CODEVERIFIER: UNIFIED GENERATIVE VERIFICATION ACROSS CODE GENERATION AND DEVELOPMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Code generation for competitive programming and repository level software engineering can be viewed as a sequence of decisions over code states such as complete programs ongoing edits and repository patches. Reinforcement learning with verifiable rewards can improve these decisions beyond finite demonstration data provided that the policy receives a reliable reward before selecting its next action. This condition is difficult to meet at scale since short test prefixes leave many candidates tied deeper execution pushes latency into a long tail and intermediate edits often lack a testable entry point while querying a larger general purpose model at every step is too costly. We turn this missing feedback into a reusable signal with CODEVERIFIER, a generative verifier that judges code states before final testing and outputs a rationale repair relevant code regions and a verdict. Supervision comes from two kinds of history where execution outcomes indicate whether a state succeeds and successful repairs indicate where a failed state must change. A two stage procedure first projects this evidence into structured targets and then optimizes verdicts and regions with separate field wise signals. CODEVERIFIER outperforms size matched code judging baselines across competitive programming and repository code by 3.2 points on average. On LiveCodeBench residual errors, one 9B judgment matches the mean detection rate of 8 additional tests; on the hardest TACO slice, matched-recall execution is 6.5× slower even with idealized 64-way parallelism. Used as a reward it improves candidate selection distinguishes successful from unsuccessful edits and raises final task success which shows that structured early judgments can fill the feedback vacuum and replace many repeated executions. The training algorithm and released weights are available at `https://anonymous.4open.science/r/CodeVerifier`.

1 INTRODUCTION

Code generation has become a central capability for solving algorithmic problems and real software tasks. Competitive programming focuses on algorithms and boundary cases and requires programs that pass an online judge. Repository-level software engineering further involves build dependencies and multi-file interactions and requires changes that pass project tests.

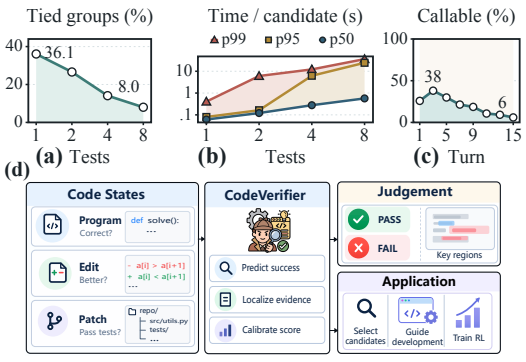

Figure 1: (a) Short prefixes tie groups; (b) deeper execution is long-tailed; (c) intermediate turns lack entry points; (d) the verifier gives early feedback.

Although the code structures and execution environments differ, both tasks can be framed as sequential decisions over code states. At each step an agent observes the task, the code, and the interaction history, then performs a submission, an edit, or a tool operation and transitions to a new state. Complete programs, ongoing edits, repository patches, and interaction traces represent different state observations within this process (Li et al., 2022; Jain et al., 2025c; Jimenez et al., 2024; Miserendino et al., 2025; Yang et al., 2024; Wang et al., 2025b). When code generation is treated as sequential decision making reinforcement learning with verifiable rewards offers a natural

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.

path beyond finite demonstration data. The space of possible code states and modification paths exceeds what any training set can cover. Errors may surface only under a particular input dependency or interaction order and many states met at deployment never appear in offline data. RLVR enables a policy to explore these states and obtain an automatically checkable signal from the task outcome. CodeRL RLTF and RLEF train code policies from execution feedback while later work extends verifiable reward training to stronger reasoning models and software engineering agents (Le et al., 2022; Liu et al., 2023a; Gehring et al., 2025; Guo et al., 2025; Da et al., 2025). Code tasks fit the paradigm especially well because tests builds and project level evaluations can confirm success without a human preference label on every candidate. RLVR can thus convert limited demonstrations into continual policy improvement provided the policy receives a reliable reward before acting.

Scaling RLVR exposes the problem of missing feedback in two settings. Figure 1 summarizes the evidence: (a) on LiveCodeBench candidate groups, short test prefixes leave many groups tied and correctness ambiguous; (b) on the same candidates, deeper execution pushes latency into the long tail; and (c) in multi-turn CollabLLM trajectories from BigCodeBench, many intermediate states still lack the callable entry point required by final testing. Limited coverage, long-tailed execution, and unevaluated edits make early decisions difficult, while repeatedly querying a larger general-purpose model is too costly for large-scale training (Chen et al., 2023; Ma et al., 2025a).

Figure 1(d) shows how to turn this missing feedback into a reusable signal. The same structured judgment can select among complete programs, assess whether an intermediate edit or repository patch is moving toward success, and provide a reward for policy learning. To realize this design, we introduce CODEVERIFIER, a unified generative verifier for code states. CODEVERIFIER produces a structured judgment consisting of a rationale, repair-relevant code regions, and a verdict, rather than returning only a binary score. The same interface covers complete programs, intermediate edits, and patch-linked repository views, and it can assess interaction traces whose final executable state has not yet formed. Training proceeds in two stages. We first let a teacher propose candidate judgments, then project them onto targets constrained by execution outcomes and by the regions changed in subsequent successful repairs. We next apply Region-aware Verifier Policy Gradient, which routes global verdict feedback and local region feedback to their corresponding output fields. Execution histories supervise whether a state succeeds, while repair histories supervise where a failed state is associated with a change. The resulting verifier can be queried before final testing to support candidate selection, editing decisions, and policy learning. We show that structured early judgments can fill the feedback vacuum in RLVR and replace many repeated executions in the early search and training loop:

- We formulate a unified code-state verification framework and a two-stage training procedure that combines execution outcomes with successful repair records to learn both global verdicts and local repair regions. Field-wise online optimization prevents global and local signals from cancelling inside a shared scalar reward.
- Across competitive programming and repository code, CODEVERIFIER improves verdict accuracy, repair-region alignment, editing-state prediction, and error detection after limited testing. It outperforms size-matched code-judging baselines across model scales and code sources by 3.2 percentage points on average, while a single judgment detects residual errors at the level of multiple additional tests and reduces execution delay.
- We show that the structured judgments translate into code decisions. They improve pairwise comparison and within-task candidate selection, distinguish score changes associated with successful and unsuccessful edits, and provide policy rewards that raise final task success in both competitive programming and repository repair.

2 FORMALIZATION AND SUPERVISION FOR CODE VERIFICATION

This section first defines the visible code states and their task-specific outcomes, then introduces the structured verifier output and its readouts. We finally specify how execution outcomes and successful repairs form supervision and how tasks are split for training and evaluation.

2.1 CODE STATES AND VERIFICATION OBJECTIVES

We represent code generation and revision as a state process $(\boldsymbol{S}_t)_{t\geq 0}$, where $\boldsymbol{S}_t \in \mathcal{S}$ contains the task, code, and interaction history available at time t. Let $g \in \mathcal{G} = \{\mathsf{sol}, \mathsf{turn}, \mathsf{prefix}, \mathsf{repo}\}$ denote

a complete program, an editing state, a trajectory prefix, or a patch-linked repository view. An observation map $\mathcal{R}_g : \mathcal{S} \to \mathcal{Z}_g$ constructs the formatted visible record presented to the verifier; $\boldsymbol{Z}_g$ denotes the corresponding random input and $\boldsymbol{z}_g$ one realized record, not a learned representation. Let $Y_g \in \{0, 1\}$ denote the binary success indicator associated with input form g, then we have

$$\boldsymbol{Z}_g = \mathcal{R}_g(\boldsymbol{S}_t), \qquad q^g(\boldsymbol{z}_g) = \mathbb{P}_g(Y_g = 1 \mid \boldsymbol{Z}_g = \boldsymbol{z}_g), \tag{1}$$

where $\mathbb{P}_g$ is the joint distribution over tasks, visible records, and outcomes induced by data collection, so $q^g(\boldsymbol{z}_g)$ is the corresponding conditional success probability. For complete programs and editing states, Y_g is the success indicator of the current code. For a trajectory prefix, it is the success indicator of the final outcome of the recorded continuation. For a repository patch p, let Y_p be its project-test outcome and $\mathcal{Z}(p) = \{\boldsymbol{z}_{p,h}\}_{h=1}^{H_p}$ its associated views, where H_p is the number of views associated with p. Each view is a numbered candidate-file snapshot, optionally paired with a task requirement, and is supervised by the source patch outcome Y_p. View scores are averaged when a patch-level score is needed for ranking or policy learning. The repair target for one view contains the repair-associated lines visible in that snapshot. Training uses sol and repo; evaluation also examines turn and prefix. Appendix A.2 gives the complete view construction.

The input form g and a verdict interface $\iota \in \mathfrak{I}_g$ jointly determine the output protocol, where $\mathfrak{I}_g$ is the set of interfaces used for form g. Let $\mathcal{V}_{g,\iota}$ be the verdict set and $\mathcal{V}_{g,\iota}^{+}$ its successful outcomes. Training and the main evaluation use $\mathcal{V}_{\mathsf{sol}} = \{\mathrm{AC}, \mathrm{WA}, \mathrm{TLE}, \mathrm{CE}\}$ with $\mathcal{V}_{\mathsf{sol}}^{+} = \{\mathrm{AC}\}$ for competitive programs, and $\mathcal{V}_{\mathsf{repo}} = \{\mathrm{PASS}, \mathrm{FAIL}\}$ with $\mathcal{V}_{\mathsf{repo}}^{+} = \{\mathrm{PASS}\}$ for repository code. AC, WA, TLE, and CE denote accepted, wrong answer, time limit exceeded, and compilation error. Fine-grained classification distinguishes these four outcomes; the binary competitive interface treats AC as success and all other verdicts as failure, denoted by $\iota = \mathsf{bin}$. Repository views instead use PASS and FAIL for the source patch's project-test outcome. Editing states and trajectory prefixes retain the verdict interface defined by their evaluation records. We omit ι when it is unambiguous.

For interfaces with code coordinates, the structured output is $\boldsymbol{y} = (\chi, \mathcal{B}, v)$, with $\mathcal{B} = \big((b_a, \kappa_a)\big)_{a=1}^{N_{\mathrm{reg}}}$. Here, $v \in \mathcal{V}_{g,\iota}$ is the verdict and χ is a concise rationale. The sequence $\mathcal{B}$ contains $N_{\mathrm{reg}} = |\mathcal{B}|$ regions ordered by importance, each consisting of a line interval $b_a = [\ell_a, u_a]$ and an explanation κ_a. These symbols name semantic fields; each interface prompt specifies their serialization. The target region sequence is empty for a successful code state. Trajectory prefixes have no stable code coordinates and therefore use only (χ, v). Regions satisfy $1 \le \ell_a \le u_a \le L^{\mathrm{code}}$, where L^{code} is the number of numbered input-code lines.

During training, $\boldsymbol{\theta}$ denotes the policy parameters; after training, $\boldsymbol{\phi}$ denotes the final checkpoint used for evaluation. For a response $\boldsymbol{y} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{z}_g, \iota)$, $\mathrm{verdict}(\boldsymbol{y}) \in \mathcal{V}_{g,\iota}$ reads its discrete verdict and $\nu_{g,\iota} : \mathcal{V}_{g,\iota} \to [0, 1]$ maps each verdict to a success utility. Repeated judgments use generative utility

$$s_{\boldsymbol{\phi}}^{\mathrm{gen},g,\iota}(\boldsymbol{z}_g) = \mathbb{E}_{\boldsymbol{y} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{z}_g, \iota)}[\nu_{g,\iota}(\mathrm{verdict}(\boldsymbol{y}))]\,. \tag{2}$$

For classification, $\nu_{g,\iota}(v) = \mathbf{1}[v \in \mathcal{V}_{g,\iota}^{+}]$, where $\mathbf{1}[\cdot]$ equals one when its condition holds and zero otherwise. A single judgment supplies the discrete verdict, while repeated judgments compare candidates by mean utility. A complementary continuous readout uses the conditional label probabilities after a fixed verdict-field prefix. Writing the conditional log-probability of label v as $\ell_{\boldsymbol{\phi}}(v; \boldsymbol{z}_g, \iota)$, the normalized success-label score is

$$s_{\boldsymbol{\phi}}^{\mathrm{lab},g,\iota}(\boldsymbol{z}_g) = \frac{\sum_{v \in \mathcal{V}_{g,\iota}^{+}} \exp \ell_{\boldsymbol{\phi}}(v; \boldsymbol{z}_g, \iota)}{\sum_{u \in \mathcal{V}_{g,\iota}} \exp \ell_{\boldsymbol{\phi}}(u; \boldsymbol{z}_g, \iota)}. \tag{3}$$

For a binary interface with success label v^{+} and failure label v^{-}, Equation (3) is the normalized likelihood of v^{+}, equivalently the sigmoid of the label margin $\ell_{\boldsymbol{\phi}}(v^{+}) - \ell_{\boldsymbol{\phi}}(v^{-})$. Here, v^{+} is AC for competitive programs and PASS for repository views. Continuous-score comparisons use this margin; repeated candidate comparisons use Equation (2), and repository policy rewards average normalized PASS scores across a patch's views. Appendix A.7 gives the prompts, readouts, and scoring procedures.

2.2 Execution outcomes and repair evidence

Each training instance is $d_i = \big(g_i, \boldsymbol{z}_i, v_i^{\star}, \Gamma_i^{\star}\big)$, where $g_i \in \mathcal{G}_{\mathrm{train}} = \{\mathsf{sol}, \mathsf{repo}\}$ specifies the input form and $\boldsymbol{z}_i$ contains the task, current code, and visible context. Execution supplies the verdict $v_i^{\star}$.

When a subsequent successful repair is available, $\Gamma_i^\star = \{\gamma_{i,h}^\star\}_{h=1}^{K_i}$ contains the repair-associated intervals visible in $\boldsymbol{z}_i$, with $\gamma_{i,h}^\star = [\ell_{i,h}^\star, u_{i,h}^\star]$. Successful instances have $\Gamma_i^\star = \varnothing$; failed instances used for region supervision have $K_i \geq 1$ (Figure 2).

2.3 Evidence projection and stage assignment

Let π_{teach} denote the teacher model, instantiated as Qwen3.5-397B-A17B (Qwen Team, 2026). For instance i, the teacher generates a candidate set $\mathcal{Y}_i = \{\boldsymbol{y}_{i,j}\}_{j=1}^{J_i}$, with each candidate decomposed into a global rationale $\chi_{i,j}$, a region sequence $\mathcal{B}_{i,j}$, and a verdict $\widehat{v}_{i,j}$. The teacher proposes rationales and candidate regions, while execution outcomes and successful repairs determine which content enters the supervision target.

Repairs can span disjoint intervals, so region quality must reflect both coverage of repaired lines and excess predicted span. Line-level intersection over union (IoU) compares the unions of predicted and repaired lines, denoted by $\Lambda(\mathcal{B})$ and $\Lambda(\Gamma)$,

$$\text{IoU}_{\text{line}}(\mathcal{B}, \Gamma) = \frac{|\Lambda(\mathcal{B}) \cap \Lambda(\Gamma)|}{|\Lambda(\mathcal{B}) \cup \Lambda(\Gamma)|}.$$

We use this line-set IoU to select teacher candidates and evaluate regions. Online training uses one-dimensional generalized IoU (GIoU), which also accounts for distance between intervals.

Let $\mathcal{J}_i^{\text{ok}}$ index teacher candidates with the correct verdict. For a failed instance with visible repair evidence, define the teacher-region quality

$$\eta_i^{\text{teach}} = \max_{j \in \mathcal{J}_i^{\text{ok}}} \text{IoU}_{\text{line}}(\mathcal{B}_{i,j}, \Gamma_i^\star). \tag{4}$$

Let c_i^{cov} indicate that every repair interval overlaps a region from a correctly classified teacher response. The two failed-instance subsets are

$$\mathcal{I}_{\text{SFT}}^- = \{i \mid K_i \geq 1,\ \eta_i^{\text{teach}} > 0.80,\ c_i^{\text{cov}} = 1\}, \quad \mathcal{I}_{\text{online}}^- = \{i \mid K_i \geq 1,\ 0.60 \leq \eta_i^{\text{teach}} \leq 0.80\}. \tag{5}$$

High-overlap instances provide structured targets, whereas intermediate-overlap instances retain the outcome and repair evidence for online optimization. Successful instances are assigned to the two stages by input form and source; those used for SFT contain a teacher response with a correct verdict and no predicted regions.

For a failed SFT instance, the target verdict is fixed to the execution outcome $v_i^\star$. After evidence normalization, retained repair intervals supply coordinates for $\mathcal{B}_i^\star$; teacher explanations follow input-specific selection and support rules. The global rationale $\chi_i^\star$ comes from a selected verdict-correct response. Successful instances use the execution verdict, an empty region target, and the teacher rationale. Evidence projection thus produces the structured target

$$\boldsymbol{y}_i^\star := \Pi_{\text{ev}}(\mathcal{Y}_i; v_i^\star, \Gamma_i^\star) = (\chi_i^\star, \mathcal{B}_i^\star, v_i^\star). \tag{6}$$

Appendix A.4 gives the complete projection rule and stage assignment.

2.4 Training sets and temporal split

Let $\mathcal{I}_{\text{SFT}}$ and $\mathcal{I}_{\text{online}}$ denote instances in the two stages. Both stages contain competitive programs from AtCoder and NowCoder, together with repository code views from SWE-Gym and R2E-Gym (Pan et al., 2025; Jain et al., 2025d).

All splits are disjoint at the competitive-programming problem or repository-task level and follow problem release time or repository-issue creation time. Earlier tasks are used for training, subsequent tasks for validation, and the latest tasks for testing. Teacher candidates are generated after the split is fixed. Appendix A gives the composition and construction procedure (Figure 7).

3 Two-stage training of CodeVerifier

CodeVerifier first learns structured responses projected onto execution and repair evidence. Region-aware Verifier Policy Gradient (RVPG) then assigns global verdict feedback and local region feedback to their corresponding fields (Figure 2). Both stages optimize parameters $\boldsymbol{\theta}$; the final checkpoint $\boldsymbol{\phi}$ supplies the downstream verdict, sampled utility, or label score.

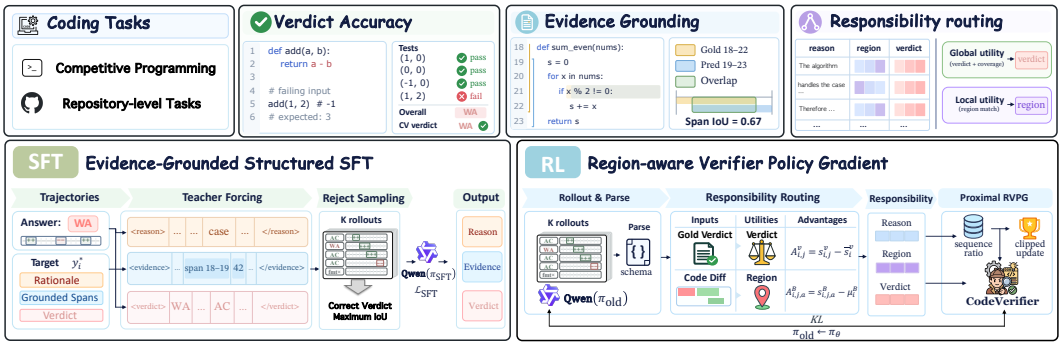

Figure 2: Two-stage training. Execution outcomes and repair regions define SFT targets; online feedback is routed by field; accuracy and line IoU score verdicts and regions.

3.1 Structured Supervision through Evidence Projection

The first stage jointly learns the rationale, regions, and verdict in Equation (6). Let $L_i = |\boldsymbol{y}_i^\star|$. The autoregressive cross-entropy objective is

$$\mathcal{L}_{\mathrm{SFT}}(\boldsymbol{\theta}) = -\frac{1}{\sum_{i\in\mathcal{I}_{\mathrm{SFT}}} L_i} \sum_{i\in\mathcal{I}_{\mathrm{SFT}}} \sum_{t=1}^{L_i} \log \pi_{\boldsymbol{\theta}}(y_{i,t}^\star \mid \boldsymbol{z}_i, \boldsymbol{y}_{i,<t}^\star). \tag{7}$$

Execution outcomes set the verdict target, successful repairs identify repair-associated lines, and teacher responses provide candidate rationales and regions. With the same instances, initialization, output structure, and likelihood objective, teacher-target SFT learns a selected verdict-correct teacher response. Evidence-projected SFT uses the same execution-aligned verdict and projects selected teacher evidence onto retained repair intervals, as defined in Equation (6). This gives the controlled Stage-I comparison in Table 1. The resulting parameters $\boldsymbol{\theta}_{\mathrm{SFT}}$ initialize the online policy, and a frozen copy defines the reference policy π_{ref}.

3.2 Field-wise Online Optimization

Online training uses a classification interface, so we omit ι below. For input $\boldsymbol{z}_i$, the old policy generates $M_i \geq 2$ responses $\boldsymbol{y}_{i,j} = (\chi_{i,j}, \mathcal{B}_{i,j}, \widehat{v}_{i,j}) \sim \pi_{\mathrm{old}}(\cdot \mid \boldsymbol{z}_i)$. The execution outcome evaluates the verdict, while the regions touched by a subsequent successful repair evaluate the local output. Let $\mathbf{C}^{(g)} \in \mathbb{R}_{\geq 0}^{|\mathcal{V}_g|\times|\mathcal{V}_g|}$ be a misclassification cost matrix with zero diagonal. The verdict utility is

$$r_{i,j}^{\mathrm{v}} = \begin{cases} \zeta_v, & \widehat{v}_{i,j} = v_i^\star, \\ -\zeta_v [\mathbf{C}^{(g_i)}]_{v_i^\star, \widehat{v}_{i,j}}, & \widehat{v}_{i,j} \neq v_i^\star, \end{cases} \qquad \zeta_v > 0. \tag{8}$$

Predicted regions are matched one-to-one with repair-associated regions by one-dimensional GIoU (Rezatofighi et al., 2019). Match quality determines each region block's local utility $s_{i,j,a}^{\mathrm{B}}$, with a penalty for unmatched predictions. The global utility $s_{i,j}^{\mathrm{v}}$ combines verdict correctness with region completeness. On failed instances, it penalizes missed repair regions and incomplete coverage of repaired lines; on successful instances, it evaluates whether the region output is empty.

Verdicts and regions describe different parts of a response. A shared scalar cannot express their advantages separately and may cancel them when their signs differ (Appendix Proposition 1). RVPG instead assigns the global and local advantages to their corresponding fields.

Let $N_{i,j} = |\mathcal{B}_{i,j}|$ and $\bar{s}_i^{\mathrm{v}} = \frac{1}{M_i}\sum_{j=1}^{M_i} s_{i,j}^{\mathrm{v}}$ be the group mean of global utility. The local reference μ_i^{B} is the group mean of region utility, optionally blended with a fixed quality level; Appendix A.5 defines it together with the field utilities. Let $\mathcal{T}_{i,j}^{\mathrm{v}}$ denote verdict-token positions and $\mathcal{T}_{i,j,a}^{\mathrm{B}}$ the positions in region block a, with all sets pairwise disjoint. The field-wise token advantage is

$$A_{i,j,t}^{\mathrm{tok}} = \begin{cases} \alpha_B(s_{i,j,a}^{\mathrm{B}} - \mu_i^{\mathrm{B}}), & t \in \mathcal{T}_{i,j,a}^{\mathrm{B}}, \quad a \in \{1,\ldots,N_{i,j}\}, \\ \alpha_v(s_{i,j}^{\mathrm{v}} - \bar{s}_i^{\mathrm{v}}), & t \in \mathcal{T}_{i,j}^{\mathrm{v}}, \\ 0, & \text{otherwise}. \end{cases} \tag{9}$$

The positive scalars α_v and α_B set the two advantage scales. Verdict tokens receive the global advantage, and each region block receives its own matching advantage. The global rationale receives no task advantage and is regularized by a Kullback–Leibler (KL) surrogate.

Let $w_{i,j,t} \in \{0,1\}$ mark non-padding tokens and $n_{i,j} = \sum_t w_{i,j,t} > 0$. Write $h_{i,j,t} = \log \pi_{\boldsymbol{\theta}}(y_{i,j,t} \mid \boldsymbol{z}_i, \boldsymbol{y}_{i,j,<t})$ and $h^{\mathrm{old}}_{i,j,t} = \log \pi_{\mathrm{old}}(y_{i,j,t} \mid \boldsymbol{z}_i, \boldsymbol{y}_{i,j,<t})$. The response-level ratio and its position-wise gradient surrogate are

$$\rho_{i,j} = \exp\left[\frac{1}{n_{i,j}} \sum_t w_{i,j,t}\big(h_{i,j,t} - h^{\mathrm{old}}_{i,j,t}\big)\right], \quad \widetilde{\rho}_{i,j,t} = \mathrm{sg}[\rho_{i,j}] \exp\big(h_{i,j,t} - \mathrm{sg}[h_{i,j,t}]\big). \tag{10}$$

Here, $\rho_{i,j}$ is the geometric mean of non-padding token likelihood ratios and sets a shared clipping scale. The stop-gradient operator keeps $\widetilde{\rho}_{i,j,t}$ equal to $\rho_{i,j}$ in value while removing cross-token derivatives, so the advantage in Equation (9) acts at its assigned position. Let $h^{\mathrm{ref}}_{i,j,t}$ be the reference policy's token log-probability. Define $d_{i,j,t} = h^{\mathrm{ref}}_{i,j,t} - h_{i,j,t}$, $k_{i,j,t} = e^{d_{i,j,t}} - d_{i,j,t} - 1$, and $\bar{\rho}_{i,j,t} = \mathrm{clip}(\widetilde{\rho}_{i,j,t}, 1-\epsilon_-, 1+\epsilon_+)$. For a response batch $\mathfrak{B}$, RVPG minimizes

$$\mathcal{L}_{\mathrm{RVPG}}(\boldsymbol{\theta}) = \frac{1}{|\mathfrak{B}|} \sum_{(i,j)\in\mathfrak{B}} \frac{1}{n_{i,j}} \sum_t w_{i,j,t}\big[-\min\big\{\widetilde{\rho}_{i,j,t} A^{\mathrm{tok}}_{i,j,t},\, \bar{\rho}_{i,j,t} A^{\mathrm{tok}}_{i,j,t}\big\} + \beta k_{i,j,t}\big]. \tag{11}$$

The response-level geometric mean follows the length-normalized sequence importance ratio of GSPO (Zheng et al., 2025). The position-wise stop-gradient surrogate preserves field-specific credit before PPO/GRPO-style clipping (Schulman et al., 2017; Shao et al., 2024). Under the sufficient condition of Corollary 3 in the appendix, the gradient at an unclipped verdict or region position follows its assigned advantage,

$$\mathrm{sign}\left(-\frac{\partial \mathcal{L}_{\mathrm{RVPG}}}{\partial h_{i,j,t}}\right) = \mathrm{sign}\big(A^{\mathrm{tok}}_{i,j,t}\big)\,.$$

Here, $\mathrm{clip}(x,a,b) = \min\{\max\{x,a\},b\}$, $0 < \epsilon_- < 1$ and $\epsilon_+ > 0$ are the lower and upper clipping widths, and $\beta \geq 0$ scales the KL surrogate. All positions update the shared parameters. After online training, we set $\boldsymbol{\phi} = \boldsymbol{\theta}$. Its verdicts and scores support code-state evaluation and candidate ranking; the fixed readout supplies terminal rewards to downstream policies.

4 Verification Capability and Code Decisions

4.1 Evaluation protocol

The main test covers two code domains with the verdict interfaces defined in Section 2. The competitive-programming set contains chronologically held-out AtCoder and NowCoder programs; the repository set contains patch-linked views from SWE-Gym and R2E-Gym. We train CodeVerifier at 2B, 4B, and 9B and compare it with direct judgments from the size-matched base models. A separate binary protocol includes CodeJudge (Tong & Zhang, 2024); its two-pass semantic judgment is mapped to success or failure. All checkpoints are selected on validation and fixed before testing.

The structured main test uses one deterministic judgment per program or repository view. It reports exact task-label accuracy A_{exact}, binary success accuracy A_{bin}, and line-level repair-region IoU I_{err} on failed examples whose associated repair is visible. The joint metric is $G = \sqrt{A_{\mathrm{exact}} I_{\mathrm{err}}}$. Competitive and repository accuracies are reported separately; A_{exact} and I_{err} are pooled over their eligible records before G is computed. The binary baseline comparison is a distinct protocol with fixed inputs and scoring for all methods. Auxiliary evaluations then use paired edits, per-test outcomes, editing states, final project outcomes, or within-task candidate sets according to the question being tested. Appendix A gives the configurations, metrics, prompts, and judgment budgets.

4.2 Main results

- Evidence projection raises the joint verdict–region score at all three scales, and field-wise online optimization achieves the strongest G throughout (Table 1). At 9B, G rises from 0.462 for teacher-target SFT to 0.553 after projection and 0.617 after online refinement, with both A and I increasing.

Table 1: Main results and ablations. Bin./Exact denote binary/four-way competitive-program accuracy; Repo denotes repository-view accuracy. Joint metrics use exact-label accuracy A and failed-instance repair-region IoU I after pooling their eligible examples from the competitive and repository domains; $G = \sqrt{AI}$ is computed before display rounding. Darker cells indicate higher values, and ♦ marks the full model.

	2B			4B			9B		
VERDICT	Bin.	Exact	Repo	Bin.	Exact	Repo	Bin.	Exact	Repo
♦ **CodeVerifier**	0.804	0.696	0.846	0.839	0.757	0.850	**0.887**	**0.807**	**0.861**
JOINT	A	I	G	A	I	G	A	I	G
♦ **CodeVerifier**	0.747	0.374	0.528	0.789	0.446	0.593	**0.825**	**0.461**	**0.617**
ABLATIONS *Stage I — Evidence projection*									
Base	0.526	0.144	0.276	0.659	0.274	0.425	0.724	0.304	0.469
Teacher SFT	0.675	0.222	0.387	0.758	0.283	0.463	0.753	0.284	0.462
Projected SFT	0.718	0.293	0.458	0.736	0.355	0.511	0.792	0.387	0.553
Stage II — Field-wise online optimization									
w/o R-adv.	0.725	0.285	0.455	0.739	0.349	0.508	0.797	0.386	0.555
Shared adv.	0.730	0.305	0.472	0.758	0.375	0.533	0.794	0.392	0.558
Shared, routed	0.725	0.283	0.453	0.737	0.349	0.507	0.781	0.380	0.545
w/o repo	0.676	0.298	0.449	0.793	0.373	0.544	0.803	0.384	0.555
w/o sol.	0.732	0.307	0.474	0.800	0.393	0.561	0.804	0.389	0.559
w/o empty fb.	0.732	0.310	0.476	0.794	0.378	0.548	0.795	0.384	0.553
w/o line cov.	0.698	0.311	0.466	0.806	0.376	0.551	0.800	0.384	0.554

- Under the dedicated binary comparison, CODEVERIFIER improves over CodeJudge by 3.2 percentage points on average and reaches 87.67% accuracy at 9B (Figure 3). The gain holds across competitive programs and repository code despite distinct verdicts, inputs, and contexts.
- Five judgments raise Top-1 by up to 6.46 pp; CODEVERIFIER rewards exceed Tests/Qwen by 1.69, 1.52, and 1.33 pp on OpenR1, LiveCodeBench v6, and SWE-bench Verified.

4.3 VERIFICATION TO CODE DECISIONS

The main test evaluates a fixed code state. During development, the state changes, execution may reveal only a partial signal, and an agent must turn a judgment into a choice or a policy update. We therefore examine four questions: response to code changes, judgment under limited execution, success prediction during editing, and the use of structured judgments for candidate selection and policy learning.

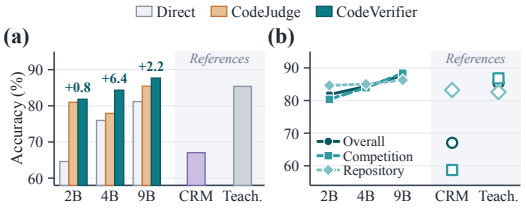

Figure 3: Binary verdict accuracy by matched size (a) and domain (b). Labels show CODEVERIFIER–CodeJudge gains (pp); CRM and Teacher form a separate reference block.

RQ1: Do verdicts and regions respond to repair-associated code changes? We examine two forms of evidence response. Bidirectional edits keep the task, language, and surrounding code fixed while changing correctness, testing whether the

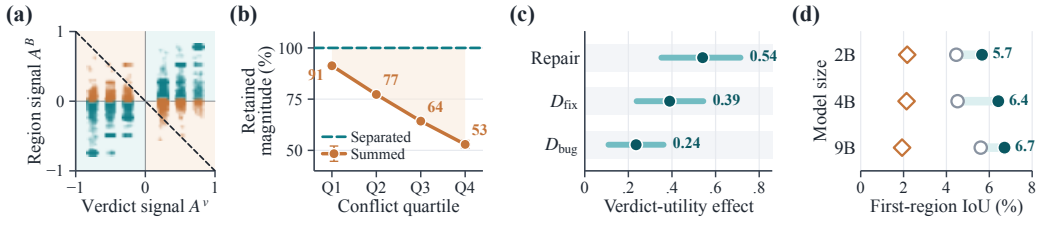

Figure 4: Field feedback and repair correspondence. (a) Centered verdict/region signals; shading marks sign agreement and the anti-diagonal exact cancellation ($A^v + A^B = 0$). (b) Summed-signal retention; dashed line: separated-field reference; error bars: one candidate-group bootstrap SD. (c) 9B verified-repair, D_{fix}, and D_{bug} effects (95% bootstrap intervals). (d) UOJ-Bench first-region IoU; open diamonds, open circles, and filled circles mark same-file translations, Base, and CODEVERIFIER.

verdict follows the edit outcome. Real UOJ-Bench submissions provide the locations later modified by programmers, testing whether predicted regions point to repair-relevant code (Xu et al., 2026).

Global verdict and local region feedback can point in opposite directions, causing part of their signal to cancel when combined into a shared scalar (Figure 4(a,b)). Compared with edits that preserve correctness, successful repairs produce a larger rise in verdict utility, while defect-inducing edits produce a larger fall (Figure 4(c)). On UOJ-Bench, the first predicted region overlaps the programmer's subsequent edit more closely than a same-file translation, and this correspondence improves with model scale (Figure 4(d)). The verdict follows changes in correctness, and the predicted regions overlap code later changed in successful repairs. Appendix B gives defect-injection and repeated-judgment analyses.

RQ2: Can model judgments reduce execution cost? After partial execution, the remaining task is to distinguish programs with the same observed pass count. LiveCodeBench retains candidates from the same problem together with per-test outcomes, allowing candidate discrimination and serving-time analysis on the same records (Jain et al., 2025c). TACO contains Codeforces and HackerEarth problems outside the competition platforms used for training and provides longer test suites, so it exposes the cost of errors that surface later (Li et al., 2023). Figure 5(d) calibrates the TACO full-suite uniform-sampling count whose mean detection rate matches the verifier's recall.

We separate single-judgment residual-error detection, five-judgment candidate selection, and matched-recall timing. On LiveCodeBench, one 9B judgment detects 81.25% of incorrect candidates that pass the first eight tests, matching the mean detection rate of eight uniformly sampled tests from each candidate's remaining suite. Increasing model scale improves discrimination, although the return tapers as serving time rises (Figure 5(a)). Across all three scales and test budgets, the mean utility of five binary judgments per candidate improves within-task selection, with the largest contribution when execution supplies the least information (Figure 5(c)). On TACO, errors that surface later require more tests and longer execution to reach the same recall. Figure 5(d) instead calibrates an independent uniform-without-replacement sample from each complete suite to the same mean recall. At the most difficult level, serial execution takes 419 times as long as one 9B judgment, while the idealized 64-process estimate remains 6.5 times as long (Figure 5(b,d)). Appendix C gives the full difficulty breakdown and residual-error analysis.

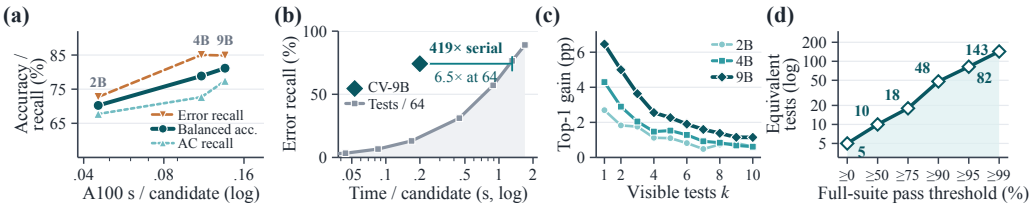

Figure 5: Verification quality and execution cost. (a) LiveCodeBench accuracy/recall versus amortized A100 time. (b) TACO error recall after $\geq 99\%$ full-suite pass: one 9B judgment versus idealized 64-process tests. (c) Top-1 gain from five-judgment tie-breaking over $k = 1, \ldots, 10$ visible tests. (d) Uniform full-suite samples matching 9B recall across pass thresholds.

RQ3: Does success prediction remain useful during editing? We evaluate whether CODEVERIFIER judges the outcome of the current editing state and whether its repository score changes consistently with observed edits. BigCodeBench supplies complete-code records, CollabLLM supplies editing-state records, and CoderForge supplies views before and after repository modifications together with the source patch outcome. We use balanced accuracy for discrete verdicts and ROC AUC for continuous verdict scores.

Table 2: BigCodeBench BAcc; right column: editing-state AUC gain over Base.

	BAcc (%)		Edit AUC gain
Size	Complete	Edit	Rel. ↑
2B	58.8	70.0	+2.3%
4B	65.0	71.0	+4.5%
♦ 9B	**72.1**	**73.2**	**+6.8%**

Table 3: Complete-code BAcc by source; ΔD is the CoderForge edit gain over Base.

	Source BAcc (%)			Repo edit	
Size	Teacher	Qwen	MiniMax	D_{repo}	ΔD↑
2B	56.5	60.6	66.0	.046	+.087
4B	65.6	62.4	67.4	.388	+.134
♦ 9B	**71.6**	**69.4**	**77.8**	**.410**	+.089

BAcc improves with scale on both complete code and editing states. On the same editing states, verdict-score AUC improves over the size-matched base at every scale, reaching a 6.8% relative gain at 9B (Table 2). Complete-code BAcc also rises across teacher-, Qwen-, and MiniMax-generated candidates. For CoderForge repository edits, D_{repo} is the difference between the mean before-to-after label-margin change of successful and failed patches. Its positive value at every scale (Table 3) means that patches which eventually pass exhibit a larger mean score change than patches that fail.

RQ4: Can structured judgments improve code decisions? Code decisions involve both selecting a reliable program from existing candidates and using reward to improve later generation. HumanEvalPack provides pairwise programs for basic success preference, while BigCodeBench provides within-task candidate pools for multi-candidate ranking (Lambert et al., 2025; Zhuo et al., 2025). OpenR1, LiveCodeBench v6, and SWE-bench Verified evaluate the resulting policy outcomes in competitive programming and repository repair (Jimenez et al., 2024).

HumanEvalPack pairwise comparisons average five judgments per program, while BigCodeBench ranks within-task candidates by one discrete verdict with uniform tie-breaking. Both protocols improve over the size-matched bases at all three scales (Figure 6(a)). Figure 6(b,c) compares policy updates from test/Qwen, CodeRM-NT, and CODEVERIFIER rewards; the leftmost bars are the three-seed untrained Qwen3.5-4B reference, not a reward-training arm. CODEVERIFIER reaches the strongest mean pass rate on OpenR1, LiveCodeBench v6, and SWE-bench Verified, exceeding the corresponding test/Qwen arm by 1.69, 1.52, and 1.33 percentage points. Appendix E reports an independent repository sampling-budget study. Together, these results show that the fixed verifier readout provides useful terminal rewards in both task families.

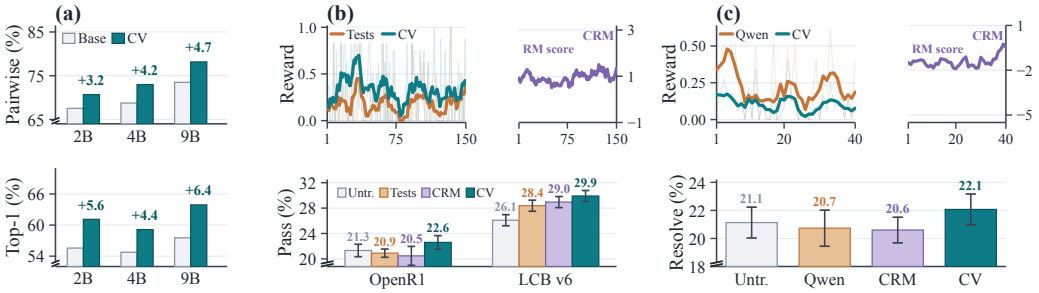

Figure 6: Code decisions. (a) HumanEvalPack pairwise accuracy and BigCodeBench Top-1. (b,c) Reward traces and outcomes for Tests/Qwen, CodeRM-NT (CRM), and CODEVERIFIER (CV) in competition and repository repair; Untr. is untrained Qwen3.5-4B. CRM uses the right RM-score axis. Faint lines show per-step means, dark lines show 10-update/5-update moving means, and bars give mean ± sample SD over three seeds.

5 RELATED WORK

Execution feedback supports code generation, selection, and policy learning. Tests expand behavioral coverage, while learned and generative verifiers rank programs and patches, supervise intermediate steps, or reward candidate and trajectory search (Chen et al., 2023; Ni et al., 2023; Tong & Zhang, 2024; Li et al., 2025; Xia et al., 2026; Zhu et al., 2026; Zhang et al., 2025a; Jimenez et al., 2024; Zhang et al., 2023; SHUM et al., 2026; Da et al., 2025). CODEVERIFIER unifies execution labels and repair regions across complete programs, editing states, and patch-linked views. Its online stage uses GSPO's length-normalized ratio with separate verdict and region advantages (Zheng et al., 2025); Appendix G gives the detailed discussion.

6 CONCLUSION

CODEVERIFIER judges evolving code states, not only completed programs, by turning execution outcomes and successful repairs into global verdict and local repair-region supervision. This structured interface lets one generative verifier span competitive programs, intermediate edits, and repository patches, supporting code-change assessment, candidate selection, and policy training before definitive execution. It bridges generation and testing while keeping execution as the final correctness criterion.

AI USE STATEMENT

Except for the experiments involving LLMs, generative AI tools were used only to polish the manuscript, check grammar, and assist with auxiliary code. The authors reviewed all code and AI-assisted text and take full responsibility for the manuscript, claims, code, and artifacts.

ETHICS STATEMENT

The study uses public and automatically generated code, execution outcomes, and repair histories. It does not involve human participants or infer identity or sensitive attributes. For human-written programs, we analyze errors and use subsequent edits as the reference for predicted regions.

REPRODUCIBILITY STATEMENT

Sections 2–3 specify the supervision, model objective, and readouts, and Section 4 defines the evaluation protocols. Appendix A documents data construction, configurations, prompts, and per-evaluation metrics; Appendix F gives the derivation supporting Corollary 3. We provide the complete training algorithms and model weights.

REFERENCES

Wasi Uddin Ahmad, Nikolai Ludwig, Somshubra Majumdar, and Boris Ginsburg. Open-swe-traces: Advancing dual-mode multilingual distillation for software engineering agents. *CoRR*, abs/2606.16038, 2026. doi: 10.48550/ARXIV.2606.16038. URL https://doi.org/10.48550/arXiv.2606.16038.

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 752–771. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/ahmed25b.html.

Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zhou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. Coderforge-preview: Sota open dataset for training efficient agents, February 2026. URL https://www.together.ai/blog/coderforge-preview. Project core leads: Alpay Ariyak; Zhongzhu Zhou; Qingyang Wu.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale, 2026. URL https://arxiv.org/abs/2602.23866.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse (eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

Maosong Cao, Alexander Lam, Haodong Duan, Hongwei Liu, Songyang Zhang, and Kai Chen. Compassjudger-1: All-in-one judge model helps model evaluation and evolution, 2024. URL https://arxiv.org/abs/2410.16256.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. Swe-universe: Scale real-world verifiable environments to millions, 2026. URL https://arxiv.org/abs/2602.02361.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/2460396f2d0d421885997dd1612ac56b-Paper-Conference.pdf.

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, 2021. URL https://arxiv.org/abs/2110.14168.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. Agent-rlvr: Training software engineering agents via guidance and environment rewards, 2025. URL https://arxiv.org/abs/2506.11425.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136. URL https://doi.org/10.1109/C-M.1978.218136.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025. doi: 10.48550/ARXIV.2509.16941. URL https://doi.org/10.48550/arXiv.2509.16941.

Deming Ding, Shichun Liu, Enhui Yang, Jiahang Lin, Ziying Chen, Shihan Dou, Honglin Guo, Weiyu Cheng, Pengyu Zhao, Chengjun Xiao, Qunhong Zeng, Qi Zhang, Xuanjing Huang, Qidi Xu, and Tao Gui. OctoBench: Benchmarking scaffold-aware instruction following in repository-grounded

agentic coding. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5958–5978, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.269. URL https://aclanthology.org/2026.acl-long.269/.

Shangbin Feng, Weijia Shi, Yike Wang, Wenxuan Ding, Vidhisha Balachandran, and Yulia Tsvetkov. Don't hallucinate, abstain: Identifying LLM knowledge gaps via multi-LLM collaboration. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14664–14690, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.786. URL https://aclanthology.org/2024.acl-long.786/.

Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller (eds.), *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pp. 416–419. ACM, 2011. doi: 10.1145/2025113.2025179. URL https://doi.org/10.1145/2025113.2025179.

Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting swe agents with prms, 2025. URL https://arxiv.org/abs/2509.02360.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 19034–19055. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall (eds.), *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pp. 213–223. ACM, 2005. doi: 10.1145/1065010.1065036. URL https://doi.org/10.1145/1065010.1065036.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, Sergei Skvortsov, and Boris Yangel. Training long-context, multi-turn software engineering agents with reinforcement learning, 2025. URL https://arxiv.org/abs/2508.03501.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/gu24c.html.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1321–1330. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/guo17a.html.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, 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, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, 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, Mingxu Zhou, 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, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, 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, 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 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè (eds.), *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pp. 837–847. IEEE Computer Society, 2012. doi: 10.1109/ICSE.2012.6227135. URL https://doi.org/10.1109/ICSE.2012.6227135.

Sihan Hu, Lyuhan Huang, Youjin Deng, and Kun Chen. Scicode-verified: How benchmark defects underestimated the scientific-coding ability of language models, 2026. URL https://arxiv.org/abs/2608.04975.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 13419–13432. Curran Associates, Inc., 2022. doi: 10.52202/068431-0976. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/5762c579d09811b7639be2389b3d07be-Paper-Conference.pdf.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 26700–26716. PMLR, 13–19 Jul 2025a. URL https://proceedings.mlr.press/v267/jain25a.html.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learn-*

ing Representations, ICLR 2025, Singapore, April 24-28, 2025. OpenReview.net, 2025b. URL https://openreview.net/forum?id=7o6SG5gVev.

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 *The Thirteenth International Conference on Learning Representations*, 2025c. URL https://openreview.net/forum?id=chfJJYC3iL.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025d. URL https://openreview.net/forum?id=7evvwwdo3z.

Juyong Jiang, Jiasi Shen, Sunghun Kim, Kang Min Yoo, Jeonghoon Kim, and Sungju Kim. ReflexiCoder: Teaching large language models to self-reflect on generated code and self-correct it via reinforcement learning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37543–37562, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1872. URL https://aclanthology.org/2026.findings-acl.1872/.

Xue Jiang, Yihong Dong, Mengyang Liu, Deng Hongyi, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, and Ge Li. CODERL+: Improving code generation via reinforcement with execution semantics alignment. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3608–3622, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.164. URL https://aclanthology.org/2026.acl-long.164/.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=q56ZI1Co43.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pp. 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

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, 2026. URL https://arxiv.org/abs/2607.05391.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 18319–18345. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/lai23b.html.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1755–1797, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.96. URL https://aclanthology.org/2025.findings-naacl.96/.

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 S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328. Curran Associates, Inc., 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf.

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015. doi: 10.1109/TSE.2015.2454513.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. CodePRM: Execution feedback-enhanced process reward model for code generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.428. URL https://aclanthology.org/2025.findings-acl.428/.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset, 2023. URL https://arxiv.org/abs/2312.14852.

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.

Zixu Li and Zhiqi Peng. Reason-code: Reliable code generation via test-driven Monte Carlo tree search. In Yunyao Li, Georg Rehm, and Mei Tu (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 447–458, San Diego, California, USA, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-394-4. doi: 10.18653/v1/2026.acl-industry.30. URL https://aclanthology.org/2026.acl-industry.30/.

Jiarong Liang, Zhiheng Lyu, Zijie Liu, Xiangchao Chen, Ping Nie, Kai Zou, and Wenhu Chen. Swenext: Scalable real-world software engineering tasks for agents. *arXiv preprint arXiv:2603.20691*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. RLTF: reinforcement learning from unit test feedback. *Trans. Mach. Learn. Res.*, 2023, 2023a. URL https://openreview.net/forum?id=hjYmsV6nXZ.

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 A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023b. doi: 10.52202/075280-0943. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. Learning code preference via synthetic evolution, 2024a. URL https://arxiv.org/abs/2410.03837.

Kaibo Liu, Yudong Han, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. Who judges the judge: An empirical study on online judge tests. In René Just and Gordon Fraser (eds.), *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pp. 334–346. ACM, 2023c. doi: 10.1145/3597926.3598060. URL https://doi.org/10.1145/3597926.3598060.

Kaibo Liu, Yudong Han, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Federica Sarro, Gang Huang, and Yun Ma. Trickybugs: A dataset of corner-case bugs in plausible programs. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pp. 113–117, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 9798400705878. doi: 10.1145/3643991.3644870. URL https://doi.org/10.1145/3643991.3644870.

Shudong Liu, Hongwei Liu, Junnan Liu, Linchen Xiao, Songyang Gao, Chengqi Lyu, Yuzhe Gu, Wenwei Zhang, Derek F. Wong, Songyang Zhang, and Kai Chen. CompassVerifier: A unified and robust verifier for LLMs evaluation and outcome reward. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33466–33494, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1698. URL https://aclanthology.org/2025.emnlp-main.1698/.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024c. URL https://openreview.net/forum?id=pPjZIOuQuF.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. Rm-bench: Benchmarking reward models of language models with subtlety and style. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 44323–44355, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/6da1eec80095dc5937f7716db15aca4b-Paper-Conference.pdf.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, Cheng Li, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 58780–58807. Curran Associates, Inc., 2025c. doi: 10.52202/085713-1967. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/54e847e1dffc87a8063844b149148557-Paper-Conference.pdf.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025d. URL https://arxiv.org/abs/2503.20783.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. Dynamic scaling of unit tests for code reward modeling. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6917–6935, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.343. URL https://aclanthology.org/2025.acl-long.343/.

Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Hao Zheng, Wenting Zhao, Junyang Lin, and Binyuan Hui. Scaling agentic verifier for competitive coding. *CoRR*, abs/2602.04254, 2026. doi: 10.48550/ARXIV.2602.04254. URL https://doi.org/10.48550/arXiv.2602.04254.

Zihan Ma, Taolin Zhang, Maosongcao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. Rethinking verification for llm code generation: From generation to testing. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz,

M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 137549–137582. Curran Associates, Inc., 2025b. doi: 10.52202/085713-4591. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c8ec1867837f5cdaa9e61d40a8b8bb4a-Paper-Conference.pdf.

Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi: 10.1109/TSE.2019.2946563.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44412–44450. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/miserendino25a.html.

Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assunção. Is llm-generated code more maintainable & reliable than human-written code? In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2025, Honolulu, HI, USA, October 2-3, 2025*, pp. 151–162. IEEE, 2025. doi: 10.1109/ESEM64174.2025.00036. URL https://doi.org/10.1109/ESEM64174.2025.00036.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 26106–26128. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/ni23b.html.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 75–84. IEEE Computer Society, 2007. doi: 10.1109/ICSE.2007.37. URL https://doi.org/10.1109/ICSE.2007.37.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Michal Young and Tao Xie (eds.),

Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, pp. 24–36. ACM, 2015. doi: 10.1145/2771783.2771791. URL https://doi.org/10.1145/2771783.2771791.

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

Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression . In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 658–666, Los Alamitos, CA, USA, June 2019. IEEE Computer Society. doi: 10.1109/CVPR.2019.00075. URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00075.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

KaShun SHUM, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. SWE-RM: Execution-free feedback for software engineering agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=H9wMe1G76j.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: an empirical study. *Empir. Softw. Eng.*, 30(3):65, 2025. doi: 10.1007/S10664-025-10614-4. URL https://doi.org/10.1007/s10664-025-10614-4.

Lingxiao Tang, He Ye, Zhaoyang Chu, Muyang Ye, Zhongxin Liu, Xiaoxue Ren, and Lingfeng Bao. ExecVerify: White-box RL with verifiable stepwise rewards for code execution reasoning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13850–13875, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.631. URL https://aclanthology.org/2026.acl-long.631/.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 30624–30650. Curran Associates, Inc., 2024. doi: 10.52202/079017-0963. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/36850592258c8c41cecdaa3dea5ff7de-Paper-Datasets_and_Benchmarks_Track.pdf.

Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating code generation with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 20032–20051, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.1118. URL https://aclanthology.org/2024.emnlp-main.1118/.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022. URL https://arxiv.org/abs/2211.14275.

Vatsal Venkatkrishna, Indraneil Paul, and Iryna Gurevych. Aletheia: What makes rlvr for code verifiers tick?, 2026. URL https://arxiv.org/abs/2601.12186.

Konstantinos Vergopoulos, Mark Niklas Mueller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 61240–61265. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/vergopoulos25a.html.

Binghai Wang, Chenlong Zhang, Dayiheng Liu, Jiajun Zhang, Jiawei Chen, Mingze Li, Mouxiang Chen, Rongyao Fang, Siyuan Zhang, Xuwu Wang, Yuheng Jing, Zeyao Ma, and Zeyu Cui. The verification horizon: No silver bullet for coding agent rewards, 2026. URL https://arxiv.org/abs/2606.26300.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TestEval: Benchmarking large language models for test case generation. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3547–3562, Albuquerque, New Mexico, April 2025a. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.197. URL https://aclanthology.org/2025.findings-naacl.197/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 143630–143664. Curran Associates, Inc., 2025c. doi: 10.52202/085713-4809. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/d38653cdaa8e992549e1e9e1621610d7-Paper-Conference.pdf.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 78500–78525. Curran Associates, Inc., 2025. doi: 10.52202/085713-2629. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/7107d4d2e837bde2171c6b71b5bde954-Paper-Conference.pdf.

Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *ACM Trans. Softw. Eng. Methodol.*, 35(3), February 2026. ISSN 1049-331X. doi: 10.1145/3736407. URL https://doi.org/10.1145/3736407.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pp. 1556–1560, New York, NY, USA, 2020. Association

for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417943. URL https://doi.org/10.1145/3368089.3417943.

Shirley Wu, Michel Galley, Baolin Peng, Hao Cheng, Gavin Li, Yao Dou, Weixin Cai, James Zou, Jure Leskovec, and Jianfeng Gao. Collabllm: From passive responders to active collaborators, 2025. URL https://arxiv.org/abs/2502.00640.

Xiao Xia, Dan Zhang, and Tianrui Sun. CodeRM-NT: Reward model for code RL without unit tests. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 43316–43333, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.2150. URL https://aclanthology.org/2026.findings-acl.2150/.

Tingqiang Xu, Hangrui Zhou, Tianle Cai, Alex Gu, and Kaifeng Lyu. Beyond problem solving: Uoj-bench for evaluating code generation, hacking, and repair in competitive programming. *CoRR*, abs/2606.12864, 2026. doi: 10.48550/ARXIV.2606.12864. URL https://doi.org/10.48550/arXiv.2606.12864.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/8b86cf5ace600c48fd188efbb8dedec8-Paper-Datasets_and_Benchmarks_Track.pdf.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. Process-supervised reinforcement learning for code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 14213–14226, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.719. URL https://aclanthology.org/2025.emnlp-main.719/.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-bench. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel Briand. Swe-abs: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark, 2026. URL https://arxiv.org/abs/2603.00520.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Xingru Jiang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Reasoning through execution: Unifying process and outcome rewards for code generation. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 72972–72994. PMLR, 13–19 Jul 2025b. URL https://proceedings.mlr.press/v267/yu25f.html.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACECODER: Acing coder RL via automated test-case synthesis. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.587. URL https://aclanthology.org/2025.acl-long.587/.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

Jiazheng Zhang, Ziche Fu, Zhiheng Xi, Wenqing Jing, Mingxu Chai, Wei He, Guoqiang Zhang, Chenghao Fan, Chenxin An, Wenxiang Chen, Zhicheng Liu, Haojie Pan, Dingwei Zhu, Tao Gui, Qi Zhang, and Xuanjing Huang. AgentV-RL: Scaling reward modeling with agentic verifier. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 23078–23100, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1156. URL https://aclanthology.org/2026.findings-acl.1156/.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025a. URL https://openreview.net/forum?id=Ccwp4tFEtE.

Taolin Zhang, Maosong Cao, Alexander Lam, Songyang Zhang, and Kai Chen. Compassjudger-2: Towards generalist judge model via verifiable rewards, 2025b. URL https://arxiv.org/abs/2507.09104.

Taolin Zhang, Zihan Ma, Maosong Cao, Junnan Liu, Songyang Zhang, and Kai Chen. Coding triangle: How does large language model understand code?, 2025c. URL https://arxiv.org/abs/2507.06138.

Xiaoqing Zhang, Yuhan Liu, Flood Sung, Xiuying Chen, Shuo Shang, and Rui Yan. Thinking before running! efficient code generation with thorough exploration and optimal refinement. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23268–23281, Vienna, Austria, July 2025d. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1195. URL https://aclanthology.org/2025.findings-acl.1195/.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pp. 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025. URL https://arxiv.org/abs/2507.18071.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46595–46623. Curran Associates, Inc., 2023. doi: 10.52202/075280-2020. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.49. URL https://aclanthology.org/2024.findings-acl.49/.

Ming Zhong, Xiang Zhou, Ting-Yun Chang, Qingze Wang, Nan Xu, Xiance Si, Dan Garrette, Shyam Upadhyay, Jeremiah Liu, Jiawei Han, Benoit Schillings, and Jiao Sun. Swe-if: Aligning code evaluation with human preference, 2026. URL https://arxiv.org/abs/2510.07315.

Xiao Zhu, Xinyu Zhou, Boyu Zhu, Hanxu Hu, Mingzhe Du, Haotian Zhang, Huiming Wang, and Zhijiang Guo. Codescaler: Scaling code llm training and test-time inference via reward models, 2026. URL https://arxiv.org/abs/2602.17684.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

A Training, Prompts, and Evaluation Details

A.1 Training data and evaluation splits

The training data contain successful and failed examples from competitive programs and patch-linked repository views. Competitive programs come from chronological AtCoder and NowCoder submission histories; repository data come from SWE-Gym (Pan et al., 2025) and R2E-Gym (Jain et al., 2025d). Structured supervision treats one prompt and its target as an example, whereas RVPG treats a group of responses to the same prompt as its unit of training. Failed instances with visible repair evidence enter structured supervision when $\eta_i^{\text{teach}} > 0.80$ and every repair interval overlaps a verdict-correct teacher region; those with $0.60 \leq \eta_i^{\text{teach}} \leq 0.80$ enter online optimization. These stage criteria use teacher regions before coordinate projection. Successful instances have no region IoU and are assigned to disjoint stage sets matched by input form and source; both their structured targets and online feedback require an empty region set. Table 4 summarizes both stages.

The three models are initialized from Qwen3.5-2B, Qwen3.5-4B, and Qwen3.5-9B. For each $\boldsymbol{z}_i$, the Qwen3.5-397B-A17B teacher generates J_i structured candidates, which are then filtered by the task outcome and repair-associated regions; candidate-group sizes follow the source rollout batches. The online stage samples eight responses for each prompt and constructs advantages within that group. Appendix A.7 gives the output conventions.

Teacher-target SFT and evidence-projected SFT in the first stage share training instances, model initialization, and output structure. The former learns the selected teacher response, while the latter

learns the target after evidence projection. Every second-stage ablation starts from the same evidence-projected SFT checkpoint; "w/o repo" and "w/o sol." remove the corresponding prompt family only from online optimization.

Training, validation, and main test examples are ordered chronologically at the task level. Earlier tasks form the training set, subsequent tasks form the validation set, and the newest tasks form the main test set; all code states from one competitive-programming problem or repository task remain in the same split. The validation and main test sets contain 760 and 1,435 code states, respectively, with 947 competitive programs and 488 repository code states in the main test set. The LiveCodeBench instances used in this paper postdate the competitive-programming training cutoff and are not included in the training collection. Teacher candidates are generated after the temporal split, and evaluation on the remaining public datasets begins only after checkpoint selection.

A.2 Repository views and patch-level outcomes

For repository tasks, a candidate patch p is the unit of outcome. The model reads the associated code views $\mathcal{Z}(p) = \{\boldsymbol{z}_{p,h}\}_{h=1}^{H_p}$, and the project-test result Y_p is paired with each view. Let Γ_p be the repair intervals associated with p, and let $\Lambda_{p,h}$ be the line set shown in view h. The view-level repair target is $\Gamma_{p,h} = \{\gamma \in \Gamma_p : \Lambda(\gamma) \cap \Lambda_{p,h} \neq \varnothing\}$. The repository interface judges the numbered snapshot shown in each view, with PASS/FAIL supervision inherited from the source patch's project-test result. A successful patch has $\Gamma_p = \varnothing$. When a patch is scored as a candidate, its reward is the mean normalized PASS-label score over its associated views; repair-region scores use the intervals visible in the corresponding view.

A.3 Training configuration and model selection

All models are trained with full-parameter BF16 optimization and AdamW. Weight decay is 0.01 for structured SFT and 0.1 for online training. Online training uses $\epsilon_- = \epsilon_+ = 0.2$ and $\beta = 0.02$. Sampling temperature is 0.8 for 2B and 0.75 for both 4B and 9B. Training uses random seed 42. We reserve validation G for model selection and report main-test results only after the model choices are fixed. For online training, the two token limits in the table denote the maximum input and generation lengths.

The online region-match threshold is $\tau_{\text{match}} = 0$, with $(\lambda_{\text{fp}}, \lambda_{\text{fn}}, \lambda_{\text{cov}}) = (0.4, 0.5, 1.0)$. For successful examples, $(b_{\text{empty}}, \lambda_{\text{sp}}, \gamma_+) = (1.0, 0.5, 1.0)$. We set $\zeta_v = 1$, $(\alpha_v, \alpha_B) = (1, 4)$, and $\omega_g = 0$ for both input forms. The verdict-cost matrix has zero diagonal, cost 1 for false rejection, cost 2 for false acceptance, and cost 1.5 for other off-diagonal errors. The full model routes field advantages token-wise, whereas "Shared adv." assigns the resulting feedback as one response-level scalar.

A.4 Evidence projection and structured supervision targets

The teacher π_{teach} generates a candidate set $\mathcal{Y}_i = \{\boldsymbol{y}_{i,j}\}_{j=1}^{J_i}$ for $\boldsymbol{z}_i$, where $\boldsymbol{y}_{i,j} = (\chi_{i,j}, \mathcal{B}_{i,j}, \widehat{v}_{i,j})$. For a single code interval and a sequence of regions, respectively, define

$$\Lambda(b) = \{\ell(b), \dots, u(b)\}, \qquad \Lambda(\mathcal{B}) = \bigcup_{a=1}^{|\mathcal{B}|} \Lambda(b_a),$$

$$\mathrm{IoU}_{\text{line}}(\mathcal{B}, \Gamma) = \frac{|\Lambda(\mathcal{B}) \cap \Lambda(\Gamma)|}{|\Lambda(\mathcal{B}) \cup \Lambda(\Gamma)|}.$$

For two individual intervals, we use the shorthand $\mathrm{IoU}(b, \gamma) = |\Lambda(b) \cap \Lambda(\gamma)| / |\Lambda(b) \cup \Lambda(\gamma)|$. Let $\mathcal{J}_i^{\text{ok}}$ index teacher responses with $\widehat{v}_{i,j} = v_i^\star$. For a failed instance with $K_i \geq 1$, teacher-region quality is

$$\eta_i^{\text{teach}} = \max_{j \in \mathcal{J}_i^{\text{ok}}} \mathrm{IoU}_{\text{line}}(\mathcal{B}_{i,j}, \Gamma_i^\star), \tag{12}$$

The maximum over an empty set is 0. Teacher-region quality is computed before coordinate projection and assigns failed instances to a training stage; successful instances have an empty region target.

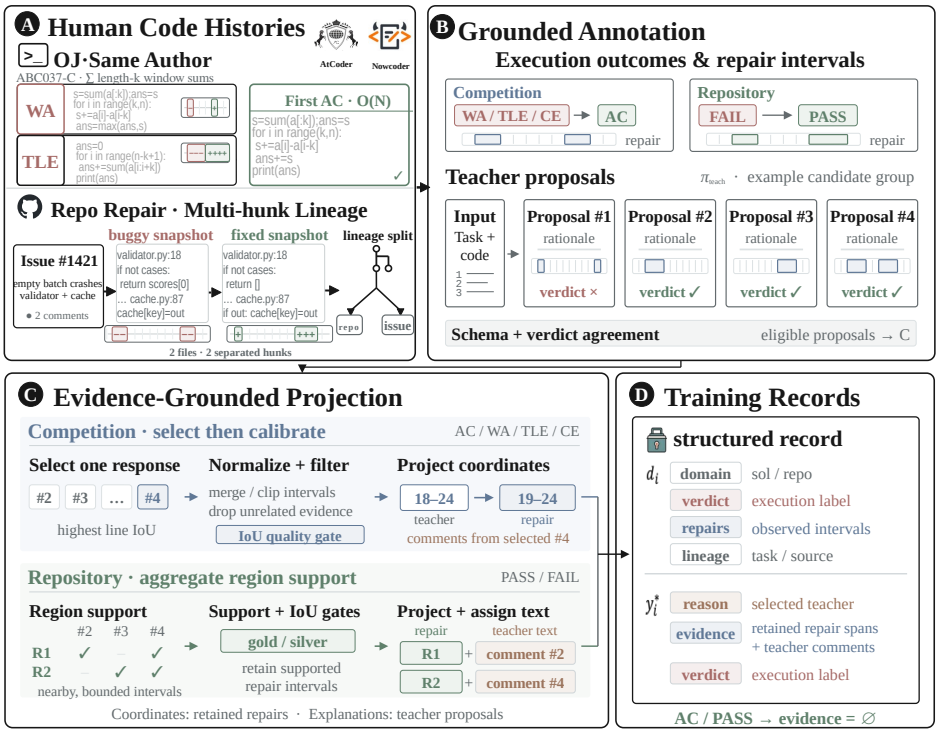

Figure 7: Dual-path construction of structured targets from execution outcomes, repair intervals, and teacher proposals. (a) Code histories and repository patches provide outcomes and repair context. (b) Competition uses AC/WA/TLE/CE and repository uses PASS/FAIL; proposals are filtered by schema and verdict agreement. (c) Competition selects and calibrates one proposal, whereas repository aggregates supported repair intervals before projecting coordinates and assigning teacher text. (d) Both paths produce the same record, with empty evidence for successful instances. The four proposals shown are illustrative; candidate-group size follows the source rollout batch.

Table 4: Training supervision by stage with execution verdicts, repair regions, and teacher candidates.

Stage	Training units	Code	Targets / feedback
Structured SFT	8,883 samples	Competitive programs and patch-linked repository views	Projected targets; failed-instance admission uses pre-projection teacher IoU > 0.80 and overlap availability for every repair interval
RVPG	1,600 prompts; eight responses per update	874 competitive prompts; 726 repository prompts	On-policy verdict and repair feedback; failed-instance admission uses pre-projection teacher IoU in $[0.60, 0.80]$

Table 5: Training configuration by scale. Batch denotes the effective SFT batch or online prompt batch; each online prompt produces eight responses, and length is the configured maximum.

Size	Learning rate	Batch	Schedule (warmup)	Length	Token limit
Stage I — Structured SFT					
2B	5×10^{-7}	64	Cosine (10%)	3 epochs	8,192 total
4B	6×10^{-7}	64	Cosine (15%)	2 epochs	6,144 total
9B	6×10^{-7}	64	Cosine (10%)	3 epochs	6,144 total
Stage II — Field-wise online optimization					
2B	5×10^{-7}	64	Constant (3%)	150 updates	6,144 / 2,048
4B	1×10^{-6}	64	Constant (3%)	100 updates	6,144 / 2,048
9B	1×10^{-6}	64	Constant (3%)	100 updates	6,144 / 2,048

For a failed instance admitted to structured supervision, the feasible region pool is

$$\mathcal{B}_i^+ = \Big\{(b,\kappa) : \exists j \in \mathcal{J}_i^{\text{ok}}, (b,\kappa) \in \mathcal{B}_{i,j}, \max_{1 \le h \le K_i} \text{IoU}(b, \gamma_{i,h}^\star) > 0\Big\}.$$

The indicator that every repair interval has an overlapping candidate region is

$$c_i^{\mathrm{cov}} = \mathbf{1}\left[\min_{1\le h\le K_i} \max_{(b,\kappa)\in\mathcal{B}_i^+} \mathrm{IoU}(b,\gamma_{i,h}^\star) > 0\right].$$

The inner maximum over an empty set is 0. Together with teacher-region quality, this defines the failed instances assigned to each stage.

$$\mathcal{I}_{\mathrm{SFT}}^- = \{i \mid K_i \ge 1,\ \eta_i^{\mathrm{teach}} > 0.80,\ c_i^{\mathrm{cov}} = 1\},\quad \mathcal{I}_{\mathrm{online}}^- = \{i \mid K_i \ge 1,\ 0.60 \le \eta_i^{\mathrm{teach}} \le 0.80\}. \tag{13}$$

SFT uses high-overlap instances for which every repair interval admits a projection. Online training evaluates current-policy responses against execution verdicts and repair regions. Successful instances, with $K_i = 0$, are assigned by input form and source to disjoint sets $\mathcal{I}_{\mathrm{SFT}}^+$ and $\mathcal{I}_{\mathrm{online}}^+$. Each successful SFT instance contains at least one correctly classified teacher response with no regions. The complete instance sets are

$$\mathcal{I}_{\mathrm{SFT}} = \mathcal{I}_{\mathrm{SFT}}^+ \cup \mathcal{I}_{\mathrm{SFT}}^-, \qquad \mathcal{I}_{\mathrm{online}} = \mathcal{I}_{\mathrm{online}}^+ \cup \mathcal{I}_{\mathrm{online}}^-.$$

The target constructor depends on the input form. For $g_i = \mathsf{sol}$, construction first selects a primary verdict-correct response using line-set IoU, intersection size, exact line-set agreement, schema validity, and deterministic tie-breaks. Its evidence is then normalized to closed intervals in the visible input, with boundaries clipped and overlapping or adjacent items merged with their comments. Evidence with no intersection with the repair reference is removed before the projection-quality check. For an admitted record, the output coordinates are the retained intervals from $\Gamma_i^\star$; each interval receives the teacher comment selected by its overlap with the processed evidence, and the global rationale comes from the selected primary response.

For $g_i = \mathsf{repo}$, construction collects verdict-correct candidates satisfying the repository schema and aggregates support for each repair interval across distinct candidates. The support rule uses the configured interval-gap and predicted-length constraints together with the applicable support tier and aggregate-overlap condition. Retained intervals use their repair-reference coordinates, while explanations are selected from supporting candidates; the global rationale is selected using coverage of retained intervals and candidate IoU/F1. The resulting target may contain a supported subset of the repair intervals.

Let $\mathcal{K}_i^{\mathrm{proj}} \subseteq \{1,\dots,K_i\}$ index the repair intervals retained by the applicable constructor. For a failed instance,

$$\mathcal{B}_i^\star = \big((\gamma_{i,h}^\star, \kappa_{i,h}^\star)\big)_{h\in\mathcal{K}_i^{\mathrm{proj}}}, \qquad \chi_i^\star = \chi_{i,j_i^\star},$$

where $j_i^\star$ records the selected rationale source and $\kappa_{i,h}^\star$ is the explanation assigned by the competition or repository constructor. Regions are serialized in target order. When all repair intervals are retained, $\mathcal{K}_i^{\mathrm{proj}} = \{1,\dots,K_i\}$; the reference $\Gamma_i^\star$ used by d_i, online utilities, and evaluation is unchanged.

For a successful instance, $\chi_i^\star$ is taken from an eligible teacher response with an empty region set, and $\mathcal{B}_i^\star = \varnothing$. Let $\boldsymbol{y}_i^{\mathrm{raw}}$ denote the selected verdict-correct teacher response used by the teacher-target SFT comparison. Teacher-target SFT learns $\boldsymbol{y}_i^{\mathrm{raw}}$, whereas evidence-projected SFT learns the target produced by the applicable constructor. The fields $\chi_i^\star$, $\mathcal{B}_i^\star$, and $v_i^\star$ define the evidence-projection operator Π_{ev} and Equation (6) in the main text.

For every failed instance admitted to structured supervision, the target verdict agrees with the execution outcome and

$$\Lambda(\mathcal{B}_i^\star) = \bigcup_{h\in\mathcal{K}_i^{\mathrm{proj}}} \Lambda(\gamma_{i,h}^\star) \subseteq \Lambda(\Gamma_i^\star).$$

The overlap and support checks constrain localization, while the global rationale and local comments remain teacher-authored language supervision.

SFT learns the complete projected response. Online training retains the input, execution verdict, and repair regions and samples responses from the current policy. The two training sets are

$$\mathcal{D}_{\mathrm{SFT}} = \{(\boldsymbol{z}_i, \boldsymbol{y}_i^\star) : i \in \mathcal{I}_{\mathrm{SFT}}\}, \qquad \mathcal{D}_{\mathrm{online}} = \{d_i : i \in \mathcal{I}_{\mathrm{online}}\}.$$

A.5 FIELD UTILITIES AND THE ONLINE OBJECTIVE

The following field utilities are used with the classification interface during online training; we omit the interface subscript ι. For each update, the old policy is the snapshot that generated the response group, whereas the frozen first-stage policy serves as the KL reference. For input $\boldsymbol{z}_i$, the old policy samples M_i responses $\boldsymbol{y}_{i,j} = (\chi_{i,j}, \mathcal{B}_{i,j}, \widehat{v}_{i,j}) \sim \pi_{\mathrm{old}}(\cdot \mid \boldsymbol{z}_i)$. Let $\mathbf{C}^{(g)} \in \mathbb{R}_{\geq 0}^{|\mathcal{V}_g| \times |\mathcal{V}_g|}$ be the misclassification cost matrix for input form g. Its diagonal satisfies $[\mathbf{C}^{(g)}]_{v,v} = 0$, while off-diagonal entries assign relative costs to different errors. Verdict utility is

$$r^{\mathrm{v}}_{i,j} = \begin{cases} \zeta_v, & \widehat{v}_{i,j} = v^\star_i, \\ -\zeta_v \big[\mathbf{C}^{(g_i)}\big]_{v^\star_i, \widehat{v}_{i,j}}, & \widehat{v}_{i,j} \neq v^\star_i, \end{cases} \qquad \zeta_v > 0. \tag{14}$$

Region utility uses one-dimensional GIoU to score matched intervals and separate penalties for unmatched predicted and repair regions. For closed line intervals $b = [\ell_b, u_b]$ and $\gamma = [\ell_\gamma, u_\gamma]$, let

$$\begin{aligned} I(b,\gamma) &= \max\{0, \min(u_b, u_\gamma) - \max(\ell_b, \ell_\gamma) + 1\}, \\ U(b,\gamma) &= |b| + |\gamma| - I(b,\gamma), \\ E(b,\gamma) &= \max(u_b, u_\gamma) - \min(\ell_b, \ell_\gamma) + 1, \end{aligned}$$

where $|b| = u_b - \ell_b + 1$ and $E(b,\gamma)$ is the length of the shortest closed interval covering both intervals. One-dimensional generalized IoU is

$$\mathrm{GIoU}_{\mathrm{1D}}(b,\gamma) = \frac{I(b,\gamma)}{U(b,\gamma)} - \frac{E(b,\gamma) - U(b,\gamma)}{E(b,\gamma)}.$$

Let $N_{i,j} = |\mathcal{B}_{i,j}|$, $K_i = |\Gamma^\star_i|$, and $\xi_{i,j,a,h} = \mathrm{GIoU}_{\mathrm{1D}}(b_{i,j,a}, \gamma^\star_{i,h})$. Given $\tau_{\mathrm{match}} \in [-1, 1]$, region pairs are sorted by decreasing $\xi_{i,j,a,h}$, with ties broken by predicted-region index and then repair-region index. We retain a pair when $\xi_{i,j,a,h} \geq \tau_{\mathrm{match}}$ and neither endpoint has been used by an earlier pair, producing the thresholded one-to-one matching $\mathcal{M}_{i,j}$. Unmatched predicted and repair regions become false positives and false negatives, respectively. Denote the line sets covered by predicted and repair regions as

$$\Lambda(\mathcal{B}_{i,j}) = \bigcup_{a=1}^{N_{i,j}} \Lambda(b_{i,j,a}), \qquad \Lambda(\Gamma^\star_i) = \bigcup_{\gamma \in \Gamma^\star_i} \Lambda(\gamma).$$

The numbers of unmatched regions and the recall of repaired lines are

$$\begin{aligned} F^{\mathrm{pred}}_{i,j} &= N_{i,j} - |\mathcal{M}_{i,j}|, & F^{\mathrm{gold}}_{i,j} &= |\Gamma^\star_i| - |\mathcal{M}_{i,j}|, \\ R^{\mathrm{line}}_{i,j} &= \frac{|\Lambda(\mathcal{B}_{i,j}) \cap \Lambda(\Gamma^\star_i)|}{|\Lambda(\Gamma^\star_i)|}. \end{aligned}$$

Region utility for a failed example is

$$r^{\mathrm{B}}_{i,j} = \sum_{(a,h) \in \mathcal{M}_{i,j}} \xi_{i,j,a,h} - \lambda_{\mathrm{fp}} F^{\mathrm{pred}}_{i,j} - \lambda_{\mathrm{fn}} F^{\mathrm{gold}}_{i,j} - \lambda_{\mathrm{cov}}(1 - R^{\mathrm{line}}_{i,j}), \tag{15}$$

where $\lambda_{\mathrm{fp}}, \lambda_{\mathrm{fn}}, \lambda_{\mathrm{cov}} \geq 0$. A successful example satisfies $\Gamma^\star_i = \varnothing$, and its region utility is

$$r^{\mathrm{B}}_{i,j} = \begin{cases} b_{\mathrm{empty}}, & N_{i,j} = 0, \\ -\lambda_{\mathrm{sp}} N_{i,j}, & N_{i,j} > 0, \end{cases} \qquad b_{empty} > 0, \quad \lambda_{sp} \geq 0.$$

The local utility of each region block is determined by the quality of its match to a repair region. A prediction that matches no repair region is treated as a false positive:

$$s^{\mathrm{B}}_{i,j,a} = \begin{cases} \xi_{i,j,a,h}, & (a,h) \in \mathcal{M}_{i,j}, \\ -\lambda_{fp}, & a \notin \{a' \mid \exists h,\ (a',h) \in \mathcal{M}_{i,j}\}. \end{cases}$$

For a failed example, the aggregate region utility can be written as

$$r^{\mathrm{B}}_{i,j} = \sum_{a=1}^{N_{i,j}} s^{\mathrm{B}}_{i,j,a} - \lambda_{\mathrm{fn}} F^{\mathrm{gold}}_{i,j} - \lambda_{\mathrm{cov}}(1 - R^{\mathrm{line}}_{i,j}).$$

The online objective assigns the first term to individual region blocks and the latter two terms to the verdict field. Let $N_i^{\mathrm{B}} = \sum_j N_{i,j}$. The group mean of region utility is

$$\bar{s}_i^{\mathrm{B}} = \begin{cases} (N_i^{\mathrm{B}})^{-1} \sum_j \sum_{a=1}^{N_{i,j}} s_{i,j,a}^{\mathrm{B}}, & N_i^{\mathrm{B}} > 0, \\ 0, & N_i^{\mathrm{B}} = 0. \end{cases}.$$

Centering within a candidate group captures relative quality across regions. A fixed reference $\vartheta_g \in \mathbb{R}$ preserves an absolute notion of quality when the group mean exceeds the fixed level. Given $\omega_g \in [0,1]$, the region reference is

$$\mu_i^{\mathrm{B}} = \begin{cases} (1-\omega_{g_i})\bar{s}_i^{\mathrm{B}} + \omega_{g_i}\vartheta_{g_i}, & \bar{s}_i^{\mathrm{B}} \geq \vartheta_{g_i}, \\ \bar{s}_i^{\mathrm{B}}, & \bar{s}_i^{\mathrm{B}} < \vartheta_{g_i}. \end{cases}.$$

Let $A_{i,j,a}^{\mathrm{B}} = \alpha_B(s_{i,j,a}^{\mathrm{B}} - \mu_i^{\mathrm{B}})$.

For failed records with visible repair evidence, missed repair regions and uncovered repaired lines enter the verdict utility; successful examples use the empty-region requirement:

$$s_{i,j}^{\mathrm{v}} = \begin{cases} r_{i,j}^{\mathrm{v}} - \lambda_{\mathrm{fn}} F_{i,j}^{\mathrm{gold}} - \lambda_{\mathrm{cov}}(1 - R_{i,j}^{\mathrm{line}}), & v_i^\star \notin \mathcal{V}_{g_i}^+, \\ r_{i,j}^{\mathrm{v}} + \gamma_+ r_{i,j}^{\mathrm{B}}, & v_i^\star \in \mathcal{V}_{g_i}^+, \end{cases} \qquad \gamma_+ \geq 0.$$

γ_+ controls the weight of empty-region feedback for successful examples. If a successful response generates an additional region, local utility acts on that region block and the set-level empty-region utility acts on the verdict field. Equation (9) constructs field-wise token advantages from $s_{i,j}^{\mathrm{v}}$, $s_{i,j,a}^{\mathrm{B}}$, and μ_i^{B}.

Equation (10) defines a length-normalized ratio relative to the old policy; the frozen structured-supervision model supplies the KL reference. Equation (11) gives the objective, and Appendix F derives its position-wise gradients.

A.6 Evaluation metrics

Let $\mathcal{D}$ be the evaluated record set; for repository data, one record is a patch-linked view whose label is the source patch outcome. Let $\mathcal{E} \subseteq \mathcal{D}$ contain failed records with visible repair regions, and let $\widehat{v}_i \in \mathcal{V}_{g_i}$ be the model verdict. Define the binary success map

$$c(v) = \begin{cases} 1, & v \in \{\mathrm{AC}, \mathrm{PASS}\}, \\ 0, & v \in \{\mathrm{WA}, \mathrm{TLE}, \mathrm{CE}, \mathrm{FAIL}\}. \end{cases}$$

Exact task-label accuracy requires the prediction to equal the execution verdict, while binary accuracy requires only the same success status:

$$A_{\mathrm{exact}} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbf{1}[\widehat{v}_i = v_i^\star], \qquad A_{\mathrm{bin}} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbf{1}[c(\widehat{v}_i) = c(v_i^\star)].$$

For predicted regions $\mathcal{B}_i$ and subsequent repair regions $\Gamma_i^\star$, define

$$I_{err} = \frac{1}{|\mathcal{E}|} \sum_{i \in \mathcal{E}} \frac{|\Lambda(\mathcal{B}_i) \cap \Lambda(\Gamma_i^\star)|}{|\Lambda(\mathcal{B}_i) \cup \Lambda(\Gamma_i^\star)|},$$
$$G = \sqrt{A_{\mathrm{exact}} I_{err}}.$$

$\mathcal{B}_i$ denotes the response regions represented as closed intervals; successful examples use an empty region set. The joint score G is the geometric mean of verdict accuracy and repair-region alignment. Holding one component fixed, G increases monotonically with the other; if either component is 0, then $G = 0$. Balanced accuracy for binary verdicts is

$$\mathrm{BAcc} = \frac{1}{2}\big(\mathrm{TPR} + \mathrm{TNR}\big),$$

where TPR and TNR are the recalls of successful and failed examples, respectively.

For a candidate set $\mathcal{C}_i$ belonging to one task, let $Y(c) \in \{0,1\}$ denote the final outcome of candidate c, let $u_{\boldsymbol{\phi}}(c)$ denote the verdict readout used in the current experiment, and let

$\mathcal{C}_i^{\max} = \arg\max_{c \in \mathcal{C}_i} u_{\boldsymbol{\phi}}(c)$ be the set of highest-scoring candidates. Under uniform tie-breaking, expected Top-1 is

$$\mathrm{Top1} = \mathbb{E}_i \left[\frac{1}{|\mathcal{C}_i^{\max}|} \sum_{c \in \mathcal{C}_i^{\max}} Y(c) \right].$$

For $L \geq 2$ fixed observation depths $0 < d_1 < \cdots < d_L \leq 1$, write $\boldsymbol{d} = (d_1, \ldots, d_L)$. Let b_ℓ be thresholded balanced accuracy and $b_\ell^{\exp}$ its expected value under the model's binary success score at depth d_ℓ. For $x = (x_1, \ldots, x_L)$, define

$$\mathrm{Trap}_{\boldsymbol{d}}(x) = \frac{1}{d_L - d_1} \sum_{\ell=1}^{L-1} \frac{d_{\ell+1} - d_\ell}{2} \big(x_\ell + x_{\ell+1}\big).$$

Trajectory balanced accuracy is summarized by $\mathrm{Trap}_{\boldsymbol{d}}(b)$, and EBA by $\mathrm{Trap}_{\boldsymbol{d}}(b^{\exp})$.

A.7 Prompts and scoring for different code states

The four code-state interfaces differ in what is visible and in the level at which the outcome is defined. Complete programs, editing turns, and patch-linked repository views also return repair-relevant regions; trajectory prefixes use a verdict and a short rationale because they do not expose stable code coordinates. For repository code, the prompt evaluates the displayed snapshot, while PASS/FAIL is the project-test result attached to its source patch; repair intervals are taken from the lines visible in that snapshot. Table 6 lists the inputs and outputs.

Prompts A.1–A.4 give the system messages for the four interfaces. The trajectory input ends at a fixed truncation point, while its label is determined by the final trajectory outcome. The templates serialize the fields used by each interface.

Prompt A.1: Complete program

```
You are an expert competitive programming judge.

Analyze the submission and determine its verdict.

Respond with a single JSON block:
{
  "verdict": "<AC|WA|TLE|CE>",
  "reason": "<one sentence>",
  "evidence": [
    {
      "start_line": <1-indexed integer>,
      "end_line": <1-indexed integer>,
      "comment": "<why this span supports the verdict>"
    }
  ]
}

Rules:
- verdict must be exactly one of AC, WA, TLE, or CE.
- For AC, evidence must be [] exactly.
- For non-AC, provide up to five suspicious line ranges.
- Each range must satisfy start_line <= end_line.
- Keep the most relevant evidence first.
```

Editing turns use the program schema with the competitive editing-turn labels AC, WA, and CE, together with the verdict semantics in Prompt A.2.

Prompt A.2: Verdict semantics for an editing turn

```
AC means that the current submission satisfies the requirement
and passes all unit tests shown below. A failing test, uncaught
exception, or missing required function is WA; a syntax or import
error is CE.
```

Table 6: Inputs and outputs for four code states. Code and diff lines are 1-indexed; predicted regions are closed intervals.

Code state	Model input	Output
Complete program	Problem, language, numbered code, and optional examples	AC/WA/TLE/CE; rationale; up to five relevant intervals
Editing turn	Current requirement, visible unit tests, and numbered code	AC/WA/CE; rationale; relevant intervals
Repository patch	Optional requirement and numbered candidate-file snapshot	PASS/FAIL; rationale; relevant code intervals
Trajectory prefix	Task, agent messages, edits, commands, and tool feedback up to a fixed depth	AC/WA; one-sentence rationale

Prompt A.3: Patch-linked code view

```
You are CodeVerifier, a strict project-level code verifier for Python and C++ repository
     code.

Goal:
Judge whether the candidate file snapshot contains a concrete defect that should be marked
     FAIL, or whether it is acceptable as PASS. The task is binary, but the review must be
     careful and evidence-driven.

Internal review checklist:
1. Local contract: infer what each visible function, class, branch, and API call promises
     from names, parameters, return values, and nearby usage.
2. Correctness: trace the important control-flow paths, including normal inputs, empty
     inputs, boundary values, duplicated values, unicode or binary data, large inputs, and
     missing optional arguments.
3. State and side effects: check whether mutations, caches, global state, filesystem/
     network effects, and object lifetimes stay consistent across all visible paths.
4. Runtime safety: look for unhandled exceptions, null/None dereferences, index/key errors,
      invalid casts, resource leaks, lock/order mistakes, and paths that skip cleanup.
5. API compatibility: verify that calls use the expected argument shape, return type,
     encoding, units, ordering, and exception behavior expected by the surrounding project.
6. Security and robustness: consider injection, path traversal, unsafe deserialization,
     credential leakage, permission bypass, weak validation, and adversarial input handling.

7. Performance: flag only severe visible issues such as unbounded memory growth, accidental
      quadratic behavior on ordinary inputs, infinite loops, or blocking work in a hot path.

8. Build validity: for Python, check parse/import-time hazards visible in the file; for C
     ++, check syntax, type/API misuse, lifetime, ownership, undefined behavior, and
     compile/link-visible mistakes.

Verdict policy:
- FAIL only when there is a specific, defensible defect visible in the shown code, or
     visible by comparing the code to the optional requirement when a requirement section
     is present. The defect must be localizable to one or more concrete line ranges.
- PASS when no concrete defect is visible, even if the code is ugly, incomplete-looking,
     under-commented, or could be refactored.
- Do not fail for style, formatting, naming, missing tests, missing documentation, TODO
     comments, or speculative missing features.
- Use only the optional requirement and the code block. Do not use or infer repository name
     , file path, source dataset, tests, issue titles, logs, or any other external metadata.

- If you cannot explain how the shown line(s) cause the defect, choose PASS.

Evidence policy for FAIL:
- Evidence must identify the smallest line range(s) that carry the defect.
- Prefer executable or load-bearing logic lines over surrounding comments, blank lines,
     imports, or formatting.
- Use multiple ranges only when the defect genuinely depends on multiple non-contiguous
     locations.
- Do not include evidence for PASS.

Comment policy:
- The top-level reason gives the one-sentence overall judgment.
- For FAIL, every evidence item must include a non-empty comment that explains why that
     exact line range proves or carries the defect. This is required, not optional.
- The evidence comment should name the broken invariant, wrong API use, unsafe state
     transition, missing edge-case handling, or other concrete cause. Do not merely repeat
     the line number.
- For PASS, evidence must be empty, so the top-level reason is the only commentary.

Output requirements:
```

Prompt A.3: Patch-linked code view (continued)

```
Think through the checklist internally. Do not reveal chain-of-thought, scratch notes, or
     step-by-step reasoning. Return exactly one JSON block and nothing else.

```json
{
  "verdict": "<PASS|FAIL>",
  "reason": "<one sentence explaining the decisive issue or why no concrete defect is
     visible>",
  "evidence": [
    {"start_line": <1-indexed int>, "end_line": <1-indexed int>, "comment": "<why this line
      range causes or proves the defect>"}
  ]
}
```

Hard schema rules:
- verdict must be exactly PASS or FAIL.
- reason must be exactly one sentence.
- For PASS, evidence must be [] exactly.
- For FAIL, evidence must contain 1 to 5 ranges.
- For FAIL, every evidence item must have a non-empty comment explaining the cause.
- Every range must satisfy 1 <= start_line <= end_line <= total shown lines.
- For a single-line defect, set start_line == end_line.
- Keep the most important evidence first.

Review Task
Determine whether the candidate code below contains a concrete defect. Use only the
     optional requirement above, if present, and the numbered code block below. Ignore all
     repository metadata and external context.

Use this structured audit order internally:
1. Infer the visible contract of the code.
2. Trace normal, boundary, and adversarial inputs.
3. Check state mutation, resource cleanup, and concurrency assumptions.
4. Check API compatibility and exception behavior.
5. Check security, robustness, severe performance, and build validity.
6. Commit to PASS or FAIL using only defects that can be tied to line numbers in the
     rendered code block.

Return only the final JSON block described above.
```

Prompt A.4: Trajectory prefix

```
You are an expert judge of coding-agent progress.

You will see the first part of an agent's interaction log on a
GitHub issue. The log is truncated and the agent continued
afterwards. Based on the approach, understanding, edits, and
verification so far, predict whether the agent's final patch will
resolve the issue and pass hidden tests.

Respond with a single JSON block:
{
  "verdict": "<AC|WA>",
  "reason": "<one sentence>"
}

AC means the trajectory is on track to produce a resolving patch.
WA means the observed trajectory indicates that the final patch
will fail.
```

Repository-region evaluation compares predicted spans with repair-associated locations within the visible code. Repository patches use the binary PASS/FAIL interface. Online optimization and region learning use this interface. For BigCodeBench complete functions and CollabLLM editing states, a single discrete judgment gives balanced accuracy (BAcc); base-model comparisons on the editing states report ROC AUC; within-task candidate selection uses expected Top-1.

Given M independent structured judgments $\boldsymbol{y}^{(m)} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{z}_g, \iota)$, the finite-sample estimate of Equation (2) is

$$\widehat{s}_{\boldsymbol{\phi}}^{\mathrm{gen},g,\iota,M}(\boldsymbol{z}_g) = \frac{1}{M}\sum_{m=1}^{M} \nu_{g,\iota}\Big(\mathrm{verdict}(\boldsymbol{y}^{(m)})\Big). \tag{16}$$

Let v^+ and v^- denote the success and failure labels of a binary interface. Given the verdict-field prefix $\boldsymbol{p}_v$, the conditional log-score of the complete label sequence is

$$\ell_{\boldsymbol{\phi}}(v^{\pm}; \boldsymbol{z}_g, \iota) = \sum_{r=1}^{|v^{\pm}|} \log \pi_{\boldsymbol{\phi}}\big(v_r^{\pm} \mid \boldsymbol{z}_g, \boldsymbol{p}_v, v_{<r}^{\pm}\big).$$

The score margin between the success and failure labels is

$$m_{\boldsymbol{\phi}}^{g,\iota}(\boldsymbol{z}_g) = \ell_{\boldsymbol{\phi}}(v^+; \boldsymbol{z}_g, \iota) - \ell_{\boldsymbol{\phi}}(v^-; \boldsymbol{z}_g, \iota). \tag{17}$$

Let $\sigma(x) = (1 + e^{-x})^{-1}$. The normalized PASS-label score for a binary interface is

$$s_{\boldsymbol{\phi}}^{\mathrm{lab},g,\iota}(\boldsymbol{z}_g) = \sigma(m_{\boldsymbol{\phi}}^{g,\iota}(\boldsymbol{z}_g)). \tag{18}$$

Continuous-score AUC uses the normalized success-label score in Equation (3); CoderForge edit-response analysis uses the PASS–FAIL margin $m_{\boldsymbol{\phi}}^{g,\iota}$, and repository rewards use the normalized PASS-label score. Table 7 gives the scoring procedure and judgment budget for each experiment.

Table 7: Inference and scoring protocols used in the experiments.

Evaluation	Score	Inference budget
Main test; direct comparisons; CodeContests; LiveCodeBench and TACO efficiency	Discrete task verdict; exact or binary label	One deterministic generation
CodeJudge comparison	Binary verdict from the published two-pass procedure	Two deterministic generations
BigCodeBench complete functions, CollabLLM editing states, and within-task selection	Discrete task verdict; expected Top-1	One deterministic generation
HumanEvalPack pairs	Mean binary verdict utility, followed by pairwise comparison	Five independent generations
Editing-state AUC	Normalized success-label score in Eq. (3)	One conditional label-score read
CoderForge edit response	PASS–FAIL label margin in Eq. (17)	One conditional label-score read
LiveCodeBench candidate selection	Mean binary verdict utility	Five independent generations per candidate
Competitive-programming policy reward	Binary verdict utility	One deterministic generation
Project-repair policy reward	Mean normalized PASS-label score across code views	One label-score read per view

B RQ1: Evidence response of verdict and repair-region alignment

B.1 Directional conflict between verdict and region feedback

Proposition 1 (Rank-one bottleneck of a shared scalar)**.** *Let $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_F)^\top$ be the advantages of $F \geq 2$ fields in one response, with aggregation weights $\boldsymbol{\lambda} \neq \mathbf{0}$. If the advantages are first aggregated as $u = \boldsymbol{\lambda}^\top \boldsymbol{\delta}$ and u is then assigned to every field, the resulting coefficient vector is*

$$\widetilde{\boldsymbol{\delta}} = \mathbf{1}\boldsymbol{\lambda}^\top \boldsymbol{\delta}.$$

The map $\mathbf{1}\boldsymbol{\lambda}^\top$ has rank 1, and its null space $\ker(\boldsymbol{\lambda}^\top)$ has dimension $F - 1$. If each field retains its own advantage, the coefficient map is the identity and none of these directions is lost.

For two fields, write their advantages as a and b, and let $\boldsymbol{\lambda} = (\lambda_a, \lambda_b)^\top$ with $\lambda_a, \lambda_b > 0$, so that $u = \lambda_a a + \lambda_b b$. If $ab < 0$, then

$$\frac{|u|}{\lambda_a|a| + \lambda_b|b|} = \frac{\big|\lambda_a|a| - \lambda_b|b|\big|}{\lambda_a|a| + \lambda_b|b|} < 1.$$

When the two weighted magnitudes are equal, $u = 0$. Otherwise, the sign of u is set by the larger weighted magnitude and opposes the advantage of the other field. Even if a common coefficient c may be chosen freely, its best squared approximation to (a, b) satisfies

$$\min_{c\in\mathbb{R}} \big[(c-a)^2 + (c-b)^2\big] = \frac{(a-b)^2}{2},$$

with the optimum attained at $c = (a+b)/2$.

Proof. $\mathbf{1}\boldsymbol{\lambda}^\top$ is the outer product of two nonzero vectors and therefore has rank 1. The rank–nullity theorem gives $\dim\ker(\boldsymbol{\lambda}^\top) = F - 1$. When $ab < 0$, the two weighted terms have opposite signs, which directly yields the magnitude ratio and sign claims. Differentiating with respect to c gives $2(c-a) + 2(c-b) = 0$; substituting $c = (a+b)/2$ gives the minimum error. □

In the candidate-set analysis, verdict utility records whether a response gives the correct label, and region utility is the IoU between the line sets of predicted and repair regions. Centering the two utilities separately within the same candidate group yields the diagnostic signals A^v and A^B, for which we set $\alpha_v = \alpha_B = 1$. Let

$$\eta_{\mathrm{sc}} = \frac{|A^{\mathrm{v}} + A^{\mathrm{B}}|}{|A^{\mathrm{v}}| + |A^{\mathrm{B}}|}, \qquad |A^{\mathrm{v}}| + |A^{\mathrm{B}}| > 0,$$

measure the relative magnitude retained after the two signals are added. Error bars in the figure are standard deviations from candidate-group cluster resampling. Among candidates for which both signals are nonzero, $(18.15 \pm .76)\%$ have opposite signs. As their magnitudes approach one another, η_{sc} decreases, consistent with the cancellation relation in Proposition 1.

The shared-routed control preserves the field positions that receive feedback but assigns one response-level advantage to them. Verdict accuracy and region IoU both decline at all three scales. The comparison therefore isolates field-specific advantage assignment while keeping the updated positions fixed (Table 1).

B.2 Score changes under bidirectional edits

Bidirectional edits form program pairs under the same task, prompt, and test suite, separating verdict changes caused by correctness from those caused by code rewriting in general. Defect injection starts from a correct program. After retesting, a program that remains correct enters $\mathcal{B}_0$, while one that becomes incorrect enters $\mathcal{B}_-$. Repair starts from an incorrect program; a successful repair enters $\mathcal{F}_+$, and a program that remains incorrect enters $\mathcal{F}_0$. Let $\boldsymbol{y}_{\boldsymbol{\phi}}(c)$ be the model's deterministic response for candidate c, with verdict utility $u_{\boldsymbol{\phi}}(c) = \mathbf{1}[\mathrm{verdict}(\boldsymbol{y}_{\boldsymbol{\phi}}(c)) \in \mathcal{V}^+_{\mathsf{sol}}]$. The utility change after an edit is $\delta_i = u_{\boldsymbol{\phi}}(c_i^{\mathrm{tgt}}) - u_{\boldsymbol{\phi}}(c_i^{\mathrm{src}})$, and we define

$$D_{\mathrm{fix}} = \mathbb{E}[\delta_i \mid i \in \mathcal{F}_+] - \mathbb{E}[\delta_i \mid i \in \mathcal{F}_0],$$
$$D_{\mathrm{bug}} = \mathbb{E}[\delta_i \mid i \in \mathcal{B}_0] - \mathbb{E}[\delta_i \mid i \in \mathcal{B}_-].$$

We combine the two directions as $D_{\leftrightarrow} = (D_{\mathrm{fix}} + D_{\mathrm{bug}})/2$. For edits that change correctness, let $\sigma_i = 1$ when $i \in \mathcal{F}_+$ and $\sigma_i = -1$ when $i \in \mathcal{B}_-$, and define

$$D_{\mathrm{flip}} = \mathbb{E}[\sigma_i \delta_i \mid i \in \mathcal{F}_+ \cup \mathcal{B}_-].$$

$D_{\mathrm{flip}} > 0$ means that a verdict rises when correctness improves and falls when correctness is harmed. Every paired edit shares its task and test suite, and execution confirms both model-generated and human repairs.

At every model scale, scores rise after a successful repair and fall after an edit makes a program fail (Table 8). Test-verified model repairs produce the same directional change (Table 9).

Table 8: Direction-aligned score effects across scales. Positive values indicate repair gains or defect penalties; values are estimates ± one task-bootstrap SD.

Size	$D_{\leftrightarrow}$	D_{flip}
2B	$+.161 \pm .054$	$+.307 \pm .047$
4B	$+.291 \pm .054$	$+.307 \pm .047$
♦ 9B	$\mathbf{+.312 \pm .049}$	$\mathbf{+.423 \pm .042}$

Table 9: 9B score changes under controlled edits and test-verified repairs. D_{fix}, D_{bug}, and D_{flip} follow the aligned directions; the last row is post-edit minus pre-edit.

Estimand	Score change
Successful-repair effect D_{fix}	$+.389$
Correctness-degrading edit effect D_{bug}	$+.236$
Correctness-flip effect D_{flip}	$+.423$
Execution-verified model repair	$+.541$

B.3 Region placement and verdict disagreement

Across five defect transformations, model-predicted regions hit the actual injection location substantially more often than length-matched random intervals. Final verdicts are also more likely to be wrong when repeated judgments disagree. These diagnostics test whether regions fall on the modified code and whether consistency across judgments reflects the difficulty of a verdict.

B.4 Evidence locations in human repairs

Region predictions are evaluated against locations subsequently changed by the programmer. UOJ-Bench pairs an erroneous submission with a later accepted version from the same user, providing both the failed state and the actual repair. The model reads the problem and erroneous program, while a line-level difference between the two versions marks the edited locations. Replacements and deletions map to lines in the original code, and pure insertions are anchored at the adjacent boundary.

We compare predicted regions with subsequent repairs from three perspectives. Edited-line AUC measures whether the model ranks subsequently changed lines above other lines. The symmetric-distance score measures proximity between the two line sets in both directions, and first-region IoU measures the overlap between the model's first output region and the actual edit location.

Let L_x be the number of lines in erroneous program x, let E_x be the evidence lines predicted by the model, and let D_x be the lines touched by the subsequent repair. Write $\bar{D}_x = \{1, \ldots, L_x\} \setminus D_x$. For an example with $0 < |D_x| < L_x$, the line score is

$$\psi_x(\ell) = -\min_{e \in E_x} |\ell - e|.$$

Treating edited lines as positive and all other code lines as negative, line-level AUC is

$$R_x = \frac{1}{|D_x||\bar{D}_x|} \sum_{d \in D_x} \sum_{u \in \bar{D}_x} \left(\mathbf{1}[\psi_x(d) > \psi_x(u)] + \frac{1}{2}\mathbf{1}[\psi_x(d) = \psi_x(u)] \right).$$

$R_x = 0.5$ means that the evidence does not distinguish edited from unedited lines. The normalized directed distance between two sets is

$$d_x(S, T) = \frac{1}{|S|} \sum_{s \in S} \frac{\min_{t \in T} |s - t|}{\max(1, L_x - 1)}.$$

The symmetric-distance score is

$$A_x = 1 - \frac{d_x(E_x, D_x) + d_x(D_x, E_x)}{2}.$$

First-region IoU uses only the first interval $E_{x,1}$ produced by the model. Let $\mathcal{T}_x$ be the set of valid translations of this interval within the same file, including its original position. Then

$$I_x = \frac{|E_{x,1} \cap D_x|}{|E_{x,1} \cup D_x|},$$
$$\widetilde{I}_x = \frac{1}{|\mathcal{T}_x|} \sum_{\tau \in \mathcal{T}_x} \frac{|(E_{x,1} + \tau) \cap D_x|}{|(E_{x,1} + \tau) \cup D_x|}.$$

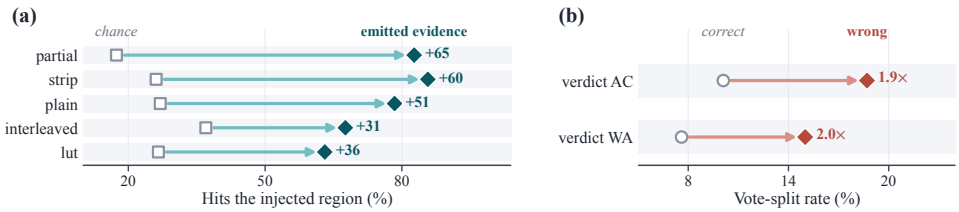

Figure 8: Region placement and verdict consistency. (a) Hits on injected regions versus length-matched translations. (b) Five-judgment disagreement by final verdict.

Each metric is computed per trajectory and then averaged over trajectories. The final column reports the mean of $I_x - \widetilde{I}_x$. Because the translation control preserves interval length and shape, this difference measures the additional overlap obtained at the actual evidence location relative to other positions in the same file.

Table 10: Repair-region alignment on same-user repair trajectories; arrows compare Base → CODEVERIFIER and the final column uses within-file translations.

Size	Edited-line AUC	Symmetric distance	First-region IoU	Location delta
2B	.5750 → .5824	.7270 → .7591	.0446 → .0566	+.0349
4B	.5927 → .6077	.7985 → .8005	.0452 → .0642	+.0428
♦ 9B	.6053 → **.6189**	.7053 → **.8183**	.0560 → **.0671**	**+.0478**

At all three scales, the first predicted region lies closer to the programmer's subsequent edit than a translated region in the same file, and the correspondence strengthens with model scale (Table 10).

C RQ2: JUDGMENT QUALITY AND EXECUTION COST

C.1 EXECUTION AVAILABILITY AND TEST LATENCY

Figure 1(a,b) uses the same candidate groups and per-test records. Each group contains both correct and incorrect programs. After a given number of tests, execution remains nondiscriminative for a group if every candidate has the same result. Candidate latency is measured along the same execution path, and the figure reports the median, 95th percentile, and 99th percentile.

Figure 1(c) summarizes code states at each interaction turn in the multi-turn BigCodeBench CollabLLM trajectories (Zhuo et al., 2025; Wu et al., 2025). Each point is normalized by the trajectories that reach that turn. A state counts as callable only when the current code already exposes the entry point required by final testing. Most trajectory states have not yet formed such an entry point.

Figure 5(a,c) uses candidates generated for the same LiveCodeBench problem by several models, together with their per-test records. Panel (a) compares judgment quality and amortized batch-processing time across the three model scales. Panel (c) fixes five-candidate pools, identifies the highest-scoring tie after the available tests, and then uses the model score to choose within that set. Panels (b,d) use complete TACO test records to compare measured judgment and execution time and to track how the number of tests needed to match mean model recall changes as errors become harder to trigger.

Complete per-test records from LiveCodeBench also allow direct analysis of errors that remain after limited execution. Of all incorrect candidates, 9.60% pass the first eight tests. The 2B, 4B, and 9B CODEVERIFIER models detect 67.24%, 79.65%, and 81.25% of these residual errors, respectively; the 9B recall matches the mean detection rate of eight uniformly sampled tests from each candidate's remaining suite.

For this residual comparison, the sampling frame is the remaining $M_c - 8$ tests after the fixed eight-test prefix; the TACO calibration below uses the complete suite. Suppose that the complete test suite for candidate c contains M_c tests, of which F_c expose its error. If k tests are drawn uniformly without replacement, the probability of detecting the error is

$$p_c(k) = 1 - \frac{\binom{M_c - F_c}{\min\{k, M_c\}}}{\binom{M_c}{\min\{k, M_c\}}}. \tag{19}$$

For TACO, the equivalent test count is the smallest integer k at which mean candidate detection probability reaches model error recall; it is the independent full-suite calibration used in Figure 5(d). Candidate selection on LiveCodeBench fixes the candidates, test order, and scoring positions across all three scales and enumerates the tied sets at every finite test budget.

Figure 5(a) measures processing time under saturated batching on one NVIDIA A100 80GB GPU. The 4B model captures most of the gain from scale, while 9B obtains higher balanced accuracy at a modest additional latency. TACO provides an independent full-suite calibration of matched recall. Under ideal 64-process CPU parallelization, execution still takes longer than model judgment at the same recall.

The first few tests usually remove obvious errors, yet some correct and incorrect candidates retain the same number of passes. CODEVERIFIER continues to rank these candidates, improving selection accuracy at every test budget for all three scales. As an incorrect program passes a larger fraction of its suite, more tests are needed to match the recall of CODEVERIFIER.

D RQ3: SUCCESS PREDICTION DURING EDITING

D.1 BIGCODEBENCH COMPLETE FUNCTIONS AND COLLABLLM EDITING STATES

BigCodeBench provides library-use tasks for complete-function evaluation, and CollabLLM trajectories on these tasks provide the editing states with execution outcomes (Zhuo et al., 2025; Wu et al., 2025). The former evaluate success prediction for finished code, while the latter evaluate whether the current state already satisfies the task. Comparison with base models on the same editing states separates the effect of model capacity from specialized training, with ROC AUC summarizing ranking quality.

CoderForge provides code states before and after real repository changes together with the project-test outcome of each patch (Ariyak et al., 2026). Let p_i denote candidate patch i, let $Y_i := Y_{p_i} \in \{0, 1\}$ be its project-test outcome, and let $\mathcal{H}_i$ be its set of modified units. The views $\boldsymbol{z}_{i,h}^{\mathrm{pre}}$ and $\boldsymbol{z}_{i,h}^{\mathrm{post}}$ are taken before and after modification h. The mean change in a candidate's label margin and the separation between successful and unsuccessful patches are

$$\Delta_i = \frac{1}{|\mathcal{H}_i|} \sum_{h \in \mathcal{H}_i} \left[m_{\boldsymbol{\phi}}^{\mathsf{repo,bin}}(\boldsymbol{z}_{i,h}^{\mathrm{post}}) - m_{\boldsymbol{\phi}}^{\mathsf{repo,bin}}(\boldsymbol{z}_{i,h}^{\mathrm{pre}}) \right], \quad D_{\mathrm{repo}} = \mathbb{E}[\Delta_i \mid Y_i = 1] - \mathbb{E}[\Delta_i \mid Y_i = 0].$$

A positive D_{repo} means that patches that eventually succeed receive a larger increase in PASS–FAIL label margin after editing. Table 2 reports BAcc on complete code and editing states, together with relative editing-state verdict-score AUC gains over the base models. Table 3 separates candidate sources and reports CoderForge edit outcomes. The two tables distinguish current-state judgment from judgment of the direction of a change.

E RQ4: CANDIDATE RANKING AND POLICY LEARNING

E.1 PUBLIC DATA AND COMPARISON PROTOCOL

All CodeJudge comparisons use binary success labels. Direct judgment and CodeJudge use Qwen3.5 at each matching scale, with CodeJudge following its published two-pass procedure. CODEVERIFIER receives the same task and code, and no method executes tests at judgment time. The teacher uses the same prompt, output structure, and examples as CODEVERIFIER. Figure 3 separates competitive-programming and repository results; Table 11 gives the exact values.

HumanEvalPack forms code pairs from a correct and an incorrect implementation of the same task, with both models receiving the same five judgments. Table 12 reports the complete pairwise results at all three scales.

Table 13 further examines the 9B model under changes in programming language, candidate source, and library-use requirements (Lambert et al., 2025; Li et al., 2022; Zhuo et al., 2025). The main HumanEvalPack comparison computes pairwise accuracy within each correct–incorrect program pair. Language slices retain continuous judgment strength, so the table summarizes ranking quality with AUC over verdict log-odds. CodeContests and BigCodeBench produce discrete success verdicts and

Table 11: Binary verdict accuracy (%) corresponding to Figure 3. ♦ marks CODEVERIFIER rows; CodeRM-NT and Qwen3.5-397B-A17B are separate references rather than size-matched entries.

Size	Method	Overall	Competition	Repository
2B	Qwen3.5	64.60	61.67	70.29
	CodeJudge	80.98	80.15	82.58
	♦ CODEVERIFIER	81.81	80.36	84.63
4B	Qwen3.5	75.96	81.84	64.55
	CodeJudge	77.91	79.30	75.20
	♦ CODEVERIFIER	84.32	83.95	85.04
9B	Qwen3.5	81.18	84.37	75.00
	CodeJudge	85.44	86.69	82.99
	♦ CODEVERIFIER	87.67	88.38	86.27
Ref.	CodeRM-NT	67.04	58.71	83.20
	Teacher	85.37	86.80	82.58

Table 12: RewardBench HumanEvalPack pairwise accuracy (%) with matched pairs and budgets.

Size	Base	CODEVERIFIER	Delta
2B	67.53	70.68	+3.15
4B	68.75	72.97	+4.22
♦ 9B	73.48	**78.15**	**+4.67**

therefore report balanced accuracy. Each base-model and CODEVERIFIER pair shares its prompt and scoring rule.

Table 13: Matched 9B comparisons under changes in programming language, candidate source, and library-use tasks. Relative gains are computed against the Base from unrounded values; shading tracks the gain (darker is larger).

Shift	Dataset / subset	Metric	Base	CODEVERIFIER	Rel. gain ↑
Language	HumanEvalPack Rust	AUC	.5980	.6850	+14.5%
	HumanEvalPack JavaScript	AUC	.7400	.7910	+6.8%
	HumanEvalPack Java	AUC	.7260	.7570	+4.3%
Candidate source	CodeContests human submissions	BAcc	.5237	.5883	+12.3%
	CodeContests model submissions	BAcc	.7497	.8668	+15.6%
Library use	BigCodeBench complete functions	BAcc	.6811	.7208	+5.8%

BigCodeBench tests within-problem selection, asking whether verdict scores rank a correct implementation first. Each pool contains correct and incorrect implementations of one task; ties receive the fraction of correct programs among the highest-scoring candidates. The main test set and independent complete-program evaluation measure overall success classification.

Table 14: Within-problem selection on BigCodeBench complete functions (%).

Size	Base	CODEVERIFIER	Delta
2B	55.56	61.11	+5.56
4B	54.76	59.13	+4.37
♦ 9B	57.54	**63.89**	**+6.35**

Within-problem Top-1 improves at all three scales, with the strongest result and largest gain at 9B. The discrete verdicts therefore help select a correct implementation from alternatives for the same task.

E.1.1 SELECTION OBJECTIVE

For a complete candidate c, let $Y(c) \in \{0, 1\}$ be its final correctness under a fixed complete test suite. The following analysis fixes a shared set of tasks and lets K be the largest common test budget recorded for every candidate. Candidates for the same task share the test order $\boldsymbol{h} = (h_1, \ldots, h_K)$. The outcome of test ℓ is $x_\ell(c) \in \{0, 1\}$, where 1 denotes a pass. After the first k tests, the number passed by a candidate is

$$e_k(c) = \sum_{\ell=1}^{k} x_\ell(c), \qquad 0 \le k \le K. \tag{20}$$

For the candidate group $\mathcal{C}_i$ of task i, candidates with the most passes form the set

$$T_{i,k} = \arg\max_{c \in \mathcal{C}_i} e_k(c). \tag{21}$$

Given a ranking function f, define the candidate selected from $T_{i,k}$ and its mean correctness as

$$\widehat{c}_{i,k}(f) \sim \mathrm{Unif}\left(\arg\max_{c \in T_{i,k}} f(c)\right), \qquad V_k(f) = \mathbb{E}_i\left[Y\left(\widehat{c}_{i,k}(f)\right)\right]. \tag{22}$$

For CODEVERIFIER, $f(c) = \widehat{s}_{\phi}^{\mathrm{gen,sol,bin},5}(\boldsymbol{z}(c))$. We first compare observed test results and then use mean utility over five binary judgments to distinguish candidates with the same number of passes.

$$c \succ_{hyb} c' \quad \Longleftrightarrow \quad \big(e_k(c), f(c)\big) >_{lex} \big(e_k(c'), f(c')\big). \tag{23}$$

Let $\widetilde{c}_{i,k}$ be a candidate chosen uniformly from $T_{i,k}$, let $H_{i,k}$ indicate whether that set contains both correct and incorrect programs, and let $\Delta_{i,k}(f)$ be the gain of within-set ranking over uniform selection. Then

$$\begin{aligned} \widetilde{c}_{i,k} &\sim \mathrm{Unif}(T_{i,k}), \\ H_{i,k} &= \mathbf{1}[\exists c, c' \in T_{i,k},\ Y(c) \neq Y(c')], \\ \Delta_{i,k}(f) &= \mathbb{E}\left[Y\left(\widehat{c}_{i,k}(f)\right) - Y(\widetilde{c}_{i,k})\right], \\ V_k(f) - V_k^{unif} &= \mathbb{E}_i[H_{i,k}\Delta_{i,k}(f)], \qquad V_k^{unif} = \mathbb{E}_i[Y(\widetilde{c}_{i,k})]. \end{aligned} \tag{24}$$

Equation (24) decomposes selection gain into the prevalence of mixed-correctness ties after limited testing and the ranking function's accuracy within them.

To express selection gains as a test budget, define for ranking function f after k executed tests

$$k^{\star}_{sel}(k, f) = \min\left\{k' \in \{0, \ldots, K\} \mid V_{k'}^{unif} \geq V_k(f)\right\}, \qquad \min \varnothing := +\infty. \tag{25}$$

$k^{\star}_{sel}(k, f)$ is the fewest tests needed to reach $V_k(f)$ from test outcomes alone, without a verifier. For finite $k^{\star}_{sel}(k, f) \geq k \geq 1$, the relative reduction in test calls is $1 - k/k^{\star}_{sel}(k, f)$.

E.1.2 COMPLETE CANDIDATE-SELECTION RESULTS

Expanding a candidate pool from two to five programs creates more groups that tie on observed tests but differ in final correctness. On the fixed five-candidate support, CODEVERIFIER improves selection across scales, pool sizes, and test counts; permuting scores within each group removes the gain (Table 15; Figure 9). Panel (a) of Table 15 uses the natural candidate support (933 groups and 4,492 candidates), whereas panels (b,c) and both frontier figures use a fixed five-candidate support (760 groups, 3,800 candidates, and 456 tasks) with exact subset enumeration.

Figure 10 shows the three-scale frontier over every test position and candidate-pool size.

E.2 POLICY LEARNING

The three competitive-programming training arms start from the same Qwen3.5-4B policy and use GRPO on the same tasks and update window. They use the mean pass rate over five tests, the released CodeRM-NT scalar, or one binary judgment from 4B CODEVERIFIER. For a policy-generated candidate c, let $x_\ell(c) \in \{0, 1\}$ indicate whether test ℓ passes, let $\boldsymbol{y}(c)$ be the response from CODEVERIFIER, and let $R_{\mathrm{NT}}(c)$ be the released CodeRM-NT score on the terminal code. The three rewards are

$$r_{test}(c) = \frac{1}{5}\sum_{\ell=1}^{5} x_\ell(c), \qquad r_{\mathrm{NT}}(c) = R_{\mathrm{NT}}(c), \qquad r_{\mathrm{CV}}(c) = \mathbf{1}[\mathrm{verdict}(\boldsymbol{y}(c)) = \mathrm{AC}].$$

Each arm is evaluated for accuracy on OpenR1 and LiveCodeBench v6 after the same update window for three paired random seeds.

Project-repair training uses the same Qwen3.5-4B policy, full-parameter GRPO on BugsInPy tasks, repositories separate from final evaluation, and three paired random seeds. The reward models are Qwen3.5-9B, the released Rishubi/CodeRM-NT model, and 9B CODEVERIFIER. Qwen and CODEVERIFIER provide normalized PASS probabilities; CodeRM-NT contributes its released raw

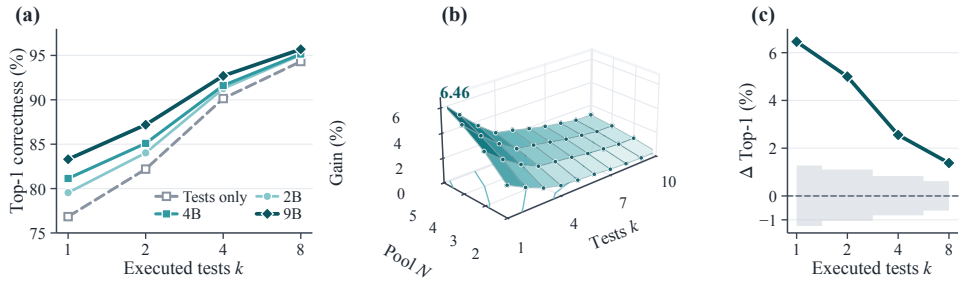

Figure 9: Candidate selection after limited testing on the fixed five-candidate support. (a) Top-1 using tests or CODEVERIFIER at three scales. (b) 9B gain over pool size N and test count k. (c) Observed and within-group-permuted gain intervals.

Table 15: Candidate selection with limited tests. (a) 9B equivalent-test operating points on the natural candidate support; (b) pool size and ties; (c) gains across scales on the fixed five-candidate support. Gains in (b,c) are absolute differences (percentage points) from uniform selection among execution-tied candidates; errors are task-bootstrap SDs.

(a) 9B natural-support equivalent test budget

k	+ CODEVERIFIER	$k^{\star}_{sel}$	Tests only	Fewer test calls
1	80.71%	3	85.02%	66.7%
2	84.42%	3	85.02%	33.3%
4	90.45%	6	91.35%	33.3%
8	94.25%	10	94.64%	20.0%

(c) Fixed five-candidate groups: absolute gain (pp)

Size	$k=1$	$k=2$	$k=4$	$k=8$
2B	+2.70	+1.83	+1.12	+.74
4B	+4.30	+2.90	+1.47	+.84
♦ 9B	**+6.46**	**+5.00**	**+2.55**	**+1.38**

(b) Fixed-support candidate-pool size and absolute gain (pp)

k	Mixed-label ties $2 \to 5$	2 candidates	5 candidates	Difference
1	17.79% → 46.18%	+2.57	+6.46	$+3.89 \pm .58$
2	12.97% → 35.26%	+1.82	+5.00	$+3.18 \pm .51$
4	7.03% → 20.53%	+.84	+2.55	$+1.72 \pm .35$
8	4.12% → 11.84%	+.43	+1.38	$+.95 \pm .28$

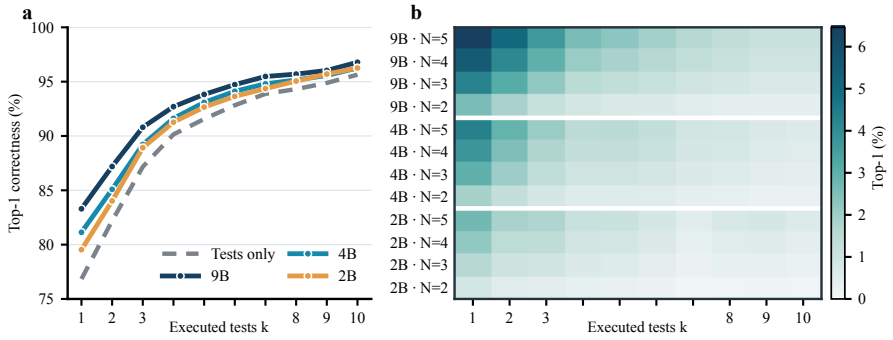

Figure 10: Candidate-selection frontier across scales on the fixed five-candidate support. (a) Top-1 using tests or CODEVERIFIER judgments. (b) Gain by pool size N and test count k.

scalar. Let $\mathcal{Z}(c)$ be the code views associated with a candidate patch and σ the logistic function. For $R \in \{\text{Qwen}, \text{CV}\}$, the normalized PASS-label score a_R and repository reward are

$$a_R(\boldsymbol{z}) = s_R^{\text{lab,repo,bin}}(\boldsymbol{z}) = \sigma\Big(m_R^{\text{repo,bin}}(\boldsymbol{z})\Big), \qquad r_R(c) = \frac{1}{|\mathcal{Z}(c)|} \sum_{\boldsymbol{z} \in \mathcal{Z}(c)} a_R(\boldsymbol{z}),$$

where $m_R^{\text{repo,bin}}$ is given by Equation (17). For CodeRM-NT, $r_{\text{NT}}(c) = R_{\text{NT}}(c)$ uses the released scalar on the terminal code under the same task interface. Final patch pass rate is evaluated on SWE-bench Verified (Jimenez et al., 2024).

An independent repository-repair study further examines score resolution among patches for the same task and the value of multiple samples. This diagnostic is separate from the normalized PASS-label policy comparison above. In all four runs, CODEVERIFIER assigns distinct scores to most same-task patch groups, and the resulting policies solve more tasks at every sampling budget (Figure 11).

Table 16: Final task success after training with different rewards (%). The Test/Qwen column is the execution-reward arm for competitive programming and the Qwen-reward arm for repository repair. Mean ± one sample SD is computed across three paired seeds.

Evaluation	Seed	Test/Qwen	CodeRM-NT	CODEVERIFIER
OpenR1	S42	20.76	19.07	**22.46**
	S43	21.61	22.03	**23.73**
	S44	20.34	20.34	**21.61**
	Mean	20.90 ± 0.65	20.48 ± 1.49	$\mathbf{22.60 \pm 1.07}$
LiveCodeBench v6	S42	28.57	29.14	**29.71**
	S43	29.14	29.71	**30.86**
	S44	27.43	28.00	**29.14**
	Mean	28.38 ± 0.87	28.95 ± 0.87	$\mathbf{29.90 \pm 0.87}$
SWE-bench Verified	S42	20.2	20.8	**22.6**
	S43	22.2	21.4	**22.8**
	S44	19.8	19.6	**20.8**
	Mean	20.7 ± 1.3	20.6 ± 0.9	$\mathbf{22.1 \pm 1.1}$

With five samples on the 107-task collection, the four trained policies solve 8.52 ± 1.49 tasks in expectation, compared with 5.49 for the untrained policy. The standard deviation is computed across the four trained runs.

F GRADIENTS OF FIELD-WISE ROUTING

Let $h_{i,j,t} = \log \pi_{\boldsymbol{\theta}}(y_{i,j,t} \mid \boldsymbol{z}_i, \boldsymbol{y}_{i,j,<t})$, let $h^{\mathrm{ref}}_{i,j,t} = \log \pi_{\mathrm{ref}}(y_{i,j,t} \mid \boldsymbol{z}_i, \boldsymbol{y}_{i,j,<t})$, and write $d_{i,j,t} = h^{\mathrm{ref}}_{i,j,t} - h_{i,j,t}$. The ratio $\widetilde{\rho}_{i,j,t}$ in Equation (10) satisfies $\widetilde{\rho}_{i,j,t} = \rho_{i,j}$. When differentiating, we treat sampled responses, non-padding indicators, field advantages, the old policy, and the reference policy as constants. Assume $0 < \epsilon_- < 1$ and $\epsilon_+ > 0$, and define the unclipped indicator

$$c^{\mathrm{clip}}_{i,j,t} = \begin{cases} 1, & A^{\mathrm{tok}}_{i,j,t} > 0 \text{ and } \rho_{i,j} < 1 + \epsilon_+, \\ 1, & A^{\mathrm{tok}}_{i,j,t} < 0 \text{ and } \rho_{i,j} > 1 - \epsilon_-, \\ 0, & \text{otherwise.} \end{cases} \tag{26}$$

Let $\mathcal{L}_{i,j}$ denote the $n_{i,j}^{-1} \sum_t$ term for response (i, j) in Equation (11). Then

$$\mathcal{L}_{\mathrm{RVPG}} = \frac{1}{|\mathfrak{B}|} \sum_{(i,j) \in \mathfrak{B}} \mathcal{L}_{i,j}.$$

Proposition 2 (Field-position gradient of the response-level surrogate)**.** *Away from clipping boundary points, the gradient contributed by response (i, j) at position s in Equation* (11) *is*

$$\frac{\partial \mathcal{L}_{i,j}}{\partial h_{i,j,s}} = \frac{w_{i,j,s}}{n_{i,j}} \left[-\rho_{i,j} c^{\mathrm{clip}}_{i,j,s} A^{\mathrm{tok}}_{i,j,s} + \beta \big(1 - e^{d_{i,j,s}}\big) \right]. \tag{27}$$

The task term at position s is determined by $A^{\mathrm{tok}}_{i,j,s}$, while all positions share the clipping scale $\rho_{i,j}$. At the upper boundary for a positive advantage or the lower boundary for a negative advantage, $c^{\mathrm{clip}}_{i,j,s}$ may be replaced by the corresponding subgradient coefficient in $[0, 1]$.

Proof. The stop-gradient operator treats $\mathrm{sg}[\rho_{i,j}]$ and $\mathrm{sg}[h_{i,j,t}]$ as constants. Therefore,

$$\frac{\partial \widetilde{\rho}_{i,j,t}}{\partial h_{i,j,s}} = \rho_{i,j} \mathbf{1}[s = t].$$

For a positive advantage, the clipped task term retains its gradient only when $\rho_{i,j} < 1 + \epsilon_+$. For a negative advantage, it retains its gradient only when $\rho_{i,j} > 1 - \epsilon_-$. These are precisely the two cases in Equation (26). Moreover, because $d_{i,j,t} = h^{\mathrm{ref}}_{i,j,t} - h_{i,j,t}$,

$$\frac{\partial}{\partial h_{i,j,s}} \big(e^{d_{i,j,t}} - d_{i,j,t} - 1\big) = \big(1 - e^{d_{i,j,t}}\big) \mathbf{1}[s = t].$$

Substituting both parts into the single-response term of Equation (11) yields Equation (27). □

Corollary 3 (Field-wise direction preservation under response-level clipping)**.** *For any $(i, j) \in \mathfrak{B}$ and any verdict or region position s satisfying $w_{i,j,s} = 1$, suppose that the task term at that position*

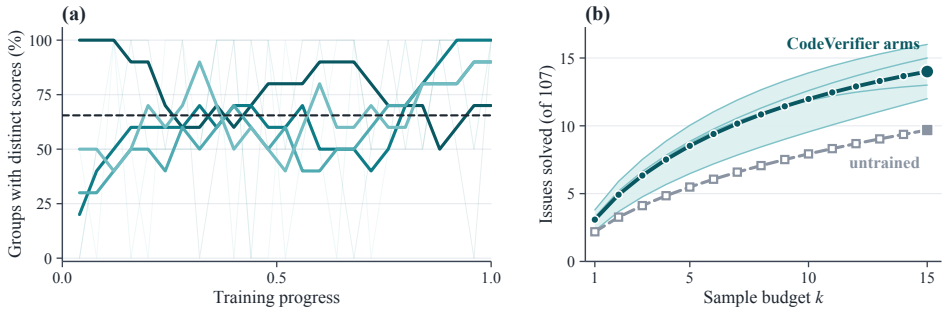

Figure 11: Repository-repair reward resolution and policy gains. (a) Fraction of same-task groups receiving nonconstant CODEVERIFIER scores across four independent runs; faint curves show recorded steps, solid curves show five-step moving means, and the horizontal dashed line shows the aggregate mean. (b) Expected solved tasks out of 107 from the pass@k estimator; thin curves are the four trained runs, the dark curve is their mean, and shading shows their run-wise min–max range.

is unclipped and $A^{\mathrm{tok}}_{i,j,s} \neq 0$*. If*

$$\rho_{i,j}|A^{\mathrm{tok}}_{i,j,s}| > \beta|1 - e^{d_{i,j,s}}|,$$

then the negative partial derivative of the batch objective with respect to that log-probability coordinate satisfies

$$\operatorname{sign}\left(-\frac{\partial \mathcal{L}_{\mathrm{RVPG}}}{\partial h_{i,j,s}}\right) = \operatorname{sign}(A^{\mathrm{tok}}_{i,j,s}) = \begin{cases} \operatorname{sign}(s^{\mathrm{v}}_{i,j} - \bar{s}^{\mathrm{v}}_{i}), & s \in \mathcal{T}^{\mathrm{v}}_{i,j}, \\ \operatorname{sign}(s^{\mathrm{B}}_{i,j,a} - \mu^{\mathrm{B}}_{i}), & s \in \mathcal{T}^{\mathrm{B}}_{i,j,a}. \end{cases} \tag{28}$$

Proof of Corollary 3. An unclipped valid position satisfies $w_{i,j,s} = c^{\mathrm{clip}}_{i,j,s} = 1$. After negating Equation (27), the strict inequality in the corollary ensures that the magnitude of the task term exceeds that of the reference term. The sign of the negative partial derivative is therefore determined by $A^{\mathrm{tok}}_{i,j,s}$. For $(i,j) \in \mathfrak{B}$,

$$\frac{\partial \mathcal{L}_{\mathrm{RVPG}}}{\partial h_{i,j,s}} = \frac{1}{|\mathfrak{B}|}\frac{\partial \mathcal{L}_{i,j}}{\partial h_{i,j,s}}.$$

Because $|\mathfrak{B}|^{-1} > 0$, batch averaging does not change this sign. Equation (9) and $\alpha_v, \alpha_B > 0$ then yield the two branches of Equation (28) for verdict and region positions. □

Equation (27) assigns the global advantage to verdict coordinates and each local advantage to its region coordinates; the remaining positions are regularized by KL. The shared model parameters accumulate these contributions through the chain rule:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{i,j} = \sum_{s} \frac{\partial \mathcal{L}_{i,j}}{\partial h_{i,j,s}} \nabla_{\boldsymbol{\theta}} h_{i,j,s}.$$

In Table 1, full RVPG and Shared, routed update the same verdict and region fields, but differ in whether those fields receive field-specific or shared advantages. Shared adv. additionally assigns the shared advantage across the complete response.

G ADDITIONAL DISCUSSION OF RELATED WORK

Our approach derives supervision from dynamic execution and successful repairs, then uses the trained verifier for code ranking, policy rewards, and repository-level agents.

G.1 DYNAMIC VERIFICATION, TEST ADEQUACY, AND PROGRAM SYNTHESIS

Dynamic software verification has long relied on finite execution to approximate program semantics. Mutation testing measures whether a test suite exposes errors through controlled program changes, while work on test oracles distinguishes input coverage from correctness determination (DeMillo

et al., 1978; Barr et al., 2015). DART and KLEE expand execution coverage with path constraints; Randoop and EvoSuite generate tests through feedback and search, respectively; and fuzzing extends the same idea to large-scale defect discovery (Godefroid et al., 2005; Cadar et al., 2008; Pacheco et al., 2007; Fraser & Arcuri, 2011; Manès et al., 2021). Finding such counterexamples requires path exploration, environment construction, or repeated execution.

Program synthesis casts code generation as search for a program satisfying a specification, while research on software naturalness provides an empirical basis for learning program properties from code distributions (Gulwani et al., 2017; Hindle et al., 2012). When finite tests approximate the specification, passing them does not necessarily imply program correctness. Empirical studies of online-judge tests and patch audits in automated repair have both observed this gap (Liu et al., 2023c; Qi et al., 2015). CODEVERIFIER learns verdicts from execution outcomes and regions from subsequent successful repairs.

G.2 CODE-GENERATION BENCHMARKS AND TEST GENERATION

Early evaluation of code language models focused on generating complete programs from problem statements. HumanEval, APPS, and AlphaCode cover function completion, programming problems, and competitive candidate search, while studies of program synthesis with language models systematically examine how prompting and sampling affect problem-solving ability (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Austin et al., 2021). Later benchmarks broaden both program types and evaluation targets. DS-1000 covers data-science code; TACO and LiveCodeBench cover a wider range of algorithmic problems; BigCodeBench emphasizes complex function calls; and CRUXEval evaluates execution reasoning (Lai et al., 2023; Li et al., 2023; Jain et al., 2025c; Zhuo et al., 2025; Gu et al., 2024). SciCode extends evaluation to scientific programming, while UOJ-Bench places generation, attack, and repair of competitive code in one task framework (Tian et al., 2024; Xu et al., 2026).

The credibility of a benchmark score depends on whether its tests cover the task requirements. Re-executing generated code against stronger tests shows that original pass rates can overestimate correctness (Liu et al., 2023b). SciCode-Verified identifies a complementary problem: defective specifications and grading can reject correct scientific code and underestimate model ability (Hu et al., 2026). TestEval and TestGenEval approach the issue from the other side by directly measuring a model's ability to generate realistic tests (Wang et al., 2025a; Jain et al., 2025b). Recent analyses of code verification further show that generation, testing, and verification capabilities do not improve in lockstep (Ma et al., 2025b; Zhang et al., 2025c). CODEVERIFIER places verdicts, repair-region alignment, and ranking after limited testing in one model.

G.3 OUTCOME VERIFIERS, PROCESS REWARDS, AND MODEL JUDGES

Learned verifiers first demonstrated the complementarity of sampling and selection in mathematical reasoning. Outcome verifiers learn whether a final answer is correct, while process supervision assigns feedback to intermediate reasoning steps (Cobbe et al., 2021; Uesato et al., 2022; Lightman et al., 2024). Generative verifiers cast judgment as autoregressive generation, allowing a model to produce both a conclusion and its basis (Zhang et al., 2025a).

Learned code verification has expanded from final candidates to execution behavior and the generation process. LEVER learns candidate correctness from execution outcomes, Fault-Aware CodeRanker predicts failure risk, and CodeJudge directly judges program semantics (Ni et al., 2023; Inala et al., 2022; Tong & Zhang, 2024). SWE-IF further aligns code evaluation with human preferences, while the CompassVerifier and CompassJudger series support multiple judgment formats and verifiable rewards (Zhong et al., 2026; Liu et al., 2025a; Cao et al., 2024; Zhang et al., 2025b). CodePRM and process-supervised reinforcement learning distribute execution feedback over generation, whereas dynamic test scheduling and CURE jointly consider testing resources and test models (Li et al., 2025; Ye et al., 2025; Ma et al., 2025a; Wang et al., 2025c).

Repository tasks extend code rewards from a single answer to a complete trajectory. SWE-RM learns repository rewards without online execution, process rewards detect trajectory deviations and guide later behavior, and CodeRM-NT and CodeScaler use learned code scores for training or inference-time selection (SHUM et al., 2026; Gandhi et al., 2025; Xia et al., 2026; Zhu et al., 2026).

ExecVerify explicitly constructs executable step-wise rewards, while Aletheia studies training when the verifier and policy change together. AgentV-RL and Agentic Verifier further allow a verifier to search actively for evidence (Tang et al., 2026; Venkatkrishna et al., 2026; Zhang et al., 2026; Ma et al., 2026). Verification Horizon systematically analyzes how the rewards available to a policy change across stages (Wang et al., 2026).

A parallel line of work studies model judges. MT-Bench uses language models for open-ended comparison, while RewardBench and RM-Bench use difficult preference pairs to test whether reward models can ignore superficial style (Zheng et al., 2023; Lambert et al., 2025; Liu et al., 2025b). Code-preference data can be constructed through synthetic evolution or model judging, and CodeUltraFeedback extends this supervision to code alignment (Liu et al., 2024a; Weyssow et al., 2026). LLM-as-a-Verifier further studies general-purpose verifiers for selection, progress estimation, and policy optimization (Kwok et al., 2026). CODEVERIFIER determines success from execution outcomes and constrains relevant regions with subsequent successful repairs.

G.4 SEARCH AND REINFORCEMENT LEARNING FROM EXECUTION FEEDBACK

Code tasks can turn execution outcomes directly into policy rewards. CodeRL and RLTF establish methods for training code models with test feedback, and RLEF applies execution feedback to stronger code policies (Le et al., 2022; Liu et al., 2023a; Gehring et al., 2025). At inference, CodeT reranks candidates with generated tests, while self-debugging, Reflexion, and LDB revise programs using language feedback or runtime states (Chen et al., 2023; 2024; Shinn et al., 2023; Zhong et al., 2024). These methods condition search on code state.

Automated test synthesis and search provide additional routes to rewards. AceCoder supports reinforcement learning with generated tests, Thinking Before Running and Reason-Code integrate execution feedback into candidate exploration, and step-level rewards can train multi-turn code generation (Zeng et al., 2025; Zhang et al., 2025d; Li & Peng, 2026; Jain et al., 2025a). rStar-Coder, CodeRL+, and ReflexiCoder improve code policies through large-scale verification data, execution-semantic alignment, and self-reflection, respectively (Liu et al., 2025c; Jiang et al., 2026b;a). Reasoning Through Execution combines process and outcome rewards within one training procedure (Yu et al., 2025b).

RLVR places this direction within a broader framework for verifiable reasoning. PPO provides a basis for stable policy updates, GRPO in DeepSeekMath reduces the cost of value estimation for group-wise policy optimization, and DeepSeek-R1 demonstrates how verifiable rewards shape reasoning behavior (Schulman et al., 2017; Shao et al., 2024; Guo et al., 2025). Later analysis shows that the outcome of R1-Zero-style training depends jointly on the initial model, reward, and optimization process (Liu et al., 2025d). The online stage of CODEVERIFIER optimizes verdict and region generation, after which our policy experiments use the trained verdict as a reward.

G.5 REPOSITORY-LEVEL CODING AGENTS AND LONG-HORIZON TRAJECTORIES

Early repository-level code research focused on using cross-file context to complete code. RepoCoder generates completions through iterative retrieval, and RepoBench systematically measures the use of repository context (Zhang et al., 2023; Liu et al., 2024c). SWE-bench later defined software repair using real issues and repository tests. SWE-agent, AutoCodeRover, and OpenHands advance automated software development through agent interfaces, repository search, and a general tool platform, respectively (Jimenez et al., 2024; Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025b). SWE-Lancer and SWE-Bench Pro extend evaluation to real assignments and long-horizon tasks, while Multi-SWE-bench covers several programming languages (Miserendino et al., 2025; Deng et al., 2025; Zan et al., 2025).

Large-scale training requires interactive environments and a continually refreshed source of tasks. SWE-Gym and R2E-Gym provide training environments for repository tasks, while SWE-smith and automated repository-task generation expand executable training instances (Pan et al., 2025; Jain et al., 2025d; Yang et al., 2025; Vergopoulos et al., 2025). SWE-rebench establishes an automated collection and decontamination pipeline, with a later version extending it to multilingual tasks. SWE-Universe and SWE-Next continue to increase the scale and update rate of realistic verifiable environments (Badertdinov et al., 2025; 2026; Chen et al., 2026; Liang et al., 2026). CoderForge-

Preview and Open-SWE-Traces complement this ecosystem with open training data and distillation from long trajectories (Ariyak et al., 2026; Ahmad et al., 2026).

As tasks grow longer, training signals must cover more intermediate states. SWE-RL applies reinforcement learning directly to software-evolution tasks, Agent-RLVR combines environment outcomes with process guidance, and long-context multi-turn training handles dependencies across interactions (Wei et al., 2025; Da et al., 2025; Golubev et al., 2025). ReVeal lets coding agents update their behavior through self-verification, while OctoBench separately examines instruction following under repository scaffolding (Jin et al., 2026; Ding et al., 2026). Terminal-Bench extends tool use to general command-line tasks (Merrill et al., 2026). Each of these settings requires the system to assess progress before a trajectory is complete.

Tests in repository benchmarks also require continued auditing. UTBoost strengthens the unit tests in SWE-bench, Otter generates patch tests from issue descriptions, and SWE-ABS uses adversarial augmentation to detect when test-based benchmarks overestimate success (Yu et al., 2025a; Ahmed et al., 2025; Yu et al., 2026). These efforts improve the reliability of final evaluation, while CODEVERIFIER provides learned feedback that can be queried repeatedly for candidates and intermediate states.

G.6 REAL DEFECTS, CODE RELIABILITY, AND SCORE CALIBRATION

Real-defect datasets provide error distributions distinct from synthetic perturbations. Defects4J, ManyBugs, BugsInPy, and TrickyBugs collect defects in Java, C, Python, and boundary cases that may remain hidden after conventional testing, respectively (Just et al., 2014; Le Goues et al., 2015; Widyasari et al., 2020; Liu et al., 2024b). Empirical studies of code generated by language models also compare defect types, maintainability, and reliability, showing that naturally generated errors differ from random syntactic corruption (Tambon et al., 2025; Molison et al., 2025). We supervise regions with code changes around real repairs, so region labels come from locations actually modified by a successful repair.

Calibration maps model scores to empirical correctness, while abstention lets a system defer low-confidence judgments (Guo et al., 2017; Feng et al., 2024). CODEVERIFIER uses label margins for continuous ranking and normalized PASS-label scores for repository-policy rewards. Classification reads the discrete verdict, while repeated candidate comparisons use generative utility.

H PREDICTION FROM PARTIAL INTERACTION RECORDS

This study asks a separate question from current-state verification: whether the visible prefix of a recorded repository interaction predicts the final outcome of its continuation. The CoderForge collection contains 477 trajectories from 12 repositories (Ariyak et al., 2026). We truncate each trajectory at 30%, 60%, and 90% of its length, retain the task description, edits, commands, and tool feedback visible at that point, and use the final recorded outcome as the label.

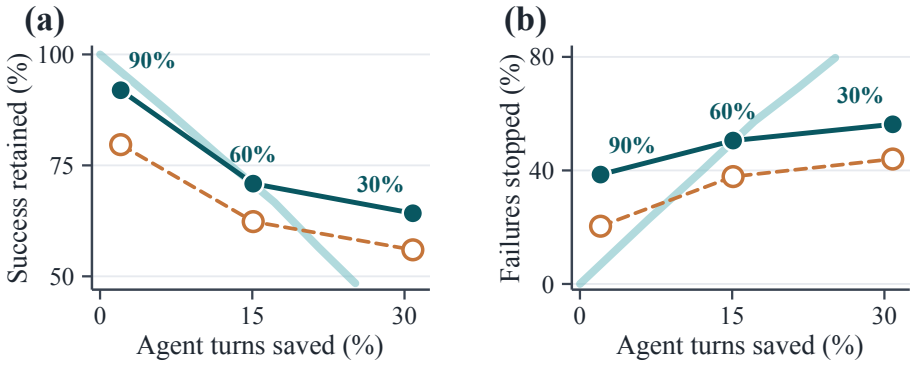

Figure 12: Offline continuation analysis. (a) Retained successful trajectories; (b) stopped failed trajectories. Filled points use 9B CODEVERIFIER and open points use random stopping at the same rate; light curves vary the voting threshold at 60% depth.

AUPVC is the normalized trapezoidal area of ROC AUC over the three prefix depths, while EBA is the same summary of the expected balanced accuracy defined above. The 9B CODEVERIFIER obtains AUPVC .6307 versus .5630 for the size-matched base and EBA .6270 versus .5655. At 60% depth, CODEVERIFIER reaches AUC .6277, compared with .5439 for the base model at 90%, indicating stronger prediction from an earlier partial record. At 90%, CODEVERIFIER rejects .2892 of failed trajectories and retains .9583 of successful trajectories, compared with .0843 and .9792 for the base model.

We also apply the prefix scores in an offline continuation-budget analysis. At matched stopping rates, CODEVERIFIER retains more successful trajectories and stops more failed trajectories than random stopping (Figure 12). This analysis operates on recorded trajectories and does not alter their continuations.