BREAKING THE SOLVER BOTTLENECK: TRAINING TASK GENERATORS AT THE LEARNABLE FRONTIER

Anonymous authors
Paper under double-blind review

ABSTRACT

The limiting resource for training agents via reinforcement learning (RL) is increasingly frontier task supply: valid, solvable tasks just difficult enough to train the current model. As reasoning and agentic models improve, fixed task distributions saturate, while naive synthetic generation yields tasks that are trivial, impossible, or ill-posed. Training a task generator with RL to optimize validity and learnability can address this bottleneck, but direct optimization requires repeated solver rollouts per candidate. For software-engineering (SWE) tasks, a single rollout can take tens of minutes, and solver-in-the-loop generator training needs several such rollouts for every candidate task at every update. We introduce PROPEL, a solver-amortized framework for training task generators at the targeted solve rate. PROPEL trains a lightweight activation probe on a one-time labelled corpus of generated tasks and solver outcomes. The probe predicts target-solver pass rate from a frozen generator reference model and serves as a proxy for solve rate during generator optimization, reducing generator evaluation to a single forward pass. Across math, code, and software-engineering at multiple model scales, PROPEL shifts generation toward the targeted solve rate: for coding, tasks generated at the learnable frontier increase from $10.1\% \rightarrow 20.0\%$ for a `Qwen2.5-3B-Instruct` solver and from $5.3\% \rightarrow 12.6\%$ for a `Qwen2.5-7B-Instruct` solver. For SWE, PROPEL increases targeted solve rate generations from $9.8\% \rightarrow 20.9\%$ for `Qwen3.5-27B` on repositories not seen during training of probe and generator.

1 INTRODUCTION

Reinforcement learning on verifiable rewards (RLVR) has become the dominant recipe for eliciting reasoning and agentic behavior from language models (Guo et al., 2025a; Lambert, 2025; Liu et al., 2025b). Progress under this recipe is gated by the supply of training tasks. As policies improve, fixed task distributions saturate, and further gains require harder tasks that remain discriminative at the current capability frontier. Hand-curated benchmarks cannot keep pace, and naïve synthetic generation tends to produce tasks that are either trivially solvable or ill-posed. A natural alternative is to train a *generator* model with RL, rewarding it for producing tasks that are well-formed and appropriately difficult for a target solver (Zhao et al., 2025; Wei et al., 2025). The implied objective is *discriminability*, tasks on which solver is challenged but does not completely fail (Wei et al., 2025).

However, evaluating a candidate task requires running the solver, and in agentic settings this can be prohibitively expensive. On SWE-bench-style tasks (Jimenez et al., 2024; Yang et al., 2026) a single rollout can take tens of minutes since it involves repository navigation, tool calls, and test execution. A reliable difficulty signal needs many such rollouts per candidate to estimate solve rate. Embedding this loop inside generator RL makes solver trials the bottleneck, requiring multiple trials for each candidate task at every rollout. The same bottleneck appears, more mildly, in competitive math and code generation, where solver trials are cheaper but still costly and variance remains high. Standard RLVR pipelines do not scale to objectives whose verifier is itself an expensive stochastic agent; synthetic generation of tasks scales doubly poorly as solve rate, complexity, and cost-to-solve are unfavorable.

We introduce **PROPEL** – ***P**robe **R**ewards for **O**ptimizing **P**roblems at the **E**dge of **L**earning* – a solver-amortized framework for training task generators. PROPEL builds on Reinforcement Learning from Feature Rewards (RLFR; Prasad et al., 2026), which uses interpretability features from hidden

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

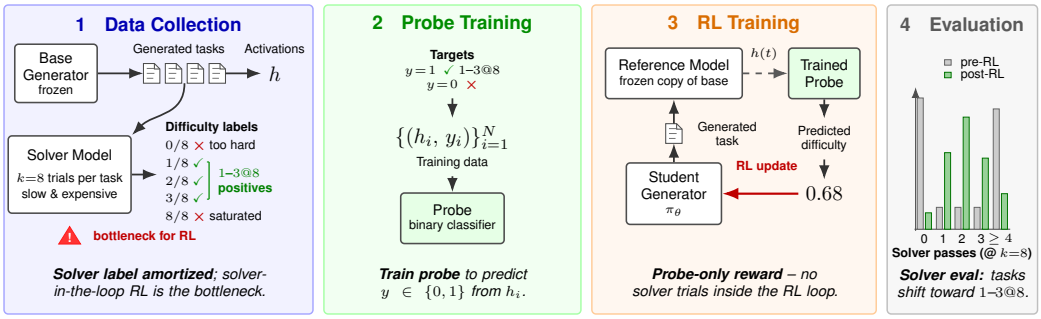


Figure 1: **Pipeline overview:** (1) A base generator produces a one-time pool of tasks that are labelled with an expensive solver. (2) A probe is trained to predict those difficulty labels from the generator's hidden states. (3) During RL, a student generator proposes tasks; a frozen reference model produces activations and the trained probe converts them into reward, so the solver is never invoked inside the inner loop. (4) The trained generator is finally evaluated against a held-out solver to confirm that probe-driven shaping translates into true difficulty gains.

states to supervise open-ended generation, and adapts that recipe to task generation with two changes that the agentic setting demands: a multi-step trajectory formulation for software-engineering tasks, and explicit treatment of fixed-probe mode collapse. A small probe is trained once on a one-time labeled corpus of (task, solver-outcome) pairs read out from a frozen reference generator's activations; during RL it replaces live solver rollouts as the reward, collapsing per-step cost from many solver trials to a single forward pass (see Figure 1). The construction exploits a well-documented property of language models, that quantities of interest are often represented internally even when the model cannot act on them reliably at generation time (Orgad et al., 2024; Zhang et al., 2025a). Provided that a candidate task's well-formedness, solvability, and difficulty-calibration is decodable from generator hidden states, the probe gives a dense, near-free signal that stands in for the true objective long before any solver rollout would confirm it.

PROPEL removes live solver calls from generator RL while still shifting generation toward the target solver's learnable frontier. On code induction it roughly doubles the rate of frontier tasks for both a `Qwen2.5-3B-Instruct` and a `Qwen2.5-7B-Instruct` solver ($10.1\% \rightarrow 20.0\%$ and $5.3\% \rightarrow 12.6\%$; Figure 2), and in SWE it doubles the frontier rate on held-out repositories. A matched validity-only control shows that these gains reflect solver-verified difficulty rather than validity. Figure 1 summarizes the pipeline: a base generator collects tasks with ground-truth labels from solver rollouts, a probe is trained to predict these labels from activations, and the probe then replaces the solver as the reward model during generator RL. Our contributions are as follows.

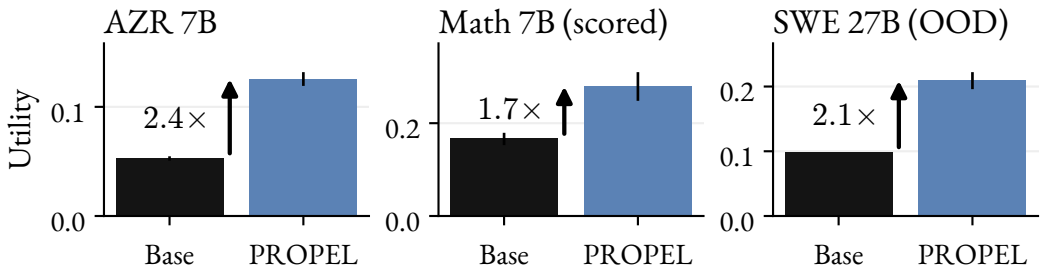


Figure 2: PROPEL increases the utility of generated tasks over the base generator. Utility is measured based on the `Qwen2.5-7B-Instruct` solver for AZR and math, and the `Qwen3.5-27B` solver on held-out OOD repositories for SWE. On math utility is reported on the post-oracle scored tasks. Error bars are ± 1 standard error across RL seeds.

- **Effective generator RL without online solver calls.** We adapt RLFR probe rewards to task generation, including multi-turn SWE trajectories, replacing live solver feedback with a frozen-reference forward pass. On AZR, PROPEL achieves higher utility than solver-in-the-loop

RL (20.0% versus 14.0%) using $22{,}592$ offline solver trials rather than $53{,}664$ online trials. The offline corpus is reused across runs for the same target solver.

- **Frontier targeting generalizes to unseen repositories.** PROPEL roughly doubles frontier-task rates on code induction at two solver sizes and on held-out SWE repositories. The SWE gains transfer to repositories unseen during both probe training and generator RL. In math, utility increases by about $1.7\times$ at both solver sizes among oracle-verified tasks; this gain is conditional on verification.
- **Probe rewards steer difficulty while preserving structural diversity.** A matched AZR control shows that the utility gain extends beyond improved validity, and targeting a centered solve-rate band demonstrates control over task difficulty. PROPEL's programs remain structurally distinct and become more complex, although representation-space coverage narrows. An ensemble reward partly mitigates topic concentration.
- **Fixed probes transfer across generator families without retraining.** Holding the probe, reference model, and solver fixed, PROPEL nearly doubles AZR utility for a `Mistral-7B-Instruct-v0.3` generator without per-family tuning. Utility also improves for `Phi-3.5-mini-instruct`, although transfer is weaker and varies across domains.

2 Related Work

Synthetic task generation and self-play. A growing body of work trains task generators that target the solver's learnable frontier, with two prior systems most directly informing our design. Absolute Zero (Zhao et al., 2025) self-plays a proposer/solver pair across deduction, abduction, and induction tasks; we adopt its induction format and its core observation that the proposer needs a difficulty signal that is neither trivial nor unsolvable. Self-play SWE-RL (Wei et al., 2025) ports this to bug injection vs. repair, with an injection reward that peaks for solver pass rates near the middle of the 0–1 range, directly motivating our fixed-student pass-at-K utility. Beyond these, task-generation work spans math and symbolic reasoning (Liang et al., 2025a; Li et al., 2025; Liu et al., 2025a; Lacombe et al., 2025) and software engineering (Sonwane et al., 2025; Pan et al., 2024; Jain et al., 2025; Xie et al., 2026; Zhang et al., 2025c; Wang et al., 2025; Zhu et al., 2025), with adjacent work on synthetic-data quality (Chen & Zhong, 2025), solver-side training (Da et al., 2025), and abstraction generation (Qu et al., 2025). Of the methods that train a task generator with RL, the reward is generally computed by running the target solver on each candidate; we replace those rollouts with a single forward pass through a probe.

Probes and internal-state rewards. Probes have been used to predict reasoning correctness (Zhang et al., 2025a; David, 2025; Cencerrado et al., 2025), score best-of-N candidates (Guo et al., 2025b), and calibrate judges (Radharapu et al., 2025). More recently, a series of works have explored using model internals as supervision during training (Zhang et al., 2026; Liang et al., 2025b; Prasad et al., 2026). Most relevant to ours, Prasad et al. (2026) introduce *RL from Feature Rewards* (RLFR), a framework using probes over model internals as scalable reward functions for open-ended tasks. While RLFR isolates features associated with hallucinations, we train a probe to predict a task's training utility (as defined in section 3.2) from the internal features of a task generator. For SWE task generation, this requires us to extend RLFR to multi-turn trajectories.

Reward overoptimisation and mode collapse. Optimizing any learned reward proxy is subject to Goodhart effects (Gao et al., 2023; Kwa et al., 2024; Moskovitz et al., 2023) and mode collapse, with KL regularization itself capable of driving collapse rather than preventing it (GX-Chen et al., 2025). Iterated RLHF retrains the proxy on the policy's collapsed outputs (Wolf et al., 2025), and verbalized sampling preserves diversity at the decoding level (Zhang et al., 2025b). Our adversarial probe-co-evolution follows the iterated-feedback recipe. Probe-high but solver-failed outputs become negatives for the next probe.

3 Probe Rewards for Optimizing Tasks at the Edge of Learning

We study reinforcement learning for *task-generator* language models across three domains of increasing complexity, math competition tasks, code-induction puzzles (Absolute Zero Reasoner - AZR),

and software-engineering tasks (SWE). We first introduce the formal setting then discuss the shared framework PROPEL and provide the per-domain specifics in Section 4.2.

3.1 Problem Setting

A generator policy π_θ synthesizes tasks $x \sim \pi_\theta(\cdot \mid c)$ given a context c that contains a short instruction, examples, or represents access to a Docker image with a passing test suite. A good task should meet the following two criteria: It should be a valid task, i.e., a math task should have a clear statement, a code-induction task should run, and a bug should apply cleanly and make a previously passing test fail. Second, it should be useful for training: a target solver should sometimes solve it and sometimes fail.

More formally, the low-cost validity check is a predicate $\mathcal{W} : \mathcal{X} \to \{0, 1\}$, where $\mathcal{W}(x) = 1$ means that x is syntactically valid and can be graded or executed. The expensive signal is a utility label $U : \mathcal{X} \to \mathbb{R}$, computed from fixed-solver solve counts, or in the case of SWE solver, rollouts graded against tests. Our objective is to train π_θ to increase $\mathbb{E}_{x\sim\pi_\theta}[U(x)]$ while avoiding expensive calls to U inside the RL loop.

3.2 Utility Signal for Task Generators

The main utility label asks whether a task lies at the target solver's learnable-frontier. For a fixed solver S, and K attempts, let $\mu_S(x)$ be the mean solve rate for group size of K. The utility is defined as:

$$U_S(x) = \mathbb{I}[a \le \mu_S(x) \le b]. \tag{1}$$

Following the observation by Wei et al. (2025), the optimal solve rate range to target with a group size $K = 8$ is $a = 1/8$ and $b = 3/8$. A task is positive if the solver solves it 1, 2, or 3 times out of 8. A task solved 0 times is too hard for that solver, and a task solved 4 to 8 times is already too easy. This gives the generator credit for tasks that are learnable but not saturated. For SWE we reduce the number of solver trials to $K = 3$ to mitigate the computational burden, and use the utility signal $U_S(x) = \mathbb{I}[1/3 \le \mu_S(x) \le 2/3]$. This slightly broader solve rate band is chosen to account for the small number of tasks generated by the base model that yield exactly one successful attempt and increases the pool of positive samples slightly.

3.3 Probe-reward RL

Evaluating U during RL requires K solver trials per task, which is slow, requires additional memory for solver inference during RL, and each attempt must be verified by a grader, which can be costly, especially in the case of SWE tasks where the grader is derived from a repository's test suite. Instead, we approximate U with an **activation probe** f_ϕ. The probe is a small classifier trained on a dataset of generated tasks whose expensive labels have been computed offline. During RL, a newly generated task is given to a frozen copy of the base generator, denoted π_{ref}. We read hidden states from this frozen reference model, not from the policy π_θ being updated. For a generated task x, the reference model processes the rendered task text or trajectory, and we extract

$$h_L(x) = \text{Pool}(\text{Hidden}_L(x;\, \pi_{\text{ref}})), \tag{2}$$

where $\text{Hidden}_L(x;\, \pi_{\text{ref}})$ are the layer-L hidden states of π_{ref} on input x and $\text{Pool}(\cdot)$ is a pooling rule. The probe maps this vector to either a positive-class probability $p_\phi(x)$ or a raw logit score $\ell_\phi(x)$. The probability is useful for calibrated screening; the logit keeps more dynamic range for RL rewards. Freezing the reference model makes the reward a function of the generated task rather than a function of the changing hidden states of the RL policy.

RL reward signal. The main reward signal uses validity as a gate and the probe as the ranking signal. Invalid generations receive a fixed penalty $r_{\text{bad}} < 0$; valid generations receive the raw probe logit:

$$R_{\text{hard}}(x) = \begin{cases} r_{\text{bad}}, & \mathcal{W}(x) = 0, \\ \ell_\phi(x), & \mathcal{W}(x) = 1. \end{cases} \tag{3}$$

The validity predicate decides whether a task can be used at all, while the probe ranks valid tasks by estimated frontier utility. Ablations on the reward signal change only this combination rule.

Probe-only drops the validity gate and rewards $p_\phi(x)$. *Soft-gated* rewards valid tasks with a clipped probability $\mathrm{clip}(p_{\phi(x)}, 0.1, 0.95)$ instead of a raw logit, while still penalizing invalid tasks.

Additionally, we investigate an ensemble of probes optimized via worst-case optimization (WCO) to mitigate topic collapse and overfitting to specific features of an individual probe Coste et al. (2024):

$$R_{\mathrm{WCO}}(x) = \begin{cases} r_{\mathrm{bad}}, & \mathcal{W}(x) = 0, \\ \min_j \ell_{\phi_j}(x), & \mathcal{W}(x) = 1. \end{cases} \tag{4}$$

Probe evaluation. The trained probes are evaluated based on standard classification metrics (Accuracy, Balanced Accuracy, F1-score) and calibration metrics (ECE). The probe-training datasets, the sweep grid, the selection procedure, and the selected probes are reported in Appendix K.

Generator RL. The generator policy is trained with GRPO for math and code (Shao et al., 2024), while SWE uses DAPO style advantage calculations with soft long context penalties (Yu et al., 2025). All methods using the probe-based reward R_ϕ for generator optimization. For each context c, the policy samples a group of G candidate tasks $\{x_i\}_{i=1}^G \sim \pi_\theta(\cdot \mid c)$. Each candidate is passed through the frozen reference π_{ref} to extract $h_L(x_i)$, from which the probe produces the scalar reward $R_\phi(x_i)$. Implementation details for RL training of the generator are specified in Appendix L. Because R_ϕ depends only on the rendered task and the frozen π_{ref}, the policy cannot improve its reward by shifting the activations the probe reads (Prasad et al., 2026). The full pipeline (Figure 1) is: collect an offline task pool, compute U for that pool, extract frozen-reference activations, train and select probes, and run RL with R_ϕ. Held-out evaluation always recomputes the true utility U on fresh generations; we do not report the probe score as the final outcome.

4 Evaluation

4.1 Metrics

The core downstream metric is the realized utility of fresh generations: we sample tasks from the trained policy, then run the solver K times per task and report the share whose realized solve count lands in the target band defined by U_S. The probe score is never used for evaluation; every result in the paper recomputes U from solver rollouts. We additionally report three text-level diversity diagnostics over all generations per condition: ***Self-BLEU-3*** (Montahaei et al., 2019), ***Distinct-3*** (unique 3-gram ratio; higher = more diverse), and ***top-topic rate***, the share of generations in the most common topic. Topics are assigned by a fixed, ordered keyword taxonomy with 10 math topics and 12 AZR algorithm families (Appendix B). For SWE, the topic is the first file edited by the bug patch, and we average the share of the most common file within each repository across repositories. For AZR we also measure code-aware diversity over valid tasks: the fraction of structurally unique programs (abstract syntax trees with identifiers and literals removed), mean cyclomatic complexity, and PRDC recall (Kynkäänniemi et al., 2019; Naeem et al., 2020), the fraction of Base tasks within the 5-nearest-neighbor neighborhoods of the generator's tasks in the frozen reference model's activation space.

4.2 Domains

Math and AZR. For both math and code, the generator is `Qwen3.5-4B` (Qwen Team, 2026) and we train probes against labels $U_S(x) = \mathbb{I}[1/8 \le \mu_S(x) \le 3/8]$ for two levels of solvers `Qwen2.5-3B-Instruct` (Qwen et al., 2025) and the `Qwen2.5-7B-Instruct` model. The two domains differ only in the task format and how the ground truth solution is established.

In math, the generator is prompted with two in-context examples from GSM8K (Cobbe et al., 2021), MATH-500 (Hendrycks et al., 2021), or AIME-style (Dekoninck et al., 2026) pools and writes a competition-style task; a solver attempt is correct when its extracted answer is symbolically equivalent to a verified reference. References use a strict two-oracle policy: `Qwen2.5-32B-Instruct` (Qwen et al., 2025) and `Phi-4` (Abdin et al., 2024b) each make two attempts, and a task is scored only when all four answers agree. This strict reference filter is applied only for evaluation and is not part of the reward or validity gate used during math generator RL. Since it can reject hard but well-posed tasks, we evaluate math both before and after the filter.

In AZR we adopt the function-induction format of Zhao et al. (2025): given a function seed, the generator produces a triple $x = (f, \{u_1, \ldots, u_n\}, m)$ of a Python function, input tuples, and a natural-language hint; the solver sees five input-output examples plus the hint and must reproduce f's behavior on the remaining 5 hidden inputs. Since the generator commits to executable ground truth, no oracle filter is needed. Validity gates, banned imports, sandboxing details, and the full generator prompts are in Appendix N.

Both domains are trained with GRPO and LoRA adapters; KL coefficients are $\beta = 0.05$ (AZR) and $\beta = 0.1$ (Math). Full hyperparameters in Appendix L.

SWE. The SWE setting is a bug-writing task on real Python repositories from the SWE-smith registry (Yang et al., 2026), partitioned into repo-disjoint probe-training, RL-training, and evaluation splits (Appendix J). The generator, `Qwen3.5-27B` (Qwen Team, 2026), acts as a multi-turn bash agent with a custom harness (see Appendix J.3) inside a sandboxed repository and must introduce a behavior-changing bug while leaving tests unmodified. The verifier checks that the patch applies, tests are unchanged, the suite still runs, and at least one previously passing test fails. For every valid bug we run $K = 3$ solver trials with the same model under the OpenCode harness (Anomaly, 2026), each in a fresh sandbox with the bug introducing patch produced by the generator applied; a trial is solved iff the solver's final edit passes the verifier suite.

Unlike math and AZR, a SWE task is a full agent trajectory: we extract activations per generated agent turn and aggregate the turn sequence before probing. The probe sweep varies reference-model layer, turn aggregation, probe architecture, and class-imbalance handling, and selects a fixed probe before RL (see Appendix K.5).

RL uses the SkyRL GRPO (Cao et al., 2025) stack with vLLM rollout engines (Kwon et al., 2023), and per-repository Docker sandboxes; training rollouts are scored by the repository verifier and the selected activation probe, while the expensive $K = 3$ OpenCode (Anomaly, 2026) solver labels are reserved for offline probe training and held-out evaluation. The challenger is trained for 30 steps with up to 35 tool-using turns per trajectory. Invalid generations receive a fixed reward of -0.2. Full hyperparameters are in Appendix L.

5 Results

PROPEL shifts generators toward the learnable frontier. PROPEL increases generator utility in AZR and SWE and improves conditional utility among oracle-verified math tasks (Figure 2 and Table 1). On code induction, it roughly doubles utility over all generations at both solver sizes, from 10.1% to 20.0% for the 3B solver and from 5.3% to 12.6% for the 7B solver, shifting tasks out of the saturated end of the solve-count distribution (Appendix G). The gains extend to multi-turn SWE with the `Qwen3.5-27B` generator. On 11 held-out repositories unseen during both probe training and RL, utility rises from 9.8% to $20.9\% \pm 1.3\%$ across two seeds, demonstrating generalization beyond both training distributions.

On math, PROPEL increases utility by about $1.7\times$ at both solver sizes among tasks verified by the strict two-oracle filter. This gain reflects a shift in the difficulty of verifiable tasks. The strict oracle filter is evaluation-only and is not part of the reward or validity gate during math generator RL. For the 7B solver, oracle survival falls from 19.9% to 12.3%, leaving target-task yield per generation approximately unchanged (3.3% versus 3.4%). The filter rejects some hard, well-posed tasks as well as defective ones, and alternative adjudication rules change the unconditional comparison (Appendix A).

PROPEL exceeds solver-in-the-loop utility with fewer solver trials. In a 30-step AZR comparison, PROPEL achieves more than twice the utility improvement of solver-in-the-loop (SIL) RL while using fewer solver trials (Table 2). SIL uses the same generator and optimizer but computes the utility reward from K online solver attempts per eligible task at every update. It consumes $53{,}664$ online trials for a 4.4-point gain over Base, compared with PROPEL's 9.9-point gain from $22{,}592$ offline trials and no solver calls during RL. The offline corpus is reused across subsequent runs for the same target solver, whereas SIL incurs solver costs anew for each run. Avoiding online feedback is especially valuable in SWE, where each solver trial is a multi-turn agent episode with a budget of up to 30 minutes.

Table 1: Generator performance across the three domains. PROPEL increases utility over Base, with reduced text diversity; WCO partly moderates this trade-off on AZR. Math utility is conditional on oracle verification. Bold marks the best mean within each domain and solver; underlining marks other means whose $\pm$ standard error intervals overlap its interval.

Method	Solver	Utility (%) ↑	Valid (%) ↑	Self-BLEU ↓	Distinct-3 ↑	Top-topic ↓
AZR						
Base	3B	10.09 ± 0.46	66.54 ± 0.42	**0.711** ± 0.003	**0.376** ± 0.003	**0.318** ± 0.013
PROPEL	3B	**19.95** ± 0.62	**90.85** ± 1.89	0.809 ± 0.031	0.266 ± 0.036	0.735 ± 0.093
PROPEL WCO	3B	17.25 ± 0.38	83.46 ± 1.70	0.714 ± 0.020	0.360 ± 0.022	0.685 ± 0.123
Base	7B	5.27 ± 0.20	67.48 ± 0.49	0.720 ± 0.002	0.369 ± 0.003	**0.293** ± 0.004
PROPEL	7B	**12.55** ± 0.63	**88.28** ± 2.15	0.734 ± 0.021	0.346 ± 0.025	0.674 ± 0.016
PROPEL WCO	7B	9.81 ± 0.34	84.57 ± 2.54	**0.700** ± 0.018	**0.377** ± 0.021	0.535 ± 0.057
Math						
Base	3B	24.74 ± 1.37	**96.63** ± 0.54	**0.631** ± 0.000	**0.505** ± 0.004	**0.605** ± 0.008
PROPEL	3B	**41.93** ± 2.21	96.04 ± 1.03	0.708 ± 0.005	0.387 ± 0.004	0.751 ± 0.015
Base	7B	16.60 ± 1.33	**96.63** ± 0.54	**0.631** ± 0.000	**0.505** ± 0.004	0.605 ± 0.008
PROPEL	7B	**27.91** ± 3.09	95.65 ± 0.83	0.652 ± 0.014	0.464 ± 0.016	**0.516** ± 0.044
SWE						
Base (OOD)	27B	9.84	13.27	**0.811**	0.085	**0.605**
PROPEL (OOD)	27B	**20.91** ± 1.31	**35.18** ± 0.64	0.862 ± 0.036	**0.107** ± 0.015	0.650 ± 0.068

Table 2: Solver cost and utility on AZR with the 3B solver. PROPEL exceeds SIL utility without online solver calls, reusing one offline corpus. Parentheses show gains over Base in percentage points.

Method	Signal cost / task	Solver trials on-/offline	Utility (%) ↑	Valid (%) ↑
SIL	8 solver trials	53.7k / 0	14.04 (+4.36)	90.66
PROPEL	1 probe eval	0 / 22.6k	19.95 ± 0.62 (+9.86)	90.85 ± 1.89

Probe rewards steer task difficulty beyond validity. A matched AZR comparison shows that the probe reward increases target-band yield beyond what validity-only RL achieves (Table 3). Across three RL seeds per condition, we compare a validity-only control with two PROPEL conditions targeting either the low (1–3/8) or a centered (3–5/8) solve-rate band, using probes re-fit in the headline configuration. The validity-only generator has the highest validity (94.8%) but barely improves either target band. Adding the low-band probe raises 1–3/8 yield by 8.8 percentage points over this control, while the centered probe raises 3–5/8 yield by 4.3 points. The reward therefore steers solver-verified difficulty and can target different solve-rate bands.

Programs remain structurally diverse while representation coverage narrows. PROPEL generates programs that are more often structurally unique and more complex than those of either baseline, with no single program structure exceeding 0.31% of any seed (Table 3). This structural diversity coexists with stronger topic concentration. With the 3B solver, about 74% of AZR tasks fall into the most common topic, and text-overlap metrics worsen (Table 1). Representation-space coverage also narrows, with PRDC recall of 0.36 for low-band PROPEL versus 0.77 for validity-only RL. A worst-case ensemble of two probes (Equation 4) partly mitigates topic concentration, reducing the top-topic share from 0.67 to 0.54 with the 7B solver while retaining most of the utility gain. An exploratory adversarial co-evolution run reaches 16.0% utility versus 10.1% for Base while keeping diversity metrics close to Base, suggesting that mining probe false positives can help recover diversity (Appendix I).

Cold transfer of the probe across generator families. We hold the probe, the `Qwen3.5-4B` reference model, and the target solver fixed while replacing the trainable generator with `Mistral-7B-Instruct-v0.3` (Jiang et al., 2023) or `Phi-3.5-mini-instruct` (Abdin et al., 2024a). All hyperparameters are inherited from the in-family `Qwen3.5-4B` runs, with no per-family tuning (Figure 3 and Appendix D). On AZR with the `Qwen2.5-3B-Instruct` solver, Mistral nearly doubles utility (5.2% → 9.6%) while increasing executable validity (39.6% → 67.4%),

Table 3: Comparison of generator rewards on AZR, including code-specific diversity metrics. Validity-only RL has little effect on task difficulty, whereas PROPEL improves target-band yield over both baselines. Bold marks each PROPEL variant's target-band yield and structural diversity exceeding both baselines.

Generator reward	Valid	1–3/8 band	3–5/8 band	Unique AST	Complexity	PRDC recall
Base (no RL)	68.9 ± 1.5	9.0 ± 1.6	9.1 ± 0.6	0.970	4.96	—
Validity only	94.8 ± 0.6	11.1 ± 0.5	11.4 ± 0.7	0.927	4.30	0.771
PROPEL, 1–3/8 probe	91.9 ± 1.5	$\mathbf{19.9} \pm 3.0$	11.8 ± 1.0	**0.997**	**7.29**	0.361
PROPEL, 3–5/8 probe	86.8 ± 1.5	17.1 ± 1.5	$\mathbf{15.7} \pm 1.6$	**0.998**	**7.54**	0.555

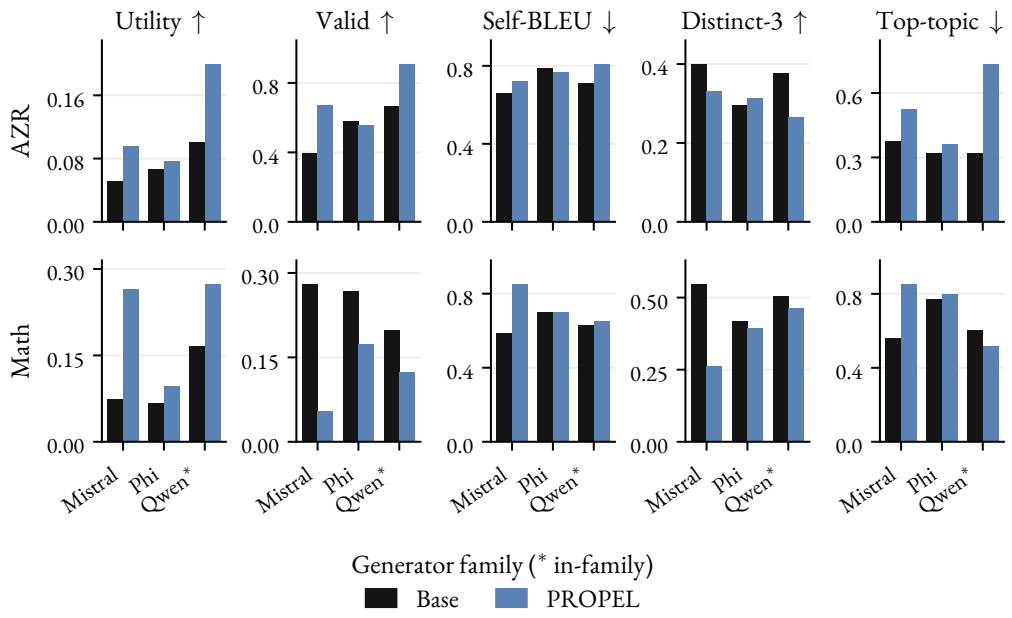
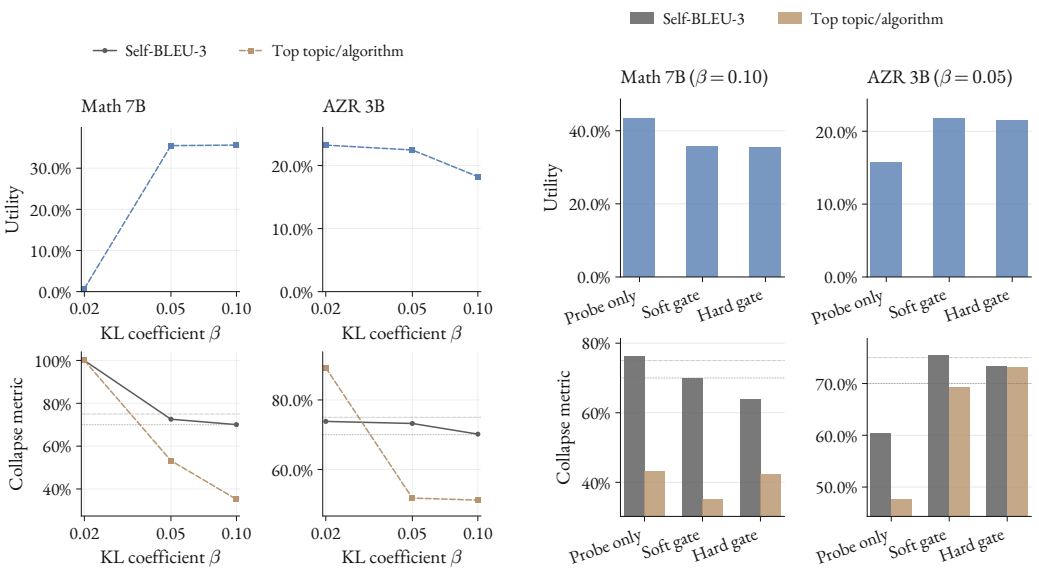


Figure 3: Cold transfer of a fixed probe across generator families ($n = 1{,}024$ generations per condition). The probe, the `Qwen3.5-4B` reference model, and the target solver are fixed; only the trainable generator changes, with hyperparameters inherited from the in-family runs. For AZR, *Valid* is executable validity; for math, it is the strict two-oracle survival rate, and *Utility* is conditional on it. `Qwen*` is the in-family generator.

demonstrating substantial gains from the transferred probe reward. Phi also improves utility, although the gain is more modest. On math, Mistral's conditional utility more than triples ($7.4\% \rightarrow 26.4\%$), but strict-oracle survival collapses ($28.0\% \rightarrow 5.5\%$) and diversity declines, so verified target-task yield per generation does not improve. These results show that the probe can guide a new generator family without probe retraining or additional probe-training data, with the strongest evidence on AZR and transfer quality varying across families and domains.

Ablations. Lower KL coefficients trade utility for diversity (Figure 4a), and the math run at $\beta = 0.02$ collapsed completely, which informed our choice of $\beta = 0.1$ for math and 0.05 for AZR. The best reward composition depends on the domain: probe-only reward gives the highest math utility but the least diverse outputs, whereas both validity-gated variants give higher AZR utility than probe-only reward at the cost of stronger topic concentration (Figure 4). The confirmatory runs use the hard gate, which assigns no probe reward to invalid tasks.



(a) KL ablation.

(b) Reward composition ablation.

Figure 4: Varying the KL regularization strength trades utility against diversity and topic concentration. The reward-composition ablation shows that the best composition depends on the domain and on the utility-diversity trade-off.

6 Conclusion

Summary. As models saturate existing benchmarks, the bottleneck of RL shifts to producing the right tasks at the right time. We have shown that a lightweight probe on a frozen reference model's activations can replace live solver calls as the reward for task-generator RL. Optimizing this reward shifts generation toward the learnable frontier of a target solver: on code induction the frontier rate roughly doubles, a matched control shows that the gain reflects solver-verified difficulty rather than validity, and the same mechanism can target a centered band. The generated programs remain structurally diverse and become more complex, although they occupy a narrower region of representation space. PROPEL replaces solver trials at every RL update with a one-time labeled corpus that all runs reuse, and the probe reward transfers without retuning to a generator from a different model family. Internal-state rewards thus amortize the solver cost that dominates generator RL and offer a route into domains where ground-truth verifiers do not exist.

Limitations. In math, the gain is conditional on a strict reference filter, and the unconditional comparison depends on how answers are adjudicated. PROPEL preserves structural diversity but narrows representation-space coverage, and our mitigations are partial. Finally, PROPEL amortizes and significantly reduces but does not eliminate solver cost, because the labeled corpus that anchors the probe requires solver rollouts to construct.

Future works. We treat task generation as a static target: tasks near the frontier of a *fixed* solver. In practice the solver itself improves, and refreshing the probe against the current solver would turn PROPEL into an inner-outer loop, a tractable path toward open-ended learning systems (Hughes et al., 2024) whose stabilization remains open.

AI use statement

In this work, we used generative AI tools, including coding agents such as Codex and Claude Code, to assist in implementing experiment and analysis code, to retrieve and identify related work, and to edit and grammar-check the paper writing. We have reviewed all AI-assisted work: all code was reviewed and tested by the authors, and retrieved related work was verified against the original sources. We

take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

Reproducibility statement

We provide the complete generator-RL hyperparameters for all three domains in Appendix L, including optimizer, sampling, reward-shaping, and compute configurations. The probe-training datasets, activation-extraction settings, sweep grids, training hyperparameters, and the selection procedure with the selected probes are reported in Appendix K. The SWE repository sourcing, split construction, bug-generation, and solver-labeling protocols are described in Appendices J–J.4, and the full generator prompts for all three domains are listed in Appendix N. All reported utilities are recomputed from fresh solver rollouts as described in Section 4.1.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024a. URL https://arxiv.org/abs/2404.14219.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report, 2024b. URL https://arxiv.org/abs/2412.08905.

Anomaly. Opencode: The open source coding agent. https://github.com/anomalyco/opencode, 2026. GitHub repository, accessed 2026-05-05.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Iván Vicente Moreno Cencerrado, Arnau Padrés Masdemont, Anton Gonzalvez Hawthorne, David Demitri Africa, and Lorenzo Pacchiardi. No answer needed: Predicting llm answer accuracy from question-only linear probes, 2025. URL https://arxiv.org/abs/2509.10625.

Arthur Chen and Victor Zhong. Synque: Estimating synthetic dataset quality without annotations. *arXiv preprint arXiv:2511.03928*, 2025.

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.

Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dcjtMYkpXx.

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.

Joey David. Temporal predictors of outcome in reasoning language models, 2025. URL https://arxiv.org/abs/2511.14773.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvalddson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms, 2026. URL https://arxiv.org/abs/2605.00674.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

Jizhou Guo, Zhaomin Wu, Hanchen Yang, and Philip S. Yu. Mining intrinsic rewards from llm hidden states for efficient best-of-n sampling, 2025b. URL https://arxiv.org/abs/2505.12225.

Anthony GX-Chen, Jatin Prakash, Jeff Guo, Rob Fergus, and Rajesh Ranganath. Kl-regularized reinforcement learning is designed to mode collapse, 2025. URL https://arxiv.org/abs/2510.20817.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. URL https://arxiv.org/abs/2103.03874.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence, 2024. URL https://arxiv.org/abs/2406.04268.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

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.

Thomas Kwa, Drake Thomas, and Adrià Garriga-Alonso. Catastrophic goodhart: regularizing rlhf with kl divergence does not mitigate heavy-tailed reward misspecification, 2024. URL https://arxiv.org/abs/2407.14503.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Tuomas Kynkäänniemi, Tero Karras, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Improved precision and recall metric for assessing generative models. In *Advances in Neural Information Processing Systems*, 2019.

Valentin Lacombe, Valentin Quesnel, and Damien Sileo. Reasoning core: A scalable rl environment for llm symbolic reasoning, 2025. URL https://arxiv.org/abs/2509.18083.

Nathan Lambert. Reinforcement learning from human feedback. *arXiv preprint arXiv:2504.12501*, 2025.

Jiazheng Li, Hongzhou Lin, Hong Lu, Kaiyue Wen, Zaiwen Yang, Jiaxuan Gao, Yi Wu, and Jingzhao Zhang. Questa: Expanding reasoning capacity in llms via question augmentation, 2025. URL https://arxiv.org/abs/2507.13266.

Xiao Liang, Zhong-Zhi Li, Yeyun Gong, Yang Wang, Hengyuan Zhang, Yelong Shen, Ying Nian Wu, and Weizhu Chen. Sws: Self-aware weakness-driven problem synthesis in reinforcement learning for llm reasoning, 2025a. URL https://arxiv.org/abs/2506.08989.

Zhenwen Liang, Ruosen Li, Yujun Zhou, Linfeng Song, Dian Yu, Xinya Du, Haitao Mi, and Dong Yu. Clue: Non-parametric verification from experience via hidden-state clustering, 2025b. URL https://arxiv.org/abs/2510.01591.

Huanyu Liu, Ge Li, Jia Li, Hao Zhu, Kechi Zhang, and Yihong Dong. Saturn: Sat-based reinforcement learning to unleash llms reasoning, 2025a. URL https://arxiv.org/abs/2505.16368.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models. *arXiv preprint arXiv:2505.24864*, 2025b.

Ehsan Montahaei, Danial Alihosseini, and Mahdieh Soleymani Baghshah. Jointly measuring diversity and quality in text generation models, 2019. URL https://arxiv.org/abs/1904.03971.

Ted Moskovitz, Aaditya K. Singh, DJ Strouse, Tuomas Sandholm, Ruslan Salakhutdinov, Anca D. Dragan, and Stephen McAleer. Confronting reward model overoptimization with constrained rlhf, 2023. URL https://arxiv.org/abs/2310.04373.

Muhammad Ferjad Naeem, Seong Joon Oh, Youngjung Uh, Yunjey Choi, and Jaejun Yoo. Reliable fidelity and diversity metrics for generative models. 2020.

Hadas Orgad, Michael Toker, Zorik Gekhman, Roi Reichart, Idan Szpektor, Hadas Kotek, and Yonatan Belinkov. Llms know more than they show: On the intrinsic representation of llm hallucinations. *arXiv preprint arXiv:2410.02707*, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Aaditya Vikram Prasad, Connor Watts, Jack Merullo, Dhruvil Gala, Owen Lewis, Thomas McGrath, and Ekdeep Singh Lubana. Features as rewards: Scalable supervision for open-ended tasks via interpretability, 2026. URL https://arxiv.org/abs/2602.10067.

Yuxiao Qu, Anikait Singh, Yoonho Lee, Amrith Setlur, Ruslan Salakhutdinov, Chelsea Finn, and Aviral Kumar. Rlad: Training llms to discover abstractions for solving reasoning problems, 2025. URL https://arxiv.org/abs/2510.02263.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

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

Bhaktipriya Radharapu, Eshika Saxena, Kenneth Li, Chenxi Whitehouse, Adina Williams, and Nicola Cancedda. Calibrating llm judges: Linear probes for fast and reliable uncertainty estimation, 2025. URL https://arxiv.org/abs/2512.22245.

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.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. Bugpilot: Complex bug generation for efficient learning of swe skills, 2025. URL https://arxiv.org/abs/2510.19898.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. Swe-dev: Building software engineering agents with training and inference scaling, 2025. URL https://arxiv.org/abs/2506.07636.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025.

Lorenz Wolf, Robert Kirk, and Mirco Musolesi. Reward model overoptimisation in iterated rlhf, 2025. URL https://arxiv.org/abs/2505.18126.

Yiqing Xie, Emmy Liu, Gaokai Zhang, Nachiket Kotalwar, Shubham Gandhi, Sathwik Acharya, Xingyao Wang, Carolyn Rose, Graham Neubig, and Daniel Fried. Hybrid-gym: Training coding agents to generalize across tasks. *arXiv preprint arXiv:2602.16819*, 2026.

John Yang, Kilian Lieret, Carlos E 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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=63iVrXc8cC.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Anqi Zhang, Yulin Chen, Jane Pan, Chen Zhao, Aurojit Panda, Jinyang Li, and He He. Reasoning models know when they're right: Probing hidden states for self-verification, 2025a. URL https://arxiv.org/abs/2504.05419.

Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R. Tomz, Christopher D. Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock llm diversity, 2025b. URL https://arxiv.org/abs/2510.01171.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Swe-flow: Synthesizing software engineering data in a test-driven manner. *arXiv preprint arXiv:2506.09003*, 2025c.

Nonghai Zhang, Weitao Ma, Zhanyu Ma, Jun Xu, Jiuchong Gao, Jinghua Hao, Renqing He, and Jingwen Xu. Silence the judge: Reinforcement learning with self-verifier via latent geometric clustering, 2026. URL https://arxiv.org/abs/2601.08427.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025. URL https://arxiv.org/abs/2505.03335.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments. *arXiv preprint arXiv:2512.12216*, 2025.

Table 4: Math target yield under alternative adjudication rules. PROPEL's gain per generation depends on the reference rule and appears only with GLM-5.2 adjudication in this comparison. GLM-5.2 is a model judge. Scored rates exclude rejected tasks; per-generation rates include them. p denotes a two-sided Fisher exact test.

	Target among scored (%)		Target per generation (%)			
Adjudication rule	Base	PROPEL	Base	PROPEL	Δ (pp)	p
Qwen2.5-32B reference only	33.3	31.2	31.3	29.7	-1.7	0.443
Phi-4 reference only	30.5	26.2	27.0	22.0	-5.0	0.010
Strict two-oracle agreement (paper)	35.5	48.5	12.4	9.2	-3.2	0.022
Strict agreement, then GLM-5.2 adjudication	29.4	34.8	17.2	21.4	$+4.2$	0.019

A Oracle dependence of the math results

Math tasks have no executable ground truth, so the paper scores a generated task only when `Qwen2.5-32B-Instruct` and `Phi-4` each answer it twice and all four answers agree. The utility in Table 1 is conditional on this filter. For the 7B confirmatory cell, the in-family rows of Table 6 report the pooled conditional utility (16.5% Base, 27.4% PROPEL) and the pooled strict-oracle survival rate (19.9% Base, 12.3% PROPEL); their product gives the approximate unconditional yields of 3.3% and 3.4% quoted in Section 5. The survival rate for the 3B confirmatory cell was not retained.

To understand which tasks the filter removes, we generated 1,024 prompt-matched tasks from Base and from a reconstructed PROPEL math generator and rescored frozen $K{=}8$ `Qwen2.5-3B-Instruct` attempts under several adjudication rules (Table 4). On a larger set of 2,279 matched prompt pairs for which both references answer both tasks, the references disagree on 59.2% of Base tasks and 78.8% of PROPEL tasks. Tasks with reference disagreement receive markedly less solver support and more dispersed answers, both from the `Qwen2.5-3B-Instruct` solver and from an independent `Mistral-7B-Instruct-v0.3` solver, and the shift is not explained by topic, length, or answer form. A qualitative audit finds a mixture of reference errors, answer-extraction failures, ambiguous or defective tasks, and apparently valid enumeration tasks that the references could not settle. Strict agreement is therefore a precision filter that selects an easier population rather than a neutral validity test. Under either single reference or strict agreement, PROPEL does not increase target yield per generation; when GLM-5.2 adjudicates the tasks lacking strict agreement, PROPEL's per-generation target yield exceeds Base by 4.2 points (independent bootstrap 95% interval $[+0.8, +7.6]$). GLM-5.2 is a model judge rather than human ground truth, and this analysis uses a reconstructed generator, so it bounds the oracle dependence of the math result rather than replacing it.

B Topic taxonomy

The top-topic rate in Table 1 uses a fixed, ordered keyword taxonomy. The math topics, in matching order, are geometry, number theory, combinatorics, probability, sequences, linear algebra, calculus, sets and functions, algebra, and inequalities and optimization. The AZR algorithm families, in matching order, are strings, matrices and grids, sorting and ordering, extrema and ranges, filtering and counting, dictionaries and sets, modular arithmetic, statistics, sequences and recursion, arithmetic aggregation, boolean conditions, and list transformations. Each task receives the first category with a keyword in the first 2,000 characters of its lower-cased text; tasks that match no category are labeled *other*, which counts as a topic when computing the top-topic rate.

C Code-aware diversity

Table 5 reports the code-aware diversity measures summarized in Section 5. Unlike the text-overlap metrics in Table 1, these measures operate on program structure and on the reference model's representation space, and they are reported for all valid tasks and for tasks inside each generator's own target band. The validity-only generator provides the matched RL reference: RL toward validity alone already reduces abstract-AST uniqueness and representation recall relative to Base, whereas the probe reward increases structural uniqueness and program complexity while narrowing representation

Table 5: Code-specific diversity on AZR. PROPEL generates more structurally distinct and complex programs than either baseline, while covering less of the Base representation distribution. Target-band subsets retain high structural diversity. Max family is the largest per-seed share of one abstract-AST family.

Generator	Subset	n	Unique abstract AST	Max family (%)	Cyclomatic complexity	Base-cluster coverage	PRDC recall
Base	valid	2,118	0.970	0.56	4.96	1.000	—
Validity only	valid	2,913	0.927	1.35	4.30	0.938	0.771
PROPEL, 1–3/8	valid	2,822	0.997	0.31	7.29	0.875	0.361
PROPEL, 3–5/8	valid	2,666	0.998	0.23	7.54	0.969	0.555
PROPEL, 1–3/8	in 1–3/8	611	0.998	0.59	6.85	0.812	0.378
PROPEL, 3–5/8	in 3–5/8	481	1.000	0.68	6.90	0.812	0.541

coverage further. Representation metrics are complementary diagnostics rather than human judgments of semantic novelty.

D COLD TRANSFER ACROSS GENERATOR FAMILIES

The probe and the `Qwen3.5-4B` reference model are held fixed, while the trainable generator is replaced by `Mistral-7B-Instruct-v0.3` or `Phi-3.5-mini-instruct`. All hyperparameters are inherited from the in-family runs without per-family tuning, so the experiment bounds how far the recipe carries without re-tuning. Figure 3 in the main text and Table 6 report the results. For AZR, *Valid* is executable validity; for math, it is the strict two-oracle survival rate, and *Utility* is conditional on it. On AZR, the Mistral generator improves validity and utility by a margin comparable to the in-family generator relative to its own base model, whereas the Phi generator improves utility only modestly. On math, Mistral's conditional utility rises from 7.4% to 26.4%, but strict-oracle survival falls from 28.0% to 5.5%, so the unconditional yield of verified target tasks does not improve under this oracle.

Table 6: Transfer of a fixed probe across generator families without retuning. Mistral improves on AZR, while Phi gains less utility. In math, higher conditional utility coincides with lower oracle survival. Math validity is the strict-oracle survival rate, and utility is conditional on it.

Method	Generator	Utility ↑	Valid ↑	Self-BLEU ↓	Distinct-3 ↑	Top-topic ↓
AZR						
Base	Qwen (in-family)	0.1009	0.6654	0.711	0.376	0.318
PROPEL	Qwen (in-family)	0.1995	0.9085	0.809	0.266	0.735
Base	Mistral	0.0518	0.3955	0.657	0.400	0.376
PROPEL	Mistral	0.0957	0.6738	0.723	0.332	0.523
Base	Phi	0.0664	0.5801	0.786	0.296	0.320
PROPEL	Phi	0.0762	0.5547	0.768	0.313	0.359
Math						
Base	Qwen (in-family)	0.1654	0.1987	0.631	0.505	0.605
PROPEL	Qwen (in-family)	0.2739	0.1230	0.652	0.464	0.516
Base	Mistral	0.0742	0.2803	0.586	0.547	0.558
PROPEL	Mistral	0.2642	0.0547	0.852	0.263	0.853
Base	Phi	0.0663	0.2676	0.697	0.418	0.769
PROPEL	Phi	0.0971	0.1729	0.702	0.394	0.795

E SWE RL TRAINING RESULTS

The selected probe (Table 15) drives the generator-RL run summarized by the SWE rows of Table 1; full hyperparameters are listed in Appendix L. Figure 5 plots the per-step mean across rollouts of the final reward fed into GRPO (validity-gated probe logit with the invalid-reward floor for non-valid bugs). The reward rises from 0.04 at the first step to 0.35 at the final training step.

Table 7 breaks down the per-bug solver outcome on the held-out repositories for the first training seed, restricted to bugs with at least two valid solver trials. The base generator places 72.1% of its

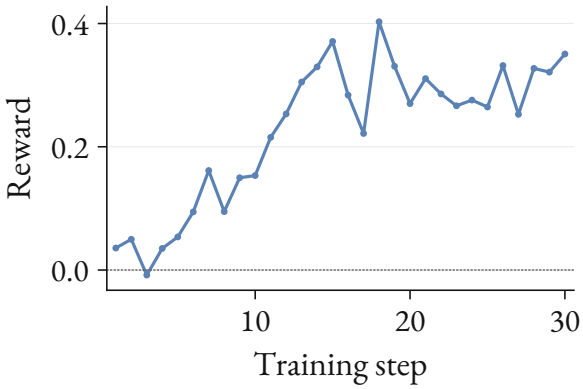


Figure 5: SWE generator-RL training curves: per-step mean of the final reward fed into the GRPO advantage (validity-gated probe logit with an invalid-reward floor of -0.2). The reward rises from 0.04 at the first step to 0.35 at the final selected checkpoint.

usable bugs in the trivially solved bucket. PROPEL reduces this share to 19.6% and moves the mass into the target band ($9.8\% \rightarrow 19.6\%$) and into bugs that the solver never solves ($18.0\% \rightarrow 60.8\%$).

Table 7: SWE bug difficulty on held-out repositories. PROPEL roughly doubles target-band yield by shifting away from trivial bugs, but also increases the unsolved share. Results use the first training seed and bugs with at least two valid solver trials.

Generator	n (≥ 2 valid)	Trivial	Too hard ($0/k$)	Target band
Base	61	44 (72.1%)	11 (18.0%)	6 (9.8%)
PROPEL	148	29 (19.6%)	90 (60.8%)	29 (19.6%)

F Additional Analysis on SWE

Table 8 shows that mean trajectory length drops from 26.8 to 15.1 generator turns over 30 steps while quality signals rise. Patch-validity climbs from 0.58 to 0.97, bug-validity from 0.32 to 0.79, and the shaped task reward from 0.28 to 0.68. The shift is also visible in the reasons for stopping: clean post-submission terminations (`stop`) grow from 151 to 247 of 256 episodes, while turn-cap exhaustions (`max_turns`) and single-response context overflows (`length`) collapse from 61 and 44 to 2 and 7 respectively.

G Solve-count distributions

The subfigures in Figure 2 report the aggregate 1–3@8 solve rate per (domain, target-solver) cell. The underlying solve-count distribution is shown in Figure 6 (PROPEL and PROPEL WCO vs. base policy). The proximal mechanism behind the main lift is the redistribution of mass from the $8/8$ bin (saturated) into the 1–3@8 band the probe is trained against.

H AZR 3B alternative Ensemble WCO pairings

The Ensemble WCO reward (Equation 4) takes the minimum logit across two probes. The AZR 3B row in Table 1 reports one choice of partner for the AZR 3B probe; here we compare it against two alternative partners drawn from the same sweep, evaluated under matched RL seeds. Variant (a) pairs with the partner that maximizes utility on the generated denominator; variant (b) is the partner used in the main results, chosen to balance utility against diversity; variant (c) pairs with a partner from a different layer and pooling, chosen to encourage representational diversity between the two probes. Per-checkpoint results are in Table 9. The three variants trace a clean utility-diversity tradeoff: (a) achieves the largest utility uplift (21.65% over a base of 10.09%) but produces the most concentrated

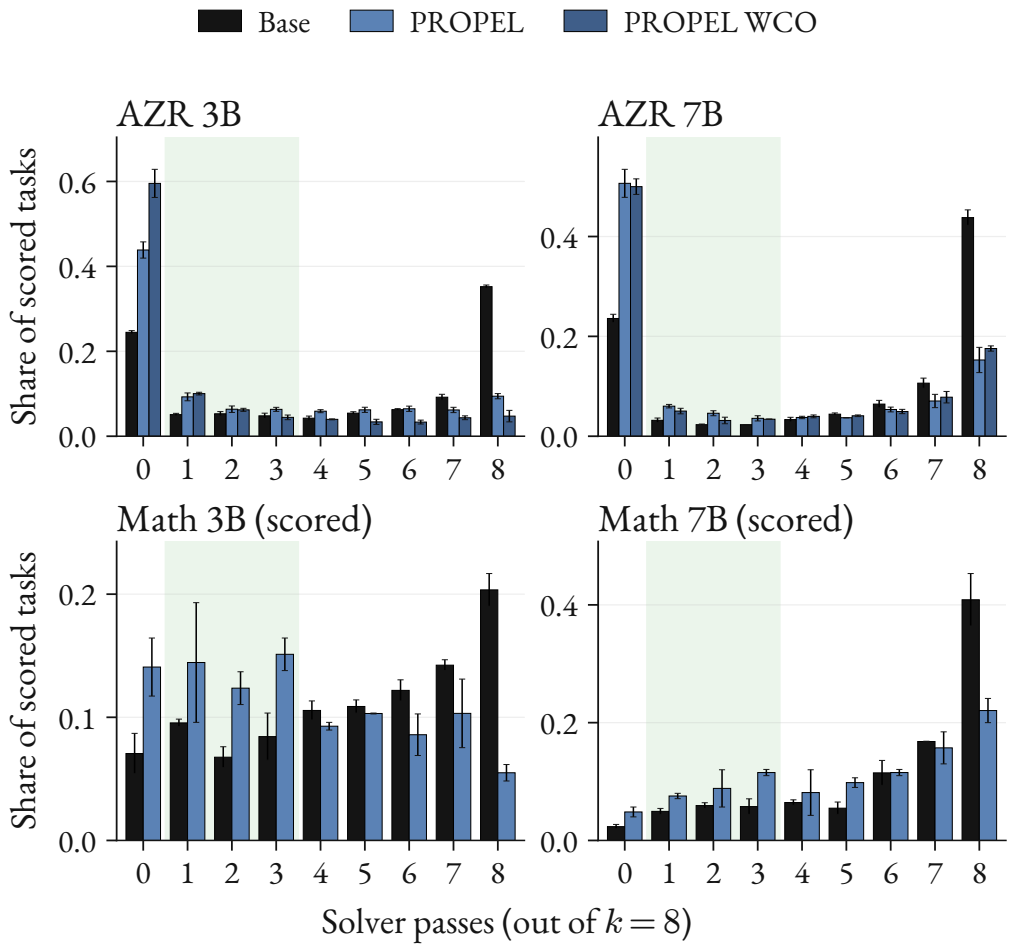


Figure 6: Solve-count distribution PROPEL vs. base policy per (domain, target-solver) cell and PROPEL WCO for the AZR domain. The shaded band marks the 1–3@8 band the probe is trained against.

Table 8: SWE training becomes shorter as validity improves. Mean trajectory length falls from 26.8 to 15.1 turns while reward rises. Stop counts are per 256 episodes; `max_turns` and `length` mark the 35-turn cap and 32k-token overflow.

Step	turns	task_rew	bug_valid	patch_app	n(`stop`)	n(`max_turns`)	n(`length`)
1	26.8	0.276	0.324	0.582	151	61	44
10	25.6	0.432	0.492	0.633	162	64	30
15	18.7	0.661	0.777	0.914	234	3	19
20	21.0	0.577	0.673	0.836	214	27	15
30	15.1	0.676	0.788	0.965	247	2	7

outputs (top-topic 0.883); (c) has the lowest top-topic concentration of any RL row (0.577) at a smaller utility uplift (16.24%); (b), the headline variant, keeps Self-BLEU-3 and top-topic clearly below (a) while retaining a large share of its utility (17.25%, $+71\%$ relative to base).

Table 9: Alternative probe pairings for AZR WCO. The utility-maximizing pairing produces the most concentrated outputs, while the headline pairing retains much of its utility with better diversity. Utility is mean $\pm$ standard error across RL seeds.

Variant	Utility ↑	Self-BLEU-3 ↓	Top-topic rate ↓
(a) utility-max	0.2165 ± 0.0210	0.812	0.883
(b) headline	0.1725 ± 0.0038	0.714	0.685
(c) layer-diverse	0.1533 ± 0.0076	0.749	0.724

I AZR 3B ADVERSARIAL COEVOLUTION

The AZR single-probe runs exhibit a standard proxy-optimization failure: the generator learns to produce tasks that score well under the probe, but many of those tasks come from the same semantic family. Adversarial co-evolution addresses this by mining false positives of the probe — valid tasks that the probe scores as useful but whose actual target-solver solve count falls outside the target band — as negatives, training an auxiliary probe on them, and folding it back into the next generator-RL run as a constraint.

Concretely, positive examples are the original probe-training corpus (valid tasks solved in 1, 2, or 3 of 8 attempts); negatives are the false positives just defined plus valid tasks from the dominant collapsed family discovered after single-probe PROPEL. The original probe continues to supply the utility signal during RL; the auxiliary probe acts as a conservative constraint, replacing the reward on valid tasks with the minimum of the two probe scores.

Using the auxiliary probe in isolation produces a different failure mode. In a standalone run the generator avoids the original collapsed family but moves to very easy string-manipulation tasks (99.7% validity, 1.8% Utility, 88.6% of generations solved on all 8 target-solver trials, valid-only top-topic share 99.9%). The auxiliary probe is therefore used as a constraint on top of the utility objective rather than as a replacement.

Table 10 reports the follow-up under the same Utility, Valid, Self-BLEU-3, Distinct-3, and Top-topic metrics used in the main AZR evaluation. The false-positive variant raises Utility over base ($0.1009 \rightarrow 0.1602$) and Valid over base ($0.6654 \rightarrow 0.8486$) while keeping the diversity metrics close to base, and recovers most of the unconstrained single-probe run's Utility (0.1995) without its semantic concentration.

Table 10: Adversarial probe coevolution on AZR. Both variants reduce repetition and topic concentration relative to the single probe, but retain less of its utility gain. Each adversarial variant uses one RL seed; the reference rows average three seeds.

Condition	Utility ↑	Valid ↑	Self-BLEU-3 ↓	Distinct-3 ↑	Top-topic ↓
Base	0.1009	0.6654	0.7110	0.3760	0.3180
Single probe	0.1995	0.9085	0.8090	0.2660	0.7350
Semantic adversarial	0.1372	0.8911	0.7200	0.3670	0.4070
False-positive adversarial	0.1533	0.8398	0.6930	0.3890	0.3210

J SWE Data Pipeline

J.1 Repository sourcing

All repositories used in this work are drawn from the **SWE-smith** Python profile registry (Yang et al., 2026), which provides for each repository a designated buggy commit, a pre-built Docker sandbox image, a default test runner, and the corresponding test framework. From this registry we apply a four-stage filter to obtain a tractable, non-flaky working set:

1. **Hard-gate exclusions.** We exclude three repositories (`paramiko`, `sqlfluff`, `autograd`) known to cause network/I/O flakiness, prohibitively long test suites, or environment build failures. (`sqlfluff` is later partially recovered through a re-evaluation pass.)
2. **Preflight quality filter.** A pre-pass runs every remaining SWE-smith repository in its sandbox once on the unmodified commit and records build status, test count, baseline test duration, and baseline test failures. Repositories pass if `status = "ok"`, tests > 0, and baseline_failures ≤ 1. The Probe-Train expansion additionally uses baseline_duration $<$ 30 s; the held-out RL-Train and Held-Out Eval selections allow a small number of slower high-coverage repositories to preserve split breadth.
3. **Quality ranking.** Surviving repositories are ranked by *tests-per-entity* (a coverage proxy) and the top-N are taken, where N is split-specific.
4. **Empirical yield filter.** After running the bug-generation agent and the K=3 solver labeling pipeline, repositories with no trajectories satisfying our usability criteria (`bug_valid_new` ≥ 1 and `valid_trials` > 0; see Sections J.3–J.4) are dropped from the labeled dataset.

Funnel. The four stages reduce the SWE-smith registry to a labeled corpus: 71 repositories were materialized across the Probe-Train, RL-Train, and Held-Out Eval collection roots. The paper split accounting below uses 68 attempted repositories: 46 packaged training repositories plus all 22 held-out RL-Train/Held-Out Eval repositories. Of these, 65 produced at least one solver-labeled bug with `valid_trials` > 0, and **64 contribute at least one row** under the ≥ 2-valid-trial filter (45 Probe-Train, 11 RL-Train, 8 Held-Out Eval row-contributing repositories). The Held-Out Eval collection root contains 11 planned repositories; three yield zero usable bugs and therefore do not contribute rows to the labeled split. Post-RL generator outputs are not counted in this dataset version.

J.2 Splits

We construct three repo-disjoint splits, summarized in Table 11.

Why ≥ 2 valid trials. The probe-training filter requires at least two valid solver trials per bug (out of K=3). A single trial gives a noisy solve-rate estimate ($\hat{r} \in \{0, 1\}$, variance 0.25 at true $r = 0.5$); two trials halve that variance and allow $\hat{r} \in \{0, 0.5, 1\}$. The filter retains 73.2% of Probe-Train rows (1,757/2,399), 78.0% of RL-Train (85/109), and 84.5% of the usable Held-Out Eval rows (49/58), while removing the noisiest single-trial bugs from training. We report main results under both filters but use ≥ 2 for all probe training and held-out scoring.

The roles align with our experimental protocol: **Probe-Train** is used for stratified-by-(repo×label) cross-validation when training the probe; **RL-Train** provides cross-repository held-out evaluation

Table 11: Repository-disjoint SWE data splits. Usable bug yield is lowest on held-out repositories despite similar solver-trial validity. Yield counts bugs with at least two valid trials per attempted episode; Valid measures trial-level validity and pooled $\bar{r}$ measures solves per valid trial.

Split	Repos	Eps	Bugs (≥ 1)	Bugs (≥ 2)	Yield (≥ 2)	Valid %	Pooled $\bar{r}$
Probe-Train	46	5,892	2,399	1,757	29.8%	72.1	0.626
RL-Train	11	550	109	85	15.5%	71.3	0.718
Held-Out Eval	11	550	58	49	8.9%	73.4	0.660
Total attempted	68	6,992	2,566	1,891	27.0%	—	—

of probe quality and is also the bug pool against which the generator is RL-trained; **Held-Out Eval** is the strict held-out generalization set for the post-RL generator and supplies bugs for downstream solver fine-tuning. Repositories never overlap across splits.

J.3 Bug generation (generator)

Bug validity (`bug_valid`) follows the criteria in Section 3; the verifier replays the agent's diff via `git apply -recount` before running the test suite. We run 50 episodes per repository for RL-Train and Held-Out Eval, and up to 280 episodes per repository for the Probe-Train pool. The Probe-Train pool is filtered to `bug_valid_new = 1` after a corrected validity pass that fixes a bash-shell parsing bug in the original verifier. The RL-Train and Held-Out Eval splits use max-turns 25, temperature 0.7, max-tokens 2048, top_p 0.95, and top_k 20.

We define the end-to-end *yield* of a repository as the fraction of attempted generator episodes that pass the bug-validity gate and yield at least two valid solver trials. Per-repository yields range from 0% to 91%; the dataset-wide yields are 29.8%, 15.5%, and 8.9% for Probe-Train, RL-Train, and the full Held-Out Eval split respectively.

The generator receives the system and instruction prompt shown in Appendix N. The full prompt is bash-only. The `browse_index` reference in the prompt body refers to a small helper script the generator can invoke *from the shell* (e.g. `bash -c "python /workspace/browse_index.py"`), that enables easier repository navigation.

J.4 Solver labeling

Each of the $K = 3$ OpenCode solver trials (Section 3) uses three tools (`browse_index`, `show`, `edit`) and runs in a fresh Docker sandbox with a $1{,}800$ s wall-clock budget and 2 CPU / 4 GB / 10 GB resource limits. A trial returns reward $\in \{0, 1\}$ if the verifier ran (the solver submitted a solution patch within the budget), else `null` (timeout, setup failure, or runtime error). The default sampling parameters used for all label runs are $T = 0.6$, top_p $= 0.95$, top_k $= 20$ (the values loaded by vLLM from Qwen3.5-27B's `generation_config.json`).

K Probe Training Data and Selection

This appendix reports the probe-training data, the sweep grid, the selection procedure, and the frozen probes used by PROPEL as the RL reward.

K.1 Probe-training data

For each (domain, target solver) pair we collect one balanced probe-training set. For SWE the positive class is too sparse to discard data, so we instead train with a class-reweighted loss on all $1{,}757$ rows of the Probe-Train split (Appendix J). The resulting dataset sizes are reported in Table 12.

K.2 Activation extraction and candidate grid

Activations are read from the frozen reference model. For math and AZR, each generated task is a single rendered output and is represented by one pooled activation vector. For SWE, each task is a

Table 12: Probe-training datasets. Math and AZR balance the target-band classes by downsampling, while SWE retains all examples and weights the loss to address class imbalance.

Domain	Target solver	Balance	Positives	Negatives	Probe-train rows
Math	Qwen2.5-3B-Instruct	downsample	2,340	2,340	4,680
Math	Qwen2.5-7B-Instruct	downsample	1,313	1,313	2,626
AZR	Qwen2.5-3B-Instruct	downsample	1,765	1,765	3,530
AZR	Qwen2.5-7B-Instruct	downsample	1,167	1,167	2,334
SWE	Qwen3.5-27B	weighted loss	362	1,395	1,757

multi-turn agent trajectory, so we extract one activation vector per generated assistant turn and pool across the trajectory. Math and AZR cover an $8 \times 3 \times 2 = 48$-cell grid per (domain, student). SWE covers a larger 810-cell grid because it additionally varies trajectory aggregation and class-imbalance handling.

Table 13: Probe sweep and training settings. SWE requires a larger search to account for trajectory pooling and class imbalance.

	Math / AZR	SWE
Reference model	`Qwen/Qwen3.5-4B`	`Qwen/Qwen3.5-27B`
Layers L	$\{3, 7, 11, 15, 19, 23, 27, 31\}$	$\{5, 15, 31, 47, 63\}$
Poolings	`last_token`, `mean_full`, `mean_last_50`	`last_token`, `mean_full`, `mean_last_{50,5,3}`, `mean_last_half`, `max`, `first_last_concat`, `mean_std_concat`
Architectures	`linear`, MLP	`linear`, MLP
Balance	downsample	downsample *or* weighted loss
Split	80/10/10 train/val/test	5-fold cross-repository
Max sequence length	2,048 (math), 4,096 (AZR)	32,768
Cells per sweep	48	810

Training. The MLP head uses $d \to 512 \to 128 \to 2$ with dropout 0.3. All probe heads are trained with AdamW at learning rate 10^{-3}, weight decay 10^{-4}, batch size 64, and up to 50 epochs with early-stopping patience 7. Math/AZR probes are fit on downsampled balanced datasets. SWE keeps all Probe-Train rows under the weighted-loss setting because the positive class is sparse.

K.3 Probe selection

For math and AZR we sweep reference-model layer, token pooling, and linear *vs.* MLP probe heads. For SWE, where each task is a multi-turn agent trajectory, we also sweep trajectory-level pooling and class-imbalance handling. For each (domain, target solver), sweep cells are ranked by held-out balanced accuracy subject to calibration; for Math/AZR, ties among otherwise comparable candidates were broken by the variance of the RL reward on a fixed sample of n=512 base-policy completions, avoiding probes whose reward is nearly constant under the base policy.

For SWE, the random 80/10/10 split is replaced by 5-fold cross-repository validation. Each fold holds out entire repositories, testing whether the probe transfers beyond repositories seen during probe training rather than merely to unseen bugs from familiar repositories. The selected SWE probe is the configuration with the best cross-fold balanced accuracy under weighted-loss training, averaged over random seeds $\{0, 1, 2\}$. On the cross-repository held-out fold, the Spearman correlation between probe score and ground-truth solver solve-rate is $\rho = 0.418$.

We only use the selected probe for each (domain, target solver); Table 14 lists the probes used as the RL reward.

Table 14: Selected reward probes. Math and AZR use MLP heads, while SWE uses a linear probe over pooled trajectory activations. ECE is measured before calibration; SWE applies temperature calibration during RL scoring.

Domain	Target solver	Layer	Pooling	Arch.	Bal. acc. ↑	ECE ↓
Math	Qwen2.5-3B	11	`last_token`	MLP	0.611	0.041
Math	Qwen2.5-7B	15	`last_token`	MLP	0.660	0.088
AZR	Qwen2.5-3B	11	`mean_last_50`	MLP	0.649	0.093
AZR	Qwen2.5-7B	11	`mean_last_50`	MLP	0.622	0.054
SWE	Qwen3.5-27B	63/64	`mean_last_half`	linear	0.594	0.294

K.4 Probe-Sweep Results AZR/Math

Figures 7 and 8 render the full $8 \times 3 \times 2$ sweep grids as validation-balanced-accuracy heatmaps for the Math and AZR domains under the 1–3/8 label scheme, faceted by architecture (rows) and target-solver size (columns).

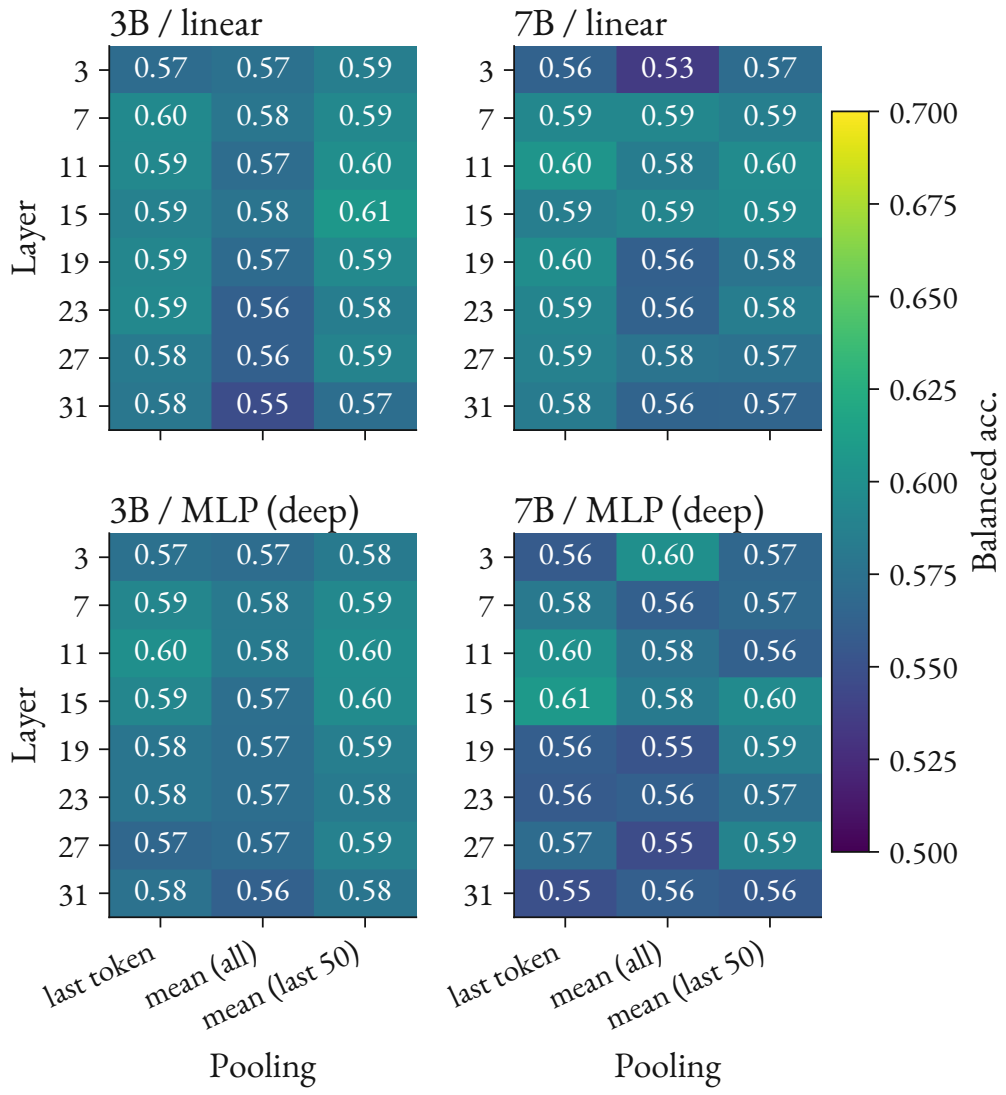


Figure 7: Probe-sweep validation balanced accuracy heatmap on the Math domain (1-3@8 label scheme), faceted by architecture (rows) and student size (columns).

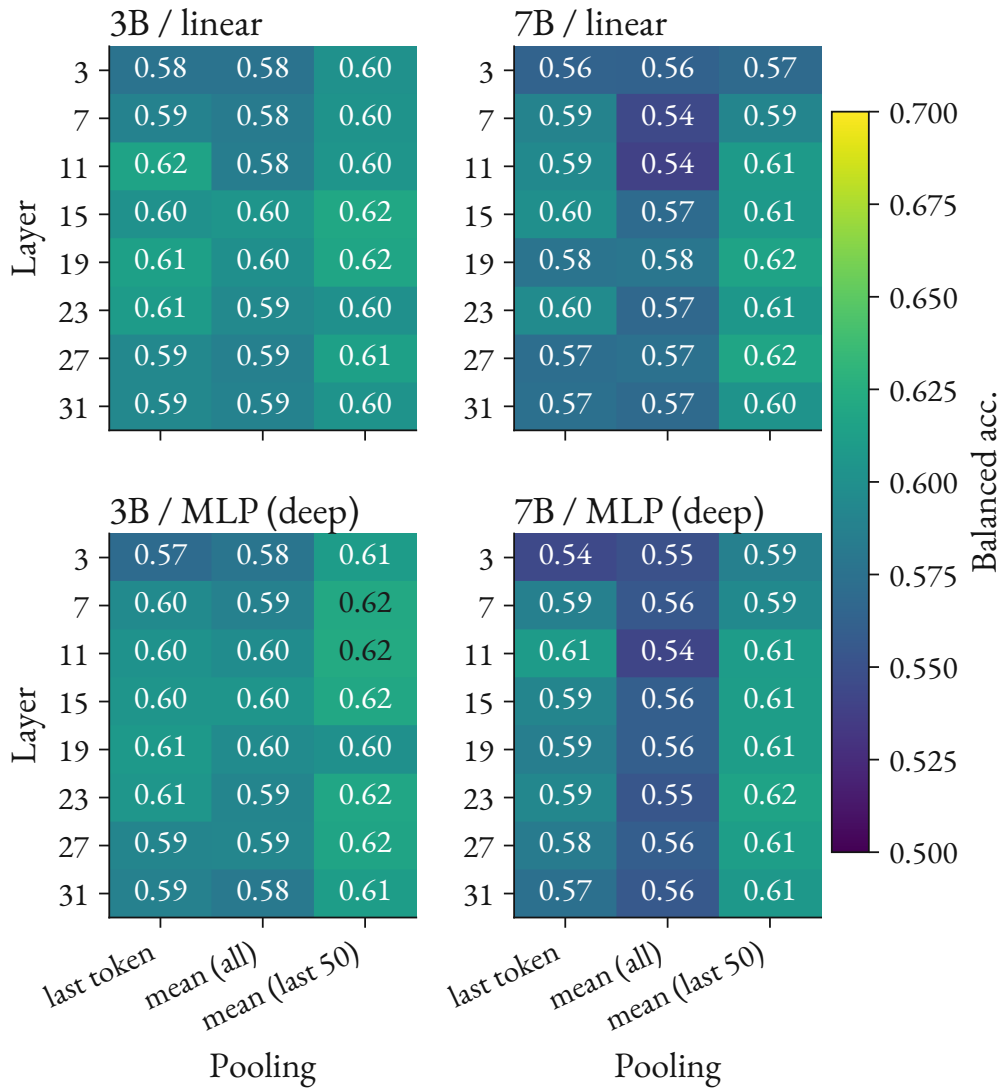


Figure 8: Probe-sweep validation balanced accuracy heatmap on the AZR domain (1-3@8 label scheme). Same layout as Figure 7.

K.5 Probe Sweep Results SWE

We run a complete probe-sweep grid for the SWE domain on the Probe-Train split (Appendix J), training each cell three times with seeds $\{0, 1, 2\}$ and averaging.

Target. The probe is trained on the fixed-student utility signal U_S (Section 3.2) with $S =$ `Qwen3.5-27B` at $K = 3$ solver trials, restricted to bugs with ≥ 2 valid solver trials: positives are bugs whose realized solve-rate is in $(0, 1)$ (i.e. exactly $1/3$, $2/3$, or $1/2$ when one of three solver trials is null); negatives are bugs that either fail every valid trial ($0/2$, $0/3$) or succeed on every valid trial ($2/2$, $3/3$). On Probe-Train this gives 362 positives and $1{,}395$ negatives ($N = 1{,}757$ across 45 repositories).

Grid. The full 810-cell SWE sweep (layer $\times$ trajectory pooling $\times$ architecture $\times$ class-imbalance strategy) and the shared training hyperparameters are listed in Table 13. Epoch selection per cell is by cross-validated balanced accuracy on the 5-fold cross-repository split.

Selected configuration. Selecting on cross-repo balanced accuracy under weighted-loss training yields the configuration in Table 15.

Table 15: Selected SWE probe. Cross-repository validation gives modest predictive accuracy, while held-out score correlation varies substantially across seeds. Values average three seeds; CV metrics additionally average five repository folds.

Layer	Aggregation	Arch	Balance	Heldout ρ	CV AUC	CV F_1	CV Bal. Acc.
63	`mean_last_half`	linear	weighted loss	0.418	0.596	0.376	0.594

Per-seed held-out Spearman ρ for the selected configuration is 0.463, 0.383, 0.407 at seeds 0, 1, 2 respectively.

L RL Hyperparameters

Table 16 provides the generator-RL hyperparameters for all three domains. Math and AZR share a single training recipe launched with `accelerate` on 4 GPUs per run; SWE uses the SkyRL GRPO stack across 3 nodes (24 H100s total) with FSDP. The KL coefficient β, group size G, and selected checkpoint were screened on a per-cell grid using validity, diversity (Self-BLEU-3, Distinct-3, top-topic), and collapse alerts. The KL and reward-composition screening sweeps are reported in Section 5 (Figures 4a and 4b).

M Qualitative examples: base vs RL on AZR

To make the frontier-rate gain in the main results concrete we surface four representative AZR-induction tasks from the confirmatory multi-seed runs underlying Table 1: two from the base generator that illustrate the failure modes the probe reward is shaping *against* (one with 0 of 8 hidden tests passed, i.e. the generated function does not match the natural-language message and the solver cannot recover it; one with 8 of 8, a textbook "square the input" that the fixed student saturates trivially), and two from the PROPEL trained generator that land in the 1–3@8 learnable-frontier band the probe targets. Each example is the shortest valid sample in its solve-count bucket, drawn from the base evaluation run and the PROPEL checkpoint at step 30.

Base policy — 0 of 8 solved (unsolvable / mis-specified).

AZR / 3B target / Base policy / eval seed 12101 / sample 946

```
# Function
def f(n):
    # your function body
    result = []
```

Table 16: Generator-RL settings. PROPEL and WCO share the same optimization settings within each domain; only the reward formulation changes.

	Math / AZR	SWE
Generator		
Base model	Qwen3.5-4B	Qwen3.5-27B
Trainable parameters	LoRA adapters	full fine-tune
LoRA rank r	16	—
LoRA α	32	—
LoRA dropout	0.05	—
LoRA target modules	all 7 attn./MLP projections	—
Mixed precision	bf16	bf16
Optimizer		
Algorithm	GRPO (Shao et al., 2024)	GRPO (SkyRL)
Optimizer	AdamW	AdamW
Learning rate	5×10^{-5}	1×10^{-6}
Weight decay	0.0 (default)	0.01
Adam (β_1, β_2)	$(0.9, 0.999)$	$(0.9, 0.999)$
Max gradient norm	1.0	1.0
Sampling & rollouts		
Temperature	0.9	0.7
Top-p	0.95	0.95
Top-k	—	20
Group size G (rollouts/prompt)	4	16
Max generation length	512 (Math) / 1024 (AZR)	2048 tokens / turn
Max input/context length	4096 tokens	32,768 tokens
Max multi-turn rollout depth	1 (single-shot)	35 turns
GRPO loop		
Per-device train batch (generations)	8	16
Gradient accumulation steps	8	1
Effective rollouts per step	$8 \times 4 \times 8 = 256$	$16 \times 16 = 256$
PPO inner iterations	1 (on-policy)	1 (on-policy)
Advantage & policy-loss		
Advantage estimator	TRL `GRPOTrainer`	advantage_estimator: loop
Policy loss type	TRL default	Dr.GRPO-style constant-length scaling
Importance-ratio clip ε	TRL default	dual-clip, ε_{low}=0.20, $\varepsilon_{\text{high}}$=0.28
KL anchor in loss	yes ($\beta \cdot \mathrm{KL}[\pi_\theta \| \pi_{\text{ref}}]$)	off (`use_kl_loss=false`)
KL anchor in reward	no	off (`use_kl_in_reward=false`)
KL coefficient β	0.05 (AZR), 0.10 (Math)	—
Reward shaping		
Reward composition	R_{hard} / R_{WCO}	R_{hard}
Invalid-reward floor r_{bad}	-0.2	-0.2
Validity gate $\mathcal{W}(x)$	sandbox-runnable (AZR); no strict-oracle gate (Math)	verifier (`bug_valid`$\geq$ threshold)
Bug-valid threshold (SWE)	—	1.0
Context-overflow reward	—	0.0
Probe gate behavior	always	gated on `bug_valid`
Schedule & selection		
Total GRPO steps	50	30
Checkpoint save interval	every 10 steps	every 5 steps
Compute		
Probe inference workers	co-located with policy	4 Ray actors $\times$2 GPUs each
Max sequence length	4,096 tokens	36,864 tokens
Parallelism	accelerate DDP, 4 GPUs	FSDP, 3 nodes $\times$8 H100s
Wall-clock per run	$\approx$ 6 h on 4$\times$H100	$\approx$ 14.5 h on 24$\times$H100

```
   while n > 0:
      digit = n % 2
      result.append(digit)
      n //= 2
   return [x for x in reversed(result)]

# Inputs (10)
 1,
 2,
 100,
 1023,
 15,
 17,
 255,
 0,
 64,
 999

# Message
This function converts a non-negative integer into its binary
    representation as a list of integers (0s and 1s), starting from
    the most significant bit. If the input is 0, return [0].
```

Base policy — 8 of 8 solved (trivial / saturated).

AZR / 3B target / Base policy / eval seed 12101 / sample 278

```
# Function
def f(n):
   return n * n

# Inputs (10)
 0,
 1,
 2,
 3,
 4,
 5,
 10,
 -3,
 100,
 -1

# Message
The function takes a single number and returns the square of that
    number. What is the output for the given inputs?
```

RL (single probe A03) — 2 of 8 solved (learnable frontier, 1-3@8 band).

AZR / 3B target / RL (single probe A03, ckpt 30) / RL seed 9101 / sample 867

```
# Function
def f(items):
   if not items:
      return 0
   sorted_items = sorted(items)
   prefix_sums = [0] * len(sorted_items)
   prefix_sums[0] = sorted_items[0]
   for i in range(1, len(sorted_items)):
      prefix_sums[i] = prefix_sums[i - 1] + sorted_items[i]
   return prefix_sums[-1]
```

```
# Inputs (10)
  [1, 2, 3],
  [10, 20, 30],
  [5, 5, 5],
  [-1, 0, 1],
  [100, 50, 25],
  [3.5, 2.5, 1.5],
  [],
  [7],
  [1, 3, 5, 7, 9],
  [9, 8, 7, 6, 5, 4, 3, 2, 1]

# Message
The function `f` sorts the input list in ascending order. It then
    computes a cumulative sum (prefix sum) of the sorted elements. The
    result is the final element of this prefix sum list (i.e., the sum
    of all sorted elements). If the input list is empty, the function
    returns 0.
```

RL (single probe A03) — 2 of 8 solved (learnable frontier, 1-3@8 band).

AZR / 3B target / RL (single probe A03, ckpt 30) / RL seed 9101 / sample 917

```
# Function
def f(lst):
   if not lst:
      return 0
   sorted_lst = sorted(lst)
   mid = len(sorted_lst) // 2
   left_sum = sum(sorted_lst[:mid])
   right_sum = sum(sorted_lst[mid:])
   if len(sorted_lst) % 2 == 1:
      center = sorted_lst[mid]
      return left_sum + center
   return left_sum + right_sum

# Inputs (10)
  [1, 2, 3],
  [10, 20, 30],
  [5, 5, 5],
  [1, 3, 5, 7, 9],
  [-2, 0, 2],
  [100],
  [],
  [1, 2, 3, 4, 5, 6],
  [99, 88, 77, 66],
  [0, 0, 0, 0, 0]

# Message
The function takes a list of numbers, sorts them, and returns the sum
    of the first half plus the middle element if the list length is
    odd. If the list length is even, it returns the sum of the two
    halves (which are equal in count). If the list is empty, it
    returns 0.
```

N GENERATOR PROMPTS

N.1 MATH DOMAIN

Math task generator — system + user template

```
[system]
You are a creative mathematics problem designer. You output ONLY a
    problem statement in the exact format below. No solutions, no
    hints, no teaching notes, no commentary about difficulty or
    student struggles.

Output format:
## Problem
[Your problem statement here]

Rules:
- Output NOTHING except the ## Problem header and the problem text
- The problem must be from a DIFFERENT area of mathematics each time
    (geometry, number theory, combinatorics, probability, algebra,
    calculus, graph theory, logic, etc.)
- The problem must be well-posed, self-contained, and have a definite
    answer
- The problem should involve a creative twist or unexpected connection

[user]
Here are example problems from different areas of mathematics:

{examples}

Create ONE new problem from a DIFFERENT area of mathematics than the
    examples above. Remember: output ONLY the problem in the format:

## Problem
[problem statement]

Do NOT include solutions, hints, teaching notes, or any other text.
```

N.2 AZR FUNCTION INDUCTION DOMAIN

AZR/induction task generator — system + user template

````
[system]
You are a creative generator of Python induction tasks. You output
    ONLY the task components below, in the exact format specified. No
    extra prose, no commentary, no teaching notes.

Output format:

```python
def f(...):
   # your function body
   ...
```

```input
arg1, arg2
```
```input
arg3, arg4
```
````

```
... (exactly {num_inputs} `input` blocks)

```message
<natural-language hint to the test subject>
```

Requirements:
- Name the entry function `f`. It must return a value. At least one
    argument.
- The function must be DETERMINISTIC: no random, time, datetime, I/O,
    or external state.
- Execution must complete in < 2 seconds of CPU time on all inputs.
- Do NOT import any of these modules: {banned}
- Each `input` block contains comma-separated Python literals -- the
    args to `f`. Wrap strings in quotes; dicts/lists/tuples are fine.
- Provide EXACTLY {num_inputs} input blocks. Each input must produce
    a different output.
- The `message` is a natural-language hint describing what `f` does,
    intended to help a test subject recover the function from only the
    (input, output) pairs.
- Do NOT include the code snippet inside the message block.

[user]
Example induction tasks (for format reference only -- invent
    something NEW):

{examples}

Now create ONE NEW induction task. The function must implement a
    DIFFERENT algorithm than the examples above. Output the code
    snippet, exactly {num_inputs} input blocks, and a message --
    nothing else.
```

N.3 SWE DOMAIN

SWE generator — system prompt + instruction

```
[system]
You are a skilled software engineer. You interact with a computer
    shell to complete coding tasks.

Your response must contain exactly ONE bash code block with ONE
    command (or commands connected with && or ||).
Include a THOUGHT section before your command where you explain your
    reasoning.
Format your response as shown:

THOUGHT: Your reasoning and analysis here.

```bash
your_command_here
```

Failure to follow these rules will cause your response to be rejected.

[instruction]
You have a Linux shell at `/workspace`. One command per turn.

Every response must be: THOUGHT (1-2 sentences), then one ```bash
    block with your command inside it.
```

````
THOUGHT: I need to browse the codebase index to find a good target.

```bash
browse_index
```

## Your goal: introduce a subtle bug that causes test failures

You are a skilled developer. Introduce a bug that:
1. Causes existing tests to fail
2. Has exactly one correct fix
3. Is hard to find by reading test error messages alone

Use `browse_index` to explore the codebase. Read test files to
    understand what's tested.
Find a function with interesting logic and make a subtle change to
    its behavior.

The hardest bugs to find change WHERE data goes -- which dict key
    stores a result,
which index slices a string, what a function returns in an edge case.
    These are hard
because the test error doesn't point directly to the changed line.

Simple condition flips (if X -> if not X) are too easy -- the test
    error immediately
reveals which branch is wrong. Avoid those.

Be creative. Explore different files and functions each time. The
    more novel, the better.
````