LEARNING TO SELF-EVOLVE

Xiaoyin Chen[1,2*] **Canwen Xu**[3] **Yite Wang**[3] **Boyi Liu**[3] **Zhewei Yao**[3] **Yuxiong He**[3]
[1]Mila – Quebec AI Institute [2]University of Montreal [3]Snowflake

ABSTRACT

We introduce Learning to Self-Evolve (LSE), a reinforcement learning framework that trains large language models (LLMs) to improve their own contexts at test time. We situate LSE in the setting of test-time self-evolution, where a model iteratively refines its context from feedback on seen problems to perform better on new ones. Existing approaches rely entirely on the inherent reasoning ability of the model and never explicitly train it for this task. LSE reduces the multi-step evolution problem to a single-step RL objective, where each context edit is rewarded by the improvement in downstream performance. We pair this objective with a tree-guided evolution loop. On Text-to-SQL generation (BIRD) and general question answering (MMLU-Redux), a 4B-parameter model trained with LSE outperforms self-evolving policies powered by GPT-5 and Claude Sonnet 4.5, as well as prompt optimization methods including GEPA and TextGrad, and transfers to guide other models without additional training. Our results highlight the effectiveness of treating self-evolution as a learnable skill.

1 INTRODUCTION

The ability to adapt and evolve in response to environmental feedback has long been considered central to human intelligence (Piaget, 1952; Sternberg, 2019). A chess player improves by analyzing past games; a software engineer grows more proficient with a codebase through months of daily work. In both cases, experience accumulates and the person adjusts their approach accordingly. Current large language model (LLM) training pipelines exhibit a similar dynamic, particularly at the post-training stage, where reinforcement learning (RL) refines the behavior of the model on its own generated data (Lightman et al., 2023; OpenAI; DeepSeek-AI, 2025). However, this learning stops once training ends. At deployment, an LLM applies the same policy regardless of how many problems it has solved in a domain, and discards all accumulated experience once the context resets. This gap between static deployment and dynamic adaptation motivates the study of *test-time self-evolving* systems: systems that continuously update themselves in response to new observations at test time.

Test-time self-evolution can be characterized along at least two dimensions: *how* the policy is updated and *when*. On one end of the first dimension, gradient-based methods modify model parameters directly; on the other, prompt-based methods rewrite the model context while keeping parameters frozen. Along the second dimension, *intra-episode* evolution updates the policy within a single episode: the model revisits its own attempts and refines its answer to a particular problem, trading additional compute for instance-level gains (Shinn et al., 2023; Kumar et al., 2025; Yuksekgonul et al., 2026). *Inter-episode* evolution updates the policy after one or more completed episodes and applies the result to new problems, extracting transferable knowledge that generalizes across tasks (Yin et al., 2024; Hu et al., 2025; Zhang et al., 2025b).

We focus on inter-episode, prompt-based self-evolution: an LLM observes its performance on a batch of problems and rewrites its own context to improve on the next batch. Several recent works explore this direction through automatic prompt optimization (Khattab et al., 2024; Agrawal et al., 2025; Yuksekgonul et al., 2024), self-referential updates (Fernando et al., 2024; Zhao et al., 2024; Zhang et al., 2025b; Hu et al., 2025; Zhang et al., 2025c), and agentic memory systems (Zhang et al., 2025a;c; Chhikara et al., 2025). These methods, however, rely entirely on the inherent ability of the

*This work was done during Xiaoyin's internship at Snowflake. Correspondence should be addressed to Canwen Xu: canwen.xu@snowflake.com.

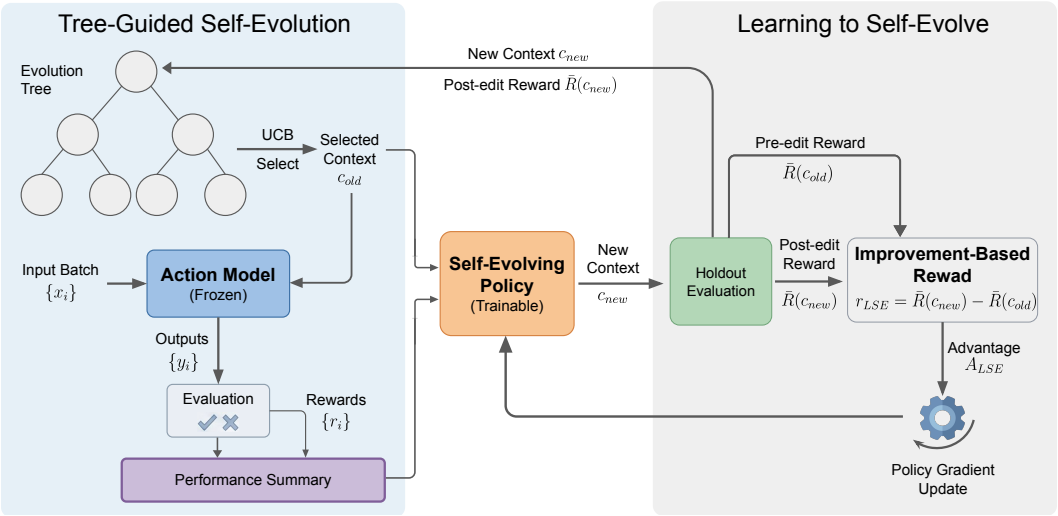


Figure 1: Overview of Learning to Self-Evolve (LSE). **Left:** Tree-guided self-evolution at test time. Upper Confidence Bound (UCB) selection chooses a context from the evolution tree; the action model generates outputs for a new batch of problems; the self-evolving policy receives the performance summary and proposes a revised context. **Right:** LSE trains the self-evolving policy via RL with an improvement-based reward computed as the difference between post-edit and pre-edit performance.

LLM to analyze feedback and propose better context. The model is never explicitly trained for this self-improvement task.

We argue that self-evolution poses a reasoning challenge distinct from other reasoning domains. The process, in essence, shares the structure of an RL problem. An RL optimizer relies on dedicated algorithms to assign credit, estimate gradients, and balance exploration against exploitation. In self-evolution, the model must perform all three implicitly, through natural language reasoning alone. It must judge which parts of the current context help and which hurt, anticipate how a revision will change downstream behavior, and decide whether to refine what works or try something new. These demands motivate explicit optimization for self-evolution.

We propose **Learning to Self-Evolve (LSE)**, an RL framework that explicitly trains an LLM to be an effective self-evolving policy. Rather than optimizing over the full multi-step evolution trajectory, LSE simplifies training to a single step: the model receives the current context and a performance summary, and produces a better context. Each edit is rewarded by the *improvement* in downstream performance, instead of the absolute post-edit score. At test time, we leverage a tree-guided evolution loop that allows the system to explore and backtrack across possible contexts.

We evaluate LSE on Text-to-SQL generation and general question answering. Despite using only a 4B-parameter model, the LSE-trained policy outperforms self-evolving policies powered by frontier models such as GPT-5 and Claude Sonnet 4.5, as well as prompt optimization methods such as GEPA and TextGrad. Our contributions are as follows:

- We formalize test-time inter-episode self-evolution and operationalize it through prompt-based updates with tree-guided search (§3.1, §3.2).
- We propose LSE, an RL framework that explicitly trains the self-evolving policy with an improvement-based reward (§3.3).
- We show that a 4B-parameter model trained with LSE outperforms larger untrained models and prompt optimization methods, and transfers to guide other models without additional training (§4).

2 Related Work

The term *self-evolution* has been used to refer to many different concepts in recent LLM research. We organize the landscape into two broad categories. *Training-time self-evolution* focuses on using LLMs to generate their own training data and learning signals during training. *Test-time self-evolution* enables a policy to continue updating itself after training, adapting dynamically based on experience accumulated during deployment.

Training-time self-evolution. A growing body of work leverages LLMs to generate their own data and learning signals during training. RL-based post-training has the model produce reasoning traces and optimizes them against verifiable rewards (Lightman et al., 2023; OpenAI; DeepSeek-AI, 2025). Bootstrapping methods such as STaR (Zelikman et al., 2022) iteratively generate candidate rationales and fine-tune on the correct ones. Self-rewarding approaches (Yuan et al., 2024; Zhao et al., 2025b) extend this by using the model itself as the reward signal. Absolute Zero (Zhao et al., 2025a) takes this to its extreme: a single model both proposes and solves tasks with no external data, using a code executor as the sole source of verifiable reward. While these methods produce stronger models, the resulting policy remains static once training ends. Our work addresses a complementary problem: enabling the policy to continue improving at test time.

Test-time self-evolution. A static policy cannot accommodate distribution shifts encountered at test time. Test-time self-evolution addresses this by enabling the model to self-update based on its own experience after deployment. This capability spans two temporal scales. *Intra-episode* methods improve on a single problem instance by allocating additional compute. Reflexion (Shinn et al., 2023) prompts the model to reflect on failed attempts and retry, SCoRe (Kumar et al., 2025) trains self-correction through RL, and TTRL (Zuo et al., 2025) applies RL directly at test time using majority voting as a proxy reward. TTT-Discover (Yuksekgonul et al., 2026) continues training the model at test time through RL to find the best solution on a single open-ended problem. These methods trade compute for accuracy on individual instances but do not transfer knowledge across problems.

Inter-episode methods accumulate experience across completed episodes and apply it to new ones. One active direction is automatic prompt optimization. GEPA (Agrawal et al., 2025) and TextGrad (Yuksekgonul et al., 2024) use natural-language feedback from rollouts to iteratively mutate and rewrite prompts. A second direction develops self-referential agents that modify their own code or instructions. ExpeL (Zhao et al., 2024) extracts transferable lessons from successful and failed trajectories. PromptBreeder (Fernando et al., 2024) evolves prompts through mutation and crossover operators. More recent systems such as ADAS (Hu et al., 2025) and Darwin Gödel Machine (Zhang et al., 2025b) extend this by recursively redesigning the self-evolving policy itself (Yin et al., 2024). A third direction builds agentic memory systems: Voyager (Wang et al., 2023) accumulates a reusable skill library from experience in Minecraft, while systems such as MemGen (Zhang et al., 2025a) and Mem0 (Chhikara et al., 2025) maintain evolving memory stores that persist across episodes (Zhang et al., 2025c). All of these methods rely on the inherent reasoning ability of the LLM to analyze feedback and propose improvements. Our work falls in this category but takes a distinct approach: rather than relying on emergent ability, we explicitly train the self-evolving policy through RL.

3 Method

We now introduce our proposed framework and method. We first formalize the test-time inter-episode self-evolution (§3.1). We then describe how we operationalize it through prompt-based updates and tree-guided search (§3.2). Finally, we present Learning to Self-Evolve (LSE), an RL framework that trains the self-evolving policy (§3.3).

3.1 Test-Time Inter-Episode Evolution

Consider a task $\mathcal{T} = (\mathcal{X}, \mathcal{Y}, R)$ comprising an input space $\mathcal{X}$, an output space $\mathcal{Y}$, and a reward function $R : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$. A policy π maps inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$. A *self-evolving policy* is a function f that updates the current policy based on experience collected during interaction. Given

a task $\mathcal{T}$, the system executes T rounds of evolution. At each round t, the current policy $\pi^{(t)}$ is applied to a batch of k problems sampled from $\mathcal{X}$, producing experience tuples $\{(x_i, y_i, r_i)\}_{i=1}^{k}$. The self-evolving policy then computes an updated policy:

$$\pi^{(t+1)} = f\big(\pi^{(t)},\ \{(x_i, y_i, r_i)\}_{i=1}^{k}\big). \tag{1}$$

This produces a sequence of policies $\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(T)}$. The objective of f is to maximize the cumulative reward over T rounds of evolution:

$$\sum_{t=0}^{T} \mathbb{E}_{x\sim\mathcal{X}}\big[R\big(x,\ \pi^{(t)}(x)\big)\big]. \tag{2}$$

In the language model setting, a policy π_θ is determined by its parameters θ and context c, comprising system prompts, instructions, skill libraries, and any other textual input that shapes behavior. This decomposition admits two natural instantiations of f:

- **Gradient-based**: f modifies θ directly (e.g., via RL or SFT on recent experience);
- **Prompt-based**: f modifies c while keeping θ frozen.

We focus on the prompt-based instantiation, where f is itself an LLM that generates updated contexts from past experience. This choice requires no gradient computation at test time, thereby avoiding the catastrophic forgetting problem with continual learning, and casts the evolution problem as a natural-language reasoning task that can itself be improved through training.

3.2 Prompt-Based Evolution with Tree Search

An LLM with frozen parameters θ defines a conditional policy $\pi_\theta(y \mid x, c)$, where $x \in \mathcal{X}$ is a problem instance and c is the context introduced in §3.1. In our implementation, we designate a special *instruction field* within c for the self-evolving policy to edit, leaving all other components (e.g., task description, format specification) fixed.

At each round, the self-evolving policy f_ψ maps the current context and a performance summary to an updated context:

$$c_{t+1} = f_\psi\big(c_t,\ S_t\big), \tag{3}$$

where $S_t = \{(x_i, y_i, y_i^*, r_i)\}_{i=1}^{k}$ is a *structured performance summary* containing the problems, the outputs of the action LLM, ground-truth answers, and per-problem correctness signals from round t.

Note that S_t contains only k problems, where k is typically small, and a different batch is drawn at each round. In-batch performance therefore provides only a noisy estimate of context quality. To obtain a consistent measure across rounds, we fix a separate holdout set $D \subset \mathcal{X}$ and define the reward of context c as

$$\bar{R}(c) \ = \ \frac{1}{|D|}\sum_{x\in D} R(x,\ y), \quad y \sim \pi_\theta(\cdot \mid x, c). \tag{4}$$

Tree-guided evolution. The linear evolution chain $c_0 \to c_1 \to \cdots$ greedily extends the most recent context, which risks committing irreversibly to a suboptimal evolution path. To enable broader exploration of the context space, we maintain an *evolution tree* $\mathcal{G}$ in which each node n stores a tuple $(c_n, S_n, \bar{R}_n, v_n)$ of context, performance summary, mean holdout reward, and visit count. At each round, rather than always extending the latest node, we select the node that maximizes the Upper Confidence Bound (UCB) score (Auer, 2002):

$$n^* = \arg\max_{n\in\mathcal{G}}\ \bar{R}_n + C\sqrt{\frac{\ln N}{v_n}}, \tag{5}$$

where N is the number of completed rounds and $C > 0$ controls the exploration-exploitation tradeoff. The context and summary of n^* are used as input to the next evolution step, and the resulting child node is appended to $\mathcal{G}$. This allows the system to revisit and branch from promising contexts discovered earlier, rather than committing to a single evolution path. The full procedure is summarized in Algorithm 1.

Algorithm 1 Prompt-Based Evolution with Tree Search

Require: Action policy π_θ; self-evolving policy f_ψ; task $\mathcal{T} = (\mathcal{X}, \mathcal{Y}, R)$; holdout set $D \subset \mathcal{X}$; initial context c_0; rounds T; batch size k; exploration constant C

Initialize tree $\mathcal{G} \leftarrow \{(c_0, \emptyset, \bar{R}(c_0), 0)\}$
for $t = 0, 1, \ldots, T-1$ **do**
  Select node $n^* \leftarrow \arg\max_{n \in \mathcal{G}} \ \bar{R}_n + C\sqrt{(\ln N)/v_n}$ ▷ *UCB select*
  Sample problems $\{x_i\}_{i=1}^k \sim \mathcal{X}$
  Generate responses $y_i \sim \pi_\theta(\cdot \mid x_i, c_{n^*})$ for $i = 1, \ldots, k$ ▷ *Act*
  Evaluate $r_i \leftarrow R(x_i, y_i)$ for $i = 1, \ldots, k$ ▷ *Evaluate*
  Construct summary $S_t = \{(x_i, y_i, y_i^*, r_i)\}_{i=1}^k$
  $c_{\text{new}} \leftarrow f_\psi(c_{n^*}, S_t)$ ▷ *Evolve*
  Evaluate $\bar{R}(c_{\text{new}})$ on holdout set D ▷ *Eq. (4)*
  Append child $(c_{\text{new}}, S_t, \bar{R}(c_{\text{new}}), 0)$ to n^* in $\mathcal{G}$; increment v_{n^*}
end for
return $\arg\max_{n \in \mathcal{G}} \ \bar{R}_n$

3.3 Learning to Self-Evolve (LSE)

While off-the-shelf LLMs already exhibit some ability to iteratively refine their own prompts (Yin et al., 2024; Agrawal et al., 2025; Zhang et al., 2025b), this ability emerges entirely from pretraining and standard post-training, and the model is never explicitly optimized for self-improvement. We propose Learning to Self-Evolve (LSE), an RL framework that explicitly trains f_ψ to be an effective self-evolving policy.

Recall from Eq. (2) that the goal of the self-evolving policy is to maximize the cumulative reward over evolution rounds. A natural training objective for f_ψ is:

$$\max_{f_\psi} \sum_{t=0}^{T} \bar{R}(c_t), \quad \text{where } c_{t+1} = f_\psi(c_t, S_t) \ \ \forall\, t. \tag{6}$$

Directly optimizing this T-step objective is costly: each rollout requires T sequential rounds of evaluation and context generation, and the trajectory-level reward introduces a long-horizon credit-assignment problem. We therefore simplify to the *single-step* setting ($T = 1$), where f_ψ produces a single context update $c_1 = f_\psi(c_0, S_0)$ and is rewarded immediately. This reduces the problem to a contextual bandit and avoids the long-horizon credit-assignment difficulty while still capturing the core challenge of learning to improve instructions from feedback.

Even in the single-step setting, the choice of reward function is consequential. A natural candidate is the post-edit reward $\bar{R}(c_1)$, the performance of the action policy under the updated context. However, this reward is biased toward contexts that are already effective. Consider two scenarios: (1) the initial context achieves 80% accuracy and drops to 70% after editing, yielding $r = 0.7$; (2) the initial context achieves 30% accuracy and improves to 60%, yielding only $r = 0.6$. The post-edit reward ranks the first scenario higher despite the *degradation* in performance, because it conflates the quality of the starting point with that of the edit. This bias encourages the policy to preserve already-effective contexts rather than genuinely learn to improve them. We instead define the reward as the *improvement in reward*:

$$r_{\text{LSE}} \ = \ \bar{R}(c_1) - \bar{R}(c_0), \tag{7}$$

which directly incentivizes f_ψ to produce edits that improve performance relative to the starting point, regardless of the initial performance level.

Notably, if r_{LSE} is used as the reward in a standard policy-gradient algorithm such as PPO or GRPO, the baseline estimator absorbs the $\bar{R}(c_0)$ term. To see this, let $s = (c_0, S_0)$ denote the state observed before the edit. A baseline $V(s)$ estimated under r_{LSE} satisfies $V'(s) = \mathbb{E}[\bar{R}(c_1) - \bar{R}(c_0) \mid s] = V(s) - \bar{R}(c_0)$, where $V(s) = \mathbb{E}[\bar{R}(c_1) \mid s]$ is the baseline under the post-edit reward. The advantage then reduces to:

$$A'(s, c_1) \ = \ r_{\text{LSE}} - V'(s) \ = \ \big(\bar{R}(c_1) - \bar{R}(c_0)\big) - \big(V(s) - \bar{R}(c_0)\big) \ = \ \bar{R}(c_1) - V(s), \tag{8}$$

which is identical to the advantage under the post-edit reward alone. That is, the delta-reward and the post-edit reward yield the same gradient estimates whenever a learned baseline is used. Rather than

Table 1: Text-to-SQL results on BIRD. All methods use Qwen3-4B-Instruct as the action policy π_θ. We report execution accuracy (%). Best result per column in **bold**.

Method	Financial	Toxicology	Codebase	Formula 1	Card Games	Avg.
Seed prompt	51.0	60.3	63.7	54.5	56.5	57.2
Qwen3-4B-Instruct	63.7	60.3	70.2	56.0	61.0	62.2
Claude Sonnet 4.5	70.8	63.8	67.8	57.3	63.0	64.5
GPT-5	70.8	65.8	**72.0**	54.3	63.3	65.2
GEPA	64.0	62.0	**72.0**	54.0	62.0	62.8
TextGrad	60.3	66.0	71.5	56.5	61.3	63.1
LSE (ours)	**72.0**	**68.5**	**72.0**	**59.8**	**64.0**	**67.3**

using a value model or group-based normalization as the baseline, we can bypass baseline estimation entirely: the pre-edit reward $\bar{R}(c_0)$ is known before f_ψ acts and equals the reward of a null edit that returns c_0 unchanged, so it can serve directly as the baseline. This yields the LSE advantage:

$$A_{\mathrm{LSE}} \;=\; \bar{R}(c_1) - \bar{R}(c_0), \tag{9}$$

and the corresponding policy-gradient estimate:

$$\nabla_\psi J \;=\; \mathbb{E}_{c_1 \sim f_\psi(\cdot \mid c_0, S_0)} \Big[A_{\mathrm{LSE}} \, \nabla_\psi \log f_\psi(c_1 \mid c_0, S_0) \Big]. \tag{10}$$

Because $\bar{R}(c_0)$ is action-independent, using it as a baseline does not alter the expected gradient. It is, however, a control variate that cancels prompt-specific offsets. In practice, evaluation noise and between-prompt difficulty variation likely dominate raw accuracy scores. Under these conditions, the improvement-based advantage provides a cleaner learning signal and more stable policy-gradient updates. It also reduces training cost, as it requires neither multiple rollouts per prompt for group-based normalization nor a separate value network.

The gradient in Eq. (10) depends on the distribution of starting states $s = (c_0, S_0)$. If c_0 is always the seed context, a mismatch arises: at test time, the policy runs for multiple rounds (Algorithm 1) and must improve contexts produced by its own prior edits. We therefore populate the tree $\mathcal{G}$ with multiple rounds of evolution to construct the training dataset, then randomly sample nodes from $\mathcal{G}$ as starting contexts at every RL step. This exposes f_ψ to a distribution of contexts similar to what it will see during multi-step evolution.

4 Experiments

We evaluate LSE on two task domains, Text-to-SQL generation and general question answering, comparing against both stronger models and alternative prompt optimization methods (§4.1–4.2). We then ablate the reward design and search strategy in §4.3.

4.1 Experimental Setup

Models. We use Qwen3-4B-Instruct as both the action policy π_θ and the self-evolving policy f_ψ, unless otherwise specified. Training details and hyperparameters can be found in Appendix A.

Tasks and datasets. We evaluate on tasks across two domains: Text-to-SQL generation, where the policy produces executable SQL queries that retrieve the data specified by a user question, and general question answering (QA), where the policy answers multiple-choice questions across diverse academic subjects. The prompts for each task are in Appendix B.

For Text-to-SQL, we use BIRD (Li et al., 2024), which pairs natural-language questions with SQL queries across database domains. Each database is a separate task domain: problems are sampled from the same domain for both evolution rounds and holdout evaluation. We train on the BIRD training split and evaluate on five randomly selected databases from the BIRD-SQL Mini-Dev split.

For general QA, we use SuperGPQA (Team et al., 2025) and MMLU-Redux (Gema et al., 2024). We convert SuperGPQA questions to four-way multiple-choice format to match MMLU-Redux,

Table 2: Question-answering results on MMLU-Redux. All methods use Qwen3-4B-Instruct as the action policy π_θ. We report accuracy (%). Best result per column in **bold**.

Method	Bus. Eth.	Phil.	Prof. Acc.	Econ.	Anat.	Security	Virol.	Moral Sc.	Glob. Facts	Prof. Law	Avg.
Seed prompt	70.5	76.0	67.7	76.2	76.0	68.2	72.5	59.7	55.5	54.0	67.6
Qwen3-4B-Instruct	75.5	79.2	73.0	83.0	82.3	68.8	77.0	59.7	56.0	57.3	71.2
Claude Sonnet 4.5	75.3	79.0	75.5	81.0	79.8	69.3	77.5	**67.3**	**57.0**	58.5	72.0
GPT-5	**77.5**	80.0	73.0	83.5	81.0	70.0	**82.0**	64.8	55.5	58.0	72.5
GEPA	76.0	80.0	**78.0**	84.0	82.0	70.0	78.0	62.0	56.0	**64.0**	73.0
TextGrad	74.0	79.0	67.8	73.3	80.0	67.8	77.3	62.5	51.3	58.3	69.1
LSE (ours)	75.0	**82.0**	73.0	**85.0**	**84.5**	**70.5**	79.0	64.5	**57.0**	62.0	**73.3**

and treat each subject as a separate task domain. As with Text-to-SQL, each evolution run operates within a single subject domain. We train on SuperGPQA and evaluate on ten subjects from MMLU-Redux.

Baselines. We first evaluate stronger models as the self-evolving policy f_ψ while keeping Qwen3-4B-Instruct as the action policy π_θ. We consider two frontier closed-source models, GPT-5 (OpenAI, 2025) and Claude Sonnet 4.5 (Anthropic, 2025)

We also compare with two alternative designs of the self-evolving policy. For both methods, we use Qwen3-4B-Instruct as the prompt proposer and optimizer.

- **GEPA** (Agrawal et al., 2025) is a reflective prompt optimizer that merges textual reflection with multi-objective evolutionary search. GEPA mutates prompts based on natural-language feedback from new rollouts and maintains a Pareto front over per-instance performance to avoid greedy local optima. Each GEPA optimization step corresponds to one evolution round: the sampled problem batch is the training data for reflection, and the holdout set D is D_{pareto} in GEPA.
- **TextGrad** (Yuksekgonul et al., 2024) decomposes each prompt update into two LLM calls: a *backward* call that critiques the current instruction given the batch failures and produces natural-language "gradients" (feedback on how the instruction should change), followed by a *Textual Gradient Descent (TGD)* call that rewrites the instruction by incorporating the feedback. We follow the example provided in the official repository[1] and treat each backward–TGD step as one evolution round.

Evaluation protocol. For all methods, we sample problems and present them to the action policy in a fixed order across runs. The holdout set D is also fixed across all evaluation runs within each task domain. We report the best performance achieved over T rounds of evolution. Additional details can be found in Appendix A.

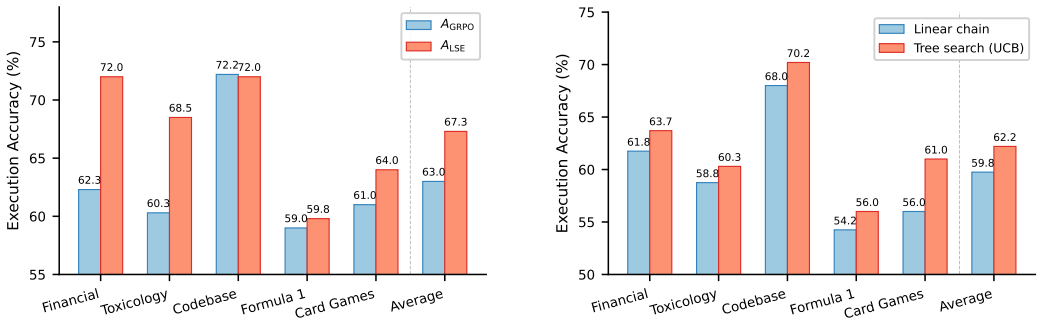
4.2 Main Results

Tables 1 and 2 report results on Text-to-SQL and QA. Even without explicit training for self-improvement, off-the-shelf LLMs can refine their own prompts when given test-time feedback. The untrained Qwen3-4B-Instruct baseline improves over the seed prompt by 5% on BIRD and 3.6% on MMLU-Redux. This confirms that LLMs can already learn from their own experience within the evolution loop of Algorithm 1.

RL training with LSE substantially improves this ability. Despite using only a 4B-parameter model, LSE outperforms both frontier models on BIRD, surpassing GPT-5 by 2.1% (67.3% vs. 65.2%) and Claude Sonnet 4.5 by 2.8%. On MMLU-Redux, LSE matches GPT-5 (73.3% vs. 72.5%) and outperforms Claude Sonnet 4.5. These results indicate that explicit RL training for self-evolution is effective, enabling a small model to match or surpass frontier models.

LSE also outperforms both prompt optimization methods. On BIRD, LSE surpasses GEPA by 4.5% (67.3% vs. 62.8%) and TextGrad by 4.2% (67.3% vs. 63.1%). On MMLU-Redux, LSE matches GEPA (73.3% vs. 73.0%) and outperforms TextGrad by 4.2%. Together, these results show

[1] https://github.com/zou-group/textgrad/blob/main/evaluation/prompt_optimization.py



(a) Effect of reward design

(b) Effect of search strategy

Figure 2: Ablation studies on reward design and search strategy. (a) A_{GRPO} uses $\bar{R}(c_1)$ with GRPO's group-based advantage; A_{LSE} uses the improvement-based reward $r_{\text{LSE}} = \bar{R}(c_1) - \bar{R}(c_0)$ (Eq. 7). (b) Tree search (UCB) vs. linear chain (always extends the most recent node), both with the untrained Qwen3-4B-Instruct as f_ψ.

that while off-the-shelf LLMs have some prompt-refinement ability, explicit training to self-evolve matches or outperforms untrained baselines, larger models, and specialized optimization methods.

Finally, both the improvement from self-evolution over the seed prompt and the additional benefit of LSE are smaller on MMLU-Redux than on BIRD. One possible explanation is the structure of the two tasks. In BIRD, all queries within a domain target the same database, so there is clear shared knowledge across problems: understanding the schema, common join patterns, or column semantics from one query directly helps with others. In MMLU-Redux, problems within the same subject are deliberately deduplicated and designed to cover broad topics. Solving one econometrics question does not guarantee useful knowledge for the next. This limits how much any self-evolving policy can improve the action policy's context from experience within a single domain.

4.3 Analysis

Effect of reward design. In §3.3 we motivated $A_{\text{LSE}} = \bar{R}(c_1) - \bar{R}(c_0)$ as a cleaner learning signal than the standard GRPO advantage A_{GRPO}, which reduces to optimizing post-edit accuracy. We train a variant with A_{GRPO}, keeping all other settings identical. On BIRD, A_{LSE} outperforms A_{GRPO} by 4.3% (67.3% vs. 63.0%; Figure 2a). These results provide empirical evidence that the improvement-based objective is more effective for training self-evolving policies.

Effect of search strategy. We compare UCB tree search against a linear-chain baseline that always extends the most recent node. Both use the untrained Qwen3-4B-Instruct as f_ψ. Figure 2b shows that tree search improves the average by 2.4% on BIRD (62.2% vs. 59.8%) and 2.2% on MMLU-Redux (71.2% vs. 69.0%; Figure 4).

The key advantage is that tree search does not commit to a bad edit irrevocably. Figure 3 shows a concrete example on the BIRD Card Games split. The linear chain's average accuracy collapses from 56% to below 30% after a sequence of bad edits, and never recovers because each round builds on the previous context. With tree search, a bad edit at an early round does not cascade: UCB selection shifts back to a higher-scoring ancestor, keeping the trajectory out of bad local optima.

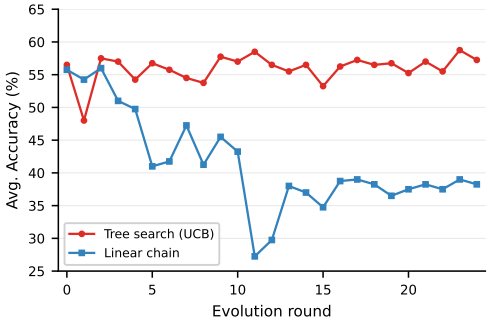


Figure 3: Per-round average accuracy on the BIRD Card Games database. The linear chain cannot recover from bad edits, while tree search (UCB) backtracks to higher-scoring ancestors.

Test-time self-evolution for specialized models. Current LLM development often involves training specialized models for a domain of tasks. Can

Table 3: Test-time self-evolution of a specialized action model on BIRD. Arctic-Text2SQL-R1-7B (Yao et al., 2025), an RL-tuned text-to-SQL model, serves as the action policy π_θ. The self-evolving policy f_ψ is the LSE-trained Qwen3-4B-Instruct from the main experiments, applied without further training. We report execution accuracy (%).

Variant	Financial	Toxicology	Codebase	Formula 1	Card Games	Avg.
Seed prompt	56.8	54.5	65.3	52.3	59.5	57.7
+ LSE evolution	68.3	62.3	71.5	57.0	63.0	64.4

test-time self-evolution further improve such models? We test this by replacing π_θ with Arctic-Text2SQL-R1-7B (Yao et al., 2025), a text-to-SQL model fine-tuned with RL on the BIRD training set, and applying the same LSE-trained f_ψ (Qwen3-4B-Instruct) without additional training.

Table 3 shows that LSE evolution improves Arctic by 6.7% on average (57.7% $\rightarrow$ 64.4%). This indicates that parameter-level and prompt-level optimization are complementary: RL training encodes general SQL patterns into model weights, while prompt evolution adapts the context to each database at test time. The result also demonstrates that the LSE-trained policy transfers across action models: although f_ψ was trained exclusively with Qwen3-4B-Instruct, the evolution strategy generalizes to guide a different model.

5 Conclusion

This work demonstrates that test-time self-evolution is a learnable skill that can be directly optimized through fine-tuning. The central design choice in LSE is a single-step RL objective that rewards the improvement each edit produces, sidestepping multi-step trajectory optimization while still capturing the core challenge of learning from feedback. Tree-guided search then composes these edits into multi-round evolution at test time. Our results show that direct optimization for self-evolution is effective, enabling a 4B-parameter model to match or surpass frontier models and prompt optimizers. Taken together, these findings highlight the benefit of targeting self-evolution as a distinct skill and designing learning algorithms for it.

Limitations. Our work has several limitations. First, we reduce the multi-step evolution problem to a single-step training objective, delegating exploration entirely to the tree search algorithm at test time. Jointly optimizing over multi-step trajectories could yield stronger policies but would introduce additional challenges in credit assignment and computational cost. Second, we train a separate self-evolving policy for each task domain. Training a single policy that generalizes across diverse tasks is a natural extension, though it likely requires large-scale training across many domains. Third, we constrain evolution to the instruction field of the context; other components such as tools, skill libraries, and external memory are not explored. More broadly, the LSE framework could be paired with updates in the latent space or parameter space (Sun et al., 2020; Tandon et al., 2025). Finally, our training and evaluation environments are relatively small in scale. Curating effective environments for test-time self-evolution is difficult, as it requires not only sufficient problems with feedback but also problems that share enough structure for evolution to be meaningful. Developing more principled and scalable approaches to environment curation and evaluation remains an important open problem.

References

Lakshya A. Agrawal, Shuo Tan, Deniz Soylu, Noah Ziems, Rishabh Khare, Keg Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Manli Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Anthropic. System card: Claude opus 4.5, 2025. URL `https://www-cdn.anthropic.com/bf10f64990cfda0ba858290be7b8cc6317685f47.pdf`.

Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of machine learning research*, 3(Nov):397–422, 2002.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In Inês Lynce, Nello Murano, Mauro Vallati, Serena Villata, Federico Chesani, Michela Milano, Andrea Omicini, and Mehdi Dastani (eds.), *ECAI 2025 - 28th European Conference on Artificial Intelligence, 25-30 October 2025, Bologna, Italy - Including 14th Conference on Prestigious Applications of Intelligent Systems (PAIS 2025)*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pp. 2993–3000. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://doi.org/10.3233/FAIA251160.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, 2024.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile van Krieken, and Pasquale Minervini. Are we done with mmlu?, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Mober, et al. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, 2024.

Aviral Kumar, Vincent Du, Ankit Singh Rawat, and Rishabh Agarwal. Training language models to self-correct via reinforcement learning. In *International Conference on Learning Representations*, 2025.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023.

OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/. Accessed: 2025-03-21.

OpenAI. Openai gpt-5 system card. *arXiv preprint arXiv: 2601.03267*, 2025.

Jean Piaget. *The Origins of Intelligence in Children*. W. W. Norton & Company, 1952. doi: 10.1037/11494-000. Trans. M. Cook.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

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

Robert J. Sternberg. A theory of adaptive intelligence and its relation to general intelligence. *Journal of Intelligence*, 7(4), 2019. ISSN 2079-3200. doi: 10.3390/jintelligence7040023. URL https://www.mdpi.com/2079-3200/7/4/23.

Yu Sun, Xiaolong Wang, Liu Zhuang, John Miller, Moritz Hardt, and Alexei A. Efros. Test-time training with self-supervision for generalization under distribution shifts. In *ICML*, 2020.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. End-to-end test-time training for long context. *arXiv preprint arXiv: 2512.23675*, 2025.

M-A-P Team, Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, Kang Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, Chujie Zheng, Kaixin Deng, Shian Jia, Sichao Jiang, Yiyan Liao, Rui Li, Qinrui Li, Sirun Li, Yizhi Li, Yunwen Li, Dehua Ma, Yuansheng Ni, Haoran Que, Qiyao Wang, Zhoufutu Wen, Siwei Wu, Tianshun Xing, Ming Xu, Zhenzhu Yang, Zekun Moore Wang, Junting Zhou, Yuelin Bai, Xingyuan Bu, Chenglin Cai, Liang Chen, Yifan Chen, Chengtuo Cheng, Tianhao Cheng, Keyi Ding, Siming Huang, Yun Huang, Yaoru Li, Yizhe Li, Zhaoqun Li, Tianhao Liang, Chengdong Lin, Hongquan Lin, Yinghao Ma, Tianyang Pang, Zhongyuan Peng, Zifan Peng, Qige Qi, Shi Qiu, Xingwei Qu, Shanghaoran Quan, Yizhou Tan, Zili Wang, Chenqing Wang, Hao Wang, Yiya Wang, Yubo Wang, Jiajun Xu, Kexin Yang, Ruibin Yuan, Yuanhao Yue, Tianyang Zhan, Chun Zhang, Jinyang Zhang, Xiyue Zhang, Xingjian Zhang, Yue Zhang, Yongchi Zhao, Xiangyu Zheng, Chenghua Zhong, Yang Gao, Zhoujun Li, Dayiheng Liu, Qian Liu, Tianyu Liu, Shiwen Ni, Junran Peng, Yujia Qin, Wenbo Su, Guoyin Wang, Shi Wang, Jian Yang, Min Yang, Meng Cao, Xiang Yue, Zhaoxiang Zhang, Wangchunshu Zhou, Jiaheng Liu, Qunshu Lin, Wenhao Huang, and Ge Zhang. Supergpqa: Scaling llm evaluation across 285 graduate disciplines, 2025. URL `https://arxiv.org/abs/2502.14739`.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Gaurav Nuti, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *arXiv preprint arXiv:2505.20315*, 2025.

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

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *arXiv preprint arXiv: 2406.07496*, 2024.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv: 2601.16175*, 2026.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Guibin Zhang, Muxin Fu, and Shuicheng Yan. Memgen: Weaving generative latent memory for self-evolving agents. *arXiv preprint arXiv: 2509.24704*, 2025a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv: 2510.04618*, 2025c.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 2024.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025a.

Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. Learning to reason without external rewards. *arXiv preprint arXiv: 2505.19590*, 2025b.

Yuxin Zuo, Junyou Zhang, Di Yang, Guojun Chen, Shuai Li, Hao Dong, Mingyu Wang, and Zongqing Xu. TTRL: Test-time reinforcement learning. *arXiv preprint arXiv:2504.16084*, 2025.

A Training and Evaluation Details

Data generation. We train a separate self-evolving policy for each task domain (Text-to-SQL and QA) using evolutions trajectories generated from the corresponding training set. For each domain, we run 200 data-generation runs, each containing 20 rounds of evolution, yielding approximately 4,000 tree nodes to sample from during RL training.

RL training. We implement our RL framework using verl (Sheng et al., 2024). We find that randomly sampling nodes from the evolution trees produces weak training signal early in training. Instead, we build a simple curriculum by preferentially sampling nodes with the highest improvement potential, defined as the difference between a node's performance and the maximum performance in its own tree. We use a learning rate of 1×10^{-5}, sample 32 nodes per batch, and generate 4 rollouts per node. We perform on-policy training and do not apply KL regularization. We train for 4 epochs and select the best checkpoint based on a separate development set.

Evaluation protocol. For every domain, we fix the holdout set D at 50 problems. Performance on the holdout set is calculated as the average over eight generations. We run 25 rounds of evolution and report the best holdout performance achieved by each self-evolving method over the course of evolution. At each round, a batch of 10 problems is sampled with replacement and presented to the action model. The random seed is fixed so that all methods observe the same sequence of problem batches.

Dataset sizes. Table 4 reports the number of evaluation problems per domain.

Table 4: Number of evaluation problems per domain.

Task	Domain	# Problems
BIRD	Financial	106
	Toxicology	145
	Card Games	191
	Formula 1	174
	Codebase	186
MMLU-Redux	Business Ethics	100
	Philosophy	100
	Professional Accounting	95
	Econometrics	100
	Anatomy	100
	Security Studies	100
	Virology	96
	Moral Scenarios	100
	Global Facts	96
	Professional Law	100

B Prompts

Each task uses three prompt templates: (1) a *system prompt* that provides the action model π_θ with task context and the current instructions, (2) a *user message* that presents each problem instance, and

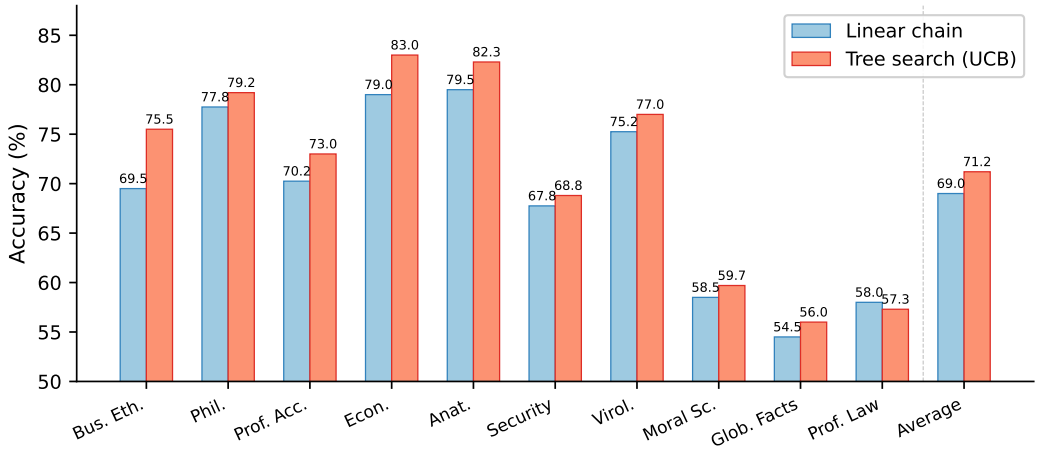


Figure 4: Search strategy ablation on MMLU-Redux, complementing Figure 2b. Both variants use the untrained Qwen3-4B-Instruct as the self-evolving policy f_ψ. Tree search improves the average accuracy from 69.0% to 71.2%.

(3) a *self-evolution prompt* that the self-evolving policy f_ψ receives to produce a revised instruction. The instruction field within the system prompt is the component that f_ψ edits at each evolution round. Below we reproduce the templates for both tasks.

B.1 TEXT-TO-SQL

Action model system prompt.

```
Task Overview:
You are a data science expert. Below, you are provided with a
database schema and a natural language question. Your task is to
understand the schema and generate a valid SQL query to answer the
question.

Database Engine:
SQLite

Database Schema:
{schema}
This schema describes the database's structure, including tables,
columns, primary keys, foreign keys, and any relevant relationships
or constraints.

**Instructions**
{instructions}
```

The seed instruction is: `Return only a single valid SQLite SQL statement in <answer>...</answer>.`

Action model user message.

```
Question:
{question}

**Instructions**
{instructions}

Follow the instructions and show your work. When you are ready,
```

```
return the query output list in tags: <answer> ... </answer>
```

Self-evolving policy prompt.

```
You are an expert at designing text-to-SQL agents. The agent is
running on a fixed database schema. Below is the current agent
prompt and a summary of recent performance. Rewrite ONLY the
instructions to improve execution accuracy while maintaining
strict output format.

Current prompt:
{old_prompt}

Evaluation summary over {n_problems} problems and the agent's
full thinking process:
{summary}

**How to write Instructions**
- The agent will continue receive different user queries so don't
  make the instructions too specific to a single question.
  Referring to the questions in the current summary with only the
  question number is not helpful.
- Keep it concise and practical.
- You may include rules, heuristics, knowledge about the database,
  low-level instructions/examples, high-level ideas/strategies,
  pitfalls and any information that you think can make the agent
  better.
- Organize however you like (bullets, headings, checklists).
- Be creative and think about the agent's behavior across
  iterations. Don't be confined by what I told you.
- Don't change the output format, the agent should still return
  the final SQL query in tags: <answer> ... </answer>.

Think step by step and show your work. Reason about the history
of the model's behavior across iterations.

When you are ready, put your revised Instructions within
<prompt>[your new instructions]</prompt> tags.
```

B.2 Question Answering

Action model system prompt.

```
Task Overview:
You are an expert taking a test. Below, you are provided with a
question and a list of choices. Your task is to select the correct
answer from the choices.

**Instructions**
{instructions}
```

The seed instruction is: `Return only the letter of the correct choice (A, B, C, or D) in <answer>...</answer>.`

Action model user message.

```
Question:
```

```
{question}

Choices:
{choices}

Follow the instructions and show your work. When you are ready,
return the answer letter in tags: <answer> ... </answer>
```

Self-evolving policy prompt.

```
You are an expert at designing agents for solving multiple-choice
questions that involve both factual knowledge and reasoning.
Below is the current agent prompt and a summary of recent
performance on a set of problems. Rewrite ONLY the instructions
to improve accuracy while maintaining strict output format.

Current prompt:
{old_prompt}

Evaluation summary over {n_problems} problems and the agent's
full thinking process:
{summary}

**How to write Instructions**
- The agent will continue to receive different questions from the
  same subjects. Don't make the instructions too specific to a
  single question.
- Keep it concise and practical.
- You may include rules, heuristics, strategies for multiple
  choice questions (e.g., elimination, careful reading), knowledge
  about the subjects (e.g., common misconceptions, important
  facts, etc.), and any information that you think can make the
  agent better.
- Organize however you like (bullets, headings, checklists).
- Be creative and think about the agent's behavior across
  iterations. Don't be confined by what I told you.
- Don't change the output format, the agent should still return
  the final answer letter in tags: <answer> ... </answer>.

Think step by step and show your work. Reason about the history
of the model's behavior across iterations.

When you are ready, put your revised Instructions within
<prompt>[your new instructions]</prompt> tags.
```

B.3 Example Instructions Found by LSE

Below we reproduce the best instructions discovered by the LSE-trained self-evolving policy on one domain from each task. These instructions replace the seed instruction in the action model prompt after multiple rounds of tree-guided evolution.

Text-to-SQL: BIRD Codebase database.

```
1. Always return exactly one valid SQLite query in the format:
   <answer>YOUR_SQL_QUERY_HERE</answer>

2. Carefully analyze the natural language question to identify:
   - The target attribute (e.g., name, count, date, status)
   - The relevant tables and their columns
```

```
   - Any joins required to connect tables via foreign keys
   - Any filters (e.g., equality, date ranges, null checks)

3. Use only the provided schema. Do not assume columns or
   tables that are not defined.

4. When joining tables:
   - Match foreign key references precisely
     (e.g., posts.OwnerUserId -> users.Id)
   - Use explicit column aliases only if needed for clarity
   - Ensure join conditions match the schema

5. For date filtering:
   - Use strftime('%Y', column) to extract year
   - Never use YEAR() -- SQLite does not support it
   - Match date format exactly (e.g., '2014-04-23 20:29:39.0')

6. For conditional outputs (e.g., "well-finished"):
   - Use CASE WHEN or IIF to map NULL / non-NULL values
   - Match the definition in the question
     (e.g., "not well-finished" = ClosedDate IS NULL)

7. Common pitfalls to avoid:
   - Misidentifying OwnerUserId vs. LastEditorUserId
   - Incorrectly joining on UserId instead of Id
   - Misspelling column names (e.g., CreaionDate)
   - Forgetting required joins for user attributes
   - Confusing UserDisplayName in comments with post ownership

8. Always use subqueries for exact values (MIN, MAX):
   - e.g., WHERE Age = (SELECT MIN(Age) FROM users)
     instead of ORDER BY Age LIMIT 1

9. For percentages or ratios, compute numerator and
   denominator separately using subqueries. Use
   CAST(... AS REAL) for floating-point division.

10. Avoid redundant joins -- if a query can be answered
    from a single table, do not introduce unnecessary joins.
```

Question answering: MMLU-Redux Anatomy subject.

```
- Return only the letter of the correct choice (A, B, C,
  or D) in <answer>...</answer>.
- Carefully analyze the question and all answer options
  before selecting.
- Use elimination to rule out clearly incorrect choices
  based on factual knowledge or logical inconsistency.
- For biological and anatomical questions, recall key
  structures and their functions (e.g., fertilization
  occurs in the fallopian tube, not the ovary or uterus;
  the pituitary is the "master gland").
- In neurology: upper motor neuron lesions cause spastic
  paralysis; lower motor neuron lesions cause flaccid
  paralysis; sympathetic pathways use noradrenaline.
- In embryology, palatine shelf elevation is due to turgor
  pressure from hydrophilic molecules, not directly from
  tongue descent or brain flexure.
- For spinal cord injuries, breathing is controlled by the
  brainstem (medulla), not the cervical spinal cord.
- Fracture types: closed = skin intact; greenstick = bent
```

```
  but not displaced; open = skin broken; spiral = twisting.
- In autonomic responses, sympathetic chain damage leads to
  pupillary constriction and vasodilation (loss of
  vasoconstriction), not increased sweating.
- In Horner's syndrome: miosis, facial vasodilation,
  decreased lacrimation, and anhydrosis.
- In cerebrovascular accidents, internal capsule lesions
  cause contralateral spastic paralysis.
- Prioritize accuracy over common assumptions -- especially
  regarding directionality (contralateral vs ipsilateral),
  timing (diastole vs systole), and structural relationships.
- Be vigilant for common misconceptions used as distractors.
- Do not over-rely on common associations; base decisions
  on precise anatomical, physiological, or pathological facts.
```