COEVO: CO-EVOLVING CONTEXT AND PARAMETERS FOR RECURSIVE SELF-IMPROVEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement (RSI) seeks to move large language models beyond static training pipelines toward systems that can participate in improving their own future behavior. Existing approaches largely follow two directions: updating model parameters through online learning, or improving the external context through search, reflection, and prompt optimization. Although both mechanisms can support continued improvement, they are typically studied independently. This separation overlooks an important interaction: the context shapes the experience from which a model learns, while an evolving model may interpret and utilize the same context differently over time. We therefore formulate RSI as a problem of parameter–context co-evolution, where model parameters and the learning context adapt within a shared feedback loop. We introduce COEVO, a framework that updates model parameters from on-policy experience while adapting contextual guidance according to the state of the evolving policy. Policy entropy and prompt-conditioned attention are used as complementary signals to guide this adaptation. Experiments show that COEVO consistently improves task performance over fixed-context reinforcement learning and produces policies that are more robust to changes in system prompts. More broadly, our results suggest that external context should be viewed not merely as a fixed interface to a large language model, but as an adaptive component of recursive self-improvement.

1 INTRODUCTION

Large language models (LLMs) have progressed from systems whose capabilities are primarily determined by large-scale pretraining to increasingly adaptive systems whose behavior can be further improved through post-training and interaction with external feedback. In particular, on-policy reinforcement learning allows a model to generate its own trajectories, receive task-level feedback, and update its parameters from the resulting experience (Shao et al., 2024; Yu et al., 2025; Feng et al., 2025). This transition points toward a broader objective beyond conventional model training: recursive self-improvement (RSI), in which the current model actively participates in constructing the process that improves its future versions. Rather than treating learning as a sequence of externally specified optimization steps, RSI seeks to close the feedback loop between the state of a model, the experience it generates, and the mechanisms used to produce subsequent improvements.

Existing approaches to RSI in LLMs can be understood through two complementary forms. The first operates on the model itself, using online learning or reinforcement learning to continually update model parameters. Parameter adaptation can internalize new reasoning strategies, but the learning procedure typically assumes that the training context remain fixed throughout optimization (Shinn et al., 2023; Yang et al., 2023; Guo et al., 2023). The second operates on the model's external context: methods based on reflection, prompt search, evolutionary optimization, or linguistic memory iteratively modify intermediate scaffolds without directly changing model parameters (Fernando et al., 2023; Zelikman et al., 2023; Agrawal et al., 2025; Zhang et al., 2025). Such approaches can rapidly alter how existing capabilities are elicited, but they generally optimize the context for a model whose underlying policy is treated as fixed during optimization.

However, parameter learning and context optimization are intrinsically coupled: the context shapes the trajectories used for parameter updates, while parameter updates change how that context is interpreted and utilized. Context effectiveness is therefore policy dependent, making RSI naturally a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

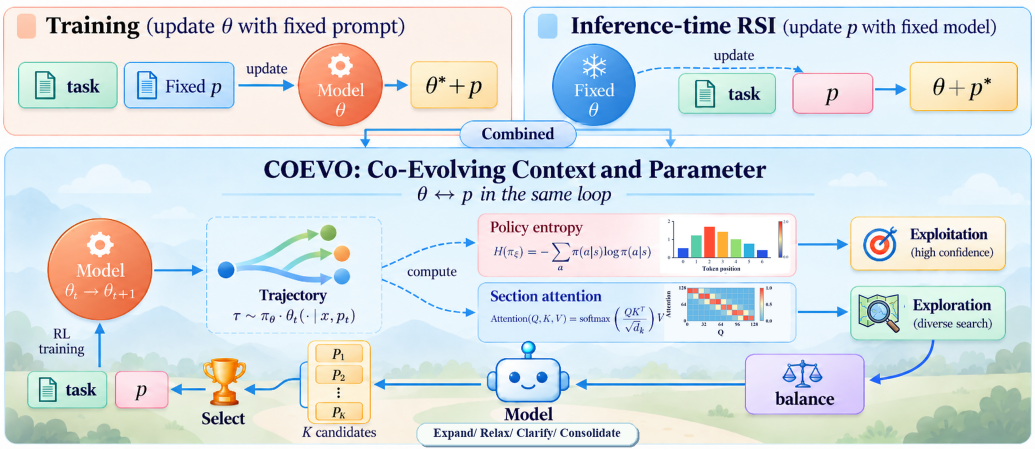


Figure 1: The context–parameter feedback loop in COEVO.

co-evolution problem. Although ESPL (Zhang et al., 2026) jointly evolves prompts and model parameters, its prompt evolution remains primarily driven by downstream reward, effectively treating context adaptation as a black-box outer-loop search. Such a signal can rank candidate prompts, but provides little information about why a context is ineffective or how it should change as the policy evolves. We instead make context evolution explicitly policy-aware, using the policy's exploration state and context utilization to guide targeted adaptation.

We propose COEVO, a reinforcement-learning-based framework for parameter–context co-evolution that closes the feedback loop between policy learning and context adaptation. During training, the current policy generates on-policy experience to update model parameters, while its evolving state guides subsequent context refinement. Specifically, we characterize the evolving policy using policy entropy, which reflects the model's exploration state, and prompt-conditioned attention, which measures its utilization of contextual guidance. These signals enable the context to adapt to the changing policy rather than being optimized solely on reward. The updated context then shapes future on-policy experience, forming a recursive interaction between parameter learning and context evolution. Figure 1 provides a conceptual illustration of COEVO. In our evaluated settings, COEVO achieves stronger task performance than fixed-prompt RL baselines. Ablation studies also show that the policy-aware signals improve over a reward-only prompt-evolution variant inspired by ESPL, indicating that entropy and attention signals benefit RL training and provide effective guidance for context evolution.

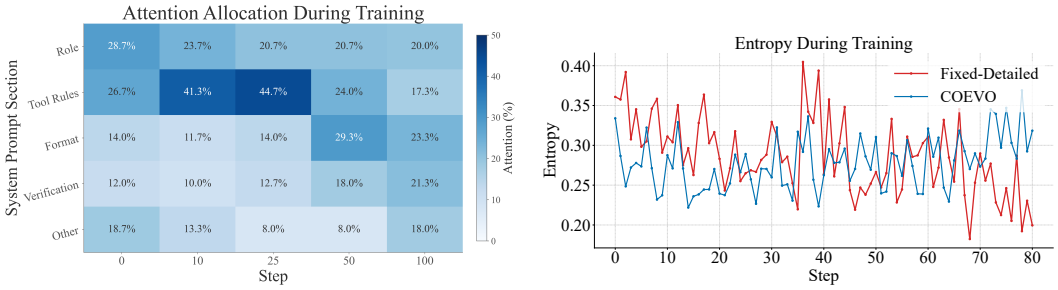
Overall, this paper makes the following contributions:

- We formulate LLM recursive self-improvement as a general and unified parameter–context co-evolution problem, where model updates change how contextual guidance is interpreted, while context updates reshape the on-policy experience used for subsequent learning.
- We introduce COEVO, a bounded RSI framework that jointly evolves model parameters and system context within a unified reinforcement-learning loop. Using policy entropy and prompt-conditioned attention to characterize the current policy state, COEVO performs targeted context adaptation and improves over standard RL algorithms.
- We show that exploration and context utilization provide complementary signals for context evolution: entropy reveals changes in behavioral diversity, while attention identifies how the strategy scaffold is being used. Ablations demonstrate that removing either signal leads to distinct failure modes, highlighting the importance of policy-aware context adaptation.

2 Problem Formulation and Design Hypotheses

2.1 Context Affects the Learning Distribution

Training context determines which experiences are available for learning. Given policy parameters θ_t and context p_t, an on-policy trajectory τ_t is generated by rolling out the policy $\pi_{\theta_t}(\cdot \mid x, p_t)$ for



(a) Prompt-section attention allocation **(b) Policy entropy**

Figure 2: **Policy-state signals evolve throughout training. (a)** Relative attention allocation across functional prompt sections changes substantially over training, indicating that the policy's utilization of the same context is non-stationary. **(b)** Policy entropy captures the concurrent evolution of behavioral exploration: under a fixed detailed context, entropy progressively decreases, whereas COEVO maintains broader exploration through context adaptation. Together, these dynamics motivate using both context utilization and policy exploration to guide context evolution.

task x under context p_t, and this trajectory then contributes to the policy-gradient estimate. The dependence is thus given by:

$$p_t \longrightarrow \tau_t \sim \pi_{\theta_t}(\cdot \mid x, p_t) \longrightarrow \widehat{\nabla_\theta J}(\theta_t, p_t) \longrightarrow \theta_{t+1}.$$

Even with a fixed task reward and optimizer, revising p_t changes the trajectory distribution entering the next update. This distinguishes training-context adaptation from choosing an inference wrapper after learning has finished.

2.2 Context Usefulness May Be Policy-Dependent

The same scaffold can induce different behavior under different parameters. Guidance that helps an early policy organize a solution may become redundant or restrictive afterward, while an instruction ignored by one policy may be actively used by another. We therefore hypothesize that a scaffold's usefulness depends on the current policy state, not on the prompt alone.

Figure 2 provides an empirical illustration of this non-stationarity. The attention heatmap (left) shows that attention is redistributed across functional prompt sections as training progresses, indicating that the policy's utilization of the same contextual scaffold changes over time. Meanwhile, the entropy trajectories (right) reveal concurrent changes in behavioral exploration: under a fixed detailed context, the policy becomes progressively more concentrated, whereas adaptive context evolution maintains broader exploration. Importantly, these observations do not imply a universal ordering of short, detailed, or strongly attended prompts. Rather, they suggest that context usefulness is relational: it depends on both how the current policy utilizes the available guidance and how broadly the policy is exploring. This motivates treating context revision as a state-dependent intervention, where the form of an update is conditioned on the evolving policy state.

2.3 Context Revision as a State-Dependent Intervention

Motivated by the dynamics in Figure 2, we characterize the current policy along two complementary coordinates: policy entropy captures behavioral exploration, while section-level attention provides an operational proxy for context utilization. Their combination helps distinguish policy states that may exhibit similar outcome feedback but call for different context interventions. Low exploration accompanied by weak use of a strategy section suggests a different intervention from low exploration accompanied by strong use. The same distinction applies when exploration is relatively high.

These coordinates guide candidate generation, while frozen-weight comparisons test candidate's immediate effects. Subsequent learning under shared evaluation contexts tests hypothesis that state-conditioned revision improves parameter acquisition. Attention remains an utilization proxy, and the reference band a calibration device; neither assumed to identify an optimal learning state.

3 CO-EVOLVING CONTEXT AND PARAMETERS

3.1 PARAMETER–CONTEXT CO-EVOLUTION

We formulate recursive self-improvement as a coupled learning process in which parameters and context evolve together. At iteration t, the state of the learning system is represented by (θ_t, p_t), where θ_t denotes the current model parameters and p_t denotes the system prompt used to condition task interaction. We decompose the prompt as $p_t = c \oplus s_t$, where c contains the fixed task specification and interface constraints, while s_t is an editable strategy scaffold. For a task x sampled from the training distribution and a trajectory τ generated by the current policy, the task objective is

$$J(\theta_t, p_t) = \mathrm{E} * x, \tau \left[R(x, \tau) \right], \qquad \tau \sim \pi * \theta_t(\cdot \mid x, p_t). \tag{1}$$

The reward remains the optimization objective throughout training; entropy and attention are used only to characterize the current policy state and guide context evolution.

Training alternates between parameter learning and context adaptation. During an RL window, the prompt p_t is held fixed and the policy collects fresh on-policy trajectories for parameter updates. After the window, the updated checkpoint is frozen temporarily and used to determine whether the current strategy scaffold is still well matched to the policy. If a context revision is accepted, the resulting prompt p_{t+1} is used to generate the next round of on-policy experience. Consequently, a context update affects not only how the current model behaves, but also which trajectories become training data for the next parameter update. Conversely, changes in θ_t alter how the model interprets and uses the same contextual guidance. The two components therefore form a feedback loop: context evolution changes future learning experience, while parameter learning changes the policy for which that context must be effective.

This formulation realizes a bounded form of RSI. The model parameters and editable strategy context participate in repeated adaptation, whereas the task objective, optimizer, evaluator, and execution interface remain fixed. The purpose of this setting is to study the interaction between two adaptive components of the learning system rather than unrestricted self-modification.

3.2 POLICY-STATE CHARACTERIZATION

A context should not be revised solely because another prompt obtains a higher reward under the current checkpoint. Such a criterion treats context optimization as an independent outer-loop search and does not explain how the current prompt has become mismatched with the evolving policy. We instead characterize the policy–context relationship along two complementary dimensions: behavioral exploration and instruction utilization. The first describes how broadly the policy distributes probability over possible continuations, while the second describes how strongly the policy uses different parts of the strategy scaffold.

We use policy entropy as the exploration signal. For assistant-generated token positions, we define

$$H_t(p) = \mathrm{E} * x, \tau, j \left[-\sum_{v} \pi * \theta_t(v \mid x, p, y_{<j}) \log \pi_{\theta_t}(v \mid x, p, y_{<j}) \right]. \tag{2}$$

A relatively low value of $H_t(p)$ indicates that the current policy is concentrated on a narrower set of continuations, whereas a relatively high value indicates broader behavioral exploration. We do not assume that a single absolute entropy value is optimal throughout training. Because the policy itself changes over time, the entropy of the current prompt is interpreted relative to historical mean. This produces a policy-relative exploration state and avoids treating global changes in model confidence as context-specific effects.

To measure instruction utilization, we partition the editable scaffold into semantic sections by splitting it into sentences. This avoids the poor generalization caused by dividing the scaffold into fixed sections. We then estimate how much attention the generated response assigns to each section. Let I_k denote the prompt tokens belonging to section k, and let $\alpha_{j,i}$ denote the attention from response token j to prompt token i. The utilization of section k is summarized as

$$U_{t,k}(p) = \mathrm{E} * x, \tau, j \left[\frac{1}{|I_k|} \sum * i \in I_k \alpha_{j,i} \right]. \tag{3}$$

Exploration	Utilization	Intervention	Context update
Low	Low	**Expand**	The policy is behaviorally concentrated while making limited use of the existing strategy guidance. Introduce an alternative strategy or scaffold to enlarge the available behavioral space.
Low	High	**Relax**	The policy is concentrated while strongly relying on the current guidance. Soften overly prescriptive instructions allowing alternative reasoning paths when appropriate.
High	Low	**Clarify**	The policy explores broadly but makes weak use of the relevant guidance. Rewrite the selected section so that its intended strategy, priority, or activation condition is more explicit.
High	High	**Consolidate**	The policy remains highly exploratory despite strong contextual utilization. Merge redundant or competing guidance into a more coherent strategy scaffold to reduce fragmentation.

Table 1: Policy-aware context intervention. Exploration is measured by policy entropy, while utilization is estimated from attention to the relevant strategy section. Both signals are interpreted relative to the current policy's calibrated reference range.

The length normalization prevents longer sections from receiving a systematically larger score simply because they contain more tokens. As with entropy, utilization is interpreted relative to measurements obtained from the current checkpoint rather than through a fixed threshold. In practice, attention can also be restricted to response stages associated with the semantic role of a section.

Entropy and utilization capture different aspects of the evolving learning state. Policy entropy indicates whether the current behavior is comparatively narrow or diffuse, while section-level utilization indicates whether the strategy guidance is being weakly or strongly used. Their combination therefore provides a policy-dependent signal for deciding both the direction and the form of a context intervention. We use attention as an operational proxy for instruction utilization, not as evidence that attention itself is causally responsible for model behavior (Jain & Wallace, 2019).

3.3 Policy-Aware Context Evolution

At each context-update boundary, we characterize the current policy along two complementary dimensions: *exploration*, measured by policy entropy, and *utilization*, estimated from the policy's attention to the relevant strategy section. Exploration captures the diversity of the policy's current behavior, whereas utilization reflects how strongly that behavior is conditioned on the editable contextual guidance. Their joint state therefore provides a more informative signal for context adaptation than either quantity alone.

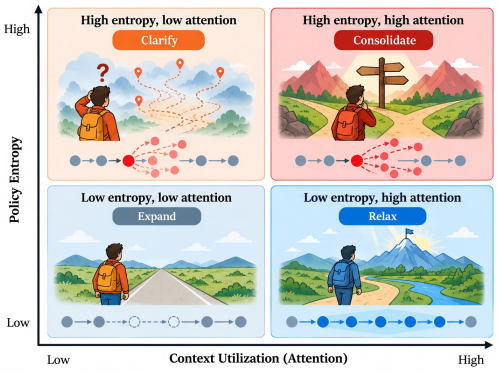


Figure 3: **Policy-aware context intervention.** Entropy measures behavioral exploration, while attention reflects policy-relative context utilization. Their combination determines how the strategy context is adapted to the current policy.

Figure 3 shows the resulting entropy–attention state space. Intuitively, low exploration indicates that the policy has become behaviorally concentrated, while high exploration indicates that multiple behavioral modes remain active. Low utilization suggests that the relevant contextual guidance is exerting little influence on the policy, whereas high utilization indicates that the policy is strongly conditioned on that guidance. These two axes induce four qualitatively different intervention regimes, summarized in Table 1.

The four regimes specify the *direction* of context adaptation rather than an unconditional editing rule. We intervene only when the measured policy state provides sufficient evidence that a context

update is warranted. In particular, if exploration remains within its calibrated reference range, or if the utilization estimate is insufficiently reliable, the incumbent prompt is preserved. When multiple editable sections exhibit substantial utilization deviations, the controller selects the section with the strongest policy-relative mismatch. The detailed editing prompts are provided in Appendix B.

Given a selected intervention, the current policy checkpoint is used as a constrained context editor. The editor receives the incumbent prompt, the target strategy section, and the intervention type, and proposes a small set of local revisions. Only the editable strategy scaffold may be modified; the task specification, output requirements, and other immutable instructions remain fixed. This restriction preserves the underlying objective and isolates context evolution from changes to the task itself.

Importantly, a revision is not accepted merely because its textual form appears consistent with the intended intervention. Context changes are policy dependent: adding an alternative strategy does not necessarily increase exploration, relaxing an instruction does not necessarily diversify behavior, and clarifying a strategy does not guarantee greater utilization. We therefore evaluate every candidate revision using the same frozen checkpoint from which the policy-state measurements were obtained. A candidate is accepted only if it shifts the relevant policy state in the intended direction while preserving task performance within a predefined tolerance. If no candidate satisfies these conditions, the incumbent prompt is retained. The complete training and context-evolution procedure is provided in Algorithm 1(Appendix A.4).

After a context revision is accepted, on-policy training resumes under the updated prompt. The resulting co-evolution cycle is

$$(\theta_t, p_t) \longrightarrow (H_t, U_t) \longrightarrow p_{t+1} \longrightarrow \mathcal{D}_{t+1}^{\text{on}} \longrightarrow \theta_{t+1}, \tag{4}$$

where (H_t, U_t) summarizes the interaction between the current policy and its contextual strategy scaffold, p_{t+1} denotes the verified context revision, and $\mathcal{D}_{t+1}^{\text{on}}$ is the subsequent on-policy experience generated under that context. The context update thus changes the experience distribution used for parameter learning, while the resulting parameter update changes the policy state against which the next context revision is evaluated. Repeating this process closes the feedback loop between parameter learning and context evolution.

4 EXPERIMENTS

Models and domains. We evaluate Qwen3.5-4B, Qwen3.5-4B-Base, Qwen3.5-9B and Qwen3.8-27B on mathematical reasoning and code generation. Math models are trained on DAPO-Math-17k and evaluated on AIME 2025 (Yu et al., 2025). Code experiments use a temporal split derived from LiveCodeBench (Jain et al., 2024). For each model scale, code RL starts from a common SFT checkpoint trained on 300 verified trajectories. Parameter updates use LoRA-based, GRPO-style optimization. Detailed training configurations, and execution protocols are provided in Appendix A.

Baselines. We compare COEVO with fixed-prompt GRPO using Simple, Medium, and Detail training contexts. These baselines keep the context unchanged throughout RL, allowing us to assess the benefit of adapting contextual guidance during training. We further evaluate the resulting checkpoints under shared inference prompts to distinguish training effects from differences in evaluation-time context. Given the computational cost of full RL training, we restrict comparisons with E-SPL-style baselines to the ablation setting Zhang et al. (2026).

Method Settings. COEVO performs one context-evolution attempt every five RL steps. At each attempt, the current policy is frozen to guide context revision and evaluate candidate prompts. If no candidate satisfies the acceptance criteria, the current prompt is retained.

Evaluation. We report Average@12 and Pass@12 in both domains. Average@12 measures the mean per-problem success rate across samples, while Pass@12 measures the fraction of problems solved by at least one sample. Cross-prompt evaluation applies the same fixed evaluation prompts to each checkpoint, assessing both performance under shared contexts and sensitivity to prompt changes. Code results are macro-averaged over the three evaluation prompts and refer to our local temporal split rather than the official LiveCodeBench benchmark.

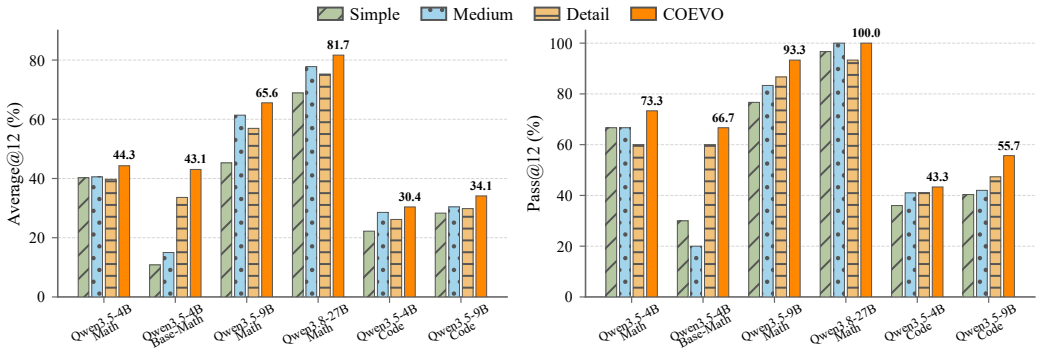


Figure 4: Performance comparison of COEVO with different prompt complexity levels across Qwen3.5 model scales and task domains. Simple, Medium, and Detail denote fixed prompts with increasing levels of prompt specification, while COEVO represents our method.

5 Results and Analysis

5.1 Main Results

Performance improvement. Figure 4 compares COEVO with fixed-prompt RL across model scales and task domains. On Math, COEVO consistently improves both Average@12 and Pass@12 at 4B and 9B scales. The improvement in Average@12 indicates that co-evolution increases the overall quality of sampled responses rather than benefiting only a small subset of generations, while the simultaneous improvement in Pass@12 shows that the learned policy is also more likely to produce at least one successful solution within a fixed sampling budget. Importantly, stronger fixed prompts alone do not lead to monotonic improvements: increasing prompt detail can improve one metric while degrading another. In contrast, COEVO adapts the training context as the policy evolves and achieves the strongest overall performance, suggesting that the benefit comes from matching contextual guidance to the evolving policy rather than simply supplying more detailed instructions.

Robustness to inference prompts. Table 2 evaluates each trained policy under all three inference prompts.COEVO achieves the strongest worst-case performance under both metrics. Its worst-case Average@12 is 43.06%, exceeding the strongest fixed-context baseline by 2.78 percentage points, while its worst-case Pass@12 reaches 73.33%, a 6.66-point improvement. Moreover, Pass@12 is identical across all three inference prompts, indicating that task-level coverage is insensitive to the evaluation prompt in this setting. For Average@12, the Detail-trained baseline exhibits a smaller max–min gap (0.83 versus 2.22), but at a substantially lower performance level (38.89 worst-case versus 43.06 for COEVO). Thus, low variation alone does not imply strong robust performance: COEVO combines high average performance with substantially stronger worst-case behavior across inference contexts. Overall, these results suggest that context evolution during training also improves the learned policy's robustness to inference-time prompt variation.

5.2 Case Study

To understand how the external context evolves with the policy, we analyze two representative training trajectories. Case I shows how different policy states lead to different types of context revision, while Case II provides complementary evidence that evolution can remove redundant guidance rather than continually adding instructions.

Context revision depends on the policy state. Case I contains 20 attempted revisions, of which 11 are accepted. Table 3 highlights representative accepted transitions. When exploration and section utilization are both low, the controller introduces additional structure: generic tool-use advice is replaced with explicit problem classification, and later expanded into symbolic, numerical, and search categories with a concrete tool-selection rule. When utilization becomes high under low exploration, the direction reverses and rigid requirements are relaxed. High-exploration states instead lead to stronger organization or consolidation of the existing strategy.

Metric	Training	Simple	Medium	Detail	Mean ↑	Worst ↑	Δ ↓
Average@12	Simple	44.72	40.28	40.28	41.76	40.28	4.44
	Medium	42.78	41.94	40.28	41.67	40.28	2.50
	Detail	39.17	38.89	39.72	39.26	38.89	**0.83**
	COEVO	43.06	45.28	44.17	**44.17**	**43.06**	2.22
Pass@12	Simple	66.67	70.00	66.67	67.78	66.67	3.33
	Medium	63.33	70.00	66.67	66.67	63.33	6.67
	Detail	63.33	63.33	60.00	62.22	60.00	3.33
	COEVO	73.33	73.33	73.33	**73.33**	**73.33**	**0.00**

Table 2: **Cross-prompt robustness on 4B-Math.** Models trained with different contexts are evaluated under the same Simple, Medium, and Detail inference prompts. **Worst** denotes the minimum performance across inference prompts, and Δ denotes the max–min gap; higher Worst and lower Δ indicate stronger robustness to inference-prompt variation.

Step	State (Ent./Util.)	Context revision
4	L / L-L-L	Generic tool advice → explicit strategy selection
29	L / L-L-L	Add problem categories and a cost–benefit decision rule
49	H / L-L-L	Strengthen guidance into an explicit decision procedure
54	L / H-H-H	"must invoke" → "consider invoking"
69	L / L-L-H	Reintroduce classification and clearer tool-use conditions
99	H / H-H-L	Consolidate guidance into a concise three-step procedure

Table 3: Representative accepted context revisions in Case I.

Context evolution is non-monotonic. COEVO does not simply make the prompt more detailed or restrictive. Early revisions add structure rules, while later revisions soften the same constraints once the policy relies heavily on them. Subsequent states can trigger more explicit guidance before the strategy is consolidated. Moreover, only 11 of 20 proposed revisions are accepted, showing that context evolution is selective rather than the cumulative addition of editor-generated changes.

Evolution can also remove redundant guidance. Case II exhibits a complementary pattern. The initial context contains detailed operational instructions about tool invocation and output formatting. During evolution, several of these details are removed or rewritten into more direct instructions while the core behavioral requirements remain intact. Table 4 summarizes the two patterns.

Together, the two cases provide a concrete view of parameter–context co-evolution. Parameter updates change how contextual guidance is used, while context updates reshape the conditions under which subsequent experience is generated. The context therefore evolves toward different forms at different stages rather than toward a single fixed prompt structure.

5.3 Ablation and Mechanism Analysis

We compare COEVO against two single-signal variants and an E-SPL-style reward-driven baseline inspired by Zhang et al. (2026).

Method	Avg.	Pass	$\overline{H}$
Attn.-only	40.2	66.7	0.35
Ent.-only	42.0	70.0	0.47
E-SPL	41.6	66.7	0.31
COEVO	**44.3**	**73.3**	**0.52**

Table 5: Ablation on Qwen3.5-4B Math.

Attention-only retains context-utilization feedback but removes the explicit exploration signal. It can identify which parts of the scaffold are being used, but cannot detect whether the policy is becoming overly concentrated; correspondingly, its entropy decreases substantially over training. Entropy-only exhibits the complementary failure mode: it observes changes in exploration but cannot localize how the current scaffold is being utilized, leading to weaker downstream performance. Together, these results suggest that entropy and attention provide complementary information for selecting context interventions.

The E-SPL-style baseline reveals a different failure mode. Although it achieves stronger on-policy performance during training, its held-out Average@12 and Pass@12 reach only 41.61% and 66.67%,

Case	Observed evolution	Implication
Case I	Structure → stronger constraints → relaxation → consolidation	Appropriate guidance changes with the current policy state.
Case II	Detailed operational guidance → more direct strategy instructions	Redundant guidance can be removed while preserving core task requirements.

Table 4: Complementary context-evolution patterns observed in the two case studies.

trailing COEVO by 2.70 and 6.66 points, respectively. This performance gap coincides with a larger reduction in policy entropy and output diversity, consistent with premature exploitation: reward-driven context updates can reinforce immediately successful strategies while narrowing the policy before learning is complete. In contrast, COEVO conditions context revision on both exploration and utilization, allowing the scaffold to adapt while preserving behavioral diversity.

6 Related Work

Reinforcement learning and policy-state characterization. Reinforcement learning has become an important mechanism for improving reasoning and decision-making in language models. GRPO and DAPO develop scalable on-policy optimization methods, while ReTool studies reinforcement learning with execution feedback (Shao et al., 2024; Yu et al., 2025; Feng et al., 2025). Recent work further explores adaptive training curricula and self-play for continued policy improvement (Ye et al., 2025). Studies of policy entropy reveal its relationship with exploration, reasoning behavior, and premature convergence during RL (Cui et al., 2025; Cheng et al., 2026; Hao et al., 2026; Xu et al., 2026). Meanwhile, attention provides a localized, though interpretation-sensitive, view of how models utilize instructional context (Jain & Wallace, 2019; Wiegreffe & Pinter, 2019). We use policy entropy and prompt-conditioned attention as complementary descriptors of the evolving policy state, capturing behavioral exploration and instruction utilization, respectively. Rather than optimizing these signals as standalone objectives, we use them to guide context evolution during reinforcement learning.

Recursive self-improvement. Recent work on recursive self-improvement explores how language models can iteratively improve their external context, learning experience, or computational structures. Reflexion and CURE study feedback-driven refinement and iterative self-improvement (Shinn et al., 2023; Chen et al., 2026a), while OPRO, EvoPrompt, Promptbreeder, GEPA, and TextGrad optimize prompts or system components through search, evolutionary operators, or textual feedback (Yang et al., 2023; Guo et al., 2023; Fernando et al., 2023; Agrawal et al., 2025). LSE trains models to evolve their contexts at test time, while in-context RL demonstrates improvement from reward-conditioned interaction histories (Chen et al., 2026b; Song et al., 2026). SPEE combines progressive experience evolution with policy optimization (Ren et al., 2026). Beyond prompt-level adaptation, STOP, Self-Developing, and the Darwin Godel Machine investigate self-improvement through program, algorithm, or agent-code modification (Zelikman et al., 2023; Ishibashi et al., 2025; Zhang et al., 2025). E-SPL couples system-prompt evolution with reinforcement-learning-based parameter updates (Zhang et al., 2026). Our work focuses on policy-aware parameter–context co-evolution: entropy and prompt-conditioned attention guide system-context adaptation during RL, while the surrounding optimization and evaluation machinery remains fixed.

7 Conclusion

This work presents parameter–context co-evolution as a broader perspective for understanding recursive self-improvement in LLMs. Rather than viewing RSI solely as repeated optimization in parameter space, we argue that the external context shaping model behavior and learning can itself become part of the adaptive process. This perspective highlights RSI as the coordinated evolution of multiple interacting components, where improvements in the model change how contextual guidance is used, while changes in context reshape the experience that drives further learning. By studying this interaction within a bounded and controllable setting, our framework provides a useful foundation for analyzing more general forms of self-improving systems and for understanding how different adaptive mechanisms may jointly contribute to recursive capability development.

AI USE STATEMENT

AI-assisted tools from OpenAI were used during the preparation of this work for methodological discussion, manuscript drafting and revision, literature search, LaTeX authoring, and figure-generation code. All experimental design decisions, reported results, and manuscript content were reviewed and verified by the authors.

ETHICS STATEMENT

Our framework restricts self-modification to the model parameters and an editable strategy component of the system context. Task objectives, evaluation procedures, essential task constraints, and execution interfaces remain fixed throughout training. Code execution follows the same isolated environment and evaluation protocol across experimental conditions. These restrictions provide a bounded setting for studying recursive self-improvement while limiting the scope of autonomous modification.

REPRODUCIBILITY STATEMENT

We provide the context-evolution procedure, prompt templates, measurement protocols, experimental settings, and ablation configurations in the Appendix. We also report the evaluation prompts and cross-prompt evaluation protocol used in our experiments. Code and additional experimental artifacts will be released to support reproduction of the reported results.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. URL https://arxiv.org/abs/2507.19457.

Guirong Chen, Shuqi Ye, Wenkai Yang, Shiqi Shen, Guangyao Shen, and Yankai Lin. CURE: Critique-driven unified reinforcement learning for test-time self-improvement. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28632–28653. Association for Computational Linguistics, 2026a. doi: 10.18653/v1/2026.acl-long.1321. URL https://aclanthology.org/2026.acl-long.1321/.

Xiaoyin Chen, Canwen Xu, Yite Wang, Boyi Liu, Zhewei Yao, and Yuxiong He. Learning to self-evolve, 2026b. URL https://arxiv.org/abs/2603.18620.

Daixuan Cheng, Shaohan Huang, Xuekai Zhu, Bo Dai, Xin Zhao, Zhenliang Zhang, and Furu Wei. Reasoning with exploration: An entropy perspective. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(36):30377–30385, 2026. doi: 10.1609/aaai.v40i36.40290.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025. URL https://arxiv.org/abs/2505.22617.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025. URL https://arxiv.org/abs/2504.11536.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023. URL https://arxiv.org/abs/2309.16797.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023. URL https://arxiv.org/abs/2309.08532.

Zhezheng Hao, Hong Wang, Haoyang Liu, Jian Luo, Jiarui Yu, Hande Dong, Qiang Lin, Can Wang, and Jiawei Chen. Rethinking entropy interventions in RLVR: An entropy change perspective. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31105–31133. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1436. URL https://aclanthology.org/2026.acl-long.1436/.

Yoichi Ishibashi, Taro Yano, and Masafumi Oyamada. Can large language models invent algorithms to improve themselves?: Algorithm discovery for recursive self-improvement through reinforcement learning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 10332–10363. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.519. URL https://aclanthology.org/2025.naacl-long.519/.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Sarthak Jain and Byron C. Wallace. Attention is not explanation. In *Proceedings of NAACL-HLT*, pp. 3543–3556, 2019. URL https://aclanthology.org/N19-1357/.

Shijie Ren, Xiting Wang, Meng Li, Yujie Guo, Yunhang Yao, Ziheng Peng, Xunlong Wang, Yuetan Chen, Haoyang Zhou, Yunlong Liang, and Fandong Meng. Self-improving large language models via progressive experience evolution, 2026. URL https://arxiv.org/abs/2608.02139.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366.

Kefan Song, Amir Moeini, Peng Wang, Lei Gong, Rohan Chandra, Shangtong Zhang, and Yanjun Qi. Reward is enough: LLMs are in-context reinforcement learners. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/b7511dfe2e7a1fa45e093cc75389abc2-Abstract-Conference.html.

Sarah Wiegreffe and Yuval Pinter. Attention is not not explanation. In *Proceedings of EMNLP-IJCNLP*, pp. 11–20, 2019. URL https://aclanthology.org/D19-1002/.

Huimin Xu, Shuai Zhao, Xiaobao Wu, and Anh Tuan Luu. Understanding and preventing entropy collapse in RLVR with on-policy entropy flow optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17759–17771, 2026. doi: 10.18653/v1/2026.findings-acl.879. URL https://aclanthology.org/2026.findings-acl.879/.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023. URL https://arxiv.org/abs/2309.03409.

Ziyu Ye, Rishabh Agarwal, Tianqi Liu, Rishabh Joshi, Sarmishta Velury, Quoc V. Le, Qijun Tan, and Yuan Liu. Reward-guided prompt evolving in reinforcement learning for LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 71910–71937. PMLR, 2025. URL https://proceedings.mlr.press/v267/ye25a.html.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023. URL https://arxiv.org/abs/2310.02304.

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*, 2025. URL https://arxiv.org/abs/2505.22954.

Lunjun Zhang, Ryan Chen, and Bradly C. Stadie. Evolutionary system prompt learning for reinforcement learning in llms. *arXiv preprint arXiv:2602.14697*, 2026. URL https://arxiv.org/abs/2602.14697.

A Experimental Setup

A.1 Models and tasks

Model	Domain	Training data	Evaluation data
Qwen 3.5-4B	Math	DAPO-Math-17k	AIME 2025
Qwen 3.5-4B-Base	Math	DAPO-Math-17k	AIME 2025
Qwen 3.5-9B	Math	DAPO-Math-17k	AIME 2025
Qwen 3.8-27B	Math	DAPO-Math-17k	AIME 2025
Qwen 3.5-4B	Code	LCB-v6 temporal train	100-task stratified subset of frozen 200-task test split
Qwen 3.5-9B	Code	LCB-v6 temporal train	100-task stratified subset of frozen 200-task test split

Table 6: Experimental settings.

For Math, after holding out a 50-problem development split, the training set contains 17,867 unique problems, and AIME 2025 contains 30 evaluation problems. We use the tokenizer and chat template bundled with each checkpoint.

The Code experiments use a fixed temporal split derived from the LiveCodeBench release-v6 data. After excluding 175 pilot-task IDs, the train, calibration, checkpoint-development, and final-test splits contain 400, 60, 100, and 200 mutually disjoint tasks, ordered by release time. The same 100 task IDs are used for both model scales. They are sampled without replacement from the frozen 200-task test split using proportional difficulty stratification, yielding 23 easy, 33 medium, and 44 hard tasks. Both model scales use a shared SFT checkpoint trained for 3 epochs on 300 verified trajectories.

A.2 RL configuration

Setting	Math	Code
Initialization	Qwen 3.5-4B, Qwen 3.5-4B-Base, Qwen 3.5-9B, and Qwen 3.8-27B	Qwen 3.5-4B and Qwen 3.5-9B initialized from a shared SFT checkpoint
RL algorithm	GRPO-style group-mean advantage without standard-deviation normalization; PPO clip $[0.8, 1.28]$	GRPO-style group-mean advantage without standard-deviation normalization; PPO clip $[0.8, 1.28]$
Optimizer	Adam, $\beta_1 = 0.9$, $\beta_2 = 0.95$	Adam, $\beta_1 = 0.9$, $\beta_2 = 0.95$
Learning rate	4×10^{-5}	4×10^{-5}
Batch size (task per step)	8 (4B), 4 (9B, 27B)	4
Rollouts per task	8	8
Training steps	100	100
Training sampling	Temperature 1.0, top-p 1.0	Temperature 1.0, top-p 1.0
Checkpoint interval	50 steps (4B), 25 steps (9B, 27B)	20 steps
Training seed	42	42
LoRA rank	32	32

Table 7: RL training hyperparameters.

Math trajectories allow at most 6 assistant turns, at most 4 (4B) or 3 (9B, 27B) code-interpreter calls, 512 tokens per tool response, and a 30-second sandbox timeout. The maximum trajectory length is 8192 tokens for 4B and 32768 tokens for 9B and 27B, with per-turn limits of 1024 and 4096 tokens, respectively.

Code trajectories allow at most 2 assistant turns, 1 isolated and stateless `run_python` call, 10,240 tokens per assistant response, 20,480 total trajectory tokens, and 512 tool-response tokens.

A.3 EVALUATION PROTOCOL

Math and Code are evaluated using Average@12 and Pass@12. For each task, twelve responses are sampled under the same evaluation configuration.

Let x denote an evaluation task and $z_{x,r} \in \{0, 1\}$ indicate whether the r-th sampled response for task x is successful. Then

$$\text{Average@12}(x) = \frac{1}{12} \sum_{r=1}^{12} z_{x,r}, \tag{5}$$

$$\text{Pass@12}(x) = \mathbf{1}\left[\sum_{r=1}^{12} z_{x,r} > 0\right]. \tag{6}$$

For Math, the 4B evaluation uses temperature 1.0 and top-p 0.7, while the 9B and 27B evaluations use temperature 0.7 and top-p 0.9. The corresponding trajectory limits are 8192 tokens for 4B and 32768 tokens for 9B and 27B, with per-turn limits of 1024 and 4096 tokens, respectively. Generation stops when a `\boxed{}` answer is emitted or when the trajectory budget is exhausted. For answer checking, the last `\boxed{...}` group, including nested braces, is extracted; whitespace, commas, and `$` are removed before exact integer comparison with the AIME ground truth.

For Code, evaluation uses temperature 1.0, top-p 0.7, at most 10,240 tokens per assistant response, two assistant turns, one isolated and stateless `run_python` call, 20,480 total trajectory tokens, and 512 tool-response tokens. The tool-call timeout is 5 seconds, the per-test-case timeout is 3 seconds, and the whole-problem judge timeout is 30 seconds.

For Code, a response is successful only when the submitted program passes the full judge. Average@12 is the empirical success rate across the twelve generations, while Pass@12 is the fraction of tasks for which at least one generation succeeds.

A.4 CONTEXT-EVOLUTION ALGORITHM

Update schedule and data reuse. Let θ_t and $p_t = c \oplus s_t$ be the policy and prompt before zero-based RL step t. Here c contains the fixed task specification, tool interface, and output constraints, and s_t is the editable strategy scaffold. At step t, the model samples training trajectories for task batch $\mathcal{B}_t$ under p_t and performs its ordinary RL update. Context evolution is attempted whenever $(t + 1) \bmod 5 = 0$, i.e., after zero-based steps $4, 9, \ldots$.

The controller uses the statistics and task examples from the last training step, rather than pooling the preceding five steps. The updated checkpoint $\bar{\theta} = \theta_{t+1}$ is frozen during prompt generation and comparison. Candidate prompts are tested on the same latest task examples $\mathcal{B}_t$ under this checkpoint. The five-step interval determines when editing is attempted; it does not define an additional evaluation set.

Historical reference used in the implementation. Let h_t and $u_{t,k}$ denote the entropy and utilization of strategy section k recorded in the latest training step. The controller compares these values with their cumulative historical means, including the current step:

$$\mu_t^H = \frac{1}{t+1} \sum_{j=0}^{t} h_j, \qquad \mu_{t,k}^U = \frac{1}{t+1} \sum_{j=0}^{t} u_{j,k}. \tag{7}$$

These historical means are computed from training records collected as the policy evolves. They are not obtained by evaluating a fixed set of reference prompts under one checkpoint. At the first context-update boundary, both exploration and utilization states are initialized to Low.

The frozen-checkpoint comparison is a separate operation: after generating candidate prompts, the incumbent and candidates are evaluated under the current frozen checkpoint on the same latest training batch. This comparison is used for candidate selection, not for constructing the historical reference.

Algorithm 1 COEVO with latest-step prompt comparison

Require: Initial checkpoint θ_0, prompt $p_0 = c \oplus s_0$, training data, RL configuration, number of steps T

Initialize the training-statistics history
for $t = 0, \ldots, T-1$ **do**
  Sample task batch $\mathcal{B}_t$
  $\mathcal{T}_t \leftarrow \mathrm{Rollout}(\theta_t, p_t, \mathcal{B}_t)$
  Record latest-step entropy h_t and section utilization $\mathbf{u}_t$
  Append these statistics to the historical records
  $\theta_{t+1} \leftarrow \mathrm{RLUpdate}(\theta_t, \mathcal{T}_t)$
  $p_{t+1} \leftarrow p_t$
  if $(t+1) \bmod 5 = 0$ **then**
    Compute historical means, including the current step
    Compare $(h_t, \mathbf{u}_t)$ with the historical means
    At the first boundary, initialize both states to Low
    Determine edit direction d_t and section instructions
    Freeze $\bar{\theta} \leftarrow \theta_{t+1}$
    Generate eight candidate edits using $\bar{\theta}$ and Appendix B
    $\mathcal{C}_t \leftarrow \{p_t\} \cup \{\text{valid edited prompts}\}$
    for each $p \in \mathcal{C}_t$ **do**
      Sample responses under $(\bar{\theta}, p)$ on the same latest tasks $\mathcal{B}_t$
      Compute $z_t(p)$ using Eq. (8)
    end for
    $p_{t+1} \leftarrow \mathrm{BestInDirection}(\mathcal{C}_t, \{z_t(p)\}, d_t)$
  end if
end for
return (θ_T, p_T)

Generation and comparison. The current frozen checkpoint proposes eight edited strategy scaffolds. Each candidate preserves the immutable component c. Let $\mathcal{C}_t$ contain the incumbent p_t and the valid edited prompts. For each $p \in \mathcal{C}_t$, generate fresh responses to the same last-step task examples under $\pi_{\bar{\theta}}(\cdot \mid x, p)$ and compute the comparison statistics:

$$z_t(p) = \big(\widehat{H}_t(p), \widehat{\mathbf{U}}_t(p), \widehat{R}_t(p)\big). \tag{8}$$

The incumbent is re-evaluated under the same checkpoint rather than being compared using its old training trajectories. Candidate comparisons reuse the existing rollout count and decoding configuration; there is no separate verification-batch-size or verification-rollout-count hyperparameter. The evaluation task IDs and actual generation settings must be recorded.

The next prompt is the candidate ranked best in the requested direction:

$$p_{t+1} = \mathrm{BestInDirection}\big(\mathcal{C}_t, \{z_t(p) : p \in \mathcal{C}_t\}, d_t\big). \tag{9}$$

State carried to the next step. Only the selected strategy scaffold changes; the task interface and reward definition remain fixed. The next RL step collects fresh on-policy trajectories under the selected prompt. A context update performed after the final RL step can change the returned prompt but cannot change the already trained weights.

B PROMPTS

B.1 FIXED PROMPTS

B.1.1 MATH PROMPTS

The following text is the protocol used in the reported math runs.

Math Simple: complete system prompt.

````
You solve math problems step by step. When you need to compute something,
write Python code in a code block like this:
```python
# your code here
print(result)
```
After you write a code block, I will execute it and show you the output.
Then you can continue reasoning. Put your final answer in \boxed{}.
````

Math Medium: complete system prompt.

```
You solve math problems step by step with help from a Python code
    interpreter.
Use the code_interpreter tool when calculation, symbolic manipulation, or
enumeration helps you solve the problem accurately and quickly.

How to use the code_interpreter tool:
- Call code_interpreter with a `code` string containing Python code.
  Call it at most once per assistant turn, then wait for the execution
    result.
- Results are captured from what your code prints with print().
  Always print the values you want to see.
- Each execution is independent: no variables, files, or state carry over
  between calls. Redefine everything you need in each piece of code.
- Code must finish within a few seconds and use little memory.
  Do not read or write files. If you enumerate or brute-force,
  keep the search space small.
- If the execution returns an error, analyze it and retry with corrected
  code when useful.

When you have the final answer, end with exactly one line in this format:
\boxed{<your final answer>}
Do not call the tool and give the final answer in the same turn.
```

Math Detail: complete system prompt.

```
You solve math problems step by step with help from a Python code
    interpreter.
Use the code_interpreter tool when calculation, symbolic manipulation, or
enumeration helps you solve the problem accurately and quickly.

To call the tool, output EXACTLY this format and then STOP immediately:
<tool_call>
<function=code_interpreter>
<parameter=code>
# python code, use print() to show results
print(result)
</parameter>
</function>
</tool_call>
Do not write anything after </tool_call>. The system will execute it and
show you the result.

How to use the code_interpreter tool:
- Call code_interpreter with a `code` string containing Python code.
  Call it at most once per assistant turn, then wait for the execution
    result.
- Results are captured from what your code prints with print().
  Always print the values you want to see.
- Each execution is independent: no variables, files, or state carry over
  between calls. Redefine everything you need in each piece of code.
- Code must finish within a few seconds and use little memory.
  Do not read or write files. If you enumerate or brute-force,
```

```
  keep the search space small.
- If the execution returns an error, analyze it and retry with corrected
  code when useful.

When you have the final answer, end with exactly one line in this format:
\boxed{<your final answer>}
Do not call the tool and give the final answer in the same turn.
```

B.1.2 CODE PROMPTS

The following text is the protocol used in the reported code runs.

Code Simple: complete system prompt.

```
You are an expert Python competitive-programming agent. Solve the user's
    problem with a correct and efficient Python 3.11 program using only
    the standard library. Follow the supplied starter code when present;
    otherwise read from stdin and write to stdout.

You may call run_python to execute code with optional stdin. Each call is
     isolated and stateless. Call it at most once per assistant turn and
    wait for the result.

When the solution is ready, return exactly one fenced Python code block
    and no other text.

You may call run_python at most 1 times in total. A tool call must be a
    short, focused check rather than a full solution or a reasoning
    scratchpad; omit long comments and exploratory code. When the tool
    budget is exhausted, or when further testing is unnecessary,
    immediately submit the final code in the required format.
```

Code Medium: text appended to "Code Simple".

```
Reason concisely about the input/output, constraints, algorithm, and edge
     cases, then prioritize submitting the final implementation directly.
     Use run_python only for one short, focused check that resolves a
    specific uncertainty; never use it to draft the full solution or as a
     reasoning scratchpad. Reserve most of the response budget for the
    final program.
```

Code Detail: text appended to "Code Simple".

```
Follow this workflow:

1. Identify the required input/output behavior, constraints, and
    important edge cases.
2. Derive a correct algorithm and verify its time and space complexity.
3. Implement the algorithm carefully in the required starter-code or
    stdin/stdout format.
4. Use run_python to check the provided examples when useful.
5. Check important boundary or adversarial cases, and fix the code if a
    check fails.
6. Review correctness, complexity, and output formatting before
    submitting the final code.
```

`run_python` tool schema.

```
{
  "type": "function",
  "function": {
    "name": "run_python",
```

```
    "description": "Execute a short, focused Python 3.11 diagnostic in a
    fresh isolated container. Do not use the tool as a scratchpad or
    include long explanatory comments. Optional stdin is passed to the
    program. The call returns exit status, stdout, stderr, and timeout
    information. No files or state persist.",
    "parameters": {
      "type": "object",
      "properties": {
        "code": {"type": "string", "description": "Python code to execute
    ."},
        "stdin": {"type": "string", "description": "Optional standard
    input."}
      },
      "required": ["code"]
    }
  }
}
```

The chat-template call uses `tools=[run_python]`, `add_generation_prompt=True`, and `enable_thinking=False`. The final answer is accepted only when it contains exactly one non-empty fenced Python code block.

B.2 Prompt-evolution editor prompts

The editor LLM receives a JSON request whose direction field selects one of four mutation strategies. The system message is always "You edit system prompts. Return valid JSON only."

Request skeleton.

```
{
  "task": "Generate alternative system prompts.",
  "direction": "<Clarify|Consolidate|Expand|Relax>",
  "strategy": "<see below>",
  "span_instructions": {
    "<section_key>": "TARGET|MINOR EDIT: <reason>. <instruction>",
    ...
  },
  "rules": [
    "Return exactly one JSON object with the same section keys.",
    "Each candidate must be a distinct rewrite.",
    "TARGET spans get the full edit; MINOR EDIT spans get minimal changes
    .",
    "Preserve the output format (e.g. boxed answer).",
    "Do not mention metrics, entropy, attention, or this request.",
    "No markdown fence -- return raw JSON only."
  ],
  "sections": { "<key>": "<current text>", ... }
}
```

Strategy texts.

Expand Model ignores this section and is too deterministic. Convert abstract instructions into concrete strategy choices. Add missing thinking directions or decision frameworks.

Relax Model attends to this section but is too deterministic. Change absolute requirements into conditional suggestions. Allow alternative paths. Reduce rigid binding.

Clarify Model ignores this section and explores randomly. Rewrite vague instructions into clear goals with explicit priorities and decision conditions.

Consolidate Model attends to this section but explores randomly. Merge duplicates, eliminate conflicts, clarify priority and conditions.

Protocol	Step	Avg.@12	Pass@12	Format	Code calls
Simple	0	2.50	16.67	2.78	0.03
	50	12.78	30.00	22.78	0.01
	100	10.83	20.00	33.89	0.10
Medium	0	1.39	6.67	1.67	0.01
	50	11.11	23.33	17.50	0.00
	100	15.00	30.00	39.17	0.00
Detail	0	6.67	30.00	27.22	2.34
	50	21.39	40.00	54.44	0.52
	100	33.61	60.00	49.17	1.70

Table 8: Qwen3.5-4B-Base training trajectory on AIME 2025.

Span-instruction logic. For each section, the instruction is determined by the attention status and the direction:

```
if direction in (scaffold, strengthen):       # low-attention operations
    if att == low:   "TARGET: ... <strategy>"
    if att == high:  "MINOR EDIT: ... keep mostly unchanged"
else:                                          # high-attention
    operations
    if att == high:  "TARGET: ... <strategy>"
    if att == low:   "MINOR EDIT: ... keep mostly unchanged"
```

Sampling parameters. temperature=0.8, top_p=0.95, max_tokens=1024, num_samples = candidate count − 1 (default 8).

C Additional Experimental Results

C.1 Math Training Trajectories

C.1.1 Qwen3.5-4B-Base

Table 8 reports the Qwen3.5-4B-Base math training trajectory through step 100 under the three tool-use protocols. Format and Code calls are auxiliary diagnostics and are reported separately from the two task-level performance metrics.

C.1.2 Qwen3.5-9B

Table 9 reports the Qwen3.5-9B math training trajectory. The reported COEVO aggregate is 65.56% Average@12 and 93.33% Pass@12, achieving the highest.

C.2 Code Checkpoint Selection

Table 10 and Table 11 report the Qwen3.5-4B and Qwen3.5-9B code training trajectories on LCB-v6, respectively. S/M/D denote the All-Case Pass Rate under the Simple/Medium/Detail inference prompts, i.e., the percentage of tasks for which all test cases pass. This differs from Case, which is the mean test-case pass rate and gives partial credit, e.g., 3/5 test cases passed yields 0.6. All entries other than Step are percentages. Checkpoints are selected on a separate 100-task development split, where each candidate is evaluated under all three inference prompts with one greedy response per task and prompt. Selection maximizes Mean, then Worst, then Case; ties are broken by the earlier step. This yields Simple step 40, Medium step 100, and Detail step 100 for 4B-Code, and Simple step 100, Medium step 60, and Detail step 60 for 9B-Code. The unequal selected steps therefore reflect development-set model selection rather than unequal training budgets.

Protocol	Step	Avg.@12	Pass@12
Simple	0	31.94	53.33
	25	43.61	73.33
	50	50.28	83.33
	75	51.67	90.00
	100	45.28	76.67
Medium	0	49.44	76.67
	25	49.44	83.33
	50	61.39	86.67
	75	61.94	96.67
	100	61.39	83.33
Detail	0	40.83	70.00
	25	53.89	86.67
	50	51.11	80.00
	75	57.78	86.67
	100	56.94	86.67
COEVO	100	**65.56**	93.33

Table 9: Qwen3.5-9B Math training trajectory on AIME 2025.

Train	Step	S	M	D	Mean	Worst	Case
Simple	20	53.00	53.00	59.00	55.00	53.00	62.41
Simple	**40**	**57.00**	**57.00**	**56.00**	**56.67**	**56.00**	**64.05**
Simple	60	48.00	52.00	55.00	51.67	48.00	61.46
Simple	80	60.00	55.00	55.00	56.67	55.00	65.95
Simple	100	56.00	58.00	54.00	56.00	54.00	63.78
Medium	20	48.00	49.00	48.00	48.33	48.00	57.61
Medium	40	57.00	59.00	53.00	56.33	53.00	65.19
Medium	60	63.00	61.00	60.00	61.33	60.00	67.75
Medium	80	59.00	61.00	60.00	60.00	59.00	66.50
Medium	**100**	**59.00**	**64.00**	**63.00**	**62.00**	**59.00**	**68.40**
Detail	20	53.00	51.00	55.00	53.00	51.00	60.89
Detail	40	47.00	50.00	52.00	49.67	47.00	56.61
Detail	60	52.00	57.00	57.00	55.33	52.00	61.99
Detail	80	53.00	59.00	56.00	56.00	53.00	62.95
Detail	**100**	**63.00**	**67.00**	**63.00**	**64.33**	**63.00**	**71.76**

Table 10: Qwen3.5-4B Code Training Trajectory on LCB-v6.

C.3 Final Code Evaluation

The final 4B-Code and 9B-Code summaries are reported in Table 12. Here Base denotes the model without RL training, i.e., step 0. At 4B, the reported COEVO reaches 30.40% Average@12 and 43.33% Pass@12; while at 9B, COEVO is 34.12% Average@12 and 55.67% Pass@12, both reaching the highest success rate.

C.4 Aggregate Results

For compactness, Table 13 reports Math and Code in a single aligned table. The Qwen3.5-4B-Base fixed-prompt entries correspond to the step-100 checkpoints in Table 8. The 4B-Math entries follow the main Figure 5 results, with COEVO reaching 44.31% Average@12 and 73.33% Pass@12. Fixed-prompt Code entries are macro-averaged over the Simple, Medium, and Detail inference prompts.

Train	Step	S	M	D	Mean	Worst	Case
Simple	20	56.00	49.00	54.00	53.00	49.00	61.47
Simple	40	59.00	61.00	56.00	58.67	56.00	64.37
Simple	60	57.00	60.00	60.00	59.00	57.00	65.53
Simple	80	55.00	61.00	56.00	57.33	55.00	61.21
Simple	**100**	**60.00**	**60.00**	**58.00**	**59.33**	**58.00**	**64.27**
Medium	20	59.00	59.00	56.00	58.00	56.00	64.04
Medium	40	66.00	63.00	64.00	64.33	63.00	70.00
Medium	**60**	**71.00**	**69.00**	**69.00**	**69.67**	**69.00**	**76.21**
Medium	80	70.00	65.00	73.00	69.33	65.00	77.70
Medium	100	66.00	65.00	67.00	66.00	65.00	74.65
Detail	20	55.00	56.00	53.00	54.67	53.00	62.89
Detail	40	60.00	60.00	54.00	58.00	54.00	66.20
Detail	**60**	**64.00**	**65.00**	**71.00**	**66.67**	**64.00**	**72.75**
Detail	80	67.00	61.00	67.00	65.00	61.00	69.93
Detail	100	69.00	62.00	65.00	65.33	62.00	70.26

Table 11: Qwen3.5-9B Code Training Trajectory on LCB-v6.

Model	Method	Step	Avg.@12	Pass@12
Qwen3.5-4B	Base	0	17.17	32.67
	Simple	40	22.22	36.00
	Medium	100	28.56	41.00
	Detail	100	26.17	41.00
	COEVO	100	**30.40**	**43.33**
Qwen3.5-9B	Base	0	26.28	42.33
	Simple	100	28.31	40.33
	Medium	60	30.42	42.00
	Detail	60	29.83	47.33
	COEVO	100	**34.12**	**55.67**

Table 12: Final Code results on fixed-prompt checkpoints.

Table 13: Aggregate results across the principal Math and Code settings.

Domain	Setting	Method	Avg.@12	Pass@12
Math	4B	Simple	40.28	66.67
		Medium	40.55	66.67
		Detail	39.72	60.00
		COEVO	**44.17**	**73.33**
Math	4B-Base	Simple	10.83	20.00
		Medium	15.00	30.00
		Detail	33.61	60.00
		COEVO	**43.06**	**66.67**
Math	9B	Simple	45.28	76.67
		Medium	61.39	83.33
		Detail	56.94	86.67
		COEVO	**65.56**	**93.33**
Math	27B	Simple	68.89	96.67
		Medium	77.78	**100.00**
		Detail	75.28	93.33
		COEVO	**81.67**	**100.00**
Code	4B	Simple	22.22	36.00
		Medium	28.56	41.00
		Detail	26.17	41.00
		COEVO	**30.40**	**43.33**
Code	9B	Simple	28.31	40.33
		Medium	30.42	42.00
		Detail	29.83	47.33
		COEVO	**34.12**	**55.67**