EvoSkill: Automated Skill Discovery for Agents

Salaheddin Alzubi
Sentient
USA

Noah Provenzano
Virginia Tech
Blacksburg, Virginia, USA

Christian Calvo
Virginia Tech
Blacksburg, Virginia, USA

Jaydon Bingham
Virginia Tech
Blacksburg, Virginia, USA

Weiyuan Chen
Virginia Tech
Blacksburg, Virginia, USA

Himanshu Tyagi
Sentient
USA

Dastin Huang
Sentient
USA

Tu Vu
Virginia Tech
Blacksburg, Virginia, USA

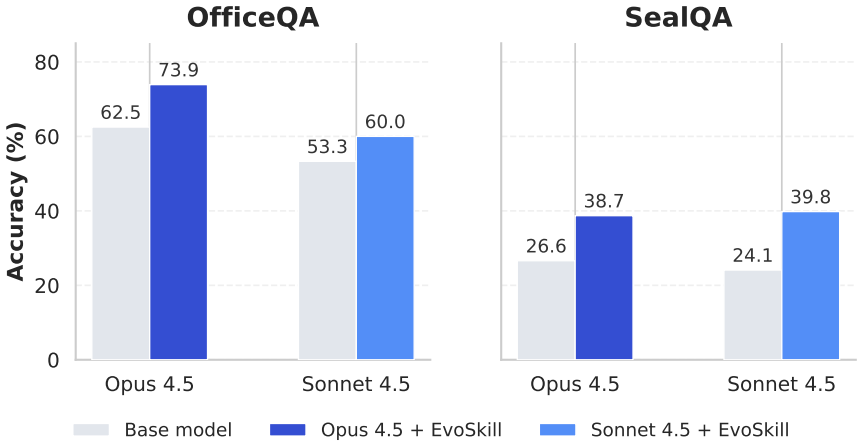

Figure 1. EvoSkill Performance on Claude Opus & Sonnet

Abstract

Coding agents are increasingly used as general-purpose problem solvers, but their flexibility does not by itself confer the domain expertise needed for specialized tasks. Recent work addresses this through *agent skills*: reusable workflows and code that augment agents with domain-specific capabilities. Most skills today are hand-crafted, and existing evolutionary approaches optimize low-level artifacts (e.g. prompts & code) that are tightly coupled to specific models and tasks. We introduce **EvoSkill**, a self-evolving framework that automatically discovers and refines agent skills through iterative failure analysis. EvoSkill analyzes execution failures, proposes new skills or edits to existing ones, and materializes them into structured, reusable skill folders. A frontier of agent programs governs selection, retaining only skills that improve held-out validation performance while the underlying model remains frozen. We evaluate EvoSkill across four benchmarks (OfficeQA, SealQA, LiveCodeBench, and FRAMES) and multiple model families including Claude, GPT-OSS, Kimi, and Gemini, achieving gains of up to **+15.7%** on SealQA and **+11.4%** on OfficeQA. EvoSkill outperforms GEPA, a prompt-level evolutionary baseline. We further show that evolved skills transfer zero-shot across both models and datasets: skills evolved on SealQA improve BrowseComp accuracy by **5.3%** without modification, and skills evolved with one model can improve performance when applied to a different model family.

CCS Concepts: • **Computing methodologies** → *Multi-agent systems.*

Keywords: Agent Skills, Skills, Automatic Skill Evolution, Skill Discovery, Skills for Multi-Agent Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AgentSkills 2026, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Salaheddin Alzubi, Noah Provenzano, Christian Calvo, Jaydon Bingham, Weiyuan Chen, Himanshu Tyagi, Dastin Huang, and Tu Vu. 2026. EvoSkill: Automated Skill Discovery for Agents. In *AgentSkills 2026: First Workshop on Agent Skills, May 26, 2026, San Jose, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Coding agents (e.g., Claude Code [6], OpenHands [18], Codex [13]) have emerged as a dominant paradigm for solving tasks across a wide range of domains. This trend is driven by the increasing use of code as a flexible intermediate representation, enabling coding agents to invoke complex abstractions and operate as general-purpose problem solvers. However, while this flexibility allows agents to interface with diverse tools and domains, it does not confer the domain expertise required to perform specialized tasks at a consistently high level.

To bridge this gap, recent work has explored *agent skills*: reusable, domain-specific capabilities that augment general-purpose coding agents with structured workflows, instructions, and supporting code. Most skills today are hand-crafted on an ad-hoc basis, requiring both domain knowledge and significant manual effort; a process that scales poorly as the number of target tasks grows. Evolutionary methods such as AlphaEvolve [12], and GEPA [2], offer a promising alternative by optimizing agent artifacts: codebases, or prompts, through iterative search. However, these approaches operate at the *artifact level*: the optimized prompts or code are tightly coupled to a specific model and task configuration, and do not naturally yield reusable components that transfer across settings.

In this work, we introduce **EvoSkill**, a self-evolving framework that operates at a higher level of abstraction: rather than optimizing prompts or code directly, EvoSkill iteratively *discovers and refines agent skills* through failure-driven textual feedback. EvoSkill maintains a frontier of agent programs and, at each iteration, analyzes execution failures to propose new skills or refine existing ones. Proposed skills are materialized into structured, reusable skill folders comprising instructions, trigger metadata, and helper scripts, and are retained only if they improve performance on a held-out validation set. Skills accumulate across iterations, progressively expanding the agent's capabilities while the underlying model remains frozen.

We validate EvoSkill across four benchmarks—OfficeQA [15], SealQA [14], LiveCodeBench [7], and FRAMES [9]—and across multiple model families including Claude, GPT-OSS, Kimi, and Gemini. EvoSkill improves accuracy on the majority of model–benchmark pairs, with gains of up to **+15.7%** on SealQA (Sonnet 4.5: 24.1% → 39.8%) and **+11.4%** on OfficeQA (Opus 4.5: 62.5% → 73.9%). We show that evolved skills transfer zero-shot both across models and across benchmarks, and compare favorably to GEPA [2], a prompt-level evolutionary baseline.

Our contributions are as follows:

1. We propose **EvoSkill**, a framework for automatically discovering and refining reusable agent skills through iterative failure analysis, operating at the skill abstraction level rather than on low-level artifacts such as prompts or codebases.
2. We demonstrate that EvoSkill yields substantial improvements across four benchmarks and multiple model families, with gains of up to **+15.7%** on SealQA and **+11.4%** on OfficeQA using only small training subsets.
3. We show that evolved skills transfer zero-shot across both models and benchmarks, and outperform GEPA [2], a prompt-level evolutionary baseline.

2 Methodology

The core idea behind EvoSkill is to iteratively discover and refine agent skills by applying textual feedback descent [10] to examples where the current agent fails. EvoSkill assumes a coding agent harness that supports skill folders (e.g., Claude Code, Codex, OpenCode [4]) and a model capable of utilizing such skills. The underlying model remains frozen throughout; only the skill repository and agent metadata evolve across iterations.

2.1 Framework Overview

EvoSkill consists of three collaborating agents:

1. **Executor Agent (A):** Executes tasks under the governance of the current agent program. The base program initializes the Executor with no skills.
2. **Proposer Agent (P):** Analyzes the Executor's output traces, predicted answers, and ground-truth answers to diagnose failures and propose high-level skill descriptions. Ground-truth answers are provided to enable root-cause diagnosis, analogous to examining labeled misclassifications during error analysis in supervised learning, and are not propagated to the generated skills themselves. The Proposer determines whether to create a new skill or refine an existing one.
3. **Skill Generator Agent (S):** Materializes a high-level proposal from the Proposer into a concrete skill folder comprising trigger metadata, procedural instructions (`SKILL.md`), and optional helper scripts (Python or Typescript code) or reference material. The Skill Generator is bootstrapped with a meta-skill that codifies best practices for skill authoring.

All agents have read access to the base agent's repository; only the Skill Generator has write permissions to the skills directory. The Proposer additionally maintains a cumulative **feedback history** $\mathcal{H}$ that logs all prior proposals, their outcomes, and score deltas. This serves two purposes: it prevents redundant proposals, and it enables the Proposer to

Algorithm 1 EvoSkill — Iterative Skill Induction via Textual Feedback

Require: Executor agent A, proposer P, skill generator S, evaluation set V, frontier capacity k, max iterations T

```
// Initialization
H ← [ ]                                          ▷ feedback history
G ← {A}                                          ▷ frontier of top-k programs
s_A ← Eval(A, V)                                 ▷ baseline score

// Evolution Loop
for t = 1 to T do
    i ← (t mod |G|);  p ← G[i]                   ▷ round-robin parent selection
    F ← CollectFailures(p)                       ▷ scores < τ
    if F = ∅ continue
    end if
    π ← P(F, H)                                  ▷ propose new skill or edit
    p̃ ← S(p, π)                                  ▷ build candidate
    // Frontier Update
    s̃ ← Eval(p̃, V)                               ▷ held-out score
    if |G| < k or s̃ > min_{q∈G} Score(q)
        G ← G ∪ {p̃}                              ▷ update frontier
        if |G| > k  G ← G \ {arg min_{q∈G} Score(q)}
        end if
    end if
    AppendFeedback(H, π, s̃)                      ▷ update history log
end for

// Output
return arg max_{q∈G} Score(q)                    ▷ best performing program
```

refine what previously partially worked and avoid making the same mistakes, making its context progressively richer across iterations.

2.2 EvoSkill Loop

EvoSkill optimizes agent programs through iterative skill mutations guided by textual feedback (Algorithm 1). A *program* p encapsulates the agent's system prompt and accumulated skills. The loop maintains a fixed-capacity frontier G of the k highest-scoring programs.

At each iteration t, a parent program p is selected from the frontier G via round-robin cycling, ensuring each frontier member is explored before any is revisited. The parent is evaluated on a training batch sampled without replacement, cycling through all examples before repeating. Responses are scored against ground-truth answers using a task-specific scoring function. Samples scoring below a fixed threshold are collected into a failure set F; if no failures are found, the iteration is skipped.

The Proposer P receives F together with the feedback history H and performs structured failure analysis: reviewing execution traces, identifying capability gaps, and auditing existing skills. It then produces a textual proposal π specifying either a new skill or an edit to an existing one.

The Skill Generator S receives the current parent program p and proposal π, and materializes the proposal into a candidate program $\tilde{p}$: the parent's configuration augmented with the new or revised skill. The candidate is evaluated on the held-out validation set V using the same scoring function.

The candidate enters the frontier G if its score exceeds that of the weakest frontier member, displacing it; otherwise the candidate is discarded. Regardless of outcome, the proposal, its score, and the selection verdict are appended to H, ensuring the Proposer can reference which strategies succeeded or regressed and why. After T iterations, the loop returns the highest-scoring program in G.

2.3 Implementation Details

2.3.1 Data Setup. Given a supervised dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ and a scoring function, we first cluster the dataset into K categories using an LLM as a classifier, assigning each example to a single category. We then either take the full dataset or take a **random sample**, representative across categories. We then perform stratified partitioning into three disjoint subsets: a **training set** used for failure detection during evolution, a **validation set** used for scoring candidate programs (frontier selection), and a **held-out test set** comprising all remaining examples, which is never exposed during evolution and is used exclusively for final evaluation. Split ratios are configurable, with defaults ensuring every category is represented in both the training and validation partitions regardless of category size. Training data are organized as category-keyed pools to support the category-aware sampling used during evolution. Full details of the splitting procedure are provided in Appendix C.

2.3.2 Environment Setup. EvoSkill operates within a git repository where the codebase is fixed. Each agent program is represented as a branch that diverges from its parent only in its skill folders and metadata (system prompt, lineage information, validation score). This design ensures that performance differences between programs are attributable solely to their evolved skills, while keeping program branches lightweight. Full details of the repository configuration are provided in Appendix D.

3 Experiments

Unless otherwise noted, all experiments use Opus 4.6 as both the Skill Proposer and Skill Generator in the evolution loop, while the Executor model varies per experiment. Claude models are benchmarked using Claude Code, and all other models use OpenCode [4].

3.1 Benchmarks

We evaluate EvoSkill across four benchmarks spanning distinct capability domains: SealQA [14] (search-augmented reasoning under adversarial retrieval), LiveCodeBench [7] (competitive programming), FRAMES [9] (multi-hop factual

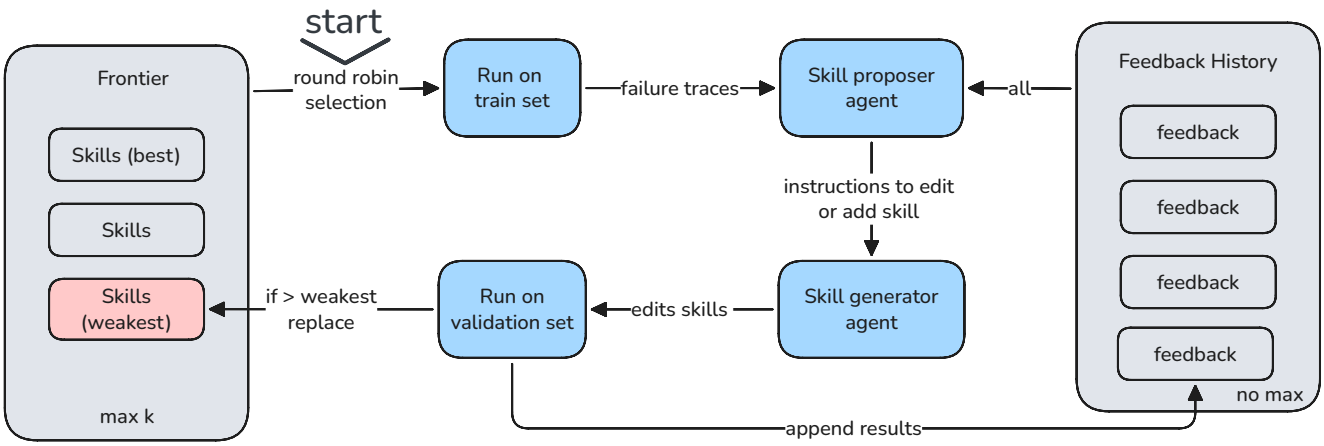

Figure 2. Overview of the EvoSkill loop. The frontier maintains up to k skill-equipped programs.

Table 1. Accuracy (%) across benchmarks and models with and without EvoSkill.

Model	SealQA	LiveCodeBench	FRAMES
GPT-OSS-120B	22.9	45.7	60.4
+ EvoSkill	**24.1**	**50.0**	**63.7**
Kimi K2.5	27.7	**70.7**	45.1
+ EvoSkill	**32.5**	**70.7**	**61.5**
Gemini 3.1 Flash Lite	21.7	**73.3**	65.9
+ EvoSkill	**22.9**	68.4	**68.1**
GPT-OSS-20B	1.2	43.8	**13.5**
+ EvoSkill	**2.4**	**50.0**	**13.5**

reasoning), and OfficeQA (grounded reasoning over dense financial documents). For each benchmark, we follow the data partitioning methodology described in Section 2.3.1, splitting into training, validation, and held-out test sets. We choose a maximum sample size of 120 to ensure a representative set of all datasets. All reported accuracies are computed on the held-out test partition. For the EvoSkill Loop, evolution is run until either the score stops improving for 5 iterations, or 20 iterations total.

3.1.1 Main Results. Table 1 presents results across four executor models and three benchmarks. EvoSkill yields improvements on the majority of model–benchmark pairs.

We observe that skills evolved for code benchmarks tend to target benchmark-specific issues (e.g., output formatting, edge-case handling) rather than improving general coding ability. More broadly, EvoSkill can surface and fix issues in the agent's prompting or tool usage that would otherwise go unnoticed, effectively making benchmark evaluation more fair. The evolved skill files also serve as a readable diagnostic: by inspecting them, developers can identify systematic failure modes in their agent

Figure 1 presents results across two executor models (Sonnet and Opus) and two benchmarks. We notice the greatest improvements with EvoSkill through Anthropic models. The larger gains with Anthropic models may reflect stronger native support for skills in the Claude Code harness.

3.1.2 OfficeQA. OfficeQA evaluates grounded reasoning over an 89,000-page U.S. Treasury corpus. Across 246 questions, agents must navigate dense tables and prose to synthesize data and perform quantitative reasoning. The complexity is reflected in the human baseline of 50 minutes per question, with most time spent on retrieval and multi-document synthesis. To score, we use the fuzzy scoring function provided by OfficeQA, which computes a weighted match across five tolerance levels favoring exact matches. Full scoring details are provided in Appendix C. Training split ablations for OfficeQA are presented in Section 3.2.

3.1.3 SealQA. SealQA [14] targets fact-seeking questions where web search yields conflicting or noisy data. Complementing OfficeQA's fixed-corpus focus, it evaluates an agent's ability to navigate the open web under adversarial conditions. This shifts the focus of **EvoSkill** from document parsing toward robust web search and source verification.

We run EvoSkill on the seal-0 split of SealQA (111 questions).

3.1.4 FRAMES. FRAMES [9] is a factual reasoning benchmark requiring agents to retrieve and synthesize information across multiple web sources. We use it alongside SealQA to evaluate whether EvoSkill shows improvement across wide retrieval challengesâĂŤspecifically the ability to reason multi-source information into coherent answers.

Table 2. Accuracy across tolerance thresholds for different training data splits.

Train Split/Tol.	0.00%	0.10%	1.00%	5.00%	10.00%
base	60.6	66.3	72.8	77.2	79.7
5%	63.4	67.4	74.3	77.6	80.1
10%	65.8	69.9	76.4	80.5	**82.5**
15%	64.5	69.9	75.6	79.3	81.3
merge-unique	**68.1**	**70.8**	**77.1**	**80.5**	82.4

Tolerance = allowable relative error. **Blue bold** = best in column.

To emphasize complex reasoning, we filter the dataset to include only questions requiring at least four source articles (27.4% of the benchmark). This subset demands broader evidence collection, effectively exercising EvoSkill's target capabilities. After filtering, data partitioning then follows Section 2.3.1.

3.2 Ablation: Training Split Size on OfficeQA

We study the effect of training set size on EvoSkill using OfficeQA, where the full benchmark's 246 questions allow controlled variation of the training partition. Following the data setup described in Section 2.3.1, we partition the benchmark into three disjoint splits: a **training set** used for failure detection during evolution, a **validation set** of 17 examples ($\approx$ 7%) used for frontier selection, and a **held-out test set** comprising the remaining questions, which is never exposed during evolution. We evaluate three training set sizes: 5% (12 examples), 10% (24 examples), and 15% (36 examples). We additionally evaluate a **skill-merge** configuration, which combines unique skills discovered across independent runs into a single skill library; when skills overlap (identified by matching names or descriptions), we retain the version from the highest-performing run. All reported accuracies are computed on the held-out test partition unless otherwise noted.

Table 2 presents results across all configurations and tolerance levels. EvoSkill yields consistent improvements over the baseline[1] across all settings. On exact match (0% tolerance), training on 5% of the data improves accuracy from 60.6% to 63.4% (+2.8%), while 10% training data yields 65.8% (+5.2%). Performance plateaus beyond 10%: the 15% split achieves 64.5%, slightly below the 10% run, suggesting diminishing returns or mild overfitting as training data grows.

The skill-merge configuration achieves the strongest result at **67.9%** exact match (+7.3% over baseline). This indicates that skills discovered from independent runs are complementary: each run surfaces different failure modes and corresponding capabilities, and combining them produces a more complete skill library. The improvement pattern is consistent across tolerance levels, with gains ranging from 2.7-4.5% at stricter tolerances.[2]

[1]The baseline was independently run & cross-referenced with authors' most recent result (see: https://github.com/databricks/officeqa/issues/10#issuecomment-3719842269).

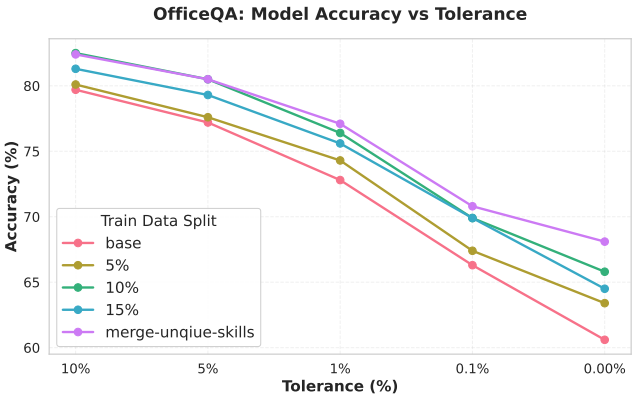

Figure 3. EvoSkill performance OfficeQA benchmark across training splits and tolerance levels.

Table 3. Accuracy comparison across benchmarks for the GPT-OSS-120B baseline, GEPA, and EvoSkill.

Method	SealQA	LiveCodeBench	FRAMES
GPT-OSS-120B	23.3	45.7	60.4
+ GEPA	21.8	49.5	30.4
+ EvoSkill	**27.3**	**50.0**	**63.7**

Bold = best in column. Implementation details provided in Appendix E

3.3 Comparison with GEPA

GEPA [2] is a reflective prompt evolution framework built on DSPy [8] that uses LLM-driven reflection to iteratively optimize the textual components of a program. At each iteration, GEPA evaluates a candidate prompt on training examples, collects feedback on failures, and uses a reflection model to propose mutations to the prompt's instruction fields. Candidates are scored on a validation set using Pareto-based selection, retaining the highest-performing variants across iterations. Because GEPA operates at the prompt level—optimizing free-form instruction text rather than structured, reusable skill modules—it provides a natural baseline for evaluating whether EvoSkill's skill-level abstraction confers advantages beyond what prompt-level evolutionary optimization can achieve. We run GEPA for 20 iterations, matching the maximum iteration budget used by EvoSkill.

3.3.1 Results. Table 3 compares EvoSkill, GEPA, and the GPT-OSS-120B baseline across three benchmarks. EvoSkill

[2]We note that each configuration was evaluated on a single run due to computational costs.

consistently outperforms both the baseline and GEPA, achieving the highest accuracy on SealQA (27.3%) and LiveCodeBench (50.0%). GEPA underperforms the baseline on SealQA (21.8% vs. 23.3%) while showing gains on LiveCodeBench (49.5% vs. 45.7%), suggesting that prompt-level optimization can be inconsistent across tasks. These results indicate that EvoSkill's skill-level abstraction yields more reliable improvements than GEPA's prompt-level evolution, particularly on tasks requiring structured domain capabilities.

3.4 Skill Transfer Across Models

We test whether evolved skills transfer across models by evolving skill libraries on SealQA using one model and evaluating them zero-shot on a different executor model. Table 4 presents the full transfer matrix.

Transfer is highly model-specific. GPT-OSS-120B struggled with Opus 4.5's evolved skills, which relied on many sequential tool calls and multi-pass reasoning; the model became increasingly lost following these complex workflows, whereas Sonnet 4.5's simpler skills transferred more effectively. GPT-OSS-20B was too weak to work reliably with the OpenCode harness for tool use, limiting the effectiveness of any transferred skills. Additionally, differences between agent harnesses (e.g., OpenCode vs. Claude Code SDK) introduce their own peculiarities that affect skill portability.

3.5 Skill Transfer Across Datasets

A key design goal of EvoSkill is that evolved skills, being structured and interpretable, should transfer across tasks without modification. A *search-persistence-protocol* skill evolved by Opus 4.5 on SealQA transfers zero-shot to BrowseComp [19] (128 examples), improving accuracy from 43.5% to **48.8%** (+5.3%). This provides direct evidence that skills discovered by EvoSkill are not overfit to their training task: the search-persistence-protocol captures a general capability that is broadly useful for fact-seeking tasks regardless of the specific benchmark.

4 Related Work

4.1 Agent Skills

Augmenting agents with reusable, modular capabilities has roots in both embodied AI and software engineering. Voyager [17] introduced an ever-growing skill library for an LLM-powered Minecraft agent, discovering skills through an automatic curriculum and iterative prompting with environment feedback. More recently, the Agent Skills specification [1] formalized skills as a portable, open format: filesystem directories containing a `SKILL.md` file with metadata and procedural instructions, optionally bundled with helper scripts. This format has been adopted across multiple agent harnesses [3, 5] and leverages progressive disclosure: metadata loaded at startup, instructions read on demand, scripts executed without entering the context window enabling agents to maintain many skills with minimal context overhead. Despite this mature infrastructure, skills today are predominantly hand-authored. EvoSkill addresses this gap by automatically discovering and refining skills through iterative failure analysis, producing artifacts that conform to the same structured format.

4.2 Textual Feedback and Evolutionary Optimization

A growing body of work uses natural-language feedback to guide iterative improvement of LLM-generated artifacts. Self-Refine [11] showed that a single LLM can improve its own outputs through a generate-critique-refine loop, though it operates on individual outputs and does not accumulate knowledge across iterations. Feedback Descent [10] formalizes this into a general optimization framework, maintaining a frontier of top-performing candidates with accumulated feedback history to drive sustained improvement across domains. EvoSkill builds directly on this paradigm, applying textual feedback descent to skill discovery rather than artifact optimization.

On the evolutionary side, AlphaEvolve [12] evolves entire codebases through an LLM-ensemble evolutionary loop grounded by automatic evaluation, while GEPA [2] uses reflective mutation and Pareto-based selection to evolve prompts within DSPy. Both demonstrate the power of evolutionary search with LLM-driven mutations, but optimize low-level artifacts (code or prompts) tightly coupled to specific tasks and models.

4.3 Transfer Learning in LLM Agents

Transfer learning has been extensively studied for neural network fine-tuning, but its application to LLM-based agents remains nascent. Voyager [17] showed that a skill library learned in one Minecraft world could solve novel tasks in a new world. Work on prompt transfer has found mixed results, prompts optimized for one setting often degrade when the model or task distribution shifts.

EvoSkill offers a different angle: because skills are self-contained folders with explicit trigger conditions and procedural instructions, they are decoupled from both the training task and the underlying model. Our cross-model and cross-dataset transfer experiments (Sections 3.4 and 3.5) provide direct evidence that skill-level optimization yields a more transferable unit than prompt-level or code-level optimization.

5 Conclusion

We presented EvoSkill, a self-evolving framework that automatically discovers and refines reusable agent skills through iterative failure analysis. By operating at the skill level rather than optimizing low-level artifacts such as prompts or codebases, EvoSkill produces structured, interpretable capabilities

Table 4. Zero-shot skill transfer across models on SealQA (% accuracy). Rows indicate the model whose EvoSkill evolution produced the skill library; columns indicate the model that executes with those skills at test time.

Skill Origin ↓	Sonnet 4.5	GPT-OSS-120B	Kimi K2.5	GPT-OSS-20B
No skills (baseline)	24.1	22.9	27.7	1.2
Opus 4.5	21.7	12.0	36.1	1.2
Sonnet 4.5	39.8	30.1	34.9	2.4
GPT-OSS-120B	19.3	24.1	18.1	1.2
Kimi K2.5	8.4	28.9	32.5	1.2

that accumulate over iterations. Our experiments demonstrate consistent improvements across four benchmarks and multiple model families, with gains of up to +15.7% on SealQA and +11.4% on OfficeQA. EvoSkill outperforms GEPA, a prompt-level evolutionary baseline, and we show that evolved skills transfer zero-shot across both models and datasets.

Several directions remain for future work. First, evaluating EvoSkill across a broader range of domains will help characterize which skills emerge as *domain-general* (e.g., search persistence, verification protocols) versus *domain-specific* (e.g., treasury table extraction). Second, the modular structure of evolved skills opens the possibility of shared skill libraries, where skills discovered on one task can be composed and reused by other agents. Finally, our cross-model and cross-dataset transfer results suggest that skill quality depends heavily on the model and harness used during evolution; understanding and mitigating these dependencies is an important open problem.

References

[1] Agent Skills. 2025. Agent Skills Specification. https://agentskills.io/specification

[2] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv:2507.19457 [cs.CL] https://arxiv.org/abs/2507.19457

[3] Salaheddin Alzu'bi, Baran Nama, Arda Kaz, Anushri Eswaran, Weiyuan Chen, Sarvesh Khetan, Rishab Bala, Tu Vu, and Sewoong Oh. 2026. ROMA: Recursive Open Meta-Agent Framework for Long-Horizon Multi-Agent Systems. arXiv:2602.01848 [cs.AI] https://arxiv.org/abs/2602.01848

[4] Anomaly. 2025. OpenCode. https://github.com/anomalyco/opencode

[5] Anthropic. 2025. Anthropic skills documentation. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview

[6] Anthropic. 2026. Claude Code Overview. https://code.claude.com/docs/en/overview

[7] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint* (2024).

[8] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *The Twelfth International Conference on Learning Representations*.

[9] Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. 2025. Fact, Fetch, and Reason: A Unified Evaluation of Retrieval-Augmented Generation. arXiv:2409.12941 [cs.CL] https://arxiv.org/abs/2409.12941

[10] Yoonho Lee, Joseph Boen, and Chelsea Finn. 2025. Feedback Descent: Open-Ended Text Optimization via Pairwise Comparison. arXiv:2511.07919 [cs.LG] https://arxiv.org/abs/2511.07919

[11] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] https://arxiv.org/abs/2303.17651

[12] Alexander Novikov, NgÃćn VÅĺ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[13] OpenAI. 2026. Codex. https://openai.com/codex/

[14] Thinh Pham, Nguyen Nguyen, Pratibha Zunjare, Weiyuan Chen, Yu-Min Tseng, and Tu Vu. 2025. SealQA: Raising the Bar for Reasoning in Search-Augmented Language Models. arXiv:2506.01062 [cs.CL] https://arxiv.org/abs/2506.01062

[15] The Mosaic Research Team. 2025. Introducing OfficeQA: A Benchmark for EndâĂŘtoâĂŘEnd Grounded Reasoning. https://www.databricks.com/blog/introducing-officeqa-benchmark-end-to-end-grounded-reasoning Accessed: 2026-02-20.

[16] Tu Vu, Kalpesh Krishna, Salaheddin Alzubi, Chris Tar, Manaal Faruqui, and Yun-Hsuan Sung. 2024. Foundational Autoraters: Taming Large Language Models for Better Automatic Evaluation. arXiv:2407.10817 [cs.CL] https://arxiv.org/abs/2407.10817

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

[18] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] https://arxiv.org/abs/2407.16741

[19] Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus,

and Amelia Glaese. 2025. BrowseComp: A Simple Yet Challenging Benchmark for Browsing Agents. arXiv:2504.12516 [cs.CL] https://arxiv.org/abs/2504.12516

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

```python
#!/usr/bin/env python3
"""
Economic Time-Series Analysis Script

Performs inflation adjustment and linear regression on
    economic data.

Usage:
    python analyze_timeseries.py <data_file> [--output <
    output_file>]

Input JSON format:
{
    "nominal_values": [
        {"period": "1970-03", "value": 100.0},
        {"period": "1970-06", "value": 110.0}
    ],
    "cpi_values": [
        {"period": "1970-03", "value": 38.8},
        {"period": "1970-06", "value": 39.0}
    ],
    "base_period": "1970-03",
    "analysis_type": "linear_regression"
}

Output JSON format:
{
    "real_values": [
        {"period": "1970-03", "nominal": 100.0, "real":
    100.0},
        {"period": "1970-06", "nominal": 110.0, "real":
    109.44}
    ],
    "regression": {
        "slope": 44.00,
        "intercept": 231.52
    },
    "formatted_result": "[44.00, 231.52]"
}
"""

import json
import sys
from typing import Optional


def load_data(filepath: str) -> dict:
    """Load JSON data from file."""
    with open(filepath, 'r') as f:
        return json.load(f)


def validate_data(data: dict) -> tuple[bool, str]:
    """Validate input data structure."""
    required = ['nominal_values', 'cpi_values', '
    base_period']
    for field in required:
        if field not in data:
            return False, f"Missing required field: {
    field}"

    if not data['nominal_values']:
        return False, "nominal_values cannot be empty"
    if not data['cpi_values']:
        return False, "cpi_values cannot be empty"

    return True, ""


def get_cpi_for_period(cpi_values: list, period: str) ->
    Optional[float]:
    """Get CPI value for a specific period."""
    for item in cpi_values:
        if item['period'] == period:
            return float(item['value'])
    return None


def adjust_for_inflation(nominal_values: list, cpi_values:
    list, base_period: str) -> list:
    """
    Adjust nominal values for inflation.

    Formula: Real Value = Nominal x (CPI_base /
    CPI_current)
    """
    cpi_base = get_cpi_for_period(cpi_values, base_period
    )
    if cpi_base is None:
        raise ValueError(f"Base period CPI not found: {
    base_period}")

    real_values = []
    for item in nominal_values:
        period = item['period']
        nominal = float(item['value'])

        cpi_current = get_cpi_for_period(cpi_values,
    period)
        if cpi_current is None:
            raise ValueError(f"CPI not found for period:
    {period}")

        real = nominal * (cpi_base / cpi_current)
        real_values.append({
            'period': period,
            'nominal': nominal,
            'real': round(real, 2)
        })

    return real_values


def linear_regression(values: list) -> tuple[float, float
    ]:
    """
    Perform simple linear regression.

```

```python
    Uses period index (0, 1, 2, ...) as x-values.
    Returns (slope, intercept) rounded to 2 decimal places.
    """
    n = len(values)
    if n < 2:
        raise ValueError("Need at least 2 data points for regression")

    # x values are indices 0, 1, 2, ...
    # y values are the real (inflation-adjusted) values
    y_values = [v['real'] for v in values]

    # Calculate means
    x_mean = (n - 1) / 2  # mean of 0, 1, ..., n-1
    y_mean = sum(y_values) / n

    # Calculate slope: sum((x-x_mean)(y-y_mean)) / sum((x-x_mean)^2)
    numerator = sum((i - x_mean) * (y - y_mean) for i, y in enumerate(y_values))
    denominator = sum((i - x_mean) ** 2 for i in range(n))

    if denominator == 0:
        slope = 0.0
    else:
        slope = numerator / denominator

    # Calculate intercept
    intercept = y_mean - slope * x_mean

    return round(slope, 2), round(intercept, 2)


def analyze(data: dict) -> dict:
    """Main analysis function."""
    # Validate
    valid, error = validate_data(data)
    if not valid:
        raise ValueError(error)

    # Adjust for inflation
    real_values = adjust_for_inflation(
        data['nominal_values'],
        data['cpi_values'],
        data['base_period']
    )

    # Perform analysis
    analysis_type = data.get('analysis_type', 'linear_regression')

    result = {
        'real_values': real_values
    }

    if analysis_type == 'linear_regression':
        slope, intercept = linear_regression(real_values)
        result['regression'] = {
            'slope': slope,
            'intercept': intercept
        }
        result['formatted_result'] = f"[{slope:.2f}, {intercept:.2f}]"

    return result


def main():
    if len(sys.argv) < 2:
        print("Usage: python analyze_timeseries.py <data_file> [--output <output_file>]")
        print("\nExample:")
        print("  python analyze_timeseries.py input.json")
        print("  python analyze_timeseries.py input.json --output results.json")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = None

    if '--output' in sys.argv:
        idx = sys.argv.index('--output')
        if idx + 1 < len(sys.argv):
            output_file = sys.argv[idx + 1]

    try:
        data = load_data(input_file)
        result = analyze(data)

        output = json.dumps(result, indent=2)

        if output_file:
            with open(output_file, 'w') as f:
                f.write(output)
            print(f"Results written to: {output_file}")
        else:
            print(output)

        # Print formatted result for easy extraction
        if 'formatted_result' in result:
            print(f"\nFormatted Answer: {result['formatted_result']}")

    except Exception as e:
        print(f"Error: {e}", file=sys.stderr)
        sys.exit(1)


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

A.2 SealQA

The following *search-persistence-protocol* skill contains comprehensive instructions for web-search for conclusive answers where conflicting sources may exist. SealQA is considered a challenging benchmark for many multi-agent systems and therefore provides a highly effective environment for demonstrating the practical value of this approach.

skills/search-persistence-protocol/SKILL.md

```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 Agent Prompts

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

B.1 Proposer

Proposer Agent 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>
"""
```

B.2 Skill-Builder

Skill Builder Agent 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.
"""
```

B.3 Auto-Grader

We use the default LLM-as-a-judge [16] template provided in the original SealQA task for all auto-grading tasks.

LLM-Judge 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 = 1
     e-9`.
```

C Scoring & 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.

D 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.

E GEPA Implementation

We integrate GEPA into our framework by wrapping the `dspy.GEPA` optimizer in a `GEPALoop` class that shares the same data splits, scoring function, and evaluation interface as EvoSkill's `SelfImprovingLoop`. The module is initialized with the same static base system prompt used by EvoSkill's Executor agent, appended with an empty skills placeholder; GEPA then evolves the instructions field, which serves as the skills container. We use a `ChainOfThought` module for question-answering tasks, with the same model serving as both the student and reflection model to match EvoSkill's configuration. Note that unlike all other experiments, the GEPA comparison does not use Opus 4.6 for evolution, since GEPA requires using the same model throughout its optimization loop. After the improvement loop completes, the best optimized prompt is exported and evaluation is performed on the held-out test set using that prompt, following the same protocol as EvoSkill to ensure a fair comparison.

F Further Ablation

In this section we provide a more in-depth analysis of different artifacts of the EvoSkill framework.

F.1 Qualitative Analysis of Discovered Skills

To illustrate the nature of skills that EvoSkill discovers, we highlight two representative examples from the evolved skill library:

Data Extraction Verification. EvoSkill discovered a skill enforcing a rigorous protocol for extracting numerical data from Treasury Bulletin tables. The skill is triggered whenever the agent extracts any value from parsed tables, and addresses concrete failure modes identified during evolution: adjacent cell misreads, wrong metric selection, and incorrect time granularity. This skill emerged directly from the Proposer's analysis of cases where the agent retrieved values from neighboring cells or confused similar-sounding metrics.

Quantitative Analysis Methodology. A second skill provides structured methodology guidance for quantitative financial analysis, including risk calculations, forecasting, currency conversion, and statistical inference. The skill enforces mandatory validation checkpoints before computation, preventing systematic errors from wrong data transformations, date misalignment, or confusion between sample and population statistics.

These examples illustrate that EvoSkill discovers interpretable, domain-relevant skills that target specific failure modes rather than producing opaque optimizations. Additional examples of discovered skills are provided in Appendix A.

G AI Disclosure

The authors acknowledge the use of large language models (LLMs) in the preparation of this work, specifically for the following:

- **Content Generation:** Assisting in drafting and refining sections of the manuscript.
- **Data and Visualization:** Generating research plots.
- **Evaluation:** Supporting the evaluative framework and the analysis of experimental results.