SLM REASONS, LLM ADVISES: A COST-EFFICIENT COLLABORATIVE REASONING FRAMEWORK

Anonymous authors
Paper under double-blind review

ABSTRACT

Deploying large language models (LLMs) for reasoning-intensive tasks presents a fundamental cost–accuracy trade-off: frontier models achieve strong performance but incur substantial inference costs, whereas small local models are inexpensive but considerably less reliable. We show that this trade-off can be substantially mitigated through selective collaboration. We introduce SRLA, a refinement framework in which a small language model (SLM) performs the primary reasoning and selectively seeks feedback from a LLM only when its reasoning is likely to contain an error. SRLA determines when and where external guidance is needed using Chunk-level Guided Scoring (CGS), a lightweight adapter trained with direct preference optimization on chunk-level correctness preferences. At inference time, CGS evaluates intermediate reasoning without access to reference answers and identifies the least reliable portion of the solution. Only when the resulting score triggers the refinement gate does SRLA invoke the LLM advisor. The advisor receives the original problem, the complete solver trajectory, and the identified problematic chunk, and returns a structured diagnosis with targeted corrective feedback. The SLM then uses this feedback to refine its solution, while otherwise reasoning entirely locally. We evaluate SRLA on five mathematical reasoning and six code generation benchmarks using Qwen3.5-9B as the local solver and GPT-5.4 as the advisor. SRLA matches or even exceeds GPT-5.4 used as a standalone solver on the majority of benchmarks while reducing per-problem API cost by up to 92% on mathematical reasoning and 94% on code generation. On several benchmarks, the advisor is invoked only rarely, yielding negligible external API cost while preserving strong accuracy. These results show that frontier LLMs need not perform the entire reasoning process: selectively using them to diagnose and advise a capable local solver can achieve frontier-level reasoning performance at substantially lower cost.

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong capabilities on complex reasoning tasks, including mathematical problem solving and code generation. In particular, generating explicit intermediate reasoning steps can substantially improve performance on multi-step problems (Wei et al., 2022). However, strong reasoning performance often comes with substantial inference cost. Frontier models can solve difficult problems reliably, but repeatedly querying them at scale can be expensive, whereas smaller language models (SLMs) are considerably cheaper to deploy locally but remain less reliable on problems requiring long reasoning chains, precise computation, or complex decisions.

This cost–capability trade-off has motivated approaches that selectively use stronger models rather than invoking them for every query. LLM cascades such as FrugalGPT (Chen et al., 2023) and learned routing approaches such as RouteLLM (Ong et al., 2025) dynamically choose between models of different capabilities to reduce inference cost. In parallel, test-time refinement methods have shown that an initial solution can often be improved through critique and revision. Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), CRITIC (Gou et al., 2024), and REFINER (Paul et al., 2024), for example, use self-feedback, external feedback, tools, or learned critics to improve generated outputs. Despite their effectiveness, these approaches leave an important question unresolved: *When is external refinement actually necessary?*

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.

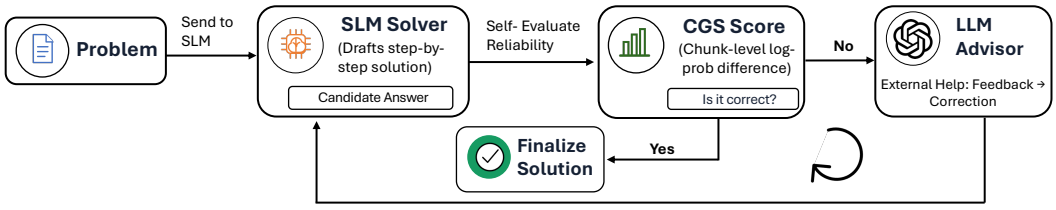

Figure 1: **Overview of SRLA.** An SLM first produces a candidate solution, which is evaluated locally using Chunk-level Guided Scoring (CGS). If the solution is sufficiently reliable, it is finalized locally. Otherwise, a frontier LLM advisor provides targeted corrective feedback, while the SLM produces the revised solution.

Answering this question is particularly challenging when the decision must be made by the inexpensive local model itself. An effective system should recognize an unreliable reasoning trajectory without access to the reference answer, without repeatedly querying a stronger model for verification, and without introducing a verification procedure whose cost eliminates the benefit of local inference. This creates a difficult self-assessment problem: the model responsible for producing an incorrect solution must also provide enough evidence to identify that its reasoning is unreliable.

We address this problem with **SRLA** (**S**LM **R**easons, **L**LM **A**dvises), a selective refinement framework in which an SLM performs the primary reasoning while a frontier LLM is reserved for targeted intervention. Rather than routing a problem to a stronger model based only on the input, SRLA evaluates the reasoning trajectory produced by the SLM and determines whether external assistance is needed. As illustrated in Figure 1, the SLM first generates a candidate solution, which is assessed locally using Chunk-level Guided Scoring (CGS). If the trajectory is considered sufficiently reliable, the solution is returned directly; otherwise, the frontier advisor is invoked to provide targeted corrective feedback.

CGS provides a chunk-level reliability signal learned through Direct Preference Optimization (DPO) (Rafailov et al., 2023). The local solver is first trained to self-correct failed reasoning using advisor-highlighted errors and targeted feedback, and is then preference-optimized to distinguish more reliable from less reliable reasoning chunks. At inference time, CGS is computed entirely locally and is used to decide whether the frontier advisor should be consulted. When external review is triggered, the advisor receives the original problem, the complete solver trajectory, and the lowest-scoring chunk as an additional localization cue. The advisor may inspect the full trajectory and identify an error anywhere in the solution; the highlighted chunk only guides attention toward the region of greatest concern. The resulting feedback is then used by the SLM to revise its own solution.

We evaluate SRLA across two reasoning-intensive domains using Qwen3.5-9B as the local solver and GPT-5.4 as the frontier advisor. Our evaluation covers five mathematical reasoning benchmarks (AIME 2024 (Zhang & Math-AI, 2024), AIME 2025 (Zhang & Math-AI, 2025), AIME 2026 (Zhang & Math-AI, 2026), AMC23 (Math-AI, 2023), and MinervaMath (Lewkowycz et al., 2022)) and six code-generation benchmarks (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CodeContests (Li et al., 2022), and the Easy, Medium, and Hard subsets of LeetCode (Xia et al., 2025).

Our contributions are summarized as follows:

- We introduce SRLA, a selective SLM–LLM collaboration framework in which an SLM performs the primary reasoning and a frontier LLM provides targeted assistance only when the local reasoning trajectory is estimated to be unreliable.
- We train the local solver for *highlighted-error self-correction*, where the advisor identifies the erroneous span and provides targeted corrective feedback, and the solver uses this localized guidance to correct its own failed reasoning and produce a revised solution.
- We introduce DPO-CGS, which combines DPO with CGS to learn fine-grained reliability signals over intermediate reasoning chunks. These scores are used at inference time to determine when external refinement is needed and to identify the lowest-scoring reasoning

chunk, which is then passed to the frontier advisor along with the original problem and the complete solver trajectory for targeted diagnosis and corrective feedback.

- We conduct a comprehensive evaluation across eleven benchmarks spanning mathematical reasoning and code generation. SRLA reduces per-problem API cost by up to 94% while matching or exceeding direct frontier-model accuracy on the majority of evaluated benchmarks, owing to the proposed selective collaboration scheme.

- We compare SRLA with established routing and iterative refinement approaches, including Self-Refine, Reflexion, CRITIC, REFINER, and RL-Tango. SRLA achieves higher accuracy than the strongest reported refinement baseline on 10 of the 11 evaluated benchmarks, including improvements of 40% on AIME 2026 and 18% on MBPP, demonstrating the effectiveness of selective frontier-model advising over conventional refinement strategies.

2 Related Work

Iterative Refinement and Feedback-Guided Correction. Prior work shows that language-model outputs can be improved through critique, feedback, and revision. Self Refine (Madaan et al., 2023) alternates between generation, self-feedback, and refinement, while CRITIC (Gou et al., 2024) incorporates external tool feedback into correction. REFINER (Paul et al., 2024) trains a separate critic to provide structured feedback over intermediate reasoning, which a generator then uses to improve its solution. More recently, DeepCritic (Yang et al., 2025) performs step-wise verification and produces detailed critiques for reasoning correction. These works demonstrate the value of fine-grained feedback for improving unsuccessful reasoning trajectories.

Small–Large Model Collaboration and Selective Routing. A growing line of work studies how models of different capabilities can collaborate while controlling inference cost. RouteLLM (Ong et al., 2025) learns to route queries between weaker and stronger models from preference data, while BEST-Route (Ding et al., 2025) jointly determines model selection and test-time sampling according to query difficulty. Route-and-Reason (Shao et al., 2025) extends this idea by decomposing problems into subtasks and assigning them to heterogeneous models. More closely related approaches retain the smaller model as the primary reasoner while using stronger models selectively. SMART (Kim et al., 2025) replaces unreliable reasoning steps with larger-model generations, MentorCollab (Wang et al., 2026) selectively incorporates mentor-generated lookahead segments, and COREA (Zhang et al., 2026) defers low-confidence queries to a larger model. These methods illustrate different strategies for reducing reliance on large-model inference through adaptive collaboration.

Process-Level Reliability and Cost-Efficient Reasoning. Another line of work develops fine-grained signals for evaluating intermediate reasoning and allocating test-time computation. Process reward models evaluate individual reasoning steps rather than only final answers. Rewarding Progress (Setlur et al., 2025) defines process rewards through changes in the likelihood of eventually reaching a correct solution, while AdaptiveStep (Liu et al., 2025) uses model confidence to identify reasoning-step boundaries. ThinkPRM (Khalifa et al., 2025) develops a generative verifier for intermediate reasoning, and ReasonFlux-PRM (Zou et al., 2026) combines step-level and trajectory-level supervision. RoRo (Ye et al., 2026) further connects process-level reliability estimation with adaptive model routing. Related work on test-time efficiency similarly allocates computation according to problem difficulty rather than applying a uniform inference budget (Ding et al., 2025; Shao et al., 2025; Setlur et al., 2025; Khalifa et al., 2025).

SRLA integrates selective SLM–LLM collaboration, localized advisor feedback, and chunk-level reliability estimation within a unified framework. Rather than routing the entire problem to a stronger model or directly replacing uncertain reasoning with frontier-model generations, SRLA keeps the SLM responsible for both the initial reasoning and subsequent self-correction. The local solver is trained to correct its own failed reasoning using an advisor-highlighted error span and targeted feedback. At inference time, SRLA derives chunk-level reliability signals locally through CGS and invokes the frontier model only when those signals indicate that external review may be useful. When invoked, the frontier model acts as an advisor rather than as the primary solver, providing targeted corrective feedback while the SLM remains responsible for producing the final solution.

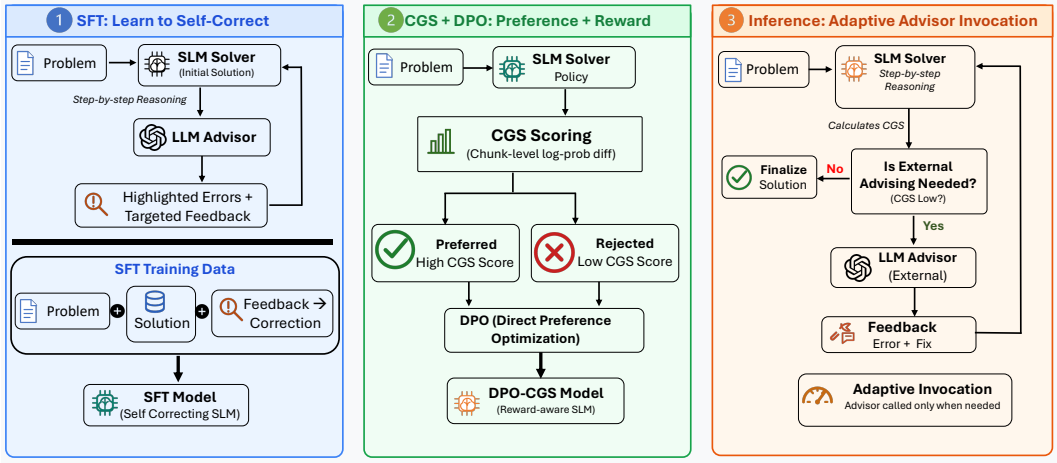

Figure 2: The SRLA framework. **Stage 1:** the SLM solver produces an initial step-by-step solution, which the LLM advisor annotates with a highlighted error span and targeted feedback; these (problem, solution, feedback → correction) triples form the SFT corpus that yields a self-correcting solver. **Stage 2:** solutions sampled from the policy are scored chunk-by-chunk with CGS — the log-probability difference between policy and reference — and high- versus low-scoring chunks form preference pairs for DPO, producing the reward-aware DPO-CGS model. **Stage 3:** at inference time, the SLM generates a complete solution, which is scored chunk by chunk using CGS. The chunk scores are aggregated to determine whether external feedback is needed. If the gate is triggered, the advisor receives the problem, complete trajectory, and lowest-scoring chunk as a localization cue, and provides corrective feedback that the SLM uses to generate a revised solution.

3 METHODOLOGY

SRLA trains an SLM to correct its own errors under supervision from a stronger model, and then, at inference time, spends that supervision sparingly. The framework has three stages, illustrated in Figure 2: supervised fine-tuning on advisor-corrected traces, preference optimization against a chunk-level reward, and an adaptive inference loop that invokes the external advisor only when the reward signal indicates it is needed.

3.1 STAGE 1: SUPERVISED FINE-TUNING FOR SELF-CORRECTION

As illustrated in Stage 1 of Figure 2, SRLA first trains the local solver to correct unsuccessful reasoning using *span-anchored advisor feedback*. Let π_θ denote the local SLM solver and $\mathcal{A}$ the external advisor. Given an input problem x, the solver generates an initial reasoning trajectory

$$y_{\text{init}} \sim \pi_\theta(\cdot \mid x).$$

When y_{init} is incorrect, the failed trajectory is provided to the advisor together with the original problem. Conditioned on x and y_{init}, the advisor identifies an erroneous span e, produces corrective feedback f, and generates a corrected trajectory y_{corr}:

$$(x, y_{\text{init}}) \xrightarrow{\mathcal{A}} (e, f, y_{\text{corr}}).$$

Here, e denotes the localized error, f the corresponding corrective guidance, and y_{corr} the corrected full solution generated in the context of the failed reasoning trajectory. Because the correction is conditioned on the failed attempt, the supervision preserves a direct connection between the solver's original error and the reasoning needed to correct it.

Each Stage 1 training instance is represented as

$$\big(x, y_{\text{init}}, e, f\big) \longrightarrow y_{\text{corr}}.$$

Defining the conditioning context as $z = (x, y_{\text{init}}, e, f)$, the resulting policy π_{sft} is trained to maximize the likelihood of the corrected trajectory. Equivalently, we minimize

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{|y_{\text{corr}}|} \log \pi_{\text{sft}} \left(y_{\text{corr},t} \mid z, y_{\text{corr},<t} \right).$$

The supervision is both *span-anchored* and *trajectory-conditioned*: feedback f is associated with a specific error span e in y_{init}, while the target is the complete corrected trajectory y_{corr}. This trains the solver to incorporate localized corrective feedback and propagate the correction through the resulting solution.

Unlike conventional solution-level SFT, which learns a mapping from a problem x to a target solution y, Stage 1 learns the feedback-conditioned transformation

$$(y_{\text{init}}, e, f) \longrightarrow y_{\text{corr}},$$

conditioned on x. The resulting policy π_{sft} provides the feedback-following capability required for selective collaboration and serves as both the initialization and frozen reference policy for Stage 2. Complete training details are provided in Appendix A.

3.2 Stage 2: Chunk-level Guided Scoring (CGS)

Stage 2 introduces CGS to measure how preference optimization changes the policy relative to the feedback-trained reference model at the level of individual reasoning chunks. Let c_k denote the k-th chunk of a reasoning trajectory and π_{sft} the frozen Stage 1 reference policy. For a chunk c_k, we define

$$\text{CGS}(c_k) = \frac{1}{|c_k|} \sum_{t \in c_k} \left[\log \pi_\theta(y_t \mid x, y_{<t}) - \log \pi_{\text{sft}}(y_t \mid x, y_{<t}) \right]. \tag{1}$$

Both likelihoods are computed by teacher forcing over the same prefix. Consequently, CGS measures the change in token likelihood induced by preference optimization while controlling for reasoning context. A positive CGS value indicates that π_θ assigns greater likelihood to the chunk than π_{sft}, whereas a negative value indicates a decrease relative to the reference policy. This chunk-level formulation preserves local variations in the reasoning trajectory that could be obscured by a single solution-level score. CGS therefore provides a localized signal over individual reasoning chunks without requiring the solver to explicitly predict whether its reasoning is correct.

Preference optimization. Starting from the Stage 1 policy, we optimize π_θ using Direct Preference Optimization (DPO), with π_{sft} kept fixed as the reference policy. For each preference triplet $(z_k, c_k^+, c_k^-) \in \mathcal{D}_{\text{pref}}$, c_k^+ denotes the preferred chunk from the corrected trajectory, c_k^- the rejected chunk from the failed trajectory, and z_k their associated reasoning context. The DPO objective is

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(z_k, c_k^+, c_k^-) \sim \mathcal{D}_{\text{pref}}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(c_k^+ \mid z_k)}{\pi_{\text{sft}}(c_k^+ \mid z_k)} - \beta \log \frac{\pi_\theta(c_k^- \mid z_k)}{\pi_{\text{sft}}(c_k^- \mid z_k)} \right) \right], \tag{2}$$

where β controls the strength of the preference objective. The resulting policy is denoted by π_{cgs}.

The same frozen Stage 1 policy π_{sft} is used as the reference for both DPO training and CGS computation. DPO encourages π_{cgs} to assign greater relative likelihood to preferred reasoning chunks, while CGS measures the resulting policy–reference likelihood shift. Thus, Stage 2 training and the selective-advisor gate are connected through the same policy–reference comparison.

3.3 Stage 3: Adaptive Advisor Invocation

At inference time, the frontier advisor is invoked only when the locally computed CGS signal indicates that external guidance may be beneficial. Stage 3 separates two functions of the chunk-level scores: their aggregate value determines *whether* the advisor should be invoked, while the lowest-scoring chunk indicates *where* the reasoning appears least supported relative to the reference policy.

Given an input problem x, SRLA proceeds as follows:

1. **Solve.** The local policy π_{cgs} generates an initial complete reasoning trajectory $y^{(1)} \sim \pi_{\text{cgs}}(\cdot \mid x)$.
2. **Score.** The generated trajectory is partitioned into K reasoning chunks $(c_1, \ldots, c_K)$, and CGS is computed for each chunk according to Equation 1. We aggregate these scores using their mean:
$$s_{\text{avg}} = \frac{1}{K} \sum_{k=1}^{K} \text{CGS}(c_k).$$
3. **Gate.** If $s_{\text{avg}} \geq \tau$, the trajectory is considered sufficiently reliable and finalized locally. Otherwise, the gate opens and SRLA invokes the frontier advisor.
4. **Localize.** When the gate opens, SRLA identifies the lowest-scoring chunk
$$k^* = \arg\min_k \text{CGS}(c_k), \qquad c_{k^*} = \arg\min_{c_k} \text{CGS}(c_k).$$
This chunk is not assumed to be the only erroneous part of the solution; rather, it serves as a localization cue indicating where the preference-optimized policy exhibits the weakest relative support.
5. **Advise.** When the gate opens, the advisor $\mathcal{A}$ receives the original problem x, the complete solver trajectory $y^{(1)}$, and the lowest-CGS chunk c_{k^*} as an additional diagnostic cue. The advisor inspects the complete trajectory and returns a highlighted error together with targeted corrective feedback.
6. **Revise.** If external guidance is obtained, the local solver performs a targeted revision of its initial solution conditioned on the advisor feedback:
$$y^{(2)} \sim \pi_{\text{cgs}}(\cdot \mid x, r_{\mathcal{A}}),$$
where $r_{\mathcal{A}}$ denotes the advisor-provided feedback. The revised trajectory incorporates this guidance while directly revising the first trajectory.

This design separates *global intervention* from *local diagnosis*. The aggregate CGS score controls whether frontier-model assistance is used, while the minimum-CGS chunk provides a localized cue for the advisor. Thus, the same chunk-level scores support both selective external computation and targeted corrective feedback.

4 Experiments

4.1 Setup

Models. We evaluate SRLA with multiple local solvers across the two task domains. For mathematical reasoning, we use `Qwen3.5-9B` (Qwen Team, 2026) and `Qwen2.5-Math-7B-Instruct` (Yang et al., 2024) as local solvers. For code generation, we use `Qwen3.5-9B` (Qwen Team, 2026) and `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024). GPT-5.4 serves as the frontier advisor in all settings. SRLA builds on each local solver through two stages of parameter-efficient adaptation: Stage 1 produces the feedback-conditioned policy π_{sft}, and Stage 2 produces the preference-optimized policy π_{cgs}. At inference time, π_{cgs} performs the primary reasoning, while GPT-5.4 is invoked only when the CGS-based gate triggers external review. We additionally evaluate the corresponding unrefined local solver (BASE) and direct GPT-5.4 inference.

Benchmarks. We evaluate SRLA across two reasoning-intensive domains: mathematical reasoning and code generation. The mathematical evaluation includes AIME 2024 (Zhang & Math-AI, 2024), AIME 2025 (Zhang & Math-AI, 2025), AIME 2026 (Zhang & Math-AI, 2026), AMC23 (Math-AI, 2023), and MinervaMath (Lewkowycz et al., 2022). For code generation, we use HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CodeContests (Li et al., 2022), and the Easy, Medium, and Hard subsets of LeetCode (Xia et al., 2025). In total, our evaluation covers eleven settings spanning competition-level mathematics, general-purpose program synthesis, and algorithmic coding problems of increasing difficulty. Descriptions of evaluation benchmarks are provided in Appendix B.1.

Evaluation and Baselines. We evaluate all methods using task accuracy and external API cost per problem. For mathematical reasoning, predictions are compared against the corresponding gold

Table 1: Mathematical reasoning performance with Qwen3.5-9B as the local solver. We report accuracy (%) and external API cost in U.S. cents per problem. Methods are grouped by whether they rely solely on local inference, use external feedback, or directly invoke the frontier model. Bold denotes the highest accuracy and, among methods using an external model, the lowest API cost for each benchmark. "–" indicates no external API cost.

	AIME-2024		AIME-2025		AIME-2026		AMC23		MinervaMath	
Method	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**
Local-only baselines										
BASE	30.00	–	23.33	–	30.00	–	75.00	–	66.18	–
Self-Refine	33.33	–	26.67	–	33.33	–	75.00	–	64.34	–
Reflexion	40.00	–	26.67	–	36.67	–	85.00	–	66.18	–
CRITIC	30.00	–	26.67	–	33.33	–	77.50	–	66.34	–
External-feedback baselines										
REFINER	40.00	0.41	33.33	0.41	40.00	0.42	77.50	**0.39**	68.01	0.35
RL-Tango	40.00	0.96	36.67	0.91	36.67	0.99	85.00	0.94	73.16	0.92
Frontier-model direct inference										
GPT-5.4	76.67	1.44	53.30	1.84	60.00	1.88	**92.50**	0.88	**76.50**	0.47
Ours										
SRLA	**80.00**	**0.14**	**63.30**	**0.14**	**80.00**	**0.14**	**92.50**	0.40	71.32	**0.04**

answers. For code generation, generated solutions are executed against the benchmark test cases and are considered correct only when they pass the required evaluation suite. Local-model inference incurs no external API cost; we therefore report external cost only for methods that invoke a frontier model or other external service. We compare SRLA against the corresponding local solver, direct GPT-5.4 inference, and several representative refinement and verification methods: Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), CRITIC (Gou et al., 2024), REFINER (Paul et al., 2024), and RL-Tango (Zha et al., 2026). These baselines cover self-generated refinement, reflection-based reasoning, tool-assisted critique, intermediate-feedback refinement, and verifier-guided optimization. A technical comparison is provided in Table 8 for these baselines.

4.2 Main Results

Mathematical reasoning. Table 1 reports the mathematical reasoning results. SRLA matches or exceeds GPT-5.4 accuracy on four of the five benchmarks while using less external API inference. On AIME 2024, SRLA achieves 80.0% accuracy, improving over GPT-5.4 by 3.3 percentage points while reducing the API cost per problem from \$0.0144 to \$0.0014. The gains are larger on AIME 2025 and AIME 2026, where SRLA reaches 63.3% and 80.0%, compared with 53.3% and 60.0% for GPT-5.4, corresponding to improvements of 10.0 and 20.0 percentage points, respectively.

On AMC23, SRLA matches GPT-5.4 at 92.5% accuracy while reducing the external API cost from \$0.0088 to \$0.0040 per problem. MinervaMath shows a different trade-off: GPT-5.4 achieves 76.5%, while SRLA obtains 71.32%. However, SRLA reduces the corresponding API cost from \$0.0047 to \$0.0004 per problem. Overall, the mathematical results show that selective advisor invocation can substantially reduce frontier-model usage while matching or improving direct GPT-5.4 performance on most benchmarks.

Code generation. Table 2 reports the code-generation results. SRLA improves substantially over the unrefined local solver on all six benchmarks and achieves the highest accuracy on HumanEval, MBPP, CodeContests, LeetCode Easy, and LeetCode Hard. On HumanEval, SRLA reaches 98.1%, compared with 94.5% for direct GPT-5.4, while reducing external API cost from 0.18 to 0.01 cents per problem. On MBPP, SRLA obtains 90.0%, outperforming GPT-5.4 at 66.5% and the strongest external-guidance baseline, REFINER, at 72.0%.

The gains extend to the more algorithmic benchmarks. SRLA achieves 47.0% on CodeContests, exceeding GPT-5.4 by 4.6 percentage points, and 76.6% on LeetCode Hard, improving over GPT-5.4 by 7.6 points. On LeetCode Easy, SRLA reaches 83.0%, compared with 82.0% for both GPT-5.4

Table 2: Code-generation performance with Qwen3.5-9B as the local solver. We report accuracy (%) and external API cost in U.S. cents per problem. Methods are grouped by whether they rely solely on local inference, use external feedback, or directly invoke the frontier model. Bold denotes the highest accuracy and, among methods using an external model, the lowest API cost for each benchmark. "–" indicates no external API cost.

	HumanEval		**MBPP**		**CodeContests**		**LeetCode Easy**		**LeetCode Medium**		**LeetCode Hard**	
Method	**Acc.** ↑	**Cost** ↓	**Acc.** ↑	**Cost** ↓	**Acc.** ↑	**Cost** ↓	**Acc.** ↑	**Cost** ↓	**Acc.** ↑	**Cost** ↓	**Acc.** ↑	**Cost** ↓
Local-only baselines												
BASE	59.76	–	43.19	–	4.55	–	69.00	–	43.00	–	18.00	–
Self-Refine	88.40	–	45.90	–	13.60	–	72.00	–	46.00	–	28.00	–
Reflexion	88.40	–	66.10	–	16.70	–	79.00	–	50.00	–	29.00	–
CRITIC	88.40	–	61.10	–	15.20	–	79.00	–	49.00	–	25.00	–
External-feedback baselines												
REFINER	90.24	0.20	72.00	0.18	30.30	0.51	82.00	0.20	60.00	0.27	36.00	0.32
RL-Tango	89.60	0.63	66.90	0.61	10.60	0.98	81.00	0.77	59.00	0.87	31.00	0.89
Frontier-model direct inference												
GPT-5.4	94.50	0.18	66.50	0.06	42.40	0.44	82.00	0.12	**73.00**	0.19	69.00	0.29
Ours												
SRLA	**98.10**	**0.01**	**90.00**	**0.04**	**47.00**	**0.37**	**83.00**	**0.11**	65.00	**0.18**	**76.60**	**0.14**

and REFINER. LeetCode Medium is the only coding benchmark on which direct GPT-5.4 performs better, achieving 73.0% compared with 65.0% for SRLA. These results indicate that selective frontier-model guidance can improve the local solver across a broad range of coding tasks, although its benefit is not uniform across all benchmarks.

4.3 Comparison with Baselines

Table 1 compares SRLA with existing refinement and verification baselines on mathematical reasoning. Across AIME 2024, AIME 2025, and AIME 2026, SRLA achieves 80.0%, 63.3%, and 80.0% accuracy, respectively, outperforming Self-Refine, Reflexion, CRITIC, REFINER, and RL-Tango wherever corresponding results are available. The largest margin appears on AIME 2026, where SRLA reaches 80.0%, compared with 40.0% for the strongest available refinement baseline. On AMC23, SRLA achieves 92.5%, exceeding RL-Tango at 85.0% and REFINER at 77.5%. MinervaMath is the only mathematical benchmark on which a refinement baseline performs better, with RL-Tango reaching 73.2% compared with 71.32% for SRLA.

Table 2 shows a consistent advantage on code generation. SRLA outperforms all five refinement baselines across all six benchmarks. On HumanEval, SRLA achieves 98.1%, compared with 90.24% for REFINER, while on MBPP it reaches 90.0% versus 72.0% for the strongest refinement baseline. The gains are also substantial on the more challenging benchmarks: SRLA obtains 47.0% on CodeContests compared with 30.3% for REFINER, and 76.6% on LeetCode Hard compared with 36.0%. On LeetCode Easy and Medium, SRLA reaches 83.0% and 65.0%, respectively, exceeding the strongest refinement baselines at 82.0% and 60.0%.

Overall, SRLA provides stronger performance than the evaluated refinement baselines across all coding benchmarks and most mathematical reasoning benchmarks. Unlike methods that apply refinement or external guidance more uniformly, SRLA uses the DPO-CGS signal to determine when additional review is needed and invokes the frontier advisor selectively. The results indicate that this combination of learned chunk-level scoring and selective advisor intervention is more effective than the evaluated refinement strategies in most of the tested settings.

4.4 Accuracy–Cost Trade-off

Table 3 summarizes the accuracy change and external API-cost reduction achieved by SRLA relative to direct GPT-5.4 inference. Across all eleven benchmarks, SRLA reduces external API cost, while matching or improving GPT-5.4 accuracy on nine of them. The results therefore highlight that the reduction in frontier-model usage generally does not come at the expense of task performance.

Table 3: Accuracy gains and API-cost savings of SRLA relative to direct GPT-5.4 inference. Δ Acc. reports the change in accuracy in percentage points (pp), where positive values favor SRLA. Cost Saving reports the percentage reduction in external API cost achieved by SRLA; higher values indicate greater savings.

Mathematical Reasoning			**Code Generation**		
Benchmark	**Δ Acc. (pp) ↑**	**Cost Saving (%) ↑**	**Benchmark**	**Δ Acc. (pp) ↑**	**Cost Saving (%) ↑**
AIME'24	+3.3	90.3	HumanEval	+3.6	94.4
AIME'25	+10.0	92.4	MBPP	+23.5	33.3
AIME'26	+20.0	92.6	CodeContests	+4.6	15.9
AMC'23	0.0	54.5	LC-Easy	+1.0	8.3
Minerva	−5.2	91.5	LC-Medium	−8.0	5.3
			LC-Hard	+7.6	51.7

For mathematical reasoning, SRLA achieves particularly favorable trade-offs on the AIME benchmarks. API cost is reduced by more than 90% on AIME 2024, AIME 2025, and AIME 2026, while accuracy improves by 3.3, 10.0, and 20.0 percentage points, respectively. On AMC23, SRLA preserves the same accuracy as direct GPT-5.4 while reducing external API cost by 54.5%. MinervaMath represents the main exception: SRLA reduces API cost by 91.5%, but its accuracy is 5.2 percentage points lower than GPT-5.4. Overall, the mathematical results show that substantial reductions in frontier-model usage can be achieved while maintaining or improving accuracy on four of the five benchmarks. A similar pattern is observed for code generation. SRLA improves accuracy on five of the six benchmarks while reducing external API cost in every case. The largest gain occurs on MBPP, where accuracy increases by 23.5 percentage points alongside a 33.3% cost reduction. SRLA also achieves higher accuracy on HumanEval, CodeContests, LeetCode Easy, and LeetCode Hard, with the largest cost saving observed on HumanEval at 94.4%. LeetCode Medium is the only coding benchmark where SRLA trades accuracy for lower cost, with an 8.0-point decrease in accuracy and a 5.3% reduction in API expenditure.

Overall, these results show that SRLA shifts the majority of reasoning to the local solver while reserving GPT-5.4 for selectively triggered review. This selective allocation consistently reduces external inference cost and, in nine of the eleven evaluation settings, preserves or improves accuracy relative to direct frontier-model inference.

5 Conclusion

We introduced SRLA, a selective collaboration framework that combines local reasoning with targeted frontier-model assistance. SRLA trains the local solver to revise failed reasoning trajectories using localized advisor feedback, then uses DPO-CGS to learn a chunk-level policy–reference scoring signal for gating external review. At inference time, the local solver remains responsible for generating and refining solutions, while GPT-5.4 provides targeted feedback only for trajectories flagged for review. Across eleven mathematical reasoning and code-generation settings, SRLA reduced external API cost on every benchmark while matching or exceeding direct GPT-5.4 accuracy on nine. It also outperformed the evaluated refinement baselines on all coding benchmarks and most mathematical benchmarks.

Beyond cost savings, SRLA suggests a broader design principle: reasoning execution and corrective expertise need not be concentrated in the same model. A small model can remain the primary problem solver while selectively drawing on a stronger model's guidance, rather than delegating the entire task to it. Effective collaboration therefore involves two complementary capabilities: learning how to incorporate external feedback and learning when to seek it. This perspective shifts the focus from the capability of an individual model to the collective capability of a coordinated system. It points toward reasoning architectures that pursue stronger performance not only through larger models, but also through learned coordination—treating frontier expertise as a selectively accessed resource rather than the default engine for every task.

AI USE STATEMENT

Generative AI tools were used in this work to assist with code development and debugging, language editing, and the preparation and refinement of figures, tables, and manuscript text. Generative AI was also an integral component of the proposed framework and experimental pipeline; in particular, GPT-5.4 was used as the frontier advisor and, during training-data construction, for correctness assessment, error localization, corrective feedback, and generation of corrected solutions, as described in the methodology and experimental sections. All AI-assisted code was reviewed and tested by the authors, all reported results were obtained by executing the corresponding experimental pipelines, and all AI-assisted manuscript text, references, and factual claims were manually reviewed for correctness and consistency with the conducted experiments. The authors take full responsibility for the methodology, experimental design, analysis, reported results, claims, citations, and final content of the paper, including material produced with the assistance of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, private or personally identifiable data, or the collection of new user data. All experiments are conducted using publicly available benchmark datasets and language models. The proposed framework is intended to improve the efficiency of language-model reasoning by reducing unnecessary reliance on frontier-model inference. As with other language-model systems, the method may inherit biases, inaccuracies, or unsafe behaviors from the underlying models, and selective advisor invocation does not guarantee correctness. We therefore evaluate the framework only on mathematical reasoning and code-generation benchmarks and do not make claims regarding high-stakes deployment settings. The authors have made efforts to report experimental procedures, costs, and limitations transparently and take responsibility for the integrity of the methodology, experiments, analysis, and reported results.

REPRODUCIBILITY STATEMENT

We provide detailed information to support reproducibility throughout the main paper and appendix. The methodology section specifies the three stages of SRLA, including feedback-conditioned supervised fine-tuning, DPO-based Chunk-level Guided Scoring, and the adaptive advisor-invocation procedure. The appendix documents the training-data construction process, chunk-level preference construction, model and optimization settings, inference hyperparameters, benchmark descriptions, prompts, evaluation procedures, and training-data generation costs. We also report the local solver and advisor models, hardware configuration, dataset sizes, and benchmark-specific evaluation settings used in our experiments. Upon acceptance, we will publicly release the processed training datasets, implementation code, configuration files, training and evaluation scripts required to reproduce the reported results.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks VS Lakshmanan, Qingyun Wu, and Victor Rühle. Bestroute: Adaptive llm routing with test-time optimal compute. *arXiv preprint arXiv:2506.22716*, 2025.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Muhammad Khalifa, Rishabh Agarwal, Lajanugen Logeswaran, Jaekyeom Kim, Hao Peng, Moontae Lee, Honglak Lee, and Lu Wang. Process reward models that think. *arXiv preprint arXiv:2504.16828*, 2025.

Yujin Kim, Euiin Yi, Minu Kim, Se-Young Yun, and Taehyeon Kim. Guiding reasoning in small language models with llm assistance. *arXiv preprint arXiv:2504.09923*, 2025.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Yuliang Liu, Junjie Lu, Zhaoling Chen, Chaofeng Qu, Jason Klein Liu, Chonghan Liu, Zefan Cai, Yunhui Xia, Li Zhao, Jiang Bian, et al. Adaptivestep: Automatically dividing reasoning step through model confidence. *arXiv preprint arXiv:2502.13943*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Math-AI. Amc23. `https://huggingface.co/datasets/math-ai/amc23`, 2023. Hugging Face dataset.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, Mohammed Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In *International Conference on Learning Representations*, volume 2025, pp. 34433–34448, 2025.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. Refiner: Reasoning feedback on intermediate representations. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1100–1126, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for llm reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 60808–60838, 2025.

Chenyang Shao, Xinyang Liu, Yutang Lin, Fengli Xu, and Yong Li. Route-and-reason: Scaling large language model reasoning with reinforced model router. *arXiv preprint arXiv:2506.05901*, 2025.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. *URL https://arxiv. org/abs/2303.11366*, 1:5, 2023.

Haojin Wang, Yike Wang, Shangbin Feng, Hannaneh Hajishirzi, and Yulia Tsvetkov. Mentor-collab: Selective large-to-small inference-time guidance for efficient reasoning. *arXiv preprint arXiv:2602.05307*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*, 2025.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

Wenkai Yang, Jingwen Chen, Yankai Lin, and Ji-Rong Wen. Deepcritic: Deliberate critique with large language models. *arXiv preprint arXiv:2505.00662*, 2025.

Shenghao Ye, Yu Guo, Zhengheng Li, Shuangwu Chen, and Jian Yang. Rubric-guided process reward for stepwise model routing. *arXiv preprint arXiv:2605.29310*, 2026.

Kaiwen Zha, Zhengqi Gao, Maohao Shen, Zhang-Wei Hong, Duane Boning, and Dina Katabi. Rl tango: Reinforcing generator and verifier together for language reasoning. *Advances in Neural Information Processing Systems*, 38:119283–119313, 2026.

Chuang Zhang, Zizhen Zhu, Yihao Wei, Bing Tian, Junyi Liu, Henan Wang, Wang Xavier, and Yaxiao Liu. Confidence-calibrated small-large language model collaboration for cost-efficient reasoning. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4480–4501, 2026.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2024. `https://huggingface.co/datasets/math-ai/aime24`, 2024.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025. `https://huggingface.co/datasets/math-ai/aime25`, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2026. `https://huggingface.co/datasets/math-ai/aime26`, 2026.

Jiaru Zou, Ling Yang, Jingwen Gu, Jiahao Qiu, Ke Shen, Jingrui He, and Mengdi Wang. Reasonflux-prm: Trajectory-aware prms for long chain-of-thought reasoning in llms. *Advances in Neural Information Processing Systems*, 38:26764–26802, 2026.

A Training and Implementation Details

A.1 Training Data Construction

We construct separate training corpora for mathematical reasoning and code generation using a *failure-driven* data construction procedure. Our objective is to train the solver from errors that it actually exhibits during generation, rather than assuming that a problem is intrinsically difficult or treating every sampled response as a negative example. This distinction is important for stochastic language models because repeated generations for the same input can follow different reasoning trajectories and may alternate between correct and incorrect outcomes.

Accordingly, we construct the training corpus at the *trajectory level*. For every generated solver response, we independently determine whether the resulting solution is correct. Correct generations are excluded from the failure pool, whereas incorrect generations are retained as genuine solver failure trajectories. Each retained trajectory is then provided to GPT-5.4 together with the original task. Conditioned on the specific failed trajectory, GPT-5.4 performs three related functions: it identifies the erroneous portion of the solver trajectory, provides targeted corrective feedback explaining how the error should be addressed, and generates a corrected full solution that follows from the solver's existing reasoning context while repairing the identified failure.

The resulting supervision is therefore not merely problem-level teacher-generated data. Each example is anchored to a particular stochastic solver failure. Multiple incorrect generations of the same problem are kept as separate training instances because they may contain different reasoning paths, erroneous intermediate decisions, or failure modes, and consequently may require different diagnoses, feedback, and corrected solutions.

Algorithm 1 summarizes the complete data construction pipeline. Let $\mathcal{D}$ denote the source dataset, S the solver, A the GPT-5.4 advisor, N the target number of unique problems exhibiting an initial solver failure, and R the total number of solver-generation rounds. The procedure first identifies a fixed set of problems on which the solver is observed to fail. It then repeatedly samples the solver on this same problem set and retains only generations that remain incorrect. Finally, GPT-5.4 processes every retained failure trajectory and produces a localized error diagnosis, corrective feedback, and a corrected full solution.

Here, e denotes the erroneous span or step identified within the failed solver trajectory, f denotes the corresponding corrective feedback, and y_{corr} denotes the corrected full solution generated by GPT-5.4 conditioned on both the original task and the solver's failed trajectory. Importantly, y_{corr} is not generated independently from the problem alone. GPT-5.4 observes the reasoning path already taken by the solver and constructs a corrected solution in the context of that trajectory. This preserves the connection between the observed failure, its diagnosis, the corresponding corrective guidance, and the final repaired reasoning process.

Mathematical reasoning. For mathematical reasoning, we use the MATH dataset (Hendrycks et al., 2021) and Qwen3.5-9B as the solver. We first apply Stage 1 of Algorithm 1 and continue generation until we identify $N = 1{,}000$ unique problems for which Qwen3.5-9B produces an incorrect solution. The first incorrect trajectory for each selected problem is retained, producing an initial collection of 1,000 unique problems and 1,000 observed Qwen3.5-9B failure trajectories.

Because identifying a problem through one failed generation does not imply that the solver will fail on every subsequent attempt, we perform three additional independent generations for each of the same 1,000 problems. Thus, for mathematical reasoning we use $R = 4$, corresponding to one initial generation followed by three additional solver runs. This yields at most 4,000 candidate Qwen3.5-9B trajectories.

Each generation is evaluated independently. When Qwen3.5-9B produces a correct solution in a particular run, that trajectory is discarded and does not contribute a negative training example. When the solution is incorrect, the complete generated trajectory is retained. Across the four rounds, 3,247 of the 4,000 candidate trajectories are verified as incorrect and retained, whereas 753 trajectories are excluded because the solver reaches the correct answer. The resulting mathematical failure corpus therefore contains 3,247 incorrect reasoning trajectories associated with 1,000 unique MATH problems.

Algorithm 1 Failure-Driven Training Data Construction

Require: Dataset $\mathcal{D}$, solver S, advisor A, target problem count N, number of generation rounds R
Ensure: Training dataset $\mathcal{D}_{\text{train}}$

$\mathcal{P} \leftarrow \emptyset$
$\mathcal{D}_{\text{fail}} \leftarrow \emptyset$

▷ **Stage 1: Select problems with observed solver failures**

for $x \in \mathcal{D}$ **do**
  $y_{\text{init}} \leftarrow S(x)$
  if y_{init} is incorrect **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{x\}$
    $\mathcal{D}_{\text{fail}} \leftarrow \mathcal{D}_{\text{fail}} \cup \{(x, y_{\text{init}})\}$
  end if
  if $|\mathcal{P}| = N$ **then**
    break
  end if
end for

▷ **Stage 2: Collect additional genuine failure trajectories**

for $r = 2$ **to** R **do**
  for $x \in \mathcal{P}$ **do**
    $y_{\text{init}}^{(r)} \leftarrow S(x)$
    if $y_{\text{init}}^{(r)}$ is incorrect **then**
      $\mathcal{D}_{\text{fail}} \leftarrow \mathcal{D}_{\text{fail}} \cup \{(x, y_{\text{init}}^{(r)})\}$
    end if ▷ Correct generations are discarded
  end for
end for

▷ **Stage 3: Construct trajectory-conditioned supervision**

$\mathcal{D}_{\text{train}} \leftarrow \emptyset$
for $(x, y_{\text{init}}) \in \mathcal{D}_{\text{fail}}$ **do**
  $(e, f, y_{\text{corr}}) \leftarrow A(x, y_{\text{init}})$
  $\mathcal{D}_{\text{train}} \leftarrow \mathcal{D}_{\text{train}} \cup \{(x, y_{\text{init}}, e, f, y_{\text{corr}})\}$
end for
return $\mathcal{D}_{\text{train}}$

The distinction between problems and trajectories is important. The 3,247 training instances do not represent 3,247 unique mathematical problems. Instead, many of the 1,000 problems contribute multiple independently generated failure trajectories. These trajectories are retained separately because the solver may make different mistakes across repeated runs, such as choosing different solution strategies, introducing errors at different reasoning steps, or following different incorrect derivations.

For every retained trajectory, GPT-5.4 receives the original problem x_i together with the complete failed Qwen3.5-9B trajectory $y_{\text{init},i}^{(r)}$. Conditioned on this trajectory, GPT-5.4 identifies an erroneous step or span $e_i^{(r)}$, generates corrective feedback $f_i^{(r)}$, and produces a corrected full solution $y_{\text{corr},i}^{(r)}$. Because all three outputs are conditioned on the specific solver attempt, different failures of the same underlying problem can result in different error diagnoses, different feedback, and different corrected reasoning trajectories.

The resulting mathematical training corpus is therefore

$$\mathcal{D}_{\text{math}} = \left\{ \left(x_i, y_{\text{init},i}^{(r)}, e_i^{(r)}, f_i^{(r)}, y_{\text{corr},i}^{(r)} \right) \right\}, \tag{3}$$

where x_i is the original MATH problem, $y^{(r)}_{\text{init},i}$ is a verified incorrect Qwen3.5-9B trajectory, $e^{(r)}_i$ is the error identified by GPT-5.4, $f^{(r)}_i$ is the corresponding corrective feedback, and $y^{(r)}_{\text{corr},i}$ is the GPT-5.4-generated corrected solution. In total,

$$|\mathcal{D}_{\text{math}}| = 3{,}247. \tag{4}$$

Code generation. For code generation, we follow the same failure-driven construction procedure using the Magicoder dataset (Wei et al., 2023). The code solver generates candidate solutions, and each generated program is evaluated independently according to the task correctness criterion. Only verified incorrect generations are retained, while successful generations are discarded rather than being treated as negative examples.

After repeated generation and correctness filtering, we obtain 3,433 incorrect code-generation trajectories. Each retained trajectory is then processed independently by GPT-5.4. Given the original programming problem and the failed solver trajectory, GPT-5.4 identifies the relevant error, produces targeted corrective feedback, and generates a corrected solution conditioned on the failed attempt. Thus, as in the mathematical setting, the corrected target is explicitly tied to the particular solver trajectory rather than generated independently from the original problem.

The resulting code-generation corpus is

$$\mathcal{D}_{\text{code}} = \left\{ \left(x_j, c^{(r)}_{\text{init},j}, e^{(r)}_j, f^{(r)}_j, c^{(r)}_{\text{corr},j} \right) \right\}, \tag{5}$$

where x_j denotes a programming problem, $c^{(r)}_{\text{init},j}$ denotes a verified incorrect solver trajectory, $e^{(r)}_j$ denotes the identified erroneous component, $f^{(r)}_j$ denotes the advisor feedback, and $c^{(r)}_{\text{corr},j}$ denotes the corresponding corrected solution generated by GPT-5.4. The final code-generation corpus contains trajectory-level training instances.

$$|\mathcal{D}_{\text{code}}| = 3{,}433 \tag{6}$$

trajectory-level training examples. Each example has the general form

$$\boxed{(x,\, y_{\text{init}},\, e,\, f,\, y_{\text{corr}})} \tag{7}$$

and captures five linked components: the original task, an observed solver failure trajectory, the localized error identified within that trajectory, the corresponding corrective feedback, and a corrected full solution generated in the context of the failed trajectory.

This construction provides substantially richer supervision than pairing a problem with an independently generated teacher solution. The model is explicitly exposed to the transformation

$$y_{\text{init}} \;\longrightarrow\; (e, f) \;\longrightarrow\; y_{\text{corr}}, \tag{8}$$

which connects an observed failure to its diagnosis, corrective guidance, and subsequent repair. This structure directly supports Stage 1 feedback-conditioned SFT, where the local solver is trained to use its previous trajectory together with the identified error and advisor feedback to reconstruct the corrected solution.

Prompt construction. We formulate Stage 1 supervised fine-tuning as a *feedback-conditioned trajectory revision task*. Each training example is constructed from an observed solver failure and consists of a structured conditioning context together with a corrected target solution generated by the advisor. In particular, the model is shown the original problem, its previous incorrect reasoning trajectory, a localized diagnosis of the error, and trajectory-specific corrective feedback.

The system message instructs the model to act as an expert mathematical reasoner and informs it that the previous solution contains an error identified by an external advisor. The user message

then provides the original problem x together with three forms of trajectory-level context: (i) the complete previous solver trajectory y_{init}, (ii) the erroneous span e identified within that trajectory, and (iii) the advisor feedback f explaining how the identified error should be corrected.

The supervision target, y_{corr}, is also generated by GPT-5.4. Importantly, GPT-5.4 does not produce this corrected solution independently from the original problem. Instead, it generates y_{corr} conditioned on the solver's actual failed trajectory y_{init}, the identified error e, and the corresponding corrective feedback f. The corrected target therefore represents a trajectory-aware revision of the solver's previous attempt rather than an unrelated teacher-generated solution to the same problem.

The Stage 1 training instance can therefore be written as

$$(x,\ y_{\text{init}},\ e,\ f) \longrightarrow y_{\text{corr}}, \tag{9}$$

where the tuple on the left constitutes the conditioning context and y_{corr} is the advisor-generated corrected reasoning trajectory used as the supervised target.

Concretely, each model input follows the structure

```
Problem: x
Previous solution: y_init
Incorrect step: e
How to fix it: f
Write the corrected full solution.
```

and the assistant target is the corresponding GPT-5.4-generated corrected solution y_{corr}, with the final answer enclosed in `\boxed{}`.

This formulation is intentionally *span-anchored* and *trajectory-conditioned*. The error diagnosis is tied to a concrete segment of the solver's own reasoning rather than expressed as generic problem-level feedback, while the target requires reconstruction of the complete solution rather than replacement of only the identified erroneous span. As a result, the model must learn to interpret the localized diagnosis, incorporate the advisor's correction, preserve valid portions of the previous trajectory when appropriate, and propagate the correction through the remaining reasoning steps to obtain a globally coherent solution.

This design provides supervision for a more specific behavior than ordinary solution imitation. Instead of learning only the mapping from a problem to a correct answer, the model learns the transformation

$$(y_{\text{init}}, e, f) \longrightarrow y_{\text{corr}}, \tag{10}$$

conditioned on the original problem x. Stage 1 therefore trains the solver to perform feedback-guided reasoning repair: given its own previous failed attempt and targeted external guidance, it learns to revise that trajectory into a corrected complete solution.

Model and parameter-efficient fine-tuning. We initialize the Stage 1 solver from `Qwen3.5-9B` and fine-tune it using Low-Rank Adaptation (LoRA), while keeping the pretrained backbone parameters frozen. We use LoRA rank $r = 16$, scaling parameter $\alpha = 32$, and dropout 0.05. The complete LoRA configuration, including the target modules, is reported in Table 4.

We denote the resulting feedback-trained policy by π_{sft}. Stage 1 is designed to teach the solver a *feedback-conditioned revision behavior*, rather than simply improving its ability to independently solve the original problem. For each training example, the model receives the original problem x, its previous incorrect trajectory y_{init}, the erroneous span e identified by the advisor, and the corresponding corrective feedback f. The supervision target is the corrected complete trajectory y_{corr} generated by GPT-5.4 for that specific failed solver attempt.

Formally, we define the conditioning context as

$$z = (x, y_{\text{init}}, e, f), \tag{11}$$

and optimize the standard autoregressive negative log-likelihood of the corrected target:

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{|y_{\text{corr}}|} \log \pi_{\text{sft}}\left(y_{\text{corr},t} \mid z, y_{\text{corr},<t}\right). \tag{12}$$

Table 4: Training configuration for Stage 1 span-anchored feedback-conditioned SFT.

Hyperparameter	**Value**
LoRA rank r	16
LoRA α	32
LoRA dropout	0.05
Training epochs	5
Per-device batch size	1
Gradient accumulation steps	8
Effective batch size per device	8
Learning rate	2×10^{-4}
Optimizer	AdamW
Learning-rate schedule	Cosine decay
Warm-up steps	50
Maximum sequence length	2,048
Numerical precision	bfloat16
Gradient checkpointing	Enabled
Evaluation interval	Every 50 optimization steps
Checkpoint interval	Every 50 optimization steps
Train/validation split	90% / 10%
Split seed	42
Checkpoint selection	Best validation checkpoint

Thus, the model is trained to map an observed failure trajectory and its associated corrective information to a revised full solution. Supervision is applied to the GPT-5.4-generated corrected trajectory, while the original problem, previous solver trajectory, localized error span, and advisor feedback are provided as conditioning context.

Optimization details. We train the LoRA parameters for five epochs using AdamW with an initial learning rate of 2×10^{-4} and a cosine learning-rate schedule with 50 warm-up steps. We use a per-device batch size of 1 and gradient accumulation over 8 steps, yielding an effective batch size of 8 examples per device. The feedback corpus is randomly partitioned into 90% training and 10% validation subsets using a fixed seed of 42. Model evaluation and checkpointing are performed every 50 optimization steps, and the checkpoint with the best validation performance is retained as the final Stage 1 model. All training experiments are conducted on a GPU computing cluster equipped with NVIDIA A100 GPUs, each providing 80 GB of GPU memory, with approximately 1 TB of available storage.

Table 4 summarizes the complete Stage 1 training configuration.

Role in subsequent stages. Stage 1 provides the feedback-following capability required by the subsequent SRLA pipeline. During selective collaboration, the frontier model serves as an advisor rather than directly replacing the local solver. The local model must therefore be able to interpret an identified reasoning error and its associated feedback, relate that information to its existing trajectory, and use it to construct a corrected continuation or revised solution. Training π_{sft} on trajectory-conditioned correction examples provides this capability before preference optimization is introduced.

The Stage 1 checkpoint also serves as the foundation for Stage 2. Specifically, the DPO-CGS policy is initialized from π_{sft}, so preference optimization begins from a model that has already learned how to incorporate localized advisor feedback. At the same time, a frozen copy of π_{sft} is retained as the reference policy. Chunk-level CGS values are subsequently defined using policy–reference log-likelihood differences with respect to this frozen Stage 1 model. Consequently, Stage 1 serves two complementary purposes: it provides the behavioral initialization for feedback-aware preference optimization and establishes the reference distribution used to quantify subsequent policy improvements.

A.2 CHUNK-LEVEL PREFERENCE DATA CONSTRUCTION

Stage 2 constructs chunk-level preference supervision from the trajectory-level corpus produced in Stage 1. Each Stage 1 training instance has the form

$$(x, y_{\text{init}}, e, f, y_{\text{corr}}), \tag{13}$$

where x is the input problem, y_{init} is a verified incorrect solver trajectory, e is the advisor-identified error span, f is the corresponding corrective feedback, and y_{corr} is the GPT-5.4-generated corrected trajectory.

Chunk-level preference construction. For each Stage 1 example, we construct a preference pair around the advisor-identified error. Let c_k^- denote the chunk of y_{init} containing the error span e, and let $y_{\text{init},<k}$ denote the reasoning prefix preceding that chunk. We define the shared conditioning context as

$$z_k = (x, y_{\text{init},<k}). \tag{14}$$

The original erroneous chunk is used as the rejected response,

$$c_k^- = y_{\text{init},k}, \tag{15}$$

while c_k^+ denotes the corrected reasoning corresponding to the same error location. Both responses are evaluated under the same problem and preceding reasoning context, yielding the preference triplet

$$(z_k, c_k^+, c_k^-). \tag{16}$$

Preference instances are retained only when the corrected and failed reasoning can be aligned under a valid shared context. This ensures that the DPO objective compares preferred and rejected reasoning continuations conditioned on the same problem and reasoning prefix:

$$\pi_\theta(c_k^+ \mid z_k) \quad \text{versus} \quad \pi_\theta(c_k^- \mid z_k). \tag{17}$$

The resulting preference signal is therefore tied to the localized reasoning error identified in Stage 1 while controlling for the preceding reasoning context. The complete collection of preference triplets is denoted by $\mathcal{D}_{\text{pref}}$.

Mathematical reasoning. For mathematical reasoning, Stage 2 starts from the 3,247 trajectory-level examples in $\mathcal{D}_{\text{math}}$. Each example contains a verified incorrect Qwen3.5-9B trajectory, an advisor-identified error span, corrective feedback, and the corresponding GPT-5.4-generated corrected trajectory. We identify the chunk containing the annotated error and construct the preferred–rejected pair using the shared context defined above. Examples for which a valid aligned preference pair cannot be established are excluded from the Stage 2 preference corpus.

Code generation. For code generation, we apply the same construction to the 3,433 trajectory-level examples in $\mathcal{D}_{\text{code}}$. Each example contains a verified failed code-generation trajectory, an advisor-identified error span, corrective feedback, and the corresponding corrected trajectory. The erroneous chunk containing the identified failure is used as c_k^-, while the aligned corrected reasoning is used as c_k^+ under the same conditioning context z_k. As in the mathematical setting, examples without a valid shared-context alignment are excluded.

Training data generation cost for mathematical reasoning and code generation. The one-time GPT-5.4 cost for constructing the SRLA training data is summarized in Table 5. GPT-5.4 is used during data construction for correctness judgment, error localization, corrective feedback, and the preference-data preparation procedure. These costs are incurred only once and are therefore amortized across subsequent evaluations and deployments.

The total data-construction cost is \$11.76 for mathematical reasoning and \$13.10 for code generation, for a combined cost of \$24.86. Together with selective advisor invocation at inference time, this limits SRLA's dependence on the frontier model to targeted supervision during training and selectively triggered feedback during deployment.

A.3 DPO-CGS Training and Implementation Details

Preference optimization. We initialize the Stage 2 policy π_θ from the Stage 1 checkpoint π_{sft} and keep a frozen copy of π_{sft} as the reference policy throughout preference optimization. The policy is trained on the chunk-level preference triplets in $\mathcal{D}_{\text{pref}}$ using Direct Preference Optimization (DPO).

For each triplet (z_k, c_k^+, c_k^-), DPO increases the relative likelihood of the preferred chunk c_k^+ over the rejected chunk c_k^- with respect to the frozen Stage 1 reference model. The resulting preference-optimized policy is denoted π_{cgs}.

CGS computation. Chunk-level Guided Scoring (CGS) compares the likelihood assigned to a reasoning chunk by π_{cgs} with that assigned by the frozen reference policy π_{sft}. Both likelihoods are evaluated by teacher forcing over the same reasoning prefix, and the token-level log-likelihood difference is normalized by chunk length.

Using the same π_{sft} as the reference during both DPO training and CGS computation keeps the policy–reference comparison consistent between preference optimization and the adaptive advisor invocation procedure used at inference time.

Table 6 summarizes the Stage 2 configuration.

Table 5: Estimated GPT-5.4 API cost for SRLA training-data generation.

Stage	**Purpose**	**Math**	**Code**
Trajectory collection	Correctness judgment	\$4.87	\$5.15
Trajectory annotation	Error localization and feedback	\$3.25	\$3.45
Preference construction	Chunk-level preference preparation	\$3.64	\$4.50
Total		\$11.76	\$13.10
Combined total		**\$24.86**	

Table 6: Hyperparameters for DPO-CGS training.

Hyperparameter	**Value**
β	0.1
Learning rate	5×10^{-7}
Weight decay	0.01
Epochs	3
Batch size	1
Warmup ratio	0.1
LR schedule	Cosine
Max sequence length	512
LoRA rank (r)	16
LoRA alpha (α)	32
Chunk size (tokens)	50

A.4 Inference Hyperparameters

Table 7 summarizes the inference configurations used for mathematical reasoning and code-generation tasks.

A.5 SRLA Inference Prompt Templates

This section provides the prompt templates used during SRLA inference for mathematical reasoning. The inference pipeline uses three prompt types: (i) initial local solver generation before advisor intervention, (ii) frontier-advisor diagnosis when the CGS gate is activated, and (iii)

Table 7: Inference hyperparameters for SRLA on mathematical reasoning and code-generation tasks.

Hyperparameter	Math	Code
Max generation tokens	20,000	4,096
Max advisor tokens	500	500
Chunk size (tokens)	50	50
CGS gate threshold	0.005	0.002
Max loops	3	3

feedback-conditioned solver revision after advisor intervention. Variables enclosed in braces, such as {problem}, are populated dynamically at inference time.

Initial solver generation. Before any advisor intervention, the local solver generates the solution incrementally, one reasoning step at a time. At each step, the solver is conditioned on the original problem and the reasoning trajectory accumulated so far.

Prompt 1: Initial Solver Generation

```
[SYSTEM]
You are an expert math problem solver.
Solve the problem step by step.

When done, put your final answer in \boxed{}.

[USER]
Problem:
{problem}

Solution so far:
{accumulated}

Write Step {step_num}:
```

Frontier-advisor diagnosis. When the CGS gate is activated, GPT-5.4 receives the original problem, the complete solver-generated solution, and the lowest-scoring reasoning chunk. The complete trajectory remains available to the advisor so that its diagnosis is not restricted to the chunk identified by CGS. The lowest-scoring chunk serves only as an additional localization cue. The advisor identifies the erroneous span and provides concise corrective feedback in a machine-readable JSON format.

Prompt 2: Frontier-Advisor Diagnosis

```
[SYSTEM]
You are an expert math competition advisor.

You will receive:
1. A mathematical problem,
2. A complete solution generated by the solver, and
3. A suspicious reasoning chunk identified by the reliability
    mechanism.

Review the complete solution. The suspicious chunk is provided only as
     a
localization cue and may not contain the actual error.

Your tasks:
1. Identify the incorrect reasoning step in the complete solution.
2. Quote the corresponding erroneous span exactly.
3. Provide concise corrective feedback explaining the error and how it
```

```
   should be corrected.

Respond ONLY with valid JSON:

{
  "highlighted_error": "<exact quote of the erroneous step>",
  "feedback": "<concise explanation of the error and correction>"
}

[USER]
Problem:
{problem}

Suspicious chunk:
{worst_chunk}

Complete solution:
{solution}

Respond ONLY with JSON:
```

Feedback-conditioned solver revision. After the advisor identifies an error, the local solver is conditioned on the original problem, its previous reasoning trajectory, the advisor-highlighted error span, and the corresponding corrective feedback. The solver then performs a new complete solve using this localized guidance, following the self-correction behavior learned during Stage 1.

Prompt 3: Feedback-Conditioned Solver Revision

```
[SYSTEM]
You are an expert math problem solver.
Your previous solution contains an error identified by the advisor.

Use the identified error and corrective feedback to revise your
    reasoning
and produce a corrected solution.

When done, put your final answer in \boxed{}.

[USER]
Problem:
{problem}

Previous solution:
{previous_solution}

Highlighted error:
{highlighted_error}

Corrective feedback:
{feedback}

Write the corrected solution:
```

These prompts correspond directly to the adaptive inference procedure described in Section 3.3. Prompt 1 supports local step-wise reasoning before external intervention. When the CGS gate opens, Prompt 2 is used to obtain a localized diagnosis from GPT-5.4. The returned `highlighted_error` and `feedback` fields are then incorporated into Prompt 3 together with the solver's previous trajectory. The local solver performs the subsequent correction, while the frontier model remains in the advisor role.

B Additional Experimental Setup Details

B.1 Additional Benchmark Details

We evaluate SRLA on a diverse collection of mathematical reasoning and code-generation benchmarks. The benchmarks vary substantially in task difficulty and evaluation format, allowing us to examine the behavior of selective advisor invocation across both symbolic reasoning and executable program synthesis.

Mathematical reasoning. AIME 2024, AIME 2025, and AIME 2026 are problems from the American Invitational Mathematics Examination (AIME) (Zhang & Math-AI, 2024; 2025; 2026). AIME consists of challenging competition-level mathematics problems covering areas such as algebra, geometry, number theory, and combinatorics. Each problem requires a numerical answer, making these benchmarks suitable for evaluating exact mathematical reasoning.

AMC23 (Math-AI, 2023) contains problems from the American Mathematics Competitions. Compared with AIME, AMC problems generally cover a broader range of competition mathematics and provide an additional evaluation setting for measuring mathematical reasoning across different difficulty levels.

MinervaMath (Lewkowycz et al., 2022) is a mathematical reasoning benchmark containing problems that span multiple areas of mathematics. It evaluates a model's ability to produce correct solutions to mathematically diverse problems rather than focusing on a single competition format.

Code generation. HumanEval (Chen et al., 2021) evaluates functional code generation from natural-language function specifications. Generated programs are executed against unit tests, and a solution is considered correct when it passes the corresponding test suite.

MBPP (Austin et al., 2021) contains short programming problems described in natural language together with test cases. The benchmark primarily evaluates the ability of a model to synthesize small programs that satisfy the requested functionality.

CodeContests (Li et al., 2022) contains competitive programming problems that typically require more complex algorithmic reasoning than short function-generation benchmarks. Solutions must infer an appropriate algorithm from the problem statement and produce executable code that satisfies the associated test cases.

LeetCode (Xia et al., 2025) provides algorithmic coding problems organized by difficulty. We evaluate the Easy, Medium, and Hard subsets separately in order to study how SRLA behaves as problem difficulty increases. As with the other code-generation benchmarks, generated solutions are evaluated through execution against the corresponding test cases.

B.2 Additional Baseline Details

Table 8 summarizes the main technical differences between SRLA and the evaluated baselines. BASE performs only local inference, while Self-Refine, Reflexion, and CRITIC improve the local solver through iterative refinement without using an external advisor. REFINER and RL-Tango incorporate external guidance, but invoke it as part of their refinement procedure rather than selectively based on the reliability of the generated trajectory. Direct GPT-5.4 inference instead relies entirely on the frontier model. In contrast, SRLA keeps the local model as the primary solver and selectively invokes the frontier advisor only when the CGS-based gate indicates that external feedback is needed, thereby limiting external API usage.

C Additional Experimental Results

C.1 With Domain-Specific Solvers

To examine whether SRLA generalizes beyond Qwen3.5-9B, we further evaluate the framework with domain-specialized 7B solvers. For mathematical reasoning, we use Qwen2.5-Math-7B-Instruct as the local solver, while for code generation we use Qwen2.5-Coder-7B-Instruct. Tables 9

Property	BASE	Self-Refine	Reflexion	CRITIC	REFINER	RL-Tango	GPT-5.4	SRLA
Solver model	Local	Local	Local	Local	Local	Local	Frontier	Local
Iterative refinement	×	✓	✓	✓	✓	✓	×	✓
External critic/advisor	×	×	×	×	✓	✓	×	✓
Selective invocation	×	×	×	×	×	×	×	✓
Requires critic training	×	×	×	×	✓	×	×	×
External API cost	None	None	None	None	Every call	Every call	Every call	Selective

Table 8: Technical comparison of SRLA against baselines. SRLA is the only method that combines local reasoning with selective frontier advisor invocation driven by a learned confidence gate.

and 10 report the corresponding results. These experiments test whether the selective advisor mechanism remains effective when applied to different local solver backbones and task-specific model families.

Qwen2.5-Math-7B-Instruct. Table 9 evaluates SRLA with Qwen2.5-Math-7B-Instruct as the local solver. SRLA consistently improves over the unrefined solver and all refinement baselines across the five mathematical reasoning benchmarks. On AIME 2024, AIME 2025, and AIME 2026, SRLA reaches 40.00%, 36.67%, and 36.67% accuracy, respectively, compared with at most 20.00%, 23.33%, and 23.33% among the refinement baselines. Similar gains are observed on AMC23 and MinervaMath, where SRLA achieves 65.00% and 58.09%, respectively. Although direct GPT-5.4 remains more accurate in this setting, SRLA substantially improves the local solver while requiring considerably less external API inference.

Qwen2.5-Coder-7B-Instruct. Table 10 reports results with Qwen2.5-Coder-7B-Instruct as the local code-generation solver. SRLA outperforms all evaluated refinement baselines across all six coding benchmarks. The largest gain occurs on MBPP, where SRLA achieves 75.49% accuracy compared with 43.33% for the strongest refinement baseline, an improvement of 32.16 percentage points. SRLA also achieves 95.73% on HumanEval, 28.80% on CodeContests, and 83.00%, 61.00%, and 28.00% on LeetCode Easy, Medium, and Hard, respectively. Compared with direct GPT-5.4, SRLA achieves higher accuracy on HumanEval, MBPP, and LeetCode Easy, while GPT-5.4 remains stronger on CodeContests, LeetCode Medium, and LeetCode Hard. SRLA nevertheless uses equal or lower external API cost across all six benchmarks.

Cross-solver consistency. Together with the Qwen3.5-9B results, these experiments show that SRLA remains effective across different local solver backbones. With Qwen2.5-Math-7B-Instruct, SRLA primarily improves the local solver while remaining below direct GPT-5.4, whereas with Qwen2.5-Coder-7B-Instruct it matches or exceeds GPT-5.4 on several benchmarks. This provides additional evidence that the selective collaboration mechanism is not tied to a single solver architecture.

C.2 Qualitative Examples

Figure 3 illustrates a representative SRLA self-correction example on AIME 2026. The example highlights the complete interaction between the local solver, the CGS-based reliability gate, and the frontier advisor.

In Loop 1, the solver follows a reasonable counting strategy but uses an incorrect recurrence and obtains $\boxed{198}$. Since its CGS score (0.0013) falls below the threshold (0.005), the advisor is invoked and points out that the recurrence overcounts paths by ignoring whether the vertical edge has already been used. In Loop 2, the solver re-solves the problem using the same overall approach while incorporating this feedback, ultimately reaching the correct answer $\boxed{243}$.

This example demonstrates the intended division of labor in SRLA: the local solver remains responsible for constructing and completing the reasoning trajectory, while the frontier model is selectively invoked when the CGS gate detects an unreliable solution. The advisor identifies the specific reasoning failure and provides corrective guidance, after which the local solver re-solves the problem while avoiding the highlighted mistake.

Table 9: Mathematical reasoning performance with Qwen2.5-Math-7B-Instruct as the local solver and GPT-5.4 as the frontier advisor. We report accuracy (%) and external API cost in U.S. cents per problem. Methods are grouped by whether they rely solely on local inference, use external guidance, or directly invoke the frontier model. Bold denotes the highest accuracy and, among methods using an external model, the lowest API cost for each benchmark. "–" indicates no external API cost.

	AIME 2024		AIME 2025		AIME 2026		AMC23		MinervaMath	
Method	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**
Local-only baselines										
BASE	13.33	–	10.00	–	6.70	–	47.50	–	21.00	–
Self-Refine	13.33	–	10.00	–	6.70	–	50.00	–	7.40	–
Reflexion	16.67	–	6.70	–	6.70	–	47.50	–	39.00	–
CRITIC	16.67	–	16.67	–	10.00	–	40.00	–	37.50	–
External-guidance baselines										
REFINER	20.00	0.74	20.00	0.78	23.33	0.85	55.00	0.62	47.05	0.53
RL-Tango	20.00	0.96	23.33	0.91	20.00	0.99	52.50	0.94	47.79	0.67
Frontier-model direct inference										
GPT-5.4	**76.67**	1.44	**53.30**	1.84	**60.00**	1.88	**92.50**	0.88	**76.50**	0.47
Ours										
SRLA	40.00	**0.41**	36.67	**0.41**	36.67	**0.42**	65.00	**0.39**	58.09	**0.35**

Table 10: Code-generation performance with Qwen2.5-Coder-7B-Instruct as the local solver and GPT-5.4 as the frontier advisor. We report accuracy (%) and external API cost in U.S. cents per problem. Methods are grouped by whether they rely solely on local inference, use external guidance, or directly invoke the frontier model. Bold denotes the highest accuracy and, among methods using an external model, the lowest API cost for each benchmark. "–" indicates no external API cost.

	HumanEval		MBPP		CodeContests		LeetCode Easy		LeetCode Medium		LeetCode Hard	
Method	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**	**Acc. ↑**	**Cost ↓**
Local-only baselines												
BASE	86.59	–	6.23	–	9.10	–	74.00	–	52.00	–	22.00	–
Self-Refine	88.41	–	15.95	–	4.55	–	70.00	–	57.00	–	27.00	–
Reflexion	88.41	–	20.23	–	0.00	–	74.00	–	55.00	–	26.00	–
CRITIC	92.07	–	16.73	–	6.06	–	77.00	–	60.00	–	25.00	–
External-guidance baselines												
REFINER	88.41	0.34	43.32	0.07	26.06	0.48	80.00	0.16	60.00	0.22	26.00	0.28
RL-Tango	87.20	0.40	43.33	0.10	24.24	0.52	79.00	0.18	57.00	0.18	26.00	0.32
Frontier-model direct inference												
GPT-5.4	94.50	0.18	66.50	**0.06**	**42.40**	0.44	82.00	**0.12**	**73.00**	0.19	**69.00**	0.29
Ours												
SRLA	**95.73**	**0.13**	**75.49**	**0.06**	28.80	**0.32**	**83.00**	**0.12**	61.00	**0.16**	28.00	**0.22**

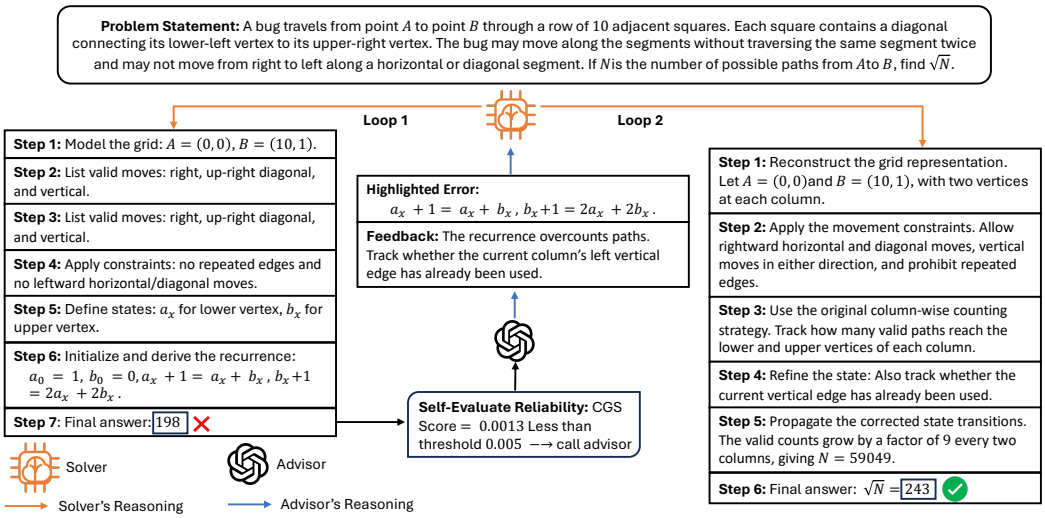

Figure 3: Qualitative example of SRLA self-correction on AIME 2026. In Loop 1, the local solver follows a plausible reasoning trajectory but introduces an incorrect recurrence and produces the wrong answer. Because the CGS score falls below the gating threshold, the advisor is invoked to identify the erroneous step and provide targeted corrective feedback. In Loop 2, the local solver follows the same overall reasoning strategy while incorporating the advisor's feedback at the identified failure point, leading to the correct answer $\boxed{243}$.