PREDICTIVE, NOT REACTIVE: TASK FILTERING FOR COMPUTE-EFFICIENT GRPO TRAINING OF CODE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Group Relative Policy Optimization (GRPO) learns by comparing several rollouts on the same task. If every rollout receives the same reward, their relative advantages are zero. The task then provides no policy gradient, even though all of its rollouts have already been generated. This is common in repository-level code repair for two reasons. A binary reward treats a partial repair as a complete failure: a patch that fixes some failing tests receives the same reward as one that fixes none. In addition, tasks that are too difficult for the current policy often give the same reward for every rollout. In both cases, GRPO spends computation on a group from which it cannot learn. We propose PRESAGE (PREdictive SAmpling with Graded Evaluation), which combines task filtering with a per-test reward. Before generating rollouts, a pass-rate predictor reads the task description and estimates the score that the current policy would obtain. Tasks with predicted scores below a threshold are skipped, so no rollout budget is spent on them. For the remaining tasks, the reward gives credit for the fraction of failing tests repaired and subtracts a penalty for breaking tests that previously passed. This allows different partial repairs to receive different scores. A task can therefore provide a learning signal even when no rollout solves it completely, as long as the rollouts make different amounts of progress. These per-test scores are also used to train the predictor. Because the policy improves during training, PRESAGE recalibrates the predictor and checks the full task pool again in every cycle. A task skipped earlier can return once the policy becomes strong enough to make progress on it. This keeps later batches from filling up with tasks that produce no gradient. With the same task pool, rollout budget, and reward function, predictive filtering increases the fraction of sampled groups that provide gradients from 16.1% to 58.6%. The same budget supports 40 policy updates with PRESAGE, compared with 7 under dynamic sampling. Compared with uniform GRPO, PRESAGE improves the fail-to-pass rate by 9.8 percentage points and the resolved rate by 10.5 points. Compared with the supervised fine-tuned starting model, the improvements are 11.1 and 12.2 points, respectively.

1 INTRODUCTION

Repository-level software engineering has become a common testbed for code agents (Jimenez et al., 2024; Yang et al., 2024), and reinforcement learning has increasingly been used to push them past what supervised fine-tuning reaches (Pan et al., 2025; Wei et al., 2025). Group Relative Policy Optimization (GRPO) (Shao et al., 2024; DeepSeek-AI, 2025) is one of the most widely used methods. It works in rounds: each round picks a batch of tasks, has the agent attempt each one K separate times, and scores every attempt by running the repository's unit tests. The learning signal comes from comparing those K scores *against each other*: attempts above the task's own average are reinforced and attempts below it discouraged, so no value network and no human preference labels are needed.

That design has a hidden cost. Suppose the agent attempts a bug fix and fails all K times. Each attempt was a full episode: tens of LLM calls, a Docker container, a complete test-suite run. But all K scores came out identical, so comparing them yields nothing and the update learns nothing from

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.

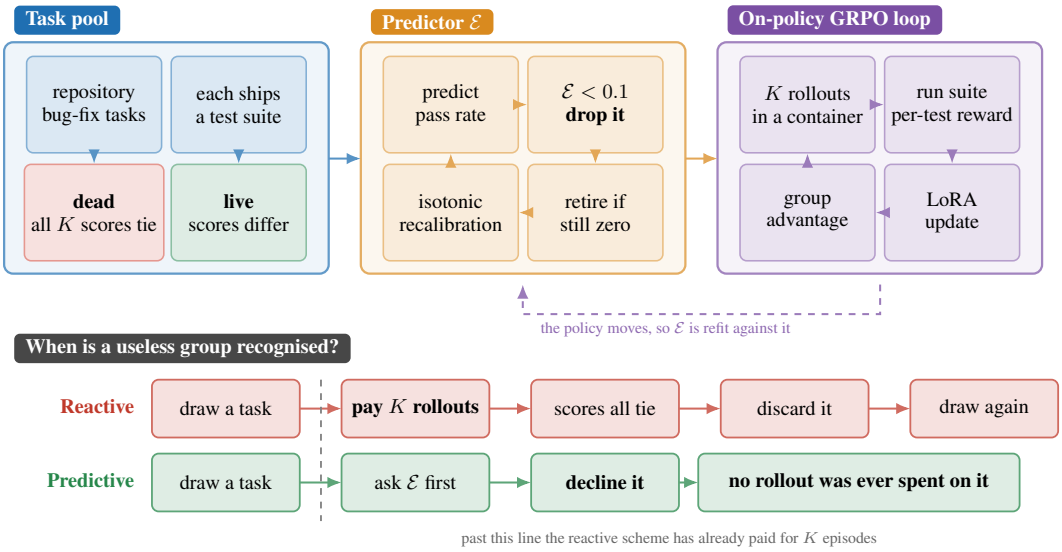

Figure 1: **Where a GRPO rollout budget goes, and where we intervene.** Only a small share of the pool is live, so a uniform sampler spends most of its budget on groups that cannot produce gradient. Prior work removes such groups after paying for them; $\mathcal{E}$ declines them beforehand.

that task, even though the compute was spent in full. The same happens if the agent succeeds every time.

This is not a rare corner case, nor peculiar to our pool. Untrained open backbones resolve only a few percent of repository-level tasks: the 32B base in SWE-Gym resolves 3.0% of SWE-bench Lite and 7.0% of Verified (Pan et al., 2025). Under a pass/fail reward that leaves most groups all-zero. In our own pool too, most tasks are *dead* under the base policy: every attempt on them scores alike, so the group carries no gradient and teaches nothing. A sampler that picks tasks uniformly sends most of its budget into exactly these wasted episodes (Figure 1). Whatever makes a group come back uniform, the sampler sees the same thing: K episodes spent, identical rewards, no gradient. A task is *live* when its attempts can still disagree; since observing that costs rollouts, we measure it once under the base policy at $K{=}8$ and count a task live when its mean clears 0.1; Section 4 gives the fraction this leaves.

Two lines of work address this waste, and each leaves something on the table. The first makes the reward finer: score a patch by how many synthesized tests it passes (Zeng et al., 2025), or feed per-test outcomes back through a learned critic (Le et al., 2022), so that repairing 3 of 5 scores 0.6 rather than 0. Neither ties that partial credit to whether the task is actually resolved, so a policy can improve its score without ever closing a task, and neither removes a task the policy cannot touch at all. The second line changes which tasks are selected: discard groups whose samples agree (Yu et al., 2025), or filter by difficulty (Bae et al., 2026). Discarding happens after the group has been rolled out, so the compute is already gone; filtering by difficulty uses a label fixed before training, which goes stale as the policy moves. The two address different causes. A task can go silent because the reward is too coarse to separate a patch repairing four of five tests from one repairing none, which a fractional reward fixes. Or because nothing the agent could write would score differently, where no scoring rule helps: grading identical attempts more finely leaves them identical, and the episodes were wasted the moment they were drawn. The only remedy is to not draw that task.

Not drawing a task means knowing in advance that it is hopeless, which is a prediction about the environment: what would it score if the current policy attempted this? Fitting that predictor needs labels with variation, and pass/fail labels have almost none, so the environment has to answer more finely first. We score an attempt by the fraction of originally failing tests its patch repairs, less a penalty on any originally passing test it breaks; on those scores we fit a pass-rate predictor $\mathcal{E}$, consult it before a cycle draws anything, and drop what it judges hopeless for no rollouts at all. Since what $\mathcal{E}$ predicts depends on the policy as well as the task, we refit it every cycle over the

whole training pool, so a task ruled out early returns once the policy grows into it. We call the three together PRESAGE.

Contributions.

- **Predictive task routing.** $\mathcal{E}$ is asked what the current policy would score on a task before any episode is run, and the tasks it puts below a threshold are dropped at no cost. A usable gradient group then costs 14 rollouts instead of the 49 a reactive scheme pays here, so the same rollout budget buys forty policy updates instead of seven. On the held-out set the resolved rate improves by 12.2 pp and the fractional rate by 11.1 pp over the supervised fine-tune the run starts from, and by 10.5 and 9.8 pp over uniform GRPO.
- **Per-test scoring of the environment.** An attempt is scored by how much of the task's own suite its patch repairs, less a penalty for tests it breaks, instead of by a single pass/fail bit. Rollouts that fail differently now score differently, which lifts the share of groups carrying gradient from 54.6% to 58.6%, and the same scores are what make $\mathcal{E}$ fittable at all: AUC 0.963, where a pass/fail label leaves most of the pool sharing one target. The instrumented pool, the sealed evaluations and the analysis code are released with the paper.
- **Per-cycle re-scoring of the whole pool.** The threshold is re-applied each cycle to a recalibrated prediction over the full pool rather than the previous survivors, so a task ruled out under a weaker policy returns once the policy has grown into it, and a task that resolves nothing still contributes the cycles where its rollouts disagree on how much they repair. Wasted draws fall from 32% to 17% over a run, against a flat 30% when the threshold is set once and 80% under uniform sampling; with the reward held fixed, dropping both the recalibration and the retirement costs 6.9 pp of held-out f2p (p=0.005).

2 BACKGROUND AND RELATED WORK

We begin with the optimizer that spends the rollout budget, and with the quantity all three contributions speak to: the fraction of a cycle's rollouts that reach the update.

2.1 GRPO'S EFFECTIVE GRADIENT BUDGET

We use group-relative policy optimization (Shao et al., 2024), the algorithm the arms in this paper all optimize with. Training proceeds in *cycles*. Each cycle samples a batch of tasks, runs K attempts on every task, scores them, and performs one policy update. Throughout this paper the unit of cost is the *rollout*: one complete agent episode.

Take a task t whose K rollouts score $r_1, \ldots, r_K$. GRPO converts those scores into per-rollout advantages by comparing each against the task's own mean: $\hat{A}_i = (r_i - \bar{r}_t)/(\sigma_{r_t} + \epsilon)$, where σ_{r_t} is the standard deviation of the K scores. The consequence that matters here is what happens when the scores are all equal. Then $\sigma_{r_t} = 0$, every advantage is zero, and task t contributes nothing to the update, after costing K full episodes. We call a group of rollouts *usable* when its scores do differ, and a task *live* when it can still produce such a group under the current policy — a property of the task, which any given draw may or may not realise.

So the budget a cycle spends and the budget it actually learns from are not the same number. Let ρ be the fraction of sampled tasks that yield zero gradient. Drawing N tasks per cycle leaves an effective gradient budget of only $(1-\rho)N$. That ρ splits into two parts:

$$\rho = \underbrace{\Pr[t \text{ is dead}]}_{\text{routing can fix this}} + \underbrace{\Pr[\sigma_{r_t} = 0 \mid t \text{ is live}] \cdot \Pr[t \text{ is live}]}_{\text{fractional reward can fix this}} \tag{1}$$

Each part of Equation 1 has its own remedy: a fractional reward shrinks the second, because partial progress becomes visible where all-or-nothing scoring hid it, and routing shrinks the first, because dead tasks are never drawn. They are not independent, though. The reward is also what makes the first term learnable: under all-or-nothing labels most of the pool shares one target, zero, and there is little to rank.

2.2 Related Work

Deciding which prompts to roll out. DAPO's dynamic sampling (Yu et al., 2025) reads informativeness off the rollouts, so a prompt dropped for scoring all-zero has already consumed its K episodes. Difficulty filters (Bae et al., 2026) decide earlier, but against a label fixed before training, which goes stale as the policy moves. GRESO (Zheng et al., 2025) decides earlier still and keeps the decision revisable, predicting a prompt's zero-variance outcome *before* rolling it out from that prompt's own reward history and holding an exploration probability so an excluded prompt can return, which cuts rollouts by $3.35\times$ against dynamic sampling on math reasoning. The signal it reads from is what bounds it: a reward history exists only for prompts already sampled, and carries information only where those rewards varied. Both conditions fail where one rollout is a full agent episode three orders of magnitude dearer than a math prompt, so that the first sampling of a task is itself the expense to be avoided, and where a pass/fail reward leaves the great majority of the pool sharing the single history "zero".

Graded rewards for code RL. Scoring a program by unit-test outcomes rather than a single pass/fail bit goes back to CodeRL (Le et al., 2022), and AceCoder scores by the fraction of synthesized tests a program passes (Zeng et al., 2025). Both operate at function level, where the base success rate is high and a finer reward mainly sharpens a signal that is already present; at repository level the dead-task rate is roughly ten times higher, and the question is whether a finer reward can create a signal rather than sharpen one. SWE-RL escapes binary scoring by a different route, similarity to the reference patch (Wei et al., 2025), which requires an oracle patch for every task and so cannot be applied to a pool that only ships tests. Appendices A and B treat five further threads.

3 Method

PRESAGE has two pieces. Section 3.1 is the predictor and the rule that turns its output into a sampling decision, which is where the rollouts are saved. Section 3.2 is the reward that makes partial progress visible, which is both what the policy optimizes and what supplies the predictor's training labels. Section 3.3 strings them into one cycle. What $\mathcal{E}$ predicts is one number per task, the score the current policy would come away with, rather than a model of what happens inside an episode; Appendix D says why that granularity is the one the decision needs.

3.1 The Pass-Rate Predictor $\mathcal{E}$

Most of the pool returns nothing under the current policy, and no scoring rule repairs that: a task the policy makes no progress on stays at zero however finely we grade it. What the sampler needs is to know which tasks those are *before* paying for their rollouts. That is what $\mathcal{E}$ is for. Given a task's text description it returns a number between 0 and 1, its estimate of the current policy's pass rate on that task. Tasks scoring below 0.1 are treated as dead and dropped from the sampling pool.

$\mathcal{E}$ is a LoRA fine-tune of the same base model with a scalar output head, fitted on $K{=}8$ rollouts over the training pool under the base policy; it separates dead tasks from live ones at AUC 0.963 over that pool (Figure 6). Appendix D gives the architecture, the labelling cost and how it amortizes.

How we use $\mathcal{E}$ for routing. The rule has four steps, run at the start of every cycle. *Threshold:* exclude every task scoring $\mathcal{E} < 0.1$, which we read as predicted dead. *Recalibrate:* fit an isotonic regression from $\mathcal{E}$'s raw outputs to the scores observed so far. The 32B backbone is trained once, before the run; what the loop refits each cycle is only this monotone map, so "refit" throughout this paper means recalibration and never retraining. *Retire:* flag any task that has yielded zero gradient across several cycles as "online dead" and exclude it from then on. *Sample:* draw **uniformly** from whatever survives. PRESAGE is exactly this, plus the per-test reward of Section 3.2.

Two details of the rule matter. Step 1 runs over the *whole* pool each cycle, not over the tasks that survived the last one, so dropping a task is never final: every task is tested again every cycle, and one comes back as soon as recalibration lifts its prediction past the threshold. Step 3 is the only permanent drop, and it takes several all-zero cycles in a row, so one unlucky cycle does not retire a task.

Algorithm 1 One cycle of PRESAGE

Input: training pool $\mathcal{T}$, retired set $\mathcal{D}$, predictor $\mathcal{E}$, isotonic map g, dead threshold τ, batch size N, group size K, policy π_θ
Output: updated π_θ, g, $\mathcal{D}$

1: $\mathcal{A} \leftarrow \{\, t \in \mathcal{T} \setminus \mathcal{D} : g(\mathcal{E}(t)) \geq \tau \,\}$; *// threshold the whole pool, not last cycle's survivors*
2: $\mathcal{B} \leftarrow$ sample N tasks uniformly from $\mathcal{A}$; *// no weighting toward high predictions*
3: **for** each $t \in \mathcal{B}$ **do**
4: run K episodes of π_θ on t, score each patch by Eq. 2 to get $r_1, \ldots, r_K$;
5: $\bar{r}_t \leftarrow \mathrm{mean}(r)$; $\sigma_t \leftarrow \mathrm{std}(r)$;
6: **if** $\sigma_t = 0$ **then** discard the group; *// no gradient, but the K episodes are already paid for*
7: **else** $\hat{A}_i \leftarrow (r_i - \bar{r}_t)/(\sigma_t + \epsilon)$ for $i = 1, \ldots, K$;
8: **end if**
9: **end for**
10: $\theta \leftarrow$ one clipped policy-gradient step on the surviving $\hat{A}_i$;
11: **if** cycle $\bmod 5 = 0$ **then** $g \leftarrow$ isotonic fit to $\{(\mathcal{E}(t), \bar{r}_t)\}$ seen so far; *// recalibration*
12: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\, t \in \mathcal{B} : \bar{r}_t = 0$ in each of its last several draws $\,\}$; *// retirement*
13: **return** π_θ, g, $\mathcal{D}$

Lines 1–2 decide before any episode exists; line 6 is the same judgement after the K episodes have been paid for, which is where a reactive filter operates.

3.2 FRACTIONAL REWARD: EXPOSING PARTIAL PROGRESS

$\mathcal{E}$ is fitted on scores, so the environment has to produce scores that vary. The standard reward asks whether a rollout's patch passes all required tests. We replace it with the fraction of originally failing tests the patch fixes, less a penalty on the originally passing tests it breaks.

That needs a baseline, taken once before training by running every task's suite on the unpatched codebase: the tests that fail there form S_{fail} and those that already pass form S_{pass}, together the "skeleton state." After a rollout produces a patch we apply it, re-run the tests, and score the net change it makes to that state, giving credit for repairs and a penalty for regressions:

$$r \;=\; \underbrace{\frac{|\{t \in S_{\text{fail}} : t \text{ passes after patch}\}|}{|S_{\text{fail}}|}}_{\text{fraction repaired}} \;-\; \lambda\, \underbrace{\frac{|\{t \in S_{\text{pass}} : t \text{ fails after patch}\}|}{|S_{\text{pass}}|}}_{\text{fraction broken}}, \qquad \lambda = 0.5 \quad (2)$$

A patch fixing 3 of 5 failing tests and breaking nothing scores 0.6, where a binary reward scores it 0, the same as a patch fixing 1, and that separation is what GRPO needs to produce any gradient. The penalty term stops the fraction being bought by regressions elsewhere in the suite, and it is inactive when S_{pass} is empty. We set λ=0.5, so that at equal fractional changes, repairs receive twice the weight of regressions. The reward is a function of the submitted patch, computed once per episode; tracking S_{fail} test by test is what makes it graded rather than binary.

This is not potential-based shaping and carries none of its invariance guarantee (Ng et al., 1999): it changes the objective, and in principle the policy could drift toward patches that pass many tests without fully resolving a task. The fractional rate cannot detect that drift, being the quantity a drifting policy would inflate, so Table 1 scores every arm a second way, counting only rollouts that repair *all* of a task's failing tests.

3.3 TRAINING LOOP

Algorithm 1 is one cycle end to end; the two decisions that drop tasks differ only in *when* they act. GRPO runs with PPO clip ϵ=0.2 and learning rate 10^{-5} (Appendix C).

4 EXPERIMENTAL SETUP

Task pool. 531 tasks from 9 Python repositories, taken as generated by SWE-smith (Yang et al., 2025), which produces repository-level bug-fix tasks at scale from existing Python projects. We did not select the repositories: SWE-smith picks pure-Python libraries whose suites run without external

services, which is what makes per-test scoring possible, and we took its output as given. Across the pool 23% of tasks are live under the base 32B policy, and they are not uniformly easy: a fifth sit just above the dead threshold ($r \in [0.1, 0.375)$) and the rest clear it comfortably.

Arms and how we name them. Each configuration is named by what it adds to plain GRPO, not by an index. **GRPO** is the method as it is usually run: a pass/fail reward on a uniform sampler over the whole training pool. Adding the **fractional reward** of Section 3.2 gives one variant (**+reward**); adding the **predictive filter** of Section 3.1 to the pass/fail reward gives another (**+filter**); PRESAGE does both. Those four form the 2×2 of Table 1. **+filter** thresholds $\mathcal{E}$'s raw prediction; PRESAGE also recalibrates and retires (Section 3.1), so the step between them bundles the graded reward with those two routing refinements rather than isolating it. Three further arms hold the reward fixed and vary only the routing rule, testing (iii) (Section 5.4).

Held-out evaluation. 24 tasks are withheld from every stage that can affect a trained model in every arm and both seeds. Although the one-off rollout pass records labels for all 531 tasks, the split is applied before predictor fitting: the held-out 24 labels are excluded from $\mathcal{E}$'s training folds, calibration, and threshold selection, and the tasks are removed from every policy-training draw. Predictor quality figures are out-of-fold over the remaining 507-task training pool (Appendix D).

Each arm is evaluated with $K{=}32$ independent rollouts per task. We report two metrics. The mean fail-to-pass (f2p) rate is the average fraction of originally failing tests that pass after the patch, which is the first term of the reward in Equation 2. The stricter *resolved* rate counts a rollout only when every originally failing test passes, and is not what the reward optimizes. Significance comes from a paired two-level bootstrap ($B{=}200{,}000$) that resamples both the tasks and, within each task, the $K{=}32$ rollouts (Section 5 and Appendix F).

Base policy. SWE-agent-LM-32B (Yang et al., 2025) served with vLLM, chosen because it is a strong open-weights agent model, already supervised fine-tuned for repository-level bug fixing, so we are not improving on a weak starting point. All arms start from this checkpoint with the same LoRA configuration.

5 RESULTS

5.1 MAIN RESULT: CHEAPER BATCHES, MORE TASKS SOLVED

Table 1 is the whole comparison: the 2×2 of reward against sampler, each arm's held-out score, and what the run cost in rollouts. Three findings summarize it, one for each contribution. **Asking $\mathcal{E}$ before paying** lifts the held-out resolved rate from 28.9% to 39.5%: $+10.5$ pp over uniform GRPO and $+12.2$ pp over the SFT checkpoint the run starts from ($p{=}0.002$), with the fractional rate moving the same way ($+9.8$ and $+11.1$ pp; Table 2 gives every interval); holding the reward fixed, it also lifts the share of sampled groups that carry any gradient from 16.1% to 58.6%, so a rollout buys $3.6\times$ the learning signal. **Scoring test by test** lifts that share again, from 54.6% to 58.6% on the same draw, and is what makes the predictor fittable at all: a pass/fail label leaves most of the pool sharing one target, zero, while on graded labels $\mathcal{E}$ separates dead from live at AUC 0.963. **Re-scoring the whole training pool every cycle** keeps the share from decaying as the policy moves: leakage from the live set falls from 32% to 17% over the run, while every arm that never refits stays flat. Together they fix the cost of an update rather than letting it float, so the same budget buys us forty policy updates where a reactive filter gets seven.

Intervals, not point estimates. Every effect in Table 2 carries a 95% interval (Agarwal et al., 2021), from a paired bootstrap ($B{=}200{,}000$) that resamples *both* the 24 tasks and the $K{=}32$ rollouts within each; resampling tasks alone treats each measured score as exact (Appendices F, I).

5.2 THE FILTER CARRIES THE GAIN

First, the filter. Adding it to a pass/fail reward moves the held-out fractional rate from 31.5% to 40.0%, and it is the one comparison that holds on both metrics, on the second seed, and task by task:

Table 1: **Main result.** Same rollout budget for every arm, evaluated at the checkpoint that budget reaches, $K{=}32$ rollouts on each of 24 held-out tasks. *f2p* is the fraction of originally failing tests a patch repairs; *resolved* counts only rollouts that repair all of them. Both Δ columns are against the SFT checkpoint every arm starts from, so each row reads as what forty cycles of that configuration add to it; Table 2 gives the arm-to-arm comparisons as well. DAPO's budget buys 7 updates rather than 40, so its row is read at cycle 6 (Section 5.3); *batch cost* is what a reactive filter would pay to fill one informative batch on that sampler. To ensure robustness, we conducted a second-seed experiment; Appendix J summarizes the repeated 2×2.

	what it uses			f2p		resolved		
Configuration	reward	sampler	updates	rate	Δ	rate	Δ	batch cost
SFT checkpoint *(reference)*	—	—	—	30.2%	—	27.2%	—	—
uniform GRPO	pass/fail	uniform	40	31.5%	+1.3 .51	28.9%	+1.7 .40	6.38×
+ fractional reward	per-test	uniform	40	33.5%	+3.3 .18	29.9%	+2.7 .30	6.20×
+ predictive filter	pass/fail	$\mathcal{E}$-filtered	40	40.0%	+9.9 .001	37.9%	+10.7 .001	1.83×
+ both (PRESAGE)	**per-test**	**$\mathcal{E}$-filtered**	**40**	**41.2%**	**+11.1 .006**	**39.5%**	**+12.2 .002**	**1.71×**
DAPO dynamic sampling	pass/fail	resampled	**7**	31.5%	+1.3 .50	28.0%	+0.8 .73	6.1×

Δ in percentage points; the small figure beside it is the two-sided bootstrap p-value.

Table 2: **Every held-out comparison.** Within a scoring the effect is fixed and only the uncertainty differs between the two bootstraps. The routing decomposition of Section 5.4 is in Table 6.

	f2p (fractional)			resolved (all-or-nothing)	
Comparison	Effect	Task-only p	Two-level p	Effect	Two-level p
PRESAGE vs. SFT	+11.1 pp	0.003	0.006	+12.2 pp	0.002
Filter alone (+filter vs. SFT)	+9.9 pp	0.000	0.001	+10.7 pp	0.001
Uniform GRPO vs. SFT	+1.3 pp	0.317	0.515	+1.7 pp	0.399
PRESAGE vs. GRPO	+9.8 pp	0.001	0.004	+10.5 pp	0.002
Filtering (+filter vs. GRPO)	+8.6 pp	0.000	0.003	+9.0 pp	0.004
Fractional reward (+reward vs. GRPO)	+2.0 pp	0.188	0.356	+1.0 pp	0.660
Reward on top of filter (PRESAGE vs. +filter)	+1.2 pp	0.633	0.675	+1.6 pp	0.602

Figure 2 shows 15 of the 24 improving and 5 worsening, with the largest single contribution 21% of the mean.

Second, uniform GRPO buys nothing over the checkpoint it starts from: forty cycles move it from 30.2% to 31.5%, an interval that straddles zero. The second seed lands in the same place (32.0%), and so does DAPO's dynamic sampling at 31.5% — three runs and two algorithms, none separable from its own starting point. This is Section 2.1's account in its plainest form: a budget spent where five groups in six come back scoring alike has almost nothing to spend, whatever the optimizer does with it. The filtered arms, on the same pool and the same budget, clear the checkpoint by 9.9 and 11.1 pp.

Third, the step from the thresholded filter to full PRESAGE is +1.2 pp, and it adds recalibration and retirement along with the reward. The reward does real work during training, raising the share of sampled groups that carry gradient from 54.6% to 58.6%.

None of this depends on where we stop the run: the ranking at cycle 39 is the one that holds over the whole second half (Appendix H).

5.3 COMPUTE EFFICIENCY: WHAT A USABLE GRADIENT GROUP COSTS

Only 23% of the pool is live, so a uniform draw lands mostly on tasks every rollout fails alike: 16.1% of drawn tasks carry gradient, against 58.6% once $\mathcal{E}$ has filtered (Figure 3a, Table 4).

Against a reactive filter. We trained DAPO's dynamic sampling (Yu et al., 2025) twice: once lent our fractional reward, so the sampler is the only difference from ours, and once exactly as published.

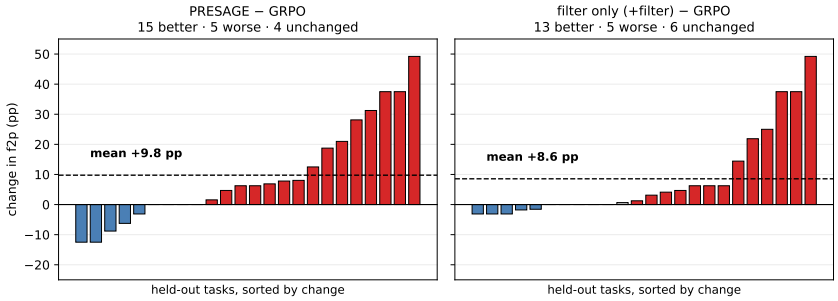

Figure 2: **The gain is broad, not carried by a few tasks.** One bar per held-out task, sorted; the dashed line is the mean Table 1 reports.

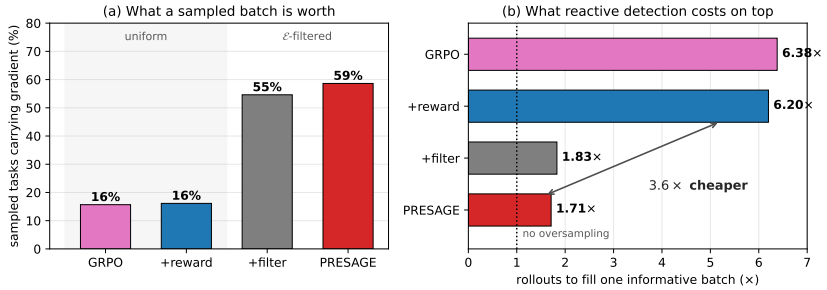

Figure 3: **Where the rollout budget goes.** (a) Share of sampled tasks whose rollouts disagree. (b) Rollouts needed to fill one batch entirely with informative groups when uselessness is detected reactively, layered on each sampler's draw. Both are computed over the tasks each arm actually drew; (b) is a lower bound.

The two schemes hold different quantities fixed. PRESAGE fixes the *cost*: a flat $16\times8 = 128$ rollouts, of which about 9 groups carry gradient, so roughly 14 rollouts per usable group. A reactive filter fixes the *yield* and lets the cost float, drawing waves until 16 groups have disagreed, which here takes 8 to 10 waves, later ones sized to the remaining shortfall rather than to a full batch: 782 rollouts per cycle, or 48.9 per usable group, which is $6.1\times$ a flat batch and matches the $6.38\times$ our own uniform arm's draw predicts (Appendix E). Spending that budget reactively buys no better policy. Evaluated at the checkpoint the shared budget reaches, its seventh update against our fortieth, the published DAPO arm scores 31.5% held-out f2p and 28.0% resolved, indistinguishable from uniform GRPO (-0.0 and -0.9 pp, both $p > 0.6$) and below ours by $+9.8$ and $+11.5$ pp (p=0.011, p=0.002); over its seven updates the training reward is flat. **Our forty updates cost** $5{,}120$ **rollouts; the reactive filter had already spent more than that by its seventh update** ($5{,}474$)**, and matching our forty would cost it about** $31{,}000$**.** Charged in full to a single run, fitting $\mathcal{E}$ costs 4,248 rollouts once, which makes our first-run total 9,368 against the reactive filter's $31{,}280$, still a factor of 3.3 with labelling included; the pass is paid per pool rather than per run (Appendix D).

5.4 DOES THE FILTER STAY CURRENT AS THE POLICY MOVES?

$\mathcal{E}$ predicts the pass rate of the *current* policy, so a router that never refits should grow more wrong about which tasks are dead. The symptom is *dead leakage*: among the tasks the filter did keep, the share that still come back with every rollout scoring alike. Figure 4 tracks it off the training logs. Uniform sampling leaks 80% and stays there, the filter under a pass/fail reward cuts that to a flat 30%, and PRESAGE falls from 32% over the first twenty cycles to 17% over the last twenty: only the arm that refits improves.

Which half of the rule earns that is a separate question. The router recalibrates $\mathcal{E}$'s predictions with isotonic regression *and* retires tasks that keep scoring zero, so we hold the reward fixed and vary only the routing rule across three further arms: one with neither step, one with each alone. Dropping

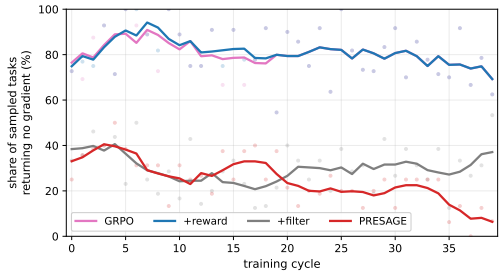

Figure 4: **Does the filter keep working as the policy moves?** One point per cycle: the share of that cycle's draw that came back all-zero. PRESAGE is the only arm that improves.

both costs 6.9 pp (95% CI $[+1.8, +13.1]$, p=0.005), which puts the routing gain in keeping the live set current rather than in the initial threshold. Dropping either one alone costs nothing measurable, 0.2 pp without retirement (p=0.91) and 0.9 pp without recalibration (p=0.61), yet either one alone recovers most of the effect, 6.7 and 6.0 pp over the arm with neither (p=0.036, p=0.024). The intervals on the two single-step drops are wide enough to admit a few points of additivity, so we do not claim the steps are interchangeable, only that neither is individually necessary on the one seed this decomposition was run on. The resolved rate agrees throughout (Appendix F).

6 CONCLUSION

Most tasks in a repository-level pool are out of reach for the current policy. GRPO still pays for K rollouts on each of them and gets nothing back. PRESAGE changes three things. Before a cycle draws its tasks, PRESAGE asks a predictor what the policy would score on each one and skips the tasks it rates near zero. Skipping costs nothing, while a reactive filter has to run the K rollouts first to learn the same fact. It also scores a patch by how many of the task's failing tests it fixes, rather than only by whether it fixes them all, so a rollout that gets part of the way scores differently from one that gets nowhere and the group still teaches something. Those graded scores are what the predictor is trained on; with pass/fail labels there is almost nothing for it to learn from. And it re-scores every task in the pool each cycle, not just the ones that survived the last cycle, so a task that was too hard early on comes back later. The effect is that a group carrying gradient costs 14 rollouts instead of 49, the same budget buys forty policy updates instead of seven, and the held-out resolved rate rises 12.2 points above the checkpoint we started from.

LIMITATIONS

Two bounds on what we claim. *The predictor adapts by recalibration, not retraining.* Its backbone is fitted once; what moves each cycle is only a monotone map from its output to a predicted pass rate (Section 3.1), which tracks the drift we see over forty cycles but could not absorb a policy that changed which *kinds* of task are reachable. *The graded reward changes the objective.* It is not potential-based shaping, so a policy could farm partial credit; the resolved rate is reported beside the fractional one so that such drift would show (Section 3.2).

AI USE STATEMENT

We used Claude Code (Anthropic) to help polish the writing of this paper. All AI-assisted text was reviewed and rewritten by the authors, who take responsibility for the final content.

REPRODUCIBILITY STATEMENT

All experiments use publicly available base models and open-source tools (SWE-agent, vLLM). The task pool, scoring pipeline, and training configuration are described in Appendix C. LoRA checkpoints and evaluation data will be released upon acceptance.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Sanghwan Bae, Jiwoo Hong, Min Young Lee, Hanbyul Kim, Jeongyeon Nam, and Donghyun Kwak. Online difficulty filtering for reasoning oriented reinforcement learning. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 700–719, 2026.

Xiaoyin Chen, Jiarui Lu, Minsu Kim, Dinghuai Zhang, Jian Tang, Alexandre Piché, Nicolas Gontier, Yoshua Bengio, and Ehsan Kamalloo. Self-evolving curriculum for llm reasoning. *arXiv preprint arXiv:2505.14970*, 2025.

DeepSeek-AI. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations (ICLR)*, 2020.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2525–2534, 2018.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 21314–21328, 2022.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning (ICML)*, pp. 278–287, 1999.

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 *International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737, 2025.

Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning (CoRL)*, volume 100 of *Proceedings of Machine Learning Research*, pp. 835–853, 2020.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

Burr Settles. Active learning literature survey. Technical Report 1648, University of Wisconsin–Madison, 2009.

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. *arXiv preprint arXiv:2402.03300*, 2024.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. In *International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 54104–54132, 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 50528–50652, 2024.

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 *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 12023–12040, 2025.

Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts. *arXiv preprint arXiv:2506.02177*, 2025.

A Further Related Work

Section 2.2 compares us against the work our three contributions are measured against, one line per contribution, because each of those lines supplies an arm we could run on this pool. The two families below supply none. They are here because each anticipates part of the idea without reaching the case we study, and marking where they stop is how we bound what we claim is new: the first selects over cheap units, so it never has to decide *when* to decide; the second decides at the right time but assumes a pool that can afford to be reshaped.

Choosing what to spend compute on, before spending it. Deciding which examples deserve computation, ahead of computing on them, is an old idea with several mature forms. Active learning ranks unlabelled examples by expected informativeness and buys labels for the best ones (Settles, 2009). Importance sampling for deep learning skips examples whose gradient contribution is predicted to be small (Katharopoulos & Fleuret, 2018). Prioritized replay revisits stored transitions in proportion to their predicted temporal-difference error, on the reasoning that a transition the model already fits teaches it nothing (Schaul et al., 2016). Our filter is the same idea in kind, and we do not claim the idea. What differs is the price of the unit being chosen over, and the difference is large enough to change which design decisions matter. Those methods select among examples that already exist, where touching one costs a forward pass or a single label. The unit $\mathcal{E}$ selects over does not exist until it has been paid for: obtaining any evidence about one repository-level task means K complete agent episodes, each up to thirty model calls driving a container through edits, installs and a test suite. Roughly three orders of magnitude separate the two prices. That ratio is why the *timing* of the decision is the whole question here, while elsewhere it is a matter of good housekeeping. It also sets the bar $\mathcal{E}$ has to clear. A 32B fine-tune looks extravagant next to a cheap heuristic, but that is the wrong comparison: $\mathcal{E}$ only has to be wrong less often than a filter that pays K full episodes to learn the same fact.

Difficulty-aware training, and why it does not transfer here. Curriculum and difficulty-filtering methods report gains in math reasoning, whether by treating problem categories as the arms of a non-stationary bandit (Chen et al., 2025), by filtering on difficulty online (Bae et al., 2026), or by steering a curriculum with learning progress in classical RL (Portelas et al., 2020). The last of these is the closest relative of $\mathcal{E}$ and the most useful contrast. Teacher algorithms fit a model of *absolute learning progress* over regions of a continuously parameterized environment space and sample in proportion to it, so the prediction is used to shape the whole distribution. $\mathcal{E}$ predicts something narrower, namely whether a group's rewards will agree, which is a prediction about zero variance rather than about progress, and we use it in only one direction, to exclude. Where we part company is not on the mechanism but on whether the pool can afford it. Those pools keep a large majority of problems live, so reshaping the distribution still leaves plenty to train on. PRESAGE leaves a live set confined to two repositories, and the filter alone already concentrates forty cycles of sampling onto 90 distinct tasks. Narrowing that further trades diversity we cannot spare for yield we do not need, which is why we use $\mathcal{E}$ only to exclude. Difficulty-aware selection therefore looks domain-dependent rather than general, which is a reading about the pool and not about the optimizer.

B Adjacent Work

Three further lines share a premise with ours — learn a model so as to spend less time in the environment, choose your data before training on it, train agents on whole repositories — but differ in what the model is of, or in when the choice is made. We record them because a reader who knows them will reach for them here, and none supplies a baseline for Table 1, which is why they sit outside the comparison in Section 2.2 rather than inside it.

RL for code agents. SWE-Gym (Pan et al., 2025) built the first training environment for SWE agents, SWE-RL (Wei et al., 2025) scaled RL training on software evolution data, and DeepSeek-R1 (DeepSeek-AI, 2025) demonstrated large-scale GRPO for reasoning. Those papers contribute environments, data and scale. We hold the environment and the data fixed and ask a different question, which is how a fixed rollout budget should be spent. The routing layer could be dropped into any of these pipelines.

Model-based reinforcement learning. Learning a model of the environment in order to spend less time in it is the motivation behind world models (Ha & Schmidhuber, 2018; Hafner et al., 2020), and we share that motivation. The form is different. Those methods model per-step transitions and use them to plan or to imagine trajectories; $\mathcal{E}$ models the outcome of a whole episode and is used for one binary decision, whether to attempt the task at all.

Data selection for LLM training. Influence-based selection picks a static training mix before fine-tuning (Xia et al., 2024). On-policy training breaks the assumption behind that. A task's value is not a property of the task alone; it belongs to the task paired with the current policy, and it moves as the policy improves. That is why $\mathcal{E}$ is refitted during training instead of once at the start, and why what it predicts is a pass rate under the present policy instead of a fixed difficulty label.

C Implementation Details

On changing the objective. Replacing pass/fail with a pass fraction is not a reparameterization we can appeal to theory about. Potential-based shaping is the construction that provably leaves the optimal policy unchanged (Ng et al., 1999), and a pass fraction is not of that form. A partial fix earns credit that binary scoring discarded, so a policy could in principle be pulled toward patches that satisfy many tests without ever resolving a task. That is why Table 1 reports an all-or-nothing *resolved* rate beside the fractional one. A policy farming partial credit would separate the two, gaining on the fractional rate while the resolved rate stayed flat. What we see instead is the opposite: our arm's gain over plain GRPO is $+9.8$ pp on the fractional rate and $+10.5$ pp on the resolved rate, so the stricter metric, which partial credit cannot reach, registers slightly the larger gain.

Scoring. Each trajectory's patch is applied inside a fresh Docker container. We run the task's full test suite and parse the JUnit XML output to get per-test pass/fail results. Those are compared

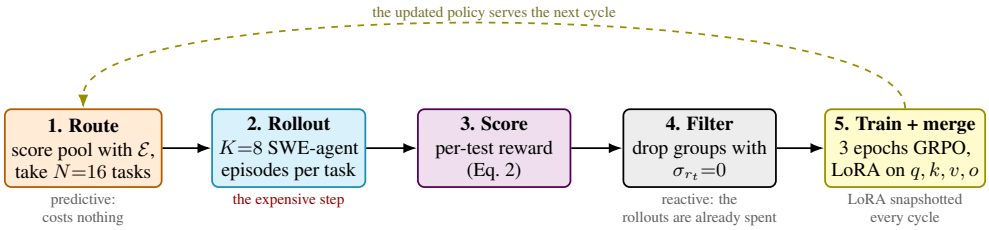

Figure 5: One GRPO cycle. Steps 1 and 4 both remove tasks that will not produce gradient, and the whole argument of this paper is about which of them does it. Step 4 is where prior work acts, after the K episodes of step 2 have been paid for. Step 1 is where $\mathcal{E}$ acts, before anything has been spent.

against the skeleton state (the same suite run with no patch) and scored by Equation 2: the fraction of S_{fail} the patch repairs, less λ=0.5 times the fraction of S_{pass} it breaks.

No KL penalty. We optimize the clipped surrogate without a KL term against a reference policy, following DAPO (Yu et al., 2025), which removes it on the grounds that a policy trained this way is meant to move away from its initialization.

GRPO training. Each cycle draws N=16 tasks and runs K=8 episodes on each, with a per-episode limit of 30 model calls, for 128 trajectories. Low-Rank Adaptation (LoRA) configuration: r=16, α=32, dropout 0.05, applied to q, k, v, o projections. Optimizer: AdamW with weight decay 0.01. Learning rate: 10^{-5}. Batch size: 4 with gradient accumulation. Epochs per cycle: 3. PPO clip: ϵ=0.2. After training the LoRA adapter is merged into the base model, which then serves the next cycle.

D Pass-Rate Predictor $\mathcal{E}$: Training Details

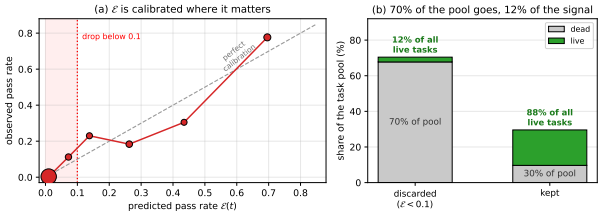

Figure 6: **What $\mathcal{E}$ knows before any rollout is spent.** Predicted pass rate against observed, over the predictor-training pool under the base policy. The predictions order correctly, and the 0.1 cut removes most of the pool while keeping most of the tasks that can still produce gradient.

What $\mathcal{E}$ predicts, and why at that granularity. $\mathcal{E}$ takes a task and returns a single number: the score the current policy is expected to come away with if it attempts that task. It says nothing about what happens inside an episode, deliberately: the decision it serves is made once per cycle and commits K complete episodes at a time, so within-episode dynamics would be machinery the decision never consults. The quantity is also not a property of a task but of the task *paired with* the current policy, which moves every cycle, hence the refitting of Section 3.1. Consulting $\mathcal{E}$ before any rollout exists is what makes the filter predictive: a task it turns down costs nothing.

Architecture and training. $\mathcal{E}$ is a fine-tune of the same base model (SWE-agent-LM-32B), using a Low-Rank Adaptation (LoRA) adapter (r=16, α=32) and a scalar output head. The labelling pass runs K=8 base-policy rollouts on some tasks from repository and records each task's average fractional reward, but the held-out 24 are removed before predictor fitting, cross-validation, calibration, or threshold selection; $\mathcal{E}$ is trained only on the remaining 507 tasks. The pass costs 4,248 rollouts, most of one arm's budget, and is worth being explicit about. It is paid once: the same training-pool

labels serve every arm and both seeds reported here, twelve runs, so the amortized cost is about 350 rollouts per run, roughly 7% of a budget. The resulting predictor separates dead from live tasks at an Area Under the Receiver Operating Characteristic Curve (AUC) of 0.963 over the predictor-training pool: the predictions order correctly, and the cut removes most of the pool while keeping most of the signal (Figure 6).

Head and inputs. The scalar head is a linear layer on the last hidden state of the [EOS] token, squashed to $[0, 1]$ by a sigmoid. LoRA dropout is 0.3. The input is the task's problem statement, truncated to 2048 tokens, so $\mathcal{E}$ sees only what is available before any episode runs.

Cross-validation. We use 5-fold cross-validation over the 507-task training pool, with tasks assigned to folds at random. The held-out 24 tasks are absent from every fold. Every quality figure we report for $\mathcal{E}$ is an out-of-fold prediction, and the routing decisions during training were made from these same out-of-fold predictions. Folds are not stratified by repository; Appendix G reports the leave-one-repository-out numbers, which are the ones to read for transfer to an unseen repository.

Training. MSE loss against the mean reward target. AdamW optimizer, learning rate 2×10^{-5}, 5 epochs, batch size 4. Training takes approximately 10 minutes on 2 GPUs.

Table 3: $\mathcal{E}$ predictor quality, out-of-fold over the 507-task training pool. AUC is for dead-vs-live binary classification at the 0.1 threshold.

Backbone	Spearman ρ	AUC	Val MSE
$\mathcal{E}$ (SWE-agent-LM-32B)	0.597	0.963	0.019

Predictor quality. The 0.1 used to binarize the label is the same cut the router applies, so this AUC measures how well $\mathcal{E}$ reproduces its own operating threshold rather than an independent one. An AUC of 0.963 is enough for the decision the router makes, which is not to rank tasks finely but to separate the ones that cannot produce gradient from the ones that can. We use the same backbone as the policy so that routing adds no second model to host: $\mathcal{E}$ is a LoRA adapter over a checkpoint already resident on the serving GPUs.

E Compute Efficiency Details

Cycles that teach nothing. When every task a cycle draws is silent the batch carries no gradient, the step is skipped, and 128 rollouts leave the policy where it started. Across two seeded runs of forty cycles, uniform sampling produces eleven such cycles and filtering none, or nineteen against zero on the looser count of cycles where *no* group disagreed.

Does the gap widen over a run? We expected that gap to widen over a run, because groups that once disagreed start passing every test, which zeroes their variance exactly as dead ones do, leaving the filter to look further to fill a batch. Our measurements are consistent with that but do not establish it. Across the strict-DAPO arm's seven cycles the cost per usable group is noisy, ranging over 44.0, 40.5, 56.5, 48.5, 48.5, 57.0 and 47.0, averaging 47.0 over the first three cycles and 52.0 over the last two, and it reached our ten-wave resampling cap twice while still filling the batch each time. What the clean run does support is the level rather than the trend: at every cycle we measured, a reactive filter paid between 40 and 57 rollouts for a group we obtained for about 14. Predicting ahead of the spend also has no equivalent failure mode, since $\mathcal{E}$ is refitted against the same drift rather than paying to rediscover it.

Table 5 reports the full compute efficiency breakdown for each arm over 40 cycles.

The two filtered arms draw three to four times as many usable tasks per cycle as the uniform ones, which is the training-side counterpart of Table 1's batch-cost column. PRESAGE sits at 73.0% usable and also gives the best held-out result on this seed.

Table 4: **Cost of filling one batch with informative groups.** A group is degenerate when its rewards are all identical and so contributes zero advantage. Degeneracy is detected from the per-task mean, which misses groups that agreed on an intermediate value, so the reactive costs are lower bounds.

Configuration	Reward	Degenerate groups	Informative	Reactive cost
uniform GRPO	pass/fail	84.3%	15.7%	6.38×
+ fractional reward	per-test	83.9%	16.1%	6.20×
+ predictive filter	pass/fail	45.4%	54.6%	1.83×
+ both (PRESAGE)	**per-test**	**41.4%**	**58.6%**	**1.71×**

Table 5: Compute efficiency by arm. This table uses a looser test than Table 1, counting a task as usable if *any* rollout scored above zero rather than requiring the rollouts to disagree; the ratio between arms is of the same order either way, $4.1\times$ here against $3.6\times$ in Table 4. "Usable fraction" is over the tasks each arm drew across 40 cycles. The ratio column is computed from the raw fractions, not from the rounded percentages, and is normalized to the uniform sampler under the same fractional reward (e.g., ours is $73.0/18.0 = 4.06\times$).

Configuration	Cycles	Usable fraction	vs. uniform
per-test reward			
uniform sampler	40	18.0%	1.0×
$\mathcal{E}$-filtered (PRESAGE)	**40**	**73.0%**	**4.1×**
pass/fail reward			
uniform sampler (GRPO)	40	18.9%	1.0×
$\mathcal{E}$-filtered	40	64.5%	3.6×

F SIGNIFICANCE TESTING: TWO-LEVEL BOOTSTRAP

Every held-out comparison in this paper is tested with a bootstrap that resamples two nested sources of randomness, the held-out tasks and the rollouts within each task.

Concretely, let $r_{a,t,k}$ be the f2p score of the k-th of $K{=}32$ rollouts for arm a on task t. One bootstrap replicate draws $n{=}24$ task indices with replacement, then for each drawn task independently draws K rollout indices with replacement *separately for each arm*, and computes the resulting difference in grand means. Arms are paired at the task level (all arms are evaluated on the same held-out set) but not at the rollout level, since their rollouts are independent samples. We use 200,000 replicates and report percentile intervals. The two-sided p-value is twice the smaller tail mass on either side of zero. The script and its output are released with the code (`compute_stats_two_level.py`, `heldout_stats_two_level.json`).

Table 6: Routing decomposition of Section 5.4, with the reward held fixed. Same two-level bootstrap as Table 2; the task-only column is omitted because these arms are compared only under the two-level resampling.

	f2p		resolved	
Comparison	Effect	p	Effect	p
Both steps (PRESAGE vs. neither)	+6.9 pp	0.005	+7.7 pp	0.002
Online retirement removed	+0.2 pp	0.911	−0.1 pp	0.986
Recalibration removed	+0.9 pp	0.610	+0.7 pp	0.751
Recalibration alone vs. neither	+6.7 pp	0.036	+7.8 pp	0.012
Retirement alone vs. neither	+6.0 pp	0.024	+7.0 pp	0.008

Multiplicity. Table 2 reports twelve comparisons without correcting for multiplicity. The conclusions we draw rest on the two largest effects, filtering and the full method, whose p-values (0.003 and 0.004) survive a Bonferroni threshold of $0.05/12$; the borderline entries in the table should be read as descriptive.

Why intervals at all. We report a 95% confidence interval (CI) alongside every point estimate rather than a p-value alone. With 24 tasks and one training run per arm, a point estimate on its own tells a reader very little about what would happen on a rerun, and interval estimates are the reporting practice this regime calls for (Agarwal et al., 2021).

Why not resample tasks only. Bootstrapping over tasks alone is the more common choice, but it treats each task's mean f2p as exact. It is not exact. It is an average over 32 stochastic rollouts, and on tasks with intermediate pass rates its standard error is substantial. Table 2 runs both analyses. The task-only version is markedly more optimistic, and the gap changes a conclusion: on the fractional-reward effect it reports p=0.19 where the two-level analysis reports p=0.36, and neither interval clears zero.

Two cautions on the interaction. Walking the chain base → GRPO → +reward → PRESAGE telescopes to the end-to-end difference by construction, whatever that difference happens to be, so no chain of this kind can be evidence of super-additivity. The 2×2 interaction is the test that can. And the interaction is significant partly because routing-under-pass/fail does badly, not only because PRESAGE does well.

G CROSS-REPOSITORY GENERALIZATION (LORO)

Zero-shot transfer. We train $\mathcal{E}$ on one repository group and evaluate on another (leave-one-repo-out).

Table 7: LORO zero-shot transfer. AUC is for dead-vs-live classification.

	marsh → spectree		spectree → marsh	
Features	Spearman	AUC	Spearman	AUC
Text only	0.161	0.583	0.585	0.861
Text + code	**0.214**	**0.616**	0.549	0.855

The direction asymmetry (0.583 vs 0.861) reflects the difficulty of predicting a small target set (spectree, 49 tasks) from a larger source. Transfer into the larger repository is usable; transfer into the smaller one is close enough to chance that we would not rely on the filter there without first labelling a few tasks. Only one pair of repositories can be tested this way, since the other seven contain no live tasks and the AUC is undefined on them.

Table 7 gives the zero-shot transfer in both directions.

H READING THE TRAINING SIGNAL ACROSS SAMPLERS

Training f2p is what a practitioner actually watches while a run is in flight, so it is worth being careful about it. Each arm's training mean is taken over the tasks its own sampler drew, and the samplers put very different things in the denominator. A uniform sampler draws mostly dead tasks. Those stay at zero no matter how much the policy improves, which pins the mean near the floor and leaves it moving with whichever tasks a cycle happened to draw. An $\mathcal{E}$-filtered sampler draws from the live pool, where the mean has room to climb. So a rising curve for one arm and a flat curve for another tells us something about the two samplers before it tells us anything about learning.

Put every arm on the same tasks and the comparison becomes readable. We take the tasks that are live under the base policy, keep only the scores each arm recorded on those tasks, and pool them into four equal slices of the rollout budget.

In Table 8 both routing arms rise into the last slice, ours from 0.515 to 0.696 and the filter from 0.561 to 0.582, while both uniform arms peak in the third slice and give the gain back. That ordering is the one the held-out numbers give, where the uniform arms settle at 31.5% and 33.5% and ours reaches 41.2%, so the held-out result is not an artifact of the evaluation set. The routing arms are also the only ones whose training signal is dense enough to read: per unit of budget they contribute four to

Table 8: Training f2p restricted to the tasks that are live under the base policy, pooled into four slices of the rollout budget. Because all arms are scored on the same population, these columns are comparable across rows. Parenthesized counts are the number of (task, cycle) observations behind each cell; the uniform arms contribute few because only about a fifth of what they draw is live.

	cumulative rollouts			
Configuration	0–1,280	1,280–2,560	2,560–3,840	3,840–5,120
uniform GRPO	0.476 (26)	0.504 (23)	0.687 (21)	0.493 (29)
+ fractional reward	0.518 (26)	0.445 (23)	0.662 (21)	0.512 (29)
+ predictive filter	0.489 (96)	0.547 (108)	0.561 (109)	0.582 (108)
+ both (PRESAGE)	0.506 (101)	0.492 (118)	0.515 (118)	**0.696** (118)

five times as many observations on live tasks, which is the efficiency gap of Section 5.3 showing up on the training side.

The uniform arms' third-slice peak is not evidence that they briefly overtake ours. Those cells rest on 21–29 observations against 101–118 for the routing arms, because a uniform draw lands on a live task only about a fifth of the time, so one well-drawn cycle moves them a long way. The asymmetry is built into the setup, not a weakness of this analysis: an arm that spends its budget on dead tasks cannot also buy a sharp estimate of its own progress on live ones.

I WHERE THE EVALUATION NOISE LIVES

The held-out set has 24 tasks and we score each with $K{=}32$ rollouts. The obvious objection is that 24 is small, and the obvious remedy is to raise K. It does not help, and the reason is worth stating because it also says what would help.

Write the held-out mean as an average over tasks of an average over rollouts. Its variance splits in two: τ^2, how much tasks genuinely differ from one another, and σ^2, how much a single task's rollouts scatter around its own rate. The standard error of the reported mean is

$$\mathrm{SE} \;=\; \sqrt{\tau^2/n \;+\; \sigma^2/(nK)}.$$

Only the second term shrinks with K. We estimate both from the held-out per-rollout scores: τ^2 as the between-task variance minus the σ^2/K that leaks into it, σ^2 as the mean within-task variance.

Table 9: Variance decomposition on the $K{=}32$ held-out evaluations, with the 95% confidence half-width the stated design would give. Raising K buys almost nothing because nearly all the variance is between tasks, not within them; the last column shows what doubling the *task count* would buy instead.

	τ^2	σ^2	between-task	95% CI half-width		
Arm	(task)	(rollout)	share	$n{=}24, K{=}32$	$n{=}24, K{=}64$	$n{=}48, K{=}32$
GRPO	0.126	0.104	97%	14.4 pp	14.3 pp	10.2 pp
+ fractional	0.133	0.098	98%	14.8 pp	14.7 pp	10.4 pp
+ filter	0.142	0.072	98%	15.2 pp	15.1 pp	10.7 pp
+ both	0.107	0.127	96%	13.3 pp	13.2 pp	9.4 pp

Table 9 puts between 96% and 98% of the variance between tasks. Doubling K to 64 narrows the interval by about a tenth of a point; sending K to infinity, which is to say measuring each task's rate exactly, would narrow it by a quarter of a point. Doubling the number of tasks would narrow it by four points.

So the honest statement is that our precision is bounded by task count, and we could not raise it. Across the two seeds the uniform arms between them drew 471 of the 507 non-held-out pool tasks, which disqualifies those from serving as held-out afterwards and leaves too few untouched ones to enlarge the set. We looked outside the pool and did not find more: of 39 candidate tasks outside the labelled pool only 16 survived a check that the tests named in each task's `fail_to_pass` list were

the ones actually collected, and those 16 come from a single repository. We report the interval that 24 tasks supports rather than a narrower one we cannot justify.

J SECOND SEED: REPLICATION SUMMARY

This paper places the two halves of the method in different roles, and a replication is the sharpest test of whether that placement is right. The filter is claimed to move the held-out number. The graded reward is claimed to earn its place by making $\mathcal{E}$ fittable (Section 3.1), not by what it adds to the policy's objective once the filter is already there. A second run should therefore reproduce the first claim and is under no obligation to reproduce the second. That is what happens.

We repeat the 2×2 of Table 1 under seed 42: same pool, same held-out set, same $K{=}32$ protocol, arms retrained from scratch. The three comparisons the paper leans on hold their direction. PRESAGE beats uniform GRPO in both runs ($+9.8$ and $+4.4$ pp). The filter's effect holds and is the largest term in both ($+8.6$ and $+12.6$ pp). The reward on its own holds and is small in both ($+2.0$ and $+2.4$ pp).