

REVES: REvision and VERification–Augmented Training for Test-Time Scaling

Yuanxin Liu^{1*} Ruida Zhou² Xinyan Zhao² Amr Sharaf² Hongzhou Lin²
Arijit Biswas² Mohammad Ghavamzadeh³ Zhaoran Wang¹ Mingyi Hong^{2,4}

¹Northwestern University ²Amazon AGI ³Qualcomm AI Research
⁴University of Minnesota

Abstract

Test-time scaling via sequential revision has emerged as a powerful paradigm for enhancing Large Language Model (LLM) reasoning. However, standard post-training methods primarily optimize single-shot objectives, creating a fundamental misalignment with multi-step inference dynamics. While recent work treats this as multi-turn reinforcement learning (RL), conventional approaches optimize over the multi-step trajectories directly, failing to further exploit the high-quality mistakes in intermediate steps that model can learn from correcting them. We propose a two-stage iterative framework that alternates between online data/prompt augmentation and policy optimization. By converting the intermediate steps (“near-miss” answers) in the successful recovery trajectories into decoupled revision and verification prompts, our approach concentrates training on both effective answer transformation and error identification. This approach enables efficient off-policy data generation and reduces the computational overhead of long-horizon sampling compared to standard multi-turn RL. On LiveCodeBench, using publicly available test cases as feedback, we observe gains of +6.5 points over the RL baseline and +4.0 points over standard multi-turn training. Beyond coding, our approach matches the previously reported SOTA result on circle packing while using the smallest base model (4B) and far fewer rollouts than the much larger evolutionary search systems. Math results under ground-truth verification further confirm improved correction ability. It also generalizes to out-of-distribution constraint-satisfaction puzzles such as `n_queens` and `mini_sudoku`, where correctness is defined entirely by problem constraints. Code is available at <https://github.com/yxliu02/REVES.git>.

1 Introduction

Large language models (LLMs) deployed in challenging settings rarely produce a correct answer on the first attempt; real-world workflows routinely loop back with feedback. A growing family of test-time scaling (TTS) algorithms formalizes this pattern, including sequential revision (Madaan et al., 2023; Shinn et al., 2023), tree search (Inoue et al., 2025), evolutionary refinement (Lee et al., 2025b), and best-of- N with a verifier (Chow et al., 2025). They all share a single primitive: given a problem, a previous attempt, and feedback, produce an improved response.

This raises a basic question: *can we design a post-training algorithm that explicitly improves a model’s test-time sequential revision capability?* We focus on sequential revision (SR) because gains on SR transfer to the rest of the family: every revision-using TTS algorithm eventually calls the policy on revise-from-prior-attempt inputs. SR is also a strong TTS algorithm in its own right, matching or surpassing tree search and evolutionary refinement on planning tasks (Figure 2). Theorem 3.1 shows that increasing the policy’s one-step revision value on SR-induced inputs also raises J_ϕ for every revision-using TTS algorithm that covers SR’s revision inputs, up to small drift terms.

*Work done during an internship at Amazon.

How, then, do we post-train an LLM to optimize the SR objective $J_{\phi_{\text{SR}}}$? Standard post-training, RLHF (Ouyang et al., 2022), RLVR, and GRPO (DeepSeek-AI et al., 2025), optimizes a single-shot expected reward, which is fundamentally misaligned with the multi-step nature of test-time deployment. The natural fix is multi-turn RL on revision rollouts, but trajectory-level credit broadcasts assign path-dependent credit. Take a wrong, wrong, correct rollout: every per-turn log-probability gradient receives the same positive credit, including the wrong intermediates. Per rollout this credit is biased, and only cancels in expectation across many rollouts.

We propose REVES, a two-stage iterative framework that targets the SR objective $J_{\phi_{\text{SR}}}$ directly through its per-state structure. We show that $J_{\phi_{\text{SR}}}$ decomposes exactly into a weighted sum of per-state one-step recovery probabilities along visited rollouts (Lemma 4.1), exposing single-state, single-turn gradients with no horizon credit assignment. REVES is an offline realization of this per-state training signal: each epoch, it runs SR rollouts under the current policy, retains those that succeed within budget, converts the intermediate (“near-miss”) answers into decoupled *revision* and *verification* prompts, and trains with standard single-turn RL on the augmented prompt set; the next epoch refreshes the augmentation. This concentrates training on per-state recovery and avoids both the bias of trajectory broadcasts and the cost of long-horizon online sampling.

Empirically, on LiveCodeBench with public test cases as feedback, REVES achieves +6.5 points over the single-shot RL baseline and +4.0 points over standard multi-turn training. On the circle packing benchmark, a Qwen3-4B base trained with REVES matches the previously reported best results from much larger evolutionary search systems built on Gemini-2.0 Pro/Flash and Qwen3-8B. Math benchmarks (MATH500, AIME24/25) show consistent improvements under both ground-truth and self-confidence stopping, and the trained policies generalize to out-of-distribution constraint-satisfaction puzzles (n_queens, mini_sudoku). Beyond standalone SR, REVES improves every revision-using TTS algorithm we tested (MCTS, AB-MCTS variants, Mind Evolution), in line with the conditional bound in Theorem 3.1.

Our contributions are threefold:

1. **Conceptual.** We formulate test-time scaling as a meta-RL problem and show that improving sequential revision also improves any revision-using TTS algorithm that covers SR’s revision inputs (Theorem 3.1).
2. **Methodological.** We exactly decompose $J_{\phi_{\text{SR}}}$ into per-state one-step recovery probabilities (Lemma 4.1) and propose REVES, a two-stage framework that recasts intermediate near-miss answers as decoupled revision and verification prompts trained with single-turn RL, avoiding the path-dependent credit assignment of multi-turn RL.
3. **Empirical.** REVES delivers consistent test-time gains across math, coding, out-of-distribution puzzles, and circle packing, and improves every revision-using TTS algorithm we evaluated.

2 Related Work

Test-time-aware post-training. A growing line of work shows the importance of aligning the post-training objective with the intended test-time inference strategy. For parallel-sampling families, this includes optimizing pass@ k (Chen et al., 2025; Walder & Karkhanis, 2025; Tang et al., 2025) when deployment uses best-of- N , majority voting (Wang et al., 2023), or learned verifiers (Chow et al., 2025). Our deployment is structurally different: the test-time strategies we care about, including sequential revision, MCTS variants, and Mind Evolution, are all *revision-using* algorithms whose calls take as input a previous attempt together with its feedback. Aligning training with this inference family therefore requires a different objective.

Training sequential revision capability. Closer to our setting, several prior methods explicitly train the policy to revise, including supervised and preference-based approaches (Qu et al., 2024; Xiong et al., 2025), RL-based approaches (Kumar et al., 2024; Jiang et al., 2025;

Lee et al., 2025a), and multi-turn RL methods that incorporate per-turn natural-language or numerical critiques (Jain et al., 2025; Li et al., 2025b;a; Zhang et al., 2026). Our work shares the broader goal of strengthening the model’s revision capability, but differs in framing: we primarily focus on how to align training with the test-time objective of a revision-using inference strategy.

Weakness-driven data synthesis and guided exploration. SwS (Liang et al., 2025) synthesizes new problems targeting the model’s weaknesses using strong external teacher models, and POPE (Qu et al., 2026) uses privileged oracle hints to guide on-policy exploration on hard problems. These methods are orthogonal to ours: they target single-shot pass@1, whereas REVES targets a fundamentally different test-time objective.

3 Test-Time Scaling as a Meta-RL Problem

Setup and two motivating examples. We fix notation used throughout the paper. Denote by \mathcal{X} the distribution over input problems, \mathcal{Y} the answer space, and $r^* : \mathcal{X} \times \mathcal{Y} \rightarrow [0, 1]$ the *ground-truth* reward, where $r^*(x, y) = 1$ indicating a fully correct answer. We focus on the verifiable-reward regime where r^* is computable during both training and at test time, e.g., public test-case execution for code, and constraint satisfaction for puzzles. When r^* is unavailable at test time, a surrogate stopping rule replaces it (Section 4). We model the LLM as a policy π_θ in RL parameterized by θ . At deployment, an LLM rarely operates by a single call to π_θ ; instead a *test-time scaling (TTS) algorithm* ϕ orchestrates up to K policy calls and extracts a final answer $\hat{y} \in \mathcal{Y}$. Two canonical examples bracket the design space: *Best-of-N (BoN)* (Chow et al., 2025) draws N candidates in parallel from $\pi_\theta(\cdot | x)$ and selects \hat{y} via an external scorer; every call uses the same original prompt. *Sequential Revision (SR)* (Madaan et al., 2023) generates an initial response $y_1 \sim \pi_\theta(\cdot | x)$ from prompt x , and for $t = 2, \dots, K$ produces a revision $y_t \sim \pi_\theta(\cdot | x, y_{t-1}, f_{t-1})$ conditioned on the previous attempt y_{t-1} and a feedback signal f_{t-1} (e.g., compiler error messages in code generation), halting at the first correct response. Detailed pseudocode is in Protocol 1 of Appendix A.

General formulation: the TTS-induced decision process. Pairing any TTS algorithm ϕ with π_θ induces a decision process with prompt-state space \mathcal{S} and response space \mathcal{Y} . At each step t , the policy emits a response $y_t \sim \pi_\theta(\cdot | s_t)$, and ϕ constructs the next prompt via a transition kernel $P_\phi(s_{t+1} | s_{1:t}, y_{1:t})$. The initial state is $s_1 = x$. Note that the policy itself remains Markov in the current state (at every step, π_θ conditions only on the local prompt s_t); the history dependence is in the kernel, because ϕ may branch from any earlier response. We focus on the family $\Phi_{\mathcal{R}}$ of TTS algorithms that issue at least one revision call on a prior attempt; this includes Sequential Revision and adaptive tree search such as AB-MCTS (Inoue et al., 2025). For such algorithms a revision-step prompt has the form $s_{t+1} = (x, y_{i_t}, f_{i_t})$, $i_t \in \{1, \dots, t\}$, where the prior step i_t is chosen by ϕ : for SR, $i_t = t$ (always the most recent response), so s_{t+1} depends only on (s_t, y_t) and the induced process is a standard MDP; for tree-search algorithms such as AB-MCTS, i_t may be any earlier response that ϕ ’s search policy decides to branch from, and the kernel is genuinely history-dependent. The procedure runs until a random stopping time $\tau := K \wedge \min\{t : r^*(x, y_t) = 1\}$ and outputs $\hat{y} = y_\tau$; that is, the procedure either *early-stops* once a verified-correct response is produced, or *exhausts the budget* K . Running π_θ inside ϕ yields the TTS-induced objective

$$J_\phi(\theta) = \mathbb{E}_{x \sim \mathcal{X}, (s_{1:\tau}, y_{1:\tau}) \sim (\pi_\theta, \phi)} [r^*(x, y_\tau)],$$

to be contrasted with the standard single-shot objective

$$J_{\text{OneShot}}(\theta) = \mathbb{E}_{x \sim \mathcal{X}, y \sim \pi_\theta(\cdot | x)} [r^*(x, y)],$$

which is what RLHF (Ouyang et al., 2022), RLVR, and GRPO (DeepSeek-AI et al., 2025) optimize. Optimizing J_ϕ rather than J_{OneShot} is a *meta-RL* problem (Figure 1): the policy must adapt across the distribution of multi-step contexts that ϕ generates, not just answer the original question. Our goal is to improve J_ϕ for every TTS algorithm $\phi \in \Phi_{\mathcal{R}}$.

This raises an important question: *does there exist a training target $J_{\phi_{\text{train}}}$ whose optimization provably improves J_ϕ for every $\phi \in \Phi_{\mathcal{R}}$?* If so, we can train against one algorithm and have

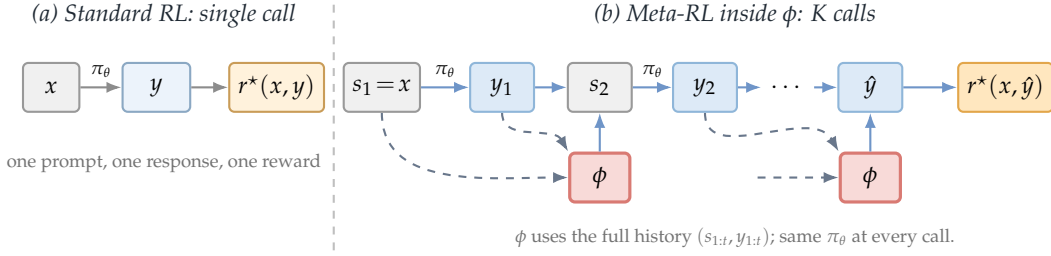


Figure 1: **(a)** Standard RL: policy π_θ maps problem x to response y with reward $r^*(x, y)$. **(b)** A TTS algorithm ϕ invokes the same π_θ over up to K steps, each prompt s_t built from the prior history (dashed arrows), and extracts a final response $\hat{y} = y_\tau$ at stopping time $\tau \leq K$; π_θ must therefore behave well across the whole context distribution induced by ϕ .

the gains transfer to every other deployment-time choice in $\Phi_{\mathcal{R}}$, side-stepping the cost of running tree search or external verifiers during training. The answer is affirmative. The key fact is that every algorithm in $\Phi_{\mathcal{R}}$ at some point invokes π_θ to revise a previous attempt, so raising the revision value on a training target’s inputs improves every member of $\Phi_{\mathcal{R}}$ that covers those inputs.

Theorem 3.1 (Sequential-revision recovery transfers to revision-using TTS). *Let π_0 be a baseline policy and π_1 an updated policy. For $\phi \in \Phi_{\mathcal{R}}$, policy π , and revision-call input z , let*

$$V_\pi(z) := \mathbb{P}_{y \sim \pi(\cdot|z)}(r^*(x, y) = 1), \quad \rho_\pi^\phi(z) := \mathbb{E}_{(\phi, \pi)} \left[\sum_{t=1}^{\tau} \mathbf{1}\{z_t = z\} \right]$$

denote the one-step recovery probability and the expected visit count to z . Assume:

- (C1) Coverage of SR by ϕ at π_0 : there exists $C_\phi \geq 1$ such that $\rho_{\pi_0}^\phi(z) \geq \rho_{\pi_0}^{\phi_{\text{SR}}}(z) / C_\phi$ for every z .
- (C2) Monotone recovery improvement: $V_{\pi_1}(z) \geq V_{\pi_0}(z)$ for every $z \in \text{supp}(\rho_{\pi_0}^\phi)$.

Then, we have the following guarantee:

$$J_\phi(\pi_1) - J_\phi(\pi_0) \geq \frac{1}{C_\phi} [J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0)] - \|\rho_{\pi_1}^\phi - \rho_{\pi_0}^\phi\|_1 - \frac{1}{C_\phi} \|\rho_{\pi_1}^{\phi_{\text{SR}}} - \rho_{\pi_0}^{\phi_{\text{SR}}}\|_1.$$

See Appendix C.2 for the proof.

Our training target $J_{\phi_{\text{SR}}}$. The theorem licenses any $\phi_{\text{train}} \in \Phi_{\mathcal{R}}$ as a training target, with the transfer guarantee holding for every deployment-time ϕ that covers it. We pick ϕ_{SR} and optimize

$$J_{\phi_{\text{SR}}}(\theta) := \mathbb{E}_{x \sim \mathcal{X}, y_{1:\tau} \sim (\pi_\theta, \phi_{\text{SR}})} [r^*(x, y_\tau)]$$

for two reasons. First, ϕ_{SR} is itself a strong TTS algorithm. Figure 2 shows that on TravelPlanner (Xie et al., 2024), ϕ_{SR} matches or surpasses Mind Evolution (Lee et al., 2025b) and AB-MCTS (Inoue et al., 2025) (left), with the Markov-revision form $\pi_\theta(\cdot | x, y_{\text{prev}}, f)$ on the best cost-accuracy frontier (right), while being the simplest to implement. Second, optimizing ϕ_{SR} transfers to the rest of $\Phi_{\mathcal{R}}$: by Theorem 3.1, improving one-step recovery on SR’s revision inputs raises J_ϕ for every revision-using ϕ that covers them.

Optimizing $J_{\phi_{\text{SR}}}$ is not optimizing single-shot pass@1. A natural question is whether standard single-shot RLVR training already addresses $J_{\phi_{\text{SR}}}$. It does not: two policies can be *indistinguishable* under J_{OneShot} on every problem in \mathcal{X} yet differ substantially under $J_{\phi_{\text{SR}}}$.

Theorem 3.2 (Objective mismatch). *For any revision length $K \geq 2$, there exist policies π_1, π_2 , $\Delta(K) > 0$, and a problem distribution \mathcal{X} such that $J_{\text{OneShot}}(\pi_1) = J_{\text{OneShot}}(\pi_2)$, while $J_{\phi_{\text{SR}}}(\pi_2) - J_{\phi_{\text{SR}}}(\pi_1) \geq \Delta(K)$.*

The implication is direct: J_{OneShot} and $J_{\phi_{\text{SR}}}$ are different objectives, and policies with identical single-shot performance can have substantially different sequential-revision performance.

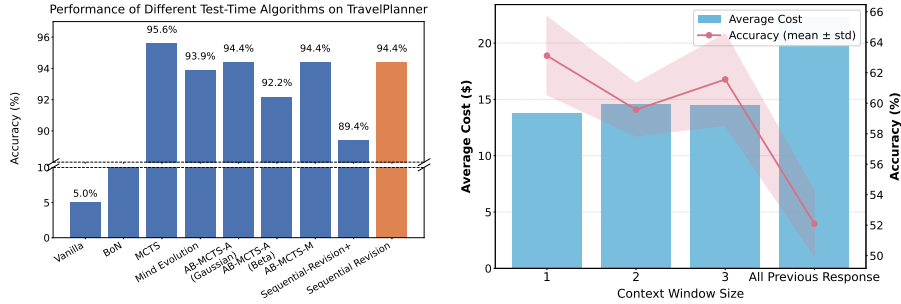


Figure 2: DeepSeek-V3 on TravelPlanner. **Left:** sequential revision vs. Mind Evolution / MCTS / AB-MCTS. **Right:** cost-accuracy as the revision context window covers the k most recent responses, $k \in \{1, 2, 3\}$, or the full history.

Our problem reduces to a single concrete question: how do we post-train π_θ so that $J_{\phi_{SR}}$ goes up? The next section presents our algorithm.

Takeaway: we adopt the sequential-revision objective $J_{\phi_{SR}}$ as our training target.

Improving test-time performance is not the same as optimizing single-shot pass@1 (Theorem 3.2). Among test-time scaling algorithms, sequential revision is both empirically strong and the one whose gains transfer to every other revision-using algorithm (Theorem 3.1), which makes it the right objective to optimize.

4 REVES: From Path-Level Credit to Recovery-Level Supervision

Section 3 established that improving $J_{\phi_{SR}}$ is our focus. This section turns that target into an algorithm. We start by writing $J_{\phi_{SR}}$ exactly as a sum of per-state one-step recovery probabilities along visited trajectories; this hazard decomposition makes the natural training signal explicit and reads off the algorithm directly from the figure.

Lemma 4.1. Let $\zeta = (z_1, y_1, \dots, z_\tau, y_\tau) \sim (\pi_\theta, \phi_{SR})$ be a stopped SR trajectory under π_θ : at each step t the model samples $y_t \sim \pi_\theta(\cdot | z_t)$ and stops at the first verified correct response or at budget K , with $z_t = (x, y_{t-1}, f_{t-1})$ the revision state before generating y_t . Let $V_\pi(z_t) := \mathbb{P}_{y' \sim \pi_\theta(\cdot | z_t)}(r^*(x, y') = 1)$ denote the one-step recovery probability at z_t , where y' is a fresh sample (not the trajectory’s own y_t), and let $\rho_\theta(z) := \mathbb{E}_{x \sim \mathcal{X}, \zeta \sim (\pi_\theta, \phi_{SR})} \left[\sum_{t=1}^\tau \mathbf{1}\{z_t = z\} \right]$ denote the expected number of times state z is visited along a SR rollout. Then

$$\begin{aligned}
 J_{\phi_{SR}}(\theta) &= \sum_{t=1}^K \mathbb{E}_{x, \zeta} [\mathbf{1}\{\tau \geq t\} V_\pi(z_t)] = \mathbb{E}_{x \sim \mathcal{X}, \zeta \sim (\pi_\theta, \phi_{SR})} \left[\sum_{t=1}^\tau V_\pi(z_t) \right] \\
 &= \sum_z \rho_\theta(z) \mathbb{E}_{y' \sim \pi_\theta(\cdot | z)} [r^*(x, y')]. \tag{1}
 \end{aligned}$$

See Appendix C.3 for the proof. Equation (1) reads the algorithm off Figure 3 (b). The horizontal SR trajectory provides the outer samples $z \sim \rho_\theta$, and at each visited state $z = (x, y_{t-1}, f_{t-1})$ the branches drawn from $\pi_\theta(\cdot | z)$ are online rollouts that estimate and optimize the per-state factor $V_\pi(z) = \mathbb{E}_{y' \sim \pi_\theta(\cdot | z)} [r^*(x, y')]$. Since $J_{\phi_{SR}}$ is a nonnegative-weighted sum of these factors, raising V_π at any visited z directly raises $J_{\phi_{SR}}$, without horizon credit assignment.

Comparison with multi-turn RL. Multi-turn RL provides an unbiased policy-gradient estimator for the trajectory-level objective. However, its per-turn learning signal is highly indirect. Consider a wrong–wrong–correct rollout: the gradients for all three turns are weighted by the same positive terminal advantage, including the two turns that produced incorrect intermediate responses. This is valid for optimizing terminal success of the whole

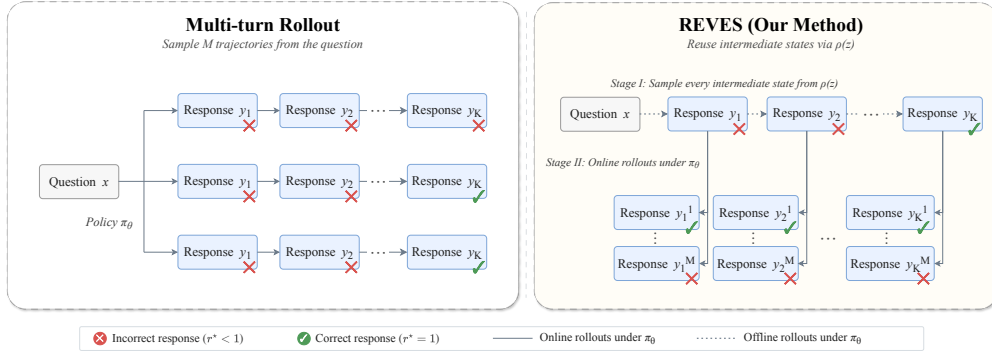


Figure 3: Mapping Eq. (1) onto the figure. **(a)** Multi-turn RL training rollouts under the current π_θ . **(b)** The horizontal hatched chain is a retained sequential-revision trajectory; it determines the visit-count weight $\rho_\theta(z)$ at each intermediate state $z = (x, y_{t-1}, f_{t-1})$. The branches at each visited z are fresh samples from $\pi_\theta(\cdot | z)$; their verifier outcomes provide the per-state estimate of $V_\pi(z) = \mathbb{E}_{y' \sim \pi_\theta(\cdot | z)} [r^*(x, y')]$.

trajectory, but it yields a high-variance and coarse signal for improving the local revision behavior at each intermediate state.

Our method instead reuses such incorrect intermediate responses as revision states. Given a state $z = (x, y_{\text{prev}}, f_{\text{prev}})$, we draw fresh samples from $\pi(\cdot | z)$ and estimate the local one-step recovery probability $V_\pi(z) = \Pr_{y \sim \pi(\cdot | z)} [r^*(x, y) = 1]$. Thus, the intermediate wrong response is not treated as an action to imitate; it is treated as a state from which the model should learn to recover. This converts a successful long-horizon trajectory into several short-horizon revision problems, providing a sharper training signal for local recovery while relying on the current policy to discover useful intermediate states.

From recovery target to algorithm. An ideal on-policy implementation of Lemma 4.1 draws $z \sim \rho_\theta$ by running SR and runs single-turn RL at each z . Three design choices turn this into the algorithm we use.

(1) *Verification prompts alongside revision data.* At test time the verifier is typically unavailable. We therefore include both *revision prompts* and *verification prompts* (asking the policy to judge whether an intermediate response is correct) in the augmented data, so the trained policy can self-stop at deployment. The split of gains across the two prompt types is in Table 5.

(2) *Offline generation.* An on-policy estimator of $\nabla J_{\phi_{\text{SR}}}$ would need fresh SR rollouts every gradient step, which is intrinsically serial and prohibitively slow. We instead generate visited states once per epoch in Stage I, reuse them through Stage II, and refresh at the next epoch.

(3) *Successful-trajectory filtering.* For a far-from-optimal base policy, online samples at states from failed rollouts rarely produce a verifier-accepted recovery and waste tokens. We restrict Stage I to states on trajectories that succeed within budget K , where at least one recovery target exists by construction; the filter loosens as π_θ improves.

Two-stage realization. Stage I (Data augmentation). For each input x , run SR under the current π_θ : $y_1 \sim \pi_\theta(\cdot | x)$ and $y_t \sim \pi_\theta(\cdot | x, y_{t-1}, \text{Feedback}(y_{t-1}))$ for $t = 2, \dots, K$, terminating at the first correct response or after K attempts. Discard trajectories that fail or are already correct at y_1 . From each retained trajectory (x, y_1, \dots, y_ℓ) with $r^*(x, y_\ell) = 1$, build (i) *revision data* $\{\text{Revision Prompt}(x, y_i)\}_{i=1}^\ell$, where each prompt asks the model to revise the previous response, and (ii) *verification data* $\{\text{Verification Prompt}(x, y_i)\}_{i=1}^\ell$, where each prompt asks the model to judge whether y_i is correct. Templates are in Appendix A. **Stage II (Single-turn RL).** Train π_θ on the union of the original RL prompts and the prompts

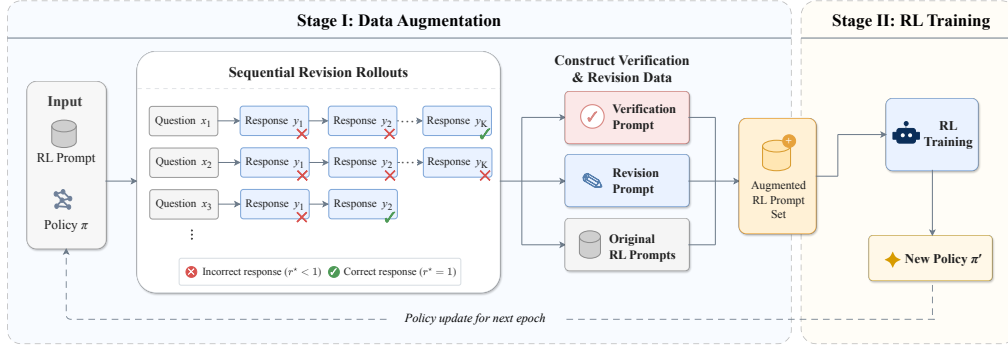


Figure 4: Overview of the proposed RL-based data augmentation framework. In Stage I, the current policy performs sequential revision; only trajectories that succeed within budget K are retained, and their intermediate states yield revision and verification prompts that are merged with the original RL prompt set. In Stage II, the policy is updated by single-turn RL on the augmented set, and the new policy regenerates the augmentation in the next epoch.

produced in Stage I. The updated policy serves as the rollout policy for the next epoch’s Stage I. Figure 4 shows the loop.

Efficiency. Our algorithm is also more efficient than standard multi-turn training. First, augmented samples are generated in an *off-policy* manner. With a well-designed infrastructure, data augmentation can run in parallel with RL training, so the model continues training on the original RL prompts while new augmented data are generated asynchronously, avoiding any interruption to the main training loop. Second, training itself is single-turn. Each gradient step processes one (prompt, response) pair as in ordinary RLVR, with no per-step sequential rollout in the inner loop, which is the dominant cost of multi-turn RL with budget K . Appendix F.2 reports the wall-clock comparison.

Test-time stopping rule. At test time the oracle stopping time τ is almost always unavailable, so we adopt a surrogate stopping time $\hat{\tau}$ that replaces r^* with a domain-specific signal.

For mathematical reasoning tasks, we leverage the *Tail Confidence* metric of Fu et al. (2025) as the surrogate signal. At each revision step t , the token-level confidence at position i of y_t is $C_i = -\frac{1}{k} \sum_{j=1}^k \log P_i(j)$, where $P_i(j)$ is the probability of the j -th most likely token (a larger C_i indicates a more concentrated, more confident distribution). The response-level score c_t averages C_i over the tail positions $i \in T_{\text{tail}}$ of y_t . The stopping time is $\hat{\tau} := K \wedge \min\{t \geq 3 : c_t / \sum_{j=1}^t c_j > c\}$ for a predefined threshold c . If $\hat{\tau} = K$ (budget exhausted without trigger), we output $\hat{y} = y_{t^*}$ with $t^* = \arg \max_{1 \leq t \leq K} c_t$, breaking ties uniformly at random when several responses share the same confidence. Selecting the most confident response across all K revisions instead of just y_K improves robustness under small budgets by mitigating noise in confidence estimates at the stopping step.

For coding tasks, we use publicly available test cases as a surrogate reward $\tilde{r} : \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$, with $\tilde{r}(x, y) = \mathbf{1}(y \text{ passes all public test cases})$. The stopping time is $\hat{\tau} := K \wedge \min\{t : \tilde{r}(x, y_t) = 1\}$, and we output $\hat{y} = y_K$ when $\hat{\tau} = K$.

5 Experimental Results

Models and data. We train Qwen2.5-7B, Qwen2.5-3B (Yang et al., 2024), Qwen3-4B (non-thinking) (Yang et al., 2025) and DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI et al., 2025) directly with RL. Training data is sampled from Skywork/Skywork-OR1-RL-Data (7298 math + 1015 coding = 8313 examples). We evaluate on MATH500 and AIME 24/25 for math;

LiveCodeBench (Jain et al., 2024) and CodeContest (Li et al., 2022) for coding; and n_queens / mini_sudoku from ReasoningGym (Stojanovski et al., 2025) for out-of-distribution puzzles.

Baselines and implementation. We compare against **RL** (single-response RL, no revision), **MultiTurn** (multi-turn rollouts without explicit self-verification), and **PAG** (Jiang et al., 2025) (multi-turn with self-verification via a designed reward). Full details in Appendix E.

5.1 Main Results

Consistent improvements on coding tasks. Table 1 demonstrates the benefits of REVES on coding benchmarks. Under test-time surrogate stopping criteria, where publicly available test cases are used to determine when to stop and execution results are provided as feedback for subsequent revisions (see Appendix A for an example), REVES consistently improves performance across different test-time budgets, demonstrating robustness beyond symbolic rewards. Additional results on DeepSeek-R1-Distill-7B (DeepSeek-AI et al., 2025) are provided in Appendix G.

Table 1: Sequential-revision test-time scaling performance on coding benchmarks (LiveCodeBench (LCB), CodeContest). The best result in each column is shown in bold, and the second-best is underlined. TC- B is sequential-revision (SR) test-time scaling with budget B , using execution-based test cases as the verifier and feeding the resulting feedback into the next revision step.

Model	LCB (Aug 24 – Jan 25)			LCB (Jan 25 – May 25)		CodeContest		
	OneShot	TC-32	TC-4	OneShot	TC-32	OneShot	TC-32	TC-4
Qwen-2.5-7B	4.8	6.1	6.1	6.0	18.9	0.4	5.7	5.9
Qwen-2.5-7B-RL	18.0	23.0	20.8	20.0	24.2	7.1	11.1	7.5
Qwen-2.5-7B-Multi-turn	<u>19.4</u>	25.5	21.3	20.7	<u>27.5</u>	<u>6.9</u>	14.3	8.1
Qwen-2.5-7B-PAG	19.5	<u>25.7</u>	<u>23.2</u>	19.8	26.4	6.5	<u>15.0</u>	<u>9.1</u>
Qwen-2.5-7B-REVES	18.9	29.5	23.8	<u>20.5</u>	30.0	7.1	16.6	11.7
Qwen-2.5-3B	5.9	15.8	7.2	7.9	18.3	1.6	6.3	2.4
Qwen-2.5-3B-RL	9.2	12.0	<u>11.5</u>	14.1	17.8	3.0	4.2	3.6
Qwen-2.5-3B-Multi-turn	9.9	<u>15.7</u>	12.1	13.2	<u>19.2</u>	<u>3.2</u>	<u>7.9</u>	4.4
Qwen-2.5-3B-PAG	8.1	14.7	11.2	<u>13.4</u>	18.5	3.6	5.7	<u>4.9</u>
Qwen-2.5-3B-REVES	<u>9.6</u>	17.7	11.0	14.3	21.6	2.2	8.5	5.5
Qwen3-4B (base)	22.6	27.8	26.8	24.7	24.7	6.1	10.9	9.7
Qwen3-4B-RL	<u>32.2</u>	41.7	42.5	<u>29.1</u>	35.4	<u>16.1</u>	30.9	<u>30.9</u>
Qwen3-4B-Multi-turn	29.2	<u>49.5</u>	<u>44.9</u>	25.3	40.9	10.6	<u>34.9</u>	34.2
Qwen3-4B-PAG	29.8	43.6	40.4	26.7	37.1	13.6	32.4	25.8
Qwen3-4B-REVES	32.7	50.9	46.3	29.7	42.0	16.4	37.9	<u>30.9</u>

Improved sequential revision capability for mathematical reasoning. Table 3 shows that REVES substantially improves sequential revision performance across all evaluated math benchmarks. Under oracle stopping (the Oracle-32 and Oracle-4 columns, where the rollout halts as soon as a generated answer matches the ground truth, isolating the model’s ability to revise an incorrect solution into a correct one), our method achieves consistently large gains. These improvements persist under the SelfConf-4 column, the practical test-time stopping rule based on the model’s own confidence, indicating that the learned capability transfers beyond the oracle setting.

Matching state-of-the-art on circle packing. We further evaluate REVES on the circle packing benchmark, where the goal is to pack n unit-area circles into a unit square to maximize the sum of their radii. We directly optimize Eq. (1) on this task. With a Qwen3-4B, our trained policy matches the previously reported best results from much larger evolutionary search systems built on Gemini-2.0 Pro/Flash and Qwen3-8B (Table 2). Full implementation details are in Appendix H.

Table 2: Circle packing benchmark (sum of radii at $n=26$, \uparrow). REVES on Qwen3-4B matches the best previously reported result, reaching the optimal 2.635983 with the *smallest* base model and fewer rollouts.

Method	Model	Sum of radii, $n=26$ (\uparrow)
AlphaEvolve (Novikov et al., 2025)	Gemini-2.0 Pro + Flash	2.635862
AlphaEvolve V2 (Georgiev et al., 2025)	Gemini-2.0 Pro + Flash	2.635983
ShinkaEvolve (Lange et al., 2025)	Ensemble	2.635982
ThetaEvolve (Wang et al., 2025)	R1-Qwen3-8B	2.635983
TTT-Discover (Yuksekgonul et al., 2026)	Qwen3-8B	2.635983
Ours (REVES)	Qwen3-4B	2.635983

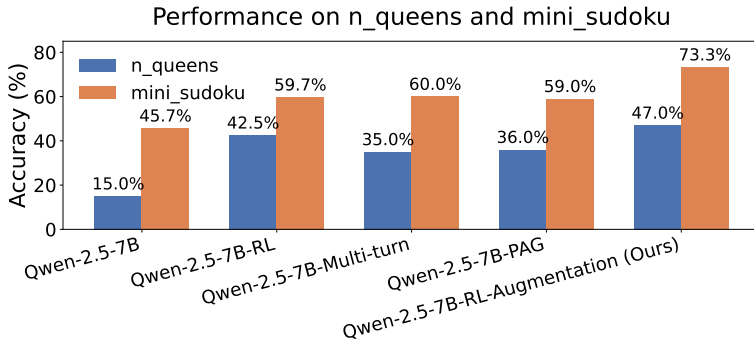


Figure 5: Out-of-distribution puzzle benchmarks (n_queens and mini_sudoku) under verifier-based stopping. REVES generalizes beyond standard reasoning benchmarks despite using checkpoints trained only on math and code.

Table 3: Sequential-revision test-time scaling performance on math benchmarks (AIME24/25, MATH500). **1-shot** is OneShot; **O-B** (Oracle) and **SC-B** (SelfConf) are sequential-revision (SR) test-time scaling with budget B , using the ground-truth answer as the oracle stopping rule and the model’s own confidence as a surrogate stopping rule, respectively. \dagger denotes evaluation using the authors’ original codebase and protocol.

Model	AIME 24				AIME 25				MATH 500			
	1-shot	O-32	O-4	SC-4	1-shot	O-32	O-4	SC-4	1-shot	O-32	O-4	SC-4
Qwen-2.5-7B	2.0	14.0	4.8	2.3	0.3	6.3	1.3	0.6	23.8	50.1	29.0	26.1
Qwen-2.5-7B-RL	17.9	33.5	21.4	18.7	10.4	22.9	13.4	12.5	76.1	85.9	79.4	78.6
Qwen-2.5-7B-Multi-turn	14.6	30.3	20.0	13.8 †	7.2	25.3	11.8	9.8 †	75.9	87.1	80.1	78.0†
Qwen-2.5-7B-PAG	15.3	31.1	19.6	13.0 †	7.5	26.8	12.8	9.4 †	76.1	86.7	80.8	77.2†
Qwen-2.5-7B-REVES	14.9	45.7	25.9	19.6	9.3	40.5	22.3	15.1	76.8	94.7	85.6	76.2
Qwen-2.5-3B	1.2	12.9	3.4	1.9	0.5	7.9	1.3	0.6	30.5	67.0	39.3	38.4
Qwen-2.5-3B-RL	6.4	16.9	8.1	7.4	1.8	9.0	3.9	2.4	64.9	80.5	69.6	65.6
Qwen-2.5-3B-Multi-turn	7.6	20.5	12.0	8.0†	2.6	12.7	5.2	2.3 †	65.3	81.2	72.7	67.0†
Qwen-2.5-3B-PAG	6.3	14.8	7.9	5.3 †	3.4	11.4	4.7	4.1†	64.0	78.8	69.9	65.6 †
Qwen-2.5-3B-REVES	6.2	33.7	14.7	8.2	2.8	24.4	7.2	4.2	64.5	89.9	77.8	66.0

Generalization to out-of-distribution puzzle tasks. As illustrated in Figure 5, we evaluate puzzle tasks using checkpoints trained only on math and code data, without any puzzle-specific training. Despite this distribution shift, REVES substantially improves performance on out-of-distribution puzzle benchmarks, indicating strong cross-task generalization.

$J_{\phi_{SR}}$ -trained policies also improve other TTS algorithms. Table 4 shows that models trained with REVES consistently improve the performance of a wide range of test-time

algorithms that rely on sequential revision, including MCTS (Inoue et al., 2025), AB-MCTS variants (Inoue et al., 2025) and Mind Evolution (Lee et al., 2025b). This suggests that sequential revision capability serves as a fundamental backbone that raises the performance ceiling of diverse test-time inference strategies.

Table 4: Comparison of model performance on different test-time strategies. Best results in each column are in bold; second-best are underlined. **1-shot** is OneShot; **MindEvo** is Mind Evolution; **ABM-A(β)**, **ABM-A(G)**, and **ABM-M** are the AB-MCTS-A (Beta), AB-MCTS-A (Gaussian), and AB-MCTS-M variants.

Model	1-shot	SeqRev	BoN	MCTS	MindEvo	ABM-A(β)	ABM-A(G)	ABM-M
Qwen-2.5-7B-RL	18.0	23.0	27.8	22.7	25.7	26.0	27.3	26.2
Qwen-2.5-7B-Multi-turn	<u>19.4</u>	25.5	28.7	26.1	27.0	28.6	<u>29.0</u>	27.8
Qwen-2.5-7B-PAG	19.5	<u>25.7</u>	30.4	<u>26.4</u>	<u>27.7</u>	<u>29.7</u>	28.1	<u>28.7</u>
Qwen-2.5-7B-REVES	18.9	29.5	<u>29.3</u>	28.9	29.0	31.0	30.0	30.4

Takeaway: REVES delivers consistent, transferable test-time gains.

REVES improves sequential revision on code and math, matches state-of-the-art on frontier open problems (circle packing) with a far smaller model, and generalizes to out-of-distribution puzzles despite training only on math and code. As predicted by Theorem 3.1, the gains further transfer to other revision-using TTS algorithms.

5.2 Ablation Study

The role of revision and verification prompts. As shown in Table 5, we investigate the roles of revision and verification prompts in sequential revision training. Qwen-2.5-7B-RevisionOnly augments RL-generated data using only revision prompts, while Qwen-2.5-7B-VerificationOnly augments data using only verification prompts. Qwen-2.5-7B-REVES combines both prompts during data augmentation. Results under Oracle-4 indicate that revision prompts primarily improve *sequential revision capability*, whereas verification prompts do not directly enhance revision but instead train the model to assess solution validity.

To isolate the effect of verification prompts, we evaluate confidence calibration using Area Under the Receiver Operating Characteristic curve (AUROC) on AIME25. For each problem, we sample multiple model responses, label them as correct or incorrect based on the final answer, and compute the probability that a randomly sampled correct response receives a higher TailConfidence score than a randomly sampled incorrect one. The final AUROC is obtained by averaging over problems with sufficient correct samples. Under this evaluation, Qwen-2.5-7B-REVES achieves a higher AUROC (74.1%) than RevisionOnly (72.1%), demonstrating that verification prompts improve confidence calibration. Thus, combining revision and verification prompts yields consistent gains by jointly enhancing revision capability and confidence estimation.

Necessity of continual data augmentation. As shown in Figure 6 (right), augmenting data only at the initial epoch leads to substantially worse performance than continuous augmentation. This is because once the model stops making earlier mistakes, stale augmented data no longer provides informative learning signals, whereas regenerating data each epoch keeps the supervision aligned with the model’s current failure modes.

Effect of augmentation budget. We study the effect of the augmentation budget on training performance. As shown in Figure 6 (left), increasing the augmentation budget consistently improves performance, indicating that our method exhibits strong scaling behavior with respect to the augmentation budget. At the same time, allowing a larger augmentation budget K naturally incurs higher computational cost, reflecting a trade-off between performance and compute in data augmentation.

Table 5: Effect of revision and verification prompts on AIME25. **Left:** 1-shot (OneShot), O-4 (Oracle-4), and SC-4 (SelfConf-4) accuracy across the four configurations: *RL* (no augmentation), *VerificationOnly* (verification prompts only), *RevisionOnly* (revision prompts only), and *REVES* (both, our method). **Right:** TailConfidence distributions for RevisionOnly and REVES; REVES produces sharper, better-separated correct/incorrect distributions, indicating improved confidence calibration.

Model	1-shot	O-4	SC-4
Qwen-2.5-7B-RL	10.4	13.4	12.5
Qwen-2.5-7B-VerificationOnly	6.3	10.3	7.1
Qwen-2.5-7B-RevisionOnly	7.9	16.5	8.3
Qwen-2.5-7B-REVES	9.5	21.2	15.8

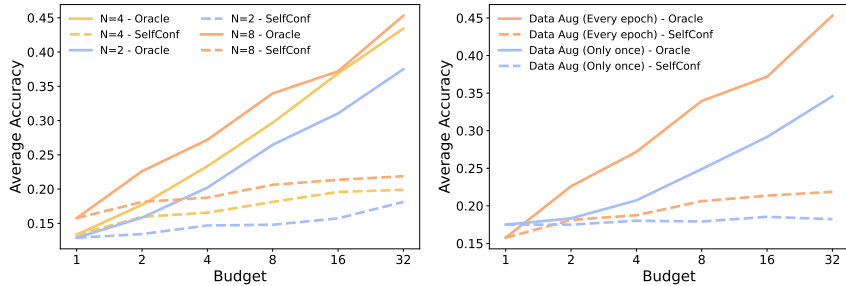
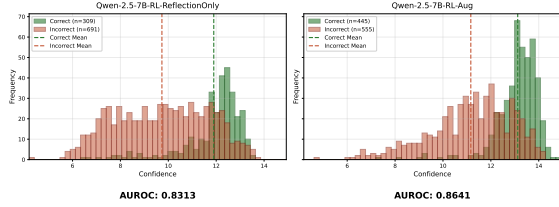


Figure 6: Ablations on data augmentation (AIME24). *Left:* effect of augmentation budget K (the maximum trajectory length allowed during augmentation); *right:* continual versus one-shot augmentation.

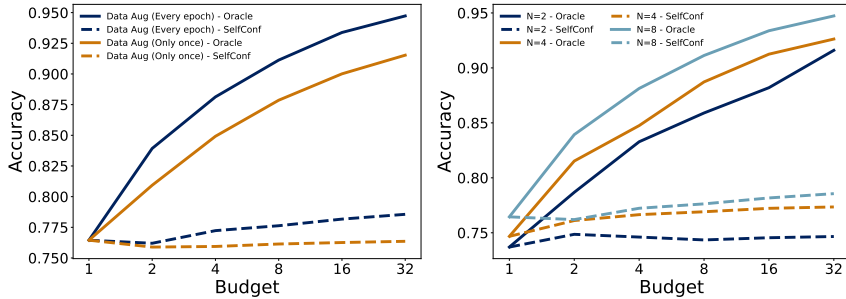


Figure 7: Ablations on data augmentation (MATH500). *Left:* effect of augmentation frequency (continual versus one-shot); *right:* effect of allowed revision budget K .

6 Conclusion

We show that standard single-shot RL objectives are misaligned with test-time sequential revision, and propose **REVES**, which pairs RL with continual, mistake-driven data augmentation so the model learns both to revise incorrect solutions and to identify errors. Augmentation must be continual, since static data becomes uninformative once mistakes disappear. Across math, coding, and out-of-distribution tasks, REVES improves test-time sequential-revision performance over the single-shot RL and multi-turn baselines under fixed budgets, and, as predicted by our transfer analysis, these gains further carry over to other revision-using test-time algorithms. On the circle packing benchmark, a Qwen3-4B trained with REVES matches the previously reported state-of-the-art achieved by much larger evolutionary search systems, using the smallest base model and far fewer rollouts. d

References

- Zhipeng Chen, Xiaobo Qin, Youbin Wu, Yue Ling, Qinghao Ye, Wayne Xin Zhao, and Guang Shi. Pass@k training for adaptively balancing exploration and exploitation of large reasoning models, 2025. URL <https://arxiv.org/abs/2508.10751>.
- Yinlam Chow, Guy Tenenholz, Izzeddin Gur, Vincent Zhuang, Bo Dai, Sridhar Thiagarajan, Craig Boutilier, Rishabh Agarwal, Aviral Kumar, and Aleksandra Faust. Inference-aware fine-tuning for best-of-n sampling in large language models, 2025. URL <https://arxiv.org/abs/2412.15287>.
- DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanbiao Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.
- Yichao Fu, Xuwei Wang, Yuandong Tian, and Jiawei Zhao. Deep think with confidence, 2025. URL <https://arxiv.org/abs/2508.15260>.
- Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.
- Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. Wider or deeper? scaling llm inference-time compute with adaptive branching tree search, 2025. URL <https://arxiv.org/abs/2503.04412>.
- Arnav Kumar Jain, Gonzalo Gonzalez-Pumariiega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards, 2025. URL <https://arxiv.org/abs/2502.20380>.
- 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, 2024. URL <https://arxiv.org/abs/2403.07974>.

- Yuhua Jiang, Yuwen Xiong, Yufeng Yuan, Chao Xin, Wenyuan Xu, Yu Yue, Qianchuan Zhao, and Lin Yan. Pag: Multi-turn reinforced llm self-correction with policy as generative verifier, 2025. URL <https://arxiv.org/abs/2506.10406>.
- Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning, 2024. URL <https://arxiv.org/abs/2409.12917>.
- Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.
- Hyunseok Lee, Seunghyuk Oh, Jaehyung Kim, Jinwoo Shin, and Jihoon Tack. Revise: Learning to refine at test-time via intrinsic self-verification, 2025a. URL <https://arxiv.org/abs/2502.14565>.
- Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. Evolving deeper llm thinking, 2025b. URL <https://arxiv.org/abs/2501.09891>.
- Ang Li, Yifei Wang, Zhihang Yuan, Stefanie Jegelka, and Yisen Wang. Lanpo: Bootstrapping language and numerical feedback for reinforcement learning in llms, 2025a. URL <https://arxiv.org/abs/2510.16552>.
- Yanyang Li, Michael Lyu, and Liwei Wang. Learning to reason from feedback at test-time, 2025b. URL <https://arxiv.org/abs/2502.15771>.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097, 2022. doi: 10.1126/science.abq1158. URL <https://www.science.org/doi/abs/10.1126/science.abq1158>.
- Xiao Liang, Zhong-Zhi Li, Yeyun Gong, Yang Wang, Hengyuan Zhang, Yelong Shen, Ying Nian Wu, and Weizhu Chen. Sws: Self-aware weakness-driven problem synthesis in reinforcement learning for llm reasoning, 2025. URL <https://arxiv.org/abs/2506.08989>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegraffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL <https://arxiv.org/abs/2303.17651>.
- Alexander Novikov, Ngan Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL <https://arxiv.org/abs/2203.02155>.
- Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve, 2024. URL <https://arxiv.org/abs/2407.18219>.

- Yuxiao Qu, Amrith Setlur, Virginia Smith, Ruslan Salakhutdinov, and Aviral Kumar. Pope: Learning to reason on hard problems via privileged on-policy exploration, 2026. URL <https://arxiv.org/abs/2601.18779>.
- 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>.
- Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards, 2025. URL <https://arxiv.org/abs/2505.24760>.
- Yunhao Tang, Kunhao Zheng, Gabriel Synnaeve, and Rémi Munos. Optimizing language models for inference time objectives using reinforcement learning, 2025. URL <https://arxiv.org/abs/2503.19595>.
- Christian Walder and Deep Karkhanis. Pass@k policy optimization: Solving harder reinforcement learning problems, 2025. URL <https://arxiv.org/abs/2505.15201>.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL <https://arxiv.org/abs/2203.11171>.
- Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.
- Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents, 2024. URL <https://arxiv.org/abs/2402.01622>.
- Wei Xiong, Chengshuai Shi, Jiaming Shen, Aviv Rosenberg, Zhen Qin, Daniele Calandriello, Misha Khalman, Rishabh Joshi, Bilal Piot, Mohammad Saleh, Chi Jin, Tong Zhang, and Tianqi Liu. Building math agents with multi-turn iterative preference learning, 2025. URL <https://arxiv.org/abs/2409.02392>.
- An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024. URL <https://arxiv.org/abs/2407.10671>.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.
- Mert Yuksekogonul, Daniel Kocejka, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time, 2026. URL <https://arxiv.org/abs/2601.16175>.

A Sequential Revision Protocol

In this section, we elaborate on the test-time protocol described in Protocol 1 and illustrate its practical usage with several concrete examples, including a mathematical reasoning task in Appendix A.1 and a coding task in Appendix A.2.

Protocol 1 Sequential Revision Protocol

```

1: Input: revision budget  $K$ , question  $x$ , model  $\pi$ 
2: Sample initial response  $y_1 \sim \pi(\cdot | x)$ 
3: for  $i = 2$  to  $K$  do
4:   Generate revised response  $y_i \sim \pi(\cdot | x, y_{i-1}, \text{Feedback}(y_{i-1}))$ 
5:   if  $r^*(x, y_i) = 1$  then
6:     # Early stopping
7:     Set  $\hat{y} = y_i$ 
8:     break for loop
9:   end if
10: end for
11: if No early stopping then
12:    $\hat{y} = y_K$ 
13: end if

```

A.1 Example Math Task

In mathematical tasks, the feedback consists solely of the sentence “The response is incorrect.” (marked in red in the figure), without offering further diagnostic information.

Input (Turn 1)

Solve the following math problem. Show brief reasoning and provide the final answer.
 Problem: A 90° rotation around $-4 - 5i$ in the clockwise direction is applied to 3 (as a complex number). What is the resulting complex number?

LLM Output (Turn 1)

To solve this problem, we need to perform a 90° clockwise rotation around the complex number $-4 - 5i$.

First, it’s useful to understand that a 90° clockwise rotation of a complex number z around another complex number a is equivalent to the following transformation:

$$z' = a + (a - z) \cdot i$$

However, since we are rotating around $-4 - 5i$, let’s denote this center of rotation as $a = -4 - 5i$. Our complex number $z = 3$ needs to be rotated.

1. Find the difference between z and a :

$$z - a = 3 - (-4 - 5i) = 3 + 4 + 5i = 7 + 5i$$

2. Perform the 90° clockwise rotation by multiplying this difference by i and adding it back to a :

$$z' = a + (z - a) \cdot i = (-4 - 5i) + (7 + 5i) \cdot i$$

Next, simplify the expression:

$$(7 + 5i) \cdot i = 7i + 5i^2$$

Since $i^2 = -1$:

$$7i + 5(-1) = 7i - 5$$

Therefore, the final answer is $\boxed{-9 + 2i}$.

Input (Turn 2)

Problem: A 90° rotation around $-4 - 5i$ in the clockwise direction is applied to 3 (as a complex number). What is the resulting complex number?

Previous Attempt: To solve this problem, we need to perform a 90° clockwise rotation around the complex number $-4 - 5i$.

First, it's useful to understand that a 90° clockwise rotation of a complex number z around another complex number a is equivalent to the following transformation:

$$z' = a + (a - z) \cdot i$$

However, since we are rotating around $-4 - 5i$, let's denote this center of rotation as $a = -4 - 5i$. Our complex number $z = 3$ needs to be rotated.

1. Find the difference between z and a :

$$z - a = 3 - (-4 - 5i) = 3 + 4 + 5i = 7 + 5i$$

2. Perform the 90° clockwise rotation by multiplying this difference by i and adding it back to a :

$$z' = a + (z - a) \cdot i = (-4 - 5i) + (7 + 5i) \cdot i$$

Next, simplify the expression:

$$(7 + 5i) \cdot i = 7i + 5i^2$$

Since $i^2 = -1$:

$$7i + 5(-1) = 7i - 5$$

...

So, the resulting complex number after the rotation is $-9 + 2i$.

Therefore, the final answer is $\boxed{-9 + 2i}$.

Verifiable Feedback: **The response is incorrect.**

Now produce a corrected solution. Keep reasoning concise.

LLM Output (Turn 2)

To correctly rotate a complex number 3 around $-4 - 5i$ by 90° clockwise, we need to use the appropriate transformation for rotations in the complex plane.

Given a complex number $z = 3$, and the center of rotation $a = -4 - 5i$, a 90° clockwise rotation of z around a can be represented as:

$$z' = a + (a - z) \cdot (-i)$$

Let's go through the steps in detail:

1. Calculate $a - z$:

$$a - z = (-4 - 5i) - 3 = -4 - 5i - 3 = -7 - 5i$$

2. Multiply by $-i$:

$$(a - z) \cdot (-i) = (-7 - 5i) \cdot (-i) = (-7)(-i) + (-5i)(-i) = 7i + 5i^2$$

Since $i^2 = -1$:

$$7i + 5(-1) = 7i - 5$$

3. Add this result back to a :

$$z' = a + (7i - 5) = (-4 - 5i) + (-5 + 7i)$$

$$z' = -4 - 5i - 5 + 7i$$

Combine the real and imaginary parts:

$$z' = (-4 - 5) + (-5i + 7i) = -9 + 2i$$

Thus, after performing the 90° clockwise rotation, the resulting complex number is $\boxed{-9 + 2i}$.

Input (Turn 3)

Problem: A 90° rotation around $-4 - 5i$ in the clockwise direction is applied to 3 (as a complex number). What is the resulting complex number?

Previous Attempt: To correctly rotate a complex number 3 around $-4 - 5i$ by 90° clockwise, we need to use the appropriate transformation for rotations in the complex plane.

Given a complex number $z = 3$, and the center of rotation $a = -4 - 5i$, a 90° clockwise rotation of z around a can be represented as:

$$z' = a + (a - z) \cdot (-i)$$

Let's go through the steps in detail:

1. Calculate $a - z$:

$$a - z = (-4 - 5i) - 3 = -4 - 5i - 3 = -7 - 5i$$

2. Multiply by $-i$:

$$(a - z) \cdot (-i) = (-7 - 5i) \cdot (-i) = (-7)(-i) + (-5i)(-i) = 7i + 5i^2$$

Since $i^2 = -1$:

$$7i + 5(-1) = 7i - 5$$

3. Add this result back to a :

$$z' = a + (7i - 5) = (-4 - 5i) + (-5 + 7i)$$

$$z' = -4 - 5i - 5 + 7i$$

Combine the real and imaginary parts:

$$z' = (-4 - 5) + (-5i + 7i) = -9 + 2i$$

Thus, after performing the 90° clockwise rotation, the resulting complex number is $-9 + 2i$.

Verifiable Feedback: **The response is incorrect.**

Now produce a corrected solution. Keep reasoning concise.

LLM Output (Turn 3)

To find the result after a 90° clockwise rotation around $-4 - 5i$, we can use the following transformation steps. We start by considering a complex number $z = 3$ and the center of rotation $a = -4 - 5i$:

1. Move z to the origin relative to a :

$$z - a = 3 - (-4 - 5i) = 3 + 4 + 5i = 7 + 5i$$

2. Rotate by 90° clockwise: A 90° clockwise rotation of a complex number w is equivalent to multiplying w by $-i$. So, we need to multiply $7 + 5i$ by $-i$:

$$(7 + 5i) \times (-i) = 7 \times (-i) + 5i \times (-i) = -7i - 5i^2$$

Since $i^2 = -1$:

$$-7i - 5(-1) = -7i + 5 = 5 - 7i$$

3. Translate back by adding a : Adding a to the resultant, we get:

$$z' = (5 - 7i) + (-4 - 5i) = 5 - 4 - 7i - 5i = 1 - 12i$$

Thus, the resulting complex number after the 90° clockwise rotation is $1 - 12i$.

A.2 Example Coding Task

At test time for coding tasks, we follow an iterative execution-based evaluation protocol. The model first generates a candidate program, which is then executed against the publicly available test cases provided by the problem. If the program fails any test, the resulting error messages or failing test feedback are collected and appended to the context for the next generation step, allowing the model to revise its solution based on execution feedback. Note

that passing the public test cases does not guarantee correctness: a solution is considered correct only if it passes all test cases, including both public and private ones.

Input (Turn 1)

You are an expert Python programmer. You will be given a problem specification and must produce a correct Python program that passes the tests.

Question: You are given a permutation $P = (P_1, P_2, \dots, P_N)$ of $(1, 2, \dots, N)$. You want to satisfy $P_i = i$ for all $i = 1, 2, \dots, N$ by performing the following operation zero or more times:

- Choose an integer k such that $1 \leq k \leq N$. If $k \geq 2$, sort the 1-st through $(k - 1)$ -th terms of P in ascending order. Then, if $k \leq N - 1$, sort the $(k + 1)$ -th through N -th terms of P in ascending order.

It can be proved that under the constraints of this problem, it is possible to satisfy $P_i = i$ for all $i = 1, 2, \dots, N$ with a finite number of operations for any P . Find the minimum number of operations required. You have T test cases to solve.

Input ...

Output

Print T lines. The i -th line should contain the answer for the i -th test case.

Constraints ...

Sample Input 1

- 3
- 5
- ...

Sample Output 1

- 1
- ...

Format: Read inputs from stdin and write the answer to stdout. Enclose your final code in the following box.

```
# YOUR CODE HERE
```

Answer: Provide *only* the final Python code.

LLM Output (Turn 1)

To solve this problem, we aim to determine the minimum number of operations required to transform a given permutation P into the identity permutation $(1, 2, \dots, N)$. The key insight is that each operation can be viewed as a global reordering step that partially resolves disorder in the permutation. Intuitively, such operations progressively eliminate inconsistencies between positions and values.

...

```
def min_operations(N, P):
    count = 0
    for i in range(N):
        if P[i] != i + 1:
            count += 1
    return count

...

for _ in range(T):
    N = int(input())
    P = list(map(int, input().split()))
    print(min_operations(N, P))
```

Input (Turn 2)

You are an expert Python programmer. Given a problem specification, produce a correct Python program that passes all tests.

Question: Given a permutation $P = (P_1, \dots, P_N)$ of $(1, \dots, N)$, the goal is to transform P into the identity permutation ...

Input / Output. ...

Previous Codes

code_v1

```
# Your task is to find the minimum number of operations required to sort the
permutation P.

def min_operations(N, P):
    operations = 0
    for i in range(N):
        while P[i] != i + 1:
            ...
    N = case[0]
    P = case[1:]
    print(min_operations(N, P))
```

Issues with code_v1

For the input

```
3
5
2 1 3 5 4
3
1 2 3
7
3 2 1 7 5 6 4
```

the expected output is

```
1
0
2
```

but code_v1 results in a runtime error. The failure is primarily due to incorrect handling of index-based updates and repeated linear searches ($P.index$), which lead to inconsistent state updates and excessive runtime overhead.

LLM Output (Turn 2)

To solve this problem, we aim to determine the minimum number of operations required to transform a given permutation P into the identity permutation $(1, 2, \dots, N)$. The key insight is that each operation can be viewed as a global reordering step that partially resolves disorder in the permutation. Intuitively, such operations progressively eliminate inconsistencies between positions and values.

...

```

def min_operations(N, P):
    count = 0
    for i in range(N):
        if P[i] != i + 1:
            count += 1
    return count

...

for _ in range(T):
    N = int(input())
    P = list(map(int, input().split()))
    print(min_operations(N, P))

```

B Limitation and Future Direction

Limitation. REVES requires a ground-truth verifier during training. Math, coding, puzzle, and circle-packing benchmarks all admit deterministic verifiers, so REVES applies directly. For tasks without such verifiers (free-form QA, creative writing, RLHF-style preference learning), REVES does not directly extend; the verification prompts trained alongside revision data can be repurposed for self-verification at test time, but their reliability depends on the policy’s own calibration.

Future direction. The verifier requirement points to a natural class of problems where REVES is especially well suited: scientific tasks whose objectives admit a clear, automated reward. Examples include solving partial differential equations (PDEs), where residual norms or analytic checks serve as verifiers; symbolic regression, where fit and complexity supply a numeric score; theorem proving, where a proof checker provides binary feedback; and discovery-style optimization tasks like circle packing (Appendix H), where the verifier is the geometric solver itself. In all of these settings, the cost of producing the verifier signal is offset by an essentially unlimited supply of training problems and a precise notion of correctness, both of which line up well with REVES’s reliance on per-state one-step recovery as the gradient signal. Scaling REVES to such scientific-discovery domains is a natural next step.

C Proofs

In this section we collect the proofs of the theoretical results stated in the main text. We first prove the objective-mismatch result (Theorem 3.2), showing that optimizing single-shot pass@1 does not optimize the sequential-revision objective $J_{\phi_{\text{SR}}}$. We then prove the transfer guarantee (Theorem 3.1), which establishes that improving SR recovery also improves every revision-using TTS algorithm that covers SR’s revision inputs. Finally, we prove the hazard decomposition (Lemma 4.1) that exposes the per-state one-step recovery structure underlying REVES.

C.1 Proof of Theorem 3.2

Theorem C.1 (Objective mismatch). *For any revision length $K \geq 2$, there exist policies π_1, π_2 and a problem distribution \mathcal{X} such that*

$$J_{\text{OneShot}}(\pi_1) = J_{\text{OneShot}}(\pi_2), \quad J_{\phi_{\text{SR}}}(\pi_2) - J_{\phi_{\text{SR}}}(\pi_1) \geq \Delta(K),$$

where $\Delta(K) > 0$.

Proof. Construction. Take \mathcal{X} supported on a single problem x with $N \geq 2$ candidate answers $\{y^{(1)}, \dots, y^{(N)}\}$ and ground-truth y^* drawn uniformly from this set; feedback is

binary correctness. Define two history-independent policies, each sampling from a fixed distribution at every step:

$$\pi_1(\cdot | \cdot) = \delta_{y^{(1)}}, \quad \pi_2(\cdot | \cdot) = \text{Uniform}\{y^{(1)}, \dots, y^{(N)}\},$$

with π_2 's draws independent across steps.

One-shot. Since y^* is uniform,

$$J_{\text{OneShot}}(\pi_1) = \mathbb{P}(y^* = y^{(1)}) = \frac{1}{N}, \quad J_{\text{OneShot}}(\pi_2) = \sum_{i=1}^N \frac{1}{N} \cdot \frac{1}{N} = \frac{1}{N},$$

so $J_{\text{OneShot}}(\pi_1) = J_{\text{OneShot}}(\pi_2)$.

Sequential revision (budget K). π_1 outputs $y^{(1)}$ at every step and therefore succeeds iff $y^* = y^{(1)}$, giving $J_{\phi_{\text{SR}}}(\pi_1) = 1/N$. For π_2 , the K i.i.d. uniform draws are independent of y^* , so the probability that none equals y^* is $(1 - 1/N)^K$, giving $J_{\phi_{\text{SR}}}(\pi_2) = 1 - (1 - 1/N)^K$.

Gap. For $K \geq 2$, monotonicity of $K \mapsto (1 - 1/N)^K$ on $K \geq 1$ gives $(1 - 1/N)^K \leq (1 - 1/N)^2 = 1 - 2/N + 1/N^2$, hence

$$\Delta(K) := J_{\phi_{\text{SR}}}(\pi_2) - J_{\phi_{\text{SR}}}(\pi_1) = 1 - (1 - \frac{1}{N})^K - \frac{1}{N} \geq \frac{N-1}{N^2} > 0. \quad \square$$

C.2 Proof of Theorem 3.1

The argument uses the recovery decomposition $J_\phi(\pi) = \sum_z \rho_\pi^\phi(z) V_\pi(z)$, which holds for any first-success transition strategy ϕ paired with any policy π . The proof of Lemma 4.1 (Appendix C.3) establishes this for $\phi = \phi_{\text{SR}}$; the same three steps – (i) writing the success event at step t as $\{\tau \geq t, y_t \in G_x\}$, (ii) Fubini against $\sum_{t=1}^K \mathbf{1}\{\tau \geq t\} = \tau$, (iii) commuting the visit-count sum with the outer expectation – depend only on the first-success stopping rule and the Markov property of π , so they generalize verbatim from $\rho_\pi^{\phi_{\text{SR}}}$ to ρ_π^ϕ for any $\phi \in \Phi_{\mathcal{R}}$ that early-stops.

Proof of Theorem 3.1. By the recovery decomposition applied to π_1 and π_0 ,

$$J_\phi(\pi_1) - J_\phi(\pi_0) = \sum_z \rho_{\pi_1}^\phi(z) V_{\pi_1}(z) - \sum_z \rho_{\pi_0}^\phi(z) V_{\pi_0}(z).$$

Add and subtract $\sum_z \rho_{\pi_0}^\phi(z) V_{\pi_1}(z)$ and group:

$$J_\phi(\pi_1) - J_\phi(\pi_0) = \underbrace{\sum_z \rho_{\pi_0}^\phi(z) (V_{\pi_1}(z) - V_{\pi_0}(z))}_{(*)} + \underbrace{\sum_z (\rho_{\pi_1}^\phi(z) - \rho_{\pi_0}^\phi(z)) V_{\pi_1}(z)}_{(+)} \quad (2)$$

Bounding (+). Since $V_{\pi_1}(z) \in [0, 1]$,

$$(+)\geq -\sum_z |\rho_{\pi_1}^\phi(z) - \rho_{\pi_0}^\phi(z)| = -\|\rho_{\pi_1}^\phi - \rho_{\pi_0}^\phi\|_1. \quad (3)$$

Bounding (*). We claim the pointwise inequality

$$\rho_{\pi_0}^\phi(z) (V_{\pi_1}(z) - V_{\pi_0}(z)) \geq \frac{1}{C_\phi} \rho_{\pi_0}^{\phi_{\text{SR}}}(z) (V_{\pi_1}(z) - V_{\pi_0}(z)) \quad \text{for every } z. \quad (4)$$

Case 1: $z \in \text{supp}(\rho_{\pi_0}^\phi)$. By (C2), $V_{\pi_1}(z) - V_{\pi_0}(z) \geq 0$. By (C1), $\rho_{\pi_0}^\phi(z) \geq C_\phi^{-1} \rho_{\pi_0}^{\phi_{\text{SR}}}(z) \geq 0$. Multiplying both sides of (C1) by the nonnegative quantity $V_{\pi_1}(z) - V_{\pi_0}(z)$ preserves the inequality and gives (4).

Case 2: $z \notin \text{supp}(\rho_{\pi_0}^\phi)$. Then $\rho_{\pi_0}^\phi(z) = 0$, and (C1) forces $0 \geq C_\phi^{-1} \rho_{\pi_0}^{\phi_{\text{SR}}}(z)$. Combined with $\rho_{\pi_0}^{\phi_{\text{SR}}}(z) \geq 0$, this gives $\rho_{\pi_0}^{\phi_{\text{SR}}}(z) = 0$, so both sides of (4) vanish.

Summing (4) over z ,

$$(\star) \geq \frac{1}{C_\phi} \sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(z) (V_{\pi_1}(z) - V_{\pi_0}(z)). \quad (5)$$

Converting (5) to $J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0)$. Apply the recovery decomposition to $\phi = \phi_{\text{SR}}$ at π_1 and π_0 , then add and subtract $\sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(z) V_{\pi_1}(z)$:

$$J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0) = \sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(z) (V_{\pi_1}(z) - V_{\pi_0}(z)) + \sum_z (\rho_{\pi_1}^{\phi_{\text{SR}}}(z) - \rho_{\pi_0}^{\phi_{\text{SR}}}(z)) V_{\pi_1}(z).$$

Rearranging and using $V_{\pi_1}(z) \in [0, 1]$,

$$\sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(z) (V_{\pi_1}(z) - V_{\pi_0}(z)) \geq [J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0)] - \|\rho_{\pi_1}^{\phi_{\text{SR}}} - \rho_{\pi_0}^{\phi_{\text{SR}}}\|_1. \quad (6)$$

Combine. Chaining (6) into (5) and adding (3),

$$J_\phi(\pi_1) - J_\phi(\pi_0) \geq \frac{1}{C_\phi} [J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0)] - \|\rho_{\pi_1}^\phi - \rho_{\pi_0}^\phi\|_1 - \frac{1}{C_\phi} \|\rho_{\pi_1}^{\phi_{\text{SR}}} - \rho_{\pi_0}^{\phi_{\text{SR}}}\|_1. \quad \square$$

Connection to our method. The third term $C_\phi^{-1} \|\rho_{\pi_1}^{\phi_{\text{SR}}} - \rho_{\pi_0}^{\phi_{\text{SR}}}\|_1$ in the bound is the price for placing the on-policy SR test-time objective $J_{\phi_{\text{SR}}}(\pi) = \sum_z \rho_\pi^{\phi_{\text{SR}}}(z) V_\pi(z)$ on the right-hand side: as the policy improves, SR's own visit measure $\rho_\pi^{\phi_{\text{SR}}}$ also shifts, and that shift contaminates $J_{\phi_{\text{SR}}}(\pi_1) - J_{\phi_{\text{SR}}}(\pi_0)$. The off-policy fixed-reference quantity

$$J_{\phi_{\text{SR}}}^{\mathcal{R}}(\pi) := \sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(z) V_\pi(z)$$

absorbs this shift by holding the visit measure at the baseline; $J_{\phi_{\text{SR}}}^{\mathcal{R}}(\pi_1) - J_{\phi_{\text{SR}}}^{\mathcal{R}}(\pi_0) = \sum_z \rho_{\pi_0}^{\phi_{\text{SR}}}(\cdot) (V_{\pi_1} - V_{\pi_0})$ exactly, and (5) terminates the proof in one step:

$$J_\phi(\pi_1) - J_\phi(\pi_0) \geq \frac{1}{C_\phi} [J_{\phi_{\text{SR}}}^{\mathcal{R}}(\pi_1) - J_{\phi_{\text{SR}}}^{\mathcal{R}}(\pi_0)] - \|\rho_{\pi_1}^\phi - \rho_{\pi_0}^\phi\|_1.$$

This off-policy form is exactly the training signal REVES optimizes (Section 4): Stage I freezes the rollout distribution at the current π_0 , generating SR trajectories whose visited states constitute samples from $\rho_{\pi_0}^{\phi_{\text{SR}}}$; Stage II then trains V_π on those fixed states via single-turn RL. Each epoch's rollout refresh resets the baseline so the ℓ_1 shift on $\rho^{\phi_{\text{SR}}}$ stays small. The two forms are mathematically equivalent; the off-policy form makes the algorithmic interpretation transparent.

Interpretation. The bound is conditional, not unconditional. It bites when (i) ϕ covers SR's revision inputs (small C_ϕ in (C1)), (ii) the policy update raises one-step recovery on those states ((C2)), and (iii) the visit measures ρ^ϕ and $\rho^{\phi_{\text{SR}}}$ are stable under the update (small ℓ_1 shifts). The empirical confirmation that $J_{\phi_{\text{SR}}}$ -trained policies improve other revision-using TTS algorithms (tree search, evolutionary refinement) is in Table 4.

C.3 Proof of Lemma 4.1

Proof. We prove the three identities in (1) in order.

First identity. SR stops at the first success, so $r^*(x, y_\tau) = 1 \iff \exists t \leq K : r^*(x, y_t) = 1$, and the success event at step t equals $\{\tau \geq t, y_t \in G_x\}$, where $G_x := \{y : r^*(x, y) = 1\}$. By the tower property and the fact that π_θ is Markov in z_t ,

$$\mathbb{P}(\tau \geq t, y_t \in G_x) = \mathbb{E}[\mathbf{1}\{\tau \geq t\} \mathbb{P}(y_t \in G_x | z_t)] = \mathbb{E}[\mathbf{1}\{\tau \geq t\} V_\pi(z_t)].$$

Summing over $t \leq K$,

$$J_{\phi_{\text{SR}}}(\theta) = \mathbb{P}(\exists t \leq K : r^*(x, y_t) = 1) = \sum_{t=1}^K \mathbb{E}[\mathbf{1}\{\tau \geq t\} V_{\pi}(z_t)].$$

Second identity. Using $\sum_{t=1}^K \mathbf{1}\{\tau \geq t\} = \tau$ and Fubini,

$$\sum_{t=1}^K \mathbb{E}[\mathbf{1}\{\tau \geq t\} V_{\pi}(z_t)] = \mathbb{E}\left[\sum_{t=1}^{\tau} V_{\pi}(z_t)\right].$$

Third identity. Substituting the definition $V_{\pi}(z) = \mathbb{E}_{y' \sim \pi_{\theta}(\cdot|z)}[r^*(x, y')]$ and exchanging the visit-count sum with the outer expectation gives

$$\mathbb{E}\left[\sum_{t=1}^{\tau} V_{\pi}(z_t)\right] = \sum_z V_{\pi}(z) \mathbb{E}\left[\sum_{t=1}^{\tau} \mathbf{1}\{z_t = z\}\right] = \sum_z \rho_{\theta}(z) V_{\pi}(z) = \sum_z \rho_{\theta}(z) \mathbb{E}_{y' \sim \pi_{\theta}(\cdot|z)}[r^*(x, y')].$$

□

D Training Prompt Template

In this section, we present the training prompt templates used for data augmentation. An important distinction from the test-time setting is that we do not append execution error information to the context during training. Instead, as illustrated in Figure 12 and Figure 13, the training prompts rely on structured supervision rather than explicit error feedback.

Prompt Template for Revision in Math Tasks

Problem

A frequency distribution of the scores for Mr. Sampson’s algebra class is shown. What percent of the class received a score in the 60% – 69% range?

Test Scores	Frequencies
90% – 100%	III
80% – 89%	III III
70% – 79%	III II
60% – 69%	III I
Below 60%	II

Previous Answer

To solve this problem, we first count the number of students in each score range. The total number of students is computed as $4 + 8 + 6 + 5 + 2 = 25$. The number of students scoring in the 60% – 69% range is 5. ...

Verifiable Feedback:

The previous answer is incorrect.

Instruction:

Now produce a corrected solution. Identify and fix potential mistakes in the previous attempt. Keep the reasoning concise and provide a corrected final answer. Do not assume access to the correct answer.

Figure 8: Prompt template for the revision stage with binary correctness feedback. Concrete problem content and partial incorrect answers are shown for illustration, with remaining content omitted for brevity.

E Implementation Details

In this section, we provide details of the training and evaluation implementations used in our experiments. All experiments are conducted on machines equipped with $8 \times$ NVIDIA H200 GPUs.

Prompt Template for Verification in Math Tasks

Problem

A frequency distribution of the scores for Mr. Sampson’s algebra class is shown. What percent of the class received a score in the 60% – 69% range?

Test Scores	Frequencies
90% – 100%	III
80% – 89%	III III
70% – 79%	III II
60% – 69%	III I
Below 60%	II

Proposed Answer

To solve this problem, we first identify the number of students in each score range from the frequency distribution. The total number of students is computed as $4 + 8 + 6 + 5 + 2 = 25$. Among them, 5 students fall into ...

Instruction:

Judge whether the proposed answer is correct. Respond concisely and end the final line with

VERDICT: True or VERDICT: False.

Figure 9: Prompt template used for the verification stage. The verifier observes the problem and a proposed solution and outputs a binary correctness judgment.

RL Baseline. Our RL baseline is implemented using the official Skywork-OR1 codebase. We directly adopt their provided training scripts and configurations, which optimize a single-shot response using reinforcement learning. No revision or multi-turn generation is involved during training for this baseline.

Multi-Turn and PAG Baselines. The Multi-Turn and PAG baselines are implemented based on the Policy-As-GenVerifier codebase. To ensure comparability with our setting, we make two modifications to the original implementation. First, we increase the maximum response length to 8092 tokens to match our training setup. Second, we adapt the reward computation function so that it correctly evaluates responses under our datasets, while keeping the reward definition itself identical to that used in our methods. Apart from these changes, the original training pipelines and optimization settings are preserved.

REVES (Ours). Our method builds on top of the same training framework as the baselines. The key difference lies in the data augmentation procedure. Before training, we first perform an offline augmentation pass to initialize the augmented dataset. During training, at the beginning of each epoch, we further augment the dataset by generating new revision trajectories using the current policy, and the newly generated samples are added to the training data. When generating augmented data, we allow a maximum revision budget of $K = 8$. The training prompts used for augmentation follow the templates described in Appendix D. The model is then trained on the mixture of original and augmented data using the same RL objective as the baseline.

Evaluation Protocol. Unless otherwise specified, all experiments are evaluated using our unified evaluation framework, which will be publicly released upon paper acceptance. Given an input problem, the model follows the corresponding test-time protocol (e.g., sequential revision when applicable) to generate a final response. The generated output is serialized into a JSON file and evaluated by an external, task-specific verifier. This pipeline is applied uniformly across all tasks and methods to ensure fair comparison: generation and evaluation are fully decoupled, and correctness is determined solely by the external verifier rather than any internal model signals. During evaluation, the maximum response length is set to 16,384 tokens for coding tasks and AIME24/25, and 8,096 tokens for MATH500.

The decoding temperature is fixed to 0.6 for coding tasks and 1.0 for all other tasks, and the confidence threshold in the SelfConf setting is set to $c = 0.5$.

For math tasks, when evaluating the Multi-Turn and PAG baselines, we follow the original evaluation implementation provided in the Policy-As-GenVerifier repository to remain faithful to their protocol. Under this evaluation pipeline, we use the same maximum response length settings for 7B models as described above. For 3B models with revision budget $K = 4$, where the effective sequence length exceeds the model context limit, we reduce the maximum response length to 4,096 tokens. All math evaluations are conducted using `math_eval`, ensuring that results are directly comparable across evaluation settings.

F Method Details: Multi-Turn Comparison, Efficiency, and Test-Time Sensitivity

F.1 Comparison with Naive Multi-Turn RL

A natural alternative to our offline scheme is online sequential revision with standard multi-turn credit assignment. Two such schemes appear in the recent literature. The first, *trajectory-level broadcast*, samples G trajectories of M turns per prompt, scores each trajectory by its final reward, computes a trajectory-level advantage, and broadcasts that single advantage uniformly to every turn in the trajectory. The second, *pooled per-response*, pools all $G \times M$ responses across the G trajectories, treats each response as an independent sample, and computes advantages within the pool.

Neither scheme suits our setting. Sequential revision is structurally different from a vanilla multi-turn task: each turn is a fresh attempt at the *same* problem x , not a step in a longer-horizon decision process with intermediate sub-rewards. Trajectory-level broadcast assigns identical credit to every turn and therefore ignores that some intermediate responses are much closer to correctness than others. Pooled per-response also fails to capture this structure: the wrong intermediate responses are treated identically, even though the near-miss failures are exactly the ones we most want the policy to learn from. Our offline selection-then-replay scheme bypasses both pathologies. We identify near-miss failures explicitly through the success-trajectory filter and replay them as single-turn revision prompts, where standard single-turn credit assignment applies cleanly.

F.2 Wall-Clock Efficiency

Beyond avoiding credit-assignment pitfalls, our offline scheme is substantially faster than online multi-turn training. Multi-turn RL pays a length- M serial sequential-revision cost per prompt at every gradient step, multiplied by the G rollouts needed to estimate the advantage. Our method pays the length- K serial cost *only once* per prompt at the offline stage, and during the online RL stage performs only single-turn sampling on the extracted prompts, which can be parallelized across N Best-of- N samples. Single-turn parallel sampling is much cheaper per gradient step than M -turn serial generation across G rollouts, since serial generation does not benefit from the parallelism that modern inference engines provide for single-turn sampling.

Empirically, on $8 \times$ NVIDIA H200 GPUs, three full iterations of our method completed in roughly 30 hours, while the equivalent online multi-turn baseline required over 48 hours on identical hardware (Table 6). With a more carefully optimized infrastructure, Stage I data generation and Stage II RL training could run concurrently (Stage II first trains on the original RL data while Stage I generates augmented data asynchronously), yielding further gains.

F.3 Sensitivity to the Confidence Threshold

We study the sensitivity of our test-time stopping rule to the confidence threshold c used in the TailConfidence stopping criterion (Section 4). Table 7 reports accuracy and the average number of revision rounds used at budget $K = 16$ across three math benchmarks. Higher

Table 6: Per-iteration wall-clock breakdown of our method on $8 \times$ NVIDIA H200 GPUs.

Iteration	Data Generation (Stage I)	RL Training (Stage II)	Total
Iter 1	3h 4m	7h 15m	10h 19m
Iter 2	2h 41m	7h 26m	10h 7m
Iter 3	3h 13m	6h 23m	9h 36m
Total	8h 58m	21h 4m	30h 2m

thresholds enforce stricter stopping (more rounds before accepting a response), and accuracy increases monotonically with the threshold over the range we tested. The number of rounds also grows with the threshold, reflecting the cost-quality trade-off.

Table 7: Sensitivity of test-time accuracy to the TailConfidence threshold c on Qwen-2.5-7B-REVES at budget $K = 16$. “Rounds” is the average number of revision rounds before stopping.

Threshold c	AIME25		AIME24		MATH500	
	Acc. (%)	Rounds	Acc. (%)	Rounds	Acc. (%)	Rounds
0.4	16.56	10.22	21.15	9.04	77.80	3.99
0.5	16.88	11.91	21.56	10.06	79.17	4.46
0.6	17.50	13.31	22.08	12.19	79.10	5.64
0.7	18.23	15.45	23.65	14.97	81.58	8.76

G Additional Experiments and Ablations

We additionally evaluate our method using a strong reasoning-oriented base model, DeepSeek-R1-Distill-7B (DeepSeek-AI et al., 2025), and compare against both the original base model and an RL baseline. Due to the substantially longer responses produced by this model (approximately 9k tokens on average, compared to $\sim 2k$ tokens for Qwen-7B), multi-turn training becomes significantly more expensive, which limits the scale of these experiments.

As shown in Table 8, our REVES model consistently improves test-case-based performance over both the base model and the RL baseline. We expect larger gains with increased data scale and longer training.

H Circle Packing Implementation Details

This section gives the implementation details behind the circle packing results in Table 2. Circle packing maximizes the sum of radii of n non-overlapping circles inscribed in the unit square, with verifier feedback given by a non-overlap and inscription check together with the realized objective value. We instantiate REVES on this benchmark by directly optimizing $J_{\phi_{SR}}$ in the recovery form of Lemma 4.1: each visited state z_t corresponds to a partially-improved candidate configuration, the policy proposes a refined configuration

Table 8: Performance on LiveCodeBench under different evaluation periods.

Method	LiveCodeBench (01/01/25–05/01/25)		LiveCodeBench (08/01/24–01/01/25)	
	OneShot	TestCases-32	OneShot	TestCases-32
Base (Distill-7B)	34.25	44.51	38.24	52.21
Base + RL	36.63	43.96	38.79	53.19
Base + REVES	35.53	46.33	40.63	54.04

$y_t \sim \pi_\theta(\cdot \mid z_t)$, and the verifier returns $r^*(x, y_t) = 1$ iff y_t improves on the running best while remaining feasible.

Reward shaping. The raw objective of a candidate y is its sum of radii $g(y) := \sum_{i=1}^n r_i(y)$ (a continuous quantity), which clusters in a narrow band near the optimum ($g \approx 2.63$ for the $n = 26$ instance we evaluate) and differs across candidates only at the third or fourth decimal. Used directly, this signal is too compressed to drive learning. We therefore shape it in two steps. Let $v(y) \geq 0$ denote the total constraint violation (circle overlap plus boundary excess), so that y is feasible iff $v(y) = 0$. First, define the raw score $s(y)$ as

$$s(y) = \begin{cases} g(y), & v(y) = 0, \\ g(y) - v(y), & v(y) > 0 \text{ and } g(y) \leq 2.64, \\ -1, & v(y) > 0 \text{ and } g(y) > 2.64, \end{cases}$$

i.e., minor violations are absorbed as a soft penalty, while implausibly high sums coupled with infeasibility are flagged as cheating and assigned $s = -1$. Second, we amplify by squaring the offset from 2.5: $R(s) := (s - 2.5)^2$. Since feasible raw scores live above 2.5, this magnifies small absolute differences, e.g., a $\Delta s = 10^{-3}$ gain near $s = 2.6$ produces a $\Delta R \approx 2 \cdot 10^{-4}$, which is no longer dwarfed by gradient noise. The reward used for a revision y from parent state x is the resulting improvement,

$$U(y \mid x) := R(s(y)) - R(s(x)),$$

so the policy is rewarded for any move that improves on the parent and penalized for any move that worsens it.

Stage I: search rollouts and prompt-set construction. We run 4 sequential-revision rollouts under the current π_θ with budget $K = 100$, where each step proposes a refinement of the previous candidate configuration and the verifier returns the shaped reward $U(y \mid x)$ defined above. From the visited intermediate states across these rollouts, we filter to retain those whose subsequent revisions produce verified improvements, and assemble the resulting revision prompts into the augmented prompt set.

Stage II: RL training. We train Qwen3-4B with GRPO (DeepSeek-AI et al., 2025) on the prompt set from Stage I. Each gradient step samples 8 prompts; for each prompt we draw 64 rollouts to form the group used for the GRPO advantage estimate.

A representative solution. Figure 10 visualizes one of the configurations REVES discovers, and Figure 11 reproduces the Python program emitted by the policy that generates it: an initial grid placement followed by an SLSQP refinement against the non-overlap and inscription constraints.

```

Circle-Packing Program

1 import numpy as np
2 from scipy.optimize import minimize
3
4
5 def run_packing():
6     num_circles = 26
7     r_initial = 0.08
8     x_positions = np.linspace(r_initial, 1 - r_initial, 6)
9     y_positions = np.linspace(r_initial, 1 - r_initial, 5)
10    positions = [(x, y) for x in x_positions for y in y_positions][:25]
11    positions.append((0.3, 0.3))
12    positions = np.unique(positions, axis=0)
13    if len(positions) < 26:
14        positions = np.append(positions, [(0.5, 0.5)] * (26 - len(positions)), axis=0)
15
16    radii = np.full(num_circles, r_initial)
17
18    variables = np.zeros(num_circles * 3)
19    for i in range(num_circles):

```

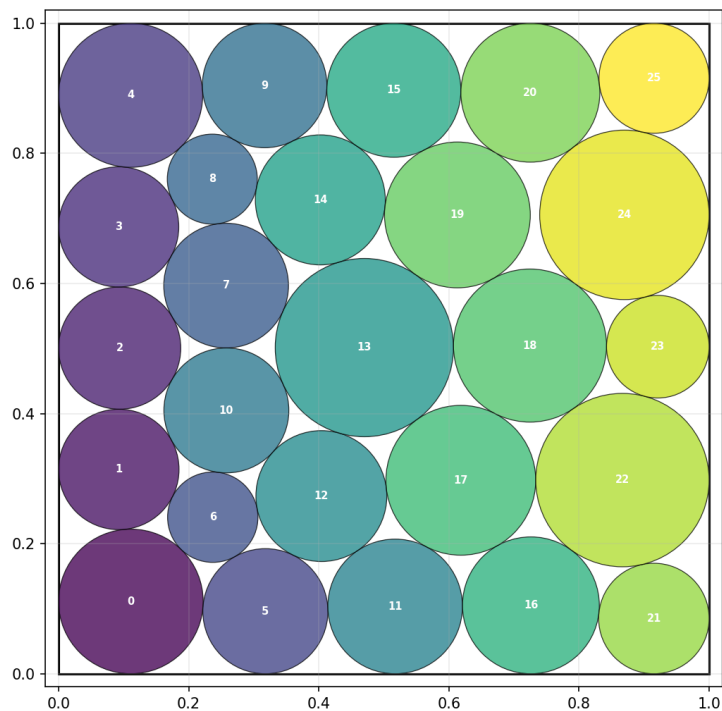


Figure 10: A circle packing configuration found by REVES on the $n = 26$ instance, achieving $\text{sum-of-radii} = 2.635983$.

```

20     variables[2 * i] = positions[i][0]
21     variables[2 * i + 1] = positions[i][1]
22 for i in range(num_circles):
23     variables[2 * num_circles + i] = radii[i]
24
25 def objective(vars):
26     return -np.sum(vars[num_circles * 2:])
27
28 constraints = []
29 for i in range(num_circles):
30     def con1(vars, i=i):
31         centers = vars[:num_circles * 2].reshape(num_circles, 2)
32         r = vars[2 * num_circles + i]
33         return centers[i, 0] - r
34     constraints.append({'type': 'ineq', 'fun': con1})
35
36     def con2(vars, i=i):
37         centers = vars[:num_circles * 2].reshape(num_circles, 2)
38         r = vars[2 * num_circles + i]
39         return 1 - centers[i, 0] - r
40     constraints.append({'type': 'ineq', 'fun': con2})
41
42     def con3(vars, i=i):
43         centers = vars[:num_circles * 2].reshape(num_circles, 2)
44         r = vars[2 * num_circles + i]
45         return centers[i, 1] - r
46     constraints.append({'type': 'ineq', 'fun': con3})
47
48     def con4(vars, i=i):
49         centers = vars[:num_circles * 2].reshape(num_circles, 2)
50         r = vars[2 * num_circles + i]
51         return 1 - centers[i, 1] - r
52     constraints.append({'type': 'ineq', 'fun': con4})
53
54 for i in range(num_circles):
55     def con5(vars, i=i):
56         return vars[2 * num_circles + i]

```

```

57     constraints.append({'type': 'ineq', 'fun': con5})
58
59     for i in range(num_circles):
60         for j in range(i + 1, num_circles):
61             def con6(vars, i=i, j=j):
62                 centers = vars[:num_circles * 2].reshape(num_circles, 2)
63                 r_i = vars[2 * num_circles + i]
64                 r_j = vars[2 * num_circles + j]
65                 dx = centers[i, 0] - centers[j, 0]
66                 dy = centers[i, 1] - centers[j, 1]
67                 return dx**2 + dy**2 - (r_i + r_j)**2
68                 constraints.append({'type': 'ineq', 'fun': con6})
69
70     result = minimize(
71         objective,
72         variables,
73         constraints=constraints,
74         method='SLSQP',
75         options={'maxiter': 2000000, 'ftol': 1e-10, 'disp': False}
76     )
77
78     centers = result.x[:num_circles * 2].reshape(num_circles, 2)
79     radii = result.x[num_circles * 2:]
80     return centers, radii, float(np.sum(radii))

```

Figure 11: Python program output by the REVES-trained Qwen3-4B policy that produces the configuration in Figure 10.

Prompt Template for Code Revision

Problem

You will be given a problem specification and are asked to generate a correct Python program that satisfies the requirements and passes all tests. The task involves extracting two integers and an operator (“gains” or “loses”) from a natural language string and returning the result of the corresponding calculation. ...

Previous Answer

To address the issue in the provided code, there are a few key points ...

```

class Solution:
    def calculate(self, string: str) -> int:
        words = string.split()
        ...
        joined_string = "".join(words)
        result = eval(joined_string)
        return result

```

...

Verifiable Feedback:

The previous answer is incorrect.

Instruction:

Please repair the code accordingly. Enclose your fixed implementation within the specified Python code delimiters. Do not assume access to the correct implementation.

Figure 12: Prompt template for the code revision stage with binary correctness feedback. Partial incorrect code is shown for illustration, with remaining content omitted for brevity.

Prompt Template for Code Verification

Problem

You will be given a problem specification and are asked to generate a correct Python program that satisfies the requirements and passes all tests. The task involves extracting two integers and an operator (“gains” or “loses”) from a natural language string and returning the result of the corresponding calculation. ...

Candidate Answer:

To address the issue in the provided code, there are a few key points ...

```
class Solution:
    def calculate(self, string: str) -> int:
        words = string.split()
        ...
        joined_string = "".join(words)
        result = eval(joined_string)
        return result
```

...

Instruction:

Please verify whether the candidate answer is correct. At the end, output the final line exactly as:
VERDICT: \boxed{True/False}

Figure 13: Prompt template for the code verification stage. The verifier evaluates a candidate solution and outputs a binary correctness verdict.