LLM-AS-AN-IMPROVER: TURNING VERIFICATION INTO BETTER CANDIDATES

Anonymous authors
Paper under double-blind review

ABSTRACT

Verifier-based selection chooses the most promising solution from a pool of candidates generated at inference time. Because it only ranks a fixed candidate pool, however, it must fail when every initial candidate is incorrect, even if verification identifies useful failure evidence. We introduce LLM-as-an-Improver and propose Verify–Repair–Reselect (VRR), which reuses that evidence to expand the candidate set before final selection. VRR retains the initial winner while conditionally generating complementary alternatives, screens invalid and duplicate outputs, and then reselects from the retained winner and the surviving alternatives. Our experiments show that VRR can recover correct solutions even when all candidates in the initial pool are incorrect. These results demonstrate the feasibility of using verification feedback not only as a ranking signal but also as a generation signal for constructing correct candidates beyond the initial pool.

1 INTRODUCTION

Test-time scaling improves the capabilities of large language models (LLMs) by allocating additional inference compute to candidate generation and selection. Generation methods increase solution coverage through repeated sampling or diverse reasoning paths (Wang et al., 2023; Brown et al., 2024), although the effectiveness of additional compute depends on both problem difficulty and its allocation (Snell et al., 2025). Some methods use verifiers, which attempt to identify the strongest candidate using outcome, process, or LLM-based judgments (Cobbe et al., 2021; Lightman et al., 2024; Kwok et al., 2026). In these methods, generation determines which candidates are available, and verification is used only to rank them. We use LLM-as-a-Verifier (LAV) (Kwok et al., 2026), a fixed-pool selection method based on fine-grained LLM judgments, as the principal verifier-based baseline.

However, these methods do not fully exploit the potential of verification. A verifier may identify violated requirements, unsupported reasoning steps, failing cases, and unresolved claims, yet selection-only inference compresses this information into a ranking and then discards it. Moreover, ranking cannot repair the search space: if every candidate in the initial pool is incorrect, even an oracle selector must fail. This leads to the central research question of this paper: *Can verification also guide the construction of better candidates?*

We refer to this expanded role as LLM-as-an-Improver: rather than using verification only to choose among existing candidates, the model reuses diagnostic evidence to construct additional candidates. We propose Verify–Repair–Reselect (VRR), a test-time method that uses structured verification results both to rank candidates and to guide additional generation. VRR first selects and retains the top-ranked initial candidate. When the initial pool appears unreliable, it generates three alternatives: a local repair of the winner, a repair of the runner-up, and a solution using a new approach. It removes invalid and duplicate alternatives, then selects the final answer from the surviving alternatives and the initial winner using the same evaluation criteria. Unlike single-trajectory refinement, VRR treats verifier feedback as a signal for expanding the choice set: it preserves the initial winner, creates complementary alternatives, and subjects all surviving candidates to reselection rather than accepting a revision automatically.

Our primary claim concerns this recovery beyond the fixed pool. On LiveCodeBench problems for which all five initial candidates are incorrect, VRR recovers 18 of 117 problems with Gemma 4 31B IT and 7 of 215 with Qwen3-32B. This result establishes that verification evidence can guide

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

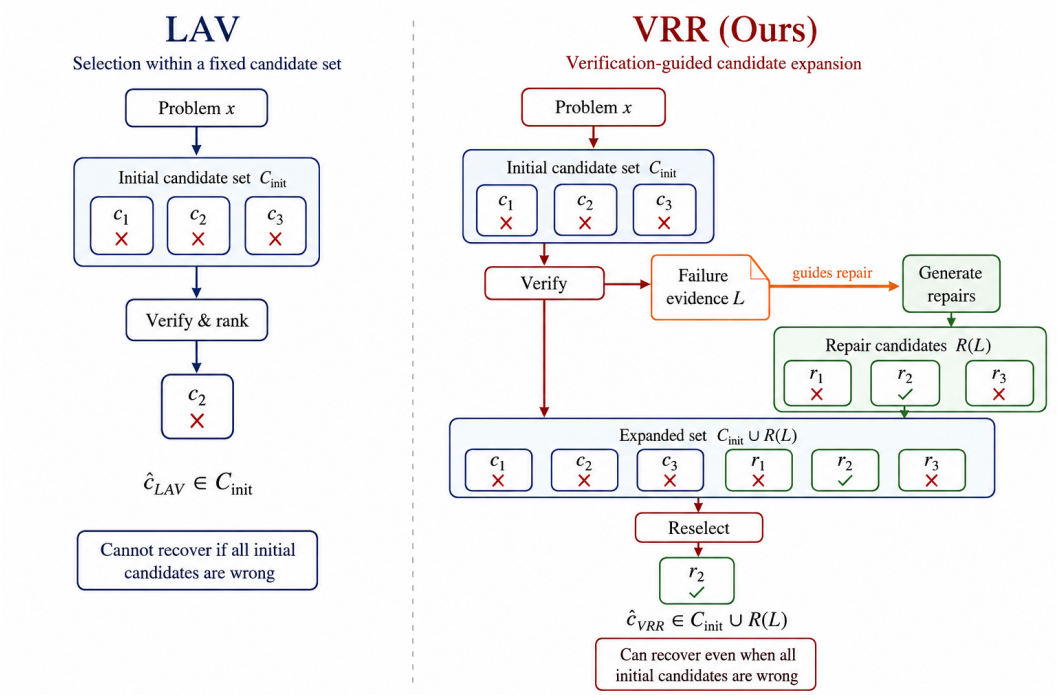

Figure 1: Two uses of verification at test time. LAV uses verification to rank a fixed initial set. VRR additionally turns diagnostic verification evidence into new candidates before reselection, making recovery possible even when every initial candidate is incorrect.

the construction of a correct candidate that ranking alone could never return. As a complementary aggregate evaluation, across two models and eight benchmarks, VRR has higher point accuracy than fixed-pool LAV selection in nine of 16 settings, ties it in four, and is lower in three.

Our contributions are threefold. First, we formalize the fixed-pool ceiling that prevents any selection-only verifier from succeeding when the initial pool contains no correct candidate. Second, we introduce VRR, which connects candidate-blind verification, a complementary repair portfolio, and reselection under the same criteria. Third, we empirically demonstrate recovery beyond the initial pool and separately report its relationship to aggregate accuracy, including errors on problems already covered by the initial pool.

Figure 1 illustrates the central distinction: selection-only inference uses verification to change the ranking, whereas evidence-guided expansion also uses it to change the candidates being ranked.

2 Problem Setting

2.1 Verification for Candidate Ranking

Candidate generation and verifier-based selection play distinct roles in test-time inference: generation determines which solutions are available, while verification determines which candidate is returned. Pass@k measures whether at least one of k candidates is correct (Brown et al., 2024); verifier-based selection attempts to convert this candidate coverage into final-answer accuracy.

Formally, let x denote a task and let $\mathcal{C}_{\text{init}} = \{c_1, \ldots, c_N\}$ be the initial candidate set produced by an LLM generator. Each c_i is a candidate solution containing a final answer and, when appropriate, its reasoning process or code. In selection-only methods, a verifier V scores or compares these candidates and returns one of them:

$$\hat{c}_{\text{select}} = \text{Select}_V\left(x, \mathcal{C}_{\text{init}}\right), \qquad \hat{c}_{\text{select}} \in \mathcal{C}_{\text{init}}. \tag{1}$$

This formulation includes outcome verifiers, process verifiers, generative verifiers, and LLM-based judges (Cobbe et al., 2021; Lightman et al., 2024; Zhang et al., 2025; Kwok et al., 2026). These assessments may be rich, but selection-only methods use them for a single purpose: ranking an already generated set.

2.2 Why Ranking Alone Is Insufficient

Let $Y_x(c) \in \{0, 1\}$ denote correctness under the task's evaluation rule. This label is used only for evaluation and is unavailable to the inference procedure. For task x, the initial-pool coverage indicator is

$$H_0(x) = \max_{c \in \mathcal{C}_{\text{init}}} Y_x(c). \tag{2}$$

We define the initial-pool oracle accuracy as

$$A_{\text{oracle}} := \mathbb{E}[H_0(x)], \tag{3}$$

where expectations are taken over the task distribution and any randomness in candidate generation and inference. This is the accuracy of an oracle selector that returns a correct candidate whenever the initial pool contains one. Because $Y_x(\hat{c}_{\text{select}}) \leq H_0(x)$ for every task x, any selection-only method satisfies

$$\mathbb{E}[Y_x(\hat{c}_{\text{select}})] \leq A_{\text{oracle}}. \tag{4}$$

For a fixed pool, $H_0(x)$ is a post-evaluation coverage indicator, not an estimate of selected-candidate correctness, and is unavailable at inference time. This ceiling requires no independence assumption. It separates candidate *coverage* from the selector's ability to identify a correct candidate. The limitation therefore lies not in verifier strength but in using verification only to rearrange candidates. A stronger verifier can approach the ceiling, but only candidate generation can raise it. Diagnostic content produced during verification offers a task-specific signal for doing so.

2.3 From Ranking to Candidate Improvement

We therefore extend the role of verification rather than replace verifier-based selection. Formally, verification produces both a ranking signal r_V and diagnostic evidence D_V:

$$(r_V, D_V) = \text{Verify}_V(x, \mathcal{C}_{\text{init}}). \tag{5}$$

Here, Verify_V evaluates the initial candidates for task x and returns both the information used to rank them, r_V, and candidate-level diagnostic evidence, D_V. Selection-only inference consumes r_V and leaves D_V unused. Equation 5 is the *Verify* stage. The diagnostic evidence is also fed back into generation, including criterion-level critiques, counterexamples, and unresolved claims:

$$\mathcal{R} = G(x, \mathcal{C}_{\text{init}}, D_V), \qquad \mathcal{R}_{\text{pass}} = \text{Screen}(x, \mathcal{R}; \mathcal{C}_{\text{init}}), \tag{6}$$

where G denotes a candidate generator that conditions on the task, the initial candidates, and the verification evidence to produce a set $\mathcal{R}$ of additional candidates. Screen applies inference-time validity checks and removes candidates that duplicate an initial or already retained candidate, yielding the admissible set $\mathcal{R}_{\text{pass}}$. Screening is pool-dependent because duplicate status depends on the other candidates. Passing the screen establishes eligibility for reselection, not correctness. Equation 6 is the *Repair* stage, including inference-time screening of the generated alternatives. A retention policy then chooses a subset $\mathcal{K} = K(\mathcal{C}_{\text{init}}, D_V)$ of initial candidates. The final output is

$$\hat{c}_{\text{expand}} = \text{Select}_V\left(x, \mathcal{K} \cup \mathcal{R}_{\text{pass}}\right). \tag{7}$$

The expanded set is then verified again, so the same capability supports both candidate improvement and final ranking. Equation 7 is the *Reselect* stage. Selection-only inference corresponds to the special case that ignores D_V, generates no new candidates, and sets $\mathcal{K} = \mathcal{C}_{\text{init}}$. Candidate expansion makes recovery possible even when $H_0(x) = 0$.

3 Verify–Repair–Reselect

VRR first ranks the initial candidates, reuses the same criterion-level evidence to guide candidate improvement, and then performs final reselection. Its implementation follows the three stages in its name:

1. **Verify:** derive candidate-blind evaluation criteria, evaluate the initial candidates, select the winner, and decide whether expansion is needed.
2. **Repair:** if triggered, generate three complementary alternatives and screen them for validity and duplication.
3. **Reselect:** compare the retained initial winner and the surviving alternatives under the original criteria.

The remainder of this section follows these three stages directly.

3.1 VERIFY

The Verify stage creates the evaluation criteria, checks every initial candidate against those criteria, and selects the initial winner while retaining diagnostic evidence for possible repair.

3.1.1 CREATING EVALUATION CRITERIA

For each task, we derive a fixed finite collection of criteria from the problem statement x:

$$\Gamma(x) = \mathcal{O}_{\mathrm{hard}}(x) \cup \mathcal{O}_{\mathrm{soft}}(x), \quad |\mathcal{O}_{\mathrm{hard}}(x)| \geq 1. \tag{8}$$

Each evaluation item o_j specifies what to judge, what evidence may be used, possible counterexamples, and a score initialization parameter π_j.

Hard items are shared within a dataset and treated as necessary conditions for correctness. For example, code tasks consider conformance to the specification, functional correctness, syntax errors, failures on public tests, and complexity violations. By contrast, soft items are generated by the LLM separately for each problem based on its problem statement and are used only to break near-ties in the hard-item scores.

3.1.2 CHECKING CANDIDATES BY EVALUATION ITEM

Before evaluating candidate c_i, we organize the information available to the verifier as

$$E_i \leftarrow \mathrm{BuildEvidenceGraph}(x, c_i). \tag{9}$$

Here, E_i collects the information used to evaluate candidate c_i. It organizes the problem statement x, candidate c_i, and public information available at inference time into a form that the verifier can reference.

For general reasoning tasks, the candidate response is divided into reasoning steps and a final answer, and each element is linked to its location in the response. For code tasks, E_i records the candidate code or patch together with syntax checks, compilation results, and the results of designated public tests. For tasks involving file changes, it also records the changed locations and target files. Each piece of information is assigned a source location or identifier so that the verifier can cite a specific part of the candidate or a public test result as the basis for its judgment.

For candidate i and evaluation item j, the verifier's t-th judgment returns

$$\boldsymbol{q}_{ij}^{(t)} = \left(q_{ij}^{\mathrm{SAT},(t)}, q_{ij}^{\mathrm{VIOL},(t)}, q_{ij}^{\mathrm{UNKNOWN},(t)}\right) \in \Delta^2. \tag{10}$$

Here, Δ^2 is the probability simplex over the three labels. The verifier also returns $g_{ij}^{(t)} \in \{0, 1\}$, which indicates whether it grounds the judgment in the available evidence. The initial batched evaluation corresponds to $t = 1$, and adaptive evaluations produce up to T_{ij} judgments for each entry. The UNKNOWN state separates missing evidence from evidence that supports satisfaction. We define the judgment weight as

$$w_{ij}^{(t)} = g_{ij}^{(t)} \left(1 - q_{ij}^{\mathrm{UNKNOWN},(t)}\right) \left(1 - \frac{H(\boldsymbol{q}_{ij}^{(t)})}{\log 3}\right). \tag{11}$$

Here, H denotes Shannon entropy. This weight lies in $[0, 1]$; it is zero for an ungrounded, fully unresolved, or uniform judgment and otherwise discounts reported uncertainty.

We convert the weighted judgments into a criterion-level verification score. Let $\pi_j \in (0,1)$ be an initialization parameter and $\epsilon > 0$ a numerical smoothing constant. Define

$$\ell_{ij} = \mathrm{logit}(\pi_j) + \sum_{t=1}^{T_{ij}} w_{ij}^{(t)} \log \frac{q_{ij}^{\mathrm{SAT},(t)} + \epsilon}{q_{ij}^{\mathrm{VIOL},(t)} + \epsilon}. \tag{12}$$

We map this value to $(0,1)$ as

$$p_{ij} = \mathrm{sigmoid}(\ell_{ij}). \tag{13}$$

Equations 12–13 define a bounded, confidence-weighted score for ranking and triggering. The verifier's reported ratio $q^{\mathrm{SAT}}/q^{\mathrm{VIOL}}$ is not an observed likelihood ratio, repeated judgments may be dependent, and evaluation criteria may overlap. The log-odds form is therefore an aggregation heuristic, not a probabilistic guarantee.

We aggregate the hard items as

$$S_i = \prod_{o_j \in \mathcal{O}_{\mathrm{hard}}} p_{ij}. \tag{14}$$

The product is a multiplicative aggregation rule that penalizes a low score on any required item. It is not an established probability that all hard criteria hold, and its scale depends on the number and definition of the criteria. Soft items break ties only when two hard-item scores differ by at most τ_{soft}.

3.1.3 Adaptive Verification and Initial Selection

The verifier first judges every candidate–item entry once in a batch. It then directs the remaining budget to entries that can change the final ranking. We use the following acquisition score:

$$A_{ij} = I_{ij}\, H_2(p_{ij})\, D_i/\kappa_{ij}. \tag{15}$$

Here, I_{ij} indicates whether item j can change the all-hard-items decision for candidate i. The Bernoulli entropy $H_2(p_{ij})$ measures uncertainty. The factor D_i prioritizes candidates near the selection boundary. The term κ_{ij} denotes the expected judgment cost. At each step, the verifier judges the entry with the largest A_{ij}.

The selector then compares the top two candidates directly. It evaluates both presentation orders, $A = c_a, B = c_b$ and $A = c_b, B = c_a$, for every evaluation item (Wang et al., 2024a). Hard-item comparison evidence updates the candidate scores. Soft items affect the decision only when the hard scores are close. This comparison selects the initial winner c_{i^*}. Let $\boldsymbol{\rho}$ denote the resulting ranking, where $c_{\rho_1} = c_{i^*}$ and c_{ρ_2} is the runner-up.

3.2 Repair

The Repair stage summarizes the verification evidence, determines whether candidate expansion is needed, generates complementary alternatives when triggered, and screens the generated candidates before reselection.

3.2.1 Summarizing Results and Triggering Expansion

We summarize the initial evaluation in L, a diagnostic record that makes information produced for ranking reusable by the generator. For each candidate, L records the evaluation items, aggregate score, uncertainty interval, and initial rank. Using the initial batched judgments, L records strengths with $q_{ij}^{\mathrm{SAT},(1)} \geq \tau_{\mathrm{sat}}$, potential issues with $q_{ij}^{\mathrm{VIOL},(1)} \geq \tau_{\mathrm{viol}}$, and unresolved items with $q_{ij}^{\mathrm{UNKNOWN},(1)} \geq \tau_{\mathrm{unk}}$. Potential issues are not treated as confirmed errors; they are passed to the repair generator as items that require re-examination.

We decide whether to generate additional candidates using two conditions:

$$\text{low-pool-score} \iff \max_i S_i < \tau_{\mathrm{score}}, \tag{16}$$

$$\text{high-unknown-count} \iff \#\left\{(i,j) : o_j \in \mathcal{O}_{\mathrm{hard}}, q_{ij}^{\mathrm{UNKNOWN},(1)} \geq \tau_{\mathrm{unk}}\right\} \geq K_{\mathrm{unk}}. \tag{17}$$

Algorithm 1 Verify–Repair–Reselect (VRR)

Require: Task x; initial candidates $\mathcal{C}_{\text{init}}$; verifier V; repair generator G; budgets $B_{\text{init}}, B_{\text{reselect}}$; thresholds $\tau_{\text{sat}}, \tau_{\text{viol}}, \tau_{\text{score}}, \tau_{\text{unk}}, K_{\text{unk}}$
Ensure: Final candidate $\hat{c}$

$\Gamma \leftarrow \text{CreateCriteria}(x)$
for all $c_i \in \mathcal{C}_{\text{init}}$ **do**
  $E_i \leftarrow \text{BuildEvidenceGraph}(x, c_i)$
end for
$(c_{i^*}, \boldsymbol{S}, L, \boldsymbol{\rho}) \leftarrow \textsc{Verify-Select}(x, \mathcal{C}_{\text{init}}, \{E_i\}, \Gamma, V, B_{\text{init}})$
$u \leftarrow \#\{(i,j) : o_j \in \mathcal{O}_{\text{hard}}, q_{ij}^{\textsc{Unknown},(1)} \geq \tau_{\text{unk}}\}$
if $\max_i S_i \geq \tau_{\text{score}}$ **and** $u < K_{\text{unk}}$ **then**
  return c_{i^*}
end if
$\mathcal{R} \leftarrow \text{Repair}(x, c_{\rho_1}, c_{\rho_2}, L, G)$ {selected, runner-up, and new-approach}
$\mathcal{R}_{\text{pass}} \leftarrow \{c \in \mathcal{R} : \text{Screen}(x, c) = 1\}$
if $\mathcal{R}_{\text{pass}} = \varnothing$ **then**
  return c_{i^*}
end if
$\mathcal{C}_{\text{reselect}} \leftarrow \{c_{i^*}\} \cup \mathcal{R}_{\text{pass}}$
for all $c \in \mathcal{R}_{\text{pass}}$ **do**
  $E_c \leftarrow \text{BuildEvidenceGraph}(x, c)$
end for
$\mathcal{E}_{\text{reselect}} \leftarrow \{E_{i^*}\} \cup \{E_c : c \in \mathcal{R}_{\text{pass}}\}$
$(\hat{c}, \cdot, \cdot, \cdot) \leftarrow \textsc{Verify-Select}(x, \mathcal{C}_{\text{reselect}}, \mathcal{E}_{\text{reselect}}, \Gamma, V, B_{\text{reselect}})$
return $\hat{c}$

The thresholds τ_{sat} and τ_{viol} determine which strengths and potential issues enter L. The threshold τ_{score} defines a low candidate score. The threshold τ_{unk} defines an unresolved judgment. The integer K_{unk} defines the required number of unresolved judgments. Appendix A specifies all thresholds. The repair trigger is

$$\mathcal{T} = \text{low-pool-score} \vee \text{high-unknown-count}. \tag{18}$$

The first condition holds when every candidate score is below τ_{score}. The second holds when at least K_{unk} hard-item judgments from the initial batch remain unresolved. If neither condition holds, VRR returns c_{i^*} without additional generation.

3.2.2 Generating Three Complementary Alternatives

When $\mathcal{T}$ is true, VRR changes the role of the initial verification result from a ranking signal into a generation signal and produces three additional candidates. The three roles cover both local errors and failures of the underlying approach, although they do not guarantee semantically distinct outputs.

Selected-candidate repair. The generator re-examines potential issues in the winner while preserving the remaining parts where possible.

Runner-up repair. It separately repairs the second-ranked candidate to retain a potentially different starting point.

New-approach solution. It identifies possible failure conditions from the winner and its evaluation, then solves the task from scratch rather than modifying that candidate.

Each prompt contains the problem, its source candidate, and the corresponding evaluation summary, but no correct or reference answer, private-test result, or correctness label. Because reported issues are verifier estimates rather than confirmed errors, the generator re-examines them and ignores unsupported feedback. Together, the roles favor minimal repair, an alternative starting point, and a distinct approach instead of repeatedly refining one trajectory (Madaan et al., 2023; Shinn et al., 2023).

3.2.3 SCREENING ADDITIONAL CANDIDATES

VRR screens each new candidate before reselection. For code tasks, we retain a candidate only if we can extract a complete artifact, the artifact compiles, and it passes every designated public test. We also require its normalized code hash to differ from those of all initial and retained candidates. For multiple-choice and free-form reasoning, we require a nonempty and nontruncated response with inspectable reasoning and an extractable final answer in the specified format. We reject normalized duplicates.

3.3 RESELECT

The Reselect stage compares the retained initial winner with the admissible alternatives under the same candidate-blind criteria used during Verify.

If at least one new candidate passes screening, VRR forms

$$\mathcal{C}_{\text{reselect}} = \{c_{i^*}\} \cup \mathcal{R}_{\text{pass}} \subseteq \mathcal{C}_{\text{init}} \cup \mathcal{R}_{\text{pass}}, \qquad 2 \leq |\mathcal{C}_{\text{reselect}}| \leq 4. \tag{19}$$

We exclude the other initial candidates to reserve the evaluation budget for verification-guided alternatives. We retain c_{i^*} so that repair does not remove the initial winner from consideration. Thus, candidate generation expands the search space, but the active reselection pool need not contain every initial candidate or be larger than the initial pool. This set inclusion does not guarantee that reselection will choose the winner when it is correct.

We reuse the evaluation criteria $\Gamma(x)$ created before inspecting the initial candidates and change only the candidate set. For every candidate in $\mathcal{C}_{\text{reselect}}$, we repeat batched judgments, adaptive additional judgments, and pairwise comparisons in both presentation orders.

Algorithm 1 summarizes the complete inference procedure.

4 EXPERIMENTS

We first evaluate the feasibility claim that motivates VRR: whether verification-guided candidate expansion can recover a correct solution when every candidate in the initial pool is incorrect. We then report aggregate accuracy against fixed-pool LAV selection to measure the net effect of recovery and errors on problems already covered by the initial pool. We use Gemma 4 31B IT (Gemma Team, 2026) and Qwen3-32B (Qwen Team, 2025); within each experiment, the same model performs generation, verification, repair, and reselection. The code benchmarks are LiveCodeBench (Jain et al., 2025), HumanEval+, and MBPP (Liu et al., 2023); the reasoning benchmarks are MMLU-Pro (Wang et al., 2024b), GPQA Diamond (Rein et al., 2024), AIME 2025 (Zhang & Math-AI Team, 2025), AMC 2023 (Math-AI, 2025), and MATH-500 (Lightman et al., 2024). Initial pools contain five candidates for each code problem and three for each reasoning problem, and all methods receive the same frozen pool. The Single baseline returns its first candidate, whereas LAV selects from the pool without additional generation. Correctness labels, reference answers, and private tests are withheld until the final answer is fixed. Detailed settings and evaluation rules are provided in Appendix A, and the generation, verification, repair, and LAV prompt templates are provided in Appendix B.

4.1 LIVECODEBENCH: RECOVERY BEYOND THE INITIAL POOL

In Figure 2, we stratify all 1,055 LiveCodeBench problems according to whether all candidates in their five-candidate pools are incorrect, some but not all are correct, or all are correct. On all-incorrect pools, where every fixed-pool selector has zero attainable accuracy, VRR recovers 18 of 117 problems (15.38%) with Gemma and 7 of 215 (3.26%) with Qwen. Each recovery necessarily comes from an additional candidate that passes screening and is selected during reselection. This is the central feasibility result: diagnostic evidence produced during verification can guide generation of a correct option outside the initial pool.

Recovery alone does not imply that aggregate accuracy must increase. The initial-pool oracle covers 938 problems for Gemma and 840 for Qwen, corresponding to accuracies of 88.91% and 79.62%, respectively. VRR produces 938 and 839 correct final outputs, corresponding to accuracies of 88.91% and 79.53%, respectively. Thus, Gemma's 18 all-incorrect-pool recoveries are balanced by 18 final

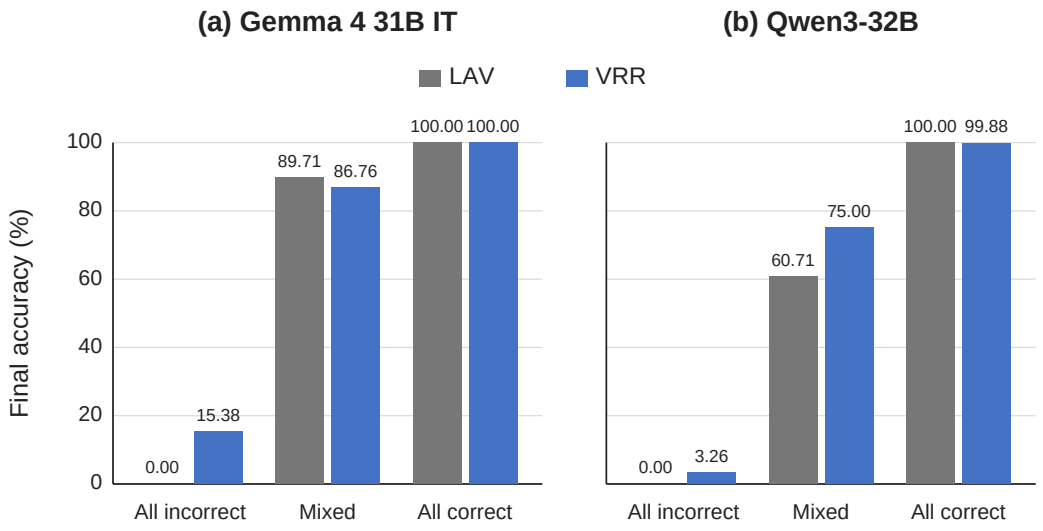

Figure 2: Final accuracy (%) on LiveCodeBench, grouped by the composition of the five-candidate initial pool. The all-incorrect stratum isolates recoveries that no fixed-pool selector can obtain.

Table 1: Accounting of LiveCodeBench recovery beyond the initial pool and final errors on problems covered by the initial-pool oracle. "Covered errors" counts final VRR errors among problems with at least one correct initial candidate; it does not by itself identify a repair-induced regression.

Model	All-incorrect	Recovered	Covered errors	Oracle correct	VRR correct
Gemma 4 31B IT	117	18	18	938	938
Qwen3-32B	215	7	8	840	839

errors on covered problems, while Qwen's seven recoveries fall one short of eight covered-problem errors. These covered-problem errors are complete-system outcomes and are not necessarily caused by repair: the initial selector may already have chosen an incorrect candidate from a covered pool. On mixed pools, VRR is correct on four fewer problems than LAV with Gemma and four more with Qwen; these differences likewise do not identify problem-level transitions from the initial winner. The single Qwen error on an all-correct pool is, however, a genuine correct-to-incorrect regression. Retaining the initial winner therefore preserves it as an option but does not guarantee non-decreasing accuracy under imperfect reselection.

4.2 Aggregate Accuracy Across Eight Benchmarks

Table 2 reports final-answer accuracy on the shared frozen pools. With Gemma, VRR outperforms LAV on four benchmarks, ties on three, and underperforms on one; its largest gains are 1.33 points on LiveCodeBench and 1.32 on MBPP, while HumanEval+ decreases by 0.61. With Qwen, VRR outperforms LAV on five benchmarks, ties it on one, and underperforms it on two; the largest gains are 4.04 points on GPQA Diamond and 2.50 on AMC, while its accuracy is lower by 0.27 points on MBPP and 0.60 points on MATH-500. Overall, VRR has higher point accuracy than LAV in nine of 16 settings, ties it in four, and is lower in three. These point estimates show that candidate expansion can improve complete-system accuracy in some settings, but they are not evidence of uniform superiority. They also motivate separating the ability to generate useful candidates from the reliability of final reselection.

Table 2: Accuracy (%) on frozen initial pools. Bold marks the highest accuracy, including ties; Δ is VRR minus LAV in percentage points.

	Gemma 4 31B IT				Qwen3-32B			
Benchmark	Single	LAV	VRR	Δ	Single	LAV	VRR	Δ
LiveCodeBench	83.22	87.58	**88.91**	+1.33	78.10	78.58	**79.53**	+0.95
HumanEval+	95.12	**95.73**	95.12	-0.61	82.93	84.76	**85.37**	+0.61
MBPP	82.01	81.75	**83.07**	+1.32	75.40	**76.46**	76.19	-0.27
MMLU-Pro	85.70	85.65	**85.75**	+0.10	75.05	75.80	**76.80**	+1.00
GPQA Diamond	78.79	**80.30**	**80.30**	0.00	52.53	53.03	**57.07**	+4.04
AIME	73.33	**76.67**	**76.67**	0.00	**23.33**	**23.33**	**23.33**	0.00
AMC	**97.50**	**97.50**	**97.50**	0.00	70.00	75.00	**77.50**	+2.50
MATH-500	98.60	98.40	**98.80**	+0.40	87.00	**89.20**	88.60	-0.60

5 Related Work

Test-time scaling and candidate search. Test-time methods improve coverage by sampling reasoning paths or solutions (Wang et al., 2023; Brown et al., 2024). Search-based methods instead alternate generation and heuristic evaluation (Yao et al., 2023). Adaptive policies can outperform uniform best-of-N sampling because each problem benefits from a different compute allocation (Snell et al., 2025). These methods motivate VRR's new-approach candidate, but VRR does not expand the pool by unconditional resampling: structured verification results determine when expansion is triggered and provide problem-specific failure evidence to the generator.

Verifiers and LLM judges. Outcome verifiers score final answers, while process verifiers score intermediate steps (Cobbe et al., 2021; Lightman et al., 2024). Other methods use language models for backward verification or train generative verifiers to produce judgments with rationales (Weng et al., 2023; Zhang et al., 2025). LLM-as-a-Verifier (LAV) is the closest method to our selection setting. It ranks a fixed candidate pool with fine-grained scores, repeated evaluations, and decomposed criteria (Kwok et al., 2026). VRR changes the use of those judgments: it preserves their ranking role but also reuses their diagnostic content to generate additional candidates before reselection. It also uses an explicit UNKNOWN state for unresolved claims and compares the leading candidates in both presentation orders to reduce position bias (Wang et al., 2024a).

Feedback-guided refinement and repair. Self-Refine and Reflexion use natural-language feedback to improve model outputs iteratively (Madaan et al., 2023; Shinn et al., 2023). CodeRL and Self-Debugging use execution feedback to guide code regeneration (Le et al., 2022; Chen et al., 2024). These methods demonstrate that feedback can improve later attempts. Our focus is the complementary question raised by verifier-based inference: how to reuse evidence originally produced for candidate ranking as a signal for candidate improvement. VRR retains the selected candidate, generates alternatives with complementary roles, and reselects under the original criteria rather than treating a revision as an automatic replacement.

6 Conclusion

We introduced LLM-as-an-Improver and proposed VRR, which reuses verification evidence to generate additional candidates before final reselection. The all-incorrect-pool results establish the central feasibility claim: VRR can recover correct solutions even when every candidate in the initial pool is incorrect. These recovered solutions lie beyond the reach of any method that only ranks the fixed initial pool. Across two models and eight benchmarks, VRR also achieves higher point accuracy than fixed-pool LAV selection in many settings, although the gains are not uniform. Thus, verification evidence can serve as a generation signal beyond the initial pool, while useful candidate generation does not by itself guarantee an aggregate improvement under imperfect reselection.

AI USE STATEMENT

Generative AI assisted the revision of organization, prose, notation, mathematical explanations and proofs, result interpretation, bibliography, and vector figures. This editing used existing measurements and produced no new experiments. Authors are responsible for checking the final manuscript against the experimental implementation.

REFERENCES

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

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. In *International Conference on Learning Representations*, 2025.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. LLM-as-a-verifier: A general-purpose verification framework. *arXiv preprint arXiv:2607.05391*, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. doi: 10.52202/068431-1549.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0943.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, 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. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2019.

Math-AI. AMC 2023 benchmark dataset, 2025. URL https://huggingface.co/datasets/math-ai/amc23.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof Q&A benchmark. In *First Conference on Language Modeling*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0377.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9440–9450. Association for Computational Linguistics, 2024a. doi: 10.18653/v1/2024.acl-long.511.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. In *Advances in Neural Information Processing Systems*, volume 37, 2024b. doi: 10.52202/079017-3018.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2550–2575. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.167.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0517.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Seyed Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations*, 2025.

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

A Detailed Experimental Configuration

A.1 Models and Initial Candidates

We run Gemma 4 31B IT (Gemma Team, 2026) and Qwen3-32B (Qwen Team, 2025) with vLLM in bfloat16 and disable thinking mode. We sample initial candidates with temperature 1.0, top-p 0.95, and a maximum output length of 4,096 tokens. We generate five candidates per code problem and three per reasoning problem. All methods share the same initial pool within each model.

A.2 Runs, Randomness, and Reporting

Each model–benchmark cell in Table 2 is one end-to-end run on a frozen initial candidate pool; the reported value is not an average across independent end-to-end runs. The initial pool is generated once per model and shared by Single, LAV, and VRR, so comparisons within a cell hold initial candidate coverage fixed. Within an end-to-end run, repeated verifier judgments are aggregated as specified by each method and should not be interpreted as independent experimental replications. We use stochastic sampling for initial generation and repair with the temperatures and top-p values reported below, and we fix the VRR reselection seed to 0. Accordingly, we report point estimates without standard deviations or confidence intervals and make no claim of statistical significance or consistency across seeds.

A.3 BENCHMARKS

We use LiveCodeBench (Jain et al., 2025) and two EvalPlus code benchmarks: HumanEval+ and the extension of Mostly Basic Python Problems (MBPP) (Liu et al., 2023). We use MMLU-Pro (Wang et al., 2024b), GPQA Diamond (Rein et al., 2024), the 2025 American Invitational Mathematics Examination (AIME) (Zhang & Math-AI Team, 2025), the 2023 American Mathematics Competitions (AMC) (Math-AI, 2025), and MATH-500 (Lightman et al., 2024) for reasoning.

A.4 VRR CONFIGURATION

We set the initial verification budget to $B_{\text{init}} = 64$ and the reselection budget to $B_{\text{reselect}} = 96$. The initial selector retains the top two candidates. We set $\epsilon = 10^{-6}$ and use soft criteria only when the hard-score difference is at most $\tau_{\text{soft}} = 0.01$. We set $\tau_{\text{sat}} = \tau_{\text{viol}} = 0.8$, $\tau_{\text{score}} = \tau_{\text{unk}} = 0.5$, and $K_{\text{unk}} = 3$. When triggered, the repair stage generates three candidates: one from the selected candidate, one from the runner-up, and one from a new approach. We use temperature 0.7, top-p 0.95, and a maximum output length of 8,192 tokens for repair. We fix the reselection seed to 0.

A.5 LAV CONFIGURATION

LAV assigns continuous scores to candidate pairs and ranks them with a Probabilistic Pivot Tournament (Kwok et al., 2026). For an evaluated pair (c_i, c_ℓ) and criterion o_j, the verifier returns a distribution over the ordered score tokens $V_{\text{score}} = \{v_1, \ldots, v_G\}$. Let $\widetilde{p}^{(t)}_{i|\ell,j}(v_g)$ denote the distribution extracted for c_i in repetition t; its conditioning includes the task, both candidates, their presentation order, and the criterion. The corresponding expected score is

$$s^{(t)}_{i|\ell,j} = \sum_{g=1}^{G} \widetilde{p}^{(t)}_{i|\ell,j}(v_g)\phi(v_g), \tag{20}$$

where $\phi(v_g)$ maps each token to its ordered numerical value. Given J evaluation criteria and K repeated evaluations, LAV aggregates the pair-conditioned expectations as

$$R_{i|\ell} = \frac{1}{JK} \sum_{j=1}^{J} \sum_{t=1}^{K} s^{(t)}_{i|\ell,j}. \tag{21}$$

Here, J is the number of evaluation criteria, K is the number of repeated evaluations per pair and criterion, and G is the number of ordered score tokens. We normalize the averaged scores to $[0, 1]$ before computing pairwise preferences. This notation follows Kwok et al. (2026). Thus, $K = 4$ means that the verifier judges each pair four times under each criterion, averages those scores, and then averages across the J criteria. Repeated evaluation is an aggregation choice; any variance reduction depends on the dependence among calls. It does not change the number of candidates, pivots, or repair iterations.

The Bradley–Terry model converts the averaged scores into pairwise preference probabilities. The tournament first evaluates candidates along a random ring. This ring places each candidate once in each presentation position. The tournament then selects the top k candidates as pivots. It compares every non-pivot with every pivot and also compares all pivot pairs. For N candidates, the procedure uses

$$N + k(N - k) + \binom{k}{2} \tag{22}$$

pairwise comparisons, which scales as $\mathcal{O}(Nk)$ for $k \ll N$. This expression counts candidate pairs, not model calls or tokens across criteria and repetitions. The method sums each candidate's preference mass and divides by its number of comparisons. It returns the candidate with the highest normalized mass.

We use three criteria, $k = 2$ pivots, and $K = 4$ evaluations per candidate pair and criterion.

A.6 INFORMATION AVAILABLE TO EACH METHOD

For code tasks, we provide only the problem statement, compilation results, and public tests. For reasoning tasks, we provide only the problem statement, available answer choices, and required

Table 3: Mean recorded runtime (seconds per problem). LAV and VRR exclude shared initial generation. Values are descriptive operational records, not a controlled efficiency comparison.

Model	Dataset	Initial generation	LAV	VRR
Gemma 4 31B IT	LiveCodeBench	39.52	156.31	108.91
Gemma 4 31B IT	MMLU-Pro	38.19	47.20	42.21
Gemma 4 31B IT	GPQA Diamond	57.08	70.79	51.05
Gemma 4 31B IT	AIME	101.85	104.12	86.90
Qwen3-32B	LiveCodeBench	36.21	169.31	69.08
Qwen3-32B	MMLU-Pro	37.35	58.42	31.89
Qwen3-32B	GPQA Diamond	63.14	74.38	44.13
Qwen3-32B	AIME	102.38	87.05	75.92

answer format. We reveal correctness labels, reference answers, and private tests only after the method fixes its final answer.

A.7 Evaluation

For code tasks, we mark a candidate correct only if it passes every benchmark test. For reasoning tasks, we normalize the final answer and compare it with the reference label. For MATH-500 (Lightman et al., 2024), `math_verify` checks mathematical equivalence.

A.8 Recorded Operational Runtime

Table 3 reports recorded wall-clock runtime as operational context only. The measurements were collected under runs with different batching, caching, and parallelization behavior, and therefore do not support a controlled claim about relative latency or compute efficiency. They are not used as evidence for VRR's effectiveness.

B Prompt Templates Used in the Experiments

This section records the prompts used for generation, verification, repair, and the LAV baseline. Text in angle brackets denotes a value inserted at run time. Evidence catalogs and audit ledgers are serialized deterministically from the corresponding task record. Line wrapping is changed for presentation, but the instructions and output formats are preserved. Some experiment runners used the earlier internal name "FOVEA" in a header; we replace that header by "VRR" below because the name itself carries no instruction or task information. No prompt contains a reference answer, correctness label, private-test result, or candidate reward.

B.1 Initial Candidate Generation

For HumanEval+ and MBPP, the user prompt is:

````
Solve the following <BENCHMARK> Python programming task.
Return one complete executable Python solution inside exactly one fenced
```python block. Preserve the requested function name and interface.
Do not include tests or explanations outside the code block.

<PROBLEM>
````

For LiveCodeBench, we use the benchmark question followed by either its starter code or a standard-input/standard-output instruction, and end with:

```
### Answer: (use the provided format with backticks)
```

For GPQA Diamond, the system prompt is "You are an expert graduate-level science reasoner." For AIME, AMC, and MATH-500, it is "You are an expert competition mathematician." The corresponding user template is:
```

```
<QUESTION OR PROBLEM, INCLUDING OPTIONS WHEN APPLICABLE>

Solve independently and show a complete derivation.
<FINAL-ANSWER FORMAT INSTRUCTION>
```

The final-answer instruction requires a separate `FINAL_ANSWER: X` line for multiple-choice tasks and a separate `FINAL_ANSWER: \boxed{answer}` line for free-form mathematics tasks.

B.2 Candidate-Blind Criterion Generation

Task-specific evaluation items are generated before any candidate is shown. Human-authored dataset-level items are supplied in `<FIXED OBLIGATIONS>` and remain part of the final contract.

```
You compile a candidate-blind success contract. You have NOT been
given any candidate answer or trajectory. Derive only atomic conditions
entailed by the task plus universal execution invariants. Fatal
requirements are hard; preferences are soft. Return JSON only:
{"obligations":[{"id":"snake_case","kind":"hard|soft",
"predicate":"one independently auditable condition",
"support":["required evidence types"],
"counterexample":["concrete falsifiers"],
"dependencies":[],"prior":0.5}]}
Do not reward effort, narration, or style. Include post-final-change
verification when the task changes executable state. The following
human-authored fixed conditions are already part of the final AND
contract and will be audited in parallel. You may overlap with them
when a task-specific clause is useful:
<FIXED OBLIGATIONS>
Return at most <MAX> concise task-specific obligations.
Compiler seed: <SEED>

TASK:
<PROBLEM>
```

The fixed code criteria ask about specification and interface adherence, functional correctness including boundary cases, and visible error signals such as syntax errors, runtime failures, failed public checks, or infeasible complexity. The fixed reasoning criteria ask about instruction and answer-format adherence, correctness of material factual, mathematical, and logical steps, and whether the reasoning entails the final answer.

B.3 Batched Initial Verification

The initial pass audits all items for one candidate in one call. The prompt version used by the implementation is `batched-candidate-svu-v3`.

```
You are a skeptical falsification auditor. Audit EVERY atomic
obligation for ONE candidate in this single response, using only that
obligation's allowed evidence records. Agent claims are not proof.
SAT needs concrete cited support. VIOL needs a concrete cited
counterexample. Missing, ambiguous, pre-change, invalidly cited, or
unavailable evidence is UNKNOWN. Never compare against another
candidate and never infer an unrecorded event.

TASK:
<PROBLEM>

AUDIT MODE: <MODE>

EVIDENCE CATALOG (id | type | exact raw span | time):
<EVIDENCE CATALOG>

<ONE BLOCK PER OBLIGATION, CONTAINING ITS ID, KIND, PREDICATE,
 EXPECTED SUPPORT, FALSIFIERS, AND ALLOWED EVIDENCE IDS>
```

```
Return exactly one block for every index, in ascending order. Cite only
ids listed for that obligation. Each verdict letter must be generated
in the response so its generated-position logits define the S/V/U
distribution.
<audit index="0"><reason>brief reason</reason>
<evidence>E000001</evidence><verdict>S</verdict></audit>
Use S=SAT, V=VIOL, U=UNKNOWN. Do not omit, duplicate, or add indexes.
```

B.4 Adaptive Atomic Verification

When the acquisition rule requests another judgment, VRR uses the following single-item prompt (implementation version `atomic-svu-v4`):

```
You are a skeptical falsification auditor. Decide one atomic success
obligation using ONLY the cited evidence records below. Agent claims
are not proof. SAT requires concrete supporting evidence. VIOL
requires a concrete counterexample. Missing, ambiguous, pre-change,
or uncited evidence is UNKNOWN. Never infer an event that is not present.

TASK:
<PROBLEM>

ATOMIC OBLIGATION [<ID>, <KIND>]:
<PREDICATE>

Expected support: <SUPPORT>
Concrete falsifiers: <FALSIFIERS>
Audit mode: <MODE>

EVIDENCE RECORDS (id | type | exact raw span | time):
<EVIDENCE RECORDS>

Return exactly these tags. Cite only listed ids. Use the one-letter
encoding S=SAT, V=VIOL, U=UNKNOWN so logits are a three-way distribution.
<reason>brief evidence-grounded reason</reason>
<evidence>E000001,E000002</evidence>
<verdict>S|V|U</verdict>
```

The implementation obtains $\boldsymbol{q}_{ij}^{(t)}$ from the generated-position log probabilities of the final verdict token. Unrecognized and omitted top-k mass is assigned to UNKNOWN, and the three components are normalized to sum to one. A SAT or VIOL verdict without valid retrieved evidence is replaced by UNKNOWN.

B.5 Pairwise Reselection Prompt

Finalists are compared in both presentation orders with prompt version `paired-evidence-abt-v3`:

```
You are a skeptical final comparison auditor. Compare two candidates
for one atomic success obligation using ONLY the evidence records
below. A and B are anonymous positions, not quality hints. Agent
claims are not proof. Missing, ambiguous, pre-change, or uncited
evidence means T (tie/unknown). Never infer an event that is not present.

TASK:
<PROBLEM>

ATOMIC OBLIGATION [<ID>, <KIND>]:
<PREDICATE>

Expected support: <SUPPORT>
Concrete falsifiers: <FALSIFIERS>
Orientation: <ORIENTATION>
```

```
CANDIDATE-ONLY EDIT DELTA:
<DELTA>

OBLIGATION/DELTA ALIGNMENT:
<ALIGNMENT>

CANDIDATE A EVIDENCE:
<A-PREFIXED EVIDENCE>

CANDIDATE B EVIDENCE:
<B-PREFIXED EVIDENCE>

Return exactly these tags. Cite only listed candidate-prefixed ids and
cite at least one A record and one B record for an A/B preference.
A means candidate A has stronger evidence, B means candidate B, and
T means tie/unknown.
<reason>brief evidence-grounded comparison</reason>
<evidence>A:E000001,B:E000001</evidence>
<verdict>A|B|T</verdict>
```

B.6 Repair Generation

Each Repair request receives one source candidate, except for a from-scratch variant when no parent is supplied, together with a compact ledger containing the criteria, initial ranking, scores, up to four strengths, up to four violation hypotheses, and up to two unresolved items per candidate. Code tasks use the following common wrapper:

```
You are generating one additional candidate for a label-blind VRR
portfolio-repair experiment. Private tests, candidate rewards, and
answer labels are unavailable. Return exactly one complete Python
solution inside a fenced ```python block and no other fenced block.

Role: <ROLE>
<ROLE-SPECIFIC INSTRUCTION>

Problem specification:
<PROBLEM>

<PARENT CANDIDATE OR "NO PARENT CANDIDATE IS SUPPLIED">

Public-evidence audit ledger:
<COMPACT LEDGER>

Requirements:
- Preserve the required interface and input/output contract.
- Handle boundary cases and the full stated constraints.
- Do not rely on private tests or candidate identities.
- Treat semantic audit claims as hypotheses, not ground truth.
- Before coding, privately establish the invariant and complexity bound.
- Check minimum, maximum, degenerate, and off-by-one cases.
- The result must compile and pass all designated public cases.
```

The three role-specific instructions are:

```
SELECTED-CANDIDATE REPAIR:
Repair the parent while preserving every sound part. Privately derive
at least one concrete edge case for every applicable audit hypothesis,
then correct the root cause rather than its symptom.

RUNNER-UP REPAIR:
Repair the alternative parent after privately stress-testing its
invariants, asymptotic complexity, and boundary cases. Preserve sound
parts, but switch algorithms if the proof does not hold.
```

```
NEW-APPROACH SOLUTION:
Use the parent only as a bug-finding target. Privately derive a minimal
valid counterexample or boundary family on which it may fail, then
reimplement the solution from a correctness argument. Prefer an
algorithmically independent formulation and do not copy the parent's
control flow mechanically.
```

For reasoning tasks, the common output portion is instead:

```
You are generating one additional candidate in a label-blind VRR
portfolio-repair experiment. The reference answer, correctness labels,
and benchmark evaluator are unavailable. Produce a complete,
self-contained solution.

Role: <ROLE>
<ROLE-SPECIFIC INSTRUCTION>

Problem:
<PROBLEM>

Parent candidate:
<PARENT>

Audit ledger (semantic audit claims are hypotheses, not ground truth):
<COMPACT LEDGER>

Requirements:
- Check the full problem and all answer choices or domain constraints.
- Do not vote among candidates or infer a hidden reference answer.
- Give enough reasoning that the final conclusion can be audited.
- <DATASET-SPECIFIC FINAL-ANSWER FORMAT>
```

The role instructions tell the model, respectively, to preserve sound reasoning while correcting ledger-indicated root causes, to independently check every pivotal fact and inference in the runner-up, or to use the parent only as a bug-finding target and solve again from first principles.

B.7 LAV Pairwise Scoring Prompt

The LAV runs use the frozen legacy prompt. The complete task and both trajectories are inserted before the pre-specified criterion:

```
You are an expert evaluator of AI coding agents. You will see a task
description and two agent trajectories. Your job is to evaluate them
on ONE specific criterion: <CRITERION NAME>.

<GROUND-TRUTH AVAILABILITY NOTE>

Task:
<PROBLEM>

Trajectory A:
<CANDIDATE A>

Trajectory B:
<CANDIDATE B>

Evaluation Guideline -- <CRITERION NAME>:
<CRITERION DESCRIPTION>

Score each trajectory ONLY on this specific criterion. Ignore other
aspects that are not relevant to the criterion.

Rating Scale:
```

```
A = clearly and completely succeeded with verified output (best)
B-D = succeeded with only minor issues
E-G = above average, mostly correct with some issues
H-J = uncertain, leans toward success
K-M = uncertain, leans toward failure
N-P = below average, significant issues remain
Q-S = failed with some partial progress
T = clearly and completely failed (worst)

Then output your final scores:
<score_A>LETTER_A_TO_T</score_A>
<score_B>LETTER_A_TO_T</score_B>

Begin your analysis now.
```

The expected score is computed from the token-level distribution over A–T, not from only the emitted letter. As described above, we use three criteria, two pivots, and four repetitions per pair and criterion.

C A Concrete All-Incorrect Recovery Example

We provide a concrete LiveCodeBench example in which VRR produces a correct solution despite all candidates in the initial pool being incorrect. The task asks for the largest value obtainable from an array by repeatedly merging two adjacent values when the left value is no larger than the right value. For example, the optimum for $[2, 3, 7, 9, 3]$ is 21: merging 7 into 9 first yields $[2, 3, 16, 3]$, after which 3 and then 2 can be absorbed into 16.

All five initial candidates were incorrect under the benchmark tests. Despite being sampled independently, they implemented essentially the same left-to-right greedy rule:

```
running_sum = nums[0]
answer = nums[0]
for x in nums[1:]:
    if running_sum <= x:
        running_sum += x
    else:
        running_sum = x
    answer = max(answer, running_sum)
```

On the example above, this rule prematurely forms $2 + 3 + 7 = 12$, which cannot be merged into 9, and therefore returns 12 rather than 21. Because LAV can only rank the five fixed candidates, it cannot recover on this problem.

The initial verification stage found that every candidate produced an incorrect answer on one of the two official public cases. The resulting Failure Ledger preserved two distinct facts: the required class and method declarations were present, but functional correctness was violated because the left-to-right accumulation did not realize the optimal merge order. Since every candidate fell below the repair threshold, VRR triggered candidate expansion and used the highest-ranked initial candidate as one repair parent.

During repair, the model explicitly reconsidered the order of operations: merging 5 into 7 produces 12 and blocks the subsequent merge into 9, whereas merging 7 into 9 first produces 16 and then permits the values to its left to be absorbed. It therefore changed the traversal direction and generated the following core algorithm:

```
current = nums[-1]
answer = current
for i in range(len(nums) - 2, -1, -1):
    if nums[i] <= current:
        current += nums[i]
    else:
```

```
        current = nums[i]
    answer = max(answer, current)
return answer
```

This right-to-left invariant represents the value already constructed to the right of position i; whenever `nums[i]` is no larger, it can be absorbed into that value.

The repaired candidate passed compilation and both public cases, was admitted to the expanded pool, and was selected during final reselection. The private tests, evaluated only after selection, confirmed that the repaired candidate was correct.