goals; for example, we use the full 246-example OfficeQA benchmark to study how performance changes with the amount of evolution data and whether skills from independent runs can be combined. As a result, the benchmarks and executor models included differ across experiments.

3.2 Does EvoSkill Improve Agent Performance?

EvoSkill improves agent performance across a range of tasks and executor models. Figure 1 reports substantial improvements with both Claude executors on OfficeQA and SealQA. The gains are consistent across both benchmarks, including improvements of 15.7 points for Sonnet 4.5 on SealQA (24.1% $\rightarrow$ 39.8%) and 11.4 points for Opus 4.6 on OfficeQA (62.5% $\rightarrow$ 73.9%). Table 1 extends these results beyond Claude to Gemini, Kimi, and gpt-oss executors. EvoSkill improves performance across most of the evaluated settings, with substantial gains for several non-Claude executors. Most notably, Kimi K2.5 improves by 16.4 points on FRAMES (45.1% $\rightarrow$ 61.5%). The gains, however, are not uniform across tasks and executors. SealQA improves across every executor evaluated, whereas results on FRAMES and LiveCodeBench are more mixed, with some settings showing little or no improvement. Despite this variation, EvoSkill delivers broad performance gains, with particularly strong improvements in several settings.

3.3 How Much Data Does EvoSkill Need?

We study how the amount of evolution data affects EvoSkill on OfficeQA. From the full 246-example benchmark, we vary the training-set size from 5% (12 examples) to 10% (24

Table 1: **EvoSkill improves accuracy across diverse benchmarks and executor models.** Claude Opus 4.6 proposes and generates skills, while the indicated model executes them.

Model	SealQA	FRAMES	LiveCodeBench
Gemini 3.1 Flash-Lite	21.7	65.9	73.3
+ EvoSkill	22.9	68.1	68.4
Kimi K2.5	27.7	45.1	70.7
+ EvoSkill	32.5	61.5	70.7
gpt-oss-120b	23.3	60.4	45.7
+ EvoSkill	27.3	63.7	50.0
gpt-oss-20b	1.2	13.5	43.8
+ EvoSkill	2.4	13.5	50.0

Table 2: **EvoSkill improves OfficeQA accuracy with limited evolution data, while merging skills from independent runs provides further gains.** We compare Claude Opus 4.6 without skills against EvoSkill trained on 5%, 10%, or 15% of the benchmark and a merged skill library, with accuracy reported across relative-error tolerances.

Method	Tolerance				
	0%	0.1%	1%	5%	10%
Claude Opus 4.6	60.6	66.3	72.8	77.2	79.7
+ 5% train	63.4	67.4	74.3	77.6	80.1
+ 10% train	65.8	69.9	76.4	80.5	82.5
+ 15% train	64.5	69.9	75.6	79.3	81.3
+ merged skills	68.1	70.8	77.1	80.5	82.4

examples) and 15% (36 examples), while using 17 examples for validation and reserving the remaining examples for held-out evaluation. Since OfficeQA answers are predominantly numerical, we report accuracy at multiple error tolerances: 0% requires an exact match, while a tolerance of 1%, for example, accepts predictions within 1% of the ground-truth answer.

EvoSkill improves over the baseline even with limited evolution data. With only 12 training examples, exact-match accuracy increases from 60.6% to 63.4%, while 24 examples increase it to 65.8%. Increasing the training set to 36 examples provides no further improvement, with accuracy reaching 64.5%. The same general pattern holds across tolerance levels (see Table 2 and Figure 3), where the 10% and 15% settings remain close. These results suggest that EvoSkill can acquire useful skills from relatively few examples, with diminishing returns from additional evolution data in this setting.

3.4 Can Skills from Independent Runs Be Combined?

We next examine whether skills discovered in independent EvoSkill runs can be combined. We merge the skill libraries from the OfficeQA runs in Section 3.3, retaining unique skills and, when skills overlap based on their names or descriptions, keeping the version from the highest-performing run.

Combining skills from independent runs further improves performance. The merged skill library yields the strongest OfficeQA result, increasing exact-match accuracy from 60.6% to 68.1% (+7.5 points), compared with 65.8% for the best individual run. It also performs strongly across the other tolerance levels (see Table 2 and Figure 3). These results suggest

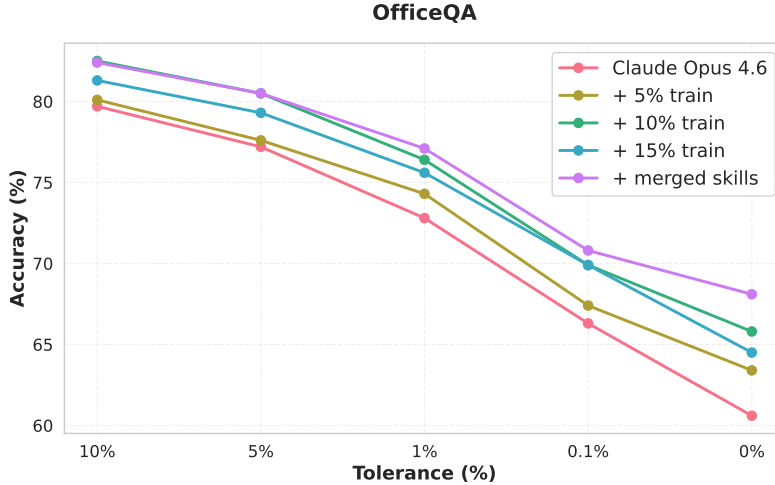


Figure 3: **EvoSkill improves OfficeQA accuracy with limited evolution data, while merging skills from independent runs provides further gains.** We compare Claude Opus 4.6 without skills against EvoSkill trained on 5%, 10%, or 15% of the benchmark and a merged skill library, with accuracy reported across relative-error tolerances.

Table 3: **EvoSkill substantially outperforms system prompt evolution.** Both methods use the same evolution procedure with Claude Opus 4.6 throughout, but evolve different artifacts: the system prompt or skill folders. Results report mean $\pm$ standard deviation over three runs.

Method	OfficeQA	SealQA
Claude Opus 4.6	61.5 $\pm$ 1.0	25.8 $\pm$ 0.8
+ System prompt evolution	62.7 $\pm$ 0.7	26.4 $\pm$ 1.4
+ EvoSkill	71.2 $\pm$ 2.5	38.9 $\pm$ 0.5

that independent runs can discover complementary skills whose benefits can be combined into a stronger skill library.

3.5 Are Skills More Effective Than Prompts?

Skill evolution substantially outperforms prompt evolution. We first compare EvoSkill with a prompt-only variant that uses the same evolution procedure but rewrites the system prompt instead of creating or refining skill folders (see Appendix E). We use Claude Opus 4.6 throughout this experiment. As shown in Table 3, prompt evolution yields only small gains on OfficeQA and SealQA, while EvoSkill improves accuracy from 61.5% to 71.2% and from 25.8% to 38.9%, respectively. This controlled comparison suggests that structured skills provide substantial benefits over free-form prompt evolution.

In a separate experiment, we compare EvoSkill with GEPA using gpt-oss-120b as the Executor Agent, with both methods run for up to 20 iterations. EvoSkill achieves the highest accuracy across all evaluated benchmarks (Table 4). On SealQA, EvoSkill reaches 27.3% compared with 21.8% for GEPA, while on LiveCodeBench, EvoSkill reaches 50.0% compared with 49.5% for GEPA. These results provide further evidence that skill evolution can be more effective than prompt evolution.

3.6 Do Evolved Skills Transfer Across Datasets?

Evolved skills can transfer to a new dataset without further evolution. We test cross-dataset transfer by applying a *search-persistence-protocol* skill evolved with Claude Opus 4.6 on SealQA to BrowseComp without modification. The transferred skill improves accuracy

Table 4: **EvoSkill consistently outperforms GEPA in our comparison.** We compare GEPA and EvoSkill using gpt-oss-120b as the Executor Agent, with both methods run for up to 20 iterations.

Method	SealQA	LiveCodeBench
gpt-oss-120b	23.3	45.7
+ GEPA	21.8	49.5
+ EvoSkill	27.3	50.0

Table 5: **Evolved skills can be reused across executor models, though their benefits vary across settings.** Rows indicate the Executor Agent whose SealQA experience drives skill evolution, while columns indicate the Executor Agent that uses the resulting skills without further evolution. Claude Opus 4.6 serves as the Skill Proposer and Skill Generator throughout.

Evolution executor ↓	Claude Sonnet 4.5	Kimi K2.5	gpt-oss-120b	gpt-oss-20b
No skills	24.1	27.7	22.9	1.2
Claude Opus 4.5	21.7	36.1	12.0	1.2
Claude Sonnet 4.5	39.8	34.9	30.1	2.4
Kimi K2.5	8.4	32.5	28.9	1.2
gpt-oss-120b	19.3	18.1	24.1	1.2

from 43.5% to 48.8%, a gain of 5.3 points. This result shows that a skill discovered on one benchmark can remain useful on a different benchmark that requires related search capabilities. We also manually inspect the evolved skill libraries and find that the skills encode procedural strategies rather than references to specific training examples, which further suggests that EvoSkill can discover reusable strategies rather than only benchmark-specific fixes.

3.7 Do Evolved Skills Transfer Across Executor Models?

Evolved skills can be reused across executor models, though their benefits vary across settings. We evolve skill libraries on SealQA using execution experience from one executor and then evaluate the resulting skills with other executors without further evolution. Claude Opus 4.6 serves as the Skill Proposer and Skill Generator throughout. As shown in Table 5, skills evolved from Claude Sonnet 4.5 experience are particularly reusable, improving performance with all evaluated executors. For example, they improve gpt-oss-120b from 22.9% to 30.1% and Kimi K2.5 from 27.7% to 34.9%. Skills evolved from Claude Opus 4.5 experience also benefit Kimi K2.5, increasing accuracy from 27.7% to 36.1%.

The benefits of reuse are not uniform. Skills evolved from Kimi K2.5 experience improve gpt-oss-120b but reduce performance with Claude Sonnet 4.5, while skills evolved from gpt-oss-120b experience provide limited benefits to other executors. These differences may reflect the strategies encoded in the skills, the capabilities of the executor that uses them, and compatibility with the underlying agent harness. The successful cases nevertheless show that evolved skills can remain useful to executors other than the one whose experience drove their evolution.

4 Related Work

4.1 Agent Skills

The idea of augmenting agents with reusable, modular capabilities has appeared in both embodied AI and software engineering. Voyager (Wang et al., 2024) introduced an ever-growing skill library of executable code for an LLM-powered Minecraft agent, where skills are stored as programs in a vector database and retrieved by semantic similarity. Skills in

Voyager are discovered through an automatic curriculum and iterative prompting with environment feedback, enabling lifelong learning without parameter updates.

More recently, Agent Skills (Zhang et al., 2025) has formalized skills as a portable, open format: each skill is a filesystem directory containing a `SKILL.md` file with metadata and procedural instructions, optionally bundled with helper scripts and reference materials. The format is supported across Claude products and third-party agent platforms, which enables skills to be reused across different agent environments (Zhang et al., 2025; Alzu'bi et al., 2026). Agent Skills use progressive disclosure: skill names and descriptions are loaded at startup, instructions are read when a skill is triggered, and additional resources and code are accessed as needed. Executable scripts can run without their code entering the context window, which allows agents to maintain many skills with limited context overhead. Despite this infrastructure, skills today are predominantly hand-authored. EvoSkill addresses this gap by automatically discovering and refining skills through iterative failure analysis while producing artifacts that conform to the same structured skill format.

4.2 Textual Feedback and Evolutionary Optimization

A growing body of work explores using natural-language feedback, rather than scalar rewards, to guide iterative improvement of LLM-generated artifacts. Self-Refine (Madaan et al., 2023) demonstrated that a single LLM can improve its own outputs through a generate-critique-refine loop, achieving consistent gains across diverse tasks without additional training. However, Self-Refine operates on individual outputs and does not accumulate knowledge across iterations.

Feedback Descent (Lee et al., 2025) formalizes this intuition into a general optimization framework, showing that rich textual feedback from evaluators can drive sustained improvement across domains including molecular design, SVG optimization, and prompt engineering. Feedback Descent maintains a frontier of top-performing candidates and accumulates feedback history, enabling an editor LLM to make increasingly informed revisions. EvoSkill builds directly on this paradigm, applying textual feedback descent to skill discovery and refinement.

On the evolutionary side, AlphaEvolve (Novikov et al., 2025) uses an ensemble of LLMs to evolve entire codebases through an evolutionary loop grounded by automatic evaluation, achieving breakthroughs in algorithm discovery and infrastructure optimization. GEPA (Agrawal et al., 2026) takes a similar evolutionary approach to prompt optimization within the DSPy framework, using reflective mutation and Pareto-based candidate selection to evolve textual components of complex systems. These approaches demonstrate how LLM-driven mutations can optimize code and prompts through execution feedback. Beyond optimizing individual artifacts, EvoTest (He et al., 2026) adapts an agent at test time by evolving its entire configuration (prompt, memory, hyperparameters, and tool-use routines) after each episode of a task; the resulting adaptation is bound to that agent instance rather than externalized as a portable, inspectable skill.

EvoSkill differs from these approaches in the artifact that it evolves. Instead of evolving prompts, code, or an entire agent configuration directly, EvoSkill evolves skills: structured, reusable capability modules that persist across tasks. Two concurrent efforts similarly treat skills as external state that can be optimized while the underlying model remains fixed. SkillOpt (Yang et al., 2026) casts skill learning as optimization: an optimizer model turns scored rollouts into bounded add/delete/replace edits on a single skill document, accepted only when a held-out score improves. GSkill (Tan et al., 2026), built on the same GEPA machinery, auto-generates verifiable coding tasks for a given repository and reflectively evolves them into a repository-specific `SKILL.md`. EvoSkill differs on both axes: rather than refining one document or specializing to one codebase, it discovers and grows a library of distinct, composable skill folders from execution failures on general agent tasks, and we evaluate whether the resulting skills can be reused across datasets and executor models.

4.3 Transfer Learning in LLM Agents

Transfer learning is well studied for neural-network fine-tuning but remains nascent for LLM-based agents. Voyager (Wang et al., 2024) showed that a code-skill library learned in one Minecraft world transfers to novel tasks in another, whereas work on prompt transfer reports mixed results, with optimized prompts often degrading under model or task shift. EvoSkill instead represents learned capabilities as self-contained skill folders with explicit triggers and procedural instructions, which can be reused without modifying the underlying model. Our results show that evolved skills can be reused across datasets and executor models, suggesting that structured skills provide a promising abstraction for transferring agent capabilities beyond the setting in which they were evolved.

5 Conclusion

We presented EvoSkill, a framework for automated skill acquisition that discovers and refines reusable agent skills through iterative failure analysis. EvoSkill identifies capability gaps from execution experience, proposes new skills or refinements to existing ones, and materializes them as structured skill folders while keeping model parameters fixed. Across diverse benchmarks and executor models, EvoSkill broadly improves agent performance. In a controlled comparison, skill evolution with EvoSkill substantially outperforms prompt evolution with GEPA. We further find evidence that evolved skills can be reused beyond the settings in which they were evolved, both across datasets and across executor models, though the benefits of reuse vary across settings.

6 Limitations

Most experiments report single runs due to computational cost; we quantify variance only for the controlled system-prompt versus skill-evolution comparison (Table 3), where results are consistent across three runs. Our evaluations also use relatively small held-out sets, and the benchmarks and executor models vary across experiments according to their purpose. Finally, EvoSkill does not improve every evaluated setting, and skills that benefit one executor do not necessarily benefit another. Our reuse experiments suggest that these differences may reflect the strategies encoded in the skills, the capabilities of the executor, and compatibility with the underlying agent harness.

7 Acknowledgments

We thank the VT LLM Lab and Sentient Labs for useful discussions and feedback. This project was partially supported by award IIS-2553891 from the National Science Foundation (NSF), as well as awards from the Amazon–Virginia Tech Initiative for Efficient and Robust Machine Learning, Google, and Sentient Labs. We thank Google for a Gemini Academic Program award that provided access to Gemini API credits. The authors acknowledge Advanced Research Computing at Virginia Tech for providing computational resources and technical support that have contributed to the results reported within this paper. URL: <https://arc.vt.edu/>.

AI Disclosure. Large language models and coding agents are used to aid with and polish writing and generate some tables and plots.

References

- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=RQm2KQTM5r>.
- Salaheddin Alzu'bi, Baran Nama, Arda Kaz, Anushri Eswaran, Weiyuan Chen, Sarvesh Khetan, Rishab Bala, Tu Vu, and Sewoong Oh. Roma: Recursive open meta-agent framework for long-horizon multi-agent systems. *arXiv preprint arXiv:2602.01848*, 2026. URL <https://arxiv.org/abs/2602.01848>.
- Anomaly. OpenCode: The open source ai coding agent. <https://opencode.ai/docs>, 2026.
- Anthropic. Claude 3.7 sonnet and claude code. <https://www.anthropic.com/news/claude-3-7-sonnet>, February 2025.
- Yufei He, Juncheng Liu, Yue Liu, Yibo Li, Tri Cao, Zhiyuan Hu, Xinxing Xu, and Bryan Hooi. Evotest: Evolutionary test-time learning for self-improving agentic systems. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=JFnnajbkvP>.
- Ana Hevesi. Openhands: From readme to open source movement. <https://www.openhands.dev/blog/openhands-from-readme-to-open-source-movement>, October 2024.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=chfJJYC3iL>.
- Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananeey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. Fact, fetch, and reason: A unified evaluation of retrieval-augmented generation. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4745–4759, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.243. URL <https://aclanthology.org/2025.naacl-long.243/>.
- Yoonho Lee, Joseph Boen, and Chelsea Finn. Feedback descent: Open-ended text optimization via pairwise comparison. *arXiv preprint arXiv:2511.07919*, 2025. URL <https://arxiv.org/abs/2511.07919>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=S37h0erQLB>.
- Alexander Novikov, Ngan Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. URL <https://arxiv.org/abs/2506.13131>.
- OpenAI. Introducing codex. <https://openai.com/index/introducing-codex/>, May 2025.
- Thinh Pham, Nguyen Phan Nguyen, Pratibha Zunjare, Weiyuan Chen, Yu-Min Tseng, and Tu Vu. SealQA: Raising the bar for reasoning in search-augmented language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=zWb7ueH16c>.

- Arnav Singhvi, Krista Opsahl-Ong, Jasmine Collins, Ivan Zhou, Cindy Wang, Ashutosh Baheti, Jacob Portes, Sam Havens, Erich Elsen, Michael Bendersky, Matei Zaharia, and Xing Chen. Introducing OfficeQA: A benchmark for end-to-end grounded reasoning. Databricks Blog, December 2025. URL <https://www.databricks.com/blog/introducing-officeqa-benchmark-end-to-end-grounded-reasoning>.
- Shangyin Tan, Lakshya A Agrawal, Rohit Sandadi, Dan Klein, Koushik Sen, Alexandros G. Dimakis, and Matei Zaharia. Automatically learning skills for coding agents. In *Proceedings of the ACM Conference on AI and Agentic Systems, CAIS '26*, pp. 1310–1315, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724152. doi: 10.1145/3786335.3813196. URL <https://doi.org/10.1145/3786335.3813196>.
- Tu Vu, Kalpesh Krishna, Salaheddin Alzubi, Chris Tar, Manaal Faruqui, and Yun-Hsuan Sung. Foundational autoraters: Taming large language models for better automatic evaluation. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 17086–17105, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.949. URL <https://aclanthology.org/2024.emnlp-main.949/>.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL <https://openreview.net/forum?id=ehfRiF0R3a>.
- Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.
- Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026. URL <https://arxiv.org/abs/2605.23904>.
- Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills. <https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills>, October 2025.

A EvoSkill Generated Skills

The following section shows some of the skills generated using EvoSkill on different agentic tasks.

A.1 OfficeQA Skills

The following *economic-timeseries-skill* is a complex skill that contains both a SKILL.md file and relevant Python scripts to be called with it.

```
skills/economic-timeseries-analysis/SKILL.md

---
name: economic-timeseries-analysis
description: >
  Streamlined workflow for economic/financial time-series analysis tasks. Use this
  skill when
  performing multi-step economic data analysis involving inflation adjustment using
  CPI values,
  linear regression on time-series data, or analysis of nominal dollar values
  converted to real
  values. Triggers on tasks requiring Treasury data analysis, BLS CPI adjustments,
  economic trend
  analysis, or formatted regression output like [slope, intercept].
---

# Economic Time-Series Analysis

Standardized workflow for inflation adjustment and statistical analysis of economic
data.

## Workflow Overview

1. Gather & Structure Data - Collect nominal values and CPI data with consistent
  period formatting
2. Adjust for Inflation - Apply CPI formula: Real = Nominal x (CPI_base /
  CPI_current)
3. Perform Analysis - Run linear regression on inflation-adjusted values
4. Format Output - Present results as [slope, intercept] rounded to 2 decimal
  places

## Step 1: Data Collection

Gather these inputs:
- Nominal values: Dollar amounts by period (e.g., monthly Treasury data)
- CPI values: Consumer Price Index for each period (BLS CPI-U series)
- Base period: Reference period for inflation adjustment (e.g., "1970-03")

Structure data with consistent period format (YYYY-MM):

```json
{
 "nominal_values": [
 {"period": "1970-03", "value": 275.52},
 {"period": "1970-04", "value": 318.44}
],
 "cpi_values": [
 {"period": "1970-03", "value": 38.8},
 {"period": "1970-04", "value": 39.0}
],
 "base_period": "1970-03"
}
```
---
```

```

## Step 2: Inflation Adjustment

Apply the standard inflation adjustment formula:

'''
Real Value = Nominal Value x (CPI_base / CPI_current)
'''

Example calculation:
- Nominal: $318.44 (April 1970)
- CPI_base (March 1970): 38.8
- CPI_current (April 1970): 39.0
- Real = 318.44 x (38.8 / 39.0) = $316.81

## Step 3: Linear Regression

Run regression with period index as x-values (0, 1, 2, ...) and real values as y-
values.

Use `scripts/analyze_timeseries.py` for automated analysis:

'''bash
python scripts/analyze_timeseries.py input.json
'''

Input JSON format:
'''json
{
    "nominal_values": [...],
    "cpi_values": [...],
    "base_period": "1970-03",
    "analysis_type": "linear_regression"
}
'''

## Step 4: Format Output

Present regression results rounded to 2 decimal places:

'''
[slope, intercept]
'''

Example: `[44.00, 231.52]`

## Quick Reference

Step	Action	Output
1	Structure data	JSON with periods, values
2	Apply CPI adjustment	Real values series
3	Linear regression	slope, intercept
4	Format	`[slope, intercept]`

## Common Data Sources

- **Treasury Bulletin**: Monthly Statement of Public Debt
- **BLS CPI-U**: Bureau of Labor Statistics Consumer Price Index

```

```
skills/economic-timeseries-analysis/scripts/analysis.py
```

```

1 #!/usr/bin/env python3
2 """

```

```

3 Economic Time-Series Analysis Script
4
5 Performs inflation adjustment and linear regression on economic data.
6
7 Usage:
8     python analyze_timeseries.py <data_file> [--output <output_file>]
9
10 Input JSON format:
11 {
12     "nominal_values": [
13         {"period": "1970-03", "value": 100.0},
14         {"period": "1970-06", "value": 110.0}
15     ],
16     "cpi_values": [
17         {"period": "1970-03", "value": 38.8},
18         {"period": "1970-06", "value": 39.0}
19     ],
20     "base_period": "1970-03",
21     "analysis_type": "linear_regression"
22 }
23
24 Output JSON format:
25 {
26     "real_values": [
27         {"period": "1970-03", "nominal": 100.0, "real": 100.0},
28         {"period": "1970-06", "nominal": 110.0, "real": 109.44}
29     ],
30     "regression": {
31         "slope": 44.00,
32         "intercept": 231.52
33     },
34     "formatted_result": "[44.00, 231.52]"
35 }
36 """
37
38 import json
39 import sys
40 from typing import Optional
41
42
43 def load_data(filepath: str) -> dict:
44     """Load JSON data from file."""
45     with open(filepath, 'r') as f:
46         return json.load(f)
47
48
49 def validate_data(data: dict) -> tuple[bool, str]:
50     """Validate input data structure."""
51     required = ['nominal_values', 'cpi_values', 'base_period']
52     for field in required:
53         if field not in data:
54             return False, f"Missing required field: {field}"
55
56     if not data['nominal_values']:
57         return False, "nominal_values cannot be empty"
58     if not data['cpi_values']:
59         return False, "cpi_values cannot be empty"
60
61     return True, ""
62
63
64 def get_cpi_for_period(cpi_values: list, period: str) -> Optional[float]:
65     """Get CPI value for a specific period."""
66     for item in cpi_values:
67         if item['period'] == period:

```

```

68         return float(item['value'])
69     return None
70
71
72 def adjust_for_inflation(nominal_values: list, cpi_values: list, base_period: str) ->
73     list:
74     """
75     Adjust nominal values for inflation.
76
77     Formula: Real Value = Nominal x (CPI_base / CPI_current)
78     """
79     cpi_base = get_cpi_for_period(cpi_values, base_period)
80     if cpi_base is None:
81         raise ValueError(f"Base period CPI not found: {base_period}")
82
83     real_values = []
84     for item in nominal_values:
85         period = item['period']
86         nominal = float(item['value'])
87
88         cpi_current = get_cpi_for_period(cpi_values, period)
89         if cpi_current is None:
90             raise ValueError(f"CPI not found for period: {period}")
91
92         real = nominal * (cpi_base / cpi_current)
93         real_values.append({
94             'period': period,
95             'nominal': nominal,
96             'real': round(real, 2)
97         })
98
99     return real_values
100
101 def linear_regression(values: list) -> tuple[float, float]:
102     """
103     Perform simple linear regression.
104
105     Uses period index (0, 1, 2, ...) as x-values.
106     Returns (slope, intercept) rounded to 2 decimal places.
107     """
108     n = len(values)
109     if n < 2:
110         raise ValueError("Need at least 2 data points for regression")
111
112     # x values are indices 0, 1, 2, ...
113     # y values are the real (inflation-adjusted) values
114     y_values = [v['real'] for v in values]
115
116     # Calculate means
117     x_mean = (n - 1) / 2 # mean of 0, 1, ..., n-1
118     y_mean = sum(y_values) / n
119
120     # Calculate slope: sum((x-x_mean)(y-y_mean)) / sum((x-x_mean)^2)
121     numerator = sum((i - x_mean) * (y - y_mean) for i, y in enumerate(y_values))
122     denominator = sum((i - x_mean) ** 2 for i in range(n))
123
124     if denominator == 0:
125         slope = 0.0
126     else:
127         slope = numerator / denominator
128
129     # Calculate intercept
130     intercept = y_mean - slope * x_mean
131

```

```

132     return round(slope, 2), round(intercept, 2)
133
134
135 def analyze(data: dict) -> dict:
136     """Main analysis function."""
137     # Validate
138     valid, error = validate_data(data)
139     if not valid:
140         raise ValueError(error)
141
142     # Adjust for inflation
143     real_values = adjust_for_inflation(
144         data['nominal_values'],
145         data['cpi_values'],
146         data['base_period']
147     )
148
149     # Perform analysis
150     analysis_type = data.get('analysis_type', 'linear_regression')
151
152     result = {
153         'real_values': real_values
154     }
155
156     if analysis_type == 'linear_regression':
157         slope, intercept = linear_regression(real_values)
158         result['regression'] = {
159             'slope': slope,
160             'intercept': intercept
161         }
162         result['formatted_result'] = f"[{slope:.2f}, {intercept:.2f}]"
163
164     return result
165
166
167 def main():
168     if len(sys.argv) < 2:
169         print("Usage: python analyze_timeseries.py <data_file> [--output <output_file>]")
170         print("\nExample:")
171         print("  python analyze_timeseries.py input.json")
172         print("  python analyze_timeseries.py input.json --output results.json")
173         sys.exit(1)
174
175     input_file = sys.argv[1]
176     output_file = None
177
178     if '--output' in sys.argv:
179         idx = sys.argv.index('--output')
180         if idx + 1 < len(sys.argv):
181             output_file = sys.argv[idx + 1]
182
183     try:
184         data = load_data(input_file)
185         result = analyze(data)
186
187         output = json.dumps(result, indent=2)
188
189         if output_file:
190             with open(output_file, 'w') as f:
191                 f.write(output)
192             print(f"Results written to: {output_file}")
193         else:
194             print(output)
195

```

```

196         # Print formatted result for easy extraction
197         if 'formatted_result' in result:
198             print(f"\nFormatted Answer: {result['formatted_result']}")
199
200     except Exception as e:
201         print(f"Error: {e}", file=sys.stderr)
202         sys.exit(1)
203
204
205 if __name__ == "__main__":
206     main()

```

A.2 SealQA

The following *search-persistence-protocol* skill contains comprehensive instructions for web-search for conclusive answers where conflicting sources may exist.

```

skills/search-persistence-protocol/SKILL.md
---
name: search-persistence-protocol
description: Enforces exhaustive search strategies before concluding factual answers.
  Use this skill when answering questions requiring web research, questions with
  potentially ambiguous terms, or enumeration/counting questions. Prevents
  premature search termination by requiring term interpretation expansion, multi-
  source verification, and completeness checks.
---

# Search Persistence Protocol

Enforces thorough search coverage before concluding. The agent's most common failure
mode is stopping searches too early.

## Rule 1: Term Interpretation Expansion

Before searching ambiguous terms, list ALL reasonable interpretations:

**Example expansions:**
- "Agency" -> government agencies, space agencies, news agencies, regulatory agencies,
  intelligence agencies
- "Most widely used" -> by active users, by market share, by downloads, by revenue
- "Popular" -> most followers, most engagement, most referenced
- "Best" -> highest rated, most sales, expert recommended

**Protocol:** Execute searches for EACH interpretation before narrowing. Do not assume
the first interpretation is correct.

## Rule 2: Three-Source Minimum

Before concluding ANY factual answer:
1. Check at least THREE independent sources
2. If only 1-2 sources found, search with different query formulations
3. For enumeration questions, cross-reference list against 2+ additional sources

**Trigger:** Before stating any factual claim, verify: "Have I checked 3+ sources?"

## Rule 3: "Unable to Find" Protocol

Before reporting inability to find data:

1. **Try 3+ query formulations** - Rephrase, use synonyms, try different term orders
2. **Try related searches** - Information may exist in adjacent contexts
3. **Attempt derivation** - Can the answer be calculated from related data?
  - If 96% have X, then ~4% don't have X

```

```
- If list shows 11 items but "top 12" is mentioned, one is missing

**Rule 3a: Data Source Follow-Through** - If you identify a specific data source that
should contain the answer (database URL, API endpoint, indicator code, dataset ID
), you MUST attempt to fetch from it before concluding "unable to find."

- [X] "The ITU DataHub has indicator 100095 for 2G coverage. To get the exact figure,
query the DataHub directly." -> WRONG: identified source but didn't fetch
- [OK] "The ITU DataHub has indicator 100095. Let me query it directly..." [proceeds
to fetch] -> CORRECT

Anti-pattern: Telling the user "you would need to query X" when you could query X
yourself.

Only report "unable to find" after exhausting these steps.

## Rule 4: Enumeration Completeness Check

When counting items in a category:

1. Search "[category] complete list" and "[category] all time"
2. Cross-reference count with 2+ ranking/list sources
3. Before finalizing: "Could I have missed any?"
4. If sources disagree on count, investigate the discrepancy

## Execution

For each factual question:

```
1. EXPAND: List all interpretations of ambiguous terms
2. SEARCH: Query each interpretation separately
3. VERIFY: Confirm 3+ independent sources checked
4. COMPLETE: For enumerations, cross-check for missing items
5. DERIVE: If direct answer unavailable, attempt calculation from related data
6. CONCLUDE: Only after steps 1-5 are satisfied
```
```

B EvoSkill Loop Pseudocode

Algorithm 1 gives the full pseudocode for the EvoSkill loop described in Section 2.2.

Algorithm 1 EVOSKILL: Iterative Skill Induction via Textual Feedback

Require: Executor Agent A , Skill Proposer P , Skill Generator S , evaluation set V , frontier capacity k , max iterations T

```

// Initialization
1:  $H \leftarrow []$ 
2:  $G \leftarrow \{A\}$ 
3:  $s_A \leftarrow \text{EVAL}(A, V)$ 

// Evolution Loop
4: for  $t = 1$  to  $T$  do
5:    $i \leftarrow (t \bmod |G|)$ ;  $p \leftarrow G[i]$ 
6:    $F \leftarrow \text{COLLECTFAILURES}(p)$ 
7:   if  $F = \emptyset$  continue
8:   end if
9:    $\pi \leftarrow P(F, H)$ 
10:   $\tilde{p} \leftarrow S(p, \pi)$ 
11:  // Frontier Update
12:   $\tilde{s} \leftarrow \text{EVAL}(\tilde{p}, V)$ 
13:  if  $|G| < k$  or  $\tilde{s} > \min_{q \in G} \text{SCORE}(q)$ 
14:     $G \leftarrow G \cup \{\tilde{p}\}$ 
15:    if  $|G| > k$   $G \leftarrow G \setminus \{\arg \min_{q \in G} \text{SCORE}(q)\}$ 
16:    end if
17:  end if
18:   $\text{APPENDFEEDBACK}(H, \pi, \tilde{s})$ 
19: end for

// Output
19: return  $\arg \max_{q \in G} \text{SCORE}(q)$ 

```

$\triangleright$ feedback history
 $\triangleright$ frontier of top-k programs
 $\triangleright$ baseline score
 $\triangleright$ round-robin parent selection
 $\triangleright$ scores $< \tau$
 $\triangleright$ propose new skill or edit
 $\triangleright$ build candidate
 $\triangleright$ held-out score
 $\triangleright$ update frontier
 $\triangleright$ update history log
 $\triangleright$ best performing program

C Agent Prompts

This appendix provides the prompts used for each agent role (placeholders shown).

C.1 Skill Proposer

Skill Proposer Prompt

```

"""
You are an expert agent performance analyst specializing in identifying opportunities
to enhance agent capabilities through skill additions or modifications. Your role
is to carefully analyze agent execution traces and propose targeted skill
improvements.

## Your Task

Given an agent's execution trace, its answer, and the ground truth answer, propose
either:
- A **new skill** (action="create") if no existing skill covers the capability gap
- An **edit to an existing skill** (action="edit") if an existing skill SHOULD have
prevented the failure but didn't

Your proposal will be passed to a downstream Skill Builder agent for implementation.

## Required Pre-Analysis Steps

BEFORE proposing any skill, you MUST use the **Brainstorming skill** (read `.claude/
skills/brainstorming/SKILL.md`):

1. **Use Brainstorming skill** (MANDATORY):
- Read and follow the process in `.claude/skills/brainstorming/SKILL.md`

```

- Propose 2-3 different approaches to address the failures
 - For each approach: describe the core idea, trade-offs, and complexity
 - Explore alternatives before settling on your final proposal
 - Apply YAGNI - choose the simplest solution that addresses the root cause
2. ****Inventory existing skills****: Review the list of existing skills provided in the query
 - Understand what capabilities are already available
 - Check if any existing skill covers similar ground
 3. ****Analyze feedback history**** for:
 - DISCARDED proposals similar to what you're considering
 - Patterns in what works vs what regresses scores
 - Skills that were active when failures occurred
 4. ****Determine action type****:
 - If an existing skill SHOULD have prevented this failure but didn't -> propose EDIT
 - If no existing skill covers this capability -> propose CREATE
 - If a DISCARDED proposal was on the right track -> explain how yours differs

Analysis Process

Before proposing a solution, work through these steps:

<analysis>

1. ****Trace Review****: Examine the agent's execution trace step-by-step
 - What actions did the agent take?
 - Where did it succeed or struggle?
 - What information was available vs. missing?
2. ****Gap Analysis****: Compare the agent's answer to the ground truth
 - What specific information is incorrect or missing?
 - What reasoning errors occurred?
 - What capabilities would have prevented these issues?
3. ****Existing Skill Check****: Review the listed existing skills
 - Does any existing skill cover this capability?
 - If yes, why did it fail to prevent the error?
 - Should that skill be EDITED instead of creating a new one?
4. ****Skill Identification****: Determine what skill would address the failure
 - What new capability, tool, or workflow would help?
 - What inputs should it accept?
 - What outputs should it produce?
 - How would it integrate with existing capabilities?

</analysis>

Anti-Patterns to Avoid

- DON'T propose a new skill if an existing one covers similar ground -> propose an EDIT instead
- DON'T ignore previous DISCARDED proposals for the same problem -> explain how yours differs
- DON'T create narrow skills that only fix one specific failure -> ensure broad applicability
- DON'T propose capabilities that overlap with existing skills -> consolidate instead

When to Propose Skills

Propose a skill when ANY of these apply:

- Agent lacks access to information, APIs, or computational capabilities
- The fix requires a multi-step procedure (>3 sequential steps)
- The fix involves output structuring, formatting, or templates
- The improvement would be reusable across different tasks

```

- The issue is about WHAT steps to take, not HOW to think

## Output Requirements

Based on your analysis, provide:

1. **action**: Either "create" for a new skill or "edit" for modifying an existing skill

2. **target_skill**: (Required if action="edit") The name of the existing skill to modify

3. **proposed_skill**: A detailed description of:
  - For CREATE: The new skill to be built (capability, inputs, outputs, problem it solves)
  - For EDIT: The modifications needed to the existing skill

4. **justification**: Explain your reasoning
  - Reference specific moments in the trace that informed your decision
  - Reference specific existing skills and why they were/weren't suitable
  - Reference any related past iterations (especially DISCARDED ones)
  - Explain how your proposal addresses the identified gap

5. **related_iterations**: List of relevant past iterations (e.g., ["iter-4", "iter-9"])

## Example Analyses

<example type="edit_existing_skill">
**Situation**: Agent failed to calculate Expected Shortfall correctly. The financial-methodology-guide skill exists but didn't cover multi-period ES calculations.

**Proposal**:
- action: "edit"
- target_skill: "financial-methodology-guide"
- proposed_skill: "Extend the ES/CVaR section to include multi-period calculations. Add: (1) rolling window ES computation, (2) confidence interval adjustment for different time horizons, (3) examples showing ES at 1-day, 5-day, and 10-day horizons."
- justification: "The existing financial-methodology-guide skill covers basic ES but doesn't address the multi-period case seen in failure 1. Rather than creating a redundant skill, we should extend the existing methodology guide. Iter-3 was DISCARDED for proposing a separate 'multi-period-risk' skill - this proposal adds to the existing skill instead."
- related_iterations: ["iter-3"]
</example>

<example type="create_new_skill">
**Situation**: Agent failed to parse Treasury bond prices in 32nds notation. No existing skill covers notation parsing.

**Proposal**:
- action: "create"
- target_skill: null
- proposed_skill: "Create a 'bond-notation-parser' skill that handles Treasury price notation. It should: (1) recognize 32nds format (e.g., 99.16 = 99 + 16/32), (2) handle the '+' suffix (e.g., 99.16+ = 99 + 16.5/32), (3) validate that fractional parts are 00-31, (4) provide clear examples and conversion formulas."
- justification: "No existing skill covers notation parsing. The trace shows the agent interpreted '99.16' as decimal 99.16 instead of 99.5 (99 + 16/32). This is a distinct capability not covered by existing methodology skills."
- related_iterations: []
</example>

<example type="data_access_skill">

```

```

**Situation**: Agent failed to answer a question about current stock prices because it
    only had access to historical data.

**Proposal**:
- action: "create"
- target_skill: null
- proposed_skill: "The agent needs a real-time stock price retrieval capability. This
    skill should accept a stock ticker symbol as input and return current market data
    including the latest price, daily change (absolute and percentage), and trading
    volume. It should handle invalid tickers gracefully and indicate whether markets
    are currently open or closed."
- justification: "At step 3 in the trace, the agent correctly identified the need for
    current pricing data and attempted to use its historical data tool. However, the
    ground truth required real-time information from today's trading session. The
    agent's reasoning was sound but it lacked the necessary data access. No existing
    skill provides real-time market data access."
- related_iterations: []
</example>
"""

```

C.2 Skill Generator

Skill Generator Prompt

```

"""
You are an expert skill developer specializing in creating tools and capabilities for
Claude Code agents. Your role is to implement well-structured, production-ready
skills based on high-level descriptions provided by a Proposer agent.

## Primary Directive

**Before implementing any skill, always read and follow the `.claude/skills/skill-
creator/SKILL.md` skill.** This skill contains essential patterns, validation
requirements, scripts, and best practices that ensure your implementations work
correctly within the Claude Code ecosystem. Follow its guidance for all skill
creation tasks.

## Your Task

Given a proposed skill description, implement a complete, functional skill that:
1. Follows the skill-creator's structure and conventions
2. Integrates properly with the Claude Code SDK
3. Is well-documented and maintainable
4. Handles edge cases gracefully

## Implementation Process

Work through these steps for each skill implementation:

<implementation_steps>
1. **Read the Skill-Creator Skill**: Load and follow `.claude/skills/skill-creator/
SKILL.md`
2. **Implement and Validate**: Build, test, and package the skill following skill-
creator guidelines
</implementation_steps>

## Quality Reminder

The context window is a shared resource. Every token in your skill competes with
conversation history, other skills, and user requests. Challenge each piece of
content: "Does Claude really need this?" Keep skills concise and let Claude's
intelligence fill in the gaps.
"""

```

C.3 Grading

We use the default LLM-as-a-judge (Vu et al., 2024) template provided in the original SealQA task for all auto-grading tasks.

```
Grading Prompt

# Auto-Grader Prompt (Placeholder)

## System
You are the Auto-Grader agent. Evaluate a model output against a reference answer
and scoring rubric.

## Inputs
- `question`: `<QUESTION>`
- `prediction`: `<MODEL_OUTPUT>`
- `reference`: `<REFERENCE_ANSWER>`
- `tolerance`: `<TOLERANCE>` (e.g., `0.01` for 1%)

## Output format
Return a JSON object with:
- `score`: `<FLOAT_0_TO_1>`
- `passed`: `<TRUE_FALSE>`
- `reason`: `<SHORT_EXPLANATION>`
- `error_breakdown`: [{`type`: `<TYPE>`, `value`: `<VALUE>`}, ...]

## Notes
- Prefer deterministic checks when possible.
- If numeric, compute relative error using placeholder formula:
  `rel_err = abs(p - r) / max(abs(r), eps)` with `eps = 1e-9`.
```

D Scoring and Data Setup

We evaluate agent responses using a deterministic fuzzy matching scorer that returns a binary correctness signal (1.0 or 0.0). Given a ground-truth answer g and a predicted answer p , the scorer first attempts to extract numerical values from both strings along with surrounding textual context (a 20-character window). When both g and p contain numeric content, the scorer normalizes each extracted number by detecting unit keywords in the local context (e.g., “million,” “billion,” “trillion”) and compares base values using a relative tolerance τ : a prediction is accepted if $|g_{\text{base}} - p_{\text{base}}| / |g_{\text{base}}| \leq \tau$. For the primary evaluation we set $\tau = 0$, requiring exact numerical agreement. To avoid spurious matches against incidental year references (e.g., “reported in 2023”), the scorer filters candidate numbers in the 1900–2100 range from predictions unless the ground truth itself is a year or contains significant non-numeric text. For hybrid answers containing both text and numbers (e.g., “March 1977”), the scorer additionally verifies that key textual elements present in the ground truth appear in the prediction via case-insensitive substring matching after stripping unit words and parenthetical abbreviations. Multi-number ground-truth answers (i.e., lists) require that all constituent values are recovered in the prediction. For purely textual answers, matching is performed via case-insensitive substring containment after normalization (whitespace trimming, quote removal, and parenthetical stripping). During the self-improvement training loop, we additionally employ a multi-tolerance scoring variant that computes a weighted average over five tolerance levels $\tau \in \{0.0, 0.01, 0.025, 0.05, 0.10\}$, with weights $w(\tau) = 1 / (1 + 20\tau)$ that favor stricter thresholds; a weighted score below 0.8 flags the example as a failure for targeted skill refinement.

E Environment Branches

We manage agent configurations (which we term programs) using a git-backed version control scheme that naturally encodes parent–child lineage and supports efficient frontier-based selection. Each program is stored on a dedicated git branch (prefixed `program/`) with a YAML configuration file (`.claude/program.yaml`) that records the program’s name, a pointer to its parent branch, a generation counter (i.e., mutation depth from the root), the agent’s system prompt, allowed tools, and evaluation metadata including scores. At initialization, a base program is created at generation 0 with no parent. At each iteration of the self-improvement loop, the system selects the highest-scoring program from a maintained frontier, a bounded set of top-performing programs tracked via git tags (prefixed `frontier/`), and designates it as the parent. The parent’s configuration is then mutated by invoking a proposer agent that analyzes sampled failures, producing a child program that inherits all parent attributes (system prompt, tool permissions) while introducing a targeted modification: either a new or edited skill file (in skill-only mode) or a rewritten system prompt (in prompt-only mode). The child is instantiated by checking out the parent branch, creating a new branch named `iter-mode- $n$` (where n is the global iteration index), writing the mutated configuration, and committing all changes. After the child is evaluated on a held-out validation set, it is admitted to the frontier if either the frontier has not reached its maximum capacity K (default $K=3$) or its score exceeds that of the current worst frontier member, in which case the weakest member is evicted. Children that fail to enter the frontier are discarded; their branches are deleted to prevent repository bloat. This design yields a tree-structured search over program space in which each node is a fully reproducible snapshot: lineage can be reconstructed by following parent pointers from any program back to the root, and any historical configuration can be restored via a single git checkout. An early-stopping criterion halts the loop after a configurable number of consecutive iterations without frontier improvement.