

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

Anonymous ACL submission

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) 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=0.5, 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] return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre>	<p>Passage: Coming off their home loss to the Jets, the Bucs flew to Qwest Field for a week 15 duel against the Seattle Seahawks. After a scoreless first quarter, Seattle got on the board first with QB Matt Hasselbeck's 29-yard pass to tight end John Carlson. The Bucs got on the board with Connor Barth's 28-yard field goal to end the first half. In the third quarter, Tampa Bay took the lead after a 45-yard field goal and a 22-yard touchdown pass by QB Josh Freeman to running back Cadillac Williams, set up by a Tanard Jackson interception. The Buccaneers further extended their lead with a 28-yard pass to running back Derrick Ward, with a successful two-point conversion...</p> <p>Question: How many yards difference was there between Josh Freeman's field goal and touchdown pass?</p> <p>Model Output: {"reasoning": "Josh Freeman's 22-yard touchdown pass to Cadillac Williams was longer than Connor Barth's 28-yard field goal. The difference in yards is calculated by subtracting the shorter distance from the longer one, which is 28 - 22 = 6 yards.", "answer": "6"} Correct Answer: '23'</p>	<p>Explanation: The solver function failed because it did not correctly interpret the question. Specifically, the solver identified the longest distance (Josh Freeman's 22-yard touchdown pass) and the second-longest distance (Connor Barth's 28-yard field goal), but it did not correctly identify that the question was asking for the difference between these two distances. The solver's reasoning was based on a misunderstanding of the question's requirement to find the difference between the field goal and touchdown pass, rather than the difference between the longest and second-longest distances mentioned.</p> <p>Instructions: To correct this mistake, the solver should explicitly extract the relevant numerical values from the passage and perform the subtraction operation directly. Here are the steps to implement the correction:</p> <p>Advice: 1. Extract all numerical values related to distances from the passage. 2. Identify the specific distances mentioned in the question (field goal and touchdown pass). 3. Subtract the shorter distance from the longer distance to get the correct answer. 4. Ensure the solver function returns a concise string as the answer.</p>
(D) Strategy and Patch	(E) Updated Policy	(F) Task Performance
<p>Strategy Extract and focus on the relevant numerical values and context-specific data from the passage before performing any calculations or comparisons.</p> <p>Patch 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 extract relevant numerical values and context-specific data from the passage.\n" "2. Please explain step by step.\n" "3. Directly answer the question.\n" "4. The answer MUST be a concise string.\n").strip(),)</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=0.5, num_of_response=1, role="read comprehension expert", return_dict_keys=["reasoning", "answer"], requirements=("1. Please extract relevant numerical values and context-specific data from the passage.\n" "2. Please explain step by step.\n" "3. Directly answer the question.\n" "4. The answer MUST be a concise string.\n").strip(),) return_dict = response[0] return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre>	<p>Passage: ...The total working population of the commune is 27,979. The rate of employment between 20 and 59 years old is 79% compared to the national average is 82.2%. There are 2,216 unemployed. In 1999 Antony had an unemployment rate of 7.9% which was much lower than the national average which was 12.9%...</p> <p>Question: How many percent lower is Antony's unemployment rate than the national average?</p> <p>Model Output with Policy: {"reasoning": "To find how many percent lower Antony's unemployment rate is compared to the national average, subtract Antony's unemployment rate from the national average and then divide by the national average, multiplying by 100. Calculation: ((12.9 - 7.9) / 12.9) * 100 = 46.44%", "answer": "46.44"} Correct Answer: '5'</p> <p>Model Output with Updated Policy: {"reasoning": "The national average unemployment rate is 12.9%, while Antony's unemployment rate is 7.9%. To find how many percent lower Antony's rate is, we calculate the difference: 12.9% - 7.9% = 5%. Thus, Antony's unemployment rate is 5 percentage points lower than the national average.", "answer": "5"} Correct Answer: '5'</p>

Figure 1: An example of policy repair via experience abstraction with POLARIS on the DROP 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).

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 partic-

ularly valuable in interactive environments where tool interfaces, data formats, or operational constraints evolve over time.

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.

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, 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 behav-

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  $\text{Memory} \leftarrow \emptyset$ 
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  $\text{SELF\_IMPROVE}(\pi, s, r, g)$ :
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  $\text{EXECUTE}(\pi, s, r, a, E)$ :
14  switch  $a.\text{name}$  do
15    case  $\text{self\_state}$  do
16       $s \leftarrow \text{SELF\_INSPECT}()$ 
17       $\text{Memory.append}(a, s)$ 
18    case  $\text{interact}$  do
19       $r, \mathcal{T} \leftarrow \text{EVALUATE}(\pi, E)$ 
20       $\text{Memory.append}(a, r)$ 
21       $\pi, s \leftarrow \text{REPAIR\_POLICY}(\pi, s, \mathcal{T}, a, r)$ 
22    case  $\text{self\_update}$  do
23       $\pi, s \leftarrow a.\text{code}$ 
24       $\text{Memory.append}(a, s)$ 
25    case  $\text{continue\_improve}$  do
26       $\pi, s \leftarrow \text{SELF\_IMPROVE}(\pi, s, r, g)$ 
27       $\text{Memory.append}(a, s)$ 
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  $\text{REPAIR\_POLICY}(\pi_t, s, \mathcal{T}, a, r)$ :
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  $\text{IntegratePatch}(\pi_t, s, p, n, a, r)$ :
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     $\text{Memory.append}(\text{self\_update}(\pi_{t+1}, s))$ 
17    return  $\pi_{t+1}, s$ 
18   $\text{Memory.append}(a, r + p)$ 

```

ior 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.

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 2–5**. 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 2), 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 3) 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.

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 2: 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 3: 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.

Patch Generation. For each synthesized strategy δ_j , the PatchGeneration operator (Algorithm 2; Figure 4) instantiates a minimal code patch p_j .

300
301
302

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 4: Prompt for generating code patches from policy repair strategies.

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 5. 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.

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 5: Prompt for integrating code patches into current policy.

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) $N=3$

	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) $N=5$

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

	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. For policy repair and experience abstraction, we consider a set of N failed instances from the validation set of each dataset.

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 validation 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 per-

formance; (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 $N = 5$. We observe fewer OOM errors with $N = 3$ 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 6 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

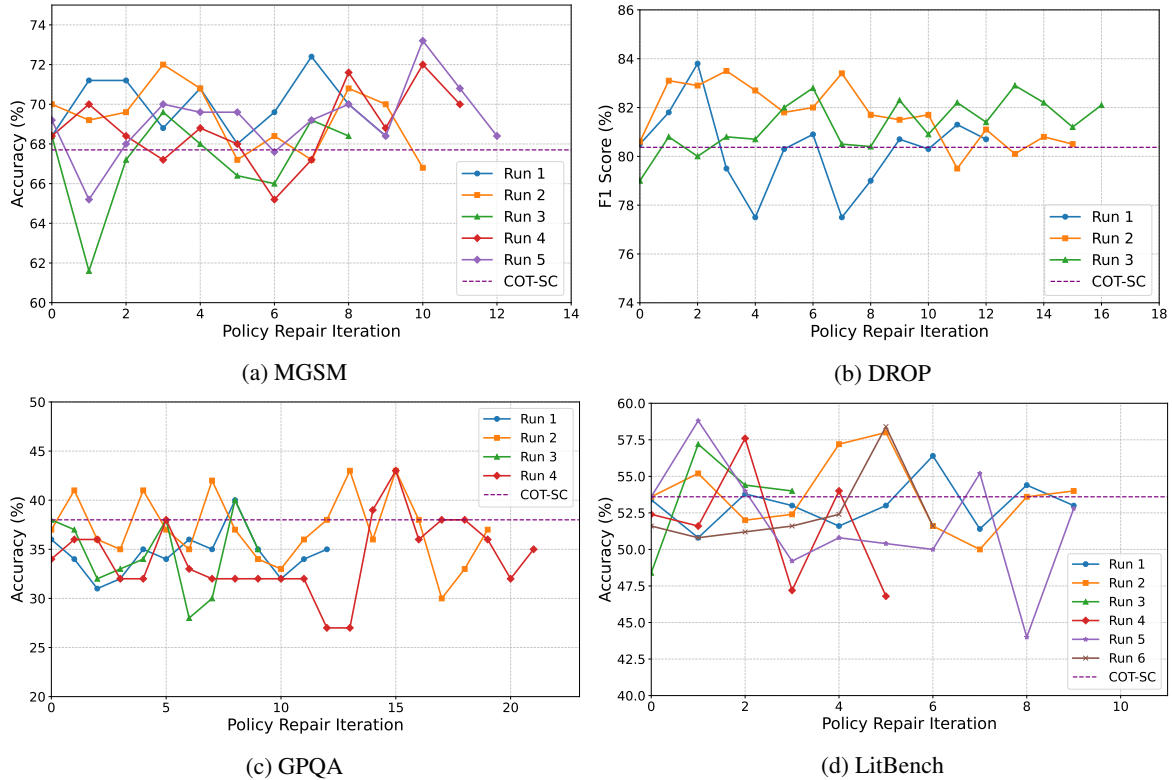


Figure 6: Successful evolution runs of POLARIS with performance improvement compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N=3$).

432 tool calls, redundant evaluations, or irrecoverable
 433 policy adjustments, consistent with prior reports on
 434 SLM limitations (Senel and Ozmen, 2025; Subra-
 435 manian et al., 2025). In initial experiments, models
 436 with comparable parameter counts (e.g., DeepSeek-
 437 Coder-6.7B-Instruct, Llama-3.1-8B-Instruct, and
 438 Mistral-7B-Instruct-v0.3) but limited tool-calling
 439 and meta-reasoning capabilities consistently failed,
 440 underscoring the challenges faced by SLMs.

441 **Policy repair through experience abstraction**
 442 **tends to capture solution complexity:** We illus-
 443 trate policy updates across datasets, showing how
 444 repairs introduce diverse enhancements such as
 445 complex instructions, post-processing steps, condi-
 446 tional logic, exception handling, and context-aware
 447 validation (see Figure 1 and Figures 14–24 in the
 448 Appendix). These observations highlight the role
 449 of experience abstraction in enabling nuanced, task-
 450 specific policy refinements. Additionally, the num-
 451 ber of repair iterations varies across datasets, under-
 452 scoring the need for adaptive, localized behavior in
 453 self-improving agents.

454 **Non-monotonic but consistent performance**
 455 **gain:** We observe that the agent can recover from
 456 local performance minima across datasets (see

Appendix, Section C.3, for a discussion on non-
 monotononic 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 effective-
 ness of POLARIS in resource-constrained settings.

6 Conclusion

We introduced POLARIS, a framework for recur-
 sive 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 up-
 dates, enabling post-deployment adaptation. Em-
 pirical results demonstrate consistent gains without
 supervision or retraining, highlighting the feasi-
 bility of stable, traceable self-referential learning
 and its potential for controlled, open-ended im-
 provement 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*.

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*.

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.

584	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 <i>First Conference on Language Modeling</i> .	636
585		637
586		638
587		639
588		
589	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. <i>arXiv preprint arXiv:2109.13202</i> .	640
590		641
591		642
592		643
593		644
594		645
595	Jürgen Schmidhuber. 2007. Gödel machines: Fully self-referential optimal universal self-improvers. In <i>Artificial general intelligence</i> , pages 199–226. Springer.	646
596		647
597		648
598	FA Senel and Hayri Baytan Ozmen. 2025. A comparative review of hallucination mitigation and performance improvement techniques in small language models. <i>Journal of Research and Design</i> , 2(1).	649
599		650
600		
601		
602	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 <i>The Eleventh International Conference on Learning Representations</i> .	651
603		652
604		653
605		654
606		655
607		
608	Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. <i>Advances in Neural Information Processing Systems</i> , 36:8634–8652.	656
609		657
610		658
611		659
612		
613	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. <i>arXiv preprint arXiv:2311.00344</i> .	660
614		661
615		662
616		663
617		664
618		665
619	Kenneth O Stanley and Joel Lehman. 2015. Why greatness cannot be planned: The myth of the objective.	666
620		667
621	Shreyas Subramanian, Vikram Elango, and Mecit Gungor. 2025. Small language models (slms) can still pack a punch: A survey. <i>arXiv preprint arXiv:2501.05465</i> .	668
622		669
623		670
624		671
625	Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. <i>Artificial intelligence</i> , 112(1-2):181–211.	672
626		673
627		674
628		675
629	Qwen Team. 2024. <i>Qwen2.5: A party of foundation models</i> .	676
630		677
631	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. <i>arXiv preprint arXiv:1901.01753</i> .	678
632		679
633		680
634		681
635		682
	Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. 2024. Knowledge editing for large language models: A survey. <i>ACM Computing Surveys</i> , 57(3):1–37.	683
		684
	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 <i>The Eleventh International Conference on Learning Representations</i> .	685
		686
	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 <i>International Conference on Learning Representations (ICLR)</i> .	687
		688
	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. <i>arXiv preprint arXiv:2410.04444</i> .	689
		690
	Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025. Darwin godel machine: Open-ended evolution of self-improving agents. <i>arXiv preprint arXiv:2505.22954</i> .	691
		692
	Quanjun 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. <i>CoRR</i> .	693
		694

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

In Figure 7, 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 8, we provide the helper agent prompt that helps correct the output format to valid JSON during the evaluation of the policy.

762 We provide examples in the prompt to obtain the
763 target behaviour.

764 C Additional Results and Analysis

765 C.1 Example runs of POLARIS

766 In Figures 1, 14, 15, and 16, we present exam-
767 ples from different datasets illustrating the steps
768 of the POLARIS algorithm. The initial policy fails
769 to correctly answer the question. Failure analysis
770 generates explanations, instructions, and advice for
771 the agent. During strategy synthesis, recommen-
772 dations are formulated to correct prior behavior by
773 updating the policy. The corresponding patch is
774 then integrated into the current policy, resulting in
775 improved task performance on unseen instances.

776 C.2 Policy update examples

777 In Figures 17–24, we present example policy up-
778 dates across datasets. We highlight changes to the
779 current policy relative to the previous policy using
780 green (additions) and red (deletions). The updates
781 span multiple dimensions, including the addition
782 of complex requirements, expert evaluators, data-
783 type changes, conditional statements, exception
784 handling, and specialized scoring parsers. These
785 niche updates underscore the role of experience
786 abstraction in enabling targeted policy refinements
787 over iterations.

788 C.3 Variance in performance

789 In Figures 12 and 13, we present a consolidated
790 view of performance variation for successful and
791 no-improvement runs of POLARIS across datasets.
792 We acknowledge that the raw reward/accuracy at
793 each self-improvement step can fluctuate, and we
794 do not claim monotone improvement of every in-
795 termediate candidate. However, this behavior is
796 expected for algorithms that: (i) operate in open-
797 ended search spaces, and (ii) deliberately explore
798 large, non-local changes to the policy or code.

799 Closely related settings such as deep reinforce-
800 ment learning and open-ended RL are well known
801 to exhibit high variance and occasional regressions
802 even under fixed hyperparameters and environ-
803 ments.

804 Henderson et al (Henderson et al., 2018) sys-
805 tematically document such instability and variance
806 across seeds in standard deep RL benchmarks and
807 argue that this variance is intrinsic to the methods
808 rather than an implementation bug. Patterson et al

(Patterson et al., 2024) similarly emphasize that per-
809 formance variation and instability are central em-
810 pirical phenomena in modern RL, and that sound
811 evaluation must explicitly account for them rather
812 than expect smooth, monotone curves. Bench-
813 marks designed for open-ended learning such as
814 MiniHack (Samvelyan et al., 2021) and Craftax
815 (Matthews et al., 2024) also explicitly highlight
816 that exploratory, open-ended agents typically show
817 jagged learning curves while still discovering sig-
818 nificantly better policies than baselines.

819 In our setting, the key quantity is therefore the
820 best-so-far performance of the agent, not the instan-
821 taneous performance of every transient candidate
822 produced during self-improvement. Our experi-
823 ments show that the final (or best-so-far) Gödel
824 agent produced by our method consistently and
825 substantially outperforms both the initial system
826 and strong non-Gödel baselines. This is analogous
827 to standard practice in anytime search and in rein-
828 forcement learning, where a potentially unstable
829 inner learner is wrapped by an outer loop that al-
830 ways retains the best model found so far.

831 Practically, a user or deployment scenario would
832 not expose intermediate, exploratory candidates.
833 Instead, one would keep a “champion” model and
834 only replace it when the self-improvement loop
835 discovers a clearly superior “challenger” based
836 on a stable evaluation protocol (a standard cham-
837 pion–challenger pattern from RL and online learn-
838 ing). This yields a monotone non-decreasing per-
839 formance profile for the deployed agent, even if
840 the internal search process remains volatile and
841 exploratory.

842 C.4 POLARIS runs with no improvement

843 In Figures 10 and 11, we present runs where the
844 agent fails to surpass the performance of the base
845 policy. Such cases are relatively rare compared to
846 successful runs across datasets. Moreover, in most
847 instances, the gap between the base policy and the
848 best performance achieved over iterations is mini-
849 mal. With a longer evolution horizon, we expect the
850 agent to recover and improve performance. This
851 behavior warrants further investigation using strate-
852 gies such as occasional resets and the integration
853 of pre-identified policy patches that are known to
854 yield performance gains, providing targeted boosts
855 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 7: 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: 110 + 16 = 127 g.",
 "answer": "127"
}
```

"))},

{"role": "user", "content": f"### Input JSON:{response}\n### Corrected JSON:"}

```

Figure 8: Helper agent prompt that helps correct the output format to valid JSON during the evaluation of the policy.

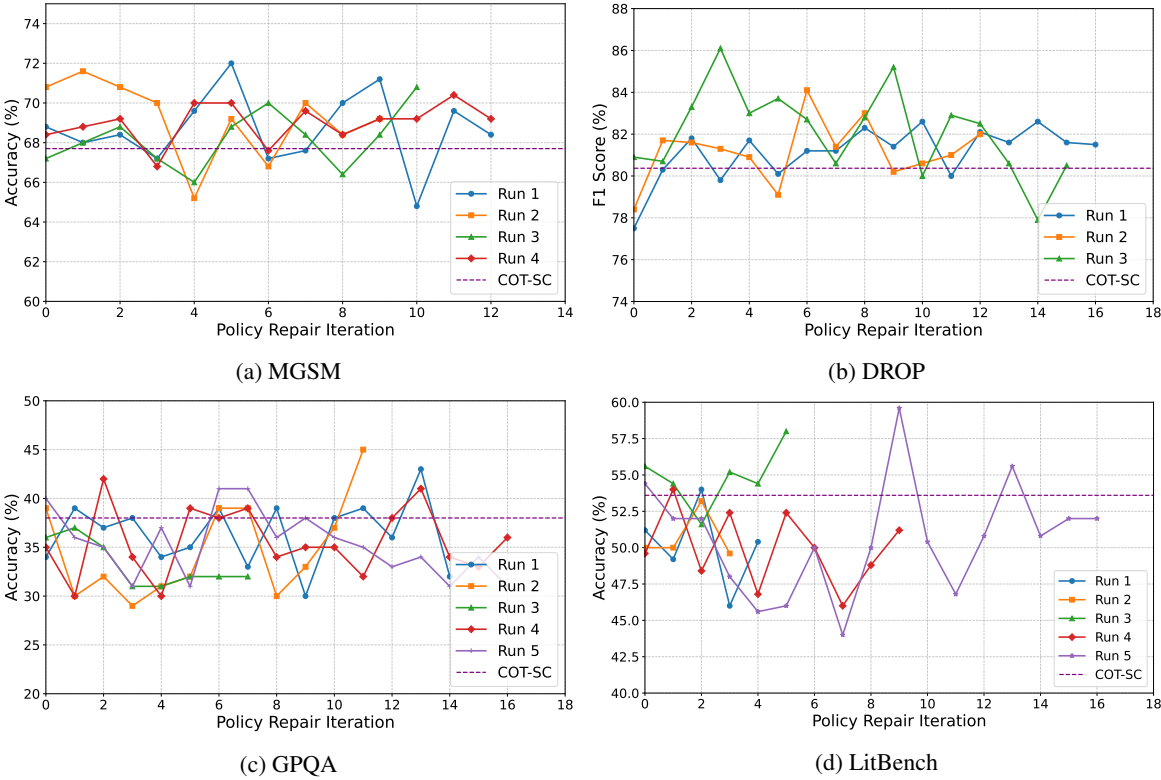


Figure 9: Successful evolution runs of POLARIS with performance improvement compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of five failed instances from the validation set of each dataset ($N=5$).

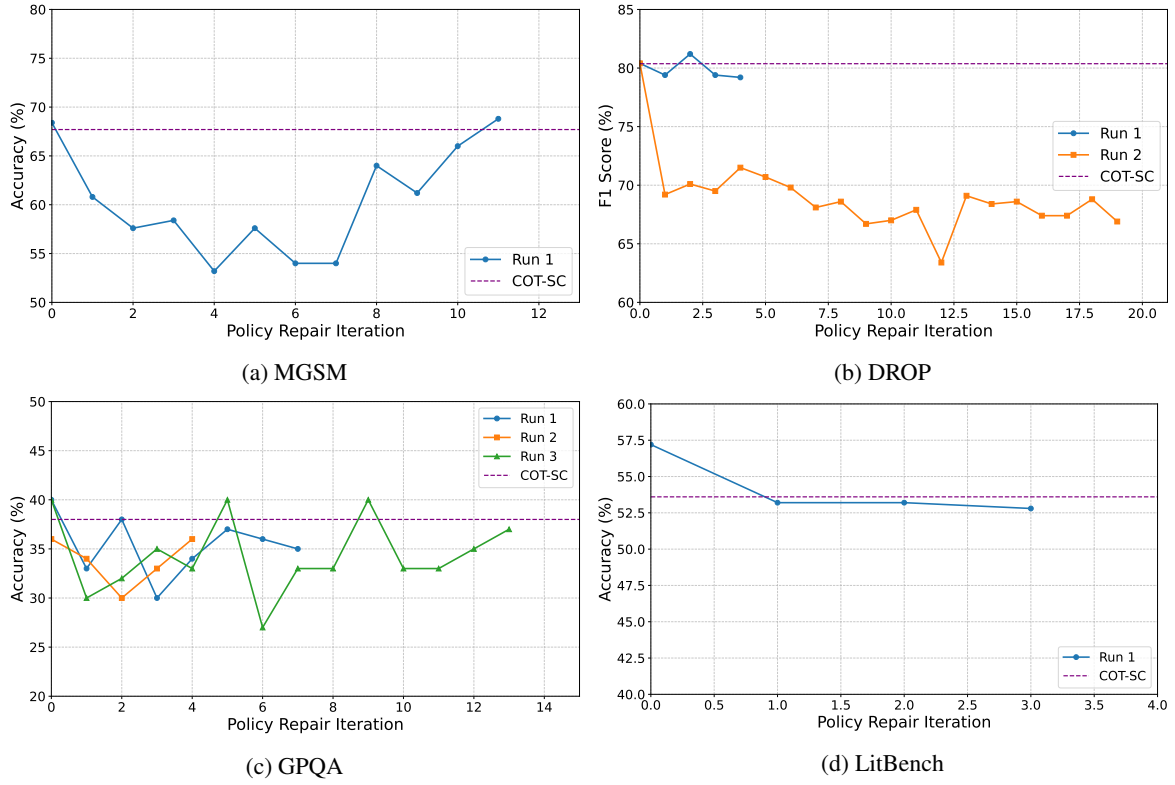


Figure 10: No Improvement runs of POLARIS with performance compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of three failed instances from the validation set of each dataset ($N=3$).

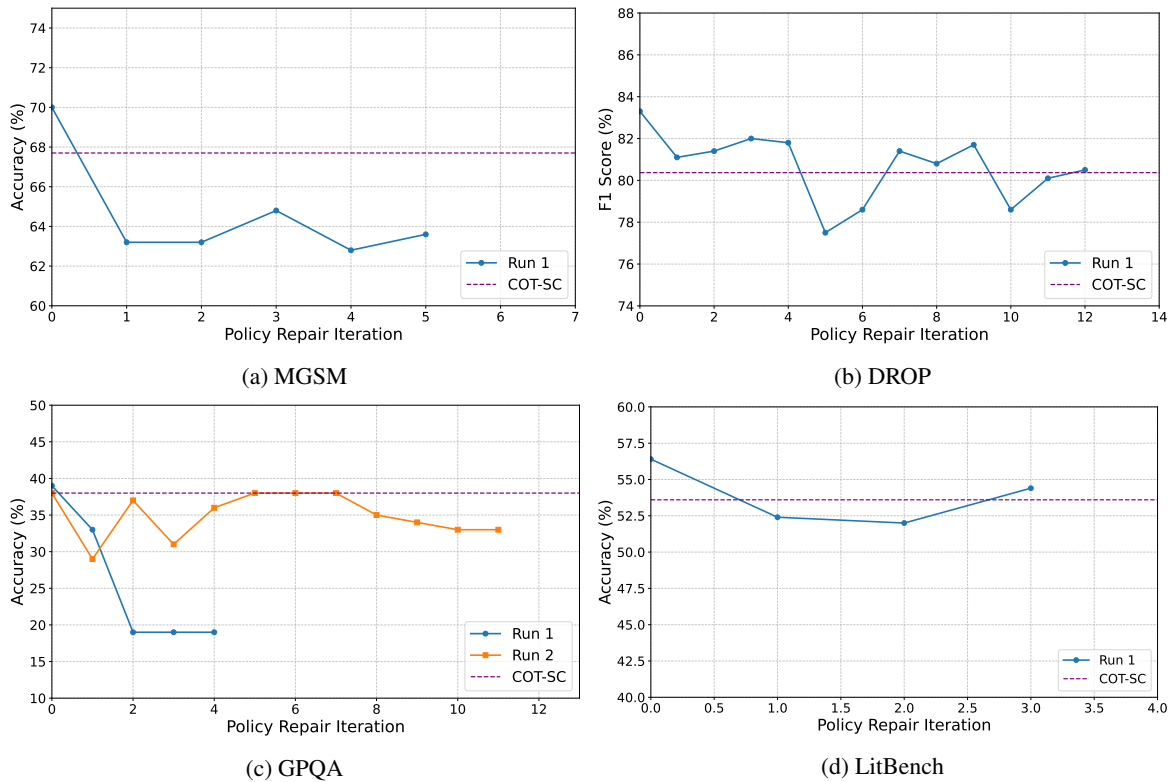


Figure 11: No Improvement runs of POLARIS with performance compared to the base policy and COT-SC. Policy Repair Iteration 0 shows the performance with the base policy. For policy repair and experience abstraction, we consider a set of five failed instances from the validation set of each dataset ($N=5$).

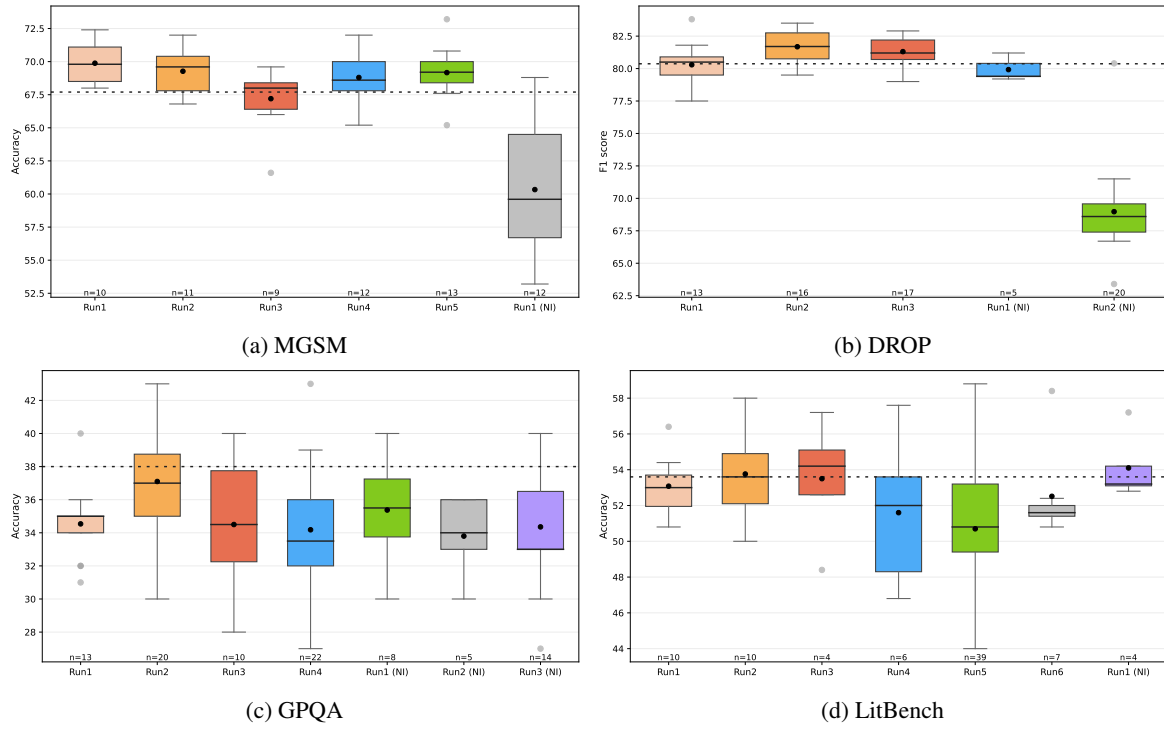


Figure 12: Performance variance across datasets for successful and no-improvement (NI) runs of POLARIS. Each plot shows the performance of the COT-SC baseline as a dotted horizontal line. The x-ticks indicate the sample size per run. Here, we consider a set of three failed instances from the validation set of each dataset ($N=3$).

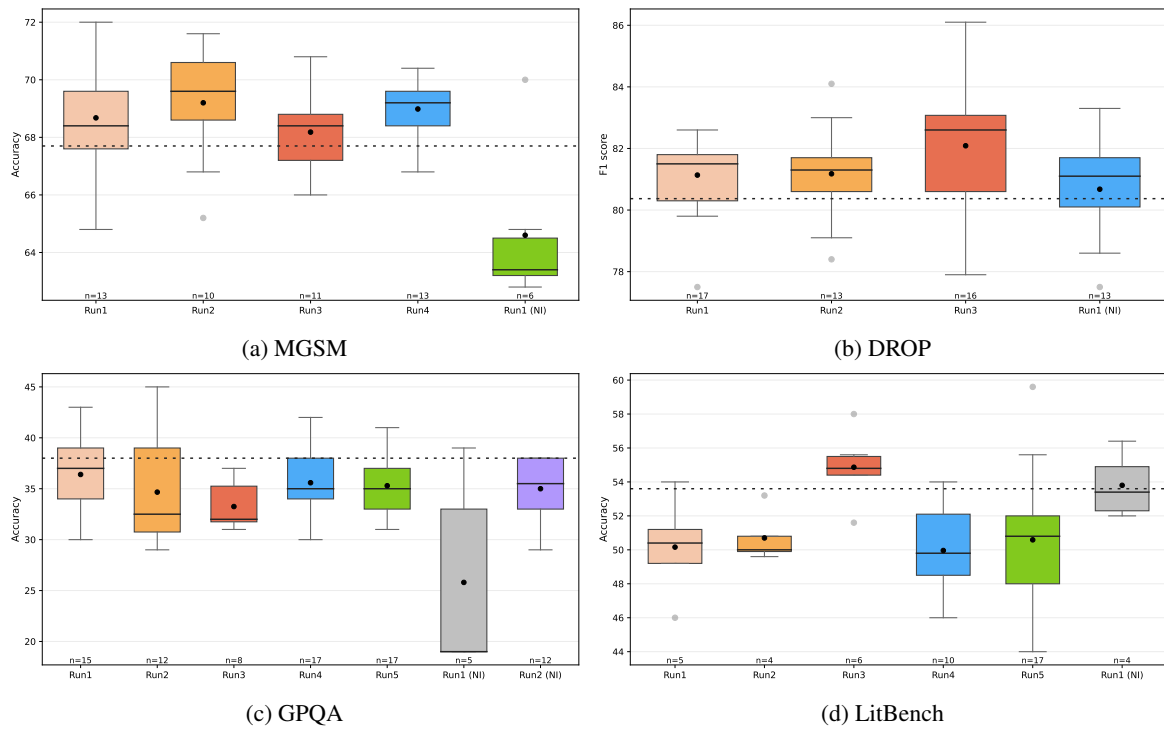


Figure 13: Performance variance across datasets for successful and no-improvement (NI) runs of POLARIS. Each plot shows the performance of the COT-SC baseline as a dotted horizontal line. The x-ticks indicate the sample size per run. Here, we consider a set of five failed instances from the validation set of each dataset ($N=5$).

| | | |
|--|--|--|
| <p>(A) Policy</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=0.7, 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>(B) Failed Tasks</p> <p>Question: Решите эту математическую задачу. В начале 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 в начале?</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"}
 Correct Answer: '60'</p> | <p>(C) Failure Analysis</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: 1. Parse the problem statement to extract the relevant numbers and operations. 2. Calculate the total allowance received over 8 weeks: 5 * 8 = 40. 3. Subtract this total from the final amount of money Bailey had: 100 - (5 * 8) = 60. 4. Return the result as an integer.</p> <p>Advice: To prevent similar failures in the future, consider the following advice: 1. Enhance the parsing logic in the solver function to ensure accurate extraction of numerical values and operations from the problem statement. 2. Implement additional checks to validate the correctness of the parsed information and the operations performed. 3. Utilize more robust error handling mechanisms to catch and correct logical errors during the problem-solving process.</p> |
| <p>(D) Strategy and Patch</p> <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> | <p>(E) Updated Policy</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", "")) 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>(F) Task Performance</p> <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"}
 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"}
 Correct Answer: '255'</p> |

Figure 14: 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).

| | | |
|---|--|--|
| <p>(A) Policy</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 ALPHABETS A or B or C or D.\n"), strip(),) return_dict = response[0] return_dict["answer"] = str(return_dict.get("answer", "")) return return_dict</pre> | <p>(B) Failed Tasks</p> <p>Question: If an equimolar mixture X of two liquids, which decolorizes bromine water, is treated with platinum when heated, then an equimolar mixture Y of two other liquids is formed as a result of disproportionation, which does not decolorize bromine water. Hydrogenation of both mixture X and mixture Y in the presence of platinum under rather severe conditions gives only one substance, a certain hydrocarbon Z, which is a constituent of mixture Y and widely used as a solvent. Substance Z does not react further with hydrogen. There are no conjugated multiple bonds in the molecules of the compounds of mixture X. Indicate the total number of hydrogen atoms in two liquids of mixture X. Choices: (A) 22 (B) 12 (C) 16 (D) 18</p> <p>Model Output: {reasoning: "Based on the information provided, the hydrocarbon Z has a mass fraction of hydrogen at 14.28%. The molecular formula for a hydrocarbon can be derived from its hydrogen content. If we consider that the total mass of the molecule is 100 units and hydrogen contributes 14.28 units, the remaining part (85.72 units) would be carbon. Using the atomic masses, if we let the number of carbon atoms be n, then the equation for hydrogen atoms becomes: 1.008 * 2n / (12 * n + 1.008 * 2n) = 14.28 / 100. Solving this, we get n = 10. Therefore, the hydrocarbon Z has 10 carbon atoms and 22 hydrogen atoms (since 2n = 22). This means that the hydrocarbon Z is a decane (C10H22). Since it is mentioned that Z is a constituent of mixture Y and is the product of hydrogenation of both mixtures X and Y, the composition of X must also reflect the same hydrogenation outcome. Given that X is an equimolar mixture of two liquids that do not have conjugated multiple bonds, and they decompose into Z upon hydrogenation, it implies each liquid in X must contain 11 hydrogen atoms (since 22/2 = 11). Therefore, the total number of hydrogen atoms in two liquids of mixture X is 22.", answer: "A"}
 Correct Answer: '23'</p> | <p>(C) Failure Analysis</p> <p>Explanation: The solver function failed because the reasoning provided was incorrect. Specifically, the solver incorrectly calculated the number of hydrogen atoms in the hydrocarbons of mixture X. The solver assumed that since the hydrogen mass fraction of the final hydrocarbon Z is 14.28%, and knowing that Z has 22 hydrogen atoms, each liquid in mixture X should have 11 hydrogen atoms, leading to a total of 22 hydrogen atoms for both liquids. However, this is incorrect because the problem states that the hydrogenation of both mixtures X and Y results in only one substance, Z, which has 22 hydrogen atoms. This implies that the total number of hydrogen atoms in both liquids of mixture X combined is 22, not 22 per liquid. The solver did not correctly interpret the problem statement and thus arrived at the wrong answer.</p> <p>Instructions: To correct the mistake, the solver needs to properly interpret the problem statement. Here are the steps to revise the solver function: 1. Understand that the hydrogenation of both mixtures X and Y results in a single hydrocarbon Z with 22 hydrogen atoms. 2. Recognize that the total number of hydrogen atoms in both liquids of mixture X is 22, not 22 per liquid. 3. Update the reasoning to reflect the correct interpretation: Since Z is the result of hydrogenation of both X and Y, and Z has 22 hydrogen atoms, the total number of hydrogen atoms in both liquids of mixture X is 22. 4. Ensure the solver function returns the correct answer 'D'.</p> <p>Advice: To prevent similar failures in the future, always ensure that the solver function thoroughly interprets the problem statement and aligns the logical steps with the given constraints. Regularly review the solver function's logic and validate it against the problem requirements. Additionally, utilize external collaboration via LLM to cross-check interpretations and refine logical steps.</p> |
| <p>(D) Strategy and Patch</p> <p>Strategy
Improve the response handling by ensuring the answer is always one of the specified alphabets (A, B, C, D) and add reasoning steps explicitly.</p> <p>Patch
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", "")) # 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 </p> | <p>(E) Updated Policy</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", "")) # 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>(F) Task Performance</p> <p>Question: Which of the answers below explains the most likely reason for it?</p> <p>Model Output with Policy: {reasoning: "Given the setup, the receptor-eGFP fusion protein should be expressed if the construct is correctly translated. The presence of the loxP-stop-loxP and lox2272-stop-lox2272 cassettes suggests that the eGFP could be absent due to the stop codons being removed upon Cre recombinase activity. However, since the question states that the construct is bred into a mouse model and crossed with SOX10-Cre hemizygous mice, and no green signal is observed, it indicates that the receptor-eGFP fusion might be stuck in the Golgi apparatus, preventing its proper localization and thus not allowing it to emit fluorescence. The other options are less likely because they would affect either the overall expression or the translation initiation, which would have been detected in the Western blot analysis.", answer: "D"}
 Correct Answer: 'B'</p> <p>Model Output with Updated Policy: {reasoning: "Let's analyze the options based on the information provided. The key points include the bicistronic lox-Cre vector with the ligand and receptor fused to mCherry and eGFP respectively, and the presence of loxP and lox2272 cassettes which indicate that the expression of eGFP is conditional. Given that the offspring do not show a green signal, the most likely explanation is that the receptor-eGFP construct is not being translated into functional eGFP due to a frameshift mutation introduced by the loxP-stop-loxP cassette, which would prevent the reading frame from continuing correctly beyond the stop codon. Therefore, the eGFP part of the construct is not in frame, leading to no green fluorescence. Options (A), (C), and (D) do not directly address the issue of the non-functional eGFP construct.", answer: "B"}
 Correct Answer: 'B'</p> |

Figure 15: An example of policy repair via experience abstraction with POLARIS on the GPQA dataset. The initial policy (A) failed to correctly answer the question (B). The failure analysis revealed that the Agent failed to provide the correct response because the reasoning provided was incorrect (C). During strategy synthesis (D), a recommendation is made to add reasoning steps explicitly along with the check for output response format. 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 16: 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"
        ).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 17: 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 logic to break down the problem and validate each part while deleting the comment for post-processing the response.

```

=====
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 18: 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 19: 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 20: 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 21: 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 23: 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 24: 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.