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. Open ended behaviour matters for agents that must keep acquiring new solution patterns, so the experience abstraction step is designed to accumulate reusable strategies, offering early signs of greater novelty, diversity, and policy complexity even though our study evaluates fixed reasoning benchmarks. On MGSM, DROP, and GPQA (covering arithmetic reasoning, compositional inference, and graduate level problem solving), a 7 billion parameter model with POLARIS yields consistent gains over the base policy and produces interpretable patches.

1 Introduction

Intelligence is not a fixed mapping from inputs to outputs; it is a process that improves itself. Decades ago, Good asked what might follow if a machine could examine its own procedures and build better ones (Good, 1966), and Schmidhuber framed this as a Gödel Machine that rewrites its own code when improvement can be proven in advance (Schmidhuber, 2007). A parallel thread argues for open-ended growth: systems should keep expanding their repertoires rather than merely optimising a static score. This shows up in novelty search and quality-diversity (Lehman and Stanley, 2011; Stanley and Lehman, 2015), in co-evolving environments such as POET (Wang et al., 2019), and in position pieces that explicitly argue for open-endedness as essential to reaching stronger capabilities (Hughes et al., 2024), with surveys and taxonomies clarifying the landscape (Song, 2022). These ideas motivate work on agents that can improve themselves and accumulate reusable strategies over time.

Modern language agents offer a practical arena to revisit these questions. Much current self-improvement either optimises responses—via reasoning and acting (ReAct) (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)—or changes parameters through 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). These strategies help, but they often blur what was learned and where the change lives: is it an output level fix for this instance, or a persistent change to the agent? Abstraction and reuse is also studied through options and hierarchical reinforcement learning (Sutton et al., 1999; Bacon et al., 2017) and fast adaptation via meta learning (Finn et al., 2017), but again without making the agent's policy an explicit object of inspection and repair. Viewed through Dennett's taxonomy, response level self correction is Skinnerian (trial and error shaping of outputs), parameter editing is Darwinian (selection over weights without explicit representation), while a Gödel agent targets Popperian rehearsal of policy revisions and Gregorian accumulation of external "mind tools" in memory (Dennett, 2008; Popper and Popper, 1979; Gregory, 1983).

We take a policy explicit route. A Gödel agent inspects its own policy and traces (self reference), proposes concrete edits, and updates the current

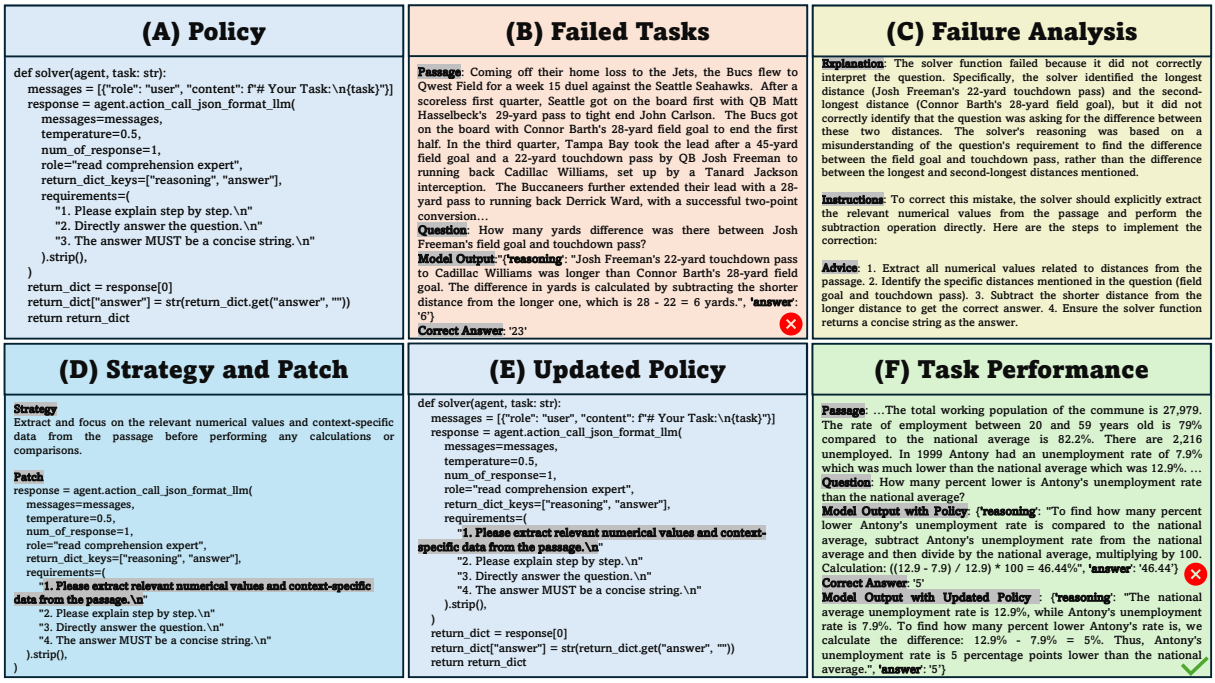


Figure 1: An example of policy repair via experience abstraction with POLARIS on the DROP dataset. The initial policy (A) failed to correctly answer the question (B). The failure analysis revealed that the Agent struggled with handling numerical data (C). During strategy synthesis (D), a recommendation 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).

policy with the suggested edits. POLARIS realises this with policy repair via experience abstraction: failures are analysed, abstracted into general strategies, rendered as minimal code patches, and integrated into current policy with retries and archival so that evolution is stable and auditable. This separates response level self correction (which leaves the policy unchanged) and parameter level tuning (opaque at the rule level) from policy level change in code, where small, auditable patches persist in the policy and are reused on unseen instances within a benchmark. We treat meta reasoning as a concrete loop: the agent diagnoses its own errors, abstracts a strategy, proposes a minimal policy edit, and update the policy with retries.

We evaluate POLARIS on three challenging benchmarks: MGSM (multilingual grade-school math) for structured numerical reasoning (Shi et al.), DROP (discrepancy-based reading comprehension) for compositional inference (Dua et al., 2019), and GPQA (graduate-level problem solving) for high-complexity reasoning (Rein et al.). Across all three datasets, the POLARIS framework demonstrates systematic accuracy improvements, interpretable policy patches, and transferable abstractions, showing that recursive self-improvement is feasible even with compact SLMs such as Qwen2.5-7B.

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, DROP, and GPQA**, demonstrating consistent performance gains and interpretable improvements in reasoning capabilities.

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 engines 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 produce 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. POLARIS integrates these principles by framing recursive self-improvement not as a one-off adaptation but as an open-ended process in which failures yield reusable strategies that refine future behavior.

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. Building on the theoretical foundation of the Gödel Machine (Schmidhuber, 2007), 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. In this way, Gödel Agent offers a practical instantiation of computational self-reflection and evolution in language-based reasoning systems.

At its core, Gödel Agent structures self-improvement as a recursive optimization loop guided by introspection and empirical feedback. 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 per-

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

```
▷ Initialize empty set to store all the policy improvement strategies
δ ← ∅
▷ Initialize Agent memory, used to store action taken and action result
Memory ← ∅
▷ Introspect and retrieve agent's own code.
s ← SELF_INSPECT()
▷ Evaluate current policy, get the policy feedback r and list of failed tasks 𝒯
r, 𝒯 ← EVALUATE(π_0, E)
▷ Perform recursive self-improvement.
π, s ← SELF_IMPROVE(π_0, s, r, g)
return π, s

Function SELF_IMPROVE(π, s, r, g):
    a_1, ..., a_n ← f_0(π, s, r, g)
    for a_i in a_1, ..., a_n do
        π, s, r ← EXECUTE(π, s, r, a_i, E)
    return π, s

Function EXECUTE(π, s, r, a, E):
    switch a.name do
        case self_state do
            s ← SELF_INSPECT()
            Memory.append(a, s)
        case interact do
            r, 𝒯 ← EVALUATE(π, E)
            Memory.append(a, r)
            π, s ← REPAIR_POLICY(π, s, 𝒯, a, r)
        case self_update do
            π, s ← a.code
            Memory.append(a, s)
        case continue_improve do
            π, s ← SELF_IMPROVE(π, s, r, g)
            Memory.append(a, s)
    return π, 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

```
Function REPAIR_POLICY(π_t, s, 𝒯, a, r):
    ▷ Initialize empty set to store Self-Reflections of each task
    A ← ∅
    foreach τ_i in 𝒯 do
        ▷ Get self-reflection A_i for each task
        A_i ← AnalyzeFailure(π_t, s, τ_i)
        A ← A ∪ {A_i}
    ▷ Get policy improvement strategies from A
    δ ← δ ∪ {StrategySynthesis(π_t, s, A)}
    ▷ Get code patch for each strategy
    p ← PatchGeneration(π_t, s, δ)
    ▷ Number of retries left
    n ← 3
    ▷ Integrate patches in current policy π_t
    π_{t+1}, s ← IntegratePatch(π_t, s, p, n, a, r)
    return π_{t+1}, s

Function IntegratePatch(π_t, s, p, n, a, r):
    ▷ Get the updated policy by integrating patches p
    π_{t+1} ← UpdatePolicy(π_t, s, p)
    ▷ Retry, if the updated policy is not executable
    if π_{t+1} exists then
        if π_{t+1} is not executable & n>0 then
            ▷ If failed, retry and decrease the number of retries left
            π_{t+1}, s ← IntegratePatch(π_t, s, p, n−1)
        ▷ Push updated policy into the codebase using self_update and update the memory
        Memory.append(self_update(π_{t+1}, s))
        return π_{t+1}, s
    ▷ Upon failing to update the policy, append code patches into memory
    Memory.append(a, r + p)
    ▷ Continue to evolve
```

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.

formance data. This process yields an explicit representation of the agent's current capabilities and limitations.

2. **Execution (Interact):** During execution, the policy π operates as an LLM-based reasoning and acting module that engages with the environment through natural language interactions, tool use, and task-specific actions. The agent records both its intermediate reasoning traces and environmental responses, which serve as empirical evidence for evaluating and refining its subsequent behavior.
3. **Self-Modification (Self-Improve):** Using the improvement logic I, the agent evaluates its performance and proposes candidate code edits or rewrites. These modifications may target the policy π to enhance problem-solving behavior or the improvement logic I itself to refine the way updates are reasoned about. Large language models serve as the generative engine for proposing, critiquing, and verifying such modifications.
4. **Recursive Continuation (Continue-Improve):** After each modification is integrated, the agent re-enters the introspection phase. This recursive loop allows both π and I to evolve jointly, producing progressively more abstract forms of self-repair and adaptation.

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

3.2 POLARIS: Policy Repair through Experience Abstraction

POLARIS implements recursive self-improvement for small language models (SLMs) as a disciplined cycle of reflection, abstraction, and repair. The framework converts execution failures into validated code-level updates while maintaining 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

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

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.

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.

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

$$A_i = (diagnosis_i,\ revision_i,\ prevention_i)$$

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.

Patch Generation. For each synthesized strategy δ_j, the `PatchGeneration` operator (Algorithm 2; Figure 4) instantiates a minimal code patch p_j. Each patch modifies only the lines required to implement δ_j and excludes any explanatory text. A lightweight validator checks syntax and formatting before a patch enters integration. The resulting patch set is denoted $\mathcal{P} = \{p_j\}$. Emphasizing locality and minimality ensures that every modification remains interpretable and that the agent's policy evolves through small, verifiable updates.

Patch Integration. Patch integration follows **Algorithm 2** and Figure 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.

Each operator in POLARIS serves a well-defined

role. Failure analysis grounds modification in self-reflection, strategy synthesis compresses experience into reusable knowledge, patch generation renders this knowledge executable, and integration enforces conservative, verifiable change. Together they realize a recursive process in which the agent incrementally restructures its own policy through introspection and abstraction.

4 Experiments

We evaluate our framework on three reasoning and comprehension benchmarks: **MGSM**, **DROP**, and **GPQA**, chosen to cover diverse reasoning modalities and difficulty levels. MGSM tests arithmetic reasoning and compositional generalization, DROP evaluates discrete reasoning over natural text, and GPQA assesses advanced factual and conceptual reasoning. For MGSM and DROP, we use 50 validation and 250 test samples. For GPQA, we evaluate on 20 validation and 100 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.

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

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

1. **Chain-of-Thought Self-Consistency (COT-SC)** (Wang et al.): Five reasoning paths are sampled per query, and the most frequent answer is chosen. We use the same validation and test splits as our experiments to report the performance. It is one of the best performing baselines reported in (Yin et al., 2024).
2. **Gödel Agent** (Yin et al., 2024): Direct replication with Qwen2.5-7B-Instruct led to repeated OOM failures before 10 hours; thus quantitative comparison was infeasible.

5 Results and Analysis

We provide a summary of various runs of POLARIS in Table 1. Our observations indicate that the success rate of runs is approximately 30-40% with N=3 and 40-60% with N=5. Evolution on MGSM and GPQA yields a higher rate of successful runs compared to the DROP dataset. This disparity may be attributed to the larger context size present in the DROP dataset, which can lead to OOM errors. Additionally, we note a greater degree of variation in performance for N=3 compared to N=5. Furthermore, we observe that the experience abstraction tends to become more generic as N increases to 5, in contrast to N=3. These insights underscore the challenges faced by SLMs in effectively abstracting strategies from a larger and more diverse set of self-reflections on failed tasks.

We present the successful evolution runs in Figure 6 and the Appendix. Our analysis shows a varying number of policy repair iterations across different datasets, emphasizing the importance of adaptive, task-specific localized behavior of the

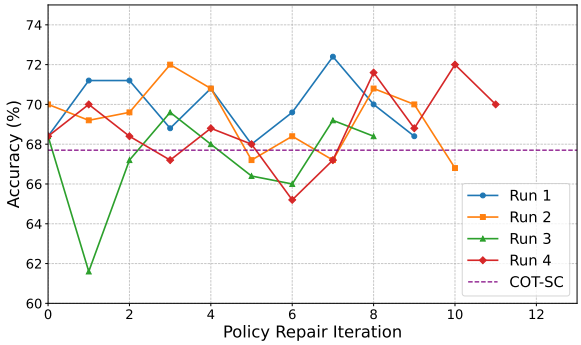
	MGSM	DROP	GPQA
Successful	4	3	4
No improvement	2	2	4
Unsuccessful	4	5	2
Total	10	10	10

(a) N=3

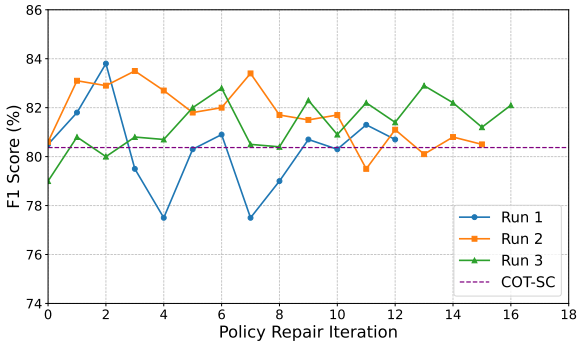
	MGSM	DROP	GPQA
Successful	3	2	3
No improvement	1	1	1
Unsuccessful	1	2	1
Total	5	5	5

(b) N=5

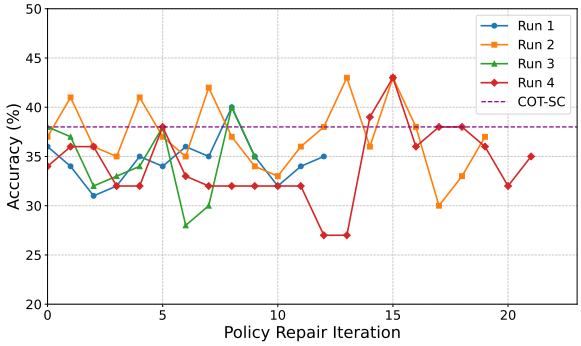
Table 1: A summary of various runs of POLARIS on datasets. For policy repair and experience abstraction, we consider a set of N failed instances from the validation set of each dataset.



(a) MGSM



(b) DROP



(c) GPQA

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).

agent. Moreover, we observe a notable pattern indicating that the agent can rebound from local performance minima across datasets. This finding emphasizes the importance of experience abstraction, which facilitates complex and niche updates to the policy (see example in Figure 1).

The model attains 67.7% accuracy on MGSM, 80.37% F1 score on DROP, and 38.0% accuracy on GPQA with the COT-SC baseline. We measure the maximum relative gains achieved by our self-evolution framework over the COT-SC baseline. For each dataset, we compute the maximum improvement in validation accuracy (MGSM, GPQA) or F1 score (DROP) across all successful runs for $N = 3$ and $N = 5$. For $N = 3$, the agent achieves +4.0% on MGSM, +3.9% on DROP, and +9.0% on GPQA. When increasing the reflection set size to $N = 5$, performance trends remain consistent, with a modest F1-score gain on DROP (+5.7%) and stable improvements on MGSM (%3.6) and GPQA (%9). The consistent improvement demonstrates the effectiveness of POLARIS under resource-constrained setting.

6 Conclusion

We presented POLARIS, a framework that operationalizes recursive self-improvement for small language models. Unlike prior approaches such as the Gödel Agent, which assume large-model capacity and unconstrained self-rewrite, POLARIS introduces a structured repair cycle that grounds self-improvement in interpretable, verifiable updates. By integrating self-reflection, strategy abstraction, patch generation, and conservative policy integration, the framework enables smaller agents to iteratively refine their behavior while preserving stability and traceability. Empirical evaluation on reasoning and comprehension benchmarks demonstrates that POLARIS improves task performance through successive refinement cycles without external supervision or retraining. More broadly, this work advances the feasibility of self-referential learning in resource-constrained agents, suggesting a path toward open-ended yet controlled self-improvement in future language model systems.

7 Limitations

POLARIS provides a practical approach to recursive self-improvement in small language models, yet some limitations remain. The reduced 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. 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*.

Daniel C Dennett. 2008. *Kinds of minds: Toward an understanding of consciousness*. Basic Books.

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.

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*.

Richard L Gregory. 1983. Mind in science. a history of explanations in psychology and physics. *Group Analysis*, 16(1):88–90.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktäschel. 2024. Position: open-endedness is essential for artificial superhuman intelligence. In *Proceedings of the 41st International Conference on Machine Learning*, pages 20597–20616.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. 2022. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089*.

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 Wiegreffe, 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.

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.

Karl Raimund Popper and Karl R Popper. 1979. *Objective knowledge: An evolutionary approach*, volume 49. Clarendon press Oxford.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Jürgen Schmidhuber. 2007. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, and 1 others. Language models are multilingual chain-of-thought reasoners. In *The Eleventh International Conference on Learning Representations*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Asiiah Song. 2022. A little taxonomy of open-endedness. In *ICLR Workshop on Agent Learning in Open-Endedness*.

Kenneth O Stanley and Joel Lehman. 2015. Why greatness cannot be planned: The myth of the objective.

Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. 2019. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*.

Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. 2024. Knowledge editing for large language models: A survey. *ACM Computing Surveys*, 57(3):1–37.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. 2024. G\" odel agent: A self-referential agent framework for recursive self-improvement. *arXiv preprint arXiv:2410.04444*.

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. *CoRR*.

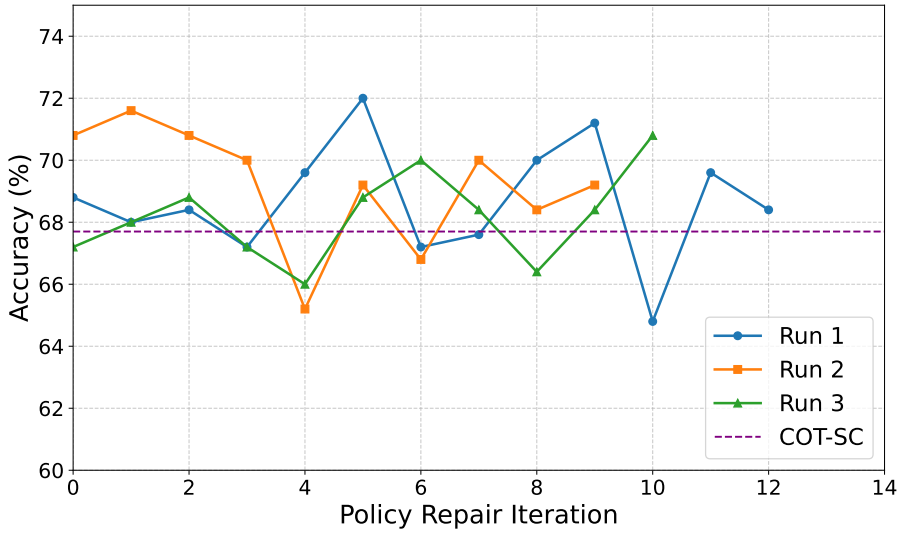
Goal prompt

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.

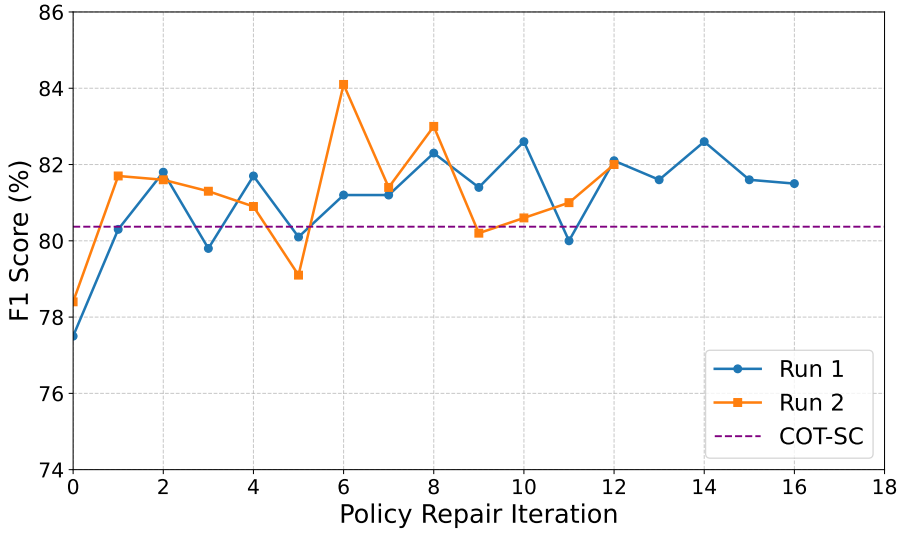
Core Methods:
+ **evolve**: Continuously enhance performance by interacting with environment.
+ **execute_action(actions)**: Execute actions based on analysis or feedback.
+ **solver(agent_instance, task_input: str)**: Solve the target task using current 'agent_instance' capabilities, and objects created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

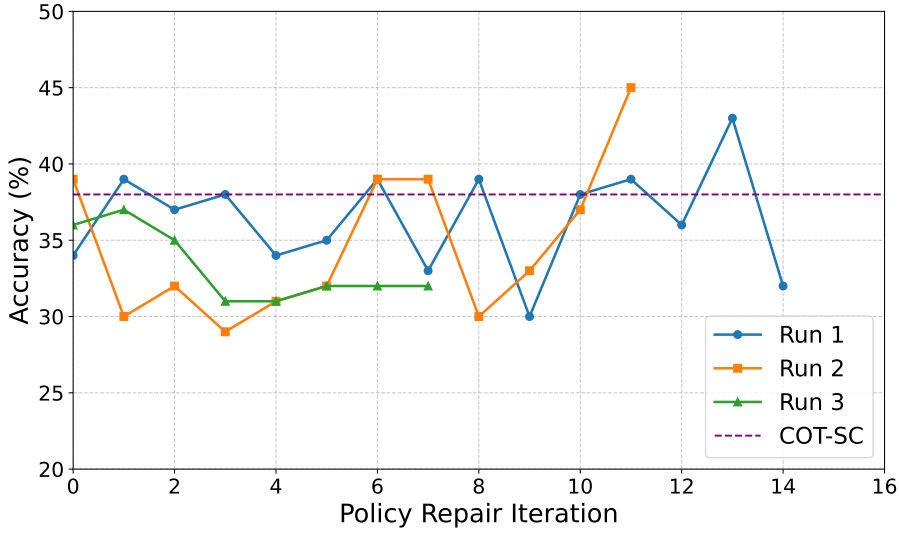
Figure 7: Goal prompt of the agent with the capabilities, core methods, and the guiding principles.



(a) MGSM



(b) DROP



(c) GPQA

Figure 8: 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).