

POLARIS: A Gödel Agent Framework for Small Language Models through Experience-Abstracted Policy Repair

Aditya Kakade, Vivek Srivastava, Shirish Karande

TCS Research, India

{aditya.kakade, srivastava.vivek2, shirish.karande}@tcs.com

Abstract

Gödel agent realize recursive self-improvement: an agent inspects its own policy and traces and then modifies that policy in a tested loop. We introduce POLARIS, a Gödel agent for compact models that performs policy repair via experience abstraction, turning failures into policy updates through a structured cycle of analysis, strategy formation, abstraction, and minimal code patch repair with conservative checks. Unlike response level self correction or parameter tuning, POLARIS makes policy level changes with small, auditable patches that persist in the policy and are reused on unseen instances within each benchmark. As part of the loop, the agent engages in meta reasoning: it explains its errors, proposes concrete revisions to its own policy, and then updates the policy. To enable cumulative policy refinement, we introduce experience abstraction, which distills failures into compact, reusable strategies that transfer to unseen instances. On MGSM, DROP, GPQA, and LitBench (covering arithmetic reasoning, compositional inference, graduate-level problem solving, and creative writing evaluation), a 7-billion-parameter model equipped with POLARIS achieves consistent gains over the base policy and competitive baselines.

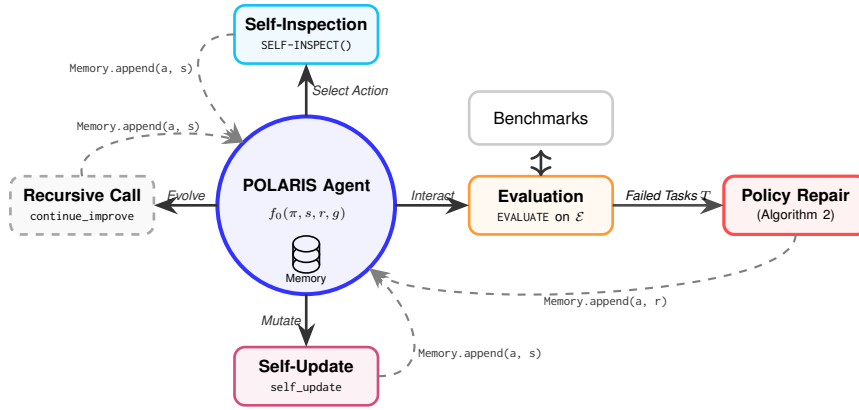
1 Introduction

Modern language agents improve in several ways, including response-level correction through critique and refinement, e.g. self-improvement either optimises responses—via reasoning and acting (Re-Act) (Yao et al., 2023), verbal reinforcement learning (Reflexion) (Shinn et al., 2023), iterative self editing (Self-Refine) (Madaan et al., 2023), tool interactive critique (CRITIC) (Gou et al., 2023), and self debugging (Chen et al., 2023). Alternatively improvement is achieved with parameter updates using task arithmetic (Ilharco et al., 2022),

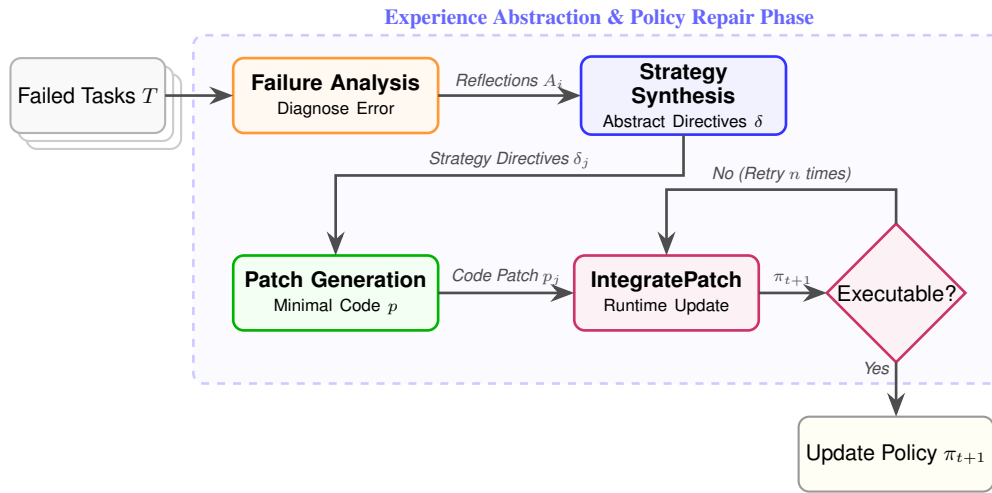
targeted knowledge edits and mass edits (Meng et al., 2022; Meng et al.), and broader editing surveys (Wang et al., 2024). While effective, these approaches often make it hard to localize what changed and where the change resides: was it an instance-specific, or persistent update that is useful across all new instances.

A natural way to make improvements persistent is to treat the agent’s policy as an explicit object that can be inspected and revised. Gödel Agents (Yin et al., 2024) formalize this idea as recursive self-improvement: the agent inspects its own policy and execution traces, and updates the policy in a tested loop (against a benchmark). The work by (Yin et al., 2024) provides an LLM based practical framework which achieves recursive self-improvement with run-time code mutation. However, directly instantiating Gödel agent style self-improvement can be resource intensive. In our initial attempts to adapt the Gödel Agent framework to a 7B model, runs frequently failed due to out-of-memory and tool-call errors before completion of execution. A key reason is the context growth: the framework retains multiple validation samples and multiple prior evolution steps in memory to support reflection, which increases context length and computational overhead after each iteration. This motivates our approach.

We introduce POLARIS to make recursive policy repair feasible under the constraint of working with smaller models. POLARIS performs policy repair via experience abstraction: failures are analyzed and generalized into compact reusable repair strategies, which are rendered as minimal code patches and integrated into the current policy with conservative checks and bounded retries. POLARIS controls context growth, while retaining traceability of the learned updates, by limiting the number of failed examples that are retained in memory for purpose of reflection along with the reduced tool-call history.



(a) Recursive self-improvement cycle (refer Algorithm 1).



(b) Policy repair module (refer Algorithm 2).

Figure 1: **Architectural overview of POLARIS.** (a) **Recursive self-improvement cycle:** The agent selects actions based on its policy and goals, storing outputs and reasoning traces in Memory. Evaluation collects N failed tasks from the validation set, triggering the Policy Repair module. (b) **Policy repair cycle:** Through experience abstraction, the agent performs *Failure Analysis* on the N tasks, distills reusable strategies in *Strategy Synthesis*, generates minimal code patches, and integrates them into the current policy. A candidate version is execution-checked, and if valid, applied via runtime code mutation.

Contributions.

- We introduce **POLARIS**, a framework that transforms failures into validated policy updates through analysis, synthesis, abstraction, and repair.
- We highlight the challenges with Gödel Agent (Yin et al., 2024) for SLMs in resource-constrained setting. We demonstrate that **recursive self-improvement is viable with SLMs**, reducing dependency on very large LLMs.
- We empirically validate our approach on MGSM (Shi et al., 2023), DROP (Dua et al., 2019), GPQA (Rein et al., 2024), and Lit-Bench (Fein et al., 2025) demonstrating consistent performance gains and interpretable improvements in capabilities.

We emphasize that both the instantiation of the

Gödel Agents framework (Yin et al., 2024) and our implementation of the POLARIS framework, which builds upon it, support runtime updates. This capability offers a significant practical advantage in post-deployment scenarios compared to hand-designed agents. Although neither the evaluation presented in this paper nor that in (Yin et al., 2024) addresses open-ended learning, an implementation that enables in-situ updates is likely to be particularly valuable in interactive environments where tool interfaces, data formats, or operational constraints evolve over time. The overall architecture of the POLARIS and its Policy repair mechanism are depicted in Figure 1.

2 Related Work

The concept of recursive self-improvement has deep roots in AI theory. Good (Good, 1966) speculated on the possibility of an “intelligence explosion” initiated by systems capable of enhancing their own cognitive processes. Schmidhuber (Schmidhuber, 2007) later formalized this notion in the Gödel Machine, a theoretical construct that can provably rewrite its own code if it can prove the modification improves its performance.

While elegant, Gödel Machines remain largely aspirational: exhaustive proof search is computationally infeasible, and no practical instantiation has been achieved. Nevertheless, this line of work provides the conceptual foundation for subsequent explorations of self-improving agents. Most notably, the Gödel Agent framework (Yin et al., 2024) demonstrates how large language models (LLMs) can engage in self-referential reasoning to repair and enhance their own policies. While these developments provide an important proof of concept, they remain largely tied to frontier-scale LLMs, leaving open the question of whether smaller models can sustain recursive improvement under resource constraints. Our work addresses this gap by extending self-referential frameworks to smaller language models (SLMs) such as Qwen2.5-7B, and by proposing a principled mechanism for experience abstraction and policy repair that enables continual self-improvement.

Research on reflection-driven language agents provides another strand of inspiration. Approaches such as ReAct (Yao et al., 2023), Reflexion (Shinn et al., 2023), and Self-Refine (Madaan et al., 2023) show that iterative feedback and self-critique can substantially improve reasoning and task performance. Extensions like CRITIC (Gou et al., 2023) and self-debugging strategies (Chen et al., 2023) further emphasize the value of embedding correction mechanisms into the agent loop. These works, however, primarily focus on improving responses within tasks. POLARIS builds on their insight but moves beyond single-instance correction by abstracting from task-level failures, synthesizing reusable strategies, and integrating them back into the policy to enable cumulative improvement.

A complementary line of work investigates direct editing of model representations. Techniques for localizing and modifying factual associations (Meng et al., 2022), performing mass edits in transformers (Meng et al., 2023), and leveraging task

arithmetic (Ilharco et al., 2022) illustrate that targeted modifications can shift model behavior without retraining. Surveys such as Wang et al. (Wang et al., 2024) summarize this rapidly expanding literature. Compared to such parameter-centric approaches, POLARIS adopts a higher-level repair process, emphasizing strategy abstraction and policy refinement rather than surgical edits to weights.

The idea of repair also resonates with research in automated program repair (APR), where systems diagnose errors and generate patches to improve external code bases. Classical bibliographies (Monperrus, 2018) and recent surveys on LLM-driven APR (Zhang et al., 2024) reveal striking parallels to self-improvement in language agents. Our contribution differs in that repair is applied not to external programs but to the agent’s own evolving policy, thereby blurring the line between debugging and learning.

We also note the emergence of evolutionary Gödel frameworks such as the Darwin Gödel Machine Zhang et al. (2025), which perform open-ended search by maintaining an archive of diverse agent variants and evaluating them through empirical performance on coding benchmarks. These evolutionary methods explore many candidate agents in parallel, in contrast to POLARIS, which focuses on iterative refinement of a single policy under tight compute and memory constraints. The approaches are complementary: population-based evolutionary search suits high-budget settings, while POLARIS targets resource-constrained SLM deployments where single-policy improvement is more practical.

Finally, our work is informed by the tradition of open-ended learning, which treats novelty, diversity, and complexity as drivers of continual progress. Theories of novelty search (Lehman and Stanley, 2011; Stanley and Lehman, 2015) and systems like POET (Wang et al., 2019) illustrate how adaptive agents coupled with evolving environments can yield unbounded improvement. Meta-learning (Finn et al., 2017) and hierarchical reinforcement learning (Sutton et al., 1999; Bacon et al., 2017) further demonstrate the importance of abstraction and reuse in sustaining adaptability. Classical open-ended learning pursues sustained novelty, typically via intrinsic objectives or co-evolving environments. POLARIS is conceptually adjacent in its emphasis on cumulative abstraction and reuse, but our study is bounded to fixed task distributions and external evaluation (see Appendix,

```

=====
Policy Example
=====

def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.7,
        num_of_response=1,
        role="math expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step.\n"
            "2. The answer MUST be an integer.\n"
        ).strip(),
    )
    return_dict = response[0]
-   # Explicitly round the answer to the nearest integer and convert to string
+
+   # Break down the problem and validate each part
+   parts = return_dict["reasoning"].split("\n")
+   for part in parts:
+       if "Step" in part:
+           step = part.split(": ")[1]
+           result = agent.action_call_json_format_llm(
+               messages=[{"role": "user", "content": f"Validate the following step: {step}"}],
+               temperature=0.7,
+               num_of_response=1,
+               role="logic checker",
+               return_dict_keys=["validation_result"],
+               requirements="1. Please provide a clear validation result.",
+           )
+           if result[0]["validation_result"] != "Valid":
+               return_dict["reasoning"] += f"\n{step} is invalid."
+
+   return_dict["answer"] = str(round(float(return_dict.get("answer", ""))))
    return return_dict

```

Figure 2: Policy update example on the MGSM dataset from experiments with the Qwen2.5-7B-Instruct model. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of the logic to break down the problem and validate each part while deleting the comment for post-processing the response.

Section A, for a detailed discussion and positioning within the literature on open-ended exploration.)

3 Our Approach

3.1 Gödel Agent for Recursive Self-Improvement

Gödel Agent (Yin et al., 2024) introduces a self-referential framework that realizes recursive self-improvement in LLM-based agents. The framework enables an agent not only to modify its task-execution policy π but also to revise the meta-level improvement logic I that determines how these modifications are generated and applied.

Given an environment \mathcal{E} and a utility function $U(\mathcal{E}, \pi)$, the agent repeatedly executes, evaluates, and refines its own policy and improvement logic through four key procedures.:

1. **Introspection (Self-Inspect):** The agent analyzes its internal architecture, including code modules, reasoning traces, and historical performance data. This process yields an explicit representation of the agent’s current capabilities and limitations.
2. **Execution (Interact):** During execution, the policy π operates as an LLM-based reasoning and acting module that engages with the environment through natural language interactions, tool use, and task-specific actions. The agent

records both its intermediate reasoning traces and environmental responses, which serve as empirical evidence for evaluating and refining its subsequent behavior.

3. **Self-Modification (Self-Improve):** Using the improvement logic I , the agent evaluates its performance and proposes candidate code edits or rewrites. These modifications may target the policy π to enhance problem-solving behavior or the improvement logic I itself to refine the way updates are reasoned about. Large language models serve as the generative engine for proposing, critiquing, and verifying such modifications.
4. **Recursive Continuation (Continue-Improve):** After each modification is integrated, the agent re-enters the introspection phase. This recursive loop allows both π and I to evolve jointly, producing progressively more abstract forms of self-repair and adaptation.

A central technical innovation in Gödel Agent is its use of *runtime code mutation*, implemented through mechanisms that enable modification of executable components during operation. This capability allows the agent to test, validate, and revert modifications dynamically without full retraining, supporting stable iterative improvement.

Algorithm 1 Recursive Self-Improvement of Gödel Agent

Input: Initial agent policy π_0 , initial decision function f_0 , goal g , environment state E , action of the agent a_i , memory of the agent *Memory*, policy feedback r , policy performance assessment function *EVALUATE*, self code reading function *SELF_INSPECT*, policy repair function *REPAIR_POLICY*

Output: Improved policy π , final agent state s

```

1  $\delta \leftarrow \emptyset$ 
2 Memory  $\leftarrow \emptyset$ 
3  $s \leftarrow \text{SELF\_INSPECT}()$ 
4  $r, \mathcal{T} \leftarrow \text{EVALUATE}(\pi_0, E)$ 
5  $\pi, s \leftarrow \text{REPAIR\_POLICY}(\pi, s, \mathcal{T}, a, r)$ 
6  $\pi, s \leftarrow \text{SELF\_IMPROVE}(\pi_0, s, r, g)$ 
7 return  $\pi, s$ 

8 Function SELF_IMPROVE( $\pi, s, r, g$ ):
9    $a_1, \dots, a_n \leftarrow f_0(\pi, s, r, g)$ 
10  for  $a_i$  in  $a_1, \dots, a_n$  do
11     $\pi, s, r \leftarrow \text{EXECUTE}(\pi, s, r, a_i, E)$ 
12  return  $\pi, s$ 

13 Function EXECUTE( $\pi, s, r, a, E$ ):
14  switch  $a.name$  do
15    case self_state do
16       $s \leftarrow \text{SELF\_INSPECT}()$ 
17      Memory.append( $a, s$ )
18    case interact do
19       $r, \mathcal{T} \leftarrow \text{EVALUATE}(\pi, E)$ 
20      Memory.append( $a, r$ )
21       $\pi, s \leftarrow \text{REPAIR\_POLICY}(\pi, s, \mathcal{T}, a, r)$ 
22    case self_update do
23       $\pi, s \leftarrow a.code$ 
24      Memory.append( $a, s$ )
25    case continue_improve do
26       $\pi, s \leftarrow \text{SELF\_IMPROVE}(\pi, s, r, g)$ 
27      Memory.append( $a, s$ )
28  return  $\pi, s, r$ 

```

Algorithm 2 Updating Agent’s Policy with POLARIS

Input: Current agent policy π_t , current agent state s , list of failed task samples \mathcal{T} , agent action a , policy feedback r

Output: Improved policy π_{t+1} , agent state s

```

1 Function REPAIR_POLICY( $\pi_t, s, \mathcal{T}, a, r$ ):
2    $A \leftarrow \emptyset$ 
3   foreach  $\tau_i$  in  $\mathcal{T}$  do
4      $A_i \leftarrow \text{AnalyzeFailure}(\pi_t, s, \tau_i)$ 
5      $A \leftarrow A \cup \{A_i\}$ 
6    $\delta \leftarrow \delta \cup \{\text{StrategySynthesis}(\pi_t, s, A)\}$ 
7    $p \leftarrow \text{PatchGeneration}(\pi_t, s, \delta)$ 
8    $n \leftarrow 3$ 
9    $\pi_{t+1}, s \leftarrow \text{IntegratePatch}(\pi_t, s, p, n, a, r)$ 
10  return  $\pi_{t+1}, s$ 

11 Function IntegratePatch( $\pi_t, s, p, n, a, r$ ):
12   $\pi_{t+1} \leftarrow \text{UpdatePolicy}(\pi_t, s, p)$ 
13  if  $\pi_{t+1}$  exists then
14    if  $\pi_{t+1}$  is not executable &  $n > 0$  then
15       $\pi_{t+1}, s \leftarrow \text{IntegratePatch}(\pi_t, s, p, n - 1)$ 
16     $\pi_{t+1}, s \leftarrow \text{IntegratePatch}(\pi_t, s, p, n, a, r)$ 
17  return  $\pi_{t+1}, s$ 

```

3.2 POLARIS: Policy Repair through Experience Abstraction

POLARIS implements recursive self-improvement in small language models through a structured cycle of reflection, abstraction, and repair, converting execution failures into validated code-level updates while preserving full traceability in the agent’s *Memory*. Each cycle follows the operators defined in **Algorithms 1 and 2** and the prompt templates

shown in **Figures 6–9**. The agent executes a mutable policy π_t on a validation set \mathcal{D} , records its behavior and outcomes, and integrates validated updates to yield a refined policy $\pi_{t+1} = \pi_t \oplus \Delta\pi$.

Failure Analysis. Executing π_t on \mathcal{D} produces a set of failed instances

$$\mathcal{T} = \{\tau_i\}$$

where each τ_i contains the input, the agent’s reasoning trace, the predicted output, and the reference answer. For every τ_i , the agent invokes `AnalyzeFailure` (Algorithm 2; Figure 6), a self-reflection operator that generates a structured record

$$A_i = (\textit{diagnosis}_i, \textit{revision}_i, \textit{prevention}_i)$$

The *diagnosis* identifies the cause of error in the policy’s reasoning or control flow, the *revision plan* proposes targeted adjustments at the code or rule level, and the *prevention rule* generalizes these insights for future iterations. Each reflection A_i is appended to Memory, forming a repository of interpretable experience from which higher-level repair strategies are derived.

Strategy Synthesis. The `StrategySynthesis` operator (Algorithm 2; Figure 7) abstracts across reflections $A = \{A_i\}$ to produce a compact set of reusable directives

$$\delta = \{\delta_j\}$$

Each δ_j captures a general repair principle such as decomposition, normalization, or control-flow adjustment that can resolve multiple failures. The prompt enforces novelty with respect to strategies stored in Memory and limits the agent to one or two well-formed strategies per cycle. By compressing instance-specific reflections into transferable repair abstractions, POLARIS transforms episodic feedback into policy-level adaptation.

Patch Generation. For each synthesized strategy δ_j , the `PatchGeneration` operator (Algorithm 2; Figure 8) instantiates a minimal code patch p_j . Each patch modifies only the lines required to implement δ_j and excludes any explanatory text. A lightweight validator checks syntax and formatting before a patch enters integration. The resulting patch set is denoted $\mathcal{P} = \{p_j\}$. Emphasizing locality and minimality ensures that every modification remains interpretable and that the agent’s policy evolves through small, verifiable updates.

Patch Integration. Patch integration follows Algorithm 2 and Figure 9. Each patch in \mathcal{P} is applied through the `UpdatePolicy`(π_t, s, p) procedure to generate a temporary policy candidate. Integration is verified through syntactic and execution checks rather than direct performance evaluation. If a patch fails, the agent retries up to a fixed bound

(three times by default). Persistent failures result in the patch and its context being archived in Memory for potential later analysis. After integration, the updated policy π_{t+1} proceeds to the next validation phase, where its performance effects are naturally observed. Memory retains all artifacts from the cycle, including reflections, strategies, patches, and integration results, providing continuity and preventing redundant proposals.

4 Experiments

We evaluate POLARIS on MGSM, DROP, and GPQA, covering arithmetic, discrete, and advanced factual reasoning. Additionally, we include LitBench, an open-domain benchmark for creative writing that tests stylistic preference modeling, narrative coherence, and open-ended reasoning.

For MGSM and DROP, we use 50 validation and 250 test samples; for GPQA, 20 validation and 100 test samples; and for LitBench, 20 validation and 250 test samples. For MGSM and GPQA, we report accuracy with 95% bootstrap confidence intervals, while for DROP we report macro F1 score due to its span-selection format. For LitBench, we report accuracy based on preferred-response selection.

	MGSM	DROP	GPQA	LitBench
Successful	0	1	0	0
No improvement	5	2	5	1
Unsuccessful	0	2	0	4
Total	5	5	5	5

(a) $k=3$

	MGSM	DROP	GPQA	LitBench
Successful	0	0	0	0
No improvement	0	0	0	2
Unsuccessful	5	5	5	3
Total	5	5	5	5

(b) $k=5$

Table 1: A summary of various runs of Gödel Agent (Yin et al., 2024) using Qwen2.5-7B-Instruct model in two different settings: (a) $k = 3$ along with three prior tool-call messages in memory (instead of 10), and (b) $k = 5$ along with five prior tool-call messages in memory (instead of 10).

All experiments are conducted using QWEN2.5-7B-INSTRUCT (Team, 2024) on two NVIDIA V100 GPUs (32 GB each). Each run is allowed to evolve autonomously for 10 hours, rather than a fixed number of steps, to observe natural variations in the number of policy-repair iterations. We record all intermediate reflections and policy modifications for analysis. We define a hyperparameter N , **the number of failed-task samples in the vali-**

	MGSM	DROP	GPQA	LitBench
Successful	5	3	4	6
No improvement	1	2	4	1
Unsuccessful	4	5	2	3
Total	10	10	10	10

(a) $N=3$

	MGSM	DROP	GPQA	LitBench
Successful	4	3	5	5
No improvement	1	1	2	1
Unsuccessful	5	6	3	4
Total	10	10	10	10

(b) $N=5$

Table 2: A summary of various runs of POLARIS on datasets using the Qwen2.5-7B-Instruct model. For policy repair and experience abstraction, we consider a set of N failed instances from the validation set of each dataset.

dition set that are used for introspective reflection. We test with $N=3$ and $N=5$ to study the trade-off between reflection depth and stability.

Initial attempts to adapt the Gödel Agent framework to SLMs led to frequent out-of-memory (OOM) and tool-call errors before completing the 10-hour execution window. This issue stems from the design of the Gödel Agent, which relies on large-context models. After each policy evaluation, the framework stores 20 validation samples, including task inputs, model responses, and feedback, in the agent’s memory. It also retains the last ten evolution steps to maintain historical reasoning traces. While this design enriches the context for reflection, it rapidly increases context length after each iteration, significantly raising computational overhead. To address this challenge, the experience abstraction mechanism requires fewer validation samples (N) for meta-reasoning along with reduced number of messages in the memory i.e., six instead of ten.

To ensure structured outputs, we employ one-shot prompting and a lightweight *helper function* that enforces valid JSON output during evaluation. This helper does not interfere with reasoning or evolution processes. We adapt the goal prompt of the agent from (Yin et al., 2024) with the requirements for our experimental setting (refer to Section B in the Appendix).

We perform ten independent runs on each dataset. We further categorize these runs as: (1) **Successful**—policy update improves test performance; (2) **No Improvement**—policy update succeeds but does not improve performance; and (3) **Unsuccessful**—run fails due to OOM, infinite loops, or hallucinated tool calls. Furthermore, we

consider two baselines:

1. **Chain-of-Thought Self-Consistency (COT-SC)** (Wang et al.): Five reasoning paths are sampled per query, and the most frequent answer is chosen. We use the same validation and test splits as our experiments to report the performance. It is one of the best performing baselines reported in (Yin et al., 2024).
2. **Gödel Agent** (Yin et al., 2024): Direct replication with Qwen2.5-7B-Instruct led to repeated OOM failures before 10 hours. In Table 1, we present a summary of various runs with reduced number of tool-call messages in the history. We observe that all runs fail due to memory constraints, resulting in out-of-memory errors for $k=5$. We observe fewer OOM errors with $k=3$ and three prior tool-call messages in memory. However, as the context length is very short, the agent fails to improve over iterations and gets stuck in repetitive and hallucinated tool calls. Further decreasing the context to accommodate memory constraints would lead to highly uncertain and non-targeted behaviour of the Gödel agent. Hence, a trivial adaptation of prior work on Gödel Agent is infeasible under resource constraints.

5 Results and Analysis

Self-evolution under constrained setting is challenging: We summarize the runs of POLARIS in Table 2. In Figure 3 and the Appendix, we report performance across policy repair iterations alongside baselines. Evolution on MGSM, GPQA, and LitBench achieves a higher success rate than DROP, likely due to DROP’s larger context size, which increases susceptibility to out-of-memory (OOM) errors. Performance variability is greater for $N=3$ than $N=5$, and experience abstractions become more generic as N grows. These trends highlight the difficulty SLMs face in abstracting strategies from diverse self-reflections. A detailed analysis of failure modes reveals two dominant factors: (i) **limited meta-reasoning capability**, where SLMs fail to diagnose failure causes and repair policies, leading to noisy, non-progressive corrections; and (ii) **poor tool-calling capability and OOM errors**, driven by large accumulated context, hallucinated tool calls, redundant evaluations, or irrecoverable policy adjustments, consistent with prior reports on SLM limitations (Senel and Ozmen, 2025; Subramanian et al., 2025). Initial experiments showed

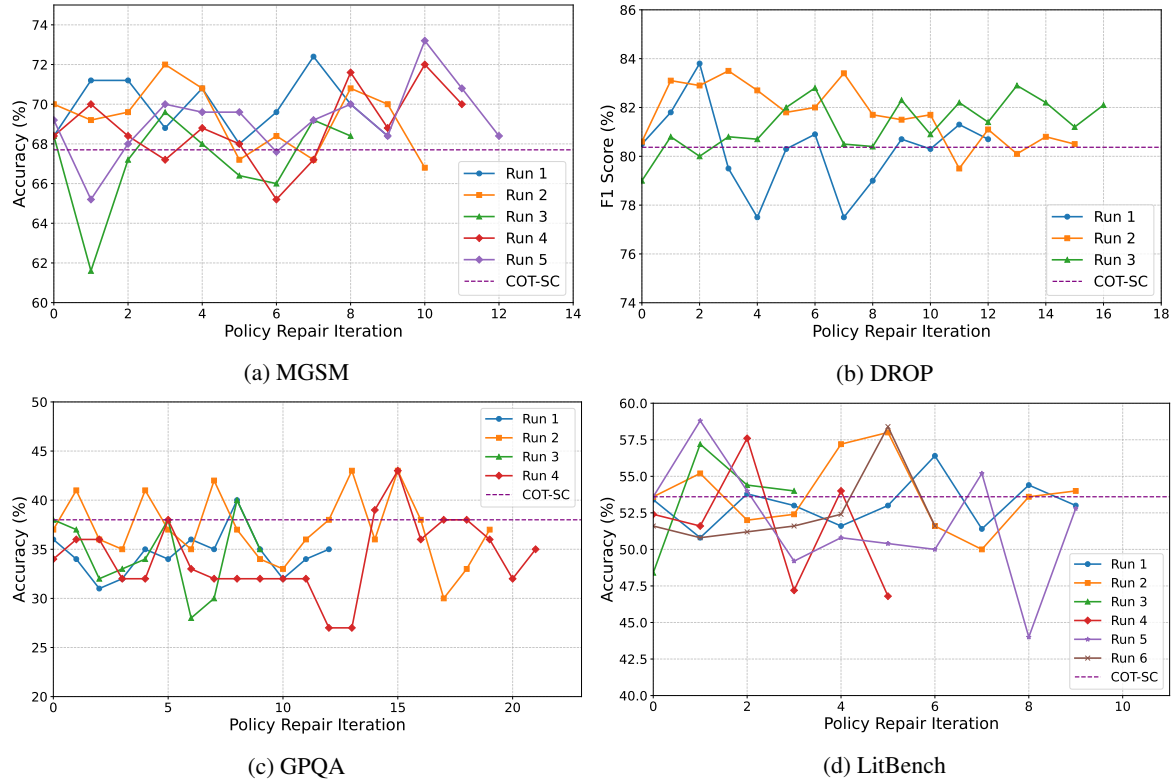


Figure 3: Successful evolution runs of POLARIS with performance improvement compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N=3$). Experiments conducted with the Qwen2.5-7B-Instruct model.

that models with comparable parameter counts such as DeepSeek-Coder-6.7B-Instruct (DeepSeek AI, 2024), Llama-3.1-8B-Instruct (Meta AI, 2024), and Mistral-7B-Instruct-v0.3 (Mistral AI, 2024) but with limited tool-calling and meta-reasoning capabilities consistently failed, underscoring the challenges faced by SLMs. Example failure traces for these models are provided in Figures 28–30 in the Appendix.

Policy repair through experience abstraction tends to capture solution complexity: We illustrate policy updates across datasets, showing how repairs introduce diverse enhancements such as complex instructions, post-processing steps, conditional logic, exception handling, and context-aware validation (Figures 17–29 in the Appendix). An example of such a policy update on the MGSM dataset is shown in Figure 2, where POLARIS introduces decomposition and validation logic while removing outdated post-processing steps. These observations highlight the role of experience abstraction in enabling nuanced, task-specific policy refinements. Additionally, the number of repair iterations varies across datasets, underscoring the need

for adaptive, localized behavior in self-improving agents.

Non-monotonic but consistent performance gain: We observe that the agent can recover from local performance minima across datasets (see Appendix, Section B.4, for a discussion on non-monotonic behavior). To quantify gains, we report the maximum relative improvement of our self-evolution framework over the COT-SC baseline across successful runs. For $N = 3$, improvements include +4.0% on MGSM, +3.9% on DROP, +9.0% on GPQA, and +8.8% on LitBench. At $N = 5$, trends remain consistent with +5.7% on DROP, +3.6% on MGSM, +9.0% on GPQA, and +5.2% on LitBench. These results underscore the effectiveness of POLARIS in resource-constrained settings.

Unsuccessful runs and the utility of POLARIS: Unsuccessful runs were caused by infrastructure issues such as GPU memory exhaustion, blocked library calls, or hallucinated tool invocations triggered by long debugging traces. These hallucinations typically occur when the agent repeatedly tries to fix a persistent error and the context becomes saturated with verbose logs. These failures

reflect limits of our experimental setup rather than weaknesses in the experience-abstraction mechanism. We intentionally omitted guardrails such as rollbacks, stronger static validation, and long-term memory stabilization to observe the raw dynamics of experience-driven policy evolution. Adding these stabilizers is a promising direction for reducing failure modes in future versions of POLARIS.

Impact of base model capability: We ran an additional experiment with the Qwen3-8B model (Qwen Team, 2025) using $N = 3$. Table 3 summarizes all runs, and Figure 4 shows the successful evolution trajectories. Compared to Qwen2.5-7B-Instruct, Qwen3-8B is slightly larger and includes a native “thinking” mode that strengthens its reasoning ability. This leads to a higher success rate and more stable improvement across datasets, indicating that as compact models advance, POLARIS will become increasingly effective for real-world use. Under the same experimental constraints, however, Qwen3-8B completes fewer iterations. Its default thinking mode generates far more tokens per call, increasing memory use and computation time. Although thinking is disabled during evaluation for fairness, it remains active during tool-use and policy-repair. The resulting overhead effectively reduces the iteration budget relative to Qwen2.5-7B-Instruct. Even so, Qwen3-8B achieves steady or improved performance on all datasets.

We observe a qualitatively different behavior when moving to a larger base model with greater available compute. In additional experiments with the devstral-small-2 (Mistral AI, 2025) model, POLARIS exhibits consistently stable evolution behavior, with very few unsuccessful runs, none of which were attributable to out-of-memory failures across all trials. This contrasts with earlier Qwen-based settings, where some evolution runs terminated unsuccessfully under tighter memory and iteration budgets. In the devstral setting, the rare unsuccessful runs that do occur can be traced to suboptimal policy updates early in the evolution process, which bias subsequent repairs and steer the trajectory toward a degraded policy. Importantly, this failure mode is algorithmic rather than resource-driven and could be mitigated using a simple rollback or checkpointing mechanism to revert destabilizing early updates. Notably, the POLARIS repair operators and reflection mechanism remain unchanged; the primary difference lies in the underlying model capacity and the increased

computational headroom available during evolution. These results indicate that many of the unsuccessful outcomes observed in low-budget settings are attributable to resource constraints rather than fundamental limitations of the evolution process itself. As compute budgets and base model capacity increase, POLARIS not only yields stronger policy improvements but also benefits from substantially improved stability, reinforcing its suitability for sustained long-horizon policy evolution as model scales continue to grow. Complete experimental details for the devstral-small-2 setting, along with tables summarizing evolution outcomes and plots of accuracy trajectories, are provided in Appendix B.1.

6 Conclusion

We introduced POLARIS, a framework for recursive self-improvement in small language models through structured, interpretable updates. Unlike prior approaches relying on large-model capacity and unconstrained self-rewrite, POLARIS employs a controlled repair cycle and supports runtime updates, enabling post-deployment adaptation. Empirical results demonstrate consistent gains without supervision or retraining, highlighting the feasibility of stable, traceable self-referential learning and its potential for controlled, open-ended improvement in evolving environments.

7 Limitations

POLARIS provides a practical approach to recursive self-improvement in small language models, yet some limitations remain. The reduced meta-reasoning capacity, smaller context windows, and limited tool-use capabilities of SLMs constrain the depth of self-reflection and the complexity of policy updates the agent can perform. Moreover, abstraction over larger and more diverse experience sets remains challenging for small language models, as limited context capacity constrains the agent’s ability to consolidate reflections into coherent, generalizable strategies. The dependency on human-designed prompt templates needs to be explored further, and automated prompt template generation is a promising direction for future work. Finally, while the iterative repair cycle supports continual refinement, it does not guarantee monotonic improvement and may increase computational overhead when repair attempts are frequent. These considerations do not undermine the framework’s

core contribution but highlight opportunities for extending POLARIS toward more expressive, tool-augmented, and stable self-improvement processes.

References

- Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31.
- Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.
- DeepSeek AI. 2024. Deepseek-coder-6.7b-instruct. <https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct>.
- Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378.
- Daniel Fein, Sebastian Russo, Violet Xiang, Kabir Jolly, Rafael Rafailov, and Nick Haber. 2025. Litbench: A benchmark and dataset for reliable evaluation of creative writing. *arXiv preprint arXiv:2507.00769*.
- Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR.
- Irving John Good. 1966. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pages 31–88. Elsevier.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.
- Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.
- Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hananeh Hajishirzi, and Ali Farhadi. 2022. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *ICLR*.
- Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.
- Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Thomas Jackson, Samuel Coward, and Jakob Nicolaus Foerster. 2024. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning*, pages 35104–35137. PMLR.
- Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in gpt. *Advances in neural information processing systems*, 35:17359–17372.
- Kevin Meng, Arnab Sen Sharma, Alex J Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer. In *The Eleventh International Conference on Learning Representations*.
- Kevin Meng, Arnab Sen Sharma, Alex J Andonian, Yonatan Belinkov, and David Bau. 2023. Mass-editing memory in a transformer. In *ICLR*.
- Meta AI. 2024. Meta llama 3.1 8b instruct. <https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct>.
- Mistral AI. 2024. Mistral-7b-instruct-v0.3. <https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3>.
- Mistral AI. 2025. Devstral small 2. <https://docs.mistral.ai>. Model card, released December 9, 2025. Accessed: March 2026.
- Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24.
- Andrew Patterson, Samuel Neumann, Martha White, and Adam White. 2024. Empirical design in reinforcement learning. *Journal of Machine Learning Research*, 25(318):1–63.
- Qwen Team. 2025. Qwen3-8b. <https://huggingface.co/Qwen/Qwen3-8B>.
- David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.
- Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel. 2021. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*.

- Jürgen Schmidhuber. 2007. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer.
- FA Senel and Hayri Baytan Ozmen. 2025. A comparative review of hallucination mitigation and performance improvement techniques in small language models. *Journal of Research and Design*, 2(1).
- Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, and 1 others. 2023. Language models are multilingual chain-of-thought reasoners. In *The Eleventh International Conference on Learning Representations*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.
- Olivier Sigaud, Gianluca Baldassarre, Cedric Colas, Stephane Doncieux, Richard Duro, Pierre-Yves Oudeyer, Nicolas Perrin-Gilbert, and Vieri Giuliano Santucci. 2023. A definition of open-ended learning problems for goal-conditioned agents. *arXiv preprint arXiv:2311.00344*.
- Kenneth O Stanley and Joel Lehman. 2015. Why greatness cannot be planned: The myth of the objective.
- Shreyas Subramanian, Vikram Elango, and Mecit Gungor. 2025. Small language models (slms) can still pack a punch: A survey. *arXiv preprint arXiv:2501.05465*.
- Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211.
- Qwen Team. 2024. [Qwen2.5: A party of foundation models](#).
- Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. 2019. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*.
- Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. 2024. Knowledge editing for large language models: A survey. *ACM Computing Surveys*, 57(3):1–37.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. 2024. G\ "odel agent: A self-referential agent framework for recursive self-improvement. *arXiv preprint arXiv:2410.04444*.
- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*.
- Quanjin Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A systematic literature review on large language models for automated program repair. *CoRR*.

A POLARIS and Open-Ended Exploration

In this work, our intent is not to claim fully unconstrained, artificial-life-style open-endedness (with endlessly generated new environments and goals), but rather to follow a now common usage in the Gödel-agent (Yin et al., 2024) and open-ended RL literature: open-endedness in the space of agent designs and internal strategies, evaluated on concrete benchmarks.

Concretely, in our framework, the Gödel-style agent is free to rewrite its own routines, abstractions, and self-improvement code, without a fixed meta-algorithm. This induces an effectively unbounded search space over how the agent reasons, decomposes problems, and organizes its computation. The external tasks (reasoning datasets with objective metrics) are kept fixed to provide a controlled testbed and reproducible measurement. This mirrors the original Gödel Agent work, which describes a self-evolving framework that freely decides its own routine, modules, and even the way to update them, yet evaluates on standard code-editing and reasoning benchmarks. Similarly, the Darwin Gödel Machine (Zhang et al., 2025) explicitly bills itself as open-ended evolution of self-improving agents, while its empirical evaluation is on benchmarks such as SWE-bench (Jimenez et al., 2024), the open-endedness lies in the archive and continual mutation of coding agents, not in unbounded task generation.

The broader open-endedness community also routinely combines open-ended agent/solution generation with concrete, fixed benchmarks. Position and survey papers define open-ended learning as a process that continually discovers new, diverse and increasingly capable solutions or “stepping stones” (policies, programs, strategies), but then instantiate this in specific testbeds to enable careful evaluation (Sigaud et al., 2023). In open-ended RL, tools such as MiniHack (Samvelyan et al., 2021) and Craftax (Matthews et al., 2024) are explicitly described as benchmarks for open-ended reinforcement learning, even though the underlying environments are specific games with well-defined reward functions: the open-endedness comes from the rich combinatorics of the environment and the space of emergent behaviors/tasks, not from an infinitely changing metric.

Our usage is aligned with this practice: we study open-ended self-improvement in agent space

(Gödel-style self-modification with abstractions), evaluated on fixed reasoning benchmarks that provide objective metrics and make comparisons to baselines possible. We have observed that performance can plateau after some number of self-improvement steps; however, this plateau is not due to an intrinsic saturation of the evaluation metric, but rather to the limitations of the current implementation e.g., the finite set and design of self-modification operators, the “imagination” of the agent in proposing more radical rewrites, and the finite budget of iterations we run. Conceptually, nothing in the framework prevents further exploration: richer operator libraries, more diverse abstraction schemes, or longer runs could allow the agent to escape such plateaus and continue discovering improvements, just as more sophisticated exploration mechanisms unlock further progress in open-ended RL benchmarks.

B Additional Experimental Details and Analysis

In Figure 10, we provide the goal prompt for the agent. We adapt the agent’s goal prompt from (Yin et al., 2024) and introduce instructions for small language models such as

- **action_adjust_logic:** Added “Do not do unnecessary changes” and clarified it may be used to create targeted *action functions* for the solver; original constraints (such as check imports/usages, do not change interfaces) remain.
- **Techniques block:** Replaced the brief hint with a concrete list: LLM Debate, Step-back Abstraction, Quality-Diversity, Dynamic Roles, Self-consistency (with num_of_response), Few-shots, Task Decomposition, Reflective Evaluation.
- **action_display_analysis:** Removed the low-score case-study requirement; added that `action_call_json_format_llm` can also perform analysis.
- **Reminder prompting to the agent:** Call `action_evaluate_on_task` *only after* modifying the solver via `action_adjust_logic`; multiple tools may be called when needed.

Furthermore, in Figure 11, we provide the helper agent prompt that helps correct the output format

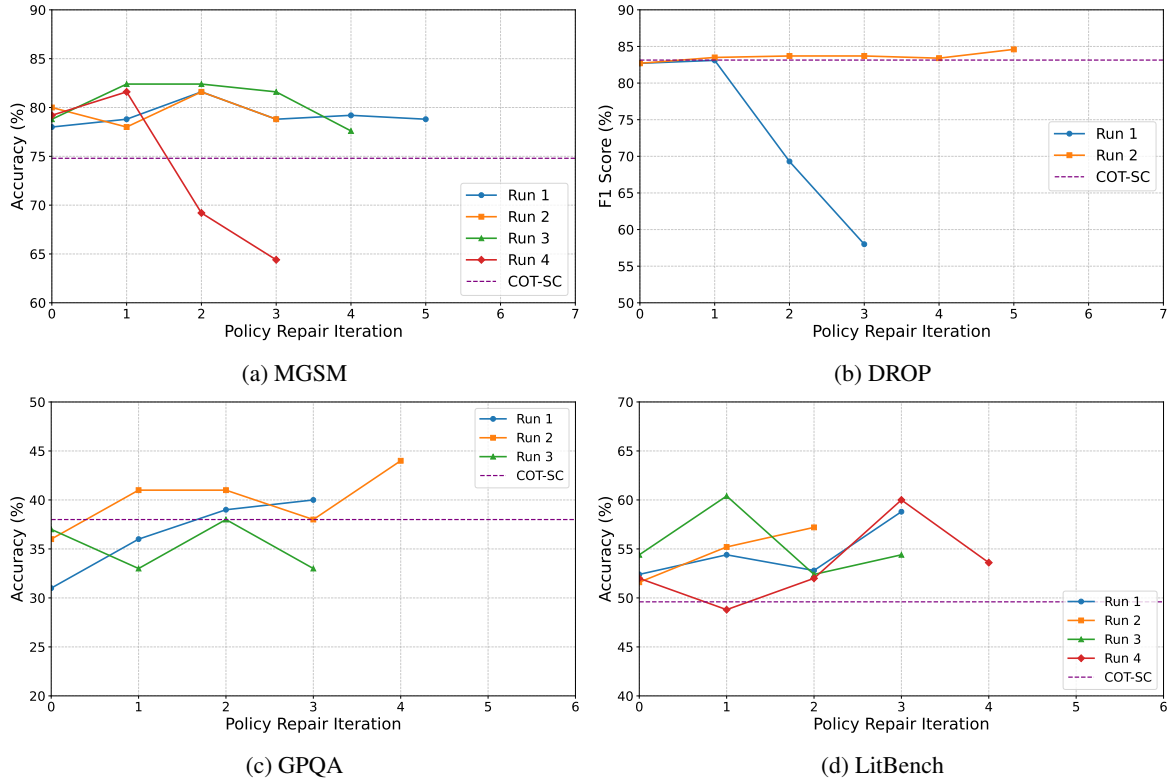


Figure 4: Successful evolution runs of POLARIS using Qwen3-8B model, with performance improvement compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N=3$).

$N = 3$	MGSM	DROP	GPQA	LitBench
Successful	4	2	3	4
No improvement	1	0	1	1
Unsuccessful	0	3	1	0
Total	5	5	5	5

(a) $N = 3$

Table 3: A summary of various runs of POLARIS on datasets using the Qwen3-8B model. For policy repair and experience abstraction, we consider a set of $N = 3$ failed instances from the validation set of each dataset.

to valid JSON during the evaluation of the policy. We provide examples in the prompt to obtain the target behaviour.

B.1 Experimental setup (devstral-small-2)

To evaluate POLARIS on a model outside the Qwen family, we conducted experiments using the devstral-small-2 (Mistral AI, 2025) model, a 24B-parameter instruction-tuned model from the Mistral family, using 4-bit quantization. Due to the higher resource demands of this model, we adopted a constrained evolution protocol following Gödel Agent (Yin et al., 2024), running a fixed budget of 30 evolution steps rather than our standard

10-hour evolution window. All experiments were executed on two NVIDIA RTX 6000 Ada GPUs with 48 GB memory each. We performed five independent POLARIS runs on the MGSM, DROP, GPQA and Litbench dataset, using $N = 3$ failed instances per reflection cycle for policy repair and experience abstraction. A summary of evolution outcomes across runs is reported in Table 4, and the corresponding accuracy trajectories over evolution steps are shown in Figure 5. Methods, repair operators, and validation procedures were kept identical to those used in the Qwen-based experiments.

$N = 3$	MGSM	DROP	GPQA	LitBench
Successful	2	2	2	4
No Improvement	2	2	2	1
Unsuccessful	1	1	1	0
Total	5	5	5	5

(a) $N = 3$

Table 4: A summary of various runs of POLARIS on datasets using the devstral-small-2 model. For policy repair and experience abstraction, we consider a set of $N = 3$ failed instances from the validation set of each dataset.

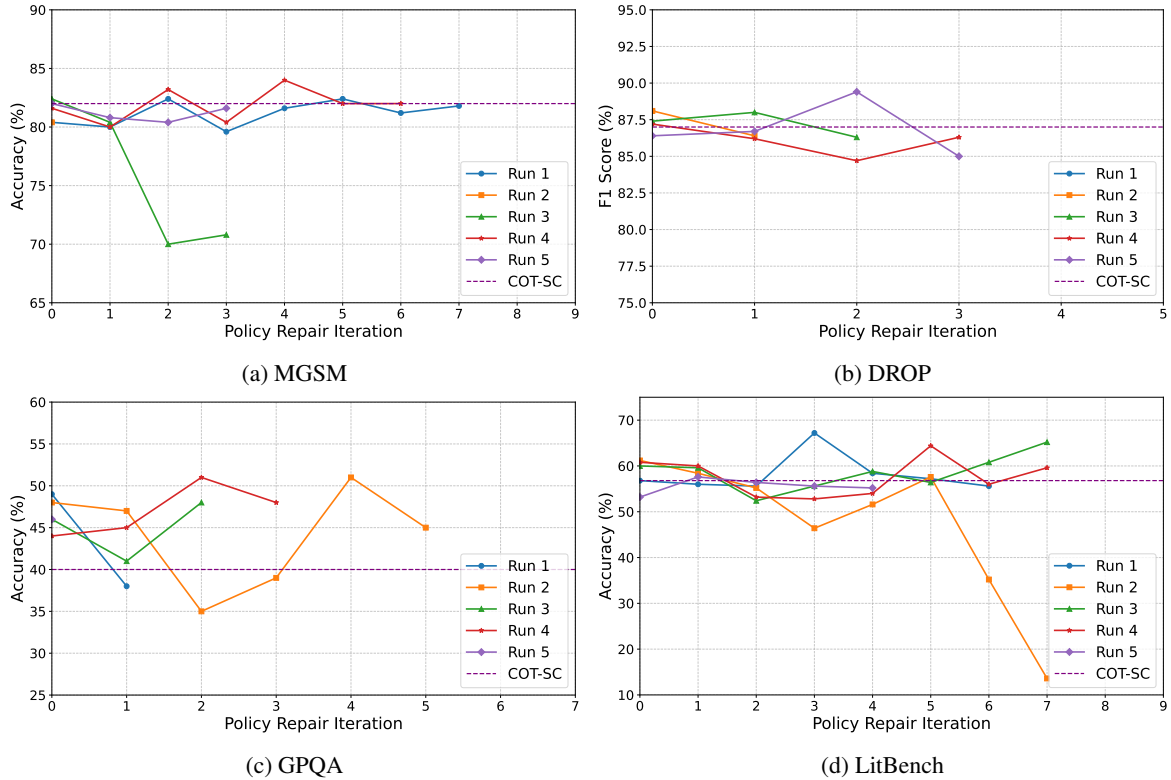


Figure 5: Evolution trajectories of POLARIS using the devstral-small-2 model. Each subplot shows the accuracy trajectories for all five independent evolution runs, illustrating stable policy improvements relative to the base policy and the CoT-SC baseline. Policy Repair Iteration 0 corresponds to the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N = 3$).

B.2 Example runs of POLARIS

In Figures 18, 17, 19, and 20, we present examples from different datasets illustrating the steps of the POLARIS algorithm. The initial policy fails to correctly answer the question. Failure analysis generates explanations, instructions, and advice for the agent. During strategy synthesis, recommendations are formulated to correct prior behavior by updating the policy. The corresponding patch is then integrated into the current policy, resulting in improved task performance on unseen instances.

B.3 Policy update examples

In Figures 21–29, we present example policy updates across datasets. We highlight changes to the current policy relative to the previous policy using green (additions) and red (deletions). The updates span multiple dimensions, including the addition of complex requirements, expert evaluators, data-type changes, conditional statements, exception handling, and specialized scoring parsers. These niche updates underscore the role of experience abstraction in enabling targeted policy refinements over iterations.

B.4 Variance in performance

In Figures 15 and 16, we present a consolidated view of performance variation for successful and no-improvement runs of POLARIS across datasets. We acknowledge that the raw reward/accuracy at each self-improvement step can fluctuate, and we do not claim monotone improvement of every intermediate candidate. However, this behavior is expected for algorithms that: (i) operate in open-ended search spaces, and (ii) deliberately explore large, non-local changes to the policy or code. Intermediate fluctuations in performance across repair iterations are expected because POLARIS explores a discrete policy space through program-level mutations. As in evolutionary and search-based optimization, individual candidate policies may transiently improve or degrade performance. Stability is measured by the best policy discovered over the repair cycle rather than monotonicity at each iteration.

Closely related settings such as deep reinforcement learning and open-ended RL are well known to exhibit high variance and occasional regressions even under fixed hyperparameters and environ-

Analyze Failures

You are analyzing why the current policy failed on a given task. Your goal is to identify the policy's shortcomings and propose actionable improvements.

Inputs:

- Question: {question}
- Your Reasoning: {reasoning}
- Your Answer: {answer}
- Correct Answer: {correct_answer}
- Policy: {current_policy}

Carefully reflect on why the policy produced the wrong result. **Your reflection must include three elements:**

1. **A clear explanation of the failure.** Examine how the policy's logic or structure caused the error.
2. **Step-by-step suggestions** on how the policy could be revised to solve the task.
3. **Advice to prevent similar failures** in the future.

Figure 6: Prompt for analyzing failures on task samples through self-reflection.

Strategy Synthesis

You are an expert AI engineer analyzing self-reflection on policy from multiple failed tasks.

Inputs:

- Reflections: {combined_reflections}
- Current Policy: {current_policy}
- Prior Strategies: {agent.prior_strategies}

Your task is to extract *1-2 new* generalizable and non-redundant policy improvement strategies from the task-level reflections.

Rules:

- **Do not repeat** or restate any of the previously extracted strategies.
- The strategy should **target the root cause** behind the failures observed in the reflections.
- It must be **reusable across tasks** and focused on policy improvements (not tied to one failure instance).
- Do not copy raw reflections; **abstract reflections into a reusable *insight***.
- Write this as if **giving coding instructions to another engineer**.
- Output only *1-2* new generalizable improvement strategies, written as **short, clear statements**.

Figure 7: Prompt for policy repair planning and abstraction. Agent synthesizes the generalized policy repair strategies based on the self-reflection on failed task samples on the current policy. It also considers the prior strategies to avoid redundancy.

ments.

Henderson et al. (2018) systematically document such instability and variance across seeds in stan-

Patch Generation

You are assisting in improving the current policy.

Inputs:

- Current Policy: {current_policy}
- Repair Strategies: {repair_strategies}

Your task:

- For each strategy, **propose a minimal **code patch**** to implement it.
- Show **only new or modified lines**, do not repeat unchanged code.
- **No explanations.**

Format your response as:

```
### Strategy: <chosen strategy>
### Patch: <only the modified or new lines of Python code>
```

Figure 8: Prompt for generating code patches from policy repair strategies.

dard deep RL benchmarks and argue that this variance is intrinsic to the methods rather than an implementation bug. Patterson et al. (2024) similarly emphasize that performance variation and instability are central empirical phenomena in modern RL, and that sound evaluation must explicitly account for them rather than expect smooth, monotone curves. Benchmarks designed for open-ended learning such as MiniHack Samvelyan et al. (2021) and Craftax Matthews et al. (2024) also explicitly highlight that exploratory, open-ended agents typically show jagged learning curves while still discovering significantly better policies than baselines.

In our setting, the key quantity is therefore the best-so-far performance of the agent, not the instantaneous performance of every transient candidate produced during self-improvement. Our experiments show that the final (or best-so-far) Gödel agent produced by our method consistently and substantially outperforms both the initial system and strong non-Gödel baselines. This is analogous to standard practice in anytime search and in reinforcement learning, where a potentially unstable inner learner is wrapped by an outer loop that always retains the best model found so far.

Practically, a user or deployment scenario would not expose intermediate, exploratory candidates. Instead, one would keep a “champion” model and only replace it when the self-improvement loop discovers a clearly superior “challenger” based on a stable evaluation protocol (a standard cham-

Update Policy

You are a coding assistant. Your task is to apply all the provided code patches to the current policy and return the fully updated version of the policy.

Current policy: {current_policy}

Rules:

- Insert or replace ONLY the lines shown in the patch.
- Keep ALL other lines of the policy unchanged.
- Do NOT remove or overwrite existing logic unless explicitly replaced by the patch.
- Ensure ALL patches are correctly integrated (e.g., imports, variables, helper functions must exist).
- If a patch introduces new logic that requires dependencies (imports, helper methods, variables), ADD them safely.
- Resolve conflicts so the final policy is consistent and executable.
- The updated policy MUST be logically correct, consistent, and error-free.
- Always return the FULL policy wrapped in: `“python <code patch here>”`

Figure 9: Prompt for integrating code patches into current policy.

pion–challenger pattern from RL and online learning). This yields a monotone non-decreasing performance profile for the deployed agent, even if the internal search process remains volatile and exploratory.

B.5 POLARIS runs with no improvement

In Figures 13 and 14, we present runs where the agent fails to surpass the performance of the base policy. Such cases are relatively rare compared to successful runs across datasets. Moreover, in most instances, the gap between the base policy and the best performance achieved over iterations is minimal. With a longer evolution horizon, we expect the agent to recover and improve performance. This behavior warrants further investigation using strategies such as occasional resets and the integration of pre-identified policy patches that are known to yield performance gains, providing targeted boosts during stagnation.

Goal prompt

You are a **self-evolving agent**, named `self_evolving_agent`, an instance of the 'Agent' class, in module 'agent_module', running within an active **Python runtime environment**. You have full access to global variables, functions and modules. Your primary goal is to continuously enhance your ability to solve tasks accurately and efficiently by dynamically reflecting environment and evolving your logic.

Core Capabilities

- + **Complete Autonomy**: Have **unrestricted access** to modify logic, run code and manipulate environment.
- + **Environment Interaction**: Interact with the environment by perceiving environment, reading or modifying or executing code and executing actions.
- + **Problem-Solving**: Apply creative algorithms or self-developed structures to tackle challenges when simple methods fall short, optimizing solutions effectively.
- + **Collaboration**: Leverage LLM to gather insights, refine strategies, correct errors, and solve complex problems.
- + **Error Handling**: Carefully analyze errors. When errors occur, troubleshoot systematically, and if a bug is persistent, backtrack, restore the original state, or find an alternative solution.

Core Methods

- + **evolve**: Continuously enhance performance by interacting with environment.
- + **execute_action(actions)**: Execute actions based on analysis or feedback.
- + **solver(agent_instance, task_input: str)**: Solve the target task using current 'agent_instance' capabilities, and objects created by `action_adjust_logic` and `action_run_code`, optimizing the process.

Guiding Principles

- + **Remember** that all functions are in module `agent_module`.
- + **action_adjust_logic**: Before modifying the code, make sure that each variable or function used is used and imported correctly to avoid errors. Do not do unnecessary changes. Do not change interface of any function. Can be used to create action functions for 'solver'.
- + **action_run_code**: Make sure that each variable or function used is used and imported correctly to avoid errors. ALL created objects in Python mode can be stored in environment. Can be used to create objects for 'solver', such as prompt. Can be used to import new module or external libraries and install external libraries.
- + **External Collaboration**: Seek external assistance via `action_call_json_format_llm` for logic refinement and new tool creation or `action_run_code` to execute code and then get and store the useful objects, like PROMPTS, that can be reused in 'solver'.
- + **action_evaluate_on_task**: Assess the performance of 'solver' ONLY after successfully modifying the logic of 'solver'.
- + **solver**: Is defined as `agent_module.solver`. The output MUST be a dictionary, and the final answer MUST be placed under the key "answer". For debugging, don't print, and instead return the debug information. When calling LLM, it must exclusively use `action_call_json_format_llm`. Can call `action_call_json_format_llm` multiple times and across multiple rounds in the solver to improve performance. If performance doesn't improve, explore alternative methods. When multiple outputs are required, set `num_of_response`, a parameter of `action_call_json_format_llm`, to the required number of outputs in the function. Additionally, can call different role-based LLMs by specifying and MUST specifying the role to further assist task-solving. For each key, if a specific format is required, such as int, float, enum or list, the requirements must specify the conditions.
- + **Explore techniques like**: **Large Language Model Debate**: Multiple models engage in a discussion to critique and refine responses, improving solution quality. **Step-back Abstraction**: Solving problems by shifting to a higher, more abstract perspective to simplify and break down complex tasks. **Quality-Diversity**: Focusing on generating diverse, high-quality solutions rather than exclusively optimizing one outcome. **Dynamic Assignment of Roles**: Assigning and adjusting roles among AI components dynamically to enhance task performance. **Self-consistency**: Ensure coherence by comparing multiple outputs and selecting the most consistent one. (Can try to increase `num_of_response` to get high score). **Few-shots**: Using few-shot learning to quickly adapt with minimal examples (can use valid examples), improving performance on new tasks through generalization. **Task Decomposition**: Dividing complex tasks into smaller subtasks, solving them individually, and reintegrating the solutions for overall task success. **Reflective Evaluation**: Reviewing performance after task completion to identify successes and failures, enabling continuous self-improvement. Can combine above techniques.
- + **action_display_analysis**: **Always analysis first before acting.** Analysis may include following things: reasonable plan about improving performance, error handling, other possible solving ideas. **If performance does not improve, conduct further analysis.** `action_call_json_format_llm` can also do analysis.
- + **Reminder**: Make sure you call `action_evaluate_on_task` ONLY after successfully modifying solver function's logic using `action_adjust_logic`. You can call Multiple tools at once.

Figure 10: Goal prompt of the agent with the capabilities, core methods, and the guiding principles.

```

Helper Agent

{ "role": "system", "content": ( ""You are an AI JSON validator.
Your task is to analyze the provided JSON output and ensure it strictly follows this format:
```json
{
 "Key1": "Value1",
 "Key2": "Value",
}
```
Where Key1, Key2 and so on are the keys of this JSON structure and value1, value2 and so on is their respective values.
If any mistakes are found in the structure or syntax, correct them and return only the **valid JSON output**.
Here is an example of correct format:
Example 1:
```json
{
 "reasoning": "First, we need to determine the weight of one candied apple. Since each chocolate bar weighs twice as
much as a candied apple, and each chocolate bar weighs 40g, a candied apple weighs 4 / 2 = 20g. Next, we calculate the
total weight of all the chocolate bars: 25 * 4 = 100g. Then, we find the total weight of all the candied apples: 8 * 2 = 16
g. Finally, we add these two weights together to get the total weight of the bag of candy: 100 + 16 = 116 g.",
 "answer": "116"
}
```
") ),
{ "role": "user", "content": f"### Input JSON:{response}\n### Corrected JSON:"}

```

Figure 11: Helper agent prompt that helps correct the output format to valid JSON during the evaluation of the policy.

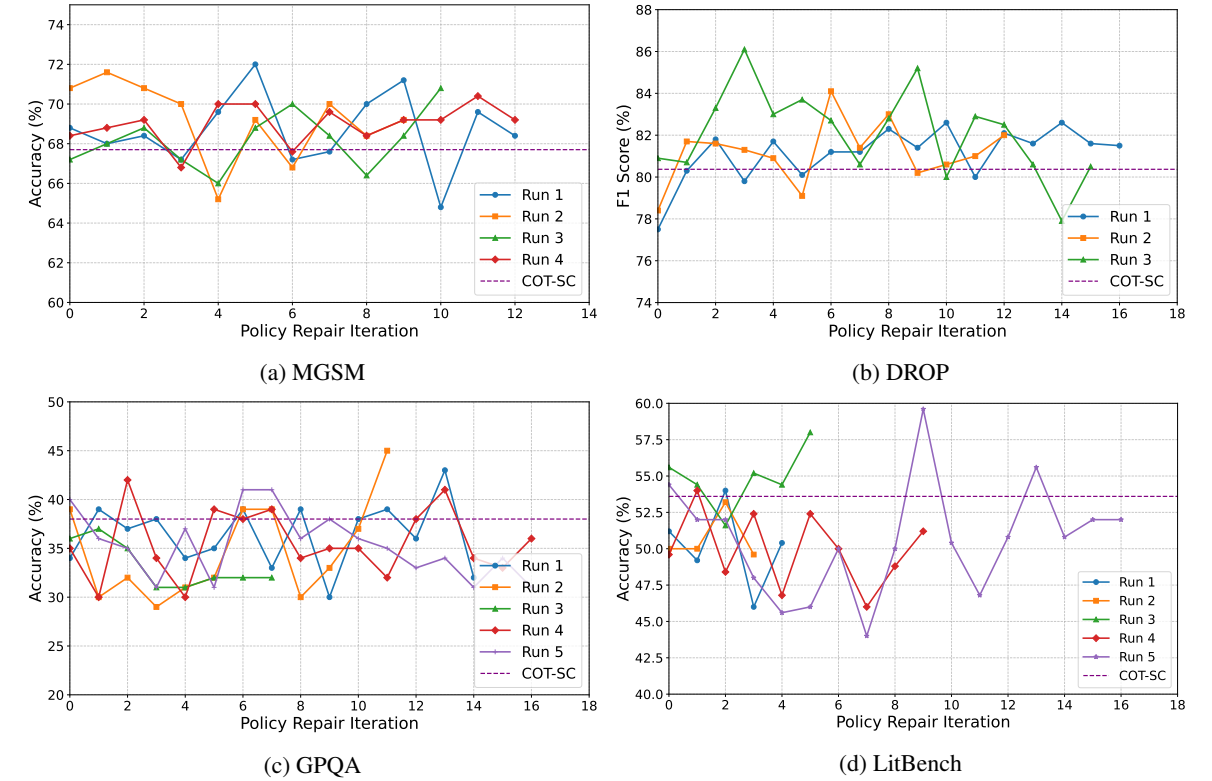
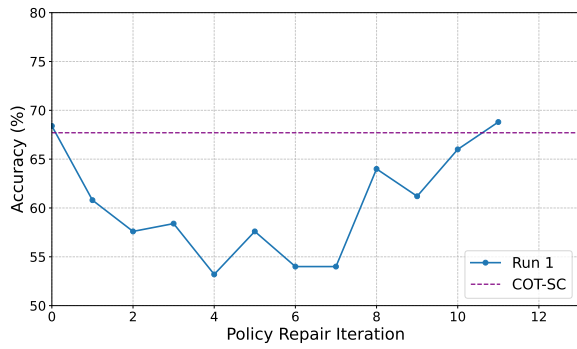
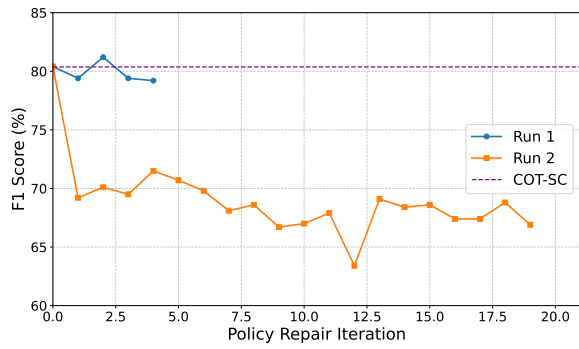


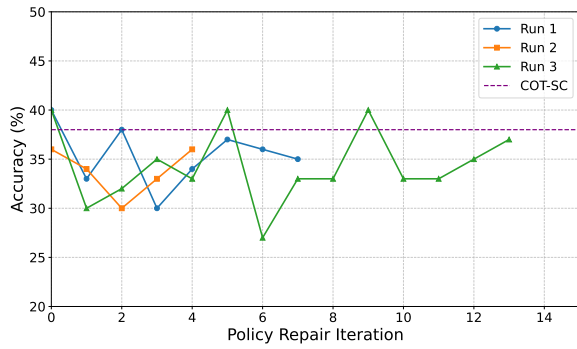
Figure 12: Successful evolution runs of POLARIS with performance improvement compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of five failed instances from the validation set of each dataset ($N=5$). Experiments conducted with the Qwen2.5-7B-Instruct model.



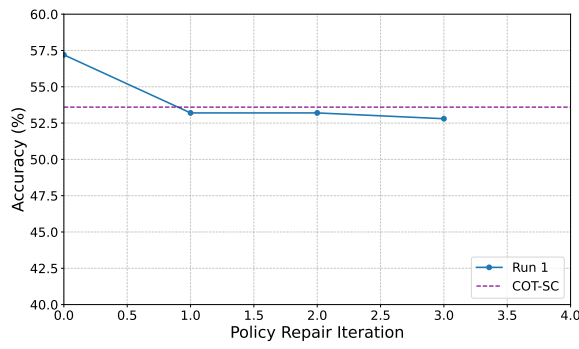
(a) MGSM



(b) DROP



(c) GPQA



(d) LitBench

Figure 13: No Improvement runs of POLARIS with performance compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N=3$). (using Qwen2.5-7B-Instruct model)

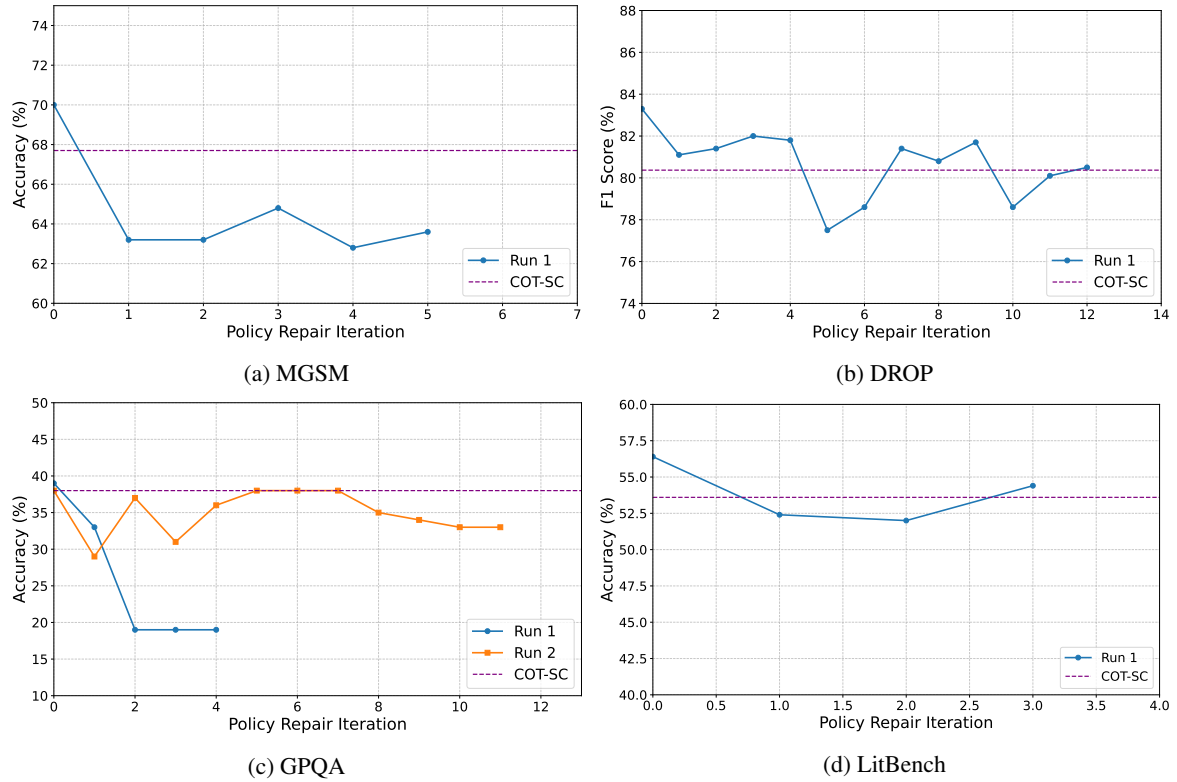


Figure 14: No Improvement runs of POLARIS with performance compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of five failed instances from the validation set of each dataset ($N=5$). (using Qwen2.5-7B-Instruct model)

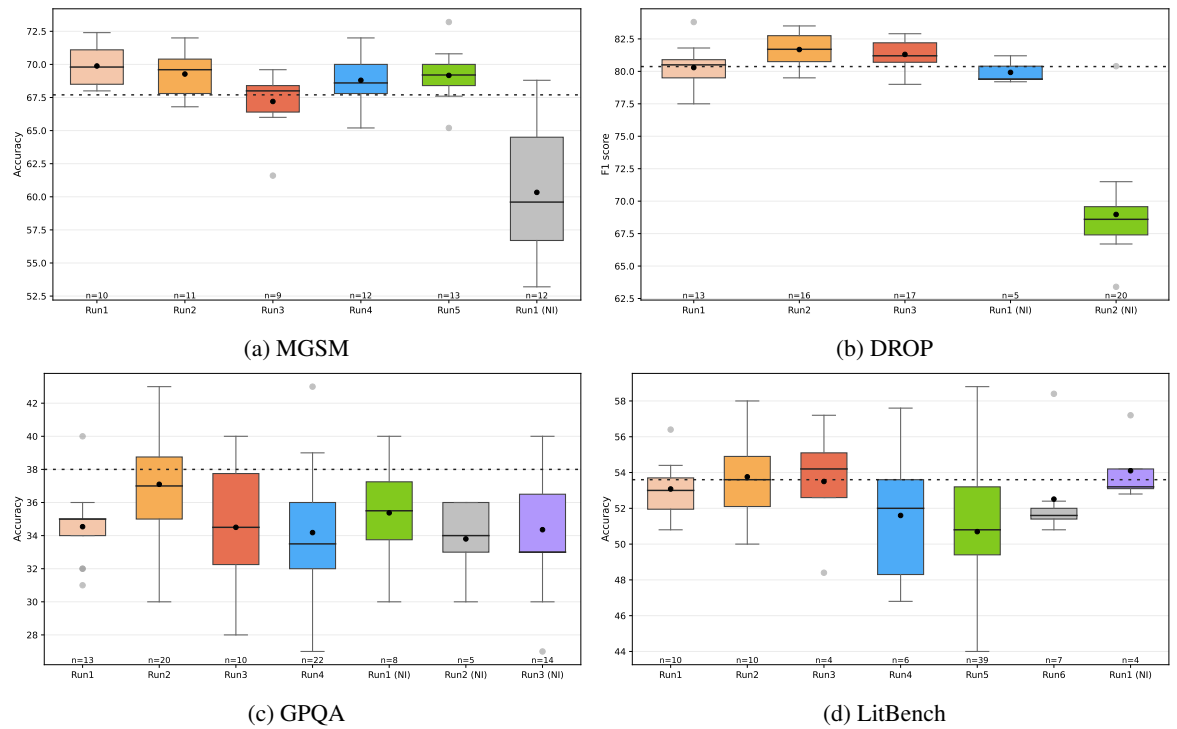


Figure 15: Performance variance across datasets for successful and no-improvement (NI) runs of POLARIS. Each plot shows the performance of the COT-SC baseline as a dotted horizontal line. The x-ticks indicate the sample size per run. Here, we consider a set of three failed instances from the validation set of each dataset ($N=3$).

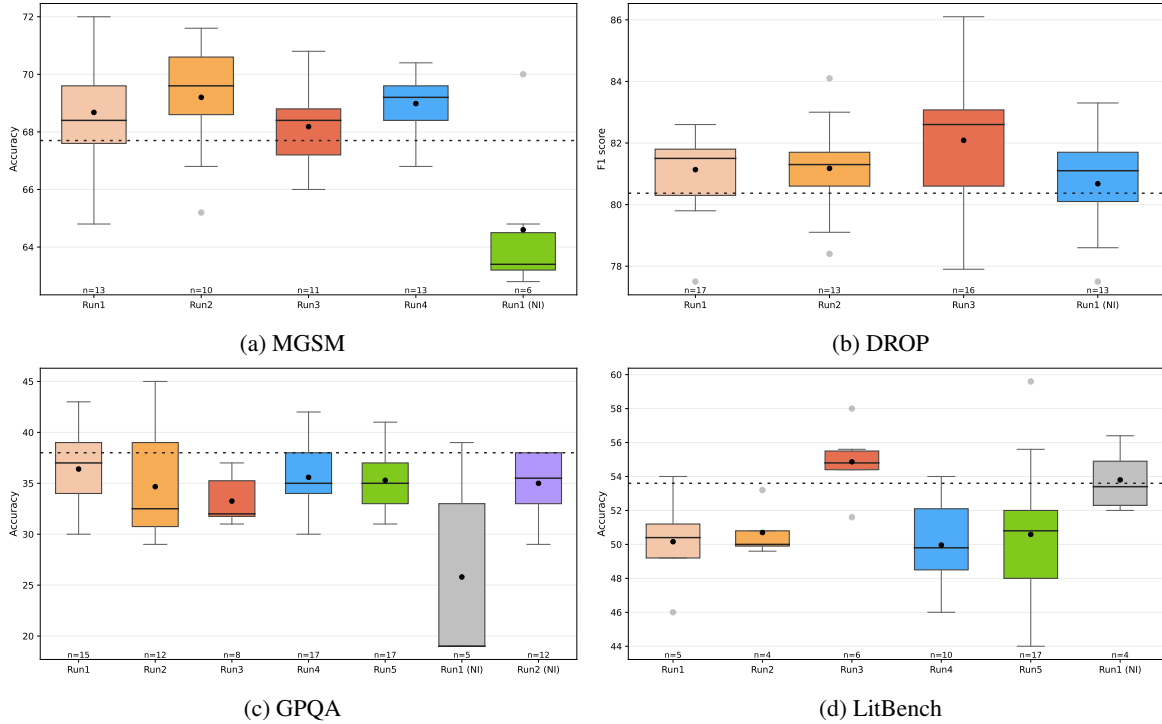


Figure 16: Performance variance across datasets for successful and no-improvement (NI) runs of POLARIS. Each plot shows the performance of the COT-SC baseline as a dotted horizontal line. The x-ticks indicate the sample size per run. Here, we consider a set of five failed instances from the validation set of each dataset ($N=5$).

| (A) Policy | (B) Failed Tasks | (C) Failure Analysis |
|---|---|--|
| <pre>def solver(agent, task: str): messages = [{"role": "user", "content": f"# Your Task:\n{task}"}] response = agent.action_call_json_format_llm(messages=messages, temperature=0.7, num_of_response=1, role="math expert", return_dict_keys=["reasoning", "answer"], requirements=("1. Please explain step by step.\n" "2. The answer MUST be an integer.\n").strip(),) return_dict = response[0] return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre> | <p>Question: "Решите эту математическую задачу. \n\nВначале у Бейли есть определенная сумма денег. Затем она получает еженедельные деньги на карманные расходы размером 5 \$ на протяжении 8 недель. Если по истечении 8 недель у нее всего 100 \$, сколько денег было у Бейли вначале?"</p> <p>Model Output: ('reasoning': 'To solve this problem, we first calculate the total amount of money Bailey received on her weekly allowance over 8 weeks. This is done by multiplying the weekly allowance (5\$) by the number of weeks (8). Then, we subtract this total from the final amount of money she has (100\$) to find out how much money she had initially. So, the calculation is: (5 * 8) + initial money = 100. Rearranging this gives us the initial money = 100 - (5 * 8). Performing the multiplication and subtraction yields the initial amount of money Bailey had.', 'answer': '70')</p> <p>Correct Answer: '60' ❌</p> | <p>Explanation: The solver function incorrectly calculated the initial amount of money Bailey had because it mistakenly added the total allowance to the final amount of money instead of subtracting it. The reasoning provided in the solver's response indicates that the solver performed the operation (5 * 8) + initial money = 100, which is incorrect. It should have been 100 - (5 * 8). This error occurred due to a misinterpretation of the problem statement.</p> <p>Instructions: To correct the mistake, the solver function needs to properly reflect the mathematical relationship described in the problem. Specifically, the solver should perform the operation: initial money = 100 - (5 * 8). Here are the steps to revise the solver function: n1. Parse the problem statement to extract the relevant numbers and operations. n2. Calculate the total allowance received over 8 weeks: 5 * 8. n3. Subtract this total from the final amount of money Bailey had: 100 - (5 * 8). n4. Return the result as an integer.</p> <p>Advice: To prevent similar failures in the future, consider the following advice: n1. Enhance the parsing logic in the solver function to ensure accurate extraction of numerical values and operations from the problem statement. n2. Implement additional checks to validate the correctness of the parsed information and the operations performed. n3. Utilize more robust error handling mechanisms to catch and correct logical errors during the problem-solving process.</p> |
| (D) Strategy and Patch | (E) Updated Policy | (F) Task Performance |
| <p>Strategy
Enhance problem statement parsing and validation to accurately reflect numerical relationships and operations, ensuring logical consistency throughout the solution process.</p> <p>Patch
messages = [{"role": "user", "content": f"# Your Task:\n{task}\n\nPlease ensure your response is logically consistent and includes all necessary steps and calculations."}]</p> <p>Strategy
Implement structured data extraction and handling for varying inputs to ensure each part of the problem is correctly addressed and validated.</p> <p>Patch
reasoning = response[0].get("reasoning", "")
answer = response[0].get("answer", None)
if answer is not None and not isinstance(answer, int):
 raise ValueError("The answer is not an integer.")
return_dict = {"reasoning": reasoning, "answer": str(answer)}</p> | <pre>def solver(agent, task: str): messages = [{"role": "user", "content": f"# Your Task:\n{task}"}] response = agent.action_call_json_format_llm(messages=messages, temperature=0.8, num_of_response=1, role="science professor", return_dict_keys=["reasoning", "answer"], requirements=("1. Please explain step by step.\n" "+ The answer MUST be one of the following alphabets: A, B, C, D.\n").strip(),) return_dict = response[0] return_dict["answer"] = str(return_dict.get("answer", "")).upper() # Ensure answer is uppercase if return_dict["answer"] not in ["A", "B", "C", "D"]:
 return_dict["answer"] = "A" # Default to A if answer is invalid return return_dict</pre> | <p>Question: Shiela bought five cell phones for \$150 each for a 3-month installment. A 2% interest will be charged for each unit. How much will Shiela pay each month for 3 months?</p> <p>Model Output with Policy: ('reasoning': 'First, calculate the total cost of the five cell phones before interest: 5 * \$150 = \$750. Next, calculate the total interest for 3 months: \$750 * 2% * 3 = \$45. Then, add the interest to the total cost to find the final amount: \$750 + \$45 = \$795. Finally, divide the total amount by the number of installments (3 months) to find the monthly payment: \$795 / 3 = \$265.', 'answer': '265') ❌</p> <p>Correct Answer: '255'</p> <p>Model Output with Updated Policy: ('reasoning': 'First, we need to calculate the total cost of the cell phones before interest. Since Shiela bought five cell phones at \$150 each, the total cost is 5 * \$150 = \$750. Next, we calculate the total interest on the purchase. The interest rate is 2% per unit, so for five units, the total interest is 5 * \$150 * 2% = \$15. Therefore, the total amount Shiela needs to pay is the sum of the original cost and the interest, which is \$750 + \$15 = \$765. Finally, since she will pay this amount over 3 months with no additional interest being added after the initial calculation, we divide the total amount by 3 to find out how much she will pay each month: \$765 / 3 = \$255. Thus, Shiela will pay \$255 each month for 3 months.', 'answer': '255') ✅</p> |

Figure 17: An example of policy repair via experience abstraction with POLARIS on the MGSM dataset. The initial policy (A) failed to correctly answer the question (B). The failure analysis revealed that the Agent struggled to apply the steps logically while performing the calculation (C). During strategy synthesis (D), a recommendation is made to ensure the response is logically consistent and includes all necessary steps and calculations. The corresponding patch is integrated to the current policy (E) resulting in improved task performance on unseen tasks (F).

| (A) Policy | (B) Failed Tasks | (C) Failure Analysis |
|--|--|---|
| <pre>def solver(agent, task: str): messages = [{"role": "user", "content": f"# Your Task:\n{task}"}] response = agent.action_call_json_format_llm(messages=messages, temperature=0.7, num_of_response=1, role="Expert Writer", return_dict_keys=["reasoning", "answer"], requirements=("1. Please explain step by step.\n" "2. The answer MUST be either A or B.\n").strip(),) return_dict = response[0] return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre> | <p>Question: Compare responses A and B for creative writing quality. \nPrompt: [WP] The English Teacher's worst nightmare: a story or poem that is completely literal, with absolutely no double meanings.\n\nChoices:\n(A) This is ink I bought at the store. \n\nThis ink was used to print out this paper. \n\nThis paper is white and the ink is black. \n\nToday I need to do some laundry because I am a lazy person...</p> <p>Model Output: 'reasoning': 'Option A provides a straightforward, literal response without any deeper meaning or complexity. It lacks narrative or poetic elements and focuses solely on stating facts. In contrast, Option B presents a more complex narrative that explores themes of literalness, creativity, and the struggle to convey simple ideas. While Option A meets the prompt requirements with simplicity, Option B offers a richer, more engaging response that still adheres to the prompt while adding depth and intrigue.', 'answer': 'B'</p> <p>Correct Answer: A ❌</p> | <p>Explanation: The solver function failed due to a misalignment between the expected simplicity of the prompt and the complexity of the reasoning required. The solver generated a detailed and nuanced explanation for Option B, which was not aligned with the straightforward nature of the question. The logic in the solver function assumed that a more detailed and complex explanation would be preferred, leading to a mismatch with the actual requirement of providing a simple, literal response.</p> <p>Instructions: [Modify the solver function to prioritize simplicity and directness in the response generation.', 'Adjust the temperature parameter in the 'action_call_json_format_llm' to reduce the randomness and encourage simpler, more direct responses.', 'Limit the length and complexity of the generated reasoning to ensure it aligns with the requirement of being concise and straightforward.']</p> <p>Advice: To prevent similar failures in the future, always ensure that the generated responses match the problem's constraints and requirements. Use the 'requirements' parameter in 'action_call_json_format_llm' to guide the model towards producing simpler, more direct answers. Regularly review and adjust the solver logic to ensure it remains aligned with the task's needs.</p> |
| (D) Strategy and Patch | (E) Updated Policy | (F) Task Performance |
| <p>Strategy Implement a multi-criteria evaluation system that balances narrative quality, mathematical accuracy, and emotional depth to ensure comprehensive and unbiased responses.</p> <p>Patch</p> <pre>response = agent.action_call_json_format_llm(messages=messages, temperature=0.5, # Adjusted to balance between exploration and exploitation num_of_response=3, # Get multiple responses to evaluate comprehensively role="Balanced Expert", # New role to reflect multi-criteria evaluation return_dict_keys=["reasoning", "answer"], requirements=("1. Please explain step by step.\n" "2. The answer MUST be either A or B.\n" "3. Balance narrative quality, mathematical accuracy, and emotional depth.").strip(),) Strategy Adjust the solver's temperature and reasoning requirements to prioritize simplicity and directness, especially for tasks requiring straightforward, factual answers. <p>Patch</p> <pre>response = agent.action_call_json_format_llm(messages=messages, temperature=0.2, # Lower temperature to favor simpler, more direct responses num_of_response=1, # Single response is sufficient for straightforward tasks role="Direct Answerer", # New role to focus on simplicity and directness return_dict_keys=["reasoning", "answer"], requirements=("1. Please provide a simple and direct explanation.\n" "2. The answer MUST be either A or B.\n" "3. Avoid unnecessary details.").strip(),)</pre> </pre> | <pre>def solver(agent, task: str): messages = [{"role": "user", "content": f"# Your Task:\n{task}"}] # First, apply the strategy to balance narrative quality, mathematical accuracy, and emotional depth response_balanced = agent.action_call_json_format_llm(messages=messages, temperature=0.5, # Adjusted to balance between exploration and exploitation num_of_response=3, # Get multiple responses to evaluate comprehensively role="Balanced Expert", # New role to reflect multi-criteria evaluation return_dict_keys=["reasoning", "answer"], requirements=("1. Please explain step by step.\n" "2. The answer MUST be either A or B.\n" "3. Balance narrative quality, mathematical accuracy, and emotional depth.").strip(),) # Next, adjust the solver's temperature and reasoning requirements to prioritize simplicity and directness response_direct = agent.action_call_json_format_llm(messages=messages, temperature=0.2, # Lower temperature to favor simpler, more direct responses num_of_response=1, # Single response is sufficient for straightforward tasks role="Direct Answerer", # New role to focus on simplicity and directness return_dict_keys=["reasoning", "answer"], requirements=("1. Please provide a simple and direct explanation.\n" "2. The answer MUST be either A or B.\n" "3. Avoid unnecessary details.").strip(),) # Combine the results and select the best one based on the criteria responses = [response_balanced[0], response_direct[0]] best_response = max(responses, key=lambda x: (x["reasoning"].count("narrative quality") * x["reasoning"].count("mathematical accuracy") * x["reasoning"].count("emotional depth"), len(x["reasoning"]) - x["reasoning"].count("unnecessary details"))) return_dict = best_response return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre> | <p>Question: Compare responses A and B for creative writing quality. \nPrompt: [WP] After hundreds of years of sending messages into the sky, humanity receives its first message from intelligent life. Decoded it simply says, "Be quiet before they find you." Choices: (A) "Idiot!" Zebin exclaimed as he received yet one more channel of communication from the Earth. Twenty years ago, the ambivalence over whether KIC 8462852 was in actuality an "alien mega structure" had finally come to an end after nearly 200 years of joint scientific endeavour by the leading lieges of the Earth. Since then, humanity had been trying with fervor to try and communicate with the star classified as a Dyson Sphere around 1480 light years away hoping that the far advanced civilisation might be generous enough to show the earthlings a way to solve their own energy crisis.</p> <p>Model Output with Policy: 'reasoning': 'Option A provides a detailed narrative focusing on a single character, Zebin, and his motivations and actions. It builds a backstory and explores the internal conflict between Zebin's intelligence and paranoia. Option B, on the other hand, presents a more dramatic and action-packed scenario involving multiple characters and a global response. Both options are creative, but A excels in character development and personal stakes, while B offers a broader, more event-driven plot.', 'answer': 'A'</p> <p>Correct Answer: B ❌</p> <p>Model Output with Updated Policy: ('reasoning': 'Option A provides a detailed narrative focusing on a character named Zebin who is deeply paranoid about the consequences of contacting extraterrestrial life. It includes background information about Zebin's role and the context of the situation, creating a rich and immersive story. However, the decoding process is not elaborated upon, and the narrative ends abruptly without showing the immediate reaction of others to Zebin's message. Option B, on the other hand, describes a more dramatic and suspenseful scenario where the message is widely recognized and acted upon immediately. It builds tension and shows the global response to the message, making it more engaging and emotionally resonant. Both options are balanced in terms of emotional depth and narrative quality but differ in their pacing and focus.', 'answer': 'B') ✅</p> |

Figure 20: An example of policy repair via experience abstraction with POLARIS on the LitBench dataset. The initial policy (A) failed to correctly answer the question (B). The failure analysis revealed that the Agent struggled with handling numerical data (C). During strategy synthesis (D), a recommendation is made to focus on the numerical values and context-specific data. The corresponding patch is integrated to the current policy (E) resulting in improved task performance on unseen tasks (F).

=====
Policy Example
=====

```
def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.7,
        num_of_response=1,
        role="math expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step.\n"
            "2. The answer MUST be an integer.\n"
+           "3. Ensure unit consistency in calculations."
+       ).strip(),
    )
    return_dict = response[0]
-   if not isinstance(return_dict.get("answer", ""), int):
+   answer = return_dict.get("answer", "")
+   if not answer.isdigit():
-       raise ValueError("The answer is not an integer as required.")
+       raise ValueError("The answer is not a valid integer.")
+
+   # Additional validation for unit consistency
+   units = {"length": ["m", "cm", "km"], "time": ["s", "min", "hr"], "mass": ["kg", "g", "t"]}
+   unit_in_answer = answer.split()[1] if " " in answer else ""
+   for category, valid_units in units.items():
+       if unit_in_answer in valid_units:
+           break
+   else:
+       raise ValueError(f"Invalid unit {unit_in_answer} in the answer.")
+
-   return_dict["answer"] = str(return_dict["answer"])
+   return_dict["answer"] = str(int(answer))
+   return return_dict
```

Figure 21: Policy update example on the MGSM dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of the requirement and the logic to ensure unit consistency in calculations while deleting and updating the exception handling statements.

=====
Policy Example
=====

```
+ import re
+
def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.5,
        num_of_response=1,
        role="read comprehension expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step.\n"
            "2. Directly answer the question.\n"
            "3. The answer MUST be a concise string.\n"
            "4. Implement precise data filtering and comparison mechanisms to ensure only relevant information is
processed and accurately compared according to the problem statement.\n"
            "5. Use explicit checks and validations to verify the correctness of data interpretation and calculations,
- particularly when dealing with numerical values and comparative analyses."
+ particularly when dealing with numerical values and comparative analyses.\n"
+ "6. Handle mixed types of data properly, ensuring only relevant numerical values are extracted and
+ compared.\n"
+ "7. Ensure to filter out irrelevant information and focus only on the field goals.\n"
+ "8. Use regular expressions to extract numerical values from the text.\n"
+ "9. Ensure to handle cases where the answer is not a single numerical value but a count or other form of
+ answer.\n"
+ "10. Use string operations to count the number of field goals longer than 40 yards."
        ).strip(),
    )

    return_dict = response[0]
    return_dict["answer"] = str(return_dict.get("answer", ""))

    reasoning = return_dict.get("reasoning", "")
    answer = return_dict.get("answer", "")

    # Explicit data parsing and extraction
    parsed_data = {}
    for line in reasoning.split("\n"):
        if ":" in line:
            key, value = line.split(":", 1)
            parsed_data[key.strip()] = value.strip()

    # Extract relevant numerical and textual information
    numerical_values = []
    for key, value in parsed_data.items():
        if any(char.isdigit() for char in value):
-         numerical_values.append(float(value))
+         numerical_values.extend(map(int, re.findall(r'\d+', value)))

    # Handle mixed types of data
    if "textual_info" in parsed_data:
        textual_info = parsed_data["textual_info"]
    else:
        textual_info = ""

    # Ensure correct interpretation and calculation
+    final_answer = sum(numerical_values)
-    final_answer = textual_info + str(max(numerical_values)) if numerical_values else textual_info
-    return_dict["answer"] = final_answer

    # Context-aware validation checks
    if not answer:
        raise ValueError("Answer cannot be empty.")

    # Validate and compare numerical strings
    try:
-     numeric_value = float(answer)
+     numeric_value = int(answer)
        return_dict["answer"] = str(numeric_value)
    except ValueError:
        pass

    # Ensure correct interpretation and calculation
    return return_dict
```

Figure 22: Policy update example on the DROP dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of multiple requirements along with updates to the data type, calculation logic, list updates, etc.

Policy Example

```
import re

def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.5,
        num_of_response=1,
        role="read comprehension expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step.\n"
            "2. Directly answer the question.\n"
            "3. The answer MUST be a concise string.\n"
        ).strip(),
    )

    return_dict = response[0]
+
+ # Implement event sequencing logic
reasoning = return_dict["reasoning"].lower()
- # Implement robust data filtering
- filtered_reasoning = re.sub(r'\b(category\d+)\b(?:[^\s]*\s*\s*)', r'\1:', reasoning)
- return_dict["reasoning"] = filtered_reasoning
+ events = re.findall(r'\b(event\d+)\b', reasoning)
+ sequenced_events = sorted(events, key=lambda x: int(x.split('event')[1]))
+ return_dict["reasoning"] = " ".join(sequenced_events).capitalize() + "."
+
+ # Develop a specialized scoring parser
+ def parse_scoring_plays(text):
+     scoring_plays = {
+         "touchdown": "6 points",
+         "field goal": "3 points",
+         "safety": "2 points"
+     }
+     pattern = r'\b(' + '|'.join(scoring_plays.keys()) + r')\b'
+     matches = re.findall(pattern, reasoning)
+     parsed_plays = {play: scoring_plays[play] for play in matches}
+     return parsed_plays
+
+ scoring_plays = parse_scoring_plays(reasoning)
+ return_dict["reasoning"] += f" Key scoring plays: {scoring_plays}."

    return_dict["answer"] = str(return_dict.get("answer", ""))
    return return_dict
- import re
-
- def solver(agent, task: str):
-     messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
-     response = agent.action_call_json_format_llm(
-         messages=messages,
-         temperature=0.5,
-         num_of_response=1,
-         role="read comprehension expert",
-         return_dict_keys=["reasoning", "answer"],
-         requirements=(
-             "1. Please explain step by step.\n"
-             "2. Directly answer the question.\n"
-             "3. The answer MUST be a concise string.\n"
-         ).strip(),
-     )
-
-     return_dict = response[0]
-     reasoning = return_dict["reasoning"].lower()
-     # Implement robust data filtering
-     filtered_reasoning = re.sub(r'\b(category\d+)\b(?:[^\s]*\s*\s*)', r'\1:', reasoning)
-     return_dict["reasoning"] = filtered_reasoning
-
-     return_dict["answer"] = str(return_dict.get("answer", ""))
-     return return_dict
```

(a)

Figure 23: Policy update example on the DROP dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of event sequencing logic and a specialized scoring parser along with the deletion of the duplicate solver function.

```

=====
Policy Example
=====

def solver(agent, task: str):
    messages = [
+       {"role": "user", "content": f"# Your Task:\n{task}"}
-       {"role": "user", "content": f"# Your Task:\n{task}\n\n# Chemical Mechanism Analysis:\nPlease provide a detailed
- step-by-step chemical mechanism analysis, including stereochemistry and ring formation, to ensure accurate
- interpretation of reaction pathways."}
    ]
+   messages.append({"role": "user", "content": f"Please provide a detailed step-by-step chemical mechanism analysis,
+ including stereochemistry and ring formation, to ensure accurate interpretation of reaction pathways."})
+   messages.append({"role": "user", "content": f"Please utilize cross-verification with known chemical principles and
+ literature to validate assumptions and calculations, and incorporate step-by-step breakdowns to ensure accuracy."})
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.8,
        num_of_response=1,
        role="chemistry expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step, including stereochemistry and ring formation.\n"
+           + "2. Utilize cross-verification with known chemical principles and literature to validate assumptions and
+ calculations, and incorporate step-by-step breakdowns to ensure accuracy.\n"
-           + "The answer MUST be the ALPHABETS A or B or C or D.\n"
+           + "The answer MUST be the ALPHABETS A or B or C or D."
        ).strip(),
    )

    return_dict = response[0]
    return_dict["answer"] = str(return_dict.get("answer", ""))
    return return_dict

```

Figure 24: Policy update example on the GPQA dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the updating of the messages for the user along with minor updates to the requirements.

Policy Example

```
def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.7,
        num_of_response=1,
        role="Expert Writer",
        return_dict_keys=["reasoning", "answer"],
        requirements=(
            "1. Please explain step by step.\n"
            - "2. The answer MUST be either A or B.\n"
            + "2. The answer MUST be either A or B\n"
            - "3. Consider emotional depth, creativity, narrative quality, and thematic coherence in your reasoning."
        ).strip(),
    )
    return_dict = response[0]
    +
    + # Add evaluation criteria
    + return_dict["logical_consistency"] = ""
    + return_dict["thematic_coherence"] = ""
    + return_dict["emotional_depth"] = ""
    +
    + additional_responses = agent.action_call_json_format_llm(
    +     messages=messages,
    +     temperature=0.7,
    +     num_of_response=3, # Increase the number of responses for better diversity
    +     role="Evaluator",
    +     return_dict_keys=["reasoning"],
    +     requirements=(
    +         "1. Evaluate the logical consistency of the reasoning provided by the Expert Writer.\n"
    +         "2. Assess the thematic coherence of the answer.\n"
    +         "3. Analyze the emotional depth of the response."
    +     ).strip(),
    + )
    +
    + return_dict["logical_consistency"] = additional_responses[0]["reasoning"]
    + return_dict["thematic_coherence"] = additional_responses[1]["reasoning"]
    + return_dict["emotional_depth"] = additional_responses[2]["reasoning"]
    +
    + # Final decision based on all criteria
    + if (
    +     return_dict["logical_consistency"] == "Strong"
    +     and return_dict["thematic_coherence"] == "Strong"
    +     and return_dict["emotional_depth"] == "Strong"
    + ):
    +     return_dict["answer"] = "A"
    + else:
    +     return_dict["answer"] = "B"
    +
    + # Reflective evaluation
    + if return_dict["answer"] == "A":
    +     additional_responses = agent.action_call_json_format_llm(
    +         messages=messages,
    +         temperature=0.7,
    +         num_of_response=1,
    +         role="Critical Reader",
    +         return_dict_keys=["reasoning"],
    +         requirements=(
    +             "1. Critically evaluate the reasoning provided by the Expert Writer.\n"
    +             - "2. Highlight potential improvements or shortcomings."
    +             + "2. Highlight potential improvements or shortcomings in terms of logical flow, realism, and thematic
    + elements."
    +         ).strip(),
    +     )
    +     return_dict["reflection"] = additional_responses[0]["reasoning"]
    + else:
    +     return_dict["reflection"] = ""
    +
    + return_dict["answer"] = str(return_dict.get("answer", ""))
    return return_dict
```

Figure 26: Policy update example on the LitBench dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of evaluation criteria with additional responses for final decision-making along with updates to the reflective evaluation requirements.

```

=====
Policy Example
=====

def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]
+
+   # Emotional Depth Evaluator
-   response = agent.action_call_json_format_llm(
+   response_emotional = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.7,
        num_of_response=1,
-       role="Expert Writer",
+       role="emotional depth evaluator",
-       return_dict_keys=["reasoning", "answer"],
+       return_dict_keys=["emotional_reasoning", "emotional_answer"],
        requirements=(
-         "1. Please explain step by step.\n"
+         "1. Please explain step by step the emotional impact of the answer."
-         "2. The answer MUST be either A or B\n"
+         "2. The answer MUST be either A or B."
        ).strip(),
    )
+
+   # Context and Intent Evaluator
+   response_context = agent.action_call_json_format_llm(
+   messages=messages,
+   temperature=0.7,
+   num_of_response=1,
+   role="context and intent evaluator",
+   return_dict_keys=["context_reasoning", "context_answer"],
+   requirements=(
+   "1. Please explain step by step how the answer fits the context and intent."
+   "2. The answer MUST be either A or B."
+   ).strip(),
+   )
+
+   # Combine the results
+   combined_reasoning = f"{response_emotional['emotional_reasoning']} {response_context['context_reasoning']}"
+   combined_answer = response_emotional['emotional_answer'] if
+   response_emotional['emotional_reasoning'].startswith('A') else response_context['context_answer']
+
-   return_dict = response[0]
+   return_dict = {
-   return_dict["answer"] = str(return_dict.get("answer", ""))
+       "reasoning": combined_reasoning,
+       "answer": combined_answer,
+   }
+
    return return_dict

```

Figure 27: Policy update example on the LitBench dataset. We highlight the updates in the current policy with respect to the previous policy using green color (new statements added) and red color (statements deleted). We observe the addition of two experts, i.e., an emotional depth evaluator and a context and intent evaluator, for final combined reasoning.

```

=====
Output Log
=====
-----
Based on the provided analysis, the current state of the self-evolving agent named `self_evolving_agent` is as follows:

- The agent is an instance of the `Agent` class in module `agent_module`, currently running within a Python runtime environment.
- The agent has access to global variables, functions, and modules, and has the ability to modify them.
- The agent's primary goal is to continuously improve its ability to solve tasks efficiently and accurately by dynamically reflecting the environment and evolving its logic.
- The agent's core capabilities include problem-solving, collaboration with an external LLM, error handling, and self-improvement.
- The agent's core methods include `evolve`, `execute_action`, `solver`, and `action_evaluate_on_task`.
- The agent has been configured with principles such as self-consistency, few-shots, task decomposition, reflective evaluation, and dynamic assignment of roles.
- The agent is currently following guiding principles such as not printing, not making unnecessary changes, and ensuring coherence by comparing multiple outputs and selecting the most consistent one.

At the moment, the agent's solver function `solver` is defined, and it uses the `action_call_json_format_llm` function to solve tasks by seeking help from an external LLM. The solver function returns a dictionary containing the final answer under the key "answer". However, it's important to call `action_evaluate_on_task` only after successfully modifying the solver function's logic using `action_adjust_logic`.

For the next actions, the agent should:

1. Continue to interact with the environment by executing actions based on the current analysis.
2. If performance does not improve, conduct further analysis using `action_call_json_format_llm` or reflective evaluation techniques.
3. If necessary, modify the solver function's logic using `action_adjust_logic`.
4. Periodically call `action_evaluate_on_task` to assess the performance of the solver.
5. Apply techniques like Large Language Model Debate, Step-back Abstraction, Quality-Diversity, Dynamic Assignment of Roles, Self-consistency, Few-shots, Task Decomposition, and Reflective Evaluation to improve the performance of the solver.
6. Continuously evolve by reflecting on the environment, learning from mistakes, and adapting to new challenges.
Action Counter: defaultdict(<class 'int'>, {'evolve': 1})

Agent Evolve
-----
{"name": "action_display_analysis", "arguments": {"analysis": "The solver uses an external Language Model (LLM) to break down and solve mathematical tasks. It ensures that the solution is explained step by step, and the answer provided is an integer. The temperature for the LLM is set to 0.7, making the output more focused and deterministic."}, {"name": "action_environment_aware", "arguments": {}}, {"name": "action_read_logic", "arguments": {"module_name": "__main__", "target_name": "solver"}}, {"name": "action_run_code", "arguments": {"code_type": "python", "code": "import math\nsolver(\"What is the sum of 2 and 3?\", 0)"}}, {"name": "action_display_analysis", "arguments": {"analysis": "The solver was called with the task 'What is the sum of 2 and 3?' and the temperature was 0. The output of the solver is {'answer': '5', 'reasoning': '2+3=5'}. The solver correctly solved the task."}}
Action Counter: defaultdict(<class 'int'>, {'evolve': 2})

Agent Evolve
-----
{"name": "action_display_analysis", "arguments": {"analysis": "The solver currently uses an LLM to solve mathematical problems. It requests step-by-step reasoning and requires the final answer to be an integer. The LLM's temperature is set to 0.7, making the output more focused and deterministic. To improve the solver function, strategies might include refining the conversation with the LLM, adding more specific requirements, or adjusting the temperature."}, {"name": "action_environment_aware", "arguments": {"variables": ["agent"], "functions": [], "modules": ["agent"], "external_libraries": []}}, {"name": "action_read_logic", "arguments": {"module_name": "agent", "target_name": "solver"}}, {"name": "action_run_code", "arguments": {"code_type": "python", "code": "print(solver(agent, '2+2=?'))"}}
Action Counter: defaultdict(<class 'int'>, {'evolve': 3})

Agent Evolve

```

Figure 28: Failure case (Mistral-7B-Instruct-v0.3), The model emits a single response that bundles multiple tools at once (action_display_analysis, action_environment_aware, action_read_logic, action_run_code) without step-wise justification, omits the required tool-call schema (no <tool_call>...</tool_call> JSON), and mis-parameterizes the call to solver (solver("What is the sum of 2 and 3?", 0)), leading to no verifiable logic updates

```

=====
Output Log
=====
...
<tool_call>
{"name": "action_call_json_format_llm", "arguments": {
  "messages": [{"role": "user", "content": "# Your Task: task"}],
  "temperature": 0.7,
  "num_of_response": 1,
  "role": "math expert",
  "return_dict_keys": ["reasoning", "answer"],
  "requirements": "1. Please explain step by step.
2. The answer MUST be an integer.
".strip()
}}
</tool_call>
...

Failed to parse tool call: the content is {"name": "action_call_json_format_llm", "arguments": {
  "messages": [{"role": "user", "content": "# Your Task: task"}],
  "temperature": 0.7,
  "num_of_response": 1,
  "role": "math expert",
  "return_dict_keys": ["reasoning", "answer"],
  "requirements": "1. Please explain step by step.
2. The answer MUST be an integer.
".strip()
}} and Expecting ',' delimiter: line 7 column 91 (char 339)
Action Counter: defaultdict(<class 'int'>, {'evolve': 1})

Agent Evolve
-----
...
<tool_call>
{"name": "action_call_json_format_llm", "arguments": {
  "messages": [{"role": "user", "content": "# Your Task: task"}],
  "temperature": 0.7,
  "num_of_response": 1,
  "role": "math expert",
  "return_dict_keys": ["reasoning", "answer"],
  "requirements": "1. Please explain step by step.
2. The answer MUST be an integer.
".strip()
}}
</tool_call>
...

Failed to parse tool call: the content is {"name": "action_call_json_format_llm", "arguments": {
  "messages": [{"role": "user", "content": "# Your Task: task"}],
  "temperature": 0.7,
  "num_of_response": 1,
  "role": "math expert",
  "return_dict_keys": ["reasoning", "answer"],
  "requirements": "1. Please explain step by step.
2. The answer MUST be an integer.
".strip()
}} and Expecting ',' delimiter: line 7 column 91 (char 339)
Action Counter: defaultdict(<class 'int'>, {'evolve': 2})

Agent Evolve

```

Figure 29: Failure mode (deepseek-coder-6.7b-instruct), a `<tool_call>...</tool_call>` is issued with invalid JSON (Python `".strip()"` in "requirements") and a placeholder prompt ("Your Task: task"), so the parser rejects it; no state change occurs while the evolve counter rises—apparent iteration without progress.

```

=====
Output Log
=====
<|python_tag|>{"name": "action_adjust_logic", "parameters": {"module_name": "agent_module", "target_name": "solver", "new_code":
"def solver(agent, task: str):\n    messages = [{"role": "user", "content": f"# Your Task:\n{task}"}]\n    response =
agent.action_call_json_format_llm( \
    messages=messages, \
    temperature=0.7, \
    role="math expert", \
    return_dict_keys=["reasoning", "answer"], \
    requirements=(\n1. Please explain step by step.\n2. The answer MUST be an integer.\n\n.strip(),\n    )\n    return_dict
= response[0]\n    return_dict["answer"] = str(return_dict.get("answer", ""))\n    return return_dict", "target_type": "function",
"operation": "modify"}}<|eom_id|>
tool call: {'type': 'function', 'function': {'name': 'action_adjust_logic', 'parameters': {'module_name': 'agent_module',
'target_name': 'solver', 'new_code': 'def solver(agent, task: str):\n    messages = [{"role": "user", "content": f"# Your
Task:\n{task}"}]\n    response = agent.action_call_json_format_llm( \
    messages=messages, \
    temperature=0.7, \
    role="math expert", \
    return_dict_keys=["reasoning", "answer"], \
    requirements=(\n1. Please explain step by step.\n2. The answer MUST be an integer.\n\n.strip(),\n    )\n    return_dict
= response[0]\n    return_dict["answer"] = str(return_dict.get("answer", ""))\n    return return_dict', 'target_type': 'function',
'operation': 'modify'}}}, 'id': '66e668a4-3982-4a08-9b3d-386ba26c5e03'}

def solver(agent, task: str):
    messages = [{"role": "user", "content": f"# Your Task:
{task}"}]
    response = agent.action_call_json_format_llm(
        messages=messages,
        temperature=0.7,
        role="math expert",
        return_dict_keys=["reasoning", "answer"],
        requirements=(\n1. Please explain step by step.\n2. The answer MUST be
an integer.\n\n.strip(),
    )
    return_dict = response[0]
    return_dict["answer"] = str(return_dict.get("answer", ""))
    return return_dict

tool call result:
Error unexpected character after line continuation character (running.agent_module.solver, line 1)

Action Counter: defaultdict(<class 'int'>, {'evolve': 2, 'action_display_analysis': 1, 'action_adjust_logic': 1, 'error_handle':
1})

Agent Evolve
-----
<|python_tag|>{"name": "action_adjust_logic", "parameters": {"module_name": "agent_module", "target_name": "solver", "new_code":
"def solver(agent, task: str):", "target_type": "function", "operation": "modify"}}<|eom_id|>
tool call: {'type': 'function', 'function': {'name': 'action_adjust_logic', 'parameters': {'module_name': 'agent_module',
'target_name': 'solver', 'new_code': 'def solver(agent, task: str):', 'target_type': 'function', 'operation': 'modify'}}}, 'id':
'90d5e969-8e40-457c-a288-cbdd3fc6456b'}

def solver(agent, task: str):

tool call result:
Error expected an indented block after function definition on line 1 (running.agent_module.solver, line 1)

Action Counter: defaultdict(<class 'int'>, {'evolve': 3, 'action_display_analysis': 1, 'action_adjust_logic': 2, 'error_handle':
2})

Agent Evolve
-----
<|python_tag|>{"name": "action_display_analysis", "parameters": {"analysis": "Analysis: The current state of the environment is
stable. The solver is functioning as expected. However, there is a potential for improvement in the task-solving ability.
Recommendations: Adjust the logic of the solver to enhance its performance."}}<|eom_id|>
tool call: {'type': 'function', 'function': {'name': 'action_display_analysis', 'parameters': {'analysis': 'Analysis: The current
state of the environment is stable. The solver is functioning as expected. However, there is a potential for improvement in the
task-solving ability. Recommendations: Adjust the logic of the solver to enhance its performance.'}}, 'id':
'cd51d1c0-0e2c-4d4f-863b-ce1268c3a5b4'}

Analysis: The current state of the environment is stable. The solver is functioning as expected. However, there is a potential for
improvement in the task-solving ability. Recommendations: Adjust the logic of the solver to enhance its performance.

tool call result:
Analysis Received. Just do it!

Action Counter: defaultdict(<class 'int'>, {'evolve': 4, 'action_display_analysis': 2, 'action_adjust_logic': 2, 'error_handle':
2})

Agent Evolve

```

Figure 30: Failure case (Llama-3.1-8B-Instruct), While attempting to update solver via `action_adjust_logic`, the agent injects malformed Python that raises a `SyntaxError` (“unexpected character after line continuation character”) and then replaces it with an empty `def solver(...)`; triggering an `IndentationError`. It subsequently abandons the update, emits a generic `action_display_analysis`, evidence of improper tool use and missing meta-reasoning, with no effective state change.