# Correctness vs Efficiency: Behavior Control with Weight-Space Extrapolation in Code RL

**Kunhao Zheng**[1,2] **Pierre Chambon**[1,3] **Juliette Decugis**[1,3] **Jonas Gehring**[1]
**Taco Cohen**[1] **Benjamin Negrevergne**[2 †] **Gabriel Synnaeve**[1 †]

[1]Meta FAIR [2]Université Paris-Dauphine PSL [3]Inria

[†]Equal supervision.

{kunhao,gab}@meta.com benjamin.negrevergne@dauphine.psl.eu

## Abstract

A model trained with an imperfect reward in RL may learn to exploit the reward with undesired behavior. We study a controlled form of this problem in competitive programming. A program may pass small tests while violating the intended efficiency requirement on larger inputs. We construct a family of rewards that progressively relax the efficiency constraint, making the reward increasingly easy to exploit, and train separate policies. This sweep reveals a *correctness–efficiency frontier*. Relaxing the reward leads to less efficient programs and more resource-limit failures. Stricter rewards suppress these failures but increase correctness failures near the model's capability boundary. This frontier is also reflected in weight space. Interpolating between the resulting models reproduces the same behavioral change. More surprisingly, extrapolation in both directions further amplifies or suppresses it beyond the trained checkpoints. Extensive experiments show that this pattern holds across pure reasoning, tool use, and agentic coding, and at both 32B and 7B scales. This suggests a simple intervention for this form of reward hacking: use a loose reward to amplify the undesired behavior, use the weight-space difference to a strict-reward model as its *steering vector*, and then extrapolate past the strict model to suppress it.

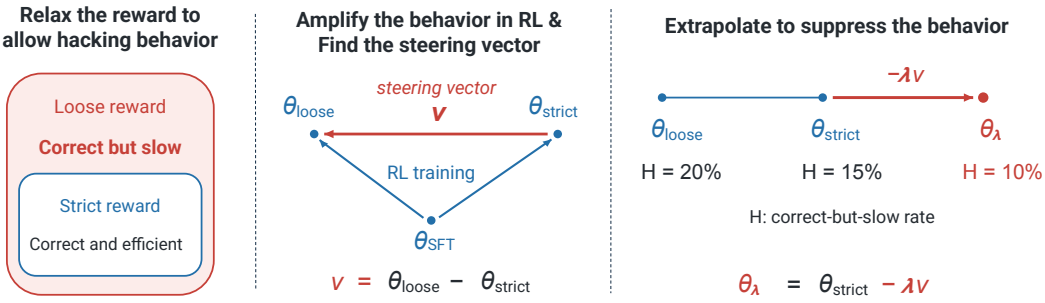



Figure 1: **Controlling reward hacking after RL.** We use a loose reward to amplify the tendency to generate correct but slow programs. We take the weight-space difference between the loose and strict models as a steering vector and extrapolate past the strict model to suppress this behavior.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

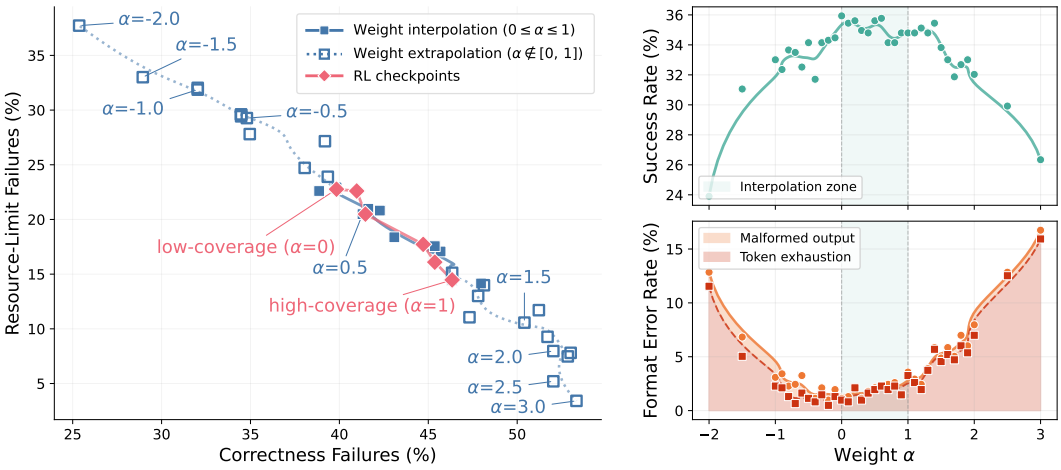



Figure 2: Linear interpolation and extrapolation between two RL checkpoints (trained under tool-use setting) on LCB/hard. **Left**: extrapolated checkpoints continue the same correctness-efficiency frontier traced by interpolated and independently trained RL checkpoints beyond any single RL training run produces. **Right**: solve rate is stable across the interpolation interval and a wide extrapolation regime, and degrades only at extreme $|\alpha|$ where format errors rise sharply through token exhaustion.

## 1 Introduction

The reward used in reinforcement learning (RL) determines which trajectories are reinforced, but may only partially capture the intended behavior, allowing reward hacking (Pan et al., 2022; Skalse et al., 2022; Gao et al., 2023). With verifiable rewards, models may exploit what the verifier checks (Baker et al., 2025; Helff et al., 2026). If relaxing a reward amplifies such undesired behavior, can we use the resulting difference in model weights to suppress it after training?

Competitive programming provides a controlled setting for studying reward exploitation, since generated programs can be tested automatically for both correctness and efficiency. Rewards based on only small tests can favor brute-force programs that larger stress tests reject. We vary reward strictness through nested input-size thresholds and train an independent policy under each reward, holding the SFT initialization and task distribution fixed. We take the weight-space difference between the loose and strict checkpoints, $v = \theta_{\text{loose}} - \theta_{\text{strict}}$, as a *steering vector* for the tendency to generate correct but slow programs. To test whether we can further amplify or suppress this behavior without additional RL, we extrapolate from the both checkpoints along $v$ (Figure 1).

We evaluate all policies on the same full test suite. Stricter rewards suppress resource-limit failures, but on hard problems, this comes at the cost of more correctness failures, leaving the overall solve rate nearly unchanged. This trade-off defines a *correctness-efficiency frontier*. The frontier gives extrapolative weight averaging a concrete target: interpolation tests whether it can be recovered between trained checkpoints, while extrapolation tests whether the same behavioral change continues beyond either endpoint, further amplifying or suppressing resource-limit failures (Figure 2).

We organize the paper around three questions:

1. **What frontier does nested unit-test coverage create?** Nested coverage gives a controlled axis along which stricter rewards suppress resource-limit failures but incur a correctness cost near the model's capability boundary, leaving solve rate nearly unchanged (Section 3.1).
2. **Can extrapolative weight averaging go beyond trained checkpoints?** Interpolation between low- and high-coverage endpoints recovers the frontier traced by independently trained checkpoints, while extrapolation continues the same frontier beyond the RL endpoints. This continuation appears across pure reasoning, tool use, agentic coding, and the model scales we study (Sections 3.2-3.4).
3. **What are the benefits and costs of this control?** Extrapolation beyond the stricter-reward endpoint produces faster correct programs on commonly solved problems without requiring

more tests, but stronger suppression can increase correctness failures. Extreme extrapolation can also introduce format errors. Moving along the frontier changes which problems are solved, providing complementary policies for inference-time scaling (Section 4.2).

## 2 Extrapolative Weight Averaging: Hypothesis and Setup

We study linear weight averaging between two RL checkpoints $\theta_A$ and $\theta_B$,

$$\theta_\alpha \;=\; \alpha\,\theta_B \;+\; (1-\alpha)\,\theta_A, \tag{1}$$

where $\alpha \in [0,1]$ gives interpolation and $\alpha \notin [0,1]$ gives extrapolation. The first empirical check is interpolation: if this arithmetic is meaningful in our setting, interpolated checkpoints should recover sensible behavior between the trained endpoints, in line with prior results (Ramé et al., 2023). The main question is whether the same arithmetic remains meaningful outside the interval spanned by trained checkpoints.

This question is meaningful only when $\alpha$ corresponds to one controlled difference: they should share an initialization, differ along a single interpretable axis, and keep the rest of training fixed. Otherwise the averaging direction mixes several changes at once, and behavior outside the trained interval is hard to attribute to one underlying cause.

Verifier strictness in competitive programming provides such an axis. Relaxed tests can reward programs that work on smaller inputs, whereas stricter tests increasingly demand asymptotically efficient algorithms. This leads to a concrete hypothesis: RL checkpoints trained under progressively stricter verifiers may induce a correctness-efficiency frontier on hard problems. If they do, interpolation should recover that frontier and extrapolation may continue it beyond the checkpoints that RL itself produced. We instantiate this axis as follows.

**Training family.** We instantiate the verifier-strictness sweep separately in three RL settings of increasing interaction capability: **pure reasoning**, which generates code after single-turn reasoning; **tool use**, which interleaves reasoning and Python tool call in multi turns; and **agentic coding**, which uses a sandboxed terminal with file-system access. In each setting, we train independent RL runs from a shared SFT checkpoint on CodeContestsPlus (Wang et al., 2025) ($1\times$ variant), under a nested family of unit-test rewards indexed by the input-length threshold $10^k$:

$$R_k(y) = \mathbf{1}\left[y \text{ passes every test with input length} < 10^k\right], \quad R_\infty(y) = \mathbf{1}[y \text{ passes every test}]. \tag{2}$$

Lower $k$ accepts more submissions; higher $k$ demands asymptotically efficient algorithms. We sweep $k \in \{2, \ldots, 6, \infty\}$ to span the no-pressure to strict-pressure endpoints, with this range chosen by the unit test distributions in CodeContestsPlus (Appendix A.1).

**Evaluation.** We evaluate checkpoints sampled throughout training, including the final checkpoints, on the same held-out benchmark with full test coverage. The benchmark is LiveCodeBench (LCB) v5 (Jain et al., 2025) (20240801-20250201), with easy, medium, and hard split.

**Outcome taxonomy.** We sort each problem's tests by input size and stop at the first failing test. Each code submission receives one of four mutually exclusive labels: **correct** (all tests pass), **resource-limit failure** (a time or memory limit is exceeded after passing the preceding tests), **correctness failure** (a wrong answer or runtime exception within resource limits), or **format error** (unparsable output). These labels describe the observed execution outcome: resource-limit failure does not guarantee algorithmic correctness; a submission may contain a bug masked by the resource limit. To validate our design, Section 3.1 presents a re-evaluation protocol and tests whether this ambiguity affects the observed trade-off.

**Models.** We study at both 32B and 7B scale, in both cases we start from a pre-RL SFT checkpoint to ensure that we study the effect of RL itself rather than layering RL on an already RL-trained model. For 32B model, we use CWM-SFT (FAIR CodeGen team, 2025). Specially for tool-use environment, we conduct two iterations of rejection sampling to teach the tool-call format and use the resulting checkpoint as the starting policy of our experiments (Appendix A.5). For 7B model, since Qwen-2.5 7B (Qwen team, 2025) does not provide a public pre-RL SFT reasoning checkpoint, we fine-tune its base model ourselves on the public OpenCodeReasoning-2 (Ahmad et al., 2025) and

OpenMathReasoning (Moshkov et al., 2025), giving us full control and transparency over the SFT datamix. Training details are in Appendices A.2 and A.5.

## 3 Extrapolation Extends the Correctness-Efficiency Frontier

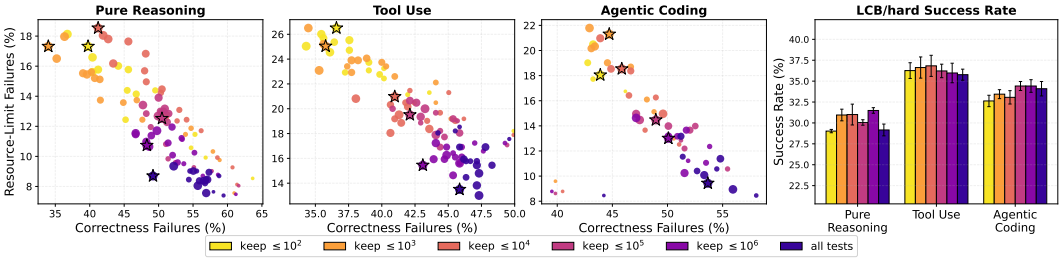



Figure 3: Correctness-efficiency frontier on LCB/hard across three inference settings. Each scatter panel shows checkpoints sampled throughout RL training under different coverage thresholds. Smaller points are earlier checkpoints, larger points are later checkpoints, and stars mark final checkpoints. In every setting, the training trajectories follow the same trade-off between optimization and correctness failures, defining the frontier we extrapolate along. **Right**: bars averaging the last four checkpoints at each threshold confirm that solve rate remains nearly unchanged across conditions in late training.

We first establish the prerequisite empirical structure for the paper: varying verifier strictness induces a frontier between resource-limit failures and correctness failures on hard problems (Section 3.1). We then test the hypothesis: linear interpolation recovers this frontier (Section 3.2), and linear extrapolation continues it beyond what any single RL run reaches (Section 3.3). We close by showing a strong generalization of both the frontier and its extension by extrapolative weight averaging across RL environment settings and model scales (Section 3.4).

### 3.1 A Correctness-Efficiency Frontier from Verifier-Strictness Sweeps

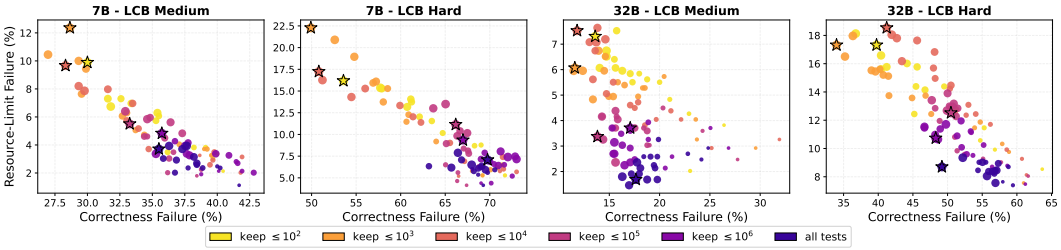



Figure 4: Pure-reasoning evaluation at 7B and 32B on medium and hard splits. The trade-off appears only on hard problems at 32B, but already on medium problems at 7B, showing that the frontier tracks model capability rather than a fixed difficulty label.

On held-out LCB/hard problems, stricter verifier coverage does not steadily raise solve rate. Instead, it organizes independently trained RL checkpoints into a narrow frontier: relaxed coverage produces more resource-limit failures, stricter coverage produces more correctness failures, and late-training solve rates remain close across thresholds. Figure 3 shows this pattern throughout training across pure reasoning, tool use, and agentic coding. The trade-off also persists across random seeds at both model scales (Appendix B.2).

Rather than attaching to one fixed benchmark label, the frontier appears wherever problems approach the model's capability limit. Figure 4 shows this shift across scales: at 32B the frontier is clear on the hard split, whereas at 7B the same structure already appears on the medium split. In both cases, the frontier emerges once problem difficulty is high enough that correctness and efficiency genuinely compete.

**Error taxonomy validation with extended re-classification.** Resource-limit failures can mask correctness bug. We re-evaluate these submissions to distinguish correct but slow programs from

programs with hidden bugs. With a 100× extended time budget, 16%-40% pass the full test suite. For the ones which still hit the resource limits, we use an LLM judge with majority voting to assess correctness and manual review the cases of disagreements (Appendix B.1). Across both stages, 5% to 32% of the original resource-limit failures are reclassified as correctness failures. The trade-off and the ordering of coverage levels persist (Figure 5). Thus, the observed frontier is robust to this re-classification of the failure labels.

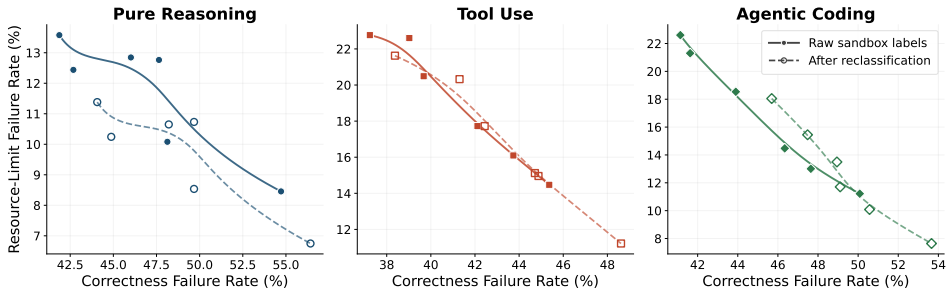



Figure 5: Trade-off curves under raw sandbox labels (solid) versus reclassified labels (dashed) for pure reasoning (left), tool use (center), and agentic coding (right). Each point is one training checkpoint under a fixed test-coverage threshold. The two labeling schemes produce nearly overlapping frontiers, validating the use of raw labels solely from execution.

Taken together, these results give the rest of the section its starting geometry: verifier strictness creates an ordered family of checkpoints along a single behavioral axis. The next two subsections show that weight averaging can follow that axis directly, first by recovering the frontier through interpolation and then by extending it beyond the checkpoints reached by RL.

### 3.2 Linear interpolation recovers the frontier

Interpolation between a lower-coverage endpoint $\theta_{\text{low}}$ and a higher-coverage endpoint $\theta_{\text{high}}$, namely $\theta_\alpha = (1-\alpha)\theta_{\text{low}} + \alpha\theta_{\text{high}}$ with $\alpha \in [0,1]$, recovers the frontier traced by the independently trained verifier sweep. As $\alpha$ moves from 0 to 1, solve rate stays nearly flat while resource-limit failures give way to correctness failures in a smooth progression (Figure 2). Two checkpoints therefore reconstruct the same trade-off that otherwise required a full family of RL runs. A single interpolation direction in weight space is sufficient to traverse that frontier.

Prior work shows that interpolation can navigate a multi-objective trade-off between checkpoints trained under different rewards (Ramé et al., 2023). In our case, the trade-off is not designed by specifying separate objectives since the rewards are nested and monotone improvement is plausible in theory. Empirically, the verifier-strictness sweep traces out a frontier across separate RL runs, and interpolation recovers that same frontier.

### 3.3 Linear extrapolation extends the frontier beyond what RL reaches

Extrapolation continues the same frontier beyond the checkpoints reached by RL (Figure 2). For $\alpha > 1$, the extrapolated checkpoints move past the higher-coverage endpoint towards fewer resource-limit failures and more correctness failures than RL itself reaches. For $\alpha < 0$, they continue the same trade-off in the opposite direction beyond the lower-coverage endpoint. Extrapolative weight averaging therefore does more than recover the verifier-strictness sweep. It extends that sweep to new checkpoints that no single RL run produces.

This continuation remains stable over a broad range of extrapolation coefficients. For $\alpha \in [-0.5, 1.5]$, solve rate stays nearly flat even though this interval already extends beyond the two trained endpoints, while resource-limit failures fall and correctness failures rise along the same pattern seen inside the trained interval (Figure 6). At more extreme coefficients, the same pattern remains visible, but solve rate drops because format errors rise sharply, mostly through token exhaustion. Extrapolative weight averaging therefore extends the verifier-strictness sweep beyond the range reached by RL, until model generations collapse.

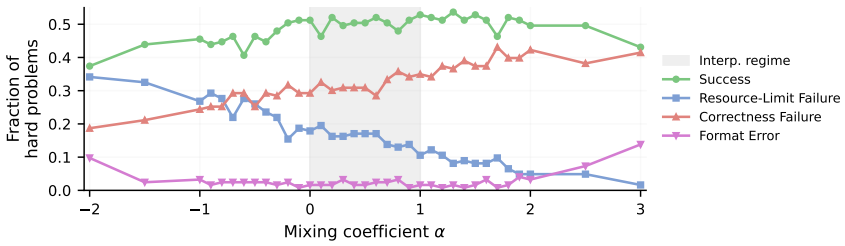



Figure 6: Outcome composition along the interpolation and extrapolation axis on LCB/hard for RL checkpoints (trained under tool-use setting). The interpolation coefficient $\alpha$ moves from the lower-coverage endpoint $\theta_{\text{low}}$ at $\alpha = 0$ to the higher-coverage endpoint $\theta_{\text{high}}$ at $\alpha = 1$. Resource-limit failures fall as correctness failures rise, while solve rate remains nearly flat across the interpolation regime and a broad extrapolation range. Format errors become substantial only at extreme $|\alpha|$. The grey band marks the interpolation interval $[0, 1]$.

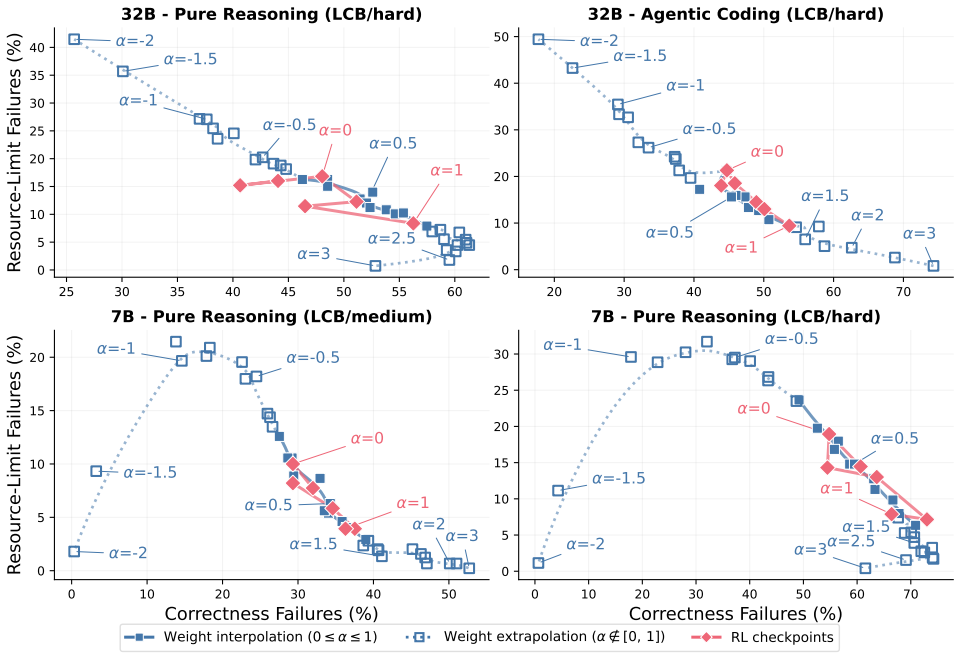



Figure 7: Extrapolative continuation across inference settings and model scale. Figure 2 provides the 32B tool-use case. We show the two remaining 32B settings on LCB/hard, pure reasoning and agentic coding, and 7B pure reasoning on LCB/medium and LCB/hard. In each plot, interpolation recovers the correctness-efficiency frontier inside the trained interval, and extrapolation continues the same trade-off outside it, while the stable extrapolation range is narrower for 7B.

### 3.4 Extrapolative weight averaging generalizes across inference settings and model scales

Figure 7 extends the 32B tool-use case in Figure 2 to the two remaining 32B inference settings. In both pure-reasoning and agentic coding, interpolation recovers the correctness-efficiency frontier inside the trained interval, and extrapolation continues the same frontier beyond it. Agentic coding shows the cleanest continuation. Pure reasoning follows the same trend, but with visibly more scatter and earlier degradation at extreme coefficients.

The scale comparison follows the same capability-relative pattern as Section 3.1. For the 7B model, Figure 7 shows both LCB/medium and LCB/hard, which shows the same frontier and extrapolative continuation. The stable range for extrapolation is narrower at 7B, especially on the low-$\alpha$ side where format errors dominate once $\alpha < -1$ (Appendix B.3.1). Extrapolative weight averaging therefore generalizes across inference settings and across model scale once problem difficulty is matched to model capability, while the width of the stable regime is setting dependent.

# 4 Analysis and Perspectives

We next examine how extrapolation changes the programs the model generates. We first measure the runtime of correct programs and show that it makes them faster (Section 4.1). We also show that moving along the axis changes which hard problems are solved, which makes ensembling extrapolated checkpoints useful for inference-time scaling (Section 4.2).

## 4.1 Extrapolation makes correct programs faster

We next ask whether extrapolation also produces faster correct programs. We compare the model trained with the stricter reward against the model obtained by extrapolating beyond it ($\alpha = 2$). For each problem solved by both models, we measure the runtime on the 5 largest test cases, and take the median of multiple repetition after warmup.

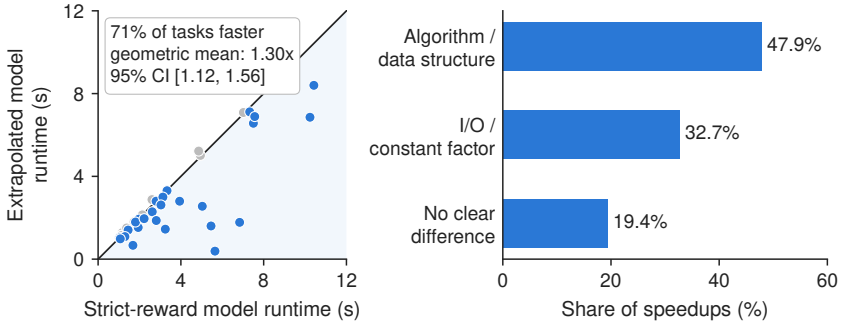



Figure 8: **Left:** Correct-program runtime on commonly solved LCB/v5 problems where the strict-reward model produce solutions whose runtime >1s. Points below the diagonal mean that extrapolated model ($\alpha = 2$) produces faster solutions. **Right:** Code differences identified by an LLM judge among pairwise runtime speedups for the extrapolated model.

Across all 141 problems solved by both models on LCB/v5 medium and hard split, extrapolation yields a $1.14\times$ geometric-mean speedup (95% CI $[1.04, 1.27]$) compared to the strict-reward model. The gains are larger on problems in which solutions take longer time to run: for these problems where strict-reward model's solutions take more than 1s to finish, extrapolation is faster on $71\%$ of them and yields a $1.30\times$ speedup (Figure 8 **Left**).

To examine what changes in the code, we compare pairs of correct programs from problems where extrapolation lowers runtime meaningfully (relative runtime delta > 5%). We use LLM judge to assess the paired solutions. The judge attributes 47.9% of improvement to algorithm or data-structure improvements and 32.7% to I/O or constant-factor improvements (Figure 8 **Right**).

## 4.2 Per-problem turnover and inference-time ensembles

The nearly flat solve rate along the extrapolation axis conceals substantial per-problem turnover. On LCB/hard, $24.4\%$ of problems are solved at every $\alpha$, $28.5\%$ are never solved, and the remaining $47.2\%$ are solved only on part of the axis (Figure 9). This middle band is the source of the solved-set diversity that ensembles can exploit. Figure 10 shows that the solve-set turnover is gradual: adjacent values of $\alpha$ share more than $80\%$ of solved problems, and the overlap falls steadily to about $55\%$ at the extremes.

This turnover gives the frontier axis a direct use at inference time: we can spread a fixed sampling budget across checkpoints along the axis and aggregate their samples. We compare the best single checkpoint with five-checkpoint pools drawn from either the interpolation only or the larger interpolation-plus-extrapolation range. At

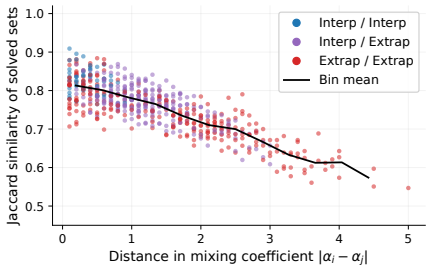



Figure 10: Jaccard similarity of solved sets as a function of $|\alpha_i - \alpha_j|$. Overlap falls steadily with distance in $\alpha$ for interpolation, extrapolation, and mixed pairs.

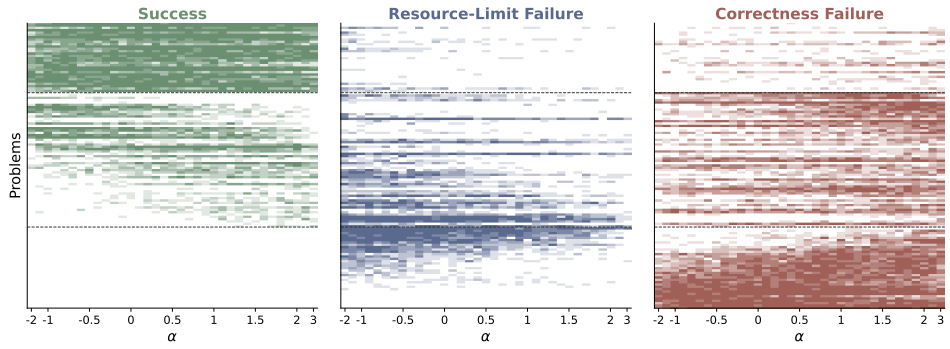



Figure 9: Per-problem outcomes across weight-averaging coefficients $\alpha$, measured with 10 samples per problem and sorted into always-solved (top), partially solved (middle), and never-solved (bottom). The middle group (47.2%) accounts for the changes in which problems are solved. Failure types also continue to trade off among never-solved problems throughout the sweep.

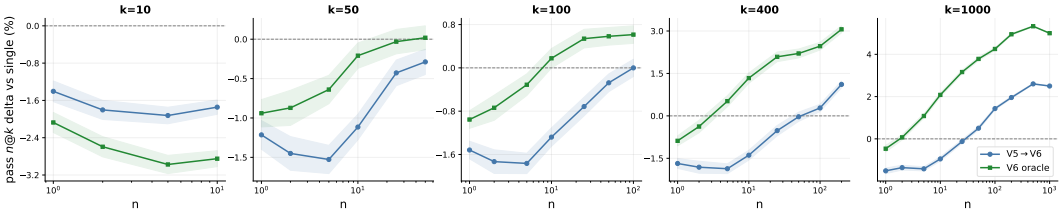



Figure 12: Transfer of ensemble selection from LCB/v5 to LCB/v6 on the hard split. Pools are selected on LCB/v5 (blue) or LCB/v6 (oracle, green). Both are evaluated on LCB/v6, with pass $n@k$ gains measured relative to the best single checkpoint selected on LCB/v6.

the same total sampling budget, the selected pool that includes extrapolated checkpoints improves pass@250 by 3.3% on LCB/v5 over the best single checkpoint (Figure 11). Since the pool is selected and evaluated on the same split, we treat this gain as an oracle-selected reference.

We apply public-test filtering (Li et al., 2022; Gehring et al., 2025; Zheng et al., 2025b) and report pass $n@k$, the success rate when selecting up to $n$ submissions from $k$ candidates. Public-test filtering can carry part of the large-candidate-budget diversity gain to smaller submission budgets (Appendix C.3).

A natural concern is that checkpoint-pool selection is tuned to LCB/v5. To test transfer, for each candidate budget $k$, we search for the best five-checkpoint pool on LCB/v5 and evaluate it on the newer LCB/v6 (20250201-20250501) hard split (Figure 12). The transferred pool becomes beneficial once both $k$ and $n$ are large enough. A pool selected directly on LCB/v6 still performs better, so extrapolation provides transferable diversity while better pool selection leads to potentially better performance.

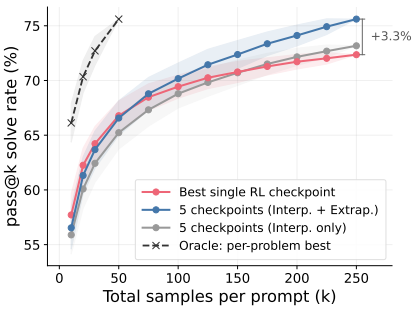



Figure 11: Matched-budget LCB/hard pass@$k$ of the best single checkpoint, the best interpolation-and-extrapolation ensemble, and the best interpolation-only ensemble.

## 5 Related Work

**RL for code reasoning and test quality.** Reinforcement learning with verifiable rewards has become a standard post-training recipe for code reasoning, from single-turn generation to tool-integrated and agentic settings (Guo et al., 2025; Liu et al., 2025; Xue et al., 2025; Minimax team, 2025; FAIR CodeGen team, 2025; Kimi team, 2026). Benchmarks such as CodeContests (Li et al., 2022), APPS (Hendrycks et al., 2021), and TACO (Li et al., 2023) established code generation under executable tests as a central evaluation setting. Later work sharpened the role of test quality, including

EvalPlus (Liu et al., 2023), LiveCodeBench (Jain et al., 2025), and CodeContestsPlus (Wang et al., 2025). At inference time, a parallel line studies how additional inference compute improves code generation through larger candidate pools, public-test filtering, and iterative candidate refinement (Li et al., 2022; Zheng et al., 2025b; Li et al., 2025a). Other methods use simulated execution for self-fixing (Maimon et al., 2026) or trajectory reuse in agentic coding (Kim et al., 2026). During training, other work moves beyond a single outcome reward by adding richer execution-level or process-level signals, through execution feedback (Gehring et al., 2025), process-level rewards (Li et al., 2025b; Dai et al., 2025), and self-verification with generated tests (Jin et al., 2025).

**Linear mode connectivity and model merging.** Mode-connectivity results identify low-loss paths between independently trained networks (Garipov et al., 2018), while stochastic weight averaging combines checkpoints along a single optimization trajectory to improve generalization (Izmailov et al., 2018). In RL, Nikishin et al. (2018) use the same idea to stabilize training and reduce policy forgetting. Frankle et al. (2020) show that vision networks trained from the same initialization can often be connected by low-loss linear paths. Related approaches include model souping (Wortsman et al., 2022), task vector editing (Ilharco et al., 2023), and layer-wise variants (Yang et al., 2024; Wang et al., 2024; Lin et al., 2024). Recently, Neural Thickets (Gan and Isola, 2026) provides a complementary view of weight-space diversity through random perturbation and ensembling. Closest to our interpolation result is Rewarded Soups (Ramé et al., 2023), which shows that interpolating checkpoints trained under explicitly different rewards traces a multi-objective Pareto front. Follow-up work extends this idea to reward models and policy models (Ramé et al., 2024; Ramé et al., 2024). Our setting differs in that the checkpoints are trained under a nested family of stricter and looser verification regimes within the same binary pass-fail signal, rather than under explicitly different reward objectives.

**Weight-space directions and extrapolation.** Task arithmetic treats differences in model weights as editing directions (Ilharco et al., 2023). Activation-steering methods instead construct directions in hidden representations and intervene during inference (Rimsky et al., 2024). For extrapolation, the nearest precedents are ExPO (Zheng et al., 2025a) and AlphaRL (Cai et al., 2026), which extrapolate along a single post-training trajectory (SFT-to-DPO or early-to-late RL). In our setting, by contrast, the extrapolation is *between two RL checkpoints*, and it continues the same frontier beyond what either training run reaches.

**Correctness and efficiency in code.** Recent work increasingly treats efficiency as an essential dimension of code quality alongside correctness. Mercury (Du et al., 2024) introduces efficiency-stratified evaluation to distinguish correct-but-slow solutions from efficient ones. ECCO (Waghjale et al., 2024) studies the tension between improving efficiency and preserving correctness in model-generated code. Related benchmarks extend this concern to algorithmic efficiency (Huang et al., 2024; Chambon et al., 2025), GPU kernels, and software engineering workloads (Ouyang et al., 2025; Ma et al., 2025). Afterburner (Du et al., 2025) uses a self-improving loop with RL to improve code efficiency. These works study settings where efficiency is made explicit, either in the benchmark or in the objective, leaving open what standard unit-test RL does when correctness and efficiency are folded into the same pass-fail supervision.

## 6 Discussion and Limitations

**Nested rewards need not yield nested policies.** In our setting, the rewards are nested by construction. However, empirically, the learned policies during on-policy RL are not ordered by dominance. Their weight difference provides a direction for controlling behavior. Even when the rewards differ along a controlled axis, the steering direction need not affect only one behavior. In our case, it changes both efficiency and correctness. Appendices E.2 and E.3 test whether reward shaping and failure-conditioned online fixing can shift this frontier during training.

**Finer-grained weight-space analysis of behavioral control.** Appendix D shows that the weight-matrix differences are high-rank, with squared weight changes distributed nearly uniformly across most layers. However, layer stitching (Appendix D.3) suggests that the behavioral effects are concentrated in the middle layers. In our experiments, swapping the middle half of the layers

reproduces the full shift of behavior (Figure 30). This motivates testing whether constraining RL updates, for example with LoRA or frozen layers, can produce a more localized direction.

**Beyond a single verifier frontier.** As preliminary evidence, Appendix E.1 averages two checkpoints optimized for different sampling behavior: one with stronger pass@1 and one with stronger pass@$k$. Interpolation traces a smooth Precision-Recall curve, and yields ensembles that outperform single-checkpoint sampling at larger budgets. Extrapolation adds only modest gains in this setting. A useful interpolation direction therefore does not necessarily support equally useful extrapolation. For other behaviors, the key question is how to design rewards that bring out the behavior we want to control while limiting changes in other capabilities.

**Limitations.** Our experiments are confined to competitive programming domain, but they are not confined to a single RL environment. Our experiments span from single-turn generation to richer agentic training, suggesting that the effect is tied to verifier-induced optimization pressure rather than to one specific inference setting. Applying the same analysis to other domains such as SWE tasks and kernel generation requires first identifying a comparable ordered axis, such as test-suite strength or performance tolerance. The input-length thresholds are a coarse proxy for test-case scale and the optimization stress it induces, while our experiments show that it is a reliable proxy that consistently exposes the trade-off. Our ensemble comparisons match code generation count rather than token or wall-clock cost.

## 7 Conclusion

Our results show that reward design can be used to construct weight-space directions for behavior control. In code RL, two runs from a shared initialization under different reward strictness provide a direction that can amplify or suppress inefficient coding behavior after RL training. Extending this approach to other behaviors requires designing rewards that reliably induce the behavioral difference we want to control. The core challenge is to make that difference pronounced enough to support weight-space control while limiting changes in other capabilities.

## Acknowledgement

We especially thank Loïc Cabannes for his "Cool! Then what about extrapolation?" question, a question we could not resist testing; Matteo Pagliardini for his "Hey, I have an interpolation script!" moment and for pointing us to the task-vector literature; and Fabian Gloeckle for his enthusiastic "Yes, the result is so cool, I like this!" when we first showed him our interpolation and extrapolation results. We thank Ori Yoran, Gautier Izacard, Keyur Muzumdar, David Zhang, Rémi Munos, François Fleuret, Wassim Bouaziz, Krunoslav Lehman Pavasovic, Theophane Vallaeys and João Maria Janeiro for many fruitful discussions. We also thank all members of the FAIR CodeGen team for their support throughout the project.

## References


Wasi Uddin Ahmad, Somshubra Majumdar, Aleksander Ficek, Sean Narenthiran, Mehrzad Samadi, Jocelyn Huang, Siddhartha Jain, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning-ii: A simple test time scaling approach via self-critique, 2025. URL https://arxiv.org/abs/2507.09075.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *CoRR*, abs/2503.11926, 2025. doi: 10.48550/ARXIV.2503.11926. URL https://doi.org/10.48550/arXiv.2503.11926.

Yuchen Cai, Ding Cao, Xin Xu, Zijun Yao, Yuqing Huang, Zhenyu Tan, Benyi Zhang, Guangzhong Sun, Guiquan Liu, and Junfeng Fang. On predictability of reinforcement learning dynamics for large language models, 2026. URL https://arxiv.org/abs/2510.00553.

Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. Bigo(bench) – can llms generate code with controlled time and space complexity?, 2025. URL https://arxiv.org/abs/2503.15242.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. Process supervision-guided policy optimization for code generation, 2025. URL https://arxiv.org/abs/2410.17621.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/1df1df43b58845650b8dada00fca9772-Abstract-Datasets_and_Benchmarks_Track.html.

Mingzhe Du, Luu Anh Tuan, Yue Liu, Yuhao Qing, Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and See kiong Ng. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization, 2025. URL https://arxiv.org/abs/2505.23387.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, Proceedings of Machine Learning Research, pages 3259–3269. PMLR, 2020. URL http://proceedings.mlr.press/v119/frankle20a.html.

Yulu Gan and Phillip Isola. Neural thickets: Diverse task experts are dense around pretrained weights, 2026. URL https://arxiv.org/abs/2603.12228.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10835–10866. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P. Vetrov, and Andrew Gordon Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems 31*, pages 8803–8812, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/be3087e74e9100d4bc4c6268cdbe8456-Abstract.html.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: grounding code llms in execution feedback with reinforcement learning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, Sept 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Lukas Helff, Quentin Delfosse, David Steinmann, Ruben Härle, Hikaru Shindo, Patrick Schramowski, Wolfgang Stammer, Kristian Kersting, and Felix Friedrich. LLMs gaming verifiers: RLVR can lead to reward hacking. *CoRR*, abs/2604.15149, 2026. doi: 10.48550/ARXIV.2604.15149. URL https://doi.org/10.48550/arXiv.2604.15149.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. Effibench: Benchmarking the efficiency of automatically generated code. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/

15807b6e09d691fe5e96cdecde6d7b80-Abstract-Datasets_and_Benchmarks_Track.html.

Gabriel Ilharco, Marco Túlio Ribeiro, Mitchell Wortsman, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=6t0Kwf8-jrj.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry P. Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 876–885. AUAI Press, 2018. URL http://auai.org/uai2018/proceedings/papers/313.pdf.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification, 2025. URL https://arxiv.org/abs/2506.11442.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srinivasan Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Gabriel Synnaeve, Ruslan Salakhutdinov, and Anirudh Goyal. Scaling test-time compute for agentic coding, 2026. URL https://arxiv.org/abs/2604.16529.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026. URL https://arxiv.org/abs/2507.20534.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Findings of*

*the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 15964–15978. Association for Computational Linguistics, 2025a. URL https://aclanthology.org/2025.findings-emnlp.865/.

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

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Yong Lin, Hangyu Lin, Wei Xiong, Shizhe Diao, Jianmeng Liu, Jipeng Zhang, Rui Pan, Haoxiang Wang, Wenbin Hu, Hanning Zhang, Hanze Dong, Renjie Pi, Han Zhao, Nan Jiang, Heng Ji, Yuan Yao, and Tong Zhang. Mitigating the alignment tax of RLHF. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 580–606. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.35. URL https://doi.org/10.18653/v1/2024.emnlp-main.35.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

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

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads?, 2025. URL https://arxiv.org/abs/2511.06090.

Gallil Maimon, Ori Yoran, Felix Kreuk, Michael Hassid, Gal Cohen, Pierre Chambon, and Yossi Adi. Self-execution simulation improves coding models, 2026. URL https://arxiv.org/abs/2604.03253.

MiniMax, :, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Enwei Jiao, Haigang Zhou, Haimo Zhang, Han Ding, Haohai Sun, Haoyu Feng, Huaiguang Cai, Haichao Zhu, Jian Sun, Jiaqi Zhuang, Jiaren Cai, Jiayuan Song, Jin Zhu, Jingyang Li, Jinhao Tian, Jinli Liu, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kaiyi Feng, Ke Yang, Kecheng Xiao, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Li, Lin Zheng, Linge Du, Lingyu Yang, Lunbin Zeng, Minghui Yu, Mingliang Tao, Mingyuan Chi, Mozhi Zhang, Mujie Lin, Nan Hu, Nongyu Di, Peng Gao, Pengfei Li, Pengyu Zhao, Qibing Ren, Qidi Xu, Qile Li, Qin Wang, Rong Tian, Ruitao Leng, Shaoxiang Chen, Shaoyu Chen, Shengmin Shi, Shitong Weng, Shuchang Guan, Shuqi Yu, Sichen Li, Songquan Zhu, Tengfei Li, Tianchi Cai, Tianrun Liang, Weiyu

Cheng, Weize Kong, Wenkai Li, Xiancai Chen, Xiangjun Song, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xinzhu Hou, Xuan Lu, Xun Zou, Xuyang Shen, Yan Gong, Yan Ma, Yang Wang, Yiqi Shi, Yiran Zhong, Yonghong Duan, Yongxiang Fu, Yongyi Hu, Yu Gao, Yuanxiang Fan, Yufeng Yang, Yuhao Li, Yulin Hu, Yunan Huang, Yunji Li, Yunzhi Xu, Yuxin Mao, Yuxuan Shi, Yuze Wenren, Zehan Li, Zelin Li, Zhanxu Tian, Zhengmao Zhu, Zhenhua Fan, Zhenzhen Wu, Zhichao Xu, Zhihang Yu, Zhiheng Lyu, Zhuo Jiang, Zibo Gao, Zijia Wu, Zijian Song, and Zijun Sun. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://arxiv.org/abs/2506.13585.

Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset, 2025. URL https://arxiv.org/abs/2504.16891.

Evgenii Nikishin, Pavel Izmailov, Ben Athiwaratkun, Dmitrii Podoprikhin, Timur Garipov, Pavel Shvechikov, Dmitry Vetrov, and Andrew Gordon Wilson. Improving stability in deep reinforcement learning with weight averaging. In *UAI 2018 Workshop on Uncertainty in Deep Learning*, 2018. URL https://www.gatsby.ucl.ac.uk/~balaji/udl-camera-ready/UDL-24.pdf.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient GPU kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=JYtwGwIL7ye.

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.

Alexandre Ramé, Guillaume Couairon, Corentin Dancette, Jean-Baptiste Gaya, Mustafa Shukor, Laure Soulier, and Matthieu Cord. Rewarded soups: towards pareto-optimal alignment by interpolating weights fine-tuned on diverse rewards. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/e12a3b98b67e8395f639fde4c2b03168-Abstract-Conference.html.

Alexandre Ramé, Nino Vieillard, Léonard Hussenot, Robert Dadashi, Geoffrey Cideron, Olivier Bachem, and Johan Ferret. WARM: on the benefits of weight averaged reward models. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 42048–42073. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/rame24a.html.

Alexandre Ramé, Johan Ferret, Nino Vieillard, Robert Dadashi, Léonard Hussenot, Pierre-Louis Cedoz, Pier Giuseppe Sessa, Sertan Girgin, Arthur Douillard, and Olivier Bachem. Warp: On the benefits of weight averaged rewarded policies, 2024. URL https://arxiv.org/abs/2406.16768.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Matt Turner. Steering llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting*

*of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15504–15522. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.828. URL https://doi.org/10.18653/v1/2024.acl-long.828.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems*, volume 35, pages 9460–9471, 2022. URL https://papers.nips.cc/paper_files/paper/2022/hash/3d719fee332caa23d5038b8a90e81796-Abstract-Conference.html.

Yunhao Tang, Kunhao Zheng, Gabriel Synnaeve, and Rémi Munos. Optimizing language models for inference time objectives using reinforcement learning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/tang25o.html.

Alexandre Verine, Florian Le Bronnec, Kunhao Zheng, Alexandre Allauzen, Yann Chevaleyre, and Benjamin Négrevergne. Improving diversity in language models: When temperature fails, change the loss. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/verine25a.html.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 15362–15376. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.859. URL https://doi.org/10.18653/v1/2024.emnlp-main.859.

Ke Wang, Nikolaos Dimitriadis, Guillermo Ortiz-Jiménez, François Fleuret, and Pascal Frossard. Localizing task information for improved model merging and compression. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 50268–50287. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/wang24k.html.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. Codecontests+: High-quality test case generation for competitive programming, 2025. URL https://arxiv.org/abs/2506.05817.

Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, Proceedings of Machine Learning Research, pages 23965–23998. PMLR, 2022. URL https://proceedings.mlr.press/v162/wortsman22a.html.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning, 2025. URL https://arxiv.org/abs/2509.02479.

Enneng Yang, Zhenyi Wang, Li Shen, Shiwei Liu, Guibing Guo, Xingwei Wang, and Dacheng Tao. Adamerging: Adaptive model merging for multi-task learning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=nZP6NgD3QY.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng,

Yuxuan Tong, Chi Zhang, Mofan 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, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Chujie Zheng, Ziqi Wang, Heng Ji, Minlie Huang, and Nanyun Peng. Model extrapolation expedites alignment. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 1025–1041. Association for Computational Linguistics, 2025a. URL https://aclanthology.org/2025.acl-long.51/.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin NéXuanjinggrevergne, and Gabriel Synnaeve. What makes large language models reason in (multi-turn) code generation? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025b. URL https://openreview.net/forum?id=Zk9guOl9NS.

# A Experimental Details

## A.1 Test Input Length Distribution

Figure 13 compares the input character length distributions of CodeContests/train and CodeContestsPlus, alongside error decompositions of RL checkpoints trained on each and evaluated on LCB/hard. Despite having more test cases, CodeContests/train produces more resource-limit failures on LCB/hard (28.9% vs. 15.8%), because its tests rarely reach the input sizes that stress algorithmic efficiency. The number of tests alone does not determine coverage; it is the range of input sizes, and thus the optimization pressure they induce, that drives the trade-off frontier.

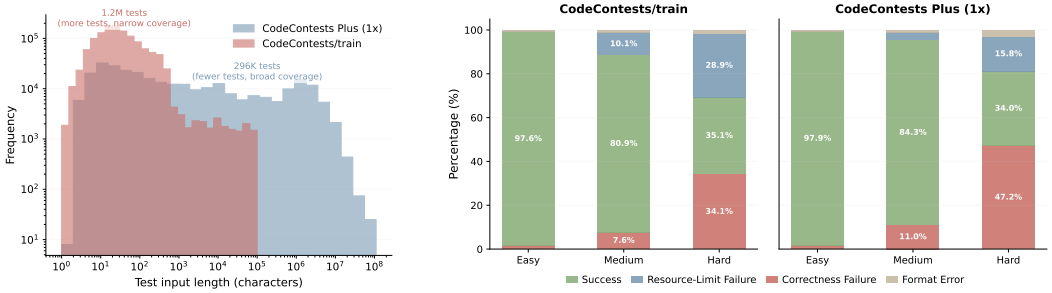



Figure 13: **Left**: Distribution of test case input lengths. CodeContestsPlus (blue) has fewer tests than CodeContests/train (red) but covers a much wider range of input sizes. **Right**: Error composition after RL training on CodeContests/train vs. CodeContestsPlus, by difficulty split on LCB. Despite having more tests, training on CodeContests/train produces more resource-limit failures on LCB/hard problems (28.9% vs. 15.8%), because its tests rarely reach the input sizes that stress algorithmic efficiency.

## A.2 7B Model SFT Training Details

We start from Qwen 2.5 7B (Qwen team, 2025) base model and fine-tune on a mixture of OpenCodeReasoning-2 (Ahmad et al., 2025) (Python and C++) and OpenMathReasoning (Moshkov et al., 2025) for exactly 1 epoch. Training runs for 24,829 steps with 16 H100 nodes. We use the following hyperparameters: local batch size 2, sequence length 32,768, learning rate $8.6 \times 10^{-6}$ with 1,000 warmup steps followed by a constant schedule, and gradient norm clipped at 1.0. We use FSDP in bf16 with tensor parallelism size 4. Linear layers use rowwise FP8 quantization (one scaling factor per matrix row for both activations and weights); all other computations are in bf16.

## A.3 7B Model RL Training Details

We train with a simplified asynchronous online RL recipe as shown in FAIR CodeGen team et al. (2025), namely Dr.GRPO (Liu et al., 2025) using DAPO-style (Yu et al., 2025) asymmetric clipping ($\epsilon_{\text{low}}$=0.2, $\epsilon_{\text{high}}$=0.25) and zero-advantage skipping, with no KL penalty ($\beta$=0) and no reference model. Each prompt produces 16 rollouts at temperature 1.0. We use AdamW ($\beta_1$=0.9, $\beta_2$=0.95, weight decay 0.1) with learning rate $1.4 \times 10^{-7}$, 100 warmup steps followed by a constant schedule, and gradient norm clipped at 0.1. Batching is token-based with a maximum of 32,768 tokens; sequence length is 32,768. The worker model is updated every 4 gradient steps in the trainer side. Training runs for 6,000 steps across 32 H100 nodes (no tensor parallelism) in bf16 with selective activation checkpointing, with 1:1 trainer:worker ratio. No FP8 is used for RL training.

We train on CodeContestsPlus (1×) with binary reward, sweeping the input-length threshold $k \in \{2, \ldots, 6, \infty\}$ across independent runs. Evaluation on LCB/v5 is with temperature 0.6, top-$p$ 0.95, and 10 samples per problem. A single full RL run takes ∼19 hours wall-clock.

### A.4 32B Pure-Reasoning and Agentic Coding RL Training Details

Both the pure-reasoning and agentic-coding settings start from the same CWM-SFT 32B checkpoint and share most hyperparameters with the tool-use setting described in Appendix A.5. We list the shared configuration first, then note setting-specific differences.

**Shared configuration.** Both settings use Dr.GRPO with DAPO-style asymmetric clipping ($\epsilon_{\text{low}}$=0.2, $\epsilon_{\text{high}}$=0.25) and zero-advantage skipping, with no KL penalty and no reference model. Reward function is binary. AdamW with learning rate $1.4 \times 10^{-7}$, constant schedule with 400-step warmup, and gradient norm clipped at 0.1. Each prompt produces 16 rollouts at temperature 1.0. The trainer-to-worker ratio is 1:1, with the model updated every 4 worker steps. Training uses 256 H100 GPUs (32 DP $\times$ 8 TP) with FSDP in bf16, torch.compile, and selective activation checkpointing. Pure reasoning trains for 8 000 steps ($\sim$44 hours per run); agentic coding trains for 4 000 steps ($\sim$16 hours per run). We sweep the input-length threshold $k \in \{2, \ldots, 6, \infty\}$ across independent runs, identical to the tool-use setting.

**Pure reasoning.** The pure-reasoning setting use sequence length 32 768 and no tool calls. The model generates a single reasoning trace in a `<think>` block followed by the code submission. Evaluation uses temperature 0.6 and top-$p$ 0.95 with 10 samples per problem.

**Agentic coding.** The agentic setting uses a sandboxed terminal environment backed by Modal, a cloud sandbox service, with sequence length 61 440 and up to 72 interaction turns per trajectory. The model can execute arbitrary shell commands, write and run code, and observe terminal output within a sandbox that enforces a 120-second session timeout and 1-8 GB memory limits with network access blocked. Evaluation uses temperature 1.0 and top-$p$ 0.95 with 5 samples per problem.

### A.5 32B Tool-Use Model Training Details

We describe two parts of the 32B tool-use setup here.

First, two RL-to-SFT bootstrapping iterations build one shared SFT initialization, because CWM-SFT does not reliably use our stdin-oriented `python-io` interface for competitive-programming tasks.

Second, starting from that same initialization, i.e., SFT checkpoint, the final RL phase launches the verifier-strictness sweep analyzed in the main text. In this final sweep, the compared checkpoints are trained under nested binary coverage rewards, with the task distribution, tool interface, and RL hyperparameters fixed across thresholds. Thus the comparison varies unit-test coverage, not the earlier bootstrapping curriculum.

The bootstrapping iteration proceeds as follows:

1. **Iteration 0 (RL).** Starting from CWM-SFT, run RL with reward shaping on competitive-programming problems to teach the model to invoke a Python interpreter that accepts standard input (`python-io`). Correct trajectories are collected, filtered and downsampled for SFT.
2. **Iteration 1 (SFT $\rightarrow$ RL).** Distill the collected tool-use trajectories from Iteration 0 via SFT back into CWM-SFT; then run RL on CodeContestsPlus with reward shaping that encourages a stress-testing tool (`python-stress`), lets the model generate large inputs and check whether its solution timeouts. This tool functions the same as normal python tool but uses a different name for better behavior tracking. Correct trajectories are collected, filtered and downsampled for SFT.
3. **Iteration 2 (SFT).** Distill the collected tool-use trajectories from Iteration 1 via SFT back into CWM-SFT.

Finally, we launch the RL sweep under different input thresholds, starting with the SFT checkpoint produced from Iteration 2.

#### A.5.1 Iteration 0: Tool-Use RL

We start from CWM-SFT 32B, which can reason in a `<think>` block but has never used a Python tool for code that reads from stdin. We run RL with Dr.GRPO on CodeContests/train (12 275 problems)

with two tools available: `python` (execute a code snippet) and `python-io` (execute code that requires stdin). The prompt includes a small number of few-shot examples demonstrating `python-io` usage. The reward function gives hierarchical partial credit for tool invocation: full reward ($+1$) for passing with `python-io` in the first turn, reduced credit ($+0.8$, $+0.5$, $0$) for passing with progressively less desirable tool usage, and asymmetric penalties on failure that are smaller when the model at least attempted tool use. This encourages the model to explore tool use early in training.

#### A.5.2 Iteration 1: Distillation and Stress-Testing RL (SFT $\rightarrow$ RL)

**SFT phase.** From the Iteration 0 RL run, we collect trajectories that (i) contain at least one `python-io` call and (ii) lead to a correct final answer. We apply diversity filtering to balance across dialog lengths, tool usage patterns, and task coverage, reducing the dataset from 804k to 276k examples. We also filter to a maximum of 32 768 tokens per trajectory and reformat reasoning tokens from the `<think>` tag into the native `<|reasoning_thinking_start|>` format reserved for CWM-SFT. SFT trains for 1 600 steps at learning rate $10^{-6}$, starting again from CWM-SFT but not the RL checkpoint, with batch size 2 and sequence length 32 768 across 32 H100 nodes (32 DP $\times$ 8 TP). Linear layers use rowwise FP8 quantization; all other computations are in bf16.

**RL phase.** Starting from the SFT checkpoint, we run RL on CodeContestsPlus ($1\times$) with the same tool-use environment as iteration 0 plus the `python-stress` tool, which lets the model write an input generator alongside its solution and reports wall-clock execution time. The reward function shifts the shaping target from `python-io` to `python-stress`: full reward ($+1$) for passing when `python-stress` was used anywhere in the trajectory, reduced credit ($+0.5$) for passing with only generic tool use, and a small penalty ($-0.2$) for failing despite using `python-stress` versus full penalty ($-1$) for failing without any tool use. This encourages the model to adopt stress testing for efficiency verification before submission.

#### A.5.3 Iteration 2: Distillation and RL Sweep

**SFT phase.** We adopt the same SFT strategy as Iteration 1. Instead of merge the SFT dataset gathered from Iteration 0 and Iteration 1, we use exclusively the dataset from Iteration 1 which supersets the behavior presented in Iteration 0. We use the same hyperparameters and training steps as in Iteration 1.

**RL phase.** Starting from the SFT checkpoint, we run RL on CodeContestsPlus ($1\times$) with the same tool-use environment as iteration 1. The reward function becomes binary which grades only the final correctness. This is the RL run where we sweep different test coverage threshold.

#### A.5.4 Shared RL Hyperparameters

All three RL phases use Dr.GRPO (Liu et al., 2025) with DAPO-style (Yu et al., 2025) asymmetric clipping ($\epsilon_{\text{low}}$=0.2, $\epsilon_{\text{high}}$=0.25) and zero-advantage skipping, with no KL penalty and no reference model. We use AdamW with learning rate $1.4 \times 10^{-7}$, constant schedule with 400-step warmup, and gradient norm clipped at 0.1. Each prompt produces 16 rollouts at temperature 1.0, with a maximum of 6 tool calls per trajectory. The trainer-to-worker ratio is 1:1, with the model updated every 4 worker steps. Training uses 256 H100 GPUs (32 DP $\times$ 8 TP) with FSDP in bf16, torch.compile, and selective activation checkpointing. The RL phases differ in training data, reward target, sequence length, and step count: iteration 0 trains for 4 000 steps on CodeContests/train with `python-io` reward shaping and sequence length 131 072; iteration 1 and 2 trains for 8 000 steps on CodeContestsPlus ($1\times$) with `python-stress` reward shaping and binary pass-fail reward respectively and sequence length 65 536 (approximately 18 hours per run).

**Observation.** During iteration 0, we ran RL on both DMC/train and CodeContestsPlus. The two training sets yield similar overall pass rates, but different error profiles: CodeContestsPlus produces fewer timeouts and more wrong answers. This observation motivated the core investigation of this paper (Section 3).

Appendix G shows one representative rollout from each iteration.

### A.6 Compute Resources

All experiments use NVIDIA H100 80 GB GPUs. Table 1 summarizes the hardware layout and wall-clock time for each training configuration. For the 7B, 32B pure-reasoning, and 32B agentic-coding settings, we sweep the input-length threshold $k \in \{2, \ldots, 6, \infty\}$ across six independent runs; total GPU-hours for those rows reflect this $\times 6$ factor. The 32B tool-call pipeline requires a bootstrapping phase (2 RL + 2 SFT runs) before the final threshold sweep. The bootstrapping rows are single runs. The Iter 0 RL uses 131k sequence length while the Iter 1 RL uses 64k sequence length, which compensates the training steps and results in roughly the same wall-clock training time.

Table 1: Compute resources per training configuration. DP = data parallelism, TP = tensor parallelism. Bootstrapping rows (marked †) are single runs that build the starting policy for the tool-call threshold sweep.

| Setting | Model | GPUs | DP×TP | Steps | Hours/run | Total GPU-h |
|---|---|---|---|---|---|---|
| *Threshold-swept settings (×6 runs)* | | | | | | |
| 7B pure reasoning RL | 7B | 32 | 32×1 | 6 000 | ∼19 | ∼3 648 |
| 32B pure reasoning RL | 32B | 256 | 32×8 | 8 000 | ∼44 | ∼67 584 |
| 32B tool-call RL | 32B | 256 | 32×8 | 8 000 | ∼18 | ∼27 648 |
| 32B agentic coding RL | 32B | 256 | 32×8 | 4 000 | ∼16 | ∼24 576 |
| *Tool-call bootstrapping† (2 RL + 2 SFT, single runs)* | | | | | | |
| Iter 0 RL | 32B | 256 | 32×8 | 4 000 | ∼18 | ∼4 608 |
| Iter 1 SFT | 32B | 256 | 32×8 | 1 600 | <1 | ∼256 |
| Iter 1 RL | 32B | 256 | 32×8 | 8 000 | ∼18 | ∼4 608 |
| Iter 2 SFT | 32B | 256 | 32×8 | 1 600 | <1 | ∼256 |

# B Frontier Validation and Robustness

This appendix checks that the frontier does not come from mislabeled failures (Appendix B.1) or from training noise (Appendix B.2), and collects further evidence for the frontier across settings and over training (Appendix B.3).

## B.1 Error Taxonomy Validation with Extended Reclassification

A submission that exceeds resource limits does not necessarily implement a correct algorithm; it may contain a bug masked by the resource limit. To assess the sensitivity of our results to this ambiguity, we apply a two-stage reclassification protocol to every submission originally labeled as a resource-limit failure. We detail the protocol in Section B.1, with Figure 5 showing the comparison of the extended reclassification and the raw sandbox signal used in the main text. Section B.1.2 reveals one interesting finding: a substantial fraction of timed-out submissions are in fact memory-exhaustion failures masked by the time limit. Whether a slow submission will eventually exhaust memory or time is fundamentally difficult to determine without running it to completion, which we believe explains why most existing work aggregates both failure modes into a single timeout category.

### B.1.1 Reclassification Protocol

**Stage 1: extended re-evaluation.** We re-execute each submission labeled as resource-limit failure with a $100\times$ time budget. Under this extended budget, submissions fall into one of four categories: (i) *all-pass*, confirming a correct but algorithmically slow implementation; (ii) *wrong answer*, revealing a correctness bug that was masked by the original time limit; (iii) *out of memory*, indicating that the submission is memory-intensive and the original timeout pre-empted the true failure mode; or (iv) *still timeout*, remaining ambiguous.

Table 2 reports the breakdown for the high-coverage training condition (all test cases included) across all three inference settings. A substantial fraction of resource-limit failures are confirmed correct under the extended budget (27-40% across settings and thresholds), while out-of-memory errors account for 5-14% of originally timed-out submissions. Table 2 reports the full breakdown.

Table 2: Stage 1 re-evaluation outcomes (% of timed-out submissions) under a $100\times$ time budget. Each column corresponds to one test-coverage threshold; each row group to one inference setting. "Total" is the number of submissions originally labeled as resource-limit failures.

| | $\leq 10^2$ | $\leq 10^3$ | $\leq 10^4$ | $\leq 10^5$ | $\leq 10^6$ | all |
|---|---|---|---|---|---|---|
| *Pure Reasoning* | | | | | | |
| Total | 153 | 167 | 157 | 158 | 124 | 104 |
| All-pass | 37.3% | 31.1% | 36.9% | 39.9% | 40.3% | 34.6% |
| Wrong answer | 3.3% | 5.4% | 4.5% | 0.6% | 2.4% | 2.9% |
| Out of memory | 9.2% | 4.8% | 7.6% | 7.6% | 5.6% | 8.7% |
| Still timeout | 47.1% | 53.9% | 43.3% | 50.0% | 46.8% | 49.0% |
| Other | 3.3% | 4.8% | 7.6% | 1.9% | 4.8% | 4.8% |
| *Tool Use* | | | | | | |
| Total | 140 | 139 | 126 | 109 | 99 | 89 |
| All-pass | 22.1% | 17.3% | 15.9% | 22.0% | 33.3% | 36.0% |
| Wrong answer | 4.3% | 3.6% | 5.6% | 3.7% | 3.0% | 11.2% |
| Out of memory | 10.0% | 7.2% | 14.3% | 11.9% | 9.1% | 10.1% |
| Still timeout | 57.9% | 68.3% | 58.7% | 58.7% | 51.5% | 41.6% |
| Other | 5.7% | 3.6% | 5.6% | 3.7% | 3.0% | 1.1% |
| *Agentic Coding* | | | | | | |
| Total | 139 | 131 | 114 | 89 | 80 | 69 |
| All-pass | 26.6% | 26.7% | 31.6% | 25.8% | 27.5% | 27.5% |
| Wrong answer | 2.9% | 4.6% | 2.6% | 3.4% | 3.8% | 5.8% |
| Out of memory | 7.2% | 6.1% | 9.6% | 6.7% | 10.0% | 13.0% |
| Still timeout | 56.8% | 55.7% | 49.1% | 60.7% | 53.8% | 52.2% |
| Other | 6.5% | 6.9% | 7.0% | 3.4% | 5.0% | 1.4% |

**Stage 2: LLM judgment.** For submissions that still exceed resource limits or sandbox error after Stage 1 (the "still timeout", "out of memory" and "Other" rows), we query a strong model, Claude-4.5-Sonnet with multiple generations and majority voting to binary classify whether the implementation is logically correct but algorithmically naive, or whether it contains a correctness bug. Table 3 reports the fraction of resource-limit failures reclassified as correctness failures after combining both stages, across all settings and thresholds. Crucially, the number of successful submissions is unchanged in every condition: reclassification only moves submissions from the resource-limit failure category to correctness failure, never to success.

Table 3: Final reclassification outcome after both stages (% of original resource-limit failures). The two rows sum to 100% for each condition.

| | $\leq 10^2$ | $\leq 10^3$ | $\leq 10^4$ | $\leq 10^5$ | $\leq 10^6$ | all |
|---|---|---|---|---|---|---|
| *Pure Reasoning* | | | | | | |
| Reclass. correctness failure | 17.6% | 16.2% | 15.9% | 17.1% | 15.3% | 20.2% |
| Remain resource-limit failure | 82.4% | 83.8% | 84.1% | 82.9% | 84.7% | 79.8% |
| *Tool Use* | | | | | | |
| Reclass. correctness failure | 5.0% | 10.1% | 13.5% | 15.6% | 6.1% | 22.5% |
| Remain resource-limit failure | 95.0% | 89.9% | 86.5% | 84.4% | 93.9% | 77.5% |
| *Agentic Coding* | | | | | | |
| Reclass. correctness failure | 20.1% | 27.5% | 27.2% | 19.1% | 22.5% | 31.9% |
| Remain resource-limit failure | 79.9% | 72.5% | 72.8% | 80.9% | 77.5% | 68.1% |

**Robustness of the trade-off.** Figure 5 compares the trade-off curve before and after reclassification across all three inference settings and all six test-coverage thresholds. In each panel, raw sandbox labels (solid) and reclassified labels (dashed) trace nearly identical frontiers. The reclassified curve shifts slightly toward higher correctness failure rates, as expected from unmasking hidden bugs, but the shape and slope of the frontier are preserved. Since the relative orders are preserved, the two labeling schemes agree closely and can be seen as a global shift of the trade-off curve, we use raw sandbox labels throughout the main paper to avoid the computational cost of running the full reclassification pipeline at every evaluation checkpoint.

### B.1.2 Hidden Memory Exhaustion in Timeout Labels

Standard competitive-programming evaluation pipelines enforce both a time limit and a memory limit, but when a submission exceeds the time limit first, the sandbox reports a timeout regardless of whether the code would also have exhausted memory. Memory exhaustion is therefore aggregated into the timeout category, making it invisible in downstream analysis.

Our extended re-evaluation (Stage 1 of the reclassification protocol, Appendix B.1) surfaces this hidden failure mode. Under the $100\times$ time budget, 5% to 14% of originally timed-out submissions across all settings and thresholds terminate with an out-of-memory error rather than a wrong answer or a continued timeout (Table 2).

These submissions implement memory-intensive algorithms (e.g., building large lookup tables or storing full search trees) whose true failure mode is masked by the shorter time limit. Since competitive-programming benchmarks are the primary training and evaluation environment for code RL, this conflation means that a non-trivial fraction of what existing work treats as "slow but possibly correct, i.e., timeout error" code is in fact code that would crash from memory exhaustion even given enough time. Disentangling time and memory failures may be important for designing reward signals that distinguish algorithmic efficiency from implementation correctness.

## B.2 Seed Variance Analysis

To quantify the statistical reliability of our main findings, we repeat the pure-reasoning RL runs for each of the three coverage thresholds (Keep $\leq 10^3$, Keep $\leq 10^5$, and All tests) with three independent random seeds at both the 7B and 32B model scales, and report mean $\pm$ one standard deviation.

**Pass@1 stability.** Figure 14 shows the LCB pass@1 training curves across seeds. At the 32B scale, all three thresholds exhibit tight variance bands throughout training, with standard deviations below 0.5% at step 6k (e.g. Keep $\leq 10^3$: $61.7 \pm 0.5\%$; All tests: $61.6 \pm 0.2\%$). The 7B runs show moderately larger variance ($\sim$0.7-1.0%), consistent with smaller models being more sensitive to initialization, but the relative ordering among thresholds is preserved across all seeds.

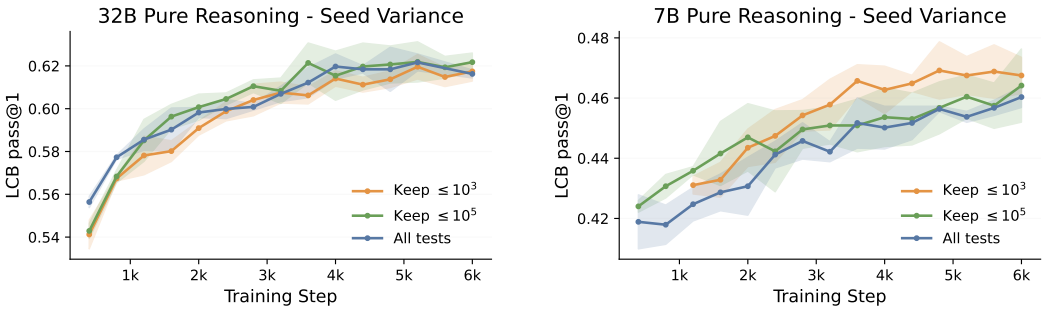



Figure 14: Pass@1 on LCB-hard across training steps for 32B (left) and 7B (right) pure-reasoning runs. Shaded regions denote $\pm 1$ standard deviation over three seeds.

**Failure-mode stability.** Figure 15 decomposes the failure rate on LCB-hard into *resource-limit failures* and *correctness failures*. The optimization-correctness trade-off reported in the main text is robust across seeds: stricter filtering (Keep $\leq 10^3$) consistently yields higher resource-limit failure rates (32B: $17.3 \pm 1.3\%$ vs. $8.7 \pm 0.9\%$ for All tests at step 6k) while reducing correctness failure rates ($37.5 \pm 0.7\%$ vs. $49.0 \pm 3.3\%$). The same pattern holds at the 7B scale. In all cases the inter-seed standard deviation is substantially smaller than the inter-threshold differences, confirming that the reported trade-offs reflect genuine training dynamics rather than seed-dependent fluctuations.

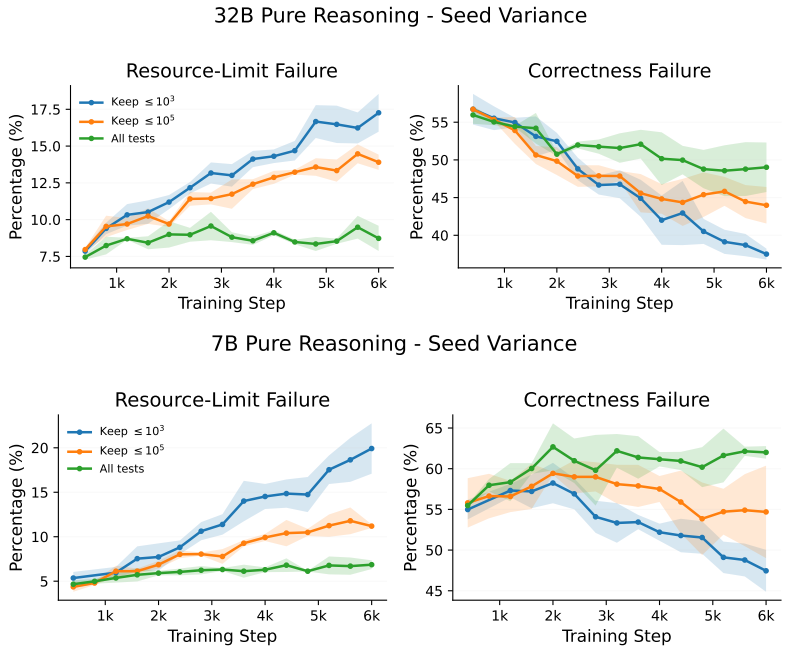



Figure 15: Resource-limit failure and correctness failure rates on LCB-hard for 32B (up) and 7B (bottom). Shaded regions denote $\pm 1$ standard deviation over three seeds. The trade-off between failure modes is consistent across seeds at both scales.

## B.3 Extended Frontier Evidence

### B.3.1 Per-Setting Extrapolation Plots

Figures 2 and 7 in the main text show the extrapolative continuation as compact panels. This subsection presents standalone plots for each setting and scale with full interpolation, extrapolation, and format-error detail.

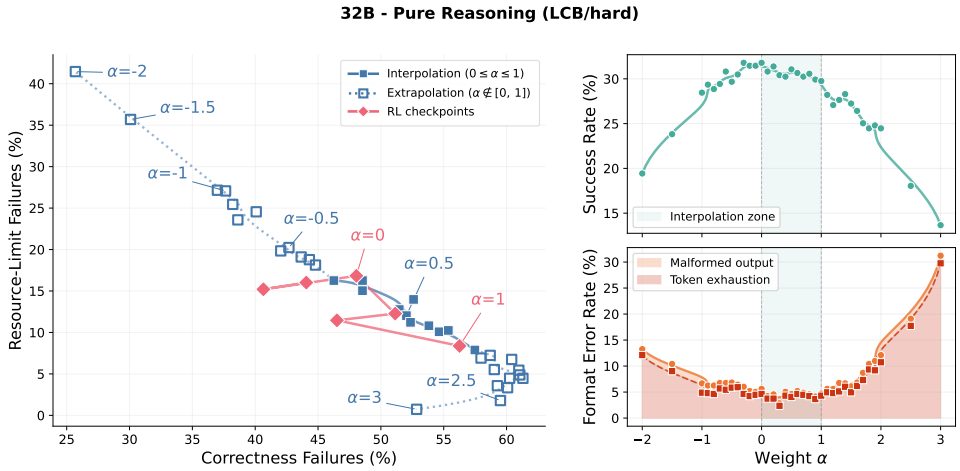



Figure 16: 32B pure reasoning on LCB/hard.

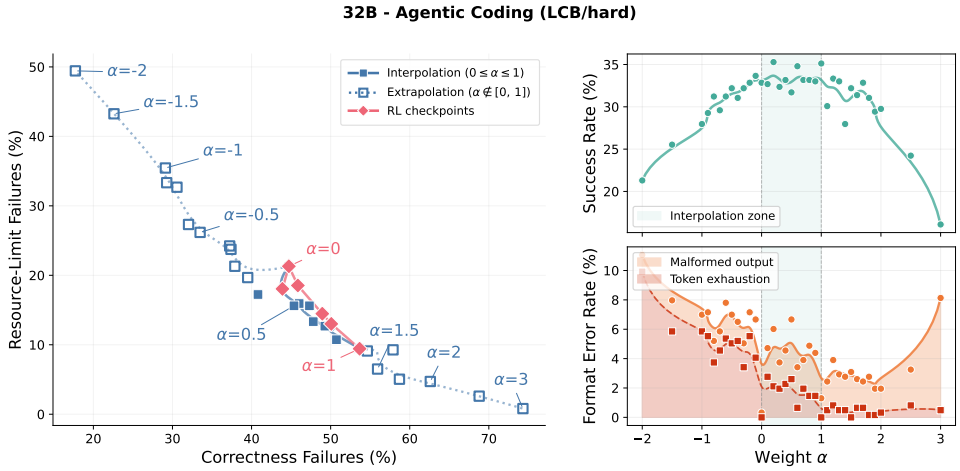



Figure 17: 32B agentic coding on LCB/hard.

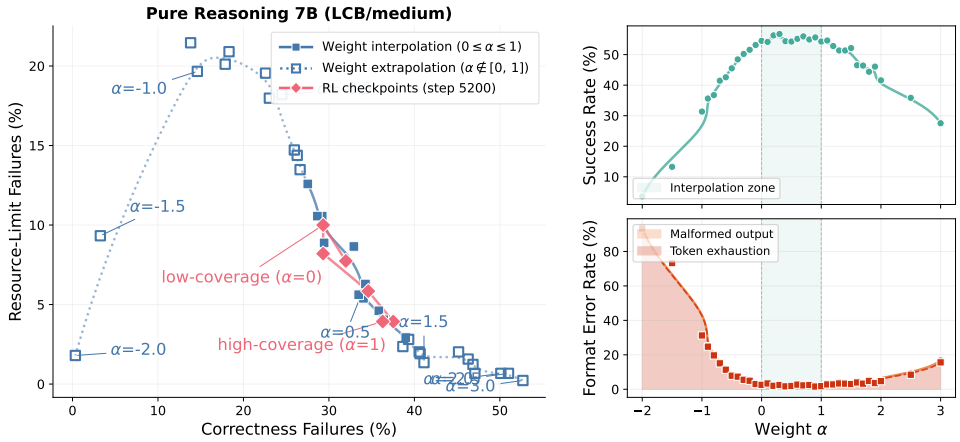



Figure 18: 7B pure reasoning on LCB/medium (capability-matched split).

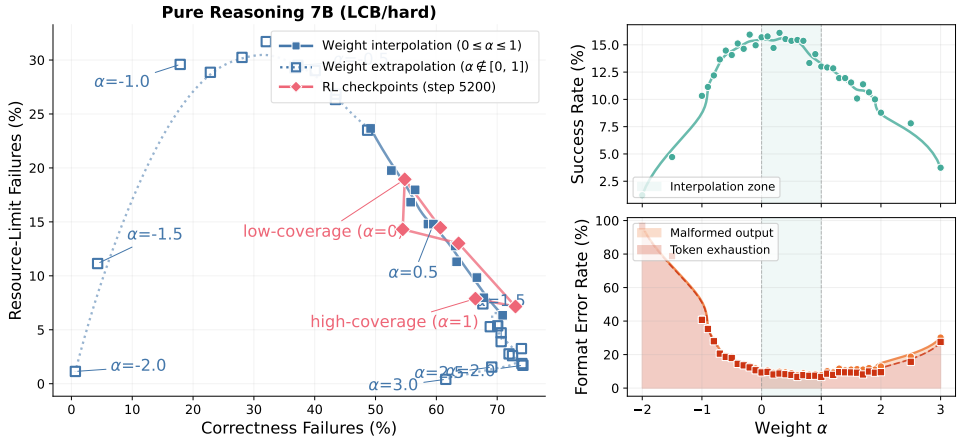



Figure 19: 7B pure reasoning on LCB/hard. The stable extrapolation range is narrower than LCB/medium, with format errors dominating earlier on the low-$\alpha$ side.

### B.3.2 Full Pairwise Transition Matrices

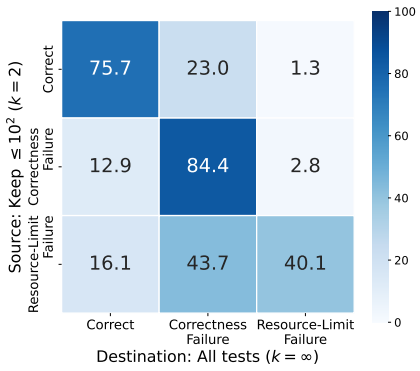



Figure 20: Task-level transition matrix from $k$=2 (keep $\leq 10^2$) to $k$=$\infty$ (all tests) on 123 hard problems. Taken from pure-reasoning setting in late stage of the training (7.2k steps). Rows are normalized to 100%. Of resource-limit failures under low coverage, 43.7% become correctness failures under high coverage, ruling out an aggregation artifact.

Figure 21 shows the complete set of task-level transition matrices between all pairs of test-coverage conditions on hard problems. The pattern is consistent throughout: the dominant off-diagonal transitions are between resource-limit failures and correctness failures, confirming that the trade-off operates at the individual task level across all condition pairs.

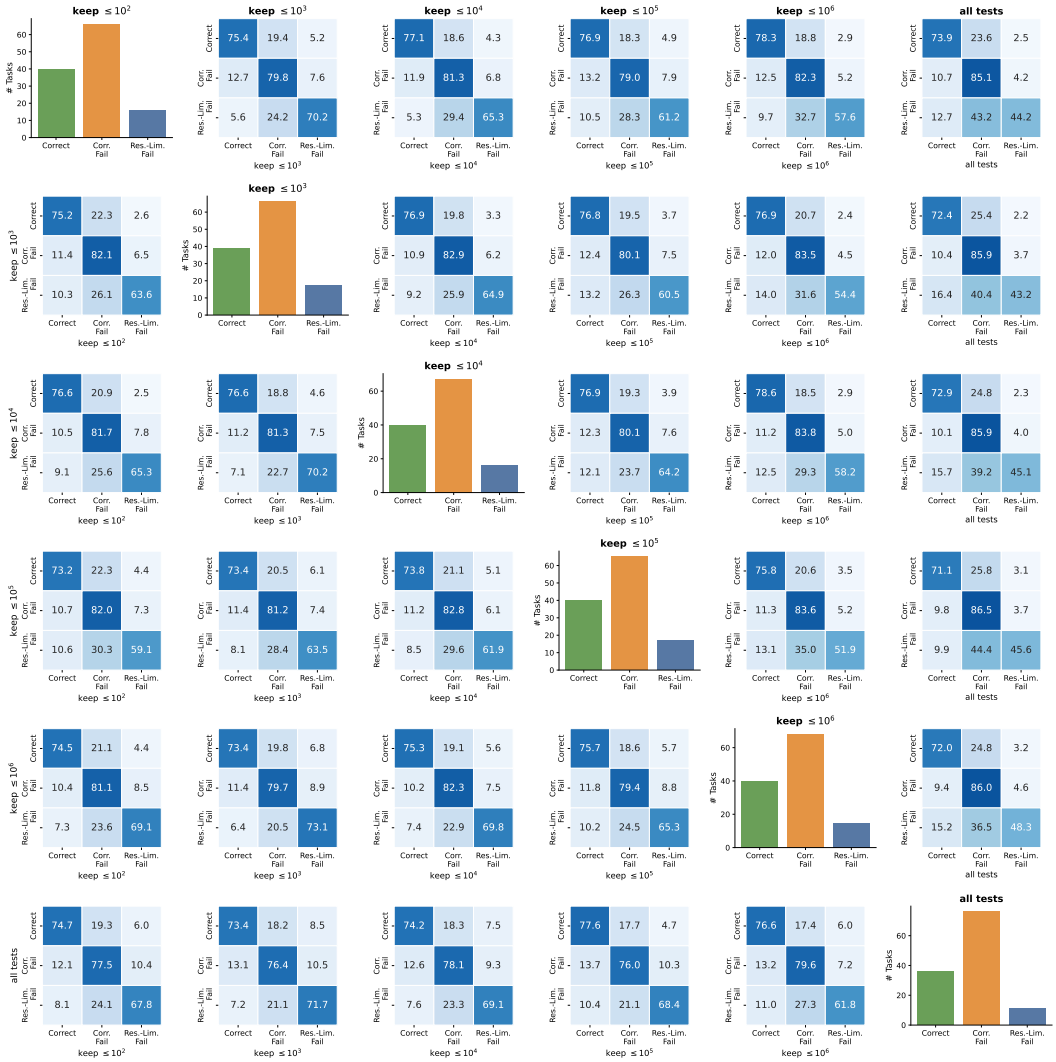



Figure 21: Full grid of pairwise task-level transition matrices on hard problems. Diagonal panels show per-condition status distributions. Taken from the pure-reasoning setting in early stage of the training (3.6k steps). Off-diagonal cells are row-normalized to 100%.

### B.3.3 Emergence of the Trade-off Curve During Training

The trade-off curve emerges progressively during RL training. At step 400 (Figure 22, left), all six conditions cluster in a tight region of the (Correctness Failure, Resource-Limit Failure) plane. By step 3600 (center), the runs have diverged along a clear trade-off curve, with the ordering matching the test-coverage threshold monotonically. By step 6400 (right), the separation widens further. This divergence occurs relatively early in training and then stabilizes, suggesting that RL quickly discovers the point on the frontier dictated by its test-coverage level.

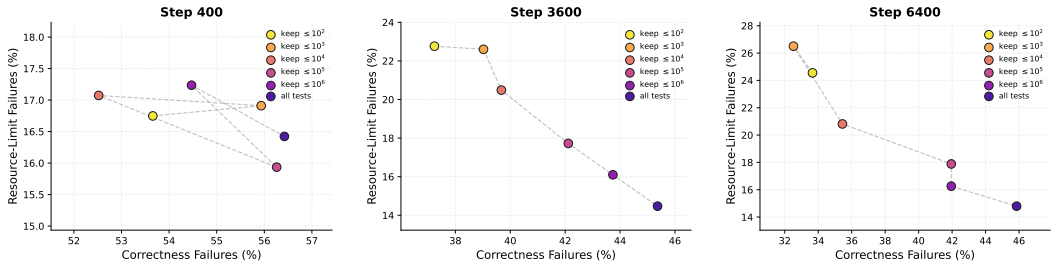



Figure 22: Emergence of the trade-off curve during RL training. At step 400, all six test-coverage conditions are clustered together. By step 3600, they have spread into a clear trade-off curve. By step 6400, the separation widens further.

### B.3.4 Training Dynamics

Figure 23 shows the resource-limit failure and correctness failure counts over an extended training horizon on 32B pure-reasoning setting. The separation between conditions is sustained at later steps, indicating that the trade-off is not a transient phenomenon of early training.

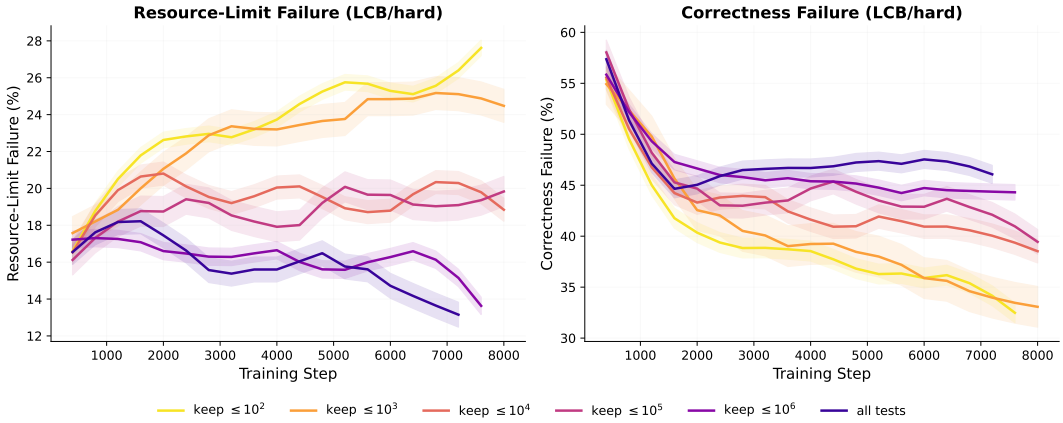



Figure 23: Extended training dynamics showing resource-limit failure and correctness failure counts on hard problems vs. training step on LCB/hard.

# C Inference-Time Ensemble Details

## C.1 Full Solved-Set Overlap

Section 4.2 summarizes how solved-set overlap varies with distance in $\alpha$. Figure 24 shows the full pairwise matrix over $\alpha \in [-2, 3]$. Overlap generally decreases with distance in $\alpha$. Even at the extremes ($\alpha = -2$ vs. $\alpha = 3$), the Jaccard similarity remains above 0.54, indicating that over half of solved problems are shared.

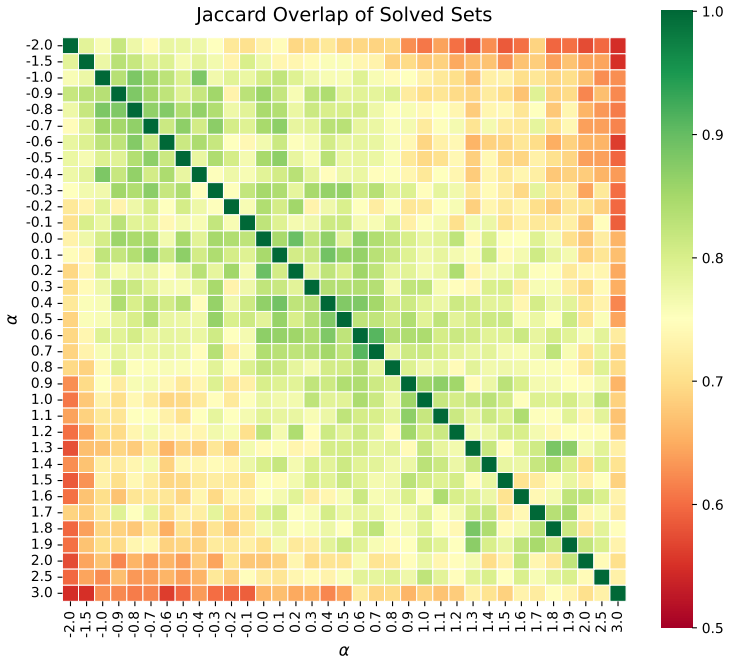



Figure 24: Full pairwise Jaccard similarity between solved sets across the extrapolation range $\alpha \in [-2, 3]$. Overlap decreases steadily with distance in $\alpha$ but remains above 0.54 even at the extremes, confirming gradual drift of the success set across the entire interpolation and extrapolation range.

## C.2 Coverage Headroom Decomposition

Different checkpoints along the interpolation axis solve different subsets of hard problems, so pooling multiple checkpoints and selecting the best submission per problem can exceed the solve rate of every individual checkpoint. We quantify this *coverage headroom* with a greedy set-cover procedure: at each step, the checkpoint whose solved set adds the most new problems is appended to the pool, and we track the cumulative number of problems solved by at least one member. Figure 25 (left) shows the resulting curve for interpolation-only and interpolation-plus-extrapolation pools; the right panel decomposes the total gain over the best single checkpoint into contributions from each regime.

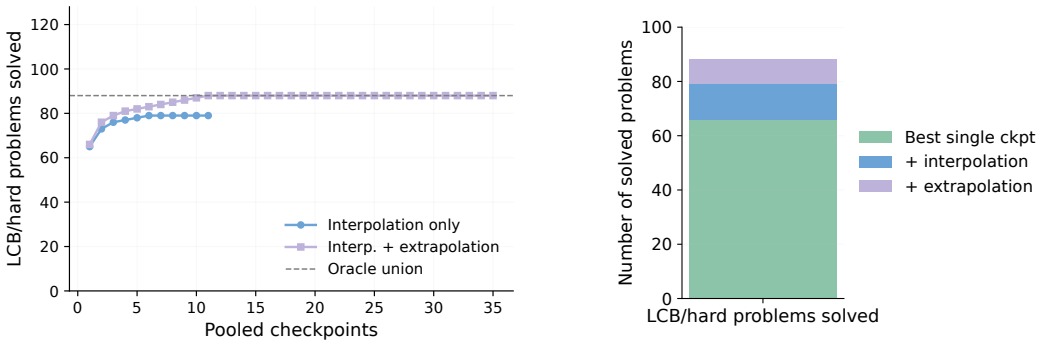



Figure 25: **Left**: Cumulative coverage of LCB/hard (123 problems) as a function of the number of checkpoints in the pool, under a greedy ordering. The interpolation-only pool (blue) plateaus at 79 problems; admitting extrapolated checkpoints (red) continues to grow until it reaches 88. **Right**: Decomposition of the 22-problem coverage gain over the best single checkpoint into the contributions from interpolation and extrapolation. Extrapolation accounts for 9 of the 22 extra problems and is the only path to the final coverage band.

### C.3 Filtered Submission Budgets on LCB/v5

In practice, competitive programming judges allow only a handful of submissions per problem, so raw pass@$k$ with large $k$ overstates the value of a diverse candidate pool. Pass $n$@$k$ measures the solve rate when $k$ candidates are sampled, filtered by public tests, and the top $n$ are submitted. Figure 26 fixes $k$=250 and sweeps $n$ from 1 to 250 on the LCB/v5 hard split. The extrapolation-inclusive five-checkpoint pool dominates the best single RL checkpoint and the interpolation-only pool at every submission budget. At $n$=1, also called best@$k$, the gap is already roughly two percentage points, showing that even a single filtered submission benefits from the broader coverage of the extrapolated checkpoints. The per-problem oracle selects the best checkpoint for each problem independently; the extrapolation pool nearly closes this gap by $n$=250.

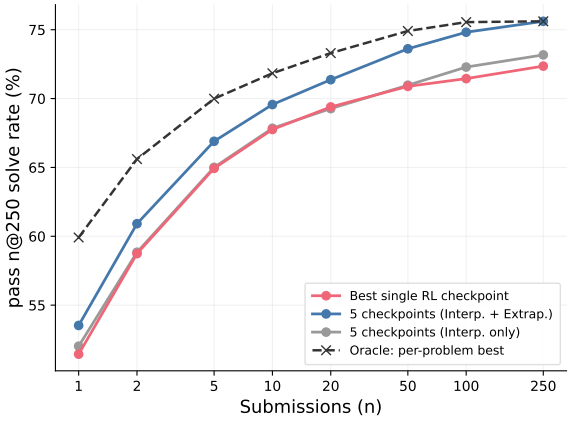



Figure 26: Pass $n$@250 with public-test filtering on LCB/v5 hard. The extrapolation-inclusive five-checkpoint pool is higher than both the best single checkpoint and the interpolation-only pool at each displayed submission budget $n$. This is an in-domain diagnostic of how filtering converts candidate diversity into gains under smaller submission budgets.

## D Weight-Space Geometry

This appendix characterizes the weight-space direction that separates the two RL endpoints along three axes: the singular-value spectrum of each layer's weight difference (Section D.1), the distribution of

squared weight change across layers (Section D.2), and layer stitching experiments that isolate causal contributions to behavior (Section D.3).

The central finding is a dissociation. RL modifies every layer by similar magnitudes and in high-dimensional directions: the top 10 singular values explain only 10 to 19% of the variance within each layer (Section D.1), and the total squared weight change is nearly uniform across layers 2 through 63 (Section D.2). Yet behavioral sensitivity is concentrated in the middle of the network. Layer stitching shows that the first and last quarter of layers (0 to 15 and 48 to 63) are nearly inert, while the middle half (16 to 47) carries the full trade-off shift, with layers 24 to 31 accounting for roughly half of it (Section D.3).

### D.1 SVD Spectrum

For each weight matrix in the 32B 64-layer model, we compute the SVD of $\delta = \theta_{\text{high}} - \theta_{\text{low}}$ and measure the cumulative variance explained by the top $k$ singular values. Figure 27 shows the result, normalized by the rank of each matrix. No component type is low-rank: the top 10 singular values of each layer's difference explain only 10 to 19% of variance, and even at 20% of rank no component exceeds 80% explained variance.

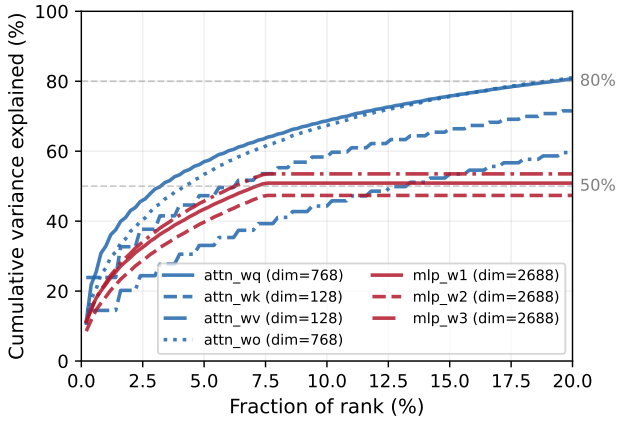



Figure 27: Cumulative variance explained by the top singular values of the weight difference $\delta = \theta_{\text{high}} - \theta_{\text{low}}$, averaged across layers and normalized by rank. Attention matrices concentrate variance faster than MLP matrices, but neither is low-rank: at 20% of rank, no component type exceeds 80% explained variance.

### D.2 Per-Layer Variance Distribution

We define the *variance ratio* of layer $\ell$ as the fraction of total squared change ($\|\delta\|^2$ summed across all parameters) attributable to that layer; under a uniform distribution each of 64 layers would contribute 1.56%. Figure 28 shows that the variance ratio is nearly uniform across layers 2 through 63, with only layers 0 and 1 markedly below baseline.

### D.3 Layer Stitching

The previous two analyses show that weight changes are distributed and high-dimensional, but this does not imply uniform causal importance. We construct Frankenstein models by taking layers 0 through $k-1$ from $\theta_{\text{high}}$ and layers $k$ through 63 from $\theta_{\text{low}}$ (and vice versa), sweeping cutpoints $k \in \{0, 16, 24, 32, 48, 64\}$. We additionally test block swaps of layers 24 to 31 (8 layers) and 16 to 47 (32 layers).

All stitched models remain on the trade-off curve, with success rates in the 32.8 to 37.4% band (Figure 29). The first and last quarter of layers are nearly inert: cutpoints at $k$=16 and $k$=48 shift timeout counts by fewer than 6 relative to the endpoints. The middle layers carry the full behavioral shift. Swapping layers 24 to 31 alone (12.5% of the network) accounts for roughly half the timeout shift; swapping layers 16 to 47 reproduces or slightly overshoots the full shift (Figure 30).

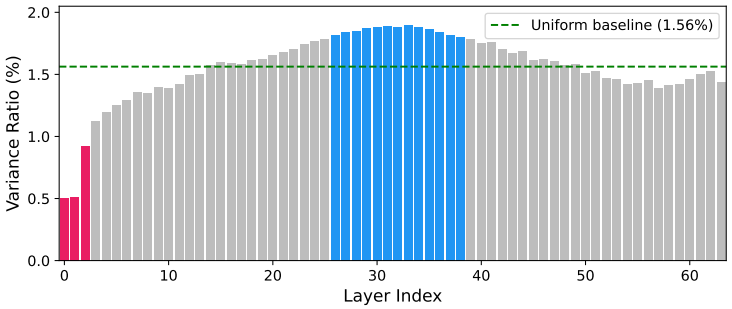



Figure 28: Per-layer variance ratio (fraction of total squared weight change). The dashed line marks the uniform baseline (1.56% per layer). Layers 0-1 (red) are markedly below baseline; layers 24-36 (blue) are mildly above. The distribution is otherwise close to uniform.

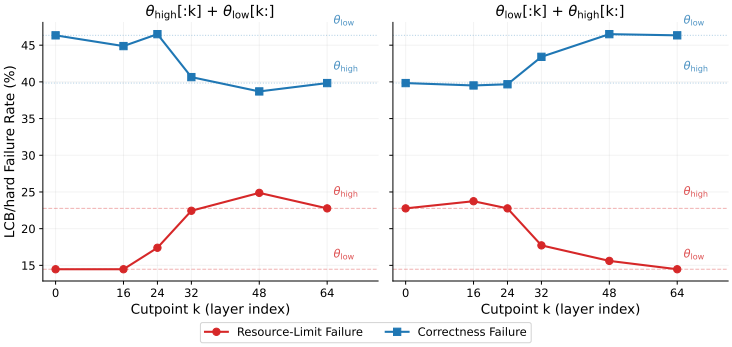



Figure 29: Layer stitching (Frankenstein models): resource-limit failure rate and correctness failure rate on LCB/hard as a function of cutpoint $k$. **Left**: layers 0 through $k-1$ from $\theta_{\text{high}}$, layers $k$ through 63 from $\theta_{\text{low}}$. **Right**: the reverse direction. Dashed lines mark the endpoint levels. The behavioral transition is concentrated in the middle layers (roughly 16-48); edge layers are nearly inert.

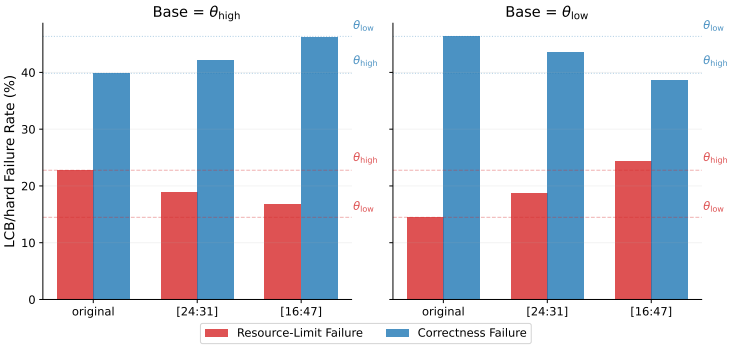



Figure 30: Block swap: replacing a contiguous range of middle layers in one endpoint with the corresponding layers from the other. **Left**: $\theta_{\text{high}}$ base with $\theta_{\text{low}}$ layers swapped in. **Right**: $\theta_{\text{low}}$ base with $\theta_{\text{high}}$ layers swapped in. Swapping layers 16-47 (half the model) nearly reproduces the full behavioral shift; swapping layers 24-31 (8 layers) produces a partial shift, confirming that a dense core of middle layers controls the trade-off.

# E Additional Experiments

This appendix reports experiments beyond the main line: a second weight-space frontier, between pass@1 and pass@$k$ (Appendix E.1), and two attempts to move the correctness-efficiency frontier, reward shaping (Appendix E.2) and failure-conditioned training (Appendix E.3).

## E.1 A Second Weight-Space Frontier: Pass@1 vs Pass@$k$ Performance

The main experiments use nested verifier coverage to create a controlled correctness-efficiency axis. To test whether the weight-averaging geometry is specific to that axis, we ran an auxiliary experiment between two RL checkpoints, trained from the same initialization (the SFT checkpoint from Qwen 2.5 7B as described in main text) but optimized for different sampling behavior. The firstll is trained using standard GRPO which exhibits stronger pass@1, and the second is trained under pass@8 objective as shown in Tang et al. (2025). We form interpolation and extrapolation checkpoints:

$$\theta_\alpha \;=\; \alpha\,\theta_{\text{p@8}} \;+\; (1-\alpha)\,\theta_{\text{p@1}},$$

and evaluate checkpoints for $\alpha \in [-0.5, 1.5]$ on LCB v5 + v6 (20240801-20250501), 454 problems in total, with temperature 1.0 and 200 samples per prompt.

**Interpolation produces a smooth behavioral frontier.** Figure 31 shows that interpolation traces a smooth curve in pass@1/pass@100 space. We follow the Precision/Recall framework (Verine et al., 2025) which uses pass@1 as a proxy for Precision, and pass@100 - pass@1 as a measure for Recall. Moving along the curve changes both the Precision (pass@1) and the Recall (pass@100 − pass@1). This confirms that the interpolation direction captures a meaningful behavioral axis even when the endpoints differ by training objective rather than by verifier coverage. Extrapolation is stable near the endpoints but gives diminishing returns: the most extreme extrapolated checkpoints ($\alpha = -0.5$, $\alpha = 1.5$) degrade on both pass@1 and pass@100.

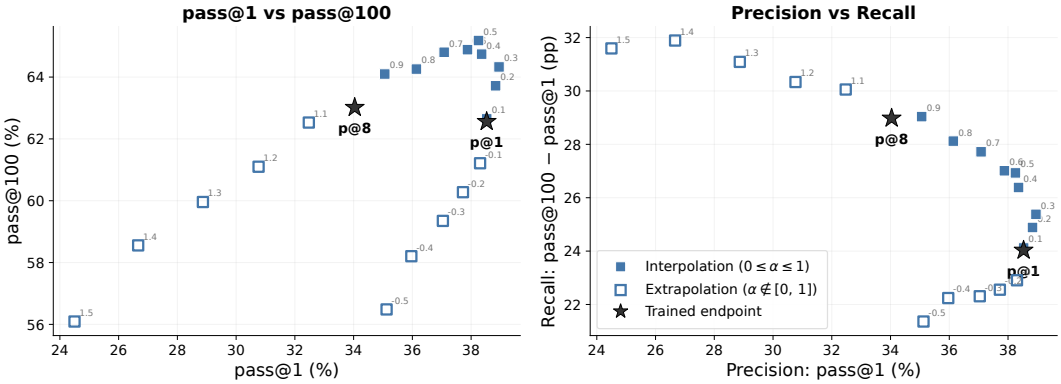



Figure 31: **A second weight-space frontier: pass@1 vs pass@100.** We average a pass@1-oriented endpoint (p@1, $\alpha$=0) and a pass@8-oriented endpoint (p@8, $\alpha$=1). **Left:** Interpolation traces a smooth curve in pass@1/pass@100 space; extrapolation remains locally meaningful but degrades at extreme values. **Right:** In the Precision/Recall view, sampling headroom peaks in the interpolation region, indicating that interpolated checkpoints have the most untapped potential under larger sampling budgets.

**Checkpoint banks exploit solved-set diversity.** As in the verifier-coverage frontier, nearby values of $\alpha$ solve overlapping but distinct problem sets (Figure 32). This turnover can be exploited by checkpoint banks. Figure 33 shows that five-checkpoint banks with 200 samples each outperform either single endpoint with 1000 samples at pass@$k$ for $k \geq 10$, reaching +3.5% on all problems and +5.4% on hard at pass@1000. Including one extrapolated checkpoint ($\alpha$=−0.2) adds marginal diversity on hard problems: it uniquely solves 7 hard problems that no interpolation checkpoint covers. Conversely, 12 problems are uniquely solved by interpolation (6 hard, 6 medium), so extrapolation trades medium-difficulty coverage for hard-problem diversity: precisely the frontier exploration that weight averaging exposes.

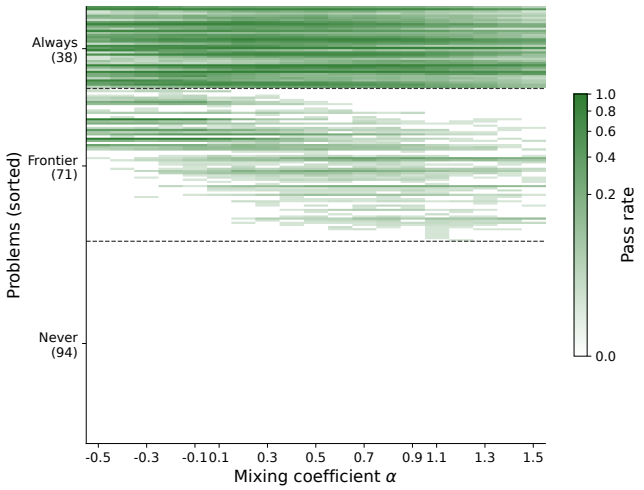



Figure 32: Solved-set turnover along the interpolation axis of pass@1-trained checkpoint and pass@8-trained checkpoint (LCB/hard). Each row is one problem; columns are $\alpha$ values. Unlike the main-text heatmap (Figure 9), which uses a linear color scale, we apply a power-law normalization ($\gamma$=0.3) to highlight problems solved at low pass rates. Of 203 hard problems, 38 are always solved, 94 are never solved, and 71 frontier problems change solvability with $\alpha$, providing the diversity that checkpoint banks exploit.

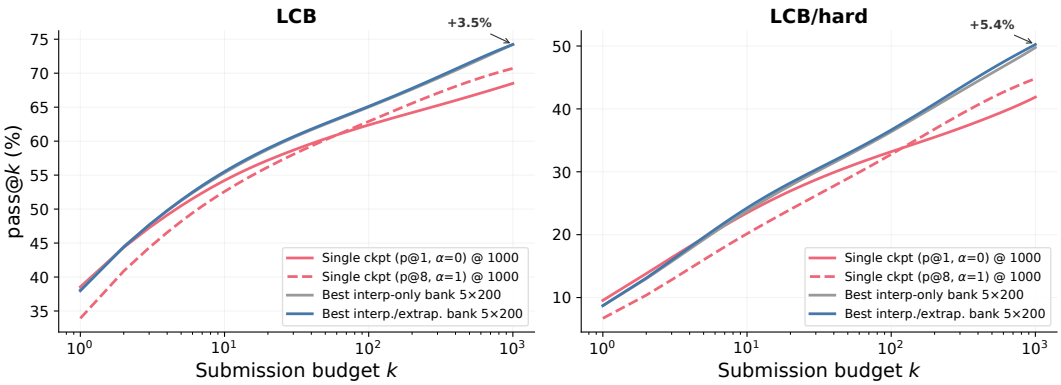



Figure 33: **Matched-budget checkpoint banks on LCB. Left:** On all problems, interpolation-only and interpolation/extrapolation banks overlap, both outperforming single-checkpoint sampling by +3.5% at pass@1000. **Right:** On hard problems, including an extrapolated checkpoint edges out interpolation-only by $\sim$0.5%, trading medium-difficulty coverage for hard-problem diversity.

**Scope and caveats.** We treat this experiment as supportive but not primary evidence. Unlike the nested verifier sweep, this setting is not a controlled sweep over a single monotone reward family, and the endpoints differ in both objective and training stage. Its role is to suggest that the frontier-navigation view can extend beyond correctness-efficiency, not to replace the controlled evidence in the main paper.

### E.2 Reward Shaping Experiments

Reward shaping is the simplest static attempt to soften the binary pass-fail reward. It assigns partial credit for progress on the same test suite, but it never reveals why a rollout failed. It therefore changes the scalarization of the same nested test outcomes without changing the information available to learning. The definitions below make this limitation concrete.

In the reward-shaping experiment, the model receives partial credit according to the largest test-coverage threshold its submission passes. We use the threshold set $\mathcal{K} = \{2, 3, 4, 5, 6, \infty\}$. The

shaped reward is

$$R_{\text{shape}}(y) = \begin{cases} 0 & \text{fails some test with input length} \le 10^2, \\ 1/6 & \text{passes all tests with input length} \le 10^2, \\ 2/6 & \text{passes all tests with input length} \le 10^3, \\ 3/6 & \text{passes all tests with input length} \le 10^4, \\ 4/6 & \text{passes all tests with input length} \le 10^5, \\ 5/6 & \text{passes all tests with input length} \le 10^6, \\ 1 & \text{passes all tests.} \end{cases} \tag{3}$$

Because the thresholded rewards are nested, this partial-credit objective is exactly the average of the binary rewards from Section 2:

$$R_{\text{shape}}(y) = \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} R_k(y). \tag{4}$$

This identity is the main reason reward shaping behaves like a reweighting of the same static outcome family rather than a new source of information.

**Why shaping stays on the frontier: a KL-regularized lens.** Our actual training uses Dr.GRPO without an explicit KL penalty. To build intuition for why reward shaping does not escape the frontier, we analyze a simplified KL-regularized surrogate. The argument is not a claim about GRPO dynamics; it is a closed-form calculation that illustrates why static shaping recombines existing outcome information rather than introducing new signal.

For a reward $R$ and reference policy $\pi_0$, it is known that the KL-regularized optimal policy with KL penalty $\beta$ is

$$\pi^*(y \mid x) \propto \pi_0(y \mid x) \, \exp\big(R(y)/\beta\big). \tag{5}$$

Write $\pi_k^*$ for the optimum under the single-threshold binary reward $R_k$. Substituting $R_{\text{shape}} = \frac{1}{|\mathcal{K}|} \sum_k R_k$ gives

$$\begin{aligned} \pi^*_{\text{shape}}(y \mid x) &\propto \pi_0(y \mid x) \, \exp\Big(\frac{1}{|\mathcal{K}|\beta} \sum_{k \in \mathcal{K}} R_k(y)\Big) \\ &\propto \Big(\prod_{k \in \mathcal{K}} \pi_k^*(y \mid x)\Big)^{1/|\mathcal{K}|}. \end{aligned} \tag{6}$$

The optimal policy for the shaped reward is therefore the normalized geometric mean of the single-threshold optimal policy, or equivalently the softmax of their arithmetic-mean logits. Because $\pi^*_{\text{shape}}$ is fully determined by the single-threshold optima $\{\pi_k^*\}$, reward shaping recombines the same nested outcome information rather than introducing a genuinely new training signal. This provides intuition for why shaped rewards do not escape the frontier in our experiments, although we note that the geometric mean is not a convex combination in the probability simplex, so a formal convex-hull guarantee does not follow from this analysis alone.

This echoes the geometry of Section 3: weight averaging combines the same family of policies in parameter space, and reward shaping combines them in reward space. These two can be connected if the model weights are under small perturbation that can be linearly approximated.

**Continuous reward shaping.** In addition to the discrete variant above, we test a continuous variant that interpolates linearly in log-space based on the smallest failing test input. Let $s(y)$ denote the character length of the smallest test input on which submission $y$ fails. The continuous shaped reward is

$$R_{\text{cont}}(y) = \begin{cases} 1 & \text{passes all tests,} \\ \dfrac{\log_{10} s(y) - 2}{4} & 10^2 \le s(y) < 10^6, \\ 0 & s(y) < 10^2. \end{cases} \tag{7}$$

This interpolates linearly in log-space from 0 (fails on the smallest inputs) to 1 (passes everything). Both variants remain static functions of the same test outcomes. As Figure 35 shows, both land on the existing trade-off curve, with continuous reward shaping slightly worse. We posit that the scalarization does not matter and what matters is that neither variant reveals why a submission failed.

Figure 34 shows the training dynamics for the two shaped-reward runs.

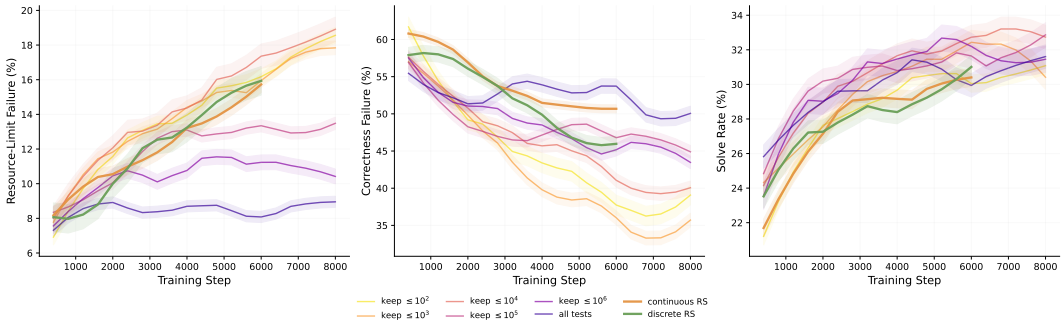



Figure 34: Training dynamics of 32B pure-reasoning setting under reward shaping. Partial credit produces a smoother reward signal but does not improve the final solve rate. The shaped-reward policy converges to the same region of the trade-off curve as the corresponding uniform-weight binary-reward baseline.

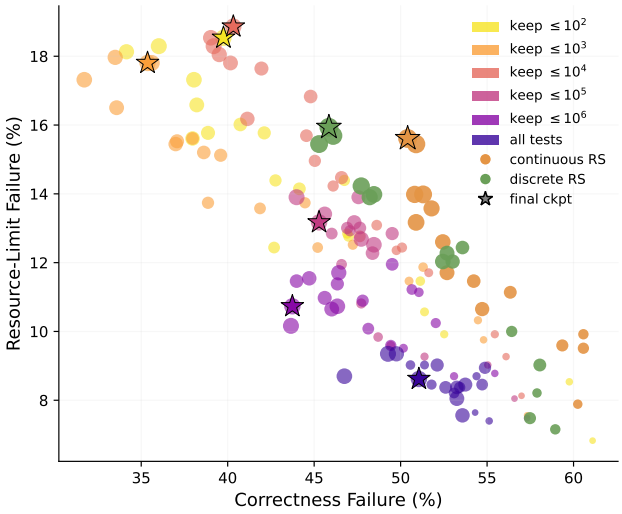



Figure 35: Reward shaping lands on the existing trade-off curve (32B pure reasoning, LCB/hard). Larger dots stand for later stage of training. Baseline single-threshold runs are shown as a color gradient from low coverage to high coverage. Both discrete (green) and continuous (orange) reward shaping checkpoints fall on the same frontier. Stars mark final checkpoints. Reward shaping, whether discrete or continuous, cannot escape the trade-off.

### E.3 Failure-Conditioned Training Experiments

The sections in the main text establish that the trade-off is robust across settings and scales. Longer training can still push the frontier outward, but a sharper question is whether a different use of the same training budget can improve it at matched trainer compute. The *online-fixing* experiments in this appendix address that question by changing how worker-side sampling is spent after failures.

Online-fixing changes two aspects of training at once: it spends more worker-side sampling on problems the model currently fails on, and it augments the online prompt distribution with failure-

conditioned prompts. The *resample-only* control matches the extra retry budget without adding failure context, so it isolates the value of the failure-conditioned prompt itself.

A key property of the design is that it keeps trainer compute fixed while increasing worker compute adaptively after failures. The computation graph is cut at the boundary between the failed rollout and the fixing prompt, so the fixing rollout is treated as a new sample rather than a second turn in the same trajectory. The trainer therefore processes the same number of gradient steps regardless of how many failure-conditioned rollouts are spawned.

### E.3.1 Repair Prompt and Repairability Metric

The repair prompt $z = \phi(x, y)$ concatenates the original problem statement $x$, the failed code $y$, and aggregated execution statistics, along with brief advice on interpreting each failure type. It does not expose private test inputs, expected outputs, or detailed failing traces.

We define fix@$k$ as the pass@$k$ of repair attempts conditioned on a failed original sample. Similar to pass@$k$, we report the aggregate number across all the fixing prompts created from the failing attempts.

### E.3.2 Online-Fixing Objective

For a prompt $x \sim \mathcal{D}$, let $y \sim \pi_\theta(\cdot \mid x)$ be a rollout and $z = \phi(x, y)$ the fixing prompt constructed from $x$ and its failed code $y$. The online-fix objective adds a failure-conditioned correction term to the standard GRPO loss $\mathcal{L}_{\mathrm{GRPO}}$:

$$\mathcal{L}_{\text{online-fixing}}(\theta) = \mathbb{E}_{x\sim\mathcal{D}}\big[\mathcal{L}_{\mathrm{GRPO}}(x;\theta)\big] + \mathbb{E}_{x\sim\mathcal{D},\, y\sim\pi_\theta(\cdot|x)}\big[\mathbf{1}[r(y)=0]\,\mathcal{L}_{\mathrm{GRPO}}(z;\theta)\big]. \quad (8)$$

The *resample-only* control keeps the same second term, but replaces $z = \phi(x, y)$ with a fresh retry prompt that omits the failed code and execution statistics.

The prompt $z$ is constructed using the following template.

**System Prompt**

```
You are a helpful AI assistant. You always reason
before responding, using the following format:

<think>
your internal reasoning
</think>
your external response
```

**User Prompt Template**

```
Solve the following competitive programming
question.

Question:
{problem statement}

For your reference, here is a previous solution
attempt that failed some unit tests:

```python
{failed code}
```

Unit tests execution statistics:
{exec_stats}

You may use this code as reference or inspiration,
but you are free to write a completely new
solution. Make sure to avoid repeating the same
errors. Consider:

- FAILURE or EXCEPTION cases: edge cases,
  boundary values, off-by-one errors, or logic
  flaws.
- TIMEOUT cases: algorithm complexity,
  better data structures, or redundant
  computations.
```

```
Your code should be enclosed in triple backticks
like so: ```python YOUR CODE HERE```. Use
the backticks for your code only.
```

### E.3.3 Effect on the Correctness-Efficiency Frontier

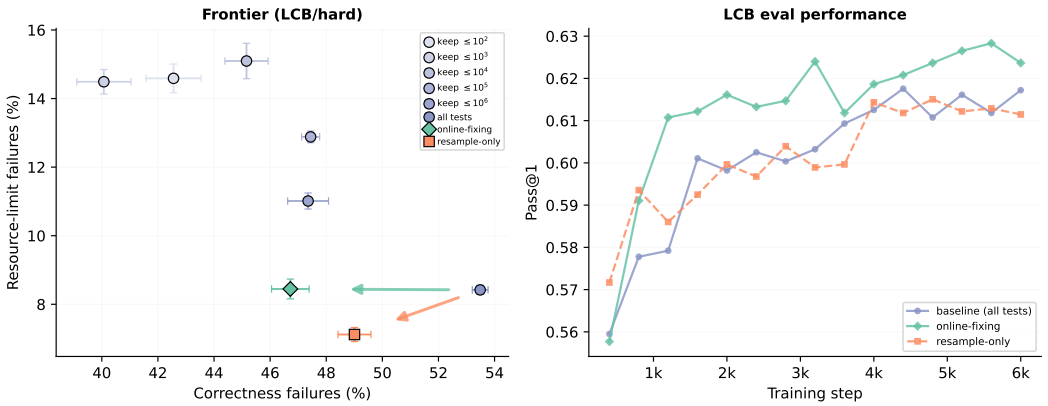



Figure 36: Failure-conditioned correction compared with a compute-matched resample-only control on 32B pure reasoning setting. **Left**: on the original code-generation task, both online-fixing (green diamond) and the resample-only control (orange square) reduce resource-limit failures relative to standard RL checkpoints (circles), but only online-fixing simultaneously reduces correctness failures. Arrows indicate the shift from the no-filter baseline. **Right**: pass@1 on LCB over training for the original code-generation task; the online-fix curve is ahead earlier in training, though the gap narrows later. See Appendix E.3.4 for the full per-setting tables.

### E.3.4 Results Across Settings

The clearest case is 32B pure reasoning: relative to the resample-only control, online fixing raises fix@1 on LCB/hard from 14.2% to 17.9% and pass@1 on full LCB from 61.2% to 62.6% (Table 4). Outside this setting, gains on the original code-generation task are smaller and less consistent. We therefore treat these results as exploratory evidence for failure-conditioned training.

Table 4: Pass rate and self-fixing solve rate for additional worker compute comparison (32B pure-reasoning setting), including online-fixing and its resample-only control, reported on LCB/hard and full LCB. Numbers are averaged from the last three evaluations.

| | LCB/hard | | LCB | | LCB/hard (fix) | | LCB (fix) | |
|---|---|---|---|---|---|---|---|---|
| | pass@1 | pass@5 | pass@1 | pass@5 | fix@1 | fix@5 | fix@1 | fix@5 |
| Baseline | 29.2±0.7 | 46.5±1.4 | 61.5±0.2 | 72.7±0.8 | 14.1±1.4 | 33.5±0.5 | 22.2±3.4 | 43.7±2.9 |
| Resample only | 30.8±0.3 | **48.4**±0.7 | 61.2±0.1 | **73.3**±0.4 | 14.2±0.5 | 34.9±1.3 | 20.8±0.5 | 44.1±0.7 |
| Online-fixing | **30.9**±1.1 | 45.8±0.6 | **62.6**±0.2 | 72.8±0.3 | **17.9**±1.3 | **37.5**±1.0 | **28.1**±1.0 | **49.1**±0.9 |

Table 5: Pass rate and self-fixing solve rate for additional worker compute comparison (7B pure-reasoning setting), including online-fixing and its resample-only control, reported on LCB/hard and full LCB. Numbers are averaged from the last three evaluations.

| | LCB/hard | | LCB | | LCB/hard (fix) | | LCB (fix) | |
|---|---|---|---|---|---|---|---|---|
| | pass@1 | pass@5 | pass@1 | pass@5 | fix@1 | fix@5 | fix@1 | fix@5 |
| Baseline | 13.7±0.6 | 27.4±0.9 | 45.7±0.1 | 59.6±0.3 | 7.4±0.5 | 19.5±0.9 | 15.4±1.9 | 34.2±2.2 |
| Resample only | **14.3**±0.3 | **29.2**±1.0 | 45.6±0.3 | **60.6**±0.7 | 7.9±0.5 | 21.9±1.5 | 17.2±0.9 | 37.2±0.1 |
| Online-fixing | 14.2±1.0 | 28.0±0.3 | **45.9**±0.6 | 60.1±0.1 | **9.7**±0.7 | **22.8**±0.7 | **18.5**±0.6 | **38.2**±0.5 |

Table 6: Pass rate and self-fixing solve rate for additional worker compute comparison (32B tool-interleaved setting), including online-fixing and its resample-only control, reported on LCB/hard and full LCB. Numbers are averaged from the last three evaluations.

| | LCB/hard | | LCB | | LCB/hard (fix) | | LCB (fix) | |
|---|---|---|---|---|---|---|---|---|
| | pass@1 | pass@5 | pass@1 | pass@5 | fix@1 | fix@5 | fix@1 | fix@5 |
| Baseline | 34.6±1.3 | 52.0±1.3 | 64.5±0.7 | 75.6±0.6 | 15.5±0.5 | 36.2±1.9 | 20.3±1.3 | 42.7±2.1 |
| Resample only | 34.7±0.9 | **54.8**±2.5 | 64.3±0.7 | **77.6**±1.4 | 13.4±2.4 | 32.7±2.7 | 19.8±3.3 | 41.7±4.0 |
| Online-fixing | **36.2**±1.0 | 52.9±1.8 | **65.2**±0.4 | 75.9±0.9 | **16.3**±1.5 | **36.3**±1.7 | **21.2**±0.7 | **43.9**±0.6 |

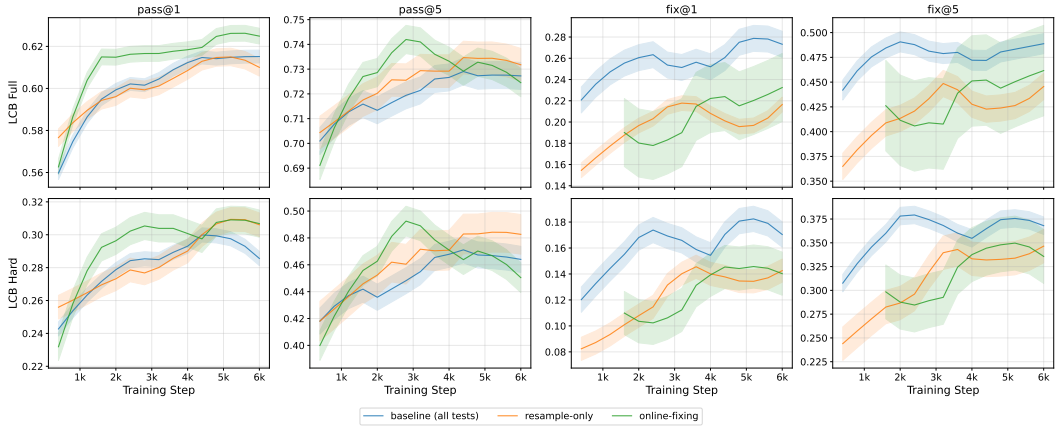



Figure 37: Evaluation curves for the 32B pure-reasoning runs over the first 6k training steps. **Top row**: full LCB evaluation set. **Bottom row**: hard subset only. Columns show pass@1, pass@5, fix@1, and fix@5 respectively. Online-fixing matches or slightly exceeds the baseline and resample-only control on the standard code-generation metrics (pass@1, pass@5), while showing a clear advantage on the self-repair metrics (fix@1, fix@5).

Beyond the original code-generation metrics, online-fix training also strengthens the self-fixing task throughout training. Figure 37 shows that the online-fix model consistently outperforms both the resample-only control and the baseline on fix@1 and fix@5.

# F Case Studies: Full Trajectories Across the Frontier

We present two case studies from the tool-coding setting that concretely illustrate the correctness-efficiency trade-off. Both come from LCB/hard problems that are *differentially solved* across the interpolation axis. For each, we show the *complete multi-turn trajectory* of a representative rollout from the two RL endpoints:

- **Low-coverage** ($\alpha$=0; CodeContestsPlus with test inputs $\leq 10^2$): trained against small-input tests only.
- **High-coverage** ($\alpha$=1; CodeContestsPlus unfiltered): trained against the full test suite including large-input stress tests.

Both checkpoints are evaluated at step 3600 with 5 samples per problem. Thinking blocks are truncated for space; all code and tool output are shown verbatim.

## F.1 Case 1: Falling Bars (hard/770)

**Problem.** Given an $H \times W$ grid with $N$ horizontal bars, simulate gravity: at each time step, bars are processed in index order $i$=$1, \ldots, N$; bar $i$ moves down one row if all cells directly below it are unoccupied. Output each bar's final row at $t$=$10^{100}$. Constraints: $H, W, N \leq 2 \times 10^5$.

**Results.** Low-coverage: **5/5 success.** High-coverage: **0/5 success** (all wrong answer, e.g. `2,3,4,4` instead of `2,4,3,4`).

### F.1.1 Low-Coverage Trajectory (Success)

**Assistant – Turn 1 (python-io): Reverse-order single pass**

```
<think>
... The key insight here is that once a bar stops moving, it
might block other bars from moving. So, the order in which bars
are processed matters. ... One approach is to ... process bars
in reverse index order, tracking the highest occupied row per
column. ...
</think>

<tool: python-io>
<code>
bars.sort(key=lambda x: -x[3])  # reverse index order
occupied = {}  # column -> highest occupied row
result = [0] * N
for R, C, L, idx in bars:
    columns = list(range(C, C + L))
    max_row = H
    for c in columns:
        h = occupied.get(c, 0)
        current_max = h - 1 if h else H
        max_row = min(max_row, current_max)
    result[idx] = max_row
    for c in columns:
        occupied[c] = max(occupied.get(c, 0), max_row)
for row in result:
    print(row)
</code>
<stdin>4 4 4
1 2 3
3 2 2
2 1 2
2 4 1</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]2
3
4
4[/stdout]
```

```
<budget> Remaining turns: 5, Remaining tokens: 45462 </budget>
```

**Turns 2 and 3.** Same reverse-order approach with minor refactors; output remains `2,3,4,4`. Stress test confirms the bug.

**Assistant – Turn 4 (python-io): Forward order with static lookahead**

```
<think>
... The reverse processing approach is fundamentally flawed for
this problem. Let me think differently. Perhaps the correct
approach is to process bars in original order and also check
initial positions of unprocessed bars ...
</think>

<tool: python-io>
<code>
for R, C, L, idx in bars:  # forward order now
    columns = list(range(C, C+L))
    min_max_row = H
    for c in columns:
        current_h = occupied.get(c, 0)
        candidate = current_h - 1 if current_h else H
        min_max_row = min(min_max_row, candidate)
    # Also check initial positions of unprocessed bars
    for j in range(N):
        if j > idx:  # unprocessed (higher index)
            r_j, c_j, l_j, _ = bars[j]
            if r_j > R:  # bar j is initially below
                overlap_start = max(C, c_j)
                overlap_end = min(C + L - 1, c_j + l_j - 1)
                if overlap_start <= overlap_end:
                    min_max_row = min(min_max_row, r_j)
    result[idx] = min_max_row
    ...
</code>
<stdin>4 4 4
1 2 3
3 2 2
2 1 2
2 4 1</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]2
1
1
1[/stdout]

<budget> Remaining turns: 2, Remaining tokens: 39584 </budget>
```

**Assistant – Turn 5 (python-io): Brute-force simulation |←| pivot**

```
<think>
... The code now produces output 2, 1, 1, 1, which is even
worse. Clearly, this approach is incorrect. The issue is that
checking the initial positions of unprocessed bars is not
sufficient, as those bars may move down in subsequent time
steps. ... Given the need to provide a correct solution, I'll
try a direct simulation approach.
</think>

<tool: python-io>
<code>
current_rows = [R for R, _, _, _ in bars]
occupied = [set() for _ in range(H+2)]
for i in range(N):
    R, C, L, _ = bars[i]
    for c in range(C, C+L):
        occupied[R].add(c)
changed = True
while changed:
    changed = False
```

```
    for i in range(N):
        R, C, L, _ = bars[i]
        current_R = current_rows[i]
        if current_R == H:
            continue
        can_move = True
        for c in range(C, C+L):
            if c in occupied[current_R + 1]:
                can_move = False
                break
        if can_move:
            for c in range(C, C+L):
                occupied[current_R].remove(c)
            current_rows[i] += 1
            for c in range(C, C+L):
                occupied[current_R + 1].add(c)
            changed = True
for row in current_rows:
    print(row)
</code>
<stdin>4 4 4
1 2 3
3 2 2
2 1 2
2 4 1</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]2
4
3
4[/stdout]

<budget> Remaining turns: 1, Remaining tokens: 38194 </budget>
```

**Turn 6.** Stress-tests the simulation on $N$=200,000; encounters a `KeyError` on random overlapping bars but the core algorithm is correct.

**Assistant – Final Submission**

```
<think>
The simulation approach produces the correct output for the
sample input. The stress test revealed a KeyError when removing
a column, likely due to overlapping bars in the random test.
Given the time constraints, I'll submit the simulation code.
</think>
```

*(submits the simulation code from Turn 5)*

### F.1.2 High-Coverage Trajectory (Wrong Answer)

**Assistant – Turn 1 (python-io): Segment tree, reverse order**

```
<think>
... I need to find a way to model the dependencies between
bars. ... Let me consider processing bars in reverse index
order, using a segment tree to track the minimum occupied row
for each column range ...
</think>

<tool: python-io>
<code>
import sys, math

def main():
    data = sys.stdin.read().split()
    H, W, N = int(data[0]), int(data[1]), int(data[2])
    bars = []
    idx = 3
    for _ in range(N):
        R, C, L = int(data[idx]), int(data[idx+1]), int(data[idx+2])
```

```
        bars.append((R, C, L)); idx += 3
    result = [0]*N
    size = 1
    while size < W: size <<= 1
    min_row = [math.inf] * (2 * size)
    def update(l, r, val):
        l += size - 1; r += size - 1
        while l <= r:
            if l % 2 == 1: min_row[l] = min(min_row[l], val); l += 1
            if r % 2 == 0: min_row[r] = min(min_row[r], val); r -= 1
            l >>= 1; r >>= 1
    def query(l, r):
        res = math.inf; l += size - 1; r += size - 1
        while l <= r:
            if l % 2 == 1: res = min(res, min_row[l]); l += 1
            if r % 2 == 0: res = min(res, min_row[r]); r -= 1
            l >>= 1; r >>= 1
        return res
    for i in reversed(range(N)):
        R, C, L = bars[i]
        current_min = query(C, C + L - 1)
        new_R = min(H, current_min - 1) if current_min != math.inf else H
        result[i] = new_R
        update_val = new_R + 1 if new_R < H else math.inf
        update(C, C + L - 1, update_val)
    for r in result: print(r)
main()
</code>
<stdin>4 4 4
1 2 3
3 2 2
2 1 2
2 4 1</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]4
4
4
4[/stdout]

<budget> Remaining turns: 5, Remaining tokens: 46198 </budget>
```

**Turn 2** (`python-io`). Fixes segment-tree indexing to 0-based; output unchanged: `4,4,4,4`.

**Assistant – Turn 3 (python-stress): Assertion failure**

```
<think>
... The output is still incorrect. All bars are at row 4, but
the correct output should be 2, 4, 3, 4. The issue must be in
the segment tree implementation or the logic of updating ...
</think>

<tool: python-stress>
... (same reverse-order segment tree code)
result = solve(H, W, N, bars)
print("Result:", result)
print("Expected:", expected)
assert result == expected, f"Test failed: {result} != {expected}"
</tool>
```

**Tool Response**

```
completed. [stdout]Result: [4, 4, 4, 4]
Expected: [2, 4, 3, 4][/stdout]
[stderr]AssertionError: Test failed: [4, 4, 4, 4] != [2, 4, 3, 4][/stderr]
```

**Assistant – Turn 4 (python-io): Plain array, still reverse order**

```
<think>
... The stress test confirms that the current approach is
incorrect. ... Let me simplify and use a plain array instead of
a segment tree, but keep the reverse processing order ...
</think>

<tool: python-io>
<code>
min_row = [H+1] * (W + 2)  # 1-based
for i in reversed(range(N)):
    R, C, L = bars[i]
    current_min = min(min_row[C : C+L])
    if current_min == H+1:
        new_R = H
    else:
        new_R = min(H, current_min - 1)
    result[i] = new_R
    for c in range(C, C+L):
        if min_row[c] > new_R:
            min_row[c] = new_R
</code>
<stdin>4 4 4
1 2 3
3 2 2
2 1 2
2 4 1</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]2
3
4
4[/stdout]

<budget> Remaining turns: 2, Remaining tokens: 40695 </budget>
```

**Turn 5 (`python-io`).** Changes update value to `new_R + 1`; output regresses to `4,4,4,4`.

**Turn 6 (`python-io`).** Reverts to `new_R`; output returns to `2,3,4,4`, which is still wrong. The model is stuck cycling between two incorrect variants.

**Assistant – Final Submission**

```
<think>
... The output is still [2, 3, 4, 4], which is close but not
correct. Bar 2 should be at row 4, not row 3. The fundamental
issue is in how the reverse processing handles the interaction
between bars ...
</think>
```

*(submits the reverse-order min_row code from Turn 4; wrong answer)*

### F.1.3 Interpretation

Both models initially try the same efficient reverse-processing idea. The low-coverage model *pivots* to brute-force simulation when it fails (Turn 5); the high-coverage model spends all 6 tool turns on variants of the single-pass approach, never attempting simulation. The low-coverage model's willingness to fall back to a correct-but-slow algorithm is the difference between 5/5 success and 0/5.

## F.2 Case 2: Largest Palindrome Divisible by $K$ (hard/680)

**Problem.** Given $n$ and $k$ ($1 \leq n \leq 10^5$, $1 \leq k \leq 9$), find the largest $n$-digit palindrome divisible by $k$.

**Results.** Low-coverage: **0/5 success** (all timeout). High-coverage: **1/5 success** (3 timeout, 1 wrong answer).

### F.2.1 Low-Coverage Trajectory (Timeout)

**Assistant – Turn 1 (python-io): Brute-force decrement**

```
<think>
... The task is to find the largest n-digit number that is both
a palindrome and divisible by k. ... Start from the largest
palindrome and work downward until we find one divisible by k.
</think>

<tool: python-io>
<code>
class Solution:
    def largestPalindrome(self, n: int, k: int) -> str:
        m = (n + 1) // 2
        first_half = '9' * m
        while True:
            if n % 2 == 0:
                palindrome = first_half + first_half[::-1]
            else:
                palindrome = first_half + first_half[:-1][::-1]
            if self.is_divisible(palindrome, k):
                return palindrome
            new_first_half = self.decrement_first_half(first_half)
            if new_first_half is None:
                return ""
            first_half = new_first_half

    def is_divisible(self, s, k):
        mod = 0
        for c in s:
            mod = (mod * 10 + int(c)) % k
        return mod == 0

    def decrement_first_half(self, s):
        s_list = list(s)
        i = len(s_list) - 1
        carry = 1
        while i >= 0 and carry:
            digit = int(s_list[i])
            if digit - carry >= 0:
                s_list[i] = str(digit - carry)
                carry = 0
            else:
                s_list[i] = '9'
                carry = 1
            i -= 1
        if s_list[0] == '0':
            return None
        return ''.join(s_list)
</code>
<stdin>3
5</stdin>
</tool>
```

**Tool Response**

```
completed.

<budget> Remaining turns: 5, Remaining tokens: 51467 </budget>
```

**Turn 2.** Tests second example (`n=1,k=4`); passes with output `"8"`.

**Assistant – Turn 3 (python-stress): Large input test**

```
<think>
... Let me test with large inputs to check performance. For
n=100000, the first_half has 50000 digits ...
</think>
```

```
<tool: python-stress>
...
test_cases = [(3, 5), (1, 4), (5, 6), (2, 2), (100000, 9)]
for n, k in test_cases:
    start = time.time()
    result = Solution().largestPalindrome(n, k)
    elapsed = time.time() - start
    print(f"n={n}, k={k}: time={elapsed:.3f}s")
</tool>
```

**Tool Response**

```
error: EXECUTION_TIMEOUT after 60 seconds
```

**Assistant – Final Submission**

```
<think>
... The stress test timed out for n=100000. The brute-force
approach is too slow for large inputs. But given the time
constraints, I'll submit this solution.
</think>
```

*(submits brute-force decrement code; timeout on hidden tests)*

### F.2.2 High-Coverage Trajectory (Success)

**Assistant – Turn 1 (python-io): Optimized remainder, still brute-force**

```
<think>
... Let me start by considering the structure of the palindrome.
For an n-digit number, each half-digit contributes to two
symmetric positions. I can precompute the coefficient of each
position modulo k to speed up the divisibility check ...
</think>

<tool: python-io>
<code>
class Solution:
    def largestPalindrome(self, n: int, k: int) -> str:
        if k == 1: return '9' * n
        m = (n + 1) // 2 if n % 2 else n // 2
        pow10_mod = [1] * n
        for i in range(1, n):
            pow10_mod[i] = (pow10_mod[i-1] * 10) % k
        def get_remainder(s):
            remainder = 0
            for i in range(len(s)):
                digit = int(s[i])
                if n % 2 == 0 or i < len(s) - 1:
                    coeff = (pow10_mod[n-1-i] + pow10_mod[i]) % k
                else:
                    coeff = pow10_mod[n-1-i] % k
                remainder = (remainder + digit * coeff) % k
            return remainder
        first_half = '9' * m
        while True:
            if get_remainder(first_half) == 0:
                if n % 2:
                    return first_half + first_half[:-1][::-1]
                else:
                    return first_half + first_half[::-1]
            s_list = list(first_half)
            i = len(s_list) - 1
            while i >= 0 and s_list[i] == '0':
                s_list[i] = '9'; i -= 1
            if i >= 0: s_list[i] = str(int(s_list[i]) - 1)
            first_half = ''.join(s_list)
            if first_half[0] == '0': return "0" * n
</code>
<stdin>3
5</stdin>
</tool>
```

**Tool Response**

```
completed.

<budget> Remaining turns: 5, Remaining tokens: 48438 </budget>
```

**Assistant – Turn 2 (python-stress): Discovers timeout**

```
<think>
... Let me test with large inputs to verify performance ...
</think>

<tool: python-stress>
test_cases = [(10**5, 9), (10**5, 2), (10**5, 5)]
for n, k in test_cases:
    start = time.time()
    result = Solution().largestPalindrome(n, k)
    elapsed = time.time() - start
    print(f"n={n}, k={k}  Time:  elapsed:.3fs")</tool>
```

**Tool Response**

```
error: EXECUTION_TIMEOUT after 60 seconds

<budget> Remaining turns: 4, Remaining tokens: 46406 </budget>
```

**Assistant – Turn 3 (python-io): DP with backward reachability |←| pivot**

```
<think>
... The stress test timed out for large n=1e5. The main issue
is the decrement loop -- for large n, the first_half can be up
to 5e4 digits, and decrementing each time is O(m). We need a
fundamentally different approach. ...

Since k <= 9, the remainder modulo k has at most 9 possible
values. For each half-digit position, we can precompute its
coefficient mod k. Then use DP: compute which remainders are
reachable from each suffix of the half-string, and greedily
assign the largest valid digit at each position.
</think>

<tool: python-io>
<code>
class Solution:
    def largestPalindrome(self, n: int, k: int) -> str:
        if k == 1: return '9' * n
        m = (n + 1) // 2 if n % 2 else n // 2
        pow10_mod = [1] * n
        for i in range(1, n):
            pow10_mod[i] = (pow10_mod[i-1] * 10) % k
        coeff = []
        for i in range(m):
            if n % 2 == 0 or i < m - 1:
                c = (pow10_mod[n-1-i] + pow10_mod[i]) % k
            else:
                c = pow10_mod[n-1-i] % k
            coeff.append(c)
        # Backward DP: which remainders are reachable?
        dp_backward = [set() for _ in range(m + 1)]
        dp_backward[m].add(0)
        for i in range(m-1, -1, -1):
            current_coeff = coeff[i]
            for d in range(10):
                if i == 0 and d == 0: continue
                contrib = d * current_coeff
                for r in dp_backward[i+1]:
                    dp_backward[i].add((r + contrib) % k)
        # Greedy forward pass
        first_half = []
        current_remainder = 0
        for i in range(m):
            for d in range(9, -1, -1):
                if i == 0 and d == 0: continue
                contrib = d * coeff[i]
```

```
                target = (-(current_remainder + contrib)) % k
                if target in dp_backward[i+1]:
                    first_half.append(str(d))
                    current_remainder = (current_remainder + contrib) % k
                    break
        first_half_str = ''.join(first_half)
        if n % 2:
            return first_half_str + first_half_str[:-1][::-1]
        else:
            return first_half_str + first_half_str[::-1]
</code>
<stdin>3
5</stdin>
</tool>
```

**Tool Response**

```
completed.

<budget> Remaining turns: 3, Remaining tokens: 41605 </budget>
```

**Assistant – Turn 4 (python-stress): Efficiency confirmed**

```
<think>
... Now let's test the DP solution with large inputs to
confirm it runs within the time limit ...
</think>

<tool: python-stress>
test_cases = [(10**5, 9), (10**5, 2), (10**5, 5)]
for n, k in test_cases:
    start = time.time()
    result = Solution().largestPalindrome(n, k)
    elapsed = time.time() - start
    print(f"n={n}, k={k}  Time:  elapsed:.3fs  PASS")
</tool>
```

**Tool Response**

```
completed. [stdout]n=100000, k=9  Time:  0.557s  PASS
n=100000, k=2  Time:  0.158s  PASS
n=100000, k=5  Time:  0.152s  PASS[/stdout]
[performance] Execution Time: 0.959s, Peak Memory: 55.1 MB

<budget> Remaining turns: 2, Remaining tokens: 39572 </budget>
```

**Turns 5 and 6.** Tests small examples (`n=1,k=4`; `n=3,k=5`; `n=5,k=6`) to verify correctness; all pass.

**Assistant – Final Submission**

```
<think>
Solution passes all tests and runs well within the time limit.
</think>

(submits the DP code from Turn 3; passes all hidden tests)
```

### F.2.3 Interpretation

Both models start with brute-force and both discover the timeout at $n{=}10^5$. The critical difference is what happens *after*: the low-coverage model has already exhausted its tool budget on minor variations and submits the brute-force code; the high-coverage model invests its remaining turns in developing a DP formulation that reduces complexity from $O(n \cdot 10^m/k)$ to $O(m \cdot k)$.

## F.3 Discussion

These two trajectories make the frontier concrete:

- **hard/770 (Falling Bars):** Both models initially try efficient reverse-processing. The low-coverage model *pivots* to brute-force simulation when it fails; the high-coverage model stays committed to the efficient-but-wrong approach.
- **hard/680 (K-Palindrome):** Both models start with brute-force and discover it times out. The high-coverage model *pivots* to dynamic programming; the low-coverage model submits the slow code.

The pattern is symmetric: each model is more likely to *abandon a failing strategy* in the direction its training reward incentivizes. The low-coverage model, never penalized for timeouts during training, readily falls back to correct-but-slow algorithms, while the high-coverage model, trained against full stress tests, readily develops efficient algorithms even when correctness is uncertain. Neither behavior dominates across all problems, which is why interpolation and extrapolation can access solutions that no single endpoint produces.

## G Example Trajectories from Tool-Use Training

We show one representative trajectory from each iteration of the tool-use training (Appendix A.5) to illustrate the tool-use patterns the model acquires. Thinking blocks are truncated for space; all code and tool output are shown verbatim.

**Iteration 0: testing with `python-io`.** The system prompt gives the model access to `python` and `python-io` tools with a budget of 5 tool turns. The following trajectory solves a phone-keypad cipher problem.

**System Prompt (Iteration 0)**

```
You are an expert Python programmer and tool user. You have access to
"python" and "python-io" tools. You are constrained by a 61440 total
context window limit. You can interact with these tools through the
following tool interface for at most 5 turns.

<tool: python>
Run Python code in an isolated sandbox.
</tool>

<tool: python-io>
Run Python code with stdin input in an isolated sandbox.
Use <code>...</code> for code and <stdin>...</stdin> for input.
</tool>

You will receive a budget notice after each tool execution indicating
your remaining turns and tokens.
```

**User**

```
Provide a Python solution for the following competitive programming
question: Solve The Mystery

Input:
First line contains T - number of test cases.
Following T lines each contains a string of characters in [a-z] only

Output: Print a numeric string for each test case.

Constraints: 1 <= T <= 100, 1 <= Length of String <= 100

SAMPLE INPUT            SAMPLE OUTPUT
10                      539
key                     86
to                      2668765
control                 6463
nine                    82457
tails                   47
is                      46
in                      9687
your                    4263
hand                    74663
phone
```

**Assistant – Turn 1 (python-io)**

```
<think>
I need to map each letter to its phone keypad digit. Looking at the
sample: k->5, e->3, y->9 gives "539". This is the standard phone
keypad mapping: abc->2, def->3, ghi->4, jkl->5, mno->6, pqrs->7,
tuv->8, wxyz->9.
</think>

<tool: python-io>
<code>
keypad = {
    'a': '2', 'b': '2', 'c': '2',
    'd': '3', 'e': '3', 'f': '3',
    'g': '4', 'h': '4', 'i': '4',
    'j': '5', 'k': '5', 'l': '5',
    'm': '6', 'n': '6', 'o': '6',
    'p': '7', 'q': '7', 'r': '7', 's': '7',
    't': '8', 'u': '8', 'v': '8',
    'w': '9', 'x': '9', 'y': '9', 'z': '9'
}
t = int(input())
for _ in range(t):
    s = input().strip()
    print(''.join(keypad[c] for c in s))
</code>
<stdin>
10
key
to
control
nine
tails
is
in
your
hand
phone
</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]539
86
2668765
6463
82457
47
46
9687
4263
74663[/stdout]

<budget> Remaining turns: 4, Remaining tokens: 59534 </budget>
```

**Assistant – Turn 2 (python-io): Edge cases**

```
<think>
The sample output matches. Let me test edge cases: 4-letter groups
(pqrs, wxyz), single characters, and all-same-group strings.
</think>

<tool: python-io>
<code>
keypad = { ... }  # same dictionary as above
t = int(input())
for _ in range(t):
    s = input().strip()
    print(''.join(keypad[c] for c in s))
</code>
<stdin>
5
aaa
pqrs
wxyz
```

```
a
z
</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]222
7777
9999
2
9[/stdout]

<budget> Remaining turns: 3, Remaining tokens: 59081 </budget>
```

**Assistant – Final Submission**

````
<think>
All edge cases pass. The solution handles all constraints correctly.
</think>

```python
keypad = {
    'a': '2', 'b': '2', 'c': '2',
    'd': '3', 'e': '3', 'f': '3',
    'g': '4', 'h': '4', 'i': '4',
    'j': '5', 'k': '5', 'l': '5',
    'm': '6', 'n': '6', 'o': '6',
    'p': '7', 'q': '7', 'r': '7', 's': '7',
    't': '8', 'u': '8', 'v': '8',
    'w': '9', 'x': '9', 'y': '9', 'z': '9'
}
t = int(input())
for _ in range(t):
    s = input().strip()
    print(''.join(keypad[c] for c in s))
```
````

**Iteration 1: stress-testing with `python-stress`.** The system prompt adds the `python-stress` tool, which runs code with automatic timing and memory measurement. The following trajectory solves a prefix-sum divisibility counting problem ($N \leq 10^5$, time limit 2 s).

**System Prompt additions (Iteration 1)**

```
You have access to "python", "python-io" and "python-stress" tools.

<tool: python-stress>
Run Python code with performance testing and automatic output
suppression to save context. This tool automatically measures
execution time and memory usage.
IMPORTANT: Output is limited to the last 50 lines. Use this for
efficiency testing with large inputs. You MUST generate test cases
programmatically inside the code - do not use <stdin>.
DO NOT print large data structures - only print performance metrics.
</tool>
```

**User**

```
Provide a Python solution for the following competitive programming
question. Time Limit: 2.0 s. Memory Limit: 1073 MB.

There are N boxes arranged in a row from left to right. The i-th box
from the left contains A_i candies. You will take out the candies
from some consecutive boxes and distribute them evenly to M children.

Find the number of pairs (l, r) such that 1 <= l <= r <= N and
A_l + A_{l+1} + ... + A_r is a multiple of M.

Constraints: 1 <= N <= 10^5, 2 <= M <= 10^9, 1 <= A_i <= 10^9

Examples:
```

```
  3 2 / 4 1 5          -> 3
  13 17 / 29 7 ... 81  -> 6
  10 400000000 / 10^9 x10 -> 25
```

**Assistant – Turn 1 (python-io): Parse input**

```
<think>
This is a classic prefix-sum modulo problem. First let me validate
the input parsing.
</think>

<tool: python-io>
<code>
n, m = map(int, input().split())
a = list(map(int, input().split()))
print(f"N: {n}, M: {m}, Array: {a}")
</code>
<stdin>
3 2
4 1 5
</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]N: 3, M: 2, Array: [4, 1, 5][/stdout]

<budget> Remaining turns: 4, Remaining tokens: 61155 </budget>
```

**Assistant – Turn 2 (python): Core algorithm test**

```
<think>
Now test the prefix-sum-modulo + frequency-counting approach.
</think>

<tool: python>
def solve():
    n, m = 3, 2
    a = [4, 1, 5]
    count = 0
    mod_freq = {0: 1}
    prefix_sum_mod = 0
    for num in a:
        prefix_sum_mod = (prefix_sum_mod + num) % m
        if prefix_sum_mod in mod_freq:
            count += mod_freq[prefix_sum_mod]
        mod_freq[prefix_sum_mod] = mod_freq.get(
            prefix_sum_mod, 0) + 1
    print(f"Count: {count}")
    assert count == 3
solve()
</tool>
```

**Tool Response**

```
completed. [stdout]Count: 3[/stdout]

<budget> Remaining turns: 3, Remaining tokens: 60885 </budget>
```

**Assistant – Turn 3 (python-io): Full solution test**

```
<think>
Core logic works. Now test the full solution with stdin.
</think>

<tool: python-io>
<code>
def solve():
    n, m = map(int, input().split())
    a = list(map(int, input().split()))
    count = 0
```

```
    mod_freq = {0: 1}
    prefix_sum_mod = 0
    for num in a:
        prefix_sum_mod = (prefix_sum_mod + num) % m
        if prefix_sum_mod in mod_freq:
            count += mod_freq[prefix_sum_mod]
        mod_freq[prefix_sum_mod] = mod_freq.get(
            prefix_sum_mod, 0) + 1
    print(count)
solve()
</code>
<stdin>
3 2
4 1 5
</stdin>
</tool>
```

**Tool Response**

```
completed. [stdout]3[/stdout]

<budget> Remaining turns: 2, Remaining tokens: 60663 </budget>
```

**Assistant – Turn 4 (python-stress): Performance test**

```
<think>
Solution is correct. Now stress-test at scale to confirm O(n)
performance within the 2s time limit.
</think>

<tool: python-stress>
import random, time

def solve(n, m, arr):
    count = 0
    mod_freq = {0: 1}
    prefix_sum_mod = 0
    for num in arr:
        prefix_sum_mod = (prefix_sum_mod + num) % m
        if prefix_sum_mod in mod_freq:
            count += mod_freq[prefix_sum_mod]
        mod_freq[prefix_sum_mod] = mod_freq.get(
            prefix_sum_mod, 0) + 1
    return count

print("PERFORMANCE PROFILE")
print("=" * 60)
for size in [10**3, 10**4, 10**5]:
    arr = [random.randint(1, 10**9) for _ in range(size)]
    start = time.time()
    solve(size, 10**9, arr)
    elapsed = time.time() - start
    print(f"n={size:>8}  Time:  elapsed:>6.3fs  PASS")
print("=" * 60)
</tool>
```

**Tool Response**

```
completed. [stdout]PERFORMANCE PROFILE
============================================================
n=    1000  Time:  0.000s  PASS
n=   10000  Time:  0.004s  PASS
n=  100000  Time:  0.048s  PASS
============================================================[/stdout]
[performance] Execution Time: 0.277s, Peak Memory: 25.3 MB

<budget> Remaining turns: 1, Remaining tokens: 60169 </budget>
```

**Assistant – Final Submission**

```
<think>
Solution passes all tests and runs well within the 2s time limit
at n=10^5. Submit with sys.stdin.read() optimization.
```

````
</think>

```python
def solve():
    import sys
    input = sys.stdin.read
    data = input().split()
    n = int(data[0])
    m = int(data[1])
    a = list(map(int, data[2:2+n]))
    count = 0
    mod_freq = {0: 1}
    prefix_sum_mod = 0
    for num in a:
        prefix_sum_mod = (prefix_sum_mod + num) % m
        if prefix_sum_mod in mod_freq:
            count += mod_freq[prefix_sum_mod]
        mod_freq[prefix_sum_mod] = mod_freq.get(
            prefix_sum_mod, 0) + 1
    print(count)
solve()
```
````

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three specific claims (frontier from nested coverage, extrapolative continuation, inference-time ensemble gains) that are directly supported by Sections 3–4 with corresponding figures and quantitative results. The scope is explicitly delimited to competitive programming in Section 6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6 contains a dedicated "Limitations" paragraph discussing the restriction to competitive programming, the coarseness of input-length thresholds as a proxy, and the open question of mechanistic understanding.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include formal theoretical results. The KL-regularized derivation in Appendix E.2 is explicitly presented as simplified intuition, not a theorem.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: All models are based on openly available base models (Qwen 2.5 7B, CWM-SFT 32B). Training uses open datasets (CodeContestsPlus, OpenCodeReasoning, OpenMathReasoning) and a public evaluation benchmark (LiveCodeBench). Training hyperparameters, RL algorithm details, and the tool-call curriculum are described in Section 2 and Appendices A.2–A.5. To the best of our knowledge, no non-disclosed information hinders reproduction.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: We do not release training code. However, all base models, training datasets, and evaluation benchmarks used are publicly available and properly cited. Training details and hyperparameters are provided in Appendix A to facilitate independent reproduction.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 2 specifies the training family, reward definitions, evaluation benchmark, and outcome taxonomy. Appendix A provides the full training details for both 7B and 32B models, including the tool-call curriculum (Appendix A.5) and hyperparameters. Appendix A.6 gives a dedicated compute-resources summary.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

Justification: For the main frontier results, rather than reporting aggregate statistics with error bars, we display checkpoints sampled throughout training across six coverage thresholds and three independent inference settings (18 RL runs for 32B), allowing readers to visually assess robustness. For the online-fixing tables (Appendix E.3.4), standard deviations are reported from the last three evaluation checkpoints. For the inference-time ensemble results (Section 4.2), shaded regions in Figure 11 show $\pm 1$ standard deviation from a subsampling bootstrap (500 iterations, resampling $k$ attempts from the available pool per problem). The 3.3% pass@250 ensemble gain is reported with a 95% confidence interval obtained via a problem-level paired bootstrap: we resample the 123 hard problems with replacement for 10,000 iterations, compute the paired pass@250 difference in each replicate, and report the 2.5th/97.5th percentile interval.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix A.6 provides a dedicated compute-resources table listing GPU counts, distributed configuration, wall-clock time per run, and total GPU-hours for every training setting, and discloses that the full research project consumed roughly $3\times$ the reported compute.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

Justification: The research studies training dynamics of code-generation models using publicly available datasets and benchmarks. It does not involve human subjects, private data, or deceptive practices.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: This is foundational research studying the training dynamics and trade-offs of RL for code generation. We do not foresee direct negative societal applications beyond the general dual-use concerns of code-generation models, which are not specific to this work.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release models, datasets, or other artifacts.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.

- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets (CodeContestsPlus, OpenCodeReasoning, OpenMathReasoning), evaluation benchmarks (LiveCodeBench), and base models (Qwen 2.5 7B, CWM-SFT) are cited with their original publications. These assets are publicly available under their respective open-source licenses.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not release new datasets, models, or code.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Claude-4.5-Sonnet is used as an LLM judge in Stage 2 of the error reclassification protocol (Appendix B.1) to classify whether timed-out submissions contain correctness bugs. This usage is described in the Appendix B.1 and does not affect the main results, which use raw sandbox labels throughout.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.