Reinforcement Learning for Code Optimization

Pierre Chambon[1,2] **Kunhao Zheng**[1,3] **Juliette Decugis**[1,2] **Benoit Sagot**[2] **Gabriel Synnaeve**[1]
[1]FAIR at Meta [2]Inria [3]Université Paris-Dauphine PSL
pchambon@meta.com

Abstract

RL for code correctness is now established: have the model generate a program, run it against hidden test cases, and reward solutions that pass. Extending this to code optimization seems straightforward: just add execution time to the reward. But in practice, once timing drives the reward, small problems in measurement noise, reward sparsity, or Group Relative Policy Optimization (GRPO) instability overwhelm the signal and make RL fail: generated solutions are barely faster, and more of them can fail. We make execution time learnable through three stages: (1) how code is tested, by building DMC-Optim with large optimization tests and a calibrated sandbox; (2) how speed is turned into reward, by composing correctness and speed in the RL environment and using an offline simulator to predict the most promising configurations; and (3) how the model learns from that reward, by adapting GRPO and evaluation to the sparser, noisier timed-execution setting. On DMC-Optim, the strongest optimization-aware configurations improve pass@1 under a top-50% speed threshold from 18.0% to 31.3% on Qwen 2.5 7B and from 30.7% to 50.4% on CWM 32B. Under a top-30% threshold, these gains further increase, with CWM 32B showing 125% relative improvement, while pure-correctness scores remain stable. When the timing sandbox is degraded, robust optimization RL reaches roughly 100%–200% improvement over standard RL with verifiable rewards (RLVR), depending on the evaluation criterion. On LiveCodeBench (LCB), CWM 32B wins up to 83% of median-sample speed comparisons against standard RLVR. Relative to the fastest correct human submissions per problem, it reaches about half the human rate of complexity-class improvements (13% vs. 22%).

1 Introduction

Training for correctness has not made code models reliably fast. On SWE-fficiency (Ma et al., 2025), Claude 4.5 Sonnet produces correct patches 81% of the time but captures only 4.1% of expert speedup. Repository scale and expensive execution do not fully explain this: the gap also appears in competitive programming, where cheap tests and ample human submissions provide speed baselines. On VENUS, o4-mini reaches 89.1% pass@1 but scores only 56.9% on BEYOND-T, a runtime-efficiency percentile against human references (Du et al., 2025), and Qwen3-32B drops from 70.0% pure-correctness pass@1 to 43.5% under a best-complexity-class requirement on BigO(Bench) (Chambon et al., 2025).

This more controlled competitive-programming setting lets us isolate where optimization RL fails. In standard reinforcement learning with verifiable rewards (RLVR) for code, execution is mostly a binary verifier: a sample is rewarded if it passes the tests. In optimization RL, execution must also measure how fast a correct program runs on the given tests, so timing noise, weak tests, and rewards that give speed credit to wrong code can corrupt the training signal. Evaluation raises the same coupling problem between correctness and speed. For evaluation, we introduce p_τ: a solution counts only if it is correct and no slower than the τ-th-percentile of the human reference leaderboard, with smaller τ stricter. This evaluation confirms that the naive optimization training objective is

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

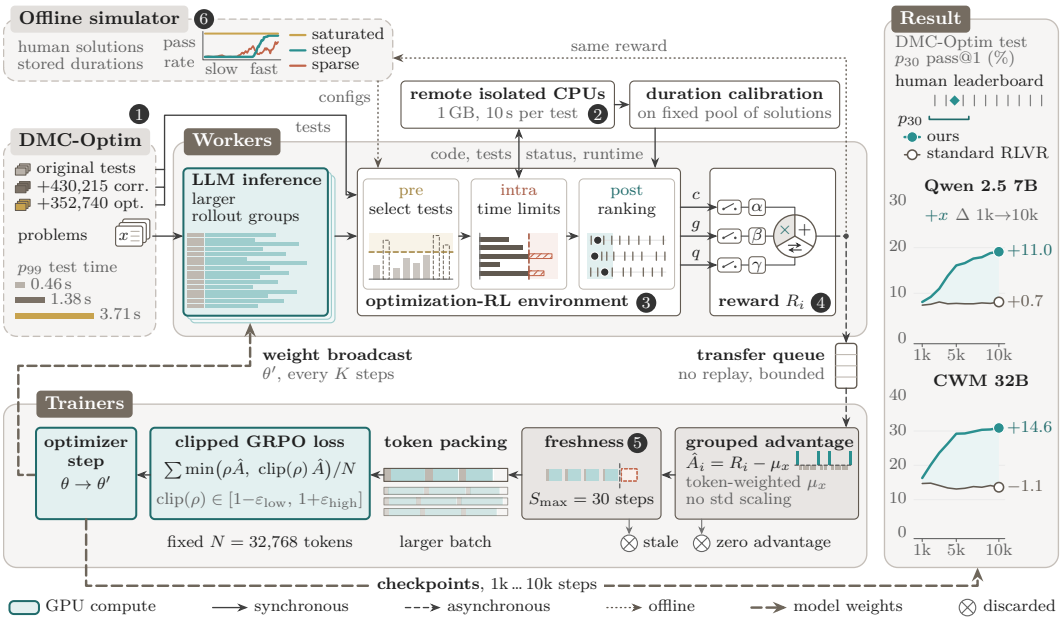


Figure 1: **Making execution time learnable for RL.** A DMC-Optim problem is sampled and the LLM generates candidate solutions. All RL environments we introduce share the same evaluation pipeline, but differ in where they inject optimization pressure. First, each candidate solution can be evaluated on standard correctness tests, but also generated large-input optimization tests (1); pre-execution environments may filter optimization tests in or out using test-level criteria. Execution of a solution on these tests is then run on a calibrated remote execution service (2); intra-execution environments apply additional per-test time limits. Finally, post-execution environments can leverage the returned recorded per-test durations for instance to compare them against calibrated human reference distributions. The environment (3) returns whether all correctness tests pass (c), whether the optimization tests pass (g), and a potential aggregated scalar quality score q. The three signals, correctness gate c, optimization gate g, and quality score q, feed into one of several reward compositions (4). An offline RL simulator (6, top left) evaluates candidate environments by replaying reference human solutions, and the most promising environment-reward pairs are used in properly tuned online asynchronous GRPO training (5, bottom), where the best optimization-RL setting reaches strict DMC-Optim pass@1 at p_{30} of 19.1% on Qwen 2.5 7B and 30.9% on CWM 32B after 10k steps, vs 7.7% and 13.7% for standard RLVR.

brittle: rewarding lower average runtime on top of a binary correctness reward changes pass@1 by only $+0.6$ points at p_{30}, whatever the reward-range mapping is, and by -0.3 to $+1.4$ points at p_{100}, which is equivalent to pure correctness. The question is what makes an execution-time signal learnable.

Optimization RL fails through a chain of transformations. The timing source must separate solutions (Section 3); the environment and reward must choose tests, time limits, reference comparisons, and how rewards trade correctness against speed without incentivizing fast-but-wrong code (Section 4); and the optimizer must remain stable under rewards that are sparser and noisier than pass/fail (Section 5). If any link fails, generated programs are not faster, and correctness can also degrade.

We make this chain explicit in Fig. 1. From the DeepMind Code Contests (DMC) corpus (Li et al., 2022), we build DMC-Optim with 2,723 cleaned problems, separating correctness tests from larger optimization tests; 1,302 problems have enough duration spread for timing-based rewards. We sweep pre-execution filtering, intra-execution time constraints, post-execution ranking against human references, and rewards from optimization-only/additive blends to multitask, collapsed, and hard-gated variants. An offline simulator screens this space before online Group Relative Policy Optimization (GRPO) (Shao et al., 2024).

This stack gives consistent in-domain gains across three starts: Qwen 2.5 7B/32B base checkpoints (Yang et al., 2025), which we SFT on a decontaminated reasoning-only mix, and the released CWM 32B SFT checkpoint (FAIR CodeGen team et al., 2025). On DMC-Optim pass@1 at p_{50}, Qwen 2.5 7B/32B and CWM 32B rise from 18.0/21.1/30.7% to 31.3/39.6/50.4%; at p_{30}, CWM 32B

rises from 13.7% to 30.9%, a 125% relative gain, while pure-correctness scores stay stable. On LiveCodeBench (LCB) (Jain et al., 2024), where timeout and percentile scores are unreliable, pairwise speed win rates show CWM 32B reaching up to 83.0% median-sample wins against standard RLVR while retaining 54.4% pass@1. Overall, our contribution is a recipe and its empirical validation for online RL training of one-shot code generators that achieve significant gains in the wall-clock efficiency of generated code, while preserving correctness capabilities on DMC-Optim (or limiting the relative pass@1 decrease to 4–8% on out-of-domain LCB for the common top-30% environment); our work is a comparative study in the online RL setting exclusively.

Our contributions are threefold:

- We build DMC-Optim and its timing stack: 2,723 cleaned problems from 12,275 raw DMC problems, 1,302 duration-filterable problems, 430,215 new correctness tests, 352,740 new optimization tests, and final-split p_{95} duration raised from $0.145\,s$ to $1.296\,s$ under calibrated remote execution.
- We explore optimization-aware RL: pre-execution test filtering, intra-execution time constraints, post-execution ranking, reward variants from additive blends to multitask/collapsed/gated forms, offline simulator screening, and GRPO adaptations for sparse timing rewards; we do not claim a new RL algorithm.
- We show optimization gains across evaluations: DMC-Optim p_{50} pass@1 improves by +13.3/+18.5/+19.7 points on Qwen 7B/Qwen 32B/CWM 32B; CWM 32B reaches an 83.0% LCB median-sample speed win rate against standard RLVR; and under degraded sandbox state, robust optimization RL improves over RLVR by roughly 100%–200%, depending on the evaluation criterion.

2 Related Work

Execution feedback and efficiency measurement. RL with verifiable execution feedback has mostly targeted correctness: CodeRL introduced unit-test-based training, and later GRPO-based work showed binary rewards can scale to reasoning and code generation (Le et al., 2022; Guo et al., 2025). Efficiency changes the object: rewards must rank already-correct programs, so test construction, runtime noise, and aggregation become part of the signal. Efficiency benchmarks make this gap visible with percentile scores against human references, stress tests, right-censored timeouts, instruction counts, and growth-curve complexity inference (Du et al., 2024; Huang et al., 2024; Qing et al., 2025; Liu et al., 2024; Qiu et al., 2025; Peng et al., 2025b; He et al., 2025; Ma et al., 2025; Chambon et al., 2025). These tools do not directly become online RL rewards: stress execution and curve fitting can be too expensive, instruction counts move away from wall-clock time, and raw timing on small tests is noisy enough that PIE reports 1.91x spurious speedups on identical code (Shypula et al., 2024). The missing piece is a training-compatible timing signal.

Learning efficient code. Prior efficiency-training work includes performance-feedback fine-tuning, including online RL on lower-level optimization tasks (Nichols et al., 2024; Wei et al., 2025a; Mikasa et al., 2026), and efficiency-aware supervised fine-tuning on curated efficient-code data (Huang et al., 2025). PIE learns from offline slow-fast edit pairs (Shypula et al., 2024), while other systems search for or iteratively refine faster programs at inference time without updating the policy (Peng et al., 2025a; Gao et al., 2025; Ou et al., 2026; Romera-Paredes et al., 2023; Novikov et al., 2025). These approaches are complementary, but easier for optimization: the system starts from an existing program, or receives repeated execution feedback during search. In one-shot generation, the model must choose the algorithm, data structures, and implementation before seeing runtime feedback.

Closest RL setting. The closest recent comparison is Afterburner (Du et al., 2025). It trains with GRPO in an iterative-refinement setting, where the model receives an existing solution plus runtime metrics and additively mixes format, correctness, and efficiency rewards. That recipe improves efficiency, but also increases the fraction of solutions slower than all human references from 0.33% to 7.33%. We study the harder one-shot setting, where the model must generate from scratch and without execution feedback a correct solution that ranks as high as possible among competitive-programmer references. MaxCode (Ou et al., 2026) instead performs execution-guided search with a frozen proposal policy, so in a head-to-head comparison both systems would use an inference-time oracle that our setting does not have. We still build on GRPO (Shao et al., 2024) and recent analyses

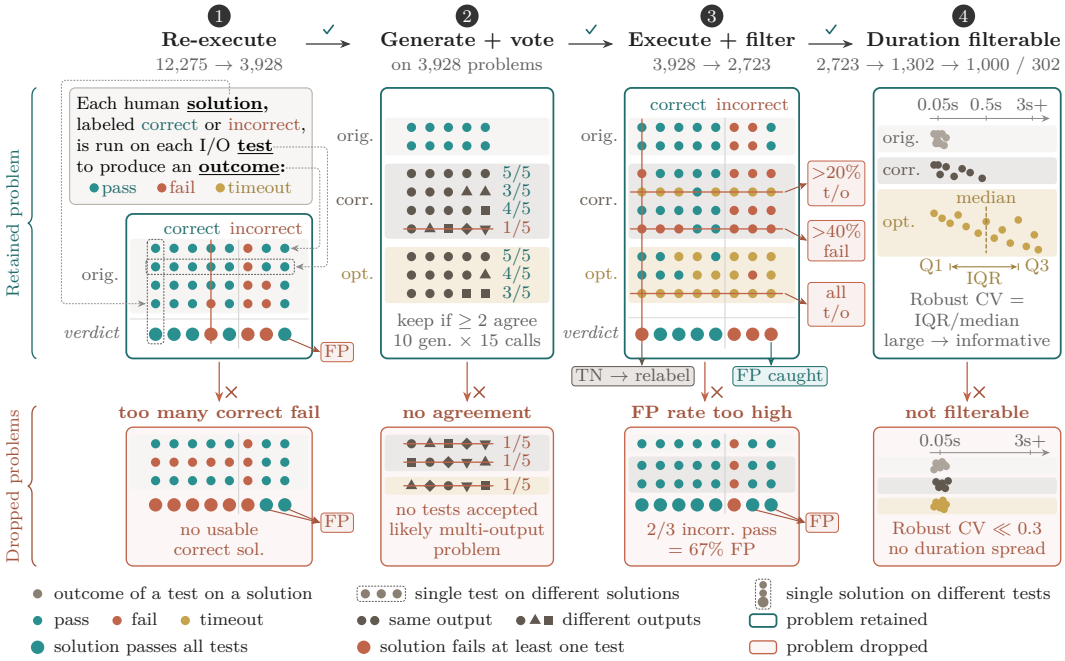


Figure 2: **DMC-Optim construction pipeline.** From 12,275 raw DMC problems, we re-execute human solutions, add 430,215 correctness tests and 352,740 optimization tests, and filter at the test, solution, and problem levels to obtain a 2,723-problem cleaned corpus. A final duration-filterability gate retains the 1,302 timing-based RL problems, split into 1,000/302 train/test; strikethroughs mark removed tests or solution relabels, and red boxes mark dropped problems.

of clipping, baselines, and degenerate groups (Yu et al., 2025; Hao et al., 2025; Liu and Chen, 2026; Xu and Ding, 2025; Liu et al., 2025), but focus on the full timed-execution recipe: a measurable signal, a correctness-preserving reward, and stable training under sparse noisy returns. A broader taxonomy of prior optimization constraints appears in Appendix D.6.

3 How to get a reliable timing measure

Timing-based RL cannot simply add runtime to the reward: execution must reject wrong code, separate correct solutions by duration, and preserve differences under noise. Raw DMC fails: on the final RL split, original tests average 0.088 s and reach only 0.145/0.463 s at $p95/p99$, so tens-of-milliseconds perturbations can dominate decisions (Appendix A).

DMC-Optim rebuilds the test regime. We therefore separate correctness from optimization. From 12,275 DMC problems, we re-execute human solutions, use verified correct and incorrect submissions as controls, add correctness tests, then generate large-input optimization tests; Fig. 2 summarizes the corpus. Adjudication is ambiguous: a new failure can mean an invalid test, a mislabeled solution, or an unsuitable problem. We filter at test, solution, and problem levels, trading recall for cleaner labels and enough RL scale. We retain Python solutions. Python is noisier and more overhead-heavy than C++, which makes it the more demanding setting for a timing methodology that must operate inside an RL loop. If optimization RL can learn useful algorithmic changes in this noisier setting, we expect the same ideas to transfer to lower-overhead languages such as C++, although language-specific calibration would still be needed.

Duration filterability selects the usable subset. Duration filterability is the final gate: averaged verified-correct human runtimes over optimization tests must have robust CV at least 0.3. Original tests satisfy this for at most 3.8% of problems; optimization tests reach 48.2%. On the training split, $p95/p99$ durations are 1.296/3.710 s versus 0.145/0.463 s for original tests, enough spread for ranking. Ablations agree: non-filterable training loses strict p_{10} in three of four optimization settings; for the top-30% leaderboard-percentile environment, replacing duration-filterable problems by non-filterable ones drops pass@1 by 43.5% relative at p_{30} and 50.0% at p_{10} after 5k training steps.

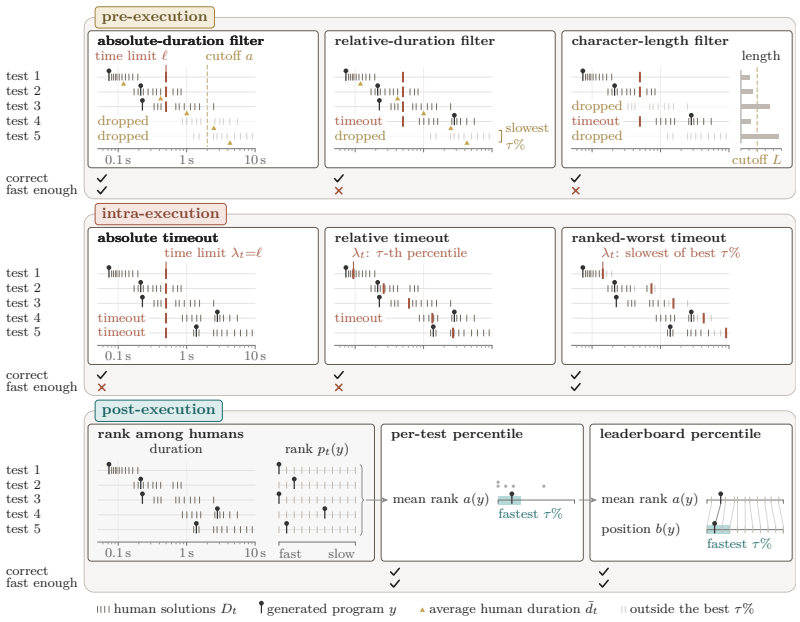


Figure 3: **Different RL environments with different ways of inserting an optimization pressure.** *Top, pre-execution:* a filter chooses the optimization tests; dropped tests (faded) never run, kept tests run under a corresponding time limit ℓ (red bars). The absolute-duration filter keeps $\bar{d}_t < a$; the relative-duration filter drops the slowest $\tau\%$ of tests by $\bar{d}_t$; the character-length filter keeps input + output $< L$ characters. *Middle, intra-execution:* every test runs, and a run with $\delta_t(y) \geq \lambda_t$ becomes a timeout (red bars). Absolute: $\lambda_t = \ell$; relative: $\lambda_t = \max\{10^{-3}, \text{Percentile}_\tau(D_t)\}$; ranked-worst: the slowest duration on test t among the best $\tau\%$ of human solutions by mean per-test rank. In both families this comes with a tunable timeout tolerance ρ. *Bottom, post-execution:* one ranking step, two scalarizations, on top of the naive raw duration approach. Tests run under the 10 s execution cap only; on each, the program's normalized percentile $p_t(y)$ is its rank among the reference solutions on this same test. The mean per-test percentile then averages these ranks into an aggregate $a(y)$; the leaderboard percentile ranks $a(y)$ among the humans' own mean ranks. Every environment returns strict correctness $\tilde{c}_{\text{cor}}$ (all correctness tests pass; executed optimization tests may time out but not fail), the optimization gate g and a graded scalar q: ϕ for pre and intra or $a(y)$ or $b(y)$.

CES rebuilds the timing regime. Better tests still fail under the wrong backend. Local execution runs on the training workers, where sandbox timing contends with inference and rollout orchestration. In standalone reruns of unchanged code, local timing moves the mean-percentile ranking by 41.2 percentage points peak-to-peak on average and flips the fitted timing-trend slope on 18 of 22 measurable problems; local-to-CES fits also have negative cross-validated R^2 on 36,660 pairs. We therefore use CES, the remote code execution service: it isolates timing from worker contention; short-run variability is small; affine service-state drift correction raises stored-vs-fresh Spearman from 0.54 to 0.96, giving Section 4 calibrated, correctness-gated timings. Appendix C further explores how to make the execution environment stable enough for timing measurements to be usable by the RL loop.

4 How to turn it into a learnable reward

Reliable durations are still not a usable RL signal. Section 3 gives us a calibrated execution record that can reject wrong code and compare correct solutions by duration; the reward problem is deciding where that record enters the RL loop, and whether the resulting scalar can be optimized without rewarding fast-but-wrong programs. We use one abstraction: timing can enter before execution, during execution, or after execution.

Where timing enters the environment. The optimization branch has three intervention points. *Pre-execution* families filter optimization tests before the candidate solution is run, using calibrated duration, within-problem relative duration, or input-output length as a workload proxy; they can

remove unstable tails, but cannot create timing information when the suite has no duration spread. *Intra-execution* families impose time constraints while tests run, from global time limits to limits derived from strong human references; they are literal efficiency constraints, but can also create all-fail batches. *Post-execution* families leave the run intact and aggregate durations afterward, typically by inserting the candidate into the calibrated human pool and ranking it across optimization tests. This retains the most information and lets the same execution be replayed under multiple criteria. Appendix D.2 defines the exact environments used in Table 1. See Fig. 3.

The reward-facing interface. After execution, all environments reduce to three reward-facing quantities (c, g, q). The correctness gate $c \in \{0, 1\}$ checks the fixed correctness tests; the optimization gate $g \in \{0, 1\}$ checks timeout tolerance and, when used, a ranking threshold; the optional graded signal $q \in [0, 1]$ is a duration-derived quality score, with lower better. A strict correctness prerequisite additionally rejects hard failures on optimization tests before efficiency credit is available. Time-limit environments can expose timeout rate as q. Ranked environments instead compare each measured duration to calibrated human references, aggregate per-test percentiles, and optionally convert the aggregate into a leaderboard percentile. We screened 17 ranking metrics under repeated CES noise and kept the mean-based percentile family because it was stable and discriminative: within-problem rerun variation was 3.0 ± 1.0 percentile points, while the strong–weak spread remained 47 points. Full screening tables are in Appendix D.3.

Correctness before efficiency. Reward composition decides whether efficiency credit can survive correctness failure. One family keeps correctness as the outer gate: $r(x, y) = -1$ when $c(x, y) = 0$, and only correct solutions receive a positive efficiency score such as $1 - q(x, y)$, which can be kept fully continuous, bucketed, or binarized. Thus a fast but wrong program cannot recover partial reward from the efficiency term. We compare correctness-only and optimization-only controls, additive blends, multitask training, two-gate rewards that grade already-correct solutions, and collapsed rewards that reduce correctness and an optimization criterion to one conjunction. The expanded sweep in Appendix F.5 shows that optimization-only training collapses, additive and multitask rewards mostly navigate a correctness–efficiency frontier, and binary collapsed rewards give the best overall balance.

Pruning the environment space. The three intervention points, scalarization q, and reward composition create a large design space, while each online run uses 8–32 GPU nodes for hours. We therefore screen with an offline simulator before online GRPO. The simulator does not predict exact learning curves: it keeps the environment and reward computation fixed, replaces model generations by sampled human solutions, replaces live CES calls by precomputed calibrated durations, and asks whether better sampled solutions receive better rewards without saturating. Figure 28 in Appendix E uses AUC to expose sparse and saturated regions and steepness to expose flat transitions; downstream DMC-Optim test performance is strongest for diagonal deviation, raw monotonicity, and steepness, while curve noise is not predictive at p_{30} ($r_s = -0.002$, $p = 0.994$). This prunes the environment/reward space before Section 5 turns to GRPO on sparse, noisy returns tied to a live execution service.

5 How to train stable RL with it, and getting the results

GRPO stability. Once efficiency has been made measurable and mapped to a reward, the remaining question is whether the signal can survive online RL. This is harder than standard RLVR because the same training run faces both sparse rewards and residual timing noise: on a Qwen 2.5 7B run, the zero-advantage-context rate at the beginning of training can be around 40–50% for binary optimization rewards. The sandbox state can also vary under concurrent CES load, so two rewards collected at different times are not perfectly comparable even if the model policy has not changed. We keep the GRPO structure but use larger same-prompt groups, center returns without standard-deviation normalization, drop zero-advantage contexts, normalize by a fixed token horizon $N = 32768$, use conservative asymmetric clipping, discard contexts older than $S_{\max} = 30$ optimizer steps, and avoid a replay buffer that would reuse rewards from older sandbox states. The ablations confirm this recipe: larger same-prompt batches help with zero-advantage problems and make runs reach useful updates faster, the staleness filter gives a 5% score gain, and increasing the trainer batch size is the largest lever, improving pure-correctness performance by 10–25%, p_{50} by up to 35%, and p_{30} by up to

60% across optimization RL environments, consistent with unstable gradient estimates under timing noise. The GRPO objective, recipe table, and async diagram are deferred to Appendix F, and the five recipe choices are detailed in Appendix F.2.

Replayed evaluation. DMC-Optim test is the in-domain benchmark for measuring whether a model produces correct code that also meets strict human-speed percentile thresholds. LiveCodeBench (LCB) (Jain et al., 2024) is the out-of-distribution transfer check, where we report both pass@1 and pairwise speed win rates against the standard-RLVR baseline. At each evaluation point, we first run LLM generation and sample 20 solutions per problem. We then run timing in shared CES runs: tests from all problems and all model dumps are shuffled and executed concurrently, and human-reference solutions are included to calibrate durations across evaluation runs. Only after this timing pass do we replay the stored per-test verdicts and durations through different percentile thresholds, time budgets, affine calibrations, and LCB win-rate comparisons without re-running either the model or the timing sandbox.

Experimental setup. We use two complementary starts. CWM 32B is a released SFT checkpoint already prepared for reasoning RL (FAIR CodeGen team et al., 2025). For Qwen 2.5 (Yang et al., 2025), we start from the 7B and 32B base checkpoints, SFT both sizes for one epoch on a public reasoning-only mix built from OpenCodeReasoning-2 and OpenMathReasoning (NVIDIA, 2025; Moshkov et al., 2025), and decontaminate the SFT data against DMC, LCB, and BigO(Bench) (Li et al., 2022; Jain et al., 2024; Chambon et al., 2025). The CWM authors report decontamination against LCB, but we cannot make the same claim for DMC-Optim test or for the RL training prompts used here. Main RL runs use 10,000 optimizer steps, temperature 1.0 sampling with no top-p truncation, learning rates 1×10^{-7} for 7B and 1.4×10^{-7} for 32B, and a 50/50 split between rollout and training nodes. The model, SFT, hyperparameter, and evaluation-task tables, and the decontamination details, are kept in Appendix F.

Negative controls. On Qwen 2.5 7B at step 10k, the control with more-correctness and optimization tests reaches 46.9% pass@1 at p_{100} and 20.6% at p_{50}, while naive raw-duration rewards reach 42.5–44.9% at p_{100} and 18.0–21.7% at p_{50}. Optimization-only rewards are worse: across the reward-structure grid, removing the correctness signal collapses p_{10} pass@1 to 0%. The useful comparison is therefore not "correctness RLVR versus duration reward", but which correctness-gated environment exposes a learnable speed signal; the full ablation summary is in Appendix F.5.

In-domain environment sweep. Table 1 tests the taxonomy of Section 4 on Qwen 2.5 7B at step 10k, using calibrated DMC-Optim resimulation. The table should be read as a staged comparison rather than a leaderboard: first isolate standard RLVR, then add the optimization and more-correctness tests from the data pipeline, and only then compare reward environments. This matters because the data/test work already improves strict p_{50} pass@1 from 18.0% to 20.6%, but it does not by itself produce the best efficient-code models. The larger gains come from three families, each corresponding to a different way of exposing the speed signal. Pre-execution absolute filtering works when the filter keeps enough optimization tests, with the 2 s filter reaching 29.2%. The ranked-worst timeout from the top 80% is one of the strongest timeout-style rows, tying the 0.5 s absolute timeout (29.3% versus 29.4%), but it loses some p_{100} correctness. The post-execution configuration with a top-30% threshold on leaderboard percentile reaches 31.3/19.1/6.0 at $p_{50}/p_{30}/p_{10}$, compared with 18.0/7.7/1.9 for standard RLVR, while preserving p_{100} at 46.2. Taking training variability into account (Table 35), a group of top configurations (the 2 s absolute filter, the 0.5 s absolute timeout, the ranked-worst timeout from the top 80%, and the top-50% and top-30% leaderboard-percentile rewards) remains within or close to the observed variance of one another and well above the baselines, so we do not claim a reliable ordering inside this top group. Figure 31 in Appendix F.7 shows the same distinction over training time: correctness-focused runs improve mostly at lenient thresholds and their gains shrink as the threshold tightens, whereas the 2 s filter and top-30/50% leaderboard-percentile runs retain much larger gains through strict thresholds.

Cross-evaluation and calibration. After Table 1, we test whether the learned speed signal survives changes in evaluation rather than only matching its own reward. Figure 32 in Appendix F.7 cross-evaluates the Qwen 2.5 7B training configurations against 60 evaluation settings. The top-50% leaderboard-percentile run is within one pass@1 point of the best run in 58/60 columns; the top-30% run and 2 s filter each do so in 52/60, and the 0.5 s absolute timeout in 54/60. This

Table 1: Qwen 2.5 7B optimization environment comparison on DMC-Optim test. Rows are RL training configurations from the taxonomy of Section 4, grouped by intervention point. The p_{100} columns check correctness preservation, while p_{50}, p_{30}, and p_{10} progressively emphasize strict speed. All values are calibrated resimulation scores at step 10k. Bold marks the best model result in each column. We used group-averages when possible, and further study variability in Appendix F.6. Single-run training-time 95% CI half-widths at step 10k are $\pm$1.4, 1.4, 1.1, and 0.5 points for pass@1 at p_{100}, p_{50}, p_{30}, and p_{10} (Table 35). Test-time bootstrap half-widths, measured on CWM 32B, are $\pm$4.3, 4.3, 3.8, and 2.3 points; Appendix F.6 combines both.

	pass@1				**pass@10**			
Configuration	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
STANDARD RLVR								
Standard RLVR	43.5	18.0	7.7	1.9	65.1	41.1	22.9	7.4
+ OPTIMIZATION/MORE-CORRECTNESS TESTS								
More-correctness tests	46.7	19.8	8.7	2.4	67.0	44.7	26.0	8.3
More-correctness tests + large 10 s timeout	46.4	19.8	8.8	2.3	68.2	46.8	27.9	8.9
More-correctness + optimization tests	46.9	20.6	9.3	2.7	67.6	47.2	28.2	9.6
+ REWARD: PRE-EXECUTION (test filtering)								
Absolute-duration filter								
$\tau = 0.1$ s	46.7	20.2	9.2	2.5	67.6	47.0	28.7	9.2
$\tau = 0.5$ s	46.5	25.3	13.3	4.3	67.2	51.9	34.9	13.8
$\tau = 2$ s	46.9	29.2	17.4	5.7	67.5	54.7	39.0	17.3
Character-length filter								
$\geq$ 1k characters	46.7	20.0	8.8	2.0	67.6	46.2	27.7	7.3
$\geq$ 100k characters	46.4	21.2	10.3	3.1	67.7	48.1	28.8	11.4
$\geq$ 10M characters	45.6	28.6	16.8	5.5	65.2	54.0	37.9	16.4
Relative-duration filter								
top 30%	**47.4**	22.8	11.7	3.7	**68.4**	48.9	32.4	12.4
top 50%	47.0	21.5	10.2	3.0	67.6	46.9	27.7	10.1
top 80%	47.2	21.3	9.7	2.7	67.0	46.8	28.0	10.3
+ REWARD: INTRA-EXECUTION (timeout-based)								
Absolute timeout								
$T = 0.5$ s	47.3	29.4	17.5	5.5	67.3	55.9	41.0	16.9
$T = 2$ s	46.6	22.1	10.6	3.4	66.8	48.7	30.4	11.8
Ranked-worst timeout								
from top 20% ranked	39.2	26.6	16.4	5.6	61.4	49.8	37.9	16.6
from top 50% ranked	45.7	23.6	12.2	3.7	66.6	48.7	32.8	11.3
from top 80% ranked	45.0	29.3	17.7	**6.0**	65.8	54.3	39.8	17.2
Relative timeout	*Substantially degrades correctness; see Table 36.*							
+ REWARD: POST-EXECUTION (ranking-based)								
Leaderboard percentile								
Two-gate bucketed	47.1	27.5	15.5	5.1	66.8	53.1	37.4	15.7
top 80%	46.2	22.9	11.7	3.5	68.0	49.7	31.7	11.9
top 50%	46.2	30.5	18.4	5.7	65.8	**56.2**	**43.3**	17.9
top 30%	46.2	**31.3**	**19.1**	**6.0**	67.8	56.1	42.7	**18.6**
Mean per-test percentile								
Two-gate bucketed	46.5	25.4	13.8	4.4	67.5	52.7	36.2	13.8

complements Fig. 31: optimization gains concentrate as evaluation tightens, and the best runs remain competitive when the evaluator changes. Figure 20 in Appendix C.9 checks CWM 32B under affine-calibration perturbations. Absolute p_{50} and p_{10} scores move, especially through β, but the ordering is mostly preserved; as the sandbox degrades, optimization-trained models reach roughly 100%–200% improvement over standard RLVR, depending on the evaluation criterion.

Across models and LCB transfer. Table 2 separates in-domain DMC-Optim gains from out-of-distribution LCB speed transfer. Trying Qwen 2.5 32B and CWM 32B with the optimization RL environments and rewards that worked best on Qwen 2.5 7B preserves most of the gains, showing that the enhanced optimization-RL training scales beyond the original Qwen 2.5 7B setting (Table 37); the exception is the 2 s filter on Qwen 2.5 32B, which loses 6.6 points of p_{100} pass@1 against standard RLVR. On DMC-Optim test, the common top-30% leaderboard-percentile environment raises p_{50} from 30.7 to 50.4 for CWM, 21.1 to 39.6 for Qwen 32B, and 18.0 to 31.3 for Qwen 7B. At p_{50},

Table 2: Cross-model DMC-Optim results and out-of-distribution transfer to LiveCodeBench (LCB). (a) DMC-Optim pass@1 for standard RLVR, RLVR with our optimization tests, and the common top-30% leaderboard-percentile training environment; p_{100} measures pure correctness and lower percentiles impose stricter speed requirements. (b) LCB pass@1 (RLVR value in parentheses) and speed win rates of the same top-30% environment against each model-family RLVR baseline. WR_b/WR_m compare the fastest/median passing sample among 20 generations and exclude draws; if only one model has a correct sample, that model wins. RS: random-sample win rate over all sample pairs (Appendix G.1); Δ is A (trained model) minus B (RLVR), both before rounding. Brackets are 95% task-bootstrap intervals and * marks a win rate whose interval excludes 50% or a margin whose paired interval excludes 0; the CWM 32B row averages four reruns (Table 39), all four significant, and its brackets span the lowest lower and highest upper bounds of the per-rerun intervals. Single-run DMC-Optim training-time 95% CI half-widths at step 10k are at most 1.5 points (measured on Qwen 2.5 7B and CWM 32B), and test-time bootstrap half-widths, measured on CWM 32B, are $\pm$4.3, 4.3, 3.8, and 2.3 points at p_{100}, p_{50}, p_{30}, and p_{10}; Appendix F.6 combines both. Since LCB and DMC-Optim have similar test-set sizes, LCB pass@1 test-set uncertainty should be of similar order (about $\pm$4 points).

(a) DMC-Optim test pass@1 (bold: best displayed value within each model and evaluation threshold)

	Standard RLVR				More-correctness + optimization tests				Leaderboard percentile, top 30%			
Model	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
Qwen 2.5 7B	43.5	18.0	7.7	1.9	**46.9**	20.6	9.3	2.7	46.2	**31.3**	**19.1**	**6.0**
Qwen 2.5 32B	54.8	21.1	9.4	2.4	**60.0**	28.7	14.2	4.4	55.7	**39.6**	**24.2**	**8.5**
CWM 32B	69.8	30.7	13.7	3.3	**71.5**	35.6	17.1	4.7	70.2	**50.4**	**30.9**	**9.8**

(b) LCB transfer of the top-30% leaderboard-percentile environment

Model	p@1 (RLVR)	WR_b	WR_m	RS: A / B / draw	RS Δ
Qwen 2.5 7B	40.6 (44.2)	60.3 [51.9, 68.0]*	68.5 [59.6, 76.9]*	22.1 / 18.9 / 59.0	+3.1 [+0.4, +6.1]*
Qwen 2.5 32B	49.1 (52.7)	61.9 [53.4, 70.1]*	72.9 [64.7, 80.5]*	27.4 / 20.4 / 52.2	+7.0 [+3.1, +10.9]*
CWM 32B (4 reruns)	53.9 (56.4)	66.3 [54.0, 78.1]*	82.9 [72.9, 94.9]*	29.4 / 20.4 / 50.3	+9.1 [+4.7, +13.9]*

the CWM 32B top-30% post-execution trained model improves over standard RLVR by 19.7 points and over optimization-test RLVR by 14.8 points, which is $9.9\times$ and $7.4\times$ the training-time half-width of differences, respectively (Table 35). On LCB, the same training reaches 68.5/72.9/82.9% median-sample speed wins for Qwen 7B/Qwen 32B/CWM 32B; the strongest CWM result reaches 83.0% (Table 43). Some configurations trade pass@1 for speed, but CWM's top-30% row retains 53.9% versus 56.4% for RLVR, and its pass@10 remains nearly unchanged (65.6 versus 66.3). The best- and median-sample win rates encompass the use case of relying on execution feedback at inference. On top of that, the random-sample win rate in Table 2 shows that overall, the optimization-RL models still outperform standard RLVR on out-of-domain data, even if some of the speed gains also come with a slight diminution of pure correctness (for instance, -2.5 points of LCB pass@1 for the top-30% environment on CWM 32B compared to RLVR). Thus the reward and environment choices improve efficient code generation under strict evaluation, with LCB transfer appearing mainly as faster passing solutions.

6 Analysis: What Does the Model Learn?

Across difficulty. Section 5 shows that optimization-aware RL improves strict efficiency metrics, but not what kind of optimization the model learned. We check whether the gain is only an easy-problem artifact. At 10k steps, the CWM 32B top-30% leaderboard-percentile run improves p_{50} pass@1 over standard RLVR by 19.4, 20.4, and 8.7 points on Easy, Medium, and Hard problems, while p_{100} changes by only +1.8, +2.4, and +1.1 points. Thus the reward teaches measurable speedups across the benchmark, but Hard remains the bottleneck, with the same run reaching 30.6% at p_{50} compared with 51.3% on Easy.

What changed in the code. We compare best-of-20 solutions from a post-execution optimization-trained CWM 32B model, its RLVR baseline, and the fastest correct human Codeforces solution on the 302 DMC-Optim test problems. All sources are re-executed in the same CES campaign; ties below 1% relative duration are removed; GPT-OSS 120B (OpenAI et al., 2025) judges blinded code pairs with 10 randomized rollouts per pair. The trained model wins 200 of 224 speed-win pairs against its baseline (89%), so the RL result is visible at the solution-pair level. Requiring at least three speed-correct judge rollouts (rollouts where the judge identifies the true fastest solution, a filtering step) leaves 174 classifiable pairs. The categories of Fig. 4 show where the gain comes from: input-output (I/O) optimization (47%) and constant factors (34%), while math shortcuts, data-structure changes, and algorithm changes remain rare.

Where the human gap remains. Humans still beat the trained model in 153 of 227 speed-win pairs (67%), and complexity improvements are more common in human comparisons: 22% for human-vs-baseline and 16% for human-vs-model, versus 13% for model-vs-baseline. The judge speed-identification accuracy, used only as a screening step, is 68%, and drops to 59% on human-versus-model pairs where the recorded-duration differences are tighter. A manual review nevertheless found 94.8% accuracy on complexity classification among the model-versus-baseline pairs deemed classifiable.

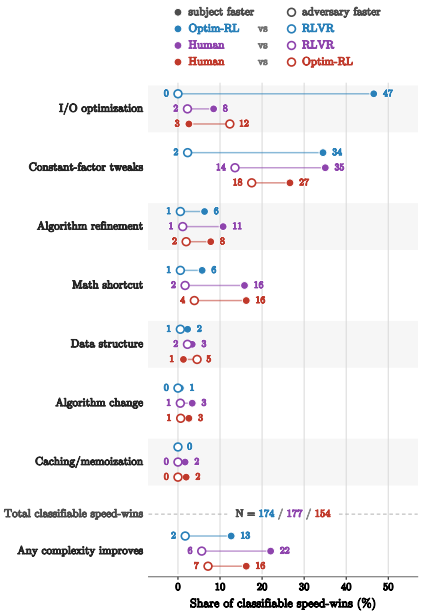


Figure 4: Judge category mix. Model wins concentrate on I/O and constants; human comparisons contain more math, data-structure, and algorithmic changes.

Limitations. DMC-Optim is still a narrow setting for software optimization. The tasks are single-file Python competitive-programming problems, some rewards compare against a fixed pool of human reference timings, and the timed signal depends on shared CES re-execution rather than a cheap static oracle. This makes the benchmark useful for isolating trainable wall-clock rewards, but it does not cover repository-scale profiling, memory objectives, multi-language systems code, or long-horizon edit loops. Recent work points toward these settings through repository workloads (He et al., 2025; Ma et al., 2025) and refinement or search-based optimization (Du et al., 2025; Peng et al., 2025a; Gao et al., 2025). Those directions will need rewards that are less tied to static human pools and cheaper than repeated timing, while still preventing correctness regressions. The full pipeline also requires stronger tests, many executions, and substantial RL compute. We are not able to release the underlying data, and the appendices provide details to reproduce the work. We study online RL only (a matched offline comparison would need its own algorithm, offline corpus, and preference pairs); since the optimization signal survives the noisier online loop, the same pipeline could also label offline data, as a replacement for or an addition to online training. It would also be interesting to combine methods and see how our trained policy could be used with inference-time search methods (Du et al., 2025; Ou et al., 2026). The judge analysis also suggests a missing ingredient: the reward observes that a solution is faster, but rarely says whether the gain came from asymptotics, data structures, math, local implementation, or most importantly complexity improvements.

Conclusion. In the online RL setting we study, RL for code optimization can be made trainable when measurement, rewards, and GRPO form one signal pipeline: larger optimization tests, shared execution, collapsed binary rewards, and pre-, intra-, or post-execution constraints. This improves strict DMC-Optim efficiency across models and difficulties while mostly preserving p_{100} correctness. The boundary is clear: the model mostly learns implementation-level speedups (I/O and constants), while humans retain more algorithmic and complexity-improving moves. We view this as a first step, not evidence that current rewards teach algorithm discovery; closing the gap may require algorithm-aware training feedback.

The same correctness-efficiency gap we originally measured on competitive programming (where standard RLVR drops from 43.5% p_{100} pass@1 to 7.7% at p_{30} on Qwen 2.5 7B, and from 69.8% to 13.7% on CWM 32B) also appears on real-world software-engineering tasks, where it is, if anything, even more significant. RL methods to improve code generation correctness have already transferred from "playground" settings to real-world software engineering (Wei et al., 2025b).

Efficiency has not been correctly mastered on software-engineering tasks yet. Not only are software-engineering tasks harder and far more expensive to execute, slowing down RL iterations, but the challenges that make competitive-programming efficiency hard (measurement reliability, reward noise, and training instability) are all exacerbated on real-world software-engineering tasks. We see this work as a step toward a similar transfer for efficiency: it develops some methods in the competitive-programming "playground", where the feedback loop is more easily handled, and may provide a basis for later adaptation to real-world software optimization.

Acknowledgments and Disclosure of Funding

We thank the authors of the CWM paper whose RL codebase made this work possible, in particular Jonas Gehring, Jade Copet, Quentin Carbonneaux, David Zhang, Badr Youbi Idrissi, Vegard Mella, and Taco Cohen. We also thank Zacharias Fisches, Sida Wang, Mathurin Videau, and João Maria Janeiro for helpful discussions. We thank Andrew Hamiel, Don Landrum, and Eslam Elnikety for their support. This work was funded by Meta.

References

Bubblewrap. `https://github.com/containers/bubblewrap`, 2026. Accessed 2026-06-07.

Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. BigO(Bench) – can LLMs generate code with controlled time and space complexity? *arXiv preprint arXiv:2503.15242*, 2025.

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

Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

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. *arXiv preprint arXiv:2505.23387*, 2025.

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. *arXiv preprint arXiv:2510.02387*, 2025.

Shuzheng Gao et al. Search-based LLMs for code optimization. In *IEEE/ACM International Conference on Software Engineering*, 2025. arXiv:2408.12159.

Daya Guo, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Yaru Hao, Li Dong, Xun Wu, Shaohan Huang, Zewen Chi, and Furu Wei. On-policy RL with optimal reward baseline. *arXiv preprint arXiv:2505.23585*, 2025.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? *arXiv preprint arXiv:2507.12415*, 2025.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. EffiBench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.

Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. EffiCoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. In *International Conference on Machine Learning*, 2025. arXiv:2410.10209.

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. *arXiv preprint arXiv:2403.07974*, 2024.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Chi Liu and Xin Chen. Adaptive-boundary-clipping GRPO: Ensuring bounded ratios for stable and generalizable training. *arXiv preprint arXiv:2601.03895*, 2026.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. EvalPerf: Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*, 2024.

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. *arXiv preprint arXiv:2503.20783*, 2025.

Jeffrey J. 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? *arXiv preprint arXiv:2511.06090*, 2025.

Ryo Mikasa, Shun-ichiro Hayashi, Daichi Mukunoki, Tetsuya Hoshino, and Takahiro Katagiri. Improving HPC code generation capability of LLMs via online reinforcement learning with real-machine benchmark rewards. *arXiv preprint arXiv:2602.12049*, 2026.

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. *arXiv preprint arXiv:2504.16891*, 2025.

Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned LLMs for generating fast code. *arXiv preprint arXiv:2404.18864*, 2024.

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

NVIDIA. OpenCodeReasoning-2. Dataset on Hugging Face, 2025. CC-BY-4.0.

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin,

Jordan Liss, Lily Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL `https://arxiv.org/abs/2508.10925`.

Jiefu Ou, Sapana Chaudhary, Kaj Bostrom, Nathaniel Weir, Shuai Zhang, Huzefa Rangwala, and George Karypis. MaxCode: A max-reward reinforcement learning framework for automated code optimization. *arXiv preprint arXiv:2601.05475*, 2026.

Yun Peng, Akhilesh Deepak Gotmare, Michael R. Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. PerfCodeGen: Improving performance of LLM generated code with execution feedback. In *IEEE/ACM International Conference on AI Foundation Models and Software Engineering*, pages 1–13, 2025a. doi: 10.1109/Forge66646.2025.00008.

Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. COFFE: A code efficiency benchmark for code generation. *Proceedings of the ACM International Conference on the Foundations of Software Engineering*, 2025b. arXiv:2502.02827.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, See-Kiong Ng, and Luu Anh Tuan. EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code. *arXiv preprint arXiv:2505.13004*, 2025.

Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is LLM-generated code? A rigorous & high-standard benchmark. *International Conference on Learning Representations*, 2025. arXiv:2406.06647.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2023. doi: 10.1038/s41586-023-06924-6.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *International Conference on Learning Representations*, 2024. arXiv:2302.07867.

Sida I. Wang. Measuring all the noises of LLM evals, 2026. URL `https://arxiv.org/abs/2512.21326`.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. SuperCoder: Assembly program superoptimization with large language models. *arXiv preprint arXiv:2505.11480*, 2025a.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution, 2025b. URL `https://arxiv.org/abs/2502.18449`.

Zhongwen Xu and Zihan Ding. Single-stream policy optimization. *arXiv preprint arXiv:2509.13232*, 2025.

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. *arXiv preprint arXiv:2412.15115*, 2025.

Qiying Yu et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

A Simulation study of efficiency metrics under measurement noise

This appendix supports the measurement argument of Section 3. It isolates the short-term perturbations that affect a single execution, and separates them from the longer-term cross-snapshot drift treated in Appendices C.7 to C.8. Our aim is threefold: state a compact noise hypothesis, relate it to the fast-test and slow-test regimes induced by different datasets, and study which simple metric families remain usable under these perturbations.

Throughout the quantitative comparison we use two datasets: LiveCodeBench (LCB) and DMC-Optim. We discuss the original DMC tests in prose only. As shown in Section 3, those original tests remain in the same fast-test regime as LCB, so the LCB comparison is the clearer representative of that setting.

A.1 Empirical timing regimes

Fast-test and slow-test regimes. The empirical difference between the two regimes is visible before any simulation. Figure 5 shows that DMC-Optim uses both larger optimization tests and more tests per problem than LCB. LCB aggregates 10,125 public/private tests over 287 problems, or 35.3 tests per problem on average, while DMC-Optim contributes 36,814 optimization tests over 302 problems, or 121.9 per problem. The typical LCB test remains in the tens of characters, whereas the typical DMC-Optim test is around 10^3 characters, with a much heavier upper tail.

The same pattern appears in runtime. On the final RL training split, the original DMC tests reach only $0.145\,\mathrm{s}$ at $p95$ and $0.463\,\mathrm{s}$ at $p99$, whereas the generated optimization tests reach $1.296\,\mathrm{s}$ and $3.710\,\mathrm{s}$ (Section 3). LCB is not the original DMC benchmark, but it occupies the same fast-test regime: Fig. 6 shows a sharp concentration on very fast executions and a broad plateau in the survival curve between 2 and 10 s, so changing the timeout across that range moves little signal. DMC-Optim instead shows a steadier decay, which means the same change in threshold can still move a non-trivial fraction of tests across the boundary.

A.2 A compact noise hypothesis

Additive and multiplicative terms. Let $d^{\text{true}}_{S,i}$ be the execution time of solution S on test i in a fixed execution environment. We model the observed duration as

$$d^{\text{obs}}_{S,i} = d^{\text{true}}_{S,i}\left(1 + \epsilon^{\text{mult}}_{S,i}\right) + \epsilon^{\text{add}}_{S,i}. \tag{1}$$

The multiplicative term captures proportional slowdowns that scale with the workload. The additive term captures fixed overheads such as scheduling jitter, dispatch latency, or context-switch costs.

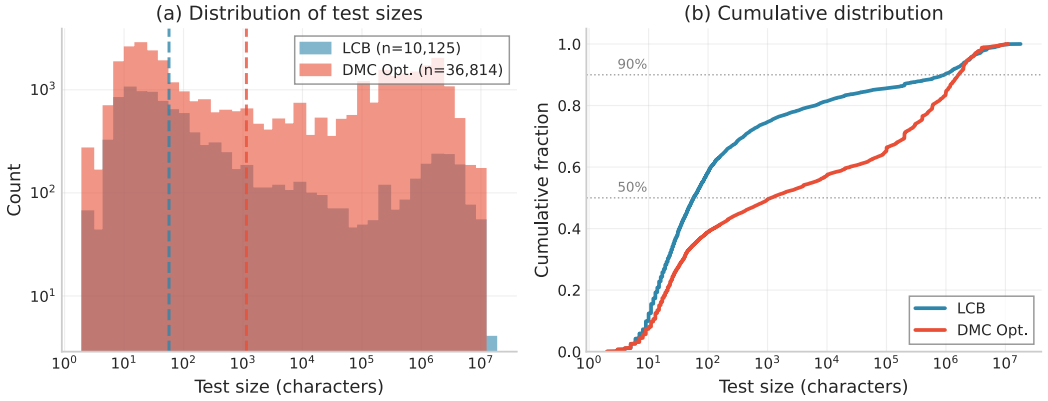


Figure 5: **Test-size regimes for LCB and DMC-Optim.** *Left:* Histograms of per-test input-plus-output character counts on log–log axes, with dashed vertical lines at the medians. LCB is concentrated on small public/private tests, while DMC-Optim optimization tests are shifted toward larger inputs and retain a much heavier upper tail. *Right:* Cumulative distributions on a log-scaled x-axis. The separation is already visible around the median and widens further in the upper tail, which is where larger inputs can induce measurable runtime differences between solutions.

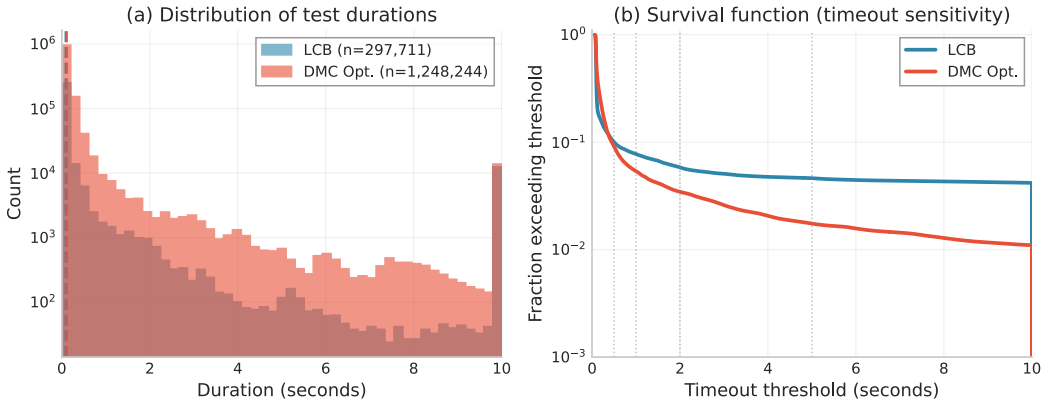


Figure 6: **Observed duration distributions for LCB and DMC-Optim.** *Left:* Histograms of per-test execution durations on a log-scaled y-axis for 297,711 LCB executions and 1,248,244 DMC-Optim executions, aggregated over two model evaluations on the DMC-Optim test split. LCB places more mass near very fast executions and exhibits a visible timeout-induced tail, while DMC-Optim has a broader body and a smoother upper tail. *Right:* Survival functions showing the fraction of tests above each threshold, again on a log-scaled y-axis. LCB becomes nearly flat between 2 and 10 s, so threshold changes in that region move little useful signal; DMC-Optim decays more steadily, making threshold choice materially more informative.

The quantity that matters on fast tests is the relative additive burden

$$r_{S,i}^{\text{add}} = \frac{\left|\epsilon_{S,i}^{\text{add}}\right|}{d_{S,i}^{\text{true}}}. \tag{2}$$

As $d_{S,i}^{\text{true}}$ becomes small, the same absolute perturbation occupies a larger fraction of the measurement.

This decomposition is not only synthetic. The stored-to-fresh CES calibration in Appendix C.7 fits an affine map with a 53 ms intercept, which is consistent with a non-negligible fixed-cost term in the execution stack. That intercept is not the same object as per-call timing noise, but it gives the right scale for the additive component studied here. As a corroborative check on the test executions used for the benchmark comparison, a hypothetical 53 ms perturbation would amount to at least half of the observed duration for 75.9% of LCB executions and 52.4% of DMC-Optim executions; at 100 ms, it would exceed the full observed duration for 68.8% and 46.3%, respectively. These auxiliary numbers are not our main duration statistics, but they point in the same direction as Section 3 and Fig. 6: a fixed tens-of-milliseconds term is much harder to absorb in the fast-test regime.

A.3 Simulation study: timeout versus win-rate

Why compare these two metrics. Every metric in this section is built from the same noisy per-test durations, but the aggregation changes how noise enters the final decision. We therefore compare a thresholded absolute metric and a threshold-free relative metric. We use win-rate here because it is the simplest relative comparator; the final reward in the paper uses mean percentile rather than raw win-rate (Section 4 and Appendix D.3), but this appendix isolates the thresholded-versus-relative distinction with the fewest additional choices.

Two simple metric families. For a fixed timeout τ, the timeout score of solution S is

$$m_\tau(S) = 1\left[\max_i d_{S,i}^{\text{obs}} \le \tau\right]. \tag{3}$$

In a pairwise comparison between solutions A and B, the timeout metric declares B the winner only if $m_\tau(B) > m_\tau(A)$. This makes the decision hinge on a single threshold. Even before adding noise, this is only possible if there exists some τ such that $\max_i d_{B,i}^{\text{true}} < \tau < \max_i d_{A,i}^{\text{true}}$. If per-test heterogeneity makes $\max_i d_{B,i}^{\text{true}} \ge \max_i d_{A,i}^{\text{true}}$ even though B is faster on average, then no timeout

threshold can separate the pair at all. Noise then compounds a structural limitation that already exists in the worst-case aggregation.

For win-rate, we compare the two observed durations test by test:

$$w(B; A) = \sum_i 1\left[d^{\text{obs}}_{B,i} < d^{\text{obs}}_{A,i}\right]. \tag{4}$$

The win-rate metric declares B the winner if $w(B; A) > w(A; B)$. It is still noisy, but it does not require tuning an external threshold and it averages the comparison over all tests rather than over a single boundary event.

Stylized problem families. Following the original methodology study, we do not reuse the raw empirical histograms directly. Instead we simulate stylized LCB-like and DMC-Optim-like problems that preserve the qualitative regime difference: an LCB-like family with 35 tests per problem and characteristic sub-second durations, and a DMC-Optim-like family with 122 tests per problem and substantially slower tests. In each trial, solution B is the true winner and solution A is 20% slower on average, with additional per-test variability so that some individual tests can still reverse the ordering. We inject either multiplicative noise or additive noise and measure how often each metric identifies the correct winner. Except for the explicit threshold sweep in Fig. 7, timeout results use an oracle threshold chosen after the fact, so the reported timeout curves should be read as optimistic. The three figures below answer complementary questions: Fig. 7 shows whether threshold tuning can rescue timeout at all, Fig. 8 shows how much more tests help each metric family, and Fig. 9 shows how the two dataset regimes separate as noise grows.

Synthetic playground specification. For regime $r \in \{\text{LCB-like}, \text{DMC-Optim-like}\}$, we first draw base test durations $b_i \sim \text{LogNormal}(\mu_r, \sigma_r)$, with $(\mu_r, \sigma_r) = (-2.3, 0.5)$ for LCB-like problems and $(1.0, 0.5)$ for DMC-Optim-like problems. We use $N_r = 35$ and $N_r = 122$ tests, respectively, to match the empirical test-count gap. We then generate the true durations of the two competing solutions as

$$d^{\text{true}}_{A,i} = b_i \left(1 + \frac{\Delta}{2} + \eta_{A,i}\right), \qquad d^{\text{true}}_{B,i} = b_i \left(1 - \frac{\Delta}{2} + \eta_{B,i}\right), \tag{5}$$

with mean gap $\Delta = 0.2$ and per-test variability $\eta_{A,i}, \eta_{B,i} \sim \mathcal{N}(0, 0.15^2)$, before applying the observation model from Eq. (1). These values are not meant as an exact parametric fit of the empirical histograms. They are stylized anchors chosen to preserve the main ingredients that matter for the argument: positive and strongly right-skewed durations, the order-of-magnitude separation between fast and slow tests, and the empirical difference in tests per problem. Each Monte Carlo trial resamples the base durations, the test-specific gap factors, and the measurement noise, so the reported curves average over both execution noise and problem-to-problem variability rather than over a single frozen synthetic instance.

Threshold fragility. Figure 7 shows the first difference between the two metric families. Win-rate is flat because it has no threshold to tune. Timeout is only informative in a narrow band between the two solutions' extremes, so its accuracy varies sharply with the chosen threshold even under multiplicative noise. Under additive noise on LCB-like problems, that useful band largely disappears and timeout remains close to chance across the full sweep, whereas win-rate stays clearly above chance.

More tests help, but not equally. Figure 8 shows that increasing the number of tests improves both metric families, but the gain is not symmetric. This matters because DMC-Optim provides about 3.5 times more tests per problem than LCB. On LCB-like problems under additive noise, win-rate keeps improving as more tests are added, whereas timeout plateaus much earlier. On DMC-Optim-like problems both metrics improve, which is exactly the behavior we want from a slower and denser test suite.

The regime split appears under additive noise. Figure 9 summarizes the comparison by sweeping the noise magnitude directly. Under multiplicative noise, both datasets degrade gradually and the gap between timeout and win-rate is modest. The qualitative split appears under additive noise. On

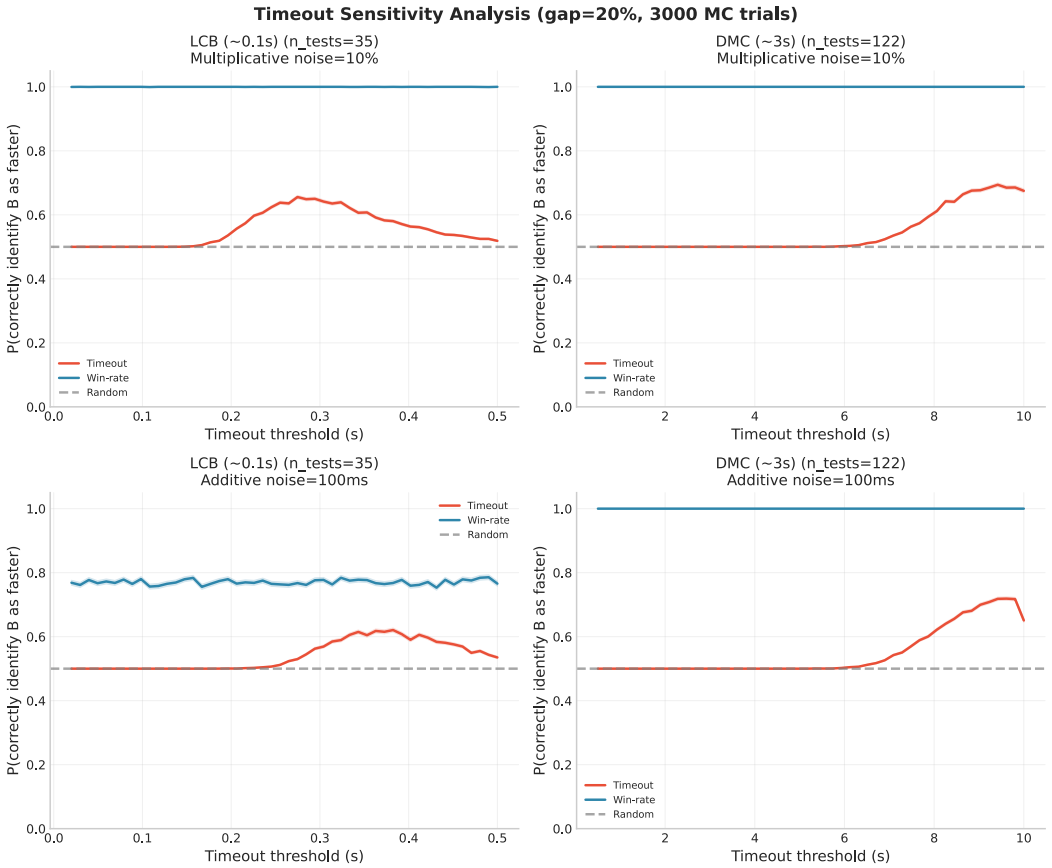


Figure 7: **Threshold sensitivity of timeout versus win-rate.** Accuracy is plotted against the timeout threshold for LCB-like problems (left) and DMC-Optim-like problems (right), under multiplicative noise (top) and additive noise (bottom). Timeout is the red curve and win-rate the blue horizontal line. Timeout is only useful in a narrow interval between the two solutions' extremes; outside that interval both solutions either pass or fail together, producing ties. On LCB-like problems with additive noise, the useful interval largely disappears and timeout remains close to chance, while win-rate stays meaningfully higher because it compares durations directly rather than through a single boundary.

LCB-like problems, timeout approaches chance while win-rate remains meaningfully above chance. On DMC-Optim-like problems, both metrics remain usable over a much wider range, although win-rate stays higher throughout.

Practical implication for experiment design. These figures suggest a simple order of operations. First change the data regime: larger inputs, longer runtimes, and more tests per problem improve both metric families at once. Only then does it become worthwhile to decide whether a hard timeout should remain part of the optimization objective. If the suite stays in the fast-test regime, tighter threshold tuning does not fix the underlying problem; it only searches for a narrow operating window in a measurement setting where thresholded decisions are intrinsically brittle. This is why our experiments first move from the original fast-test regime to DMC-Optim, and only then study how different efficiency metrics behave on top of that dataset.

A.4 Takeaways

This appendix supports three conclusions. First, a mixed noise model with an additive term is the relevant abstraction for fast test suites. Second, longer and more numerous optimization tests make both thresholded and relative efficiency metrics less sensitive to measurement noise; original DMC and LCB remain on the fragile side of this divide, while DMC-Optim moves enough mass into the slower regime to make timing-based decisions more informative. Third, on fast tests, win-rate

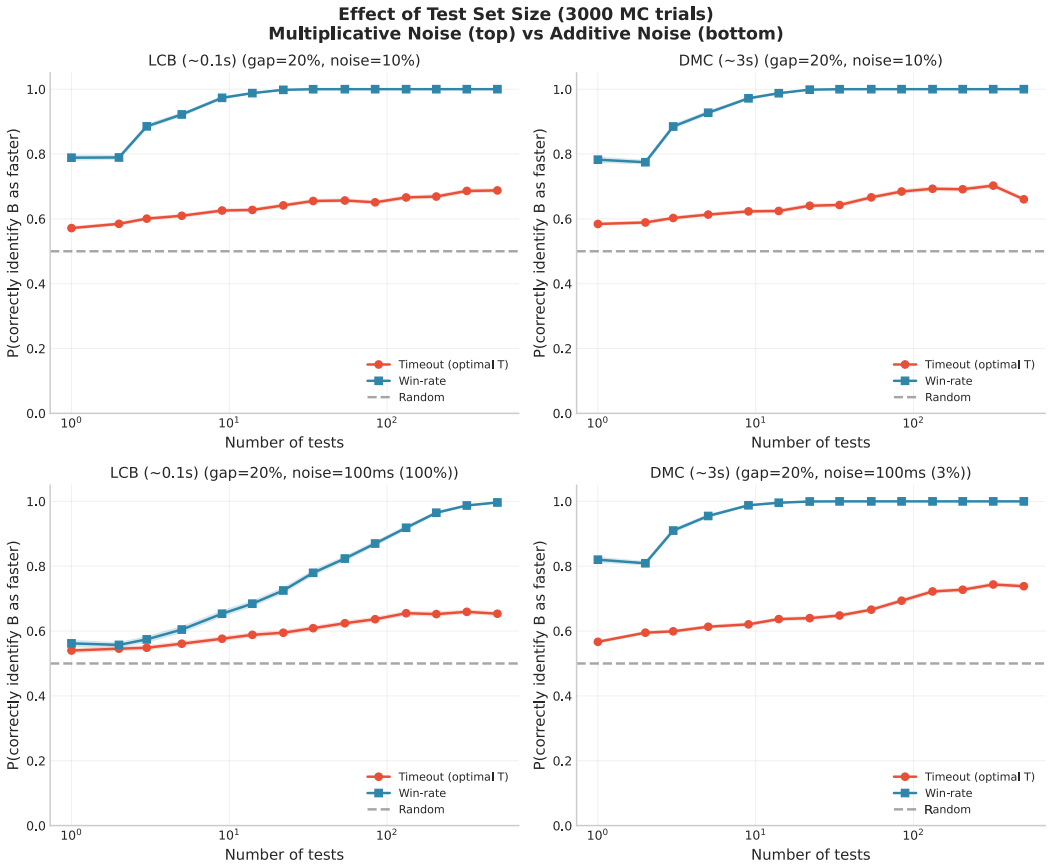


Figure 8: **Effect of the number of tests per problem.** Accuracy is plotted against the number of tests for LCB-like problems (left) and DMC-Optim-like problems (right), under multiplicative noise (top) and additive noise (bottom). On fast problems with additive noise, win-rate continues to gain from additional tests because it averages pairwise comparisons across the suite, while timeout plateaus earlier because every decision still hinges on a single threshold. On slower problems, both metrics benefit from having more tests, which is one reason DMC-Optim is easier to measure reliably than LCB.

is more reliable than timeout because it aggregates pairwise comparisons across tests instead of hinging on a single boundary. Timeout can still be useful when a hard latency budget is semantically meaningful, but it degrades earlier than win-rate as tests get shorter or the additive term gets larger.

The same analysis also suggests what a useful optimization test suite should provide: inputs large enough to expose algorithmic differences, enough tests per problem to average over test-specific reversals, and characteristic durations long enough that a fixed 10–100 ms perturbation remains a modest fraction of the signal. DMC-Optim does not remove measurement noise, but it moves evaluation much closer to that regime than LCB or the original DMC tests.

This is why Section 3 changes the data regime before changing the reward. Once the test suite provides larger inputs, longer durations, and more trials per problem, both metric families become easier to use. In the fast-test regime, by contrast, the thresholded metric fails first.

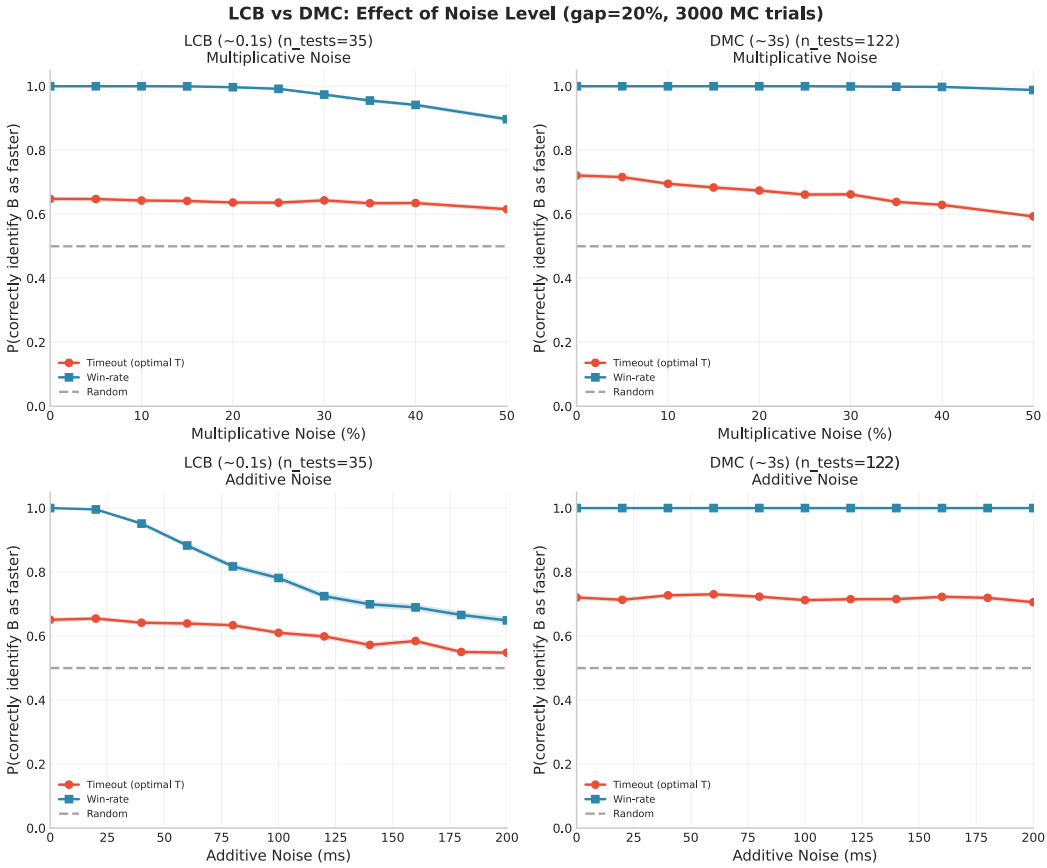


Figure 9: **Noise sensitivity across the LCB-like and DMC-Optim-like regimes.** Each panel plots accuracy against noise magnitude for LCB-like problems (left) and DMC-Optim-like problems (right), under multiplicative noise (top) and additive noise (bottom). Multiplicative noise degrades both families gradually. Additive noise instead separates the regimes: on LCB-like problems the timeout metric approaches chance while win-rate remains meaningfully above chance, whereas on DMC-Optim-like problems both metrics remain usable over a much wider range. Longer and more numerous tests therefore help both metric families, but the thresholded one fails earlier.

B DMC-Optim construction and filterability details

This appendix complements Section 3. It records the dataset-construction thresholds and filterability diagnostics that support the one-page main-paper summary.

We retain Python solutions. C++ is the dominant language in competitive programming, but it is also the most controlled setting for low-level speed: compiler optimizations, memory layout, and hand-tuned implementation details can dominate the methodology. Python is noisier and more overhead-heavy, which makes it the more demanding setting for a timing methodology that must operate inside an RL loop. If optimization RL can learn useful algorithmic changes in this noisier setting, we expect the same ideas to transfer to lower-overhead languages such as C++, although language-specific calibration would still be needed.

B.1 Construction stages

The DMC-Optim pipeline produces two related endpoints. The first is a 2,723-problem cleaned corpus with verified solutions and multiple generated test categories. The second is the RL pool used in the main paper: after selecting problems whose optimization tests are duration-filterable, we obtain 1,302 problems and resplit them into 1,000 training and 302 test problems.

Table 3: **Dataset construction pipeline.** Each row shows a processing stage and the number of problems remaining afterward. The pipeline yields two related artifacts: a 2,723-problem intermediate cleaned dataset, and the 1,302-problem duration-filterable pool used for optimization RL. The experiments in this paper resplit that pool into 1,000 training and 302 test problems.

Stage	Operation	Problems
0	Raw DMC data	12,275
1	Deduplication against evaluation benchmarks	11,468
2	Require both correct & incorrect solutions	6,706
3	Execution-based solution validation	3,928
4	Test augmentation & quality filtering	3,061
5	Intersection across test categories	2,978
6	Require non-empty correctness and optimization tests	2,723
7	Select duration-filterable problems	1,302
8	Resplit into RL train / test	1,000 / 302

The first generated-test campaign targets correctness on the 3,928 execution-validated problems and yields 430,215 tests. The second targets optimization on the cleaner 3,061-problem subset and yields 352,740 tests. In both cases, each problem receives up to 10 LLM-written input generators and up to 15 candidate tests per generator, with acceptance based on agreement across human solutions.

Deduplication and quality filtering. To reduce leakage between training and evaluation, we deduplicate at the problem level against evaluation benchmarks such as LiveCodeBench (Jain et al., 2024). Matching is performed on problem descriptions. We also remove malformed, missing, or ambiguous problem statements identified by an instruction-tuned model. The deduplication stage and quality filtering stage reduce the corpus from 12,275 to 11,468 problems.

B.2 Filtering parameters

The correctness-cleaning round is the stricter one: it is responsible for relabeling solutions, removing ambiguous problems, and tightening the false-positive rate. The later optimization round relaxes timeout handling because large-input timeouts are part of the efficiency signal rather than necessarily evidence of an invalid test.

B.3 Filterability diagnostics

For each test category of each problem, we average runtimes over verified correct human solutions to obtain one representative duration per test. We then define the robust coefficient of variation

$$\text{Robust CV} = \frac{\text{IQR}(\bar{d}_1, \ldots, \bar{d}_N)}{\text{median}(\bar{d}_1, \ldots, \bar{d}_N)}, \tag{6}$$

Table 4: **Post-processing filtering parameters.** The correctness round is the main correctness-cleaning stage. The optimization round reuses the same exception filtering and correct-solution drop cap, but relaxes timeout handling because large-input timeouts are part of the efficiency signal rather than necessarily an error.

Parameter	Correctness	Optimization
Min correct solutions	5	1
Min incorrect solutions	1	1 (all retained)
Max correct-solution drop rate	25%	25%
Max false-positive rate	5%	100%
Timeout fraction threshold	20%	90%
Timeout treated as success	no	yes
Timeout removal cap	25%	25%
Exception fraction threshold	40%	40%
Exception/failure removal cap	30%	30%

Table 5: **Filterability of test suites across the pre-resplit $2{,}423$-problem DMC-Optim training partition.** A problem is *duration filterable* (D) if Robust CV ≥ 0.3 and *length filterable* (L) if Pearson $r \geq 0.9$ between test length and duration. Columns D and L report marginal filterability rates; "Both" and "Neither" report the corresponding intersections. Original tests rarely span enough workload to support timing-based optimization.

Test category	Duration (D)	Length (L)	Both	Neither
Public tests	24 (1.0%)	181 (7.5%)	18 (0.7%)	2,236 (92.3%)
Private tests	92 (3.8%)	131 (5.4%)	13 (0.5%)	2,213 (91.3%)
Generated tests	88 (3.6%)	110 (4.5%)	12 (0.5%)	2,237 (92.3%)
Correctness tests	368 (15.2%)	882 (36.4%)	209 (8.6%)	1,382 (57.0%)
Optimization tests	1,169 (48.2%)	1,031 (42.6%)	863 (35.6%)	1,086 (44.8%)

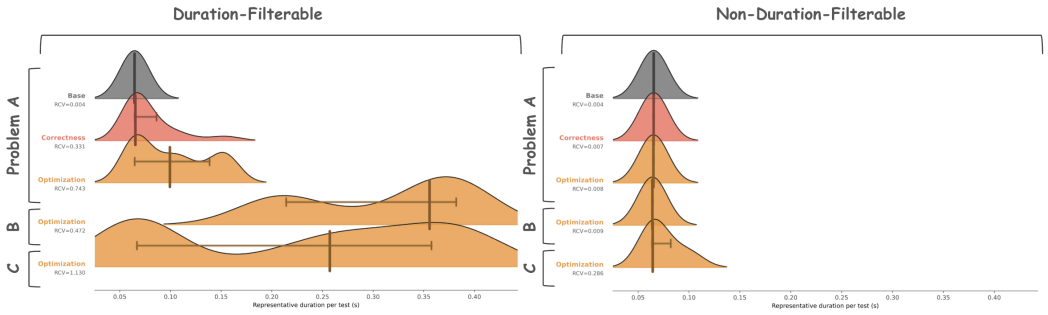


Figure 10: **Duration-filterable versus non-duration-filterable optimization profiles under a shared x-axis scale.** Within each panel, Problem A shows the full base, correctness, and optimization profiles for one representative problem, while Problems B and C show optimization-only profiles for additional examples on the same side of the duration-filterability threshold. The retained problems span visibly broader optimization-test duration ranges, while the rejected problems remain compressed near the sub-second regime. What matters for timing-based optimization is therefore not only making a few tests slower, but making the optimization suite cover a broader workload range within a problem.

and call a problem duration-filterable when this value is at least 0.3. Because some environment variants use test length before execution, we also track *length filterability* through the Pearson correlation between test length and duration, with $r \geq 0.9$ as the proxy threshold.

Optimization tests are the final gate because they are the only category that remains informative at scale. On the pre-resplit training partition, the original public, private, and generated tests are almost never duration-filterable. Optimization tests behave differently: 1,169 problems (48.2%) are duration-filterable, and 863 (35.6%) satisfy both duration and length filterability. Requiring both would therefore discard many problems whose runtimes are already informative even when length is only a weak proxy.

Table 6: **Pooled timing statistics on the final $1{,}000$-problem RL training split.** Original DMC tests pool public, private, and DMC-generated tests. Statistics are pooled over all tests in a group rather than averaged per problem. The median remains sub-second in every group, so the separation that matters for optimization RL appears in the upper tail.

Test group	**Pooled tests**	**Mean (s)**	$p90$ **(s)**	$p95$ **(s)**	$p99$ **(s)**	**Tests** > 1 **s**
Original DMC tests	106,974	0.088	0.090	0.145	0.463	0.42%
Correctness tests	118,551	0.137	0.209	0.407	1.378	1.54%
Optimization tests	119,420	0.334	0.736	1.296	3.710	6.87%

Across the final duration-filterable train/test pool, DMC-Optim stores $176{,}249$ verified correct human reference solutions together with duration annotations on $607{,}849$ tests. Optimization tests are the only category that keep a substantial persistent slow-test tail across many problems.

C Timing backend and calibration details

This appendix expands the second half of Section 3. Appendix B first showed why DMC-Optim needs optimization tests with useful duration spread: without such spread, a timing reward has little signal. That is still not enough. If the backend cannot measure these durations reliably, the apparent spread will not be usable for training or evaluation, no matter how carefully the tests were generated. In correctness-only RLVR, the sandbox mostly decides whether a solution passes before the timeout; small runtime perturbations are secondary. In optimization RL, the runtime itself enters the reward, so worker load, retries, fallback behavior, and backend drift can change the learning signal. Here we record how our initial local-sandbox setup was discarded and replaced by a more controlled remote code execution service (CES), how CES is used during construction, training, and evaluation, how infrastructure failures are handled, and how stored human-reference durations are calibrated against fresh executions.

Reading map. Appendix C.1 states what a code execution backend must provide for stable optimization RL. Appendix C.2 then starts from the initial local-sandbox setup, which is our base setup for standard RLVR, and shows why it fails for optimization RL: it changes tight-timeout rankings, creates unstable within-problem orderings, inflates duration filterability, fails calibration or correction attempts, and breaks down inside the RL loop. Appendices C.3 to C.6 describe CES as the more controlled replacement: the execution path, fallback policy, empirical failure rates, and alternative fallback designs we use or considered. Appendices C.7 to C.10 analyze the remaining variability of that CES setup, separating short-run noise during live execution from long-term drift between stored human-reference durations and fresh CES measurements, then testing whether affine calibration is stable enough for the metrics used in the paper both at evaluation time and when the calibration is changed during RL training. Appendix C.11 summarizes the lessons we take from these checks for timing-based training and evaluation, and Appendix C.12 lists the measurement questions that remain open. In this appendix, top-30% denotes the top-30% leaderboard-percentile reward and two-gate denotes the leaderboard-percentile two-gate bucketed reward; figure legends use the short labels of Table 30, where p_{30}, Ranked p_{30}, and QP p_{30} denote the former and QP the latter.

C.1 What a code execution backend has to establish

For correctness-only RL, occasional timeout noise can remain secondary because the reward is dominated by pass/fail outcomes. For optimization RL, the measured duration itself can determine whether a rollout receives credit. This matters in exactly the regime created by DMC-Optim: even after large-input test generation, the pooled optimization-test $p95$ is 1.296 s, the $p99$ is 3.710 s, and only 6.87% of optimization tests exceed 1 s (Table 6). Sub-second perturbations can therefore change rankings, thresholds, and rewards. The measurement backend must isolate code timing from worker contention, expose failures explicitly, and support calibration when service state changes.

We considered two practical backends. The first was a local sandbox, which we can consider our base RLVR setup, running on the same training workers that host model inference, rollout orchestration, and other CPU and memory load from the RL job. The second is a dedicated remote sandboxed execution service, denoted CES below, that runs each code-test pair under a fixed resource envelope, with 1 GB memory and a 10 s hard limit, and returns both status and duration. CES has more overhead per call, but it separates timing from worker-side contention, lets us control retries and concurrency, and can be provisioned independently of the GPU job. The evidence that follows led us to use the calibrated CES setup for all optimization-RL runs in the paper.

Stable timing is needed not only to train with a speed reward, but also to compare ablations. Controlling execution noise makes differences between runs interpretable as changes in the dataset, reward, or training recipe, rather than uncontrolled changes in the backend.

C.2 Local sandbox and why it is bad for timing measurements

Our base RLVR execution path is a local Python I/O-pair sandbox. A training worker sends the candidate program and a batch of input/output pairs to a sandboxed Python runner. The runner compiles the program once, replays the inputs sequentially by redirecting standard input and output, compares the produced output with the expected output, and returns one status and one elapsed duration per test case. The runner is launched through a fork server whose default backend is Bubblewrap

(Bubblewrap): the submitted code runs in a Linux namespace sandbox with dropped capabilities, a fresh temporary filesystem, resource limits, and process cleanup. This is a reasonable setup for correctness-only RLVR because it safely turns untrusted programs into pass/fail outcomes.

Three details matter for timing measurements. First, the local timer wraps the execution of each input/output pair after compilation, so it excludes sandbox startup and the one-time compilation step, but it still measures local worker wall-clock time. Second, the input/output pairs are replayed sequentially in one runner process; global state, imports, caches, or input/output redirection state can therefore carry across cases. Third, timeouts are detected by the parent process waiting for runner responses, so timeout behavior also depends on the local worker's ability to observe and clean up the sandboxed process on schedule. These details are acceptable for pass/fail RLVR, but they make the local path a weak timing instrument because process scheduling, shared worker load, and batch execution state can enter the measured duration.

BigO(Bench) further investigates how to make local sandbox timing more stable, including by refining the clock used to time programs and by trying to control CPU contention in its complexity-labeling sandbox (Chambon et al., 2025).[1] The main difference here is the workload around the sandbox. BigO(Bench) runs execution as an offline labeling job where execution can be isolated as the main workload. In optimization RL, execution runs concurrently with model inference, rollout orchestration, logging, and other worker-side load. Even if the sandbox itself is stabilized, the surrounding RL system can still perturb the durations used as rewards.

The local-timing failure has three layers: the local harness runs in a much faster regime, returns fewer usable timing cells, and gives unstable repeated measurements of the same code and tests. We then compare local and CES outputs directly: the mismatch is not just extra noise or a simple deterioration, because the two backends change which problems look duration-filterable and post-hoc fits cannot recover the CES ordering. Finally, we test the practical effect of local execution inside the RL loop, first with synthetic concurrent load and then with real local-execution RL runs.

Local sandbox speed pushes evaluation into the noisy sub-0.1 s regime. As a post-hoc diagnostic, we can take four checkpoints from the result study in Section 5: a 32B baseline, a 32B optimization-RL model, a 7B baseline, and a 7B optimization-RL model. We evaluated the same generated solutions with two harnesses, one backed by CES and one backed by the local sandbox. The local harness is much faster. At a 10 s timeout the two environments are still close because almost all correct solutions have enough time to finish, but the difference grows rapidly as the timeout enters the sub-second range. At 0.1 s, local execution inflates pass@1 by 21.3× on the 32B baseline, 32.8× on the 32B RL model, 50.3× on the 7B baseline, and 20.0× on the 7B RL model (Table 7). This extra speed is not a useful measurement signal: it moves many executions into the regime where tiny wall-clock perturbations dominate. The table already shows the resulting instability. Under local execution at 0.1 s, the 7B baseline scores higher than the 7B optimization-RL model, even though the calibrated DMC-Optim results show the optimization model is stronger at strict duration thresholds. Local execution therefore introduces a first problem: it places the evaluation in a much faster timing regime, where the instabilities studied next can only become more important.

Table 7: **Tight-timeout pass@1 inflation under local execution.** The same solution sets are evaluated under CES and local execution. At a 0.1 s timeout, local execution reports many more completions because the same programs receive much shorter measured runtimes under the local harness. The table is used as a backend diagnostic, not as a model-quality result.

Checkpoint	CES pass@1 at 0.1 s	Local pass@1 at 0.1 s	Inflation
32B baseline	0.84%	17.91%	21.3×
32B optimization RL	0.57%	18.71%	32.8×
7B baseline	0.28%	14.09%	50.3×
7B optimization RL	0.56%	11.20%	20.0×

Local sandbox distorts execution results and lowers coverage. Putting aside the timing values themselves, we look at whether the harness returns a usable timing matrix at all. We ran the same optimization-test family once on CES and once on the local sandbox. On a 50-problem slice of the raw outputs, local execution produced fewer usable timing measurements: success rate fell from

[1] https://github.com/facebookresearch/BigOBench/tree/main/src/complexity/sandbox

99.62% to 97.41%, non-success rate rose from 0.38% to 2.59%, and usable per-solution per-test timing coverage fell from 99.2% to 93.3%. Some problems deteriorated much more sharply, with timing-matrix coverage falling by up to 50%. This is already a trustworthiness issue, independent of how the remaining durations are distributed. Missing cells are not harmless blanks: local out-of-memory events can be harder to isolate from the host worker, small runtime perturbations can push near-limit executions across a timeout boundary, and worker-side instability can change the execution status rather than only the measured duration.

Ask which code is fastest, get a different answer each time. We next tested whether the local sandbox can at least produce a stable timing signal relative to itself. In a dedicated local measurement campaign, we repeatedly re-evaluated unchanged candidate solutions on the same problems, asking only whether one rerun recovers the same within-problem timing ranking as another. For each optimization test, the candidate duration is ranked against the human-reference durations for that same test, with 0% denoting the fastest position and 100% the slowest. The mean percentile is the average of these per-test percentiles across the optimization tests. A mean percentile of 20% means the solution looks faster than most references; a mean percentile of 70% means it looks much slower. The diagnostic asks whether rerunning the same code on the same tests keeps that leaderboard position fixed.

Across 23 problems and 425 reruns, repeated local re-execution moved the mean-percentile score by 41.2 percentage points peak-to-peak on average: for each problem, we take the difference between the best and worst mean-percentile score obtained across reruns, then average that range across problems. Equivalently, the within-problem standard deviation of mean percentile averaged 10.961 percentage points and the within-problem range averaged 41.173 percentage points. On a ranked reward, this is not a cosmetic fluctuation: an 11-point standard deviation is large enough to move a solution across a top-30% gate, and a 41-point range is the difference between looking like a clearly fast solution and a clearly mediocre one. In leaderboard terms, a solution can move from around the top 30% to around the bottom 30% across local reruns without any code change. The largest observed within-problem range, 73.743 percentage points, would be enough to move a candidate across almost the whole useful part of the leaderboard without changing the code.

We also check whether local reruns preserve the ordering of the tests themselves. For each rerun, we take each optimization test's median duration across re-executed human reference solutions, and fit a line from the stored median duration of that test to the newly measured local median duration. A stable timing backend should give a positive and similar slope across reruns: if the stored data says test A is a heavier workload than test B, rerunning locally should not make A look easier than B. A sign flip is the clearest failure case, because the fitted line changes direction; one rerun says larger stored workloads remain larger, whereas another says they become smaller. This happened on 18 of 22 problems with enough measurements. We also compute the fitted-slope range, the maximum fitted slope minus the minimum fitted slope across local reruns for the same problem. This range averaged 67.213, reached 208.401 at the 90th percentile, and reached 651.462 in the worst case. These are slopes in a linear map from stored seconds to local seconds after aggregating by test median, so their absolute scale is less important than the instability: the same problem can move from an almost flat fit to a very steep fit across reruns. The worst-problem R^2 medians were near zero, meaning the stored workload ordering explains almost none of the local timing variation on those problems. Together with the mean-percentile movement above, the same code and test suite can therefore look substantially stronger or weaker across local reruns.

Local and CES measurements are incompatible. After setting up a new remote execution backend that we call CES, we could compare the two backends on the same optimization tests. CES is not an absolute ground truth, but the two backends would need to be commensurate if they were ever mixed in the same reward path, for example by using local execution as a fallback when CES is overloaded. The comparison also checks whether our default local backend and the remote backend differ only by degree, with one being noisier but still agreeing on average, or whether they induce different timing orderings altogether. In the latter case, it probably means at least one backend cannot be trusted for timing measurements. They are not commensurate: local and CES measurements disagree by more than a small perturbation around the same signal. On the full dataset, the same optimization tests look much more duration-filterable locally: 81.1% of problems are duration-filterable under the local sandbox versus 48.2% on CES. Because the filterability criterion is based on a robust coefficient of variation, this gap is difficult to explain by a simple global speedup. A uniform rescal-

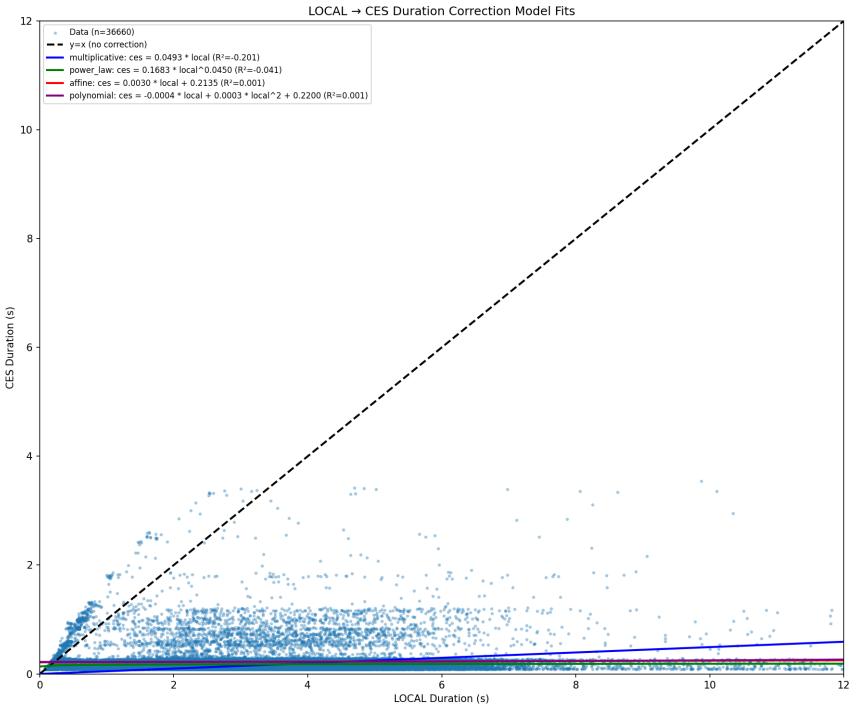


Figure 11: **Attempting to fit local runtimes back to CES runtimes fails.** The plot aggregates 36,660 joined local/CES duration pairs. The identity line is shown only as a reference: the thin diagonal band above it is the benign case where local execution is faster than CES but still roughly ordered with it. The larger pattern is different. Many points with very small CES durations spread over a wide range of local durations, forming a long horizontal plume rather than a tight line. This is the failure mode: local execution sometimes adds worker-side delay that is not explained by the code-test pair itself, so the same CES-fast execution can look locally fast, moderately slow, or extremely slow. The fitted mappings therefore collapse toward nearly flat predictions, and all four have negative leave-one-problem-out R^2. Local timing is not missing only a scale factor; it contains extra variation that cannot be removed by a simple post-hoc calibration.

ing would mostly preserve robust CV. The more plausible explanation is that local execution injects extra timing spread, making some otherwise non-filterable problems appear useful for timing. The 50-problem comparison points in the same direction: local execution reduces usable duration-cell coverage while making more problems appear duration-filterable. Finally, if local timing were only biased by a scale factor, we could fit a mapping from local durations to CES durations. We tried this on 36,660 joined local/CES duration pairs. Multiplicative, power-law, affine, and polynomial mappings all failed under leave-one-problem-out evaluation, with negative cross-validated R^2 (Table 8). The fitted maps collapse toward nearly flat predictions rather than recovering CES durations, and ranking quality remains badly misaligned (Figs. 11 and 12). Local timing is therefore not a quantity we can adjust or calibrate.

Table 8: **Failed local-to-CES correction on 36,660 paired measurements.** The mappings are fit under leave-one-problem-out cross-validation on the overlapping local/CES data. If local timing were only a rescaled version of CES timing, one of these maps would predict held-out CES durations from local durations. Instead, all cross-validated R^2 values are negative, meaning the fitted maps are worse than ignoring the local duration and predicting an average held-out duration.

Mapping family	**Cross-validated R^2**
Multiplicative	-0.360
Power-law	-0.057
Affine	-0.213
Polynomial	-0.254

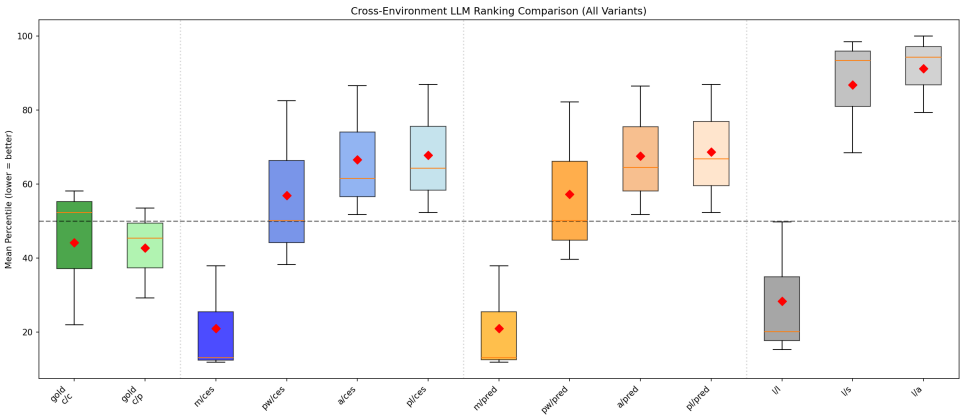


Figure 12: **Local rankings remain misaligned even after cross-environment correction.** The figure ranks model-generated solutions against distributions of human-reference solutions on the same problems and tests. For one problem and one optimization test, we take the duration of the model solution under the protocol named on the x-axis, insert it into the duration distribution of human solutions for that same test, and record its percentile. A percentile of 0% means the model solution is faster than all human references on that test; 50% means it is near the middle of the human distribution; 100% means it is slower than all human references. The y-axis is the mean of these per-test percentiles over the optimization tests of the problem, so lower values mean that the model solution ranks faster relative to the human reference set. Each box summarizes the three problem-level mean-percentile scores available in this paired local/CES diagnostic, and the red diamond marks their mean. The dashed line at 50% is the middle of the human reference distribution. The x-axis labels are comparison protocols: *gold c/c* ranks model durations measured on CES against human durations measured on CES, while *gold c/p* keeps model CES durations but replaces human CES durations by affine-corrected stored human durations. The blue and orange groups test whether local model durations can be repaired by local-to-CES maps: m, pw, a, and pl denote multiplicative, power-law, affine, and polynomial corrections; the suffix */ces* ranks those corrected local model durations against human CES durations, and */pred* ranks them against affine-corrected stored human durations. The gray diagnostic labels compare raw local model durations against raw local human durations (*l/l*), stored human durations (*l/s*), or affine-corrected stored human durations (*l/a*). The reference is therefore *gold c/c*: a successful replacement for CES timing would put its box close to this one, problem by problem. Lower is not automatically better in this diagnostic. A box far below *gold c/c* means local timing makes the same model solutions look spuriously fast; a box far above it means the correction makes them look spuriously slow. *Gold c/p* stays close to *gold c/c*, which is the desired behavior of the stored-duration affine correction when model executions are still measured on CES. Among local-to-CES repairs, the power-law variants are the least bad, but they still move the mean percentile from 44.1% under *gold c/c* to 56.9%–57.3%; the multiplicative variants move in the opposite direction to 20.9%, and the affine/polynomial variants move to 66.6%–68.6%. The gray raw-local diagnostics are worse as practical alternatives. Even *l/l*, the most favorable raw-local comparison because it re-executes both the model solution and all human reference solutions locally every time a model solution is ranked, is still far from the CES reference and makes the model look spuriously fast. The mixed-backend raw-local comparisons are worse: *l/s* and *l/a* rank raw local model durations against stored or affine-corrected stored human durations, and push the mean percentile to 86.8%–91.2%. This is the failure one would expect from comparing quantities produced by different timing backends. This is why local timing cannot be used as a drop-in replacement for CES timing: even after correction, it does not recover the ranking used by the reward and evaluation.

Local sandboxes collapse under CPU contention. The previous diagnostics already identify three failure points that make local timing hard to use for optimization RL. The remaining operational question is how local execution behaves when the worker is also under CPU pressure, as it would be during rollout collection. To measure sensitivity to contention, we reran the same 724 local timing jobs while adding background CPU load on the same node at 50%, 100%, and 200% of node CPU capacity. Even with no added load, 294 of 724 jobs did not complete. This rose to 485/724 at 50% load, 531/724 at 100%, and 613/724 at 200%. We observed very slow executions, local out-of-

memory failures, and worker-side instability. These events cannot be isolated by a service-side retry because worker-side contention is the failure mechanism.

Local sandbox optimization RL fully fails. The standalone diagnostics above were obtained outside the full RL loop, which can probably only make things worse. We therefore tried the direct ablation: eight local-execution variants of existing CES-based Qwen 2.5 7B configurations, each targeting 5,000 RL steps and using direct local outcomes. These variants skipped CES during training, while the evaluation section still used CES, so the ablation isolates training-time execution rather than changing the reported evaluation backend as well. The correctness-only baseline, which runs no optimization tests, completed successfully. The seven optimization-test variants did not: each exhausted all six scheduler attempts (the original launch plus five restarts), accumulated 12–32 hours of wall-clock time, and reached only 160–1,620 training steps. In standalone measurements, local execution appeared faster per test than CES; inside the RL loop, the full runs were much slower. For configurations with optimization tests, local execution was 4–8$\times$ slower than CES end to end. The failures were out-of-memory kills on the training nodes, followed by distributed-communication failures across ranks, because local execution consumes memory on the same machines already holding the model and optimizer states. This is not a proof that no local-execution training configuration can ever run: lowering rollout concurrency, reducing the number of executed tests per step, or allocating more CPU and memory to the training workers would likely avoid some crashes. But those changes would directly reduce throughput, and the current local runs were already slow when they made progress: data-loading time reached 355 s per step in the worst case, compared with single-digit seconds in the corresponding CES runs. This end-to-end ablation turns the timing argument into an operational one: for optimization-test RL at this model scale and throughput target, CES is not only a cleaner timing source but also what makes the training run feasible.

Local execution remains useful only for non-timing fallback. It can still help recover a verdict when a remote request fails. It cannot be used as the timing source for optimization rewards or evaluation, and it cannot be mixed with CES timings as if the two were commensurate. The two backends are fully misaligned for timing: local durations are faster, less complete, unstable across reruns, not calibrated back to CES, and coupled to the training workload itself. Mixing local and CES durations would therefore perturb any model trying to learn a serious timing-based reward.

C.3 Remote execution service

CES is a pre-existing remote execution backend that we use for timing-sensitive dataset construction, training, and evaluation; building the service itself is not a contribution of this paper. A training or evaluation worker sends candidate code and tests to the service; CES schedules the executions on service-managed virtual-machine capacity, runs them in remote sandboxes, and returns execution statuses and durations. The service property we rely on is isolation: each test runs under a 1 GB memory limit and a 10 s hard limit, and the execution capacity is provisioned so CPU cores are isolated and assigned to sandboxed jobs rather than implicitly shared with rollout processes. Sub-second reward thresholds are applied after CES returns durations rather than by changing the service limit. Building such a service at scale is an infrastructure problem in its own right and is outside the scope of this paper.

The main architectural property we use is separation from the rollout workers. Candidate code is no longer executed on the same machines that run model inference, rollout orchestration, data movement, and model or optimizer state. CES instead places execution behind a separately provisioned queue where concurrency can be controlled, retries are visible, and one submitted program is less likely to spill over into another through shared CPU or memory pressure. This is why a remote service can be preferable even if one isolated code-test call has more overhead than a local sandbox.

CES is not assumed to be noiseless: it has service overhead, infrastructure failures, and timing drift. Its role is to put timing under one monitored remote backend whose failures can be retried and whose stored durations can be calibrated against fresh executions. Appendix C.4 defines the fallback rule, Appendix C.5 quantifies infrastructure failures, and Appendix C.7 explains the stored-duration calibration.

C.4 Fallback after remote execution failures

Fallback path. When CES returns an inconclusive result, the system first retries on CES. If a code-test pair keeps failing to return a definitive remote result, local fallback may be used to recover a correctness verdict. Definitive CES results are kept as-is.

Early stopping interacts with fallback. When CES returns a definitive hard failure on any test, the system can skip local reruns for remaining inconclusive tests because the trajectory will receive a negative reward regardless. If the only definitive non-success status is timeout, local reruns may still proceed for inconclusive tests because an unresolved test could reveal a wrong output or runtime error, which can be more penalized than a timeout in reward variants that treat timeouts as a separate outcome.

Timing measurements for fallback executions. Local execution on a training node can be faster than CES. A solution that completes locally may not complete under the remote sandbox because of service overhead, different CPU allocation, or memory behavior. A local success therefore cannot be treated as evidence that CES would have reported success within the same time limit. This is even more true for the local duration itself, which has no reason to be close to the CES duration that would have been returned for the same code-test pair. For example, consider one problem with a large edge-case optimization test. Early in RL, a model rollout may solve the test but allocate an unnecessarily large data structure; CES returns a slow duration, and the timing reward can penalize it. After policy updates, a later rollout on the same problem may deteriorate on that edge case and allocate even more memory. If CES still returns a definitive result, that slow duration or failure remains the correct training signal. If instead the request crosses the remote execution envelope and ends without a usable CES duration, local fallback can rerun the same code-test pair under different CPU or memory pressure and report a successful shorter runtime. Importing that local runtime would reverse the signal: the model would receive better timing evidence for a generation that was worse under the remote execution backend.

To avoid trusting faster local timing, the system applies a conservative conversion on timing-sensitive optimization tests: a test that succeeds locally during fallback is reclassified as a timeout with duration set to the time limit. Correctness tests use local fallback only to recover pass/fail verdicts when CES remains inconclusive; the local runtime is not imported as timing evidence.

The two test categories are treated differently because a timeout has different consequences in each one. On correctness tests, converting a locally recovered success into a timeout would make an otherwise correct rollout fail the all-tests-must-pass gate and receive reward -1. We therefore use local fallback there only to recover the pass/fail verdict, not to create a timing value. On optimization tests, timeouts are part of the timing signal. In the main experiments, the timeout tolerance is 1.0, so optimization timeouts are tolerated by the binary gate. A converted local success then contributes a 10 s duration to the ranking computation and can lower the quality score, but it does not by itself make the solution incorrect. This is the intended tradeoff: conservative timing for optimization tests, without creating false correctness failures on correctness tests.

C.5 Quantitative impact of infrastructure failures

Fallback only enters the reward when CES does not return a definitive result. We therefore first ask how often those missing remote results can matter at the scale of a training run.

Rare failures add up over training. Let f be the per-test infrastructure-failure rate and n the number of correctness tests. Consider a correct solution that would pass whenever CES returns a definitive answer. If failures hit tests independently, the probability that at least one correctness test reaches fallback is

$$P(\text{at-risk fallback} \mid \text{correct solution}) = 1 - (1 - f)^n. \tag{7}$$

The independence assumption is a simplification, but it is enough to show why a uniform fallback rule can be dangerous on correctness tests. A per-test failure rate can look negligible while the probability of seeing at least one fallback in a rollout is not negligible. Table 9 shows the sensitivity. At a 2% per-test failure rate with 30 correctness tests, 45.5% of correct rollouts would reach fallback at least once. Even at $f = 0.1\%$, the risk is 3.0% with 30 tests, 9.5% with 100 tests, and 13.9% with 150 tests.

The scale becomes even larger once we multiply by the optimizer steps. A Qwen 2.5 7B run has 10,000 optimizer steps, 16 optimizer data-parallel replicas, and up to 32,768 packed tokens per replica per step. Across the run, the optimizer can process up to 5.24B packed training tokens. If each consumed code snippet contributes about 11k tokens to the trainer, filling those batches requires

about 4.8×10^5 code snippets. Because the trainer packs whole snippets and does not split a long one, a stricter count with uniformly 11k-token snippets would fit two snippets per replica per step, or $10{,}000 \times 16 \times 2 = 3.2 \times 10^5$ consumed snippets. At $f = 0.1\%$ and $n = 150$, the 13.9% per-rollout risk therefore corresponds to roughly 4.5×10^4–6.6×10^4 trainer-consumed snippets, or about 0.5–0.7B packed training tokens. The current training path avoids the specific false-negative failure mode by not converting locally recovered correctness successes into timeouts.

Table 9: **Per-rollout fallback exposure from rare per-test infrastructure failures.** Values are computed with Eq. (7) as a function of per-test infrastructure failure rate f and number of correctness tests n. If locally recovered correctness successes were converted to timeouts, for example, any fallback exposure in this table would create a false-negative risk for an otherwise correct rollout.

Failure rate f	$n = 10$	$n = 30$	$n = 50$	$n = 100$	$n = 150$
2.0%	18.3%	45.5%	63.6%	86.7%	95.2%
0.5%	4.9%	14.0%	22.2%	39.4%	52.9%
0.1%	1.0%	3.0%	4.9%	9.5%	13.9%
0.01%	0.1%	0.3%	0.5%	1.0%	1.5%
0.001%	0.01%	0.03%	0.05%	0.10%	0.15%

CES reliability study. We next check how these rates look in actual RL runs. The case study covers 13 training runs: six CWM 32B runs and seven Qwen 2.5 7B runs, all trained for 10,000 optimizer steps with CES used for timing-sensitive execution. Metrics are logged every 10 optimizer steps. Within each model and reward family, the runs use the same dataset, environment, model initialization, seed, learning rate, GRPO settings, and duration calibration; the main uncontrolled variable is launch time, which changes the CES service state seen during rollout collection. The distributed asynchronous training system also introduces nondeterminism in rollout ordering and batch composition, so identical configurations are not expected to produce identical trajectories.

Table 10 reports post-warmup statistics from all 13 runs, computed after step 1,000. The logged CES-only rate is a rollout-level quantity: it is high only when the model output parses, the code-test execution is correctly sent to CES, and CES resolves it without launching a local sandbox fallback. Local fallback is the complementary recovery path: it tracks cases where a local sandbox had to be launched after CES did not provide a definitive usable result. CES-only can therefore drop for two different reasons: parse failures before CES, or remote-execution failures after a parseable program is sent to CES. After step 1,000, parsing is above 97% in these runs, so the remaining drops are mostly informative about CES and fallback behavior. In that post-warmup regime, Qwen 2.5 7B has 99.3% CES-only execution on average, while CWM 32B has 98.1%. The difference is expected because CWM 32B produces more correct programs, so more rollouts proceed to optimization-test execution and place concurrent load on CES. Local fallback averages 0.13% for Qwen 2.5 7B and 0.21% for CWM 32B; infrastructure and unknown failures average 0.10% and 0.05%, respectively. Here, an unknown result means that CES did not return a status that could be parsed into a normal program outcome. Since parse failures, wrong answers, exceptions, and timeouts are logged separately, these unknown results are treated as infrastructure-side inconclusive results in the analysis below.

Parsing warmups. Figure 13 also shows a universal early-training dip during the first roughly 500 optimizer steps, with recovery by step 1,000. This is not a CES infrastructure failure. The logged metrics show that this early dip tracks parse rate: rollouts that fail to produce parseable code never reach CES, while local rerun and infrastructure-failure rates remain near zero during the dip. For CWM 32B, the main early issue is the output-format switch between SFT and RL documented in the CWM report (FAIR CodeGen team et al., 2025): around step 10, roughly 26% of rollouts fail to parse while the model re-adapts to the required RL format. Qwen 2.5 7B has a much smaller dip because its SFT checkpoint already produces well-formatted code for this environment. Once parsing stabilizes above 97%, lower CES-only values are no longer a format-learning artifact and can be interpreted as remote-execution or fallback events. This is why the tables below report post-warmup statistics.

Random bursts of infra errors. After parsing warmup, the main visible failures are short bursts. Table 11 lists every post-warmup interval where CES-only falls below 90%. There are four events:

Table 10: **Post-warmup CES execution statistics across 13 RL runs.** Statistics are computed over logged training metrics after step 1,000. "CES-only" is the logged fraction of rollout attempts that parse, are sent to CES, and are resolved by CES without launching a local sandbox fallback. The infrastructure-failure column combines explicit infrastructure failures with unknown or inconclusive CES results; unknown results are unparsed CES outcomes after normal program errors and timeouts have been separated out. The run-wide means are high, but the minimum column already shows that several runs contain short degraded intervals that are hidden by the mean.

Run	Reward	CES-only mean	CES-only min	Local fallback	Infra/unknown
CWM 32B top-30% run 1	Top 30%	98.53%	95.09%	0.0003%	0.0001%
CWM 32B top-30% run 2	Top 30%	97.18%	54.33%	0.77%	0.16%
CWM 32B top-30% run 3	Top 30%	97.99%	91.59%	$< 0.001\%$	0.0001%
CWM 32B two-gate run 1	Two-gate	98.40%	93.44%	0.0001%	0.0001%
CWM 32B two-gate run 2	Two-gate	98.04%	92.38%	0.49%	0.13%
CWM 32B two-gate run 3	Two-gate	98.52%	94.37%	0.0003%	0.002%
Qwen 2.5 7B top-30% run 1	Top 30%	99.43%	97.98%	0.0006%	0.001%
Qwen 2.5 7B top-30% run 2	Top 30%	99.26%	79.94%	0.23%	0.18%
Qwen 2.5 7B top-30% run 3	Top 30%	98.68%	80.86%	0.71%	0.53%
Qwen 2.5 7B top-30% run 4	Top 30%	99.56%	97.70%	0.0004%	0.0007%
Qwen 2.5 7B two-gate run 1	Two-gate	99.42%	97.63%	0.0007%	0.001%
Qwen 2.5 7B two-gate run 2	Two-gate	99.44%	97.66%	0.0002%	0.0004%
Qwen 2.5 7B two-gate run 3	Two-gate	99.35%	97.12%	0.0008%	0.002%

Table 11: **Post-warmup CES burst events.** Each row is a contiguous window after step 1,000 where CES-only execution falls below 90%. The records column counts logged metric records; metrics are logged every 10 optimizer steps. The return columns compare the mean trajectory return during the burst with the post-warmup run-wide mean. Bursts are short but severe: the worst event lasts only 8 logged records but drops CES-only to 54.3%.

Run	Window	Records	CES-only min	Peak fallback	Peak timeout	Return: burst / run
CWM 32B top-30% run 2	9,240–9,310	8	54.3%	41.6%	14.1%	$+0.02/+0.34$
CWM 32B top-30% run 2	9,720–10,000	29	63.5%	35.7%	10.1%	$+0.17/+0.34$
Qwen 2.5 7B top-30% run 2	4,160–4,220	7	79.9%	19.0%	4.7%	$-0.49/-0.31$
Qwen 2.5 7B top-30% run 3	5,310–5,390	9	80.9%	18.4%	5.2%	$-0.64/-0.33$

two in CWM 32B top-30% run 2, one in Qwen 2.5 7B top-30% run 2, and one in Qwen 2.5 7B top-30% run 3. The worst CWM 32B event drops CES-only to 54.3%, pushes local rerun to 41.6%, and raises timeout rate to 14.1%. The Qwen 2.5 7B bursts are milder, with CES-only minima around 80% and local rerun peaks around 19%. Within the same model and reward family, other runs with the same configuration do not show these bursts, so these events are best treated as service-state events rather than properties of the reward.

The burst mechanism is not a large spike in hard infrastructure errors. In the worst single logged step, CWM 32B top-30% run 2 at step 9,260, hard infrastructure errors are 0.00%, unknown or inconclusive CES results are 6.3%, unknown-triggered local reruns reach 44.5%, and timeouts reach 14.1%. These unknown results are not model parse failures or ordinary test failures; those statuses are logged separately. They are CES outcomes that were not parsed into a normal verdict and are therefore grouped with infrastructure-side inconclusive results. A monitor that only tracks hard infrastructure errors would miss this degradation. The service is still returning responses, but too many of them are inconclusive, which pushes execution through fallback and changes the optimization-test durations seen by the reward.

Beyond infra errors, silent service degradations. Figure 16 compares two CWM 32B top-30% runs with the same configuration. Run 1 has almost no visible fallback or infrastructure events: 98.53% mean CES-only, 95.09% minimum CES-only, 0.0003% local fallback, and 0.0001% infrastructure or unknown failures. Run 2 has visible late bursts: 97.18% mean CES-only, 54.33% minimum CES-only, 0.77% local fallback, and 0.16% infrastructure or unknown failures. Yet run 1 has much noisier training returns between steps 1,000 and 9,000: mean return -0.14, standard deviation 0.34, and range 1.43, compared with $+0.34$, 0.15, and 0.85 for run 2. This comparison does not prove that the noisier run is caused by CES state alone; asynchronous rollout ordering and early exploration differences can also change the trajectory, and the pass-rate means differ from

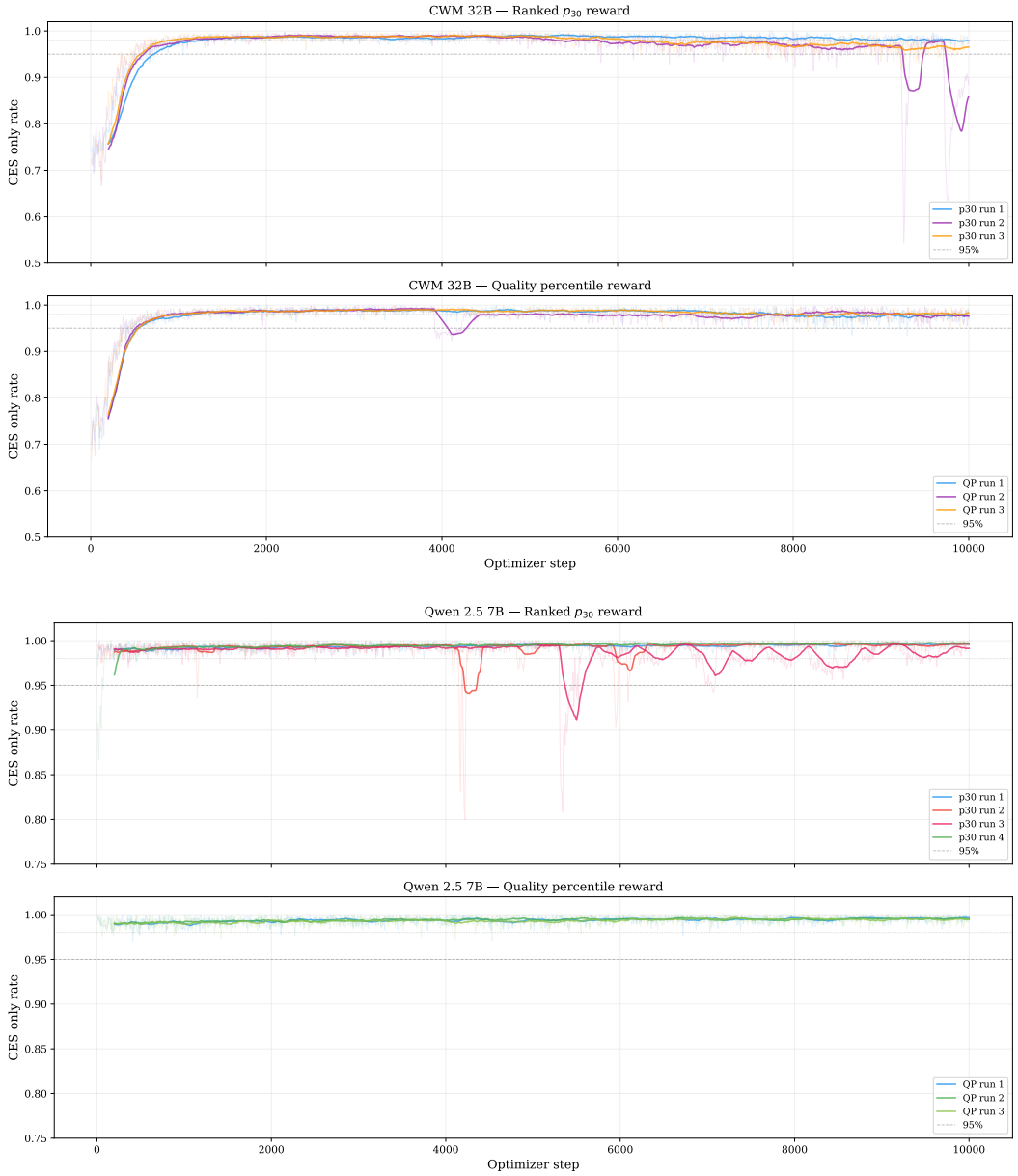


Figure 13: **CES-only execution over 10,000 training steps.** Top: six CWM 32B runs, split between the top-30% and two-gate rewards. Bottom: seven Qwen 2.5 7B runs, split between the top-30% and two-gate rewards. All runs show an early dip before step 1,000, then recover to the post-warmup operating regime. This initial dip reflects parsing warmup rather than CES health: rollouts that fail to produce parseable code are never sent to CES and therefore do not count as CES-only. The effect is largest for CWM 32B because the CWM recipe changes output format between SFT and RL (FAIR CodeGen team et al., 2025); at the start of RL, the model temporarily re-adapts to the required RL format. Qwen 2.5 7B starts much closer to the final parse rate. CWM 32B operates at slightly lower post-warmup CES-only rate because it produces more correct code and executes more tests. After parsing has stabilized, the sharp drops in CWM 32B top-30% run 2 and Qwen 2.5 7B top-30% runs 2–3 are short remote-execution burst events rather than format-learning effects.

early training onward. It does show the failure mode we have to monitor: CES can keep returning verdicts, so pass/fail outcomes and fallback rates look healthy, while slower or noisier service state shifts the recorded durations used by the ranking reward.



Figure 14: **Anatomy of CES burst events.** Top: the two CWM 32B top-30% run 2 bursts. Bottom: the two Qwen 2.5 7B top-30% bursts. Each panel shows CES-only execution, local rerun and timeout rates, and trajectory return around the degraded interval. The mechanism is consistent across models: CES-only drops, inconclusive-result reruns rise, timeout rate rises, and the return falls during the burst. For CWM 32B top-30% run 2, CES-only drops from about 98% to 54% within two logged records and recovers within eight logged records in the first burst.

For CWM 32B top-30% run 2, 37 post-warmup logged records fall below 90% CES-only and 73 fall below 95%. With 16 optimizer data-parallel replicas and about two consumed snippets per replica per step, this corresponds to roughly 1,200 consumed code snippets below 90% CES-only and 2,300 below 95%. These are small fractions of the 10,000-step run, 0.4% and 0.8% under the logged-step

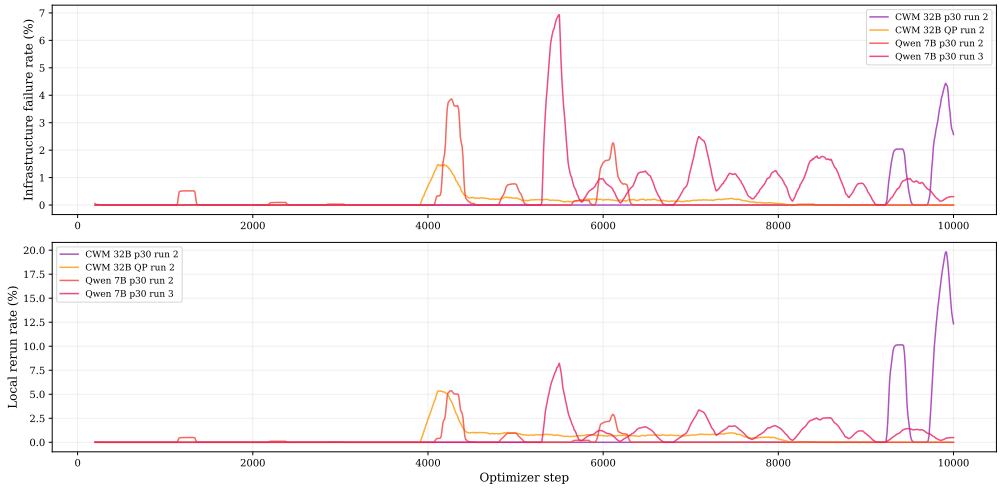


Figure 15: **Inconclusive CES results and local fallback during bursty runs.** The figure tracks infrastructure or unknown-result rates together with local fallback for the four runs that contain nonzero post-warmup events. Unknown results are unparsed CES outcomes after normal test failures and timeouts have been logged separately, so they are treated as infrastructure-side inconclusive results here. Failures are temporally concentrated: most steps have near-zero rates, then short intervals produce visible spikes. The worst CWM 32B burst is dominated by unknown or inconclusive CES results that trigger local rerun, not by hard infrastructure-error responses.

Table 12: **Two CWM 32B top-30% runs with different CES symptoms.** Both runs use the same model, data, reward, and hyperparameters. Run 1 has almost no visible fallback or infrastructure events but volatile returns; run 2 has late visible bursts but smoother returns before the bursts. The comparison motivates monitoring the reward stream and duration distributions, not only hard failure rates.

Metric	Run 1: few visible failures, noisy return	Run 2: late bursts
CES-only mean after step 1,000	98.53%	97.18%
CES-only minimum after step 1,000	95.09%	54.33%
Local fallback mean	0.0003%	0.77%
Infrastructure/unknown mean	0.0001%	0.16%
Return mean, steps 1,000–9,000	−0.14	+0.34
Return std., steps 1,000–9,000	0.34	0.15
Return range, steps 1,000–9,000	1.43	0.85
Pass-rate mean after step 1,000	0.45	0.68

count, but the per-step effect can be severe: during the worst burst, up to 41.6% of executions are handled by local fallback. Fallback can be rare in the run-wide average, concentrated enough to affect only a small fraction of optimizer steps, and still large enough during a burst to change the reward stream for those steps.

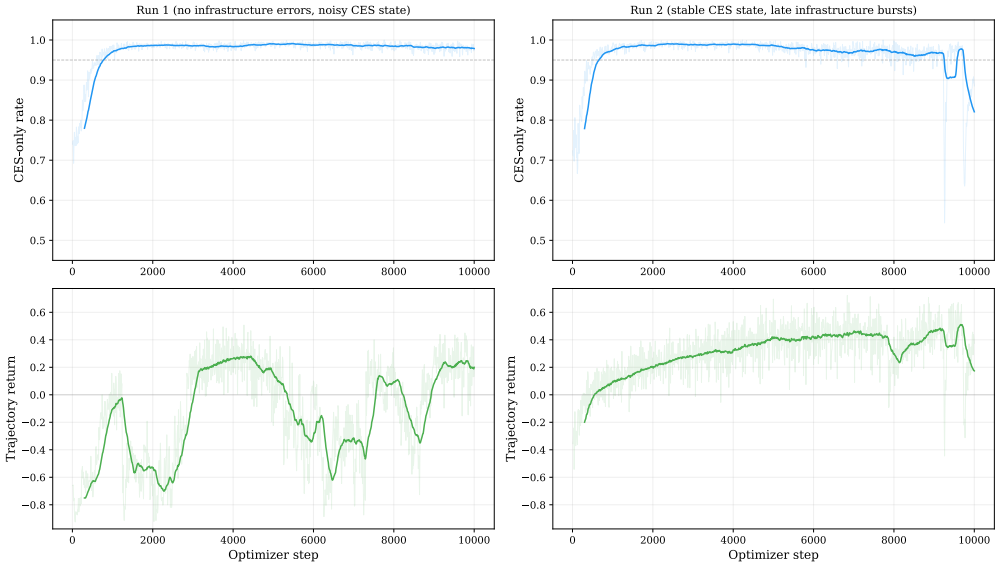


Figure 16: **Silent CES degradation can affect timing rewards.** The figure compares CWM 32B top-30% run 1 and run 2. Run 1 keeps CES-only above 95% and has almost no local fallback, but its trajectory returns are volatile throughout training. Run 2 has two late visible CES bursts, but between step 1,000 and step 9,000 its returns are smoother and higher. This does not identify a single cause for run 1: training nondeterminism and early exploration can also change returns. It shows why a service-state degradation can matter even without hard infrastructure errors: if CES still returns verdicts but records shifted or noisier durations, pass/fail outcomes may look normal while the timing reward changes.

C.6 Alternative fallback designs

Per-category conversion. In training, we handle correctness and optimization tests differently. For optimization tests, where the duration enters the reward, a locally recovered success is converted to a timeout. For correctness tests, where the goal is only to recover a pass/fail verdict, we keep the local verdict because the 10 s limit makes a sub-second local success likely to remain a CES success. This removes the correctness false-negative mode while keeping conservative timing for optimization. If CES reliability degrades, the remaining refinement would be to tune category-specific fallback thresholds and decide when a trajectory should be discarded instead of recovered.

Immediate CES retry. Before local fallback is used, inconclusive CES results are retried immediately, before the rollout reward is finalized. This is useful for transient request failures: the reward can still be computed from a remote execution without mixing timing backends. It does not solve persistent bursts. If the same batch keeps returning infrastructure or unknown results, retrying only delays the step and the system still needs the per-category conversion above or a discard rule. The retry budget also has to stay short in asynchronous RL. If a rollout is retried for too long, it is eventually consumed by an optimizer whose policy has already moved on. The same issue appears inside a prompt group: if some rollouts for a problem are measured immediately while the inconclusive ones are re-executed much later, transient changes in CES speed can make the durations compared by the advantage computation less comparable.

Discarding or skipping affected data. One alternative is to discard any trajectory where at least one test falls back to local execution. This removes both false positives and false negatives from fallback, but it loses too much data. With n_{total} total tests, the discard rate is $1 - (1 - f)^{n_{\text{total}}}$. At a 2% per-test failure rate with 80 total tests, about 80% of trajectories would be discarded. Even at 0.5%, the discard rate reaches 33%. The discard probability also correlates with the number of tests per problem, biasing training toward smaller suites. Another option is to skip failed tests and evaluate on the remaining tests. This avoids false negatives from converted timeouts, but it can create false positives: if the skipped test is the only one that catches a bug, a wrong solution receives positive reward. For correctness tests, where one missed bug can flip the reward sign, this trade-off is unfavorable over millions of trajectories.

Quarantining degraded windows. A stronger operational control would be to pause rollout collection, drop affected batches, or roll back to an earlier checkpoint when the service enters a bad state. This is plausible for both visible bursts and silent slowdowns if the degradation signal is reliable enough. It is also expensive: the policy may have already consumed noisy rewards before the degradation is diagnosed, discarding batches wastes GPU time, and rolling back a distributed RL run adds another source of operational complexity. We therefore do not implement such routine for the RL experiments of this paper.

Live calibration probes. Another natural idea is to send a fixed set of code snippets with known reference durations during training, use their measured slowdown to detect silent CES degradation, and rescale current rollout durations. This can work only if the probe set estimates the current service state accurately. Too few probes are noisy, especially in the sub-second regime where tens of milliseconds matter. Too many probes compete with the training workload for CES capacity; the monitoring traffic can slow the service and training progress enough to become part of the problem. This makes live probing more suitable as a health diagnostic than as a cheap per-step correction mechanism.

Reward choices. Reward design can reduce, but not remove, sensitivity to service drift. Same-prompt grouping and GRPO advantages compare rollouts collected close together, which helps when the service shift is common across a batch. A purely within-batch timing comparison would push this idea further, but we did not use it as the main reward because our ranked rewards also compare fresh model executions against stored human-reference durations. Those stored references still need the affine calibration studied next, and the reward ablations in Appendix F show that data quality, fallback behavior, calibration, and reward design have to be handled together.

C.7 CES noise, drift, and calibration

Switching to CES does not remove noise. It concentrates noise in a backend whose behavior can be monitored, retried, and recalibrated. CES timings affect four stages of the pipeline: dataset construction, where reference durations are stored; RL training, where each rollout is evaluated; evaluation, where timing-sensitive outcomes are measured again; and cross-run comparisons, where current results are compared either with stored human-reference durations used by ranked reward variants or with evaluation results from experiments run under an earlier CES state. The first three stages depend on live measurement noise. The fourth introduces service-state drift: stored durations or stored evaluation results may have been collected on a different CES fleet, cluster, load condition, or sandbox version than the current execution.

We therefore treat CES timing in two layers. Live CES noise asks whether repeated executions under the current service state are stable enough for ranking. Long-term CES drift asks whether historical measurements, either human-reference durations or evaluation outputs from earlier runs, remain comparable to current model executions. For stored reference durations, we apply the affine correction below; for cross-run evaluation comparisons, the same issue motivates re-execution or calibration-sensitivity checks.

Same-time grouping attenuates live drift. During RL, rollouts for the same prompt are executed and grouped contemporaneously, and the trainer does not use a replay buffer for these configurations. This helps because common-mode backend shifts within a prompt group are partially shared and are not mixed with trajectories from much older service regimes. It does not remove reward noise: ranked rewards still compare model executions against calibrated historical human-reference durations, so calibration and health monitoring remain necessary.

Calibration campaign setup. The first calibration campaign fits the correction on human-reference duration pairs. Each run also records one model solution for ranking diagnostics, but the fitted map itself is estimated from human references because the goal is to map stored reference durations into the current CES timing scale before a fresh model solution is inserted into the same leaderboard. We sample 33 problems from the filtered DMC-Optim pool. Across these problems, the human-reference pool contains 10,841 solutions, with 14–3,185 solutions per problem and median 158. The optimization-test suites contain 4,035 tests in total, with 30–150 tests per problem and median 135. Each problem is re-executed independently about 20 times through CES; 27 problems have all 20 runs and 6 problems have 19 runs because this calibration setup does not allow local fallback, so runs that remain too unstable without fallback are removed from the fit. This gives 654 problem-runs.

The raw scale is larger than the problem count suggests. A single problem-run executes all human references on the optimization tests, so the campaign yields 26,675,153 raw pairs $(d_{\text{stored}}, d_{\text{fresh}})$. Before fitting, we pool repeated measurements by the unique triple $(\text{problem}, \text{test}, \text{solution})$: the stored duration is fixed by the dataset, while the fresh CES duration is averaged across the 19–20 runs. This gives 1,369,381 pooled observations. The fit uses only optimization tests, because these are the tests whose duration enters the ranking reward; correctness tests can be useful for pass/fail recovery, but they are not the target distribution for timing calibration.

Short-run CES variability. Repeated CES executions show that short-run noise is present, but the noise scale is far below the local-sandbox instability in Appendix C.2. At the raw problem-test-solution level, the coefficient of variation across repeated CES measurements has mean 9.1%, median 8.2%, and a p5–p95 range of 3.5%–18.0%. In absolute terms, the standard deviation of a repeated duration measurement has mean 12.4 ms and median 9.9 ms, with p5–p95 range 6.2–26.7 ms. These numbers are not zero: they are large enough to matter for the fastest tests, where a few tens of milliseconds can move a solution across a tight percentile boundary. They are also much smaller than the cross-solution timing signal the reward tries to exploit.

The same conclusion holds after converting durations into the leaderboard statistic used by the ranked rewards. For each optimization test, the candidate duration is ranked against human-reference durations for that same test, with 0% denoting fastest and 100% slowest. The mean percentile is the average of these per-test percentiles across optimization tests. Across the 20 repeated CES runs, the standard deviation of mean percentile averages 2.1 percentage points, with median 2.0 and range 1.1–4.5 percentage points. A two-point percentile noise scale is still visible

Table 13: **Short-run CES variability in the first calibration campaign.** The top rows summarize repeated duration measurements for each pooled problem-test-solution triple. The bottom row summarizes the induced variability after those durations are converted into the mean-percentile leaderboard statistic used by ranked rewards.

Statistic	Mean	Median	Range / p5–p95
Duration CV across repeated CES runs	9.1%	8.2%	3.5–18.0%
Duration std. across repeated CES runs	12.4 ms	9.9 ms	6.2–26.7 ms
Mean duration of pooled observations	0.160 s	0.108 s	0.080–6.38 s
Mean-percentile std. across runs	2.1 pp	2.0 pp	1.1–4.5 pp

in strict metrics, but it is far from the 10.961-point local standard deviation and 41.173-point local range reported above.

Variance decomposition. The raw-duration decomposition is favorable. The between-triple variance is 0.02906, while the within-triple variance from repeated CES measurement is 0.00027, against total variance 0.02994. Equivalently, about 97.1% of the variation is explained by real differences between problem-test-solution triples, while the repeated-measurement component is about 0.9% of the total scale. This is the reason averaging fresh measurements across runs gives a stable fit: the calibration is fitting the systematic relationship between stored and fresh durations, not mostly CES jitter. For duration-ranked rewards, the remaining 2.1-point mean-percentile noise corresponds to a reward perturbation on the order of 0.01–0.02 per episode, small relative to the correctness swing from -1 to a positive reward and partly averaged out across the roughly 120 optimization tests per problem in the final training split.

Long-term CES drift. The larger issue is comparing current CES measurements against previously stored durations, either from human references or from model generations evaluated under an earlier CES state. An early 20-problem pilot showed fresh-to-stored duration ratios ranging from 0.96 to 1.34, with mean 1.15, too heterogeneous for one global multiplier. The full 33-problem campaign makes the shape of the drift clearer. Taking a fixed set of previously stored human-reference durations, fresh CES measurements from the new calibration campaign are often below the stored values on longer tests, so a slope below one is needed. For the fastest tests, however, a fixed overhead appears: a stored duration near zero does not map to a fresh duration near zero. This is exactly the regime used by strict timing percentiles, so ignoring the intercept can distort the ranking even if a multiplicative fit looks plausible on medium and long tests. In the larger calibration study, uncorrected stored rankings have only 0.54 Spearman correlation with fresh CES rankings, and 29 of 33 problems are biased in the same direction by an average of 19.6 percentage points. Even if live CES calls were noiseless, frozen stored durations would distort reference rankings unless recalibrated.

Affine correction. We correct long-term drift by fitting an affine map from stored to current CES durations:

$$d_{\text{corrected}} = \alpha \cdot d_{\text{stored}} + \beta, \qquad \alpha = 0.6306, \quad \beta = 0.0529\,\text{s}, \quad R^2_{\text{CV}} = 0.993. \tag{8}$$

The fit is performed with leave-one-problem-out cross-validation: each fold holds out one complete problem, fits on the other 32, and predicts fresh durations on the held-out problem. This is stricter than randomly holding out individual duration pairs, because all tests and solutions from the held-out problem are unseen during fitting. Table 14 reports the four families considered. The intercept is needed. A purely multiplicative correction reaches only $R^2_{\text{CV}} = 0.900$, while the affine model captures a fixed overhead of about 53 ms on top of proportional scaling. The polynomial has a slightly higher R^2_{CV}, 0.9936 versus 0.9933, and a slightly lower RMSE, 13.7 ms versus 13.9 ms, but this is not a meaningful gain for the reward path and adds an extra parameter. Applying the affine correction raises the Spearman correlation between stored and fresh CES rankings from 0.54 to 0.96. At data-load time, invalid stored durations are replaced by a conservative 10 s fallback before correction, and corrected durations are clamped to the valid $[0, 10]$ second execution range before use.

Table 14: **Stored-to-fresh duration correction families fit on the first CES calibration campaign.** All models are fit on 1,369,381 pooled problem-test-solution observations. Cross-validation leaves out an entire problem at a time. The affine model is the chosen calibration because it captures the fixed overhead with essentially the same held-out error as the polynomial, while the multiplicative and power-law families leave structured residuals.

Model	**Form**	**Train R^2**	**CV R^2**	**CV RMSE**
Multiplicative	$d_{\text{fresh}} = 0.7190\, d_{\text{stored}}$	0.9254	0.8998	54.0 ms
Power law	$d_{\text{fresh}} = 0.5551\, d_{\text{stored}}^{0.6661}$	0.7896	0.7798	80.0 ms
Affine (selected)	$d_{\text{fresh}} = 0.6306\, d_{\text{stored}} + 0.0529$	0.9945	0.9933	13.9 ms
Polynomial	$d_{\text{fresh}} = 0.6491\, d_{\text{stored}} - 0.0040\, d_{\text{stored}}^2 + 0.0502$	0.9948	0.9936	13.7 ms

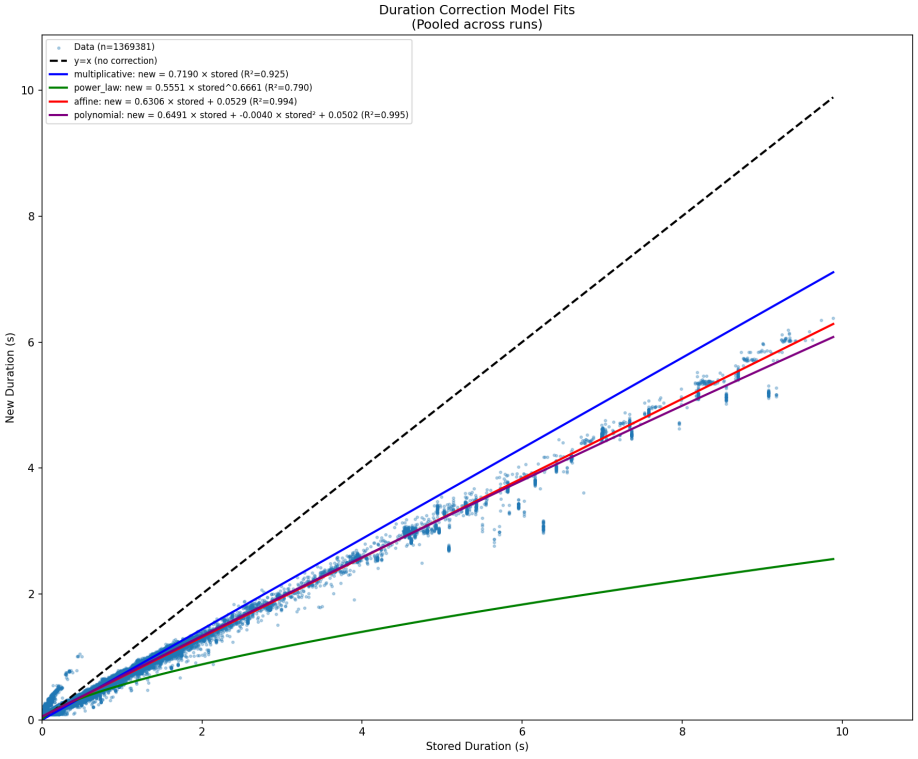


Figure 17: **Stored-versus-fresh CES durations and the correction families considered for calibration.** Each point is one pooled problem-test-solution observation from the first 33-problem calibration campaign, after averaging fresh CES measurements across repeated runs. The dashed line is the identity relation. Fresh CES durations are not obtained by applying a single global speed multiplier to stored durations: long tests need a slope below one, while fast tests show a visible additive floor. The affine and polynomial fits nearly overlap across the observed range, which is why the calibration uses the affine map in Eq. (8).

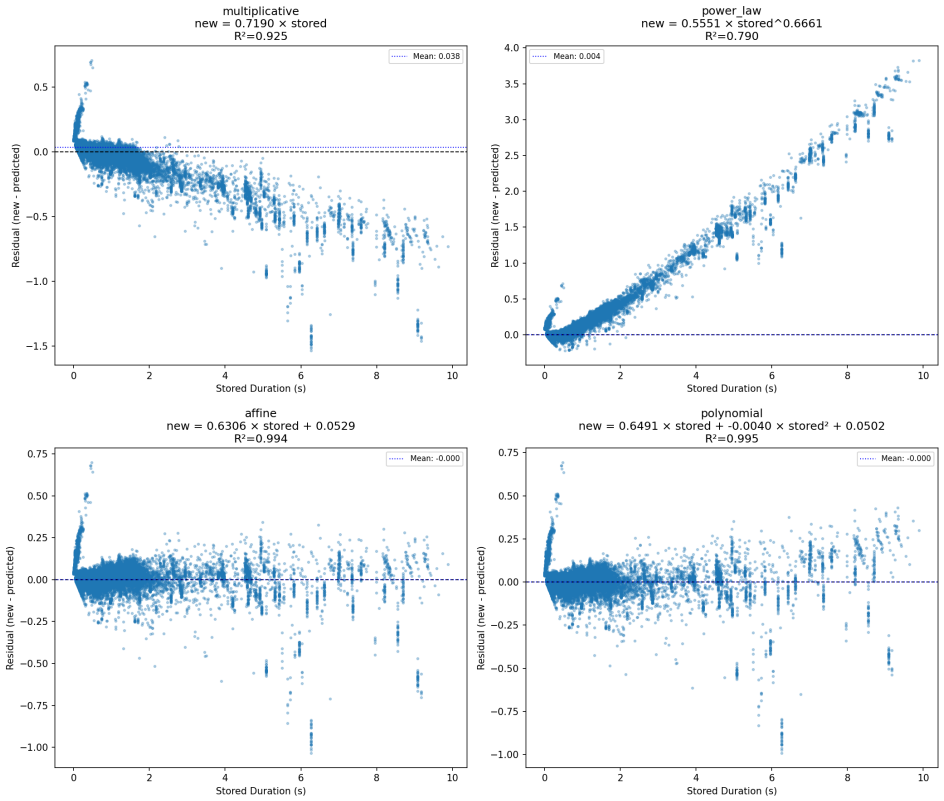


Figure 18: **Residual structure for the candidate duration-correction families.** The residual is $d_{\text{fresh}} - \hat{d}_{\text{fresh}}$, plotted against stored duration. The multiplicative model leaves a duration-dependent bias because it cannot represent the fixed overhead on fast tests and the proportional speed change on longer tests at the same time. The power-law fit bends in the wrong way in the original duration scale. Affine and polynomial correction leave residuals centered near zero across most of the observed range; the polynomial reduces error only marginally relative to affine, so the simpler affine correction is used for training and evaluation.

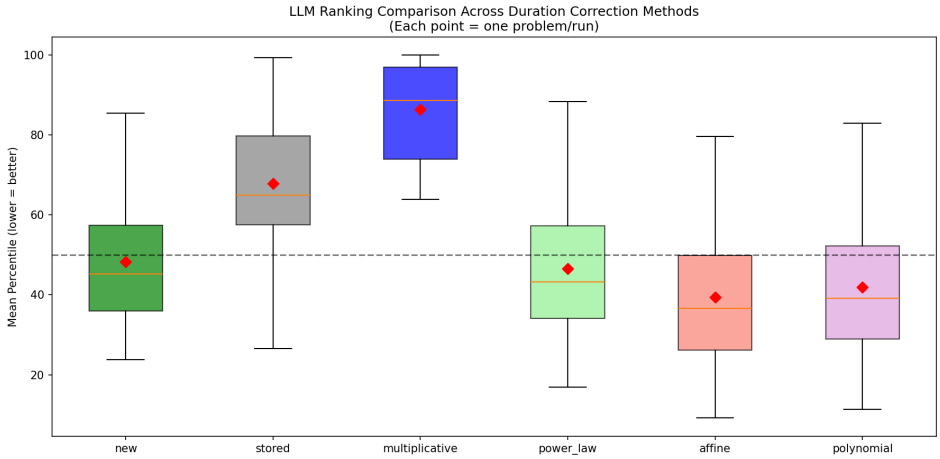


Figure 19: **Impact of duration correction on solution rankings.** Each box summarizes 654 problem-runs: 33 selected problems, each re-executed about 20 times. For each selected problem, the diagnostic uses one model-generated candidate solution from a saved rollout and compares it to the human-reference solutions for that same problem. On each optimization test, the candidate's fresh CES duration is ranked against the human-reference durations for that test; the plotted value is the mean of these per-test percentiles across the optimization suite. Lower is better because the candidate is faster than more human references. The boxes differ only in how the human-reference durations are represented: the green box uses freshly measured CES human-reference durations and is treated as the target distribution, while the other boxes use uncorrected stored durations or stored durations transformed by one correction family. Uncorrected stored durations shift the mean percentile from 48.3 to 67.9, a 19.6-point bias. The multiplicative correction is worse, with mean 86.4, because it removes proportional scale without correcting the fast-test offset. The affine correction moves the mean to 39.4 and the polynomial to 42.0; both are much closer to the fresh-CES reference than uncorrected stored durations, and the affine family is kept because its residual and cross-validation behavior are nearly identical to the polynomial with one fewer parameter.

C.8 Temporal stability of calibration

The preceding subsection used a first calibration campaign to map fixed stored dataset durations into a newer CES timing scale. That is the high-shift case, especially relevant after execution-service infrastructure changes such as moving to a new CPU fleet. There is also a more routine question: when the service infrastructure is nominally fixed, how much does the calibration move across day-to-day reruns? During the course of our experiments, we repeated the calibration several times on the same 33 problems with the same repeated-execution design. In this subsection, we report one such rerun on top of the results already reported above. Each campaign produces approximately 27M raw duration pairs, pooled to about 1.37M unique observations. Both campaigns fit the same correction families using leave-one-problem-out cross-validation.

Table 15 compares the affine parameters. The slope changes by only 1.0% ($0.6306 \rightarrow 0.6243$), while the intercept shifts from 53 ms to 38 ms. For a 1 s stored duration, this changes the predicted execution time by roughly 3%; for a 5 s duration, by roughly 1.5%. Cross-validated R^2 remains above 0.98 in both campaigns.

Table 15: **Affine calibration parameters and fit quality across two independent campaigns.** The slope α is stable to within 1%. The intercept β shifts by 15 ms, which has small impact on predicted durations for typical stored values. Both campaigns achieve cross-validated $R^2 > 0.98$, confirming that the affine model captures the stored-to-measured relationship reliably.

Parameter	**Campaign 1**	**Campaign 2**	Δ	Δ **(%)**
α (slope)	0.6306	0.6243	-0.006	-1.0%
β (intercept, s)	0.053	0.038	-0.015	-28%
Train R^2	0.9945	0.9858	-0.009	—
CV R^2	0.9933	0.9820	-0.011	—
CV RMSE (s)	0.014	0.023	+0.009	—

The model-performance profile across problems is also stable between campaigns. For each of the 33 problems, we compute the model candidate's fresh-CES mean percentile against the human references for that problem: lower means that the candidate is faster relative to more human solutions on that problem. We then ask whether the same problems remain easy or hard for the model across the two calibration campaigns. The Spearman correlation is $\rho = 0.99$, meaning that problems where the model candidate looks relatively strong in the first campaign are mostly the same problems where it looks relatively strong in the rerun campaign, and likewise for weak problems. This supports using the same affine family for day-to-day recalibration, while still allowing the numerical (α, β) values to move with the service state.

C.9 Evaluation-time sensitivity to affine calibration

Timing calibration keeps freshly recorded model-generated solution durations comparable to the fixed stored human-solution durations used by the ranking metric. We now ask whether the calibration itself affects the evaluation results: do different plausible calibrations lead to different optimization scores, or do they mainly move the absolute score scale while preserving comparisons between models? We therefore sweep a current affine fit, holding model executions fixed and changing only the corrected human-reference durations used by the p_{50} and p_{10} ranking metrics. This is an evaluation-time sensitivity check: decreasing α or β makes corrected references shorter, so the same model executions are judged against a stricter speed threshold. Figure 20 shows the result for CWM 32B. In relative terms, the sweep moves β by about 31% and α by about 19% from their starting values; the β movement also matches more or less the drift pattern in Appendix C.8, where β moved by 28% across calibration campaigns while α moved by 1%, and for α we choose to do a wider cautious sweep. Perturbing the additive offset β remains at least as important as perturbing the multiplicative slope α: reducing β from 0.0498 to about 0.0344 s drops the baseline p_{10} pass@1 from 38.8% to 1.2%, while reducing α from 0.7019 to 0.5692 with fixed β drops the baseline from 31.1% to 1.4%. The scale explains why this effect appears on strict percentiles. The full β sweep changes every corrected reference by about 15.3 ms, whereas the α sweep changes a stored duration d by $0.1326d$: 13.3 ms at $d = 100$ms, 132.6 ms at $d = 1$s, and 663.2 ms at $d = 5$s. Thus the intercept is the larger perturbation in the fast-reference regime, while the slope is larger on the multi-second tail. Table 6 shows that optimization tests create a much heavier tail than the original DMC tests,

but most duration records are still sub-second: the pooled median is 0.099 s, $p90$ is 0.736 s, and only 6.87% exceed 1 s. Moreover, a p_{10} decision compares the model execution against the fastest human-reference durations for each test, not against the median human duration. The relevant cutoff is therefore often in the part of the distribution where a 15 ms additive shift is comparable to, or larger than, the effect of the tested slope perturbation.

The bottom panels give the more important training conclusion. Making the calibration stricter lowers all absolute p_{10} scores, but it lowers the correctness-only baseline faster than the optimization-trained models. At the lenient beta endpoint, the 2 s filter, the two-gate reward, and the top-30% reward score 56.3%, 55.8%, and 57.8%, compared with 38.8% for the baseline; their relative gains are 45%, 44%, and 49%. At a more stringent beta point, $\beta \simeq 0.042$, the baseline falls to 9.2%, while the same three optimization-trained models remain at 22.2%, 21.8%, and 22.2%; the relative gains grow to 141%, 137%, and 141%. At the most stringent beta point, the baseline falls to 1.2%, while the same three optimization-trained models remain at 4.0%, 3.9%, and 4.2%; the relative gains grow to 233%, 225%, and 250%. This endpoint is less interesting because the baseline is already so low that any nonzero optimization-trained score looks much larger in relative terms. The alpha sweep shows the same pattern: at $\alpha = 0.6321$, the baseline is 6.5%, while the three optimization-trained models reach 17.5%, 17.0%, and 17.8%, for gains of 169%, 162%, and 174%. The control with more-correctness and optimization tests also stays above the baseline, but with smaller gains: 14% at the lenient beta endpoint and 58% at the stringent beta point. This means that stricter calibration does not erase the optimization-RL effect; on the contrary, it makes the gap more visible, because the baseline loses a larger share of its top-decile solutions. The optimization-RL setups used here therefore appear particularly beneficial under more demanding evaluation settings.

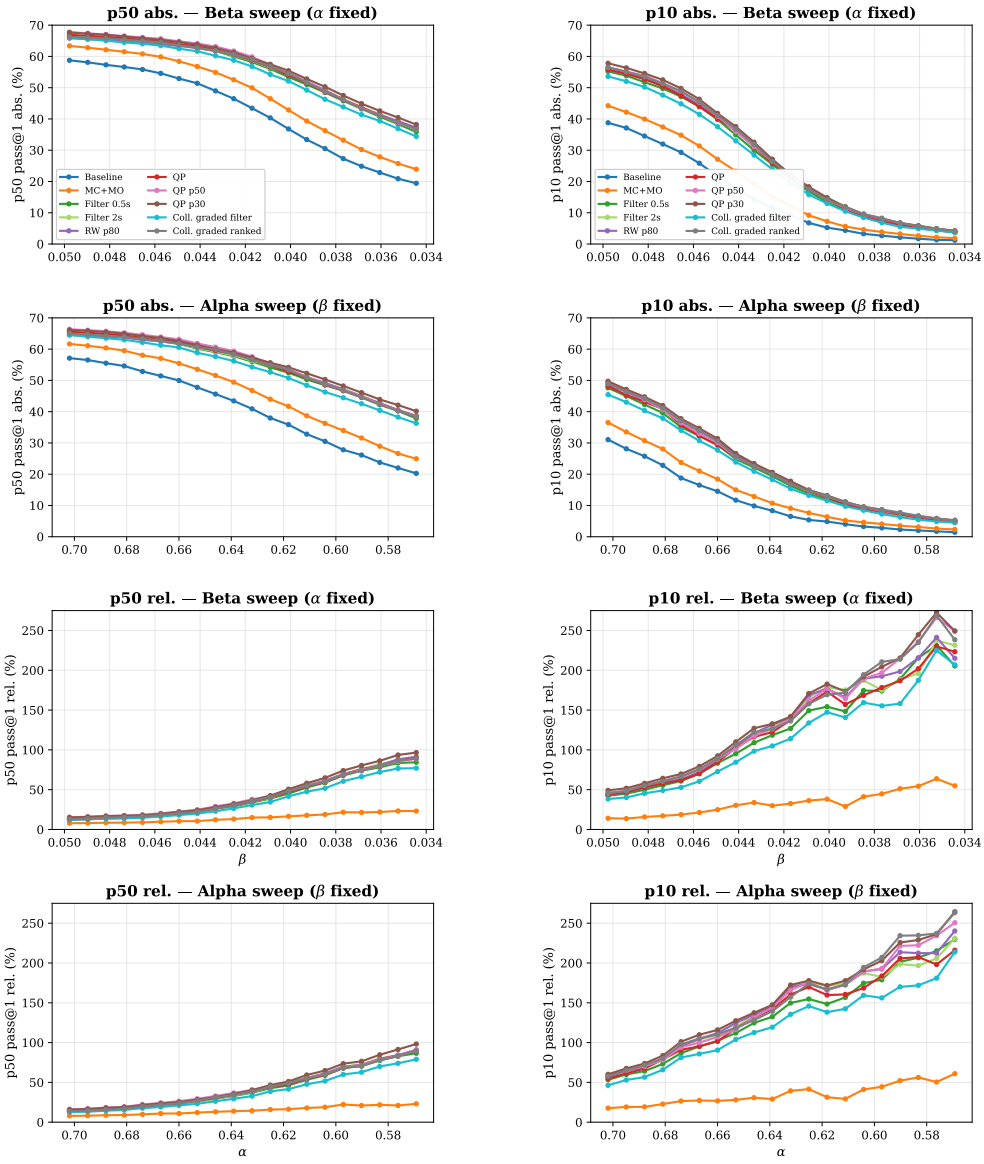


Figure 20: **Sensitivity of p_{50} and strict p_{10} pass@1 to affine calibration parameters.** *Left:* Absolute CWM 32B DMC-Optim pass@1 and relative improvement at p_{50} as β is swept with α fixed or α is swept with β fixed. *Right:* The same evaluation-time sensitivity check at p_{10}. The additive offset matters because it is a large fraction of the corrected duration for short tests. The relative ordering of optimization-trained configurations remains stable across the displayed range. As β becomes more stringent, the relative gains of the 2 s filter, the two-gate reward, and the top-30% reward grow from 45%, 44%, and 49% at the lenient endpoint to 141%, 137%, and 141% at the intermediate β value, and up to 233%, 225%, and 250% at the most stringent endpoint, though there the baseline is so low that any nonzero optimization-trained score looks much larger in relative terms.

C.10 Joint training-time and evaluation-time calibration sweeps

Appendix C.9 held the trained models fixed and changed only the affine correction used for post-training scoring. The next question is whether calibration also matters while learning. This is reward-dependent: rewards based only on pass/fail outcomes or raw timeouts do not need a calibrated human-reference distribution, but the post-execution ranked rewards do, because they compare each rollout to a fixed distribution of stored human-reference durations. If such rewards were very sensitive to the exact (α, β) used while training, then a small calibration mismatch could change the learned policy even if final scoring rankings were stable. We therefore run a two-dimensional

sensitivity study on Qwen 2.5 7B: one axis changes the affine correction used during RL training, and the other axis changes the affine correction used during post-training resimulation.

Training-time calibration sweep. We train 16 Qwen 2.5 7B models across two post-execution ranking environments, both defined in Appendix D.2. The first is the top-30% leaderboard-percentile reward (top-30% for short): after execution, the candidate is inserted into the calibrated human-reference leaderboard and receives the binary optimization gate only if it falls inside the top 30% of the reference-quality distribution. The second is the leaderboard-percentile two-gate bucketed reward (two-gate for short): the same leaderboard construction is used, but each correct candidate receives a graded reward according to its leaderboard percentile over the full reference pool. We train eight models with each environment. Thus the two families use the same underlying calibrated duration distribution but expose it to RL differently: top-30% turns the ranking into a sharper pass/fail optimization gate, while two-gate gives a smoother graded signal to all correct candidates.

Within each family, the only intended training change is the affine correction applied to stored reference durations before duration filters and reference rankings are computed. We index the eight reward-calibration points as ab1–ab8. They linearly sweep from $\alpha = 0.594939, \beta = 0.054482$ to $\alpha = 0.719699, \beta = 0.048993$, a range chosen from the calibration variation observed across repeated CES recalibration campaigns. The ab3 point, $\alpha = 0.630585, \beta = 0.052914$, matches the original affine fit from Appendix C.7 up to rounding. The ablation runs are launched as one concurrent sweep under the same CES setup, so the live execution backend is held fixed while the stored-reference correction used by the reward changes. Two replication references trained with the ab3 coefficients under older CES states are included to check how much changes when the reward calibration is fixed but the service state differs.

Evaluation-time calibration sweep. All swept training models and the two replication references are re-executed concurrently on CES to obtain fresh candidate timings under the same service state. We then resimulate each model under 10 scoring calibrations, denoted s0–s9, linearly interpolating from $(\alpha = 0.701876, \beta = 0.049777)$ to $(\alpha = 0.606033, \beta = 0.038306)$. These endpoints are two plausible affine corrections from repeated recalibration runs and are used here to test whether the choice of correction matters once the model outputs are fixed. Lowering α and β makes the corrected human references shorter, so the same model execution is judged against a stricter speed target. We evaluate p_{100}, p_{20}, and p_{10} pass@1, plus p_{10} pass@10. Table 16 records the two calibration grids.

Table 16: **Reward and post-training scoring calibration grids for the two-dimensional alpha/beta sensitivity study.** Reward-calibration points ab1–ab8 change the affine correction used inside RL. Scoring points s0–s9 change the affine correction used after re-executing the trained models together on CES. The range is chosen to cover the variation observed across different calibration campaigns during this work.

Reward	α	β	Score	α	β
ab1	0.594939	0.054482	s0	0.701876	0.049777
ab2	0.612762	0.053698	s1	0.691227	0.048502
ab3	0.630585	0.052914	s2	0.680578	0.047228
ab4	0.648407	0.052130	s3	0.669928	0.045953
ab5	0.666230	0.051345	s4	0.659279	0.044679
ab6	0.684053	0.050561	s5	0.648630	0.043404
ab7	0.701876	0.049777	s6	0.637981	0.042130
ab8	0.719699	0.048993	s7	0.627331	0.040855
—	—	—	s8	0.616682	0.039581
—	—	—	s9	0.606033	0.038306

Results. Figures 21 to 25 show complementary projections of the same 16-by-10 scoring matrix. Figure 21 averages over reward calibrations and shows that the post-training scoring calibration is the dominant axis. Figure 22 fixes the reward-calibration axis on the x-axis and shows that, at a fixed scoring point, changing reward calibration produces nearly flat curves. Figure 23 overlays percentiles and shows why strict metrics are more sensitive. Figure 24 focuses on strict p_{10}, where most variation follows the scoring axis rather than the reward-calibration axis. Figure 25 shows the same scale separation as a dot strip: reward calibrations cluster tightly within each scoring calibration, while row centers move substantially across scoring calibrations. Table 17 gives the full strict p_{10} pass@1 matrix, and Table 18 reports p_{100}, p_{20}, and p_{10} pass@10 anchors at s0, s5, and s9.

Table 17: **Raw strict p_{10} pass@1 scores for the reward and scoring alpha-beta sweep.** Rows are trained Qwen 2.5 7B models. Columns are scoring calibrations s0–s9, with s0 and s9 denoting the two endpoints in Table 16. All values are percentages at step 10k after shared CES re-execution.

Family	Reward cal.	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9
Top 30%	ref	41.1	40.6	40.0	39.0	37.7	36.3	32.5	24.8	13.3	5.8
Top 30%	ab1	41.0	40.4	39.9	39.0	37.8	36.2	32.6	24.1	12.6	5.3
Top 30%	ab2	41.9	41.3	40.8	39.9	38.5	36.8	33.3	24.9	12.7	5.7
Top 30%	ab3	41.1	40.9	40.2	39.0	37.9	36.3	32.6	24.4	12.8	5.4
Top 30%	ab4	40.5	40.2	39.7	38.7	37.3	35.3	31.6	23.8	12.0	5.3
Top 30%	ab5	39.3	38.9	38.3	37.5	36.4	34.8	31.1	23.4	12.0	5.0
Top 30%	ab6	40.4	40.0	39.4	38.5	37.2	35.3	31.9	23.9	12.6	5.2
Top 30%	ab7	40.7	40.3	39.5	38.6	37.4	35.5	31.4	22.6	11.5	5.1
Top 30%	ab8	40.8	40.3	39.8	38.7	37.4	35.2	31.0	22.5	12.1	5.2
Two-gate	ref	40.1	39.6	38.8	37.8	36.2	34.2	29.7	21.2	10.5	4.6
Two-gate	ab1	40.3	40.0	39.3	38.2	36.9	35.1	31.5	23.7	12.8	5.2
Two-gate	ab2	39.3	38.8	38.2	37.2	35.8	33.8	29.6	21.4	10.9	4.9
Two-gate	ab3	39.7	39.4	38.6	37.6	36.1	34.2	30.2	21.3	10.8	4.6
Two-gate	ab4	39.4	38.9	38.3	37.3	35.5	33.4	29.6	21.1	10.9	4.6
Two-gate	ab5	39.3	38.7	38.0	36.8	35.3	33.0	28.4	19.5	9.7	3.8
Two-gate	ab6	39.8	39.0	38.0	36.6	34.9	32.2	27.7	18.7	9.4	4.2
Two-gate	ab7	39.1	38.4	37.6	36.5	34.9	32.4	27.8	19.2	9.6	4.1
Two-gate	ab8	39.1	38.8	38.0	36.8	35.0	32.7	28.1	19.1	9.0	4.4

Table 18: **Raw anchor scores for correctness, p_{20}, and pass@10 under the reward and scoring alpha-beta sweep.** The table reports the scoring endpoints s0 and s9 plus the midpoint s5. p_{100} is invariant to scoring calibration and is shown once per trained model. All values are percentages at step 10k after shared CES re-execution.

			p_{20} pass@1			p_{10} pass@1			p_{10} pass@10		
Family	Reward cal.	p_{100}	s0	s5	s9	s0	s5	s9	s0	s5	s9
Top 30%	ref	46.2	42.1	39.1	12.0	41.1	36.3	5.8	61.6	57.1	17.8
Top 30%	ab1	46.0	41.9	38.9	11.5	41.0	36.2	5.3	61.4	57.1	17.0
Top 30%	ab2	46.9	42.9	39.8	11.6	41.9	36.8	5.7	60.8	56.7	16.8
Top 30%	ab3	46.3	41.9	39.2	11.8	41.1	36.3	5.4	60.8	56.9	16.8
Top 30%	ab4	45.9	41.6	38.5	11.2	40.5	35.3	5.3	60.2	56.0	15.7
Top 30%	ab5	45.0	40.1	37.5	11.1	39.3	34.8	5.0	60.5	56.7	14.7
Top 30%	ab6	45.9	41.5	38.4	11.6	40.4	35.3	5.2	62.3	58.1	16.6
Top 30%	ab7	46.6	42.1	38.7	10.6	40.7	35.5	5.1	61.8	57.3	15.8
Top 30%	ab8	46.7	42.1	38.8	11.0	40.8	35.2	5.2	62.0	56.5	15.6
Two-gate	ref	47.4	41.7	37.6	9.9	40.1	34.2	4.6	59.4	54.2	13.7
Two-gate	ab1	47.1	41.5	38.3	11.5	40.3	35.1	5.2	62.9	58.4	15.9
Two-gate	ab2	45.9	40.6	36.9	10.0	39.3	33.8	4.9	59.7	55.8	15.1
Two-gate	ab3	46.4	41.0	37.4	9.8	39.7	34.2	4.6	59.8	55.1	13.7
Two-gate	ab4	46.3	40.8	36.9	10.3	39.4	33.4	4.6	61.3	56.5	14.6
Two-gate	ab5	47.0	40.8	36.3	8.9	39.3	33.0	3.8	60.1	54.8	12.7
Two-gate	ab6	48.4	41.4	36.1	8.2	39.8	32.2	4.2	59.4	53.8	12.9
Two-gate	ab7	46.8	40.5	36.3	8.8	39.1	32.4	4.1	59.3	54.0	12.4
Two-gate	ab8	46.9	40.7	36.3	8.4	39.1	32.7	4.4	61.2	56.0	14.2

Reward calibration has small effect. At any fixed scoring calibration, the spread across the eight reward calibrations is small compared with the absolute score movement across scoring calibrations. For p_{100}, which does not depend on scoring alpha/beta, the spread is 1.9 percentage points for top-30% and 2.5 points for two-gate. This is the purest training-time readout: changing the calibration used inside the reward does not materially change correctness. The strict p_{10} readout is similar. At s0, top-30% spans 39.3–41.9% across ab1–ab8, a 2.6 point range; two-gate spans 39.1–40.3%, a 1.2 point range. At s5, the spreads are 2.0 and 2.9 points. These ranges are comparable to the 1–2.5 point run-to-run variation observed in the rerun-robustness analysis, so we should not interpret small differences between adjacent reward-calibration points as meaningful model-quality changes.

Figure 24 and Fig. 25 test whether reward and scoring calibrations need to be matched. If they did, Fig. 24 would show a diagonal ridge: models trained under lower reward-calibration points would do best under lower scoring calibrations, and models trained under higher reward-calibration points would do best under higher scoring calibrations. We do not see that pattern. The contours are nearly vertical, and Fig. 25 shows tight within-row clusters. The practical conclusion is that the tested calibration range changes the reward scale but does not select a different kind of solution strongly enough to dominate training noise. This is useful operationally because optimization-RL runs can span two days or more: within the range tested here, the training run does not need continuous reward-side recalibration as long as the service-state variation stays comparable to the range we study. A likely reason is that GRPO computes advantages within the on-policy batch, so common shifts in absolute reward values are partly absorbed before the parameter update. The post-training scoring side is different. The ranking of runs remains stable, but absolute strict-percentile scores can move substantially with the scoring calibration, and stricter scoring conditions emphasize optimization-trained models that preserve more top-decile solutions.

Scoring calibration dominates absolute strict metrics. The scoring axis has much larger effect. For the top-30% replication reference, p_{10} pass@1 falls from 41.1% at s0 to 5.8% at s9, a 35.3 point drop. For two-gate, it falls from 40.1% to 4.6%, a 35.5 point drop. This is more than an order of magnitude larger than the reward-calibration spread at a fixed scoring point. The same pattern holds at p_{20}, though the drop is less severe: top-30% moves from 42.1% to 12.0% and two-gate from 41.7% to 9.9%. Scoring calibration therefore controls the absolute strict-percentile scale. This does not mean the models change; the same shared re-executed model outputs are being rescored against shorter corrected references.

The percentile dependence is useful. p_{100} is invariant because it measures correctness under the timing-sensitive reexecution, not membership in a speed percentile. p_{20} retains 28.5% of the s0 top-30% score at s9 and 23.7% of the two-gate score. Strict p_{10} retains only 14.1% and 11.5%, respectively. p_{10} pass@10 is less brittle than pass@1 in relative terms, retaining 28.9% for top-30% and 23.1% for two-gate, but its absolute drop is even larger because the starting pass@10 scores are higher. This is the expected behavior of a strict percentile metric: as references are shortened, the boundary for being in the top decile moves faster than the boundary for being correct or merely within the top fifth.

The two reward families respond similarly, but top-30% is consistently stronger on strict optimization. The top-30% and two-gate families are not two calibration methods; they are two ways of turning the same calibrated human-reference leaderboard into reward. Two-gate supplies a graded reward across the full leaderboard, while top-30% makes the top-30% event a binary optimization gate. Across this sweep, top-30% is consistently ahead on strict p_{10} pass@1. Averaging over ab1–ab8, the top-30% advantage over two-gate is 1.2 points at s0, 2.3 points at s5, and 0.8 points at s9. The gap is modest compared with the scoring-calibration drop, but it persists across the calibration axis. This supports the reward-design result from Appendix D.2: for this Qwen 7B setting, the sharper top-30% gate produces a stronger strict-optimization signal than the smoother two-gate reward, even though both use the same calibrated duration reference.

The two-gate family sometimes has slightly higher p_{100}, for example 48.4% for two-gate ab6 versus a top-30% maximum of 46.9%. That does not contradict the strict optimization result. It suggests that the smoother two-gate reward can preserve or improve broad correctness while being less selective for top-decile speed. Conversely, top-30% is more aligned with the strict p_{10} target, but does not dominate every correctness-side number.

Cross-service-state replication. For the two reward families, we also include reference runs trained with the same affine coefficients as ab3 but collected under a different CES service state. After shared re-execution, these references can be compared with ab3 to ask whether changing the service state during training, while keeping the reward calibration fixed, visibly changes the learned model. The answer is mostly no. For top-30%, the reference and ab3 rows are identical or within 0.5 points across the strict p_{10} scoring grid. For two-gate, they are also closely aligned. This is the same effect we tried to simulate when we keep CES executions fixed and vary the calibrated reference durations: it changes the relative position of model timings and human-reference timings. The replication check confirms that, within this range, changing CES conditions under fixed reference calibration behaves similarly to changing reference calibration under fixed CES executions.

The absolute scores in this subsection are diagnostic and should be compared only within this shared re-execution campaign. In the main result tables, we rerun a calibration campaign at evaluation time and use the resulting fit ($\alpha = 0.604$, $\beta = 0.038\,\mathrm{s}$) to approximate, as closely as possible, the score we would obtain in an ideal no-shift, no-noise timing system: running the same snippet on the same test would always return the same duration, so stored human-reference durations and freshly executed model durations would already live on the same timing scale. That correction moves stored human-reference durations toward the timing scale of the CES state used by the evaluation. For example, the top-30% replication reference scores $p_{10} = 41.1\%$ here under s0, while the corresponding main-result table reports 6.0%. This difference is not a model improvement; it comes from using a different CES state and calibration. The point of this subsection is therefore the within-campaign comparison across reward and scoring calibrations, not an update of the headline scores.

Takeaway. The two-dimensional sweep separates three facts that are easy to conflate. First, reward-time affine calibration over the tested range has weak effect on the learned Qwen 7B models. Second, post-training scoring calibration has large effect on absolute strict-percentile scores, especially p_{10}, because it changes the corrected reference durations against which fixed model executions are ranked, though it preserves the relative ordering of the runs in this sweep, which supports the robustness of the comparisons. Third, the relative comparison between the two reward families is stable enough to interpret: top-30% remains the stronger strict-optimization training signal, while two-gate can preserve a bit more correctness.

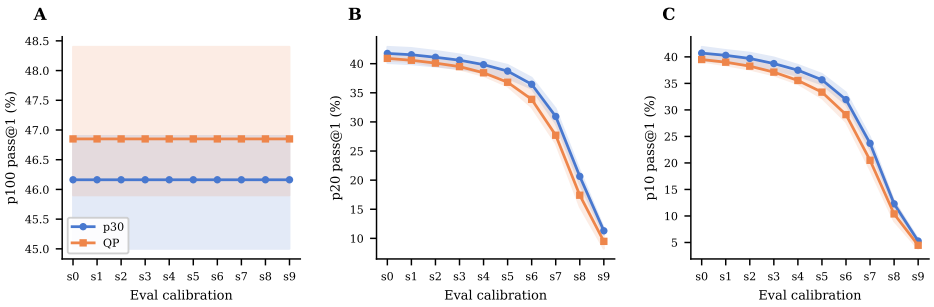


Figure 21: **Post-training scoring calibration drives the mean score, while reward calibration mostly appears as a narrow band.** Each panel plots score against scoring calibration s0–s9. Colors distinguish the top-30% and two-gate reward families; shaded bands are the min–max range over the eight reward calibrations ab1–ab8. The p_{100} panel is flat because correctness does not depend on reference-duration calibration. At p_{20} and especially p_{10}, lowering the scoring calibration from s0 to s9 makes corrected human references shorter and sharply reduces absolute scores, while the within-band spread from changing the reward calibration remains small.

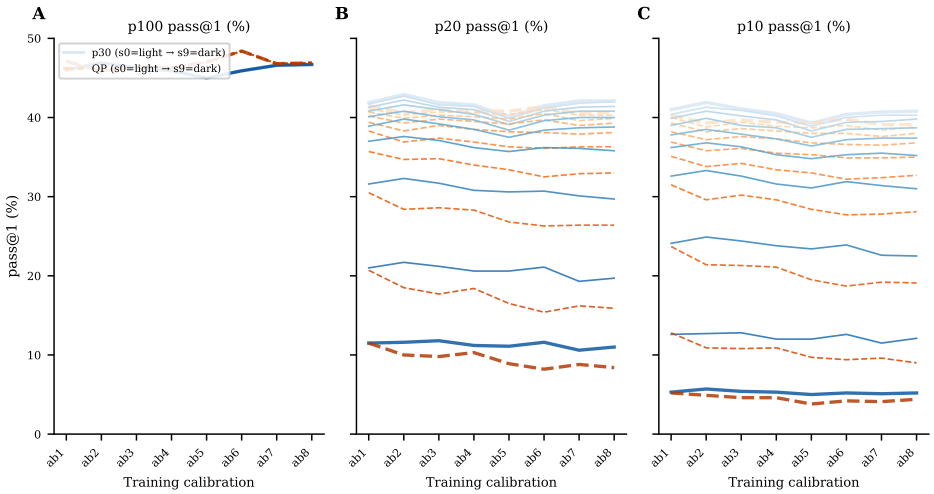


Figure 22: **Reward calibration has weak effect at fixed scoring calibration.** The x-axis is the reward calibration ab1–ab8. Each curve is one scoring calibration s0–s9; blue solid curves are top-30% and orange dashed curves are two-gate. Within a fixed scoring curve, the lines are nearly flat compared with the vertical separation between scoring calibrations. This is the main evidence that changing the reward-side (α, β) during training does not strongly change the learned model quality over the tested range.

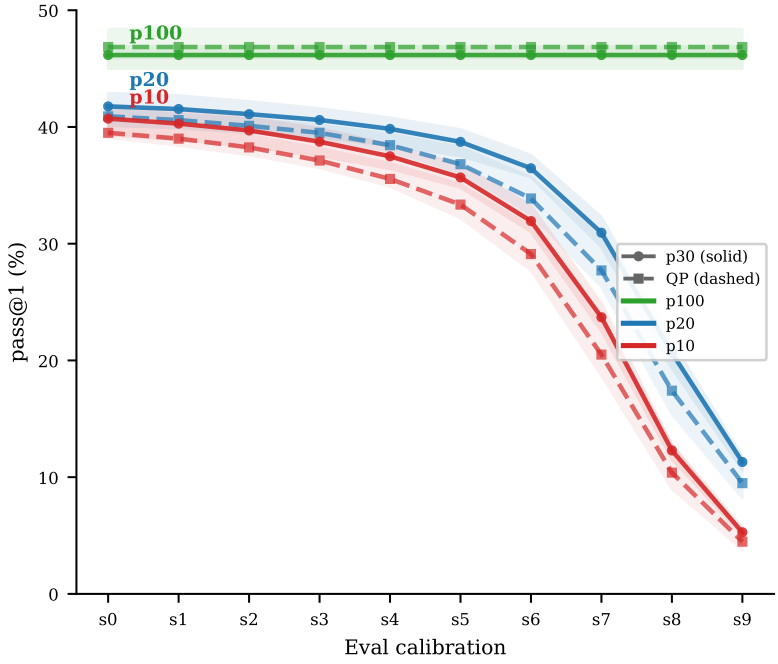


Figure 23: **Stricter optimization percentiles amplify scoring-calibration sensitivity.** All three pass@1 percentiles are plotted against scoring calibration in one view, with min–max bands over reward calibrations. p_{100} remains invariant because it only asks whether the solution is correct. p_{20} drops substantially as the scoring calibration moves from s0 to s9, and p_{10} drops further because top-decile membership depends on the shortest corrected reference durations. The figure separates correctness effects from timing-threshold effects: reward calibration barely changes the bands, while scoring calibration changes the strict optimization metrics.

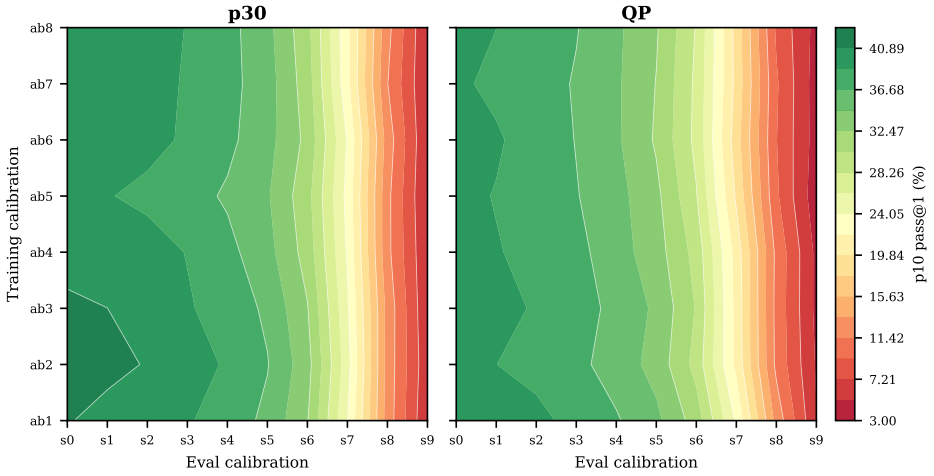


Figure 24: **Strict p_{10} score surface over reward and scoring calibration.** The color shows p_{10} pass@1 for each reward calibration ab1–ab8 and scoring calibration s0–s9. The contours are nearly vertical in both the top-30% and two-gate panels, meaning that the score changes mainly when the scoring calibration changes, not when the reward calibration changes. There is no visible diagonal ridge suggesting that training with a reward calibration close to the later scoring calibration is necessary.

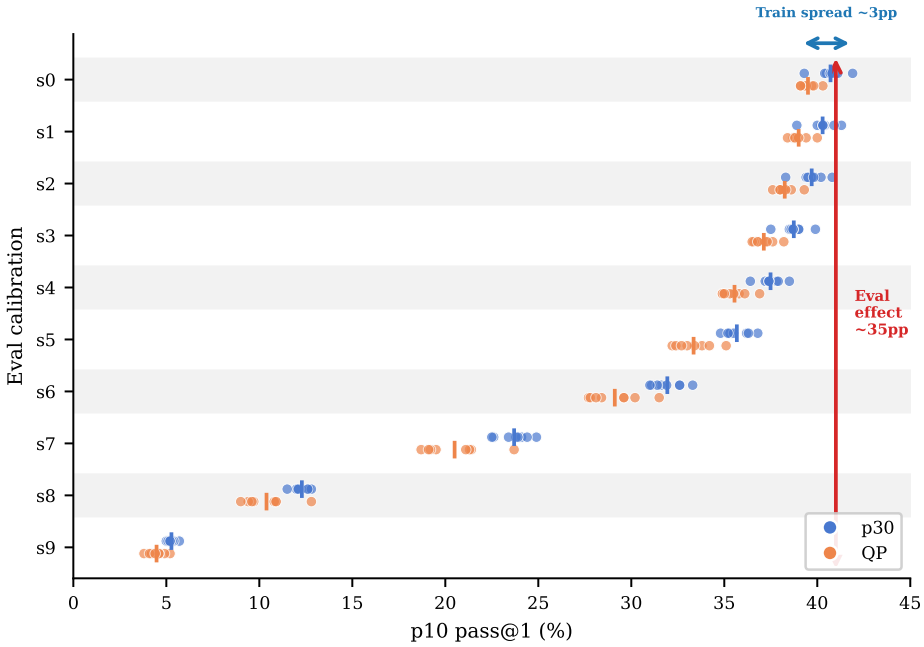


Figure 25: **Scale separation between reward-calibration spread and scoring-calibration movement.** Each row fixes a scoring calibration and shows the eight reward calibrations as dots. The dots cluster tightly within a row, while the row centers move by roughly 35 percentage points from s0 to s9 at strict p_{10}. This visualizes the main quantitative conclusion: reward-side alpha/beta choices move the final score by only a few points, while post-training scoring calibration determines the absolute strict-p_{10} scale.

C.11 Lessons learned for timing-based RL

From the analyses above, we retain that:

1. Timing-sensitive dataset construction, training, and evaluation should use a controlled, confined service to measure runtimes of generated code snippets, rather than local sandboxes running concurrently with the RL run: the issue is not only an extra degree of timing noise; it can completely corrupt the timing signal.
2. Local sandbox execution on the RL workers can recover inconclusive correctness verdicts, as long as timeouts are generous and not constraining, but its durations should not enter timing rewards or timing-sensitive metrics.
3. Aggregate service health can hide short harmful windows: rare infrastructure bursts can affect a large fraction of rollouts in a few optimizer steps, and silent service slowdowns can shift measured durations even when pass/fail counters look normal.
4. Model comparisons should use shared and concurrent re-execution in the same controlled setting rather than mixing timings collected under different service states.
5. A calibration campaign should be rerun for timing-sensitive evaluation, and the affine stored-duration correction should be refit when moving to a new execution-service fleet, after a sandbox upgrade, or after a sustained service-state change.
6. GRPO within-batch advantage computation can partially absorb common live-service shifts during training; post-training scoring is more sensitive to calibration in absolute strict scores, although the relative model ranking is stable and optimization-trained models stand out more under stricter scoring settings.

An exact reproduction of our sandbox code is not necessarily needed, because what matters most is: isolating code timing from training worker contention, using a dedicated CPU pool; controlling the resource envelope assigned to the sandbox, and keeping it fixed; exposing failures and their categories explicitly; controlling retries and concurrency; and most importantly supporting calibration when the service state changes.

C.12 Improvements and experiments we did not have the time to conduct

Controlled timing-noise injection. The studies above use noise that happened in real runs: repeated CES reruns, bursts, slowdowns, and calibration changes. A next study could add known noise after the sandbox returns a result. For example, we could add 5–50 ms to each duration, recompute whether each test should now count as a timeout, and update the stored duration. We could also multiply durations, add larger noise on short tests, or simulate bursts that last for several optimizer steps. Because the injected noise is known, we could compare reward environments directly and see which ones break first, and which ones stay more stable.

Replaying service states on fixed rollouts. The RL-run analysis is hard to isolate because the policy, sampled problems, sampled rollouts, and CES state all change at the same time. A cleaner study would keep the rollout logs fixed. Given saved rollouts with code, tests, statuses, and durations, we could recompute rewards under different service traces: healthy execution, real burst windows, synthetic slowdowns, different fallback policies, discard, or quarantine. This could help measure whether the RL environment can correctly smooth part of the execution noise after sandbox execution. It could also estimate how much reward variance comes from the service state. GRPO could then use this estimate to normalize advantages, downweight noisy batches, or trigger extra re-execution. Replaying fixed rollout batches under controlled service states could also measure how the policy update would drift under each service state.

Storing standard deviation estimates. Several environments use human-reference durations. The repeated-CES campaign could let us store not only one corrected duration, but also an estimate of its standard deviation. This could be stored for each problem-test-solution duration, or more coarsely for each test-level reference distribution. These estimates could then be used in several places. Filtering could avoid tests whose duration spread is mostly measurement noise. Timeout choices could include a safety margin for noisy tests. Ranking rewards could soften comparisons when the generated snippet and the reference boundary are too close to distinguish reliably. The

same estimate could also trigger selective re-execution, or give GRPO a reward-standard-deviation estimate when it computes or normalizes advantages.

Adaptive monitoring and recalibration. Live calibration probes could help detect silent service drift, but they also use execution capacity. Too few probes are noisy; too many slow down rollout execution. A sentinel suite could be run from time to time during training and used to detect changes in slope, offset, or tail behavior. When the shift is large enough, the system could refit the affine correction, quarantine recent batches, or pause rollout collection.

D Reward and environment details

Optimization tests by themselves do not define an RL reward. They produce execution records, and the reward design decides which parts of those records should affect learning. This appendix gives the detailed version of the argument compressed in Section 4: timing constraints can enter the environment before execution, during execution, or after execution; all settings are reduced to a common reward-facing interface that separates correctness, hard optimization gate, and duration quality into three numerical signals that the reward can then be built upon.

D.1 Problem taxonomy

For this paper, an RL environment $\mathcal{E}$ is the full routine around the model to prepare a rollout that the optimizer will then take a step upon. Given a programming problem x, the environment prepares the prompt shown to the model, receives a generated answer y, decides which tests to run and under which constraints, aggregates the resulting statuses and durations into outcomes, and returns a reward $R_{\mathcal{E}}(x, y)$. We define the taxonomy so that each RL environment exposes a common set of signals, from which different reward functions map the rollout to the reward range $[-1, 1]$.

Each test t is an input-output evaluation item with input u_t, expected output v_t, and a default execution cap τ_t; optimization environments may later replace this cap by an effective limit λ_t. Running a candidate program y on t returns a measured duration $\delta_t(y)$ and a status

$$s_t(y) \in \mathcal{S} = \{\text{success}, \text{timeout}, \text{failure}\}. \tag{9}$$

Here success means that the output matches v_t within the effective time limit, timeout means that the run did not finish within that limit and therefore produced no output to compare with v_t, and failure means that the run finishes within the effective time limit but its produced output does not match v_t. The executor records more detailed statuses, but the reward notation below uses only this three-way status. For each problem we also have a calibrated human reference pool $H(x)$: accepted reference solutions with stored duration traces on optimization tests. For $h \in H(x)$, $\tilde{d}_{t,h}$ denotes the calibrated duration of reference solution h on test t; this pool is used to construct reference-derived limits and percentile rankings, not as a model output.

We split the tests attached to x into correctness tests $T^{\mathrm{C}}(x)$ and a raw optimization-test pool $T^{\mathrm{O}}(x)$. The correctness tests are fixed for a given environment and produce the base correctness signal; executed optimization tests can then refine this signal by rejecting failures. The design choices studied in Section 4 act on the optimization-test pool, the timing constraints, or the duration record. At this level, the optimization-side routine can be written as three parameterized operators:

$$T^{\mathrm{O}}_{\mathrm{used}} = F_{\mathrm{pre}}(T^{\mathrm{O}}(x); \theta_{\mathrm{pre}}), \tag{10}$$

$$\mathcal{Z}^{\mathrm{O}}(x, y) = E_{\mathrm{intra}}(y, T^{\mathrm{O}}_{\mathrm{used}}; H(x), \theta_{\mathrm{intra}}), \tag{11}$$

$$\big(g_{\mathrm{post}}(x, y), q_{\mathrm{post}}(x, y)\big) = Q_{\mathrm{post}}(\mathcal{Z}^{\mathrm{O}}(x, y); H(x), \theta_{\mathrm{post}}), \tag{12}$$

where $\mathcal{Z}^{\mathrm{O}}(x, y)$ is the optimization-side execution record, including the executed tests, statuses, durations, and intended limits. These operators are meta-classes of environments. A pre-execution environment is defined by the routine used to instantiate F_{pre}, such as an absolute-duration filter, a relative-duration filter, or a character-length filter. An intra-execution environment is defined by the constraints in E_{intra}, such as absolute timeouts, relative timeouts, or ranked-worst time limits. A post-execution environment is defined by whether Q_{post} turns the observed durations into a ranked gate, a graded ranked scalar, or both. After these routines run, the reward-facing interface contains a correctness signal c_{cor}, its strict optimization-aware refinement $\tilde{c}_{\mathrm{cor}}$, a hard optimization gate g, and a graded duration quality signal q.

D.2 Classes of Optimization RL Environments

The construction and timing stack in Appendices B and C give us optimization tests, calibrated human-reference durations, and a sandbox that can record per-test timing, but these ingredients are not yet a good optimization RL environment. A good environment should induce a useful learning curve: early rollouts should still receive a correctness signal, correct but slow snippets should not be indistinguishable from wrong code, and progressively faster correct snippets should see progressively better optimization feedback. Left unshaped, raw optimization tests can make the rollout

Table 19: **Notation for optimization-aware RL environments.** The same symbols apply to pre-execution filters, intra-execution time limits, and post-execution ranking rewards.

Symbol	Meaning
$\mathcal{E}, x, y$	RL environment, programming problem, and generated candidate program
u_t, v_t, τ_t	input, expected output, and default execution cap for test t
$T_0(x)$	original public, private, and generated correctness tests
$T_+(x)$	additional correctness tests used in more-correctness settings
$T^{\mathrm{C}}(x)$	correctness tests used to judge semantic correctness
$T^{\mathrm{O}}(x)$	raw optimization-test pool
$T^{\mathrm{O}}_{\mathrm{used}}(x)$	optimization tests retained by pre-execution filtering
$\mathcal{Z}^{\mathrm{O}}(x, y)$	optimization-side execution record for candidate y
$H(x)$	calibrated human/reference solutions with stored duration traces
$\tilde{d}_{t,h}$	calibrated duration of reference solution h on test t
D_t	calibrated duration distribution for test t over $H(x)$
$A, \bar{d}_t$	aggregation operator, either mean or median, and aggregate duration $\bar{d}_t = A(D_t)$ used by duration filters
$\mathcal{S}$	execution statuses: success, timeout, or failure
$s_t(y), \delta_t(y)$	execution status and measured duration for candidate y on test t
λ_t	intended optimization time limit for test t
ϕ, ρ	optimization timeout fraction and timeout-tolerance threshold
$p_t(y), a(y)$	normalized per-test reference percentile and mean per-test percentile for candidate y, with lower values faster
$c_{\mathrm{cor}}, \tilde{c}_{\mathrm{cor}}$	correctness-test pass indicator and strict optimization-aware correctness prerequisite used before efficiency credit
$g, g_{\mathrm{to}}, g_{\mathrm{dur}}$	final hard optimization gate, timeout-ratio gate, and duration-ranking gate
$q, q_{\mathrm{to}}, q_{\mathrm{leader}}, q_{\mathrm{test}}$	final graded optimization scalar, timeout-rate scalar, leaderboard percentile, and mean per-test percentile, all normalized so 0 is best or neutral and 1 is worst

mostly a harder correctness check, make the reward too sparse because only near-top solutions ever receive efficiency credit, or make the scalar too focused on speed and give credit to degenerate fast-but-wrong programs. The sparsity regime also matters: a constraint that gives efficiency credit only to the top 1% of isolated candidate snippets may never reward early GRPO rollouts, while a loose constraint can produce dense rewards with little optimization pressure. The raw-duration controls in Appendix D.5 and the baseline discussion in Section 5 show this limitation empirically: even when optimization tests are included in the rollout, directly adding duration to the scalar reward gives only small strict-threshold gains relative to reference-normalized environment/reward choices.

We study optimization RL as a one-shot generation-from-scratch problem. The model produces one program from the problem statement, not an iterative refinement trajectory around a known correct solution, and the environment assigns $R_{\mathcal{E}}(x, y)$ to that one generated program y for one problem x. Calibrated human references may set filters, time limits, or ranks, but any relative comparison among model snippets appears later, when GRPO turns the scalar rewards of a sampled batch into advantages; it is not part of the environment score itself. The goal of the taxonomy is therefore to be exhaustive with respect to this rollout pipeline: a timing constraint can enter before sandbox execution, during sandbox execution, or after execution. Within each stage, an environment family fixes the type of constraint, and a concrete environment instance fixes its parameterization.

Any environment starts from the loaded sample x, before anything is queried to the model. The sample provides the problem statement, the correctness tests $T_0(x)$ and $T_+(x)$, the raw optimization-test pool $T^{\mathrm{O}}(x)$, and the reference duration traces from $H(x)$. At this stage, the environment fixes which correctness tests will judge the answer, with $T^{\mathrm{C}} = T_0$ for our baseline standard RLVR and $T^{\mathrm{C}} = T_0 \cup T_+$ for any of our optimization-aware environments. In the case of any optimization environment, it starts with the full pool $T^{\mathrm{O}}(x)$ of optimization tests before any pre-execution filter or time-limit rule is applied. Stored reference human durations on these optimization tests pass through

$$\psi(d) = \begin{cases} d, & d > 0, \\ 10, & \text{otherwise}, \end{cases} \qquad \tilde{d} = \min\{10, \max\{0, \Gamma(\psi(d))\}\}, \tag{13}$$

where Γ is either the identity or an affine correction fitted from repeated CES measurements; Appendix C.7 gives the calibration evidence. Candidate durations $\delta_t(y)$ are not transformed by Γ: they are measured in the live execution-service scale and compared to calibrated reference distributions

only after execution. For each optimization test t, let $D_t = \{\tilde{d}_{t,h} : h \in H(x)\}$ be the sanitized reference-duration distribution when available. The aggregate duration used by duration filters is $\bar{d}_t = A(D_t)$, where A is either the mean or the median.

The model is then queried with the problem statement and returns a candidate program y. The environment must now judge the quality of y, beginning with the subset of optimization tests that will be run.

Pre-execution filtering is the part of the environment that acts before the generated program is run. The no-op filter keeps the entire optimization pool,

$$F_{\mathrm{id}}(T) = T. \tag{14}$$

Active pre-execution families, in contrast, are parameterized filters that select a subset of these optimization tests. We explore the following subclasses of pre-execution environments:

1. **Absolute-duration filtering** uses the calibrated human reference durations to keep only optimization tests whose aggregate duration is below a threshold a:

$$F^{a}_{\mathrm{abs}}(T) = \{t \in T : \ D_t \text{ usable}, \ \bar{d}_t < a\}. \tag{15}$$

   More concretely, what it does is:

   (a) For each optimization test, look at the recorded durations of the references on that test and compute the test-level mean duration $\bar{d}_t$.

   (b) Keep only tests with $\bar{d}_t < a$. For example, with $a = 2\,\mathrm{s}$, keep only optimization tests whose mean reference duration is below $2\,\mathrm{s}$.

   (c) Run only these retained tests with a uniform optimization timeout, for instance $\ell = 0.5\,\mathrm{s}$. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.

2. **Character-length filtering** uses the size of the input-output pair as a proxy for workload and keeps only tests below a cutoff L:

$$F^{L}_{\mathrm{len}}(T) = \{t \in T : \ |u_t| + |v_t| < L\}. \tag{16}$$

   More concretely, what it does is:

   (a) For each optimization test, compute the serialized input-output size $|u_t| + |v_t|$.

   (b) Keep only tests with $|u_t| + |v_t| < L$. For example, with $L = 10^3$, keep tests whose input plus output has fewer than $1{,}000$ characters.

   (c) Run only these retained tests with a uniform optimization timeout, for instance $\ell = 0.5\,\mathrm{s}$. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.

3. **Relative-duration filtering** removes the slowest calibrated fraction of the optimization pool. Let $T_D = \{t \in T : D_t \text{ usable}\}$ and let $S_r(T_D)$ be the $\lfloor r|T_D|/100 \rfloor$ slowest tests in T_D under $\bar{d}_t$. Then

$$F^{r}_{\mathrm{rel}}(T) = T \setminus S_r(T_D). \tag{17}$$

   More concretely, what it does is:

   (a) For each optimization test, look at the recorded durations of the references on that test, compute the test-level mean duration $\bar{d}_t$, and sort the rankable optimization tests by $\bar{d}_t$, slowest first.

   (b) Drop the slowest $r\%$ of rankable optimization tests. For example, with $r = 80$, drop the slowest 80% and keep roughly the fastest 20%.

   (c) Run only these retained tests with a uniform optimization timeout, for instance $\ell = 0.5\,\mathrm{s}$. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.

Thus F^{a}_{abs} retains only tests whose calibrated aggregate is defined and below the cutoff, F^{L}_{len} depends only on input-output character count, and F^{r}_{rel} retains tests without usable aggregate durations while dropping the slowest calibrated fraction among the tests that can be ranked. These are parameterized families: the environment sweep varies a, L, or r, and otherwise leaves the rest of the rollout routine fixed.

These filters do not by themselves reward speed; they choose which workloads are allowed to contribute to the later timeout or ranking signal. Absolute-duration filtering makes the workload scale explicit. With $a = 2\,\mathrm{s}$, for example, the retained optimization tests are those whose calibrated aggregate human runtime is below two seconds, so choosing a comparable optimization cap has an interpretable scale. The same global threshold can nevertheless mean different things across problems: some problems already live entirely below two seconds and are effectively unfiltered, while others have a duration profile that the threshold cuts into two very different subsets. Character-length filtering connects to the duration/length filterability analysis in Section 3 and Appendix B.3. It is cheap, deterministic, and can be applied even without stored reference durations, but it assumes serialized input-output length is a usable proxy for computational workload; this assumption can fail when runtime is driven by input structure, values, or control flow rather than character count. Relative-duration filtering avoids a global cutoff by removing a per-problem fraction of slow calibrated tests, which can look better matched to heterogeneous tasks. Its risk is that the removed tail may be exactly where the optimization pressure lives: even a small removal fraction can delete hard cases on some problems, and because the ordering comes from the human reference pool, it may overfit to references by discarding tests that are slow on average across human solutions but could have produced a broad duration spread across model rollouts during RL. Tuning a, L, and r is therefore part of defining the learning curve, trading broad coverage against reward sparsity and strict optimization pressure.

Once the optimization-test pool is fixed (either as the full pool under F_{id} or as a subset selected by a pre-execution environment), the candidate program y and the retained tests are sent to the execution sandbox. The sandbox is controlled by the timeout assigned to each test. In the default execution setting, the intended time limit is the sample's default cap, $\lambda_t = \tau_t$. Active intra-execution environments act on the sandbox by replacing this default with a parameterized optimization timeout:

1. **Absolute-timeout environments** use the same intended optimization limit ℓ for every retained optimization test:
$$\lambda_t^{\mathrm{abs}}(\ell) = \ell. \tag{18}$$
More concretely, what it does is:
   (a) Start from the retained optimization-test pool.
   (b) Set the same intended optimization timeout for every retained test. For example, with $\ell = 0.5\,\mathrm{s}$, every retained optimization test has deadline $0.5\,\mathrm{s}$.
   (c) Run the retained tests. If a successful execution has $\delta_t(y) \geq \ell$, record that test as a timeout. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.
2. **Relative-timeout environments** set a per-test limit from the calibrated reference distribution for that test:
$$\lambda_t^{\mathrm{rel}}(p) = \max\{10^{-3}, \mathrm{Percentile}_p(D_t)\}. \tag{19}$$
More concretely, what it does is:
   (a) Start from the retained optimization-test pool, and for each retained test read the recorded reference durations D_t.
   (b) Set the intended timeout of each test to a percentile of its own reference durations. For example, with $p = 30$, test t receives the 30th-percentile duration of D_t as λ_t.
   (c) Run the retained tests. If a successful execution has $\delta_t(y) \geq \lambda_t$, record that test as a timeout. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.
3. **Ranked-reference timeout environments** first select a top reference set $H_p(x)$, then set each test limit from the selected reference durations:
$$\lambda_t^{\mathrm{ranked}}(p, B) = B\big(\{\tilde{d}_{t,h} : h \in H_p(x)\}\big). \tag{20}$$
More concretely, what it does is:
   (a) Rank the reference solutions on the problem by their average per-test rank on the optimization tests, and keep the best $p\%$ as $H_p(x)$.
   (b) For each optimization test t, set λ_t from the selected reference durations $\{\tilde{d}_{t,h} : h \in H_p(x)\}$. For example, ranked-worst $p = 80$ with $B = \max$ keeps the top 80% of references and sets λ_t to their slowest duration on each test.

(c) Run the retained tests. If a successful execution has $\delta_t(y) \geq \lambda_t$, record that test as a timeout. With timeout tolerance $\rho = 0.1$, the optimization side passes if at most 10% of retained tests time out.

Here ℓ is an absolute time limit in seconds, p is a reference percentile, B is either the maximum or median aggregation, and $H_p(x)$ is the best $p\%$ of reference solutions after ranking references by their average per-test rank on $T^{\mathrm{O}}_{\mathrm{used}}(x)$. The retained ranked-worst environments use $B = \max$. When a reference-derived limit needs D_t, tests without usable reference durations can be omitted before execution; in that case $T^{\mathrm{O}}_{\mathrm{used}}$ denotes the remaining executed optimization tests. Whenever the CES path is given an optimization time-limit override, execution runs under the 10s service cap and then applies the intended limit by post-filtering: a successful test is changed to timeout whenever $\delta_t(y) \geq \lambda_t$.

Intra-execution environments introduce optimization pressure through the sandbox budget itself. This is different from pre-execution filtering: a pre-execution filter chooses which tests form the workload and then lets execution unfold blindly, whereas an intra-execution environment keeps the retained workload fixed and changes how much time the candidate is allocated to complete it. These choices are therefore not equivalent. Removing a test removes its evidence from both the hard gate and the graded signal, while tightening λ_t leaves the test in the rollout but changes whether the run becomes a timeout. Absolute timeouts are the simplest version of this idea and make the deadline easy to read, but a single ℓ can be too coarse when the retained tests span very different workloads. Relative timeouts adapt test by test by extracting a quantile from D_t, but they can assemble an unrealistic simultaneous constraint when different tests reward different algorithmic trade-offs: the 20% fastest duration on each test need not come from the same kind of solution. Ranked-reference timeouts try to avoid that per-test mixture by first selecting a coherent pool of strong reference solutions and then deriving deadlines from their traces. They can fail in the opposite direction: if each selected reference is weak on a different subset of tests, the resulting deadlines may allow a candidate to match the worst selected reference on every test without matching any reference's strengths. Both reference-derived variants also inherit the biases of the human reference pool, so their parameterization controls not only strictness but also whether the environment potentially overfits the observed reference pool.

After execution, the optimization-side record is

$$\mathcal{Z}^{\mathrm{O}}(x, y) = \{(t, s_t(y), \delta_t(y), \lambda_t) : t \in T^{\mathrm{O}}_{\mathrm{used}}(x)\}. \tag{21}$$

Once execution finishes, statuses and durations are summarized separately. Execution statuses produce the correctness signal in the same way for all environment families:

$$c_{\mathrm{cor}}(x, y) = 1\{s_t(y) = \text{success}\ \forall t \in T^{\mathrm{C}}(x)\}, \tag{22}$$

where $T^{\mathrm{C}} = T_0$ for standard RLVR and $T^{\mathrm{C}} = T_0 \cup T_+$ for the "more-correctness" setting and for all optimization RL environment classes considered below. Optimization-aware rewards use the stricter prerequisite

$$\tilde{c}_{\mathrm{cor}}(x, y) = c_{\mathrm{cor}}(x, y)\, 1\{s_t(y) \in \{\text{success}, \text{timeout}\}\ \forall t \in T^{\mathrm{O}}_{\mathrm{used}}(x)\}. \tag{23}$$

Thus executed optimization tests may be correct or may time out, but any failure on an executed optimization test makes $\tilde{c}_{\mathrm{cor}} = 0$. At this point, the environment has produced its first reward-facing output signal, $\tilde{c}_{\mathrm{cor}}$: a strict correctness prerequisite that can be fed to a reward function.

The same status summary also records the optimization timeout fraction

$$\phi(x, y) = \frac{1}{|T^{\mathrm{O}}_{\mathrm{used}}(x)|} \sum_{t \in T^{\mathrm{O}}_{\mathrm{used}}(x)} 1\{s_t(y) = \text{timeout}\}, \tag{24}$$

with $\phi = 0$ when no optimization tests are used. We use the same lower-is-better convention as the percentile quantities below: $q = 0$ is best and $q = 1$ is worst, with reward functions later converting q into higher-is-better credit. A timeout tolerance ρ defines the status-side optimization hard gate, and the same timeout fraction can be exposed as a graded optimization scalar:

$$g^{\rho}_{\mathrm{to}}(x, y) = 1\{\phi(x, y) \leq \rho\}, \qquad q_{\mathrm{to}}(x, y) = \phi(x, y). \tag{25}$$

The setting $\rho = 1$ makes the timeout gate a no-op, while smaller values turn the timeout fraction into a hard optimization constraint. This tolerance parameterization is not a class of its own but

an attached parameter of the different classes of optimization RL environments, because it directly controls how timeout statuses are transferred into downstream reward impact. Set to $\rho = 1$, it shuts down any optimization pressure through the hard optimization gate for the workload selected by pre-execution filters and constrained by intra-execution time limits.

Active post-execution environments process the measured per-test durations $\{\delta_t(y) : t \in T^{\mathrm{O}}_{\mathrm{used}}\}$ after the sandbox has finished. A raw-duration reward is the simplest version of this idea: it averages durations over a chosen test set and normalizes that average into the reward range, without using the calibrated reference pool. The reference-normalized variants go further by comparing the candidate to $H(x)$. For each test in $T^{\mathrm{O}}_{\mathrm{used}}$ with usable D_t, the candidate duration is inserted into the reference durations and scored as a normalized percentile $p_t(y) \in [0, 1]$, with 0 assigned to the fastest participant and 1 to the slowest; exact ties receive the previous participant's percentile. The mean-percentile statistic averages only over retained tests with usable reference durations:

$$a(y) = \frac{1}{|\{t \in T^{\mathrm{O}}_{\mathrm{used}} : D_t \text{ usable}\}|} \sum_{\substack{t \in T^{\mathrm{O}}_{\mathrm{used}} \\ D_t \text{ usable}}} p_t(y). \tag{26}$$

This gives the mean per-test percentile family: in the retained ranking variants, the per-test rank is normalized as the percentile $p_t(y)$, and this family uses the aggregate $a(y)$ directly,

$$q_{\mathrm{test}}(y) = a(y). \tag{27}$$

The leaderboard percentile family uses the same aggregate only as an intermediate score: it sorts the candidate and references by $a(\cdot)$, then converts the candidate's solution-level position into a leaderboard percentile. Let $b(y)$ be the zero-indexed position of y in that ordering, and let N_{rank} be the number of ranked participants. Its graded scalar is

$$q_{\mathrm{leader}}(y) = \frac{b(y)}{\max\{1, N_{\mathrm{rank}} - 1\}}. \tag{28}$$

Post-execution ranking is parameterized by a duration threshold $p \in [0, 1]$. This turns into a hard optimization gate on the selected ranked scalar:

$$g^p_{\mathrm{dur,test}}(y) = 1\{q_{\mathrm{test}}(y) < p\}, \tag{29}$$

$$g^p_{\mathrm{dur,leader}}(y) = 1\{(b(y) + 1)/N_{\mathrm{rank}} \le p\}, \tag{30}$$

$$g^p_{\mathrm{dur}}(y) \in \{g^p_{\mathrm{dur,test}}(y), g^p_{\mathrm{dur,leader}}(y)\}, \tag{31}$$

$$q_{\mathrm{dur}}(y) \in \{q_{\mathrm{test}}(y), q_{\mathrm{leader}}(y)\}. \tag{32}$$

The leaderboard gate always admits the first position, and references tied with y are placed before it. When no duration post-processing is configured, $g_{\mathrm{dur}} \equiv 1$ and the final scalar below falls back to $q = q_{\mathrm{to}}$ by setting $q_{\mathrm{dur}} \equiv 0$. Conversely, when an environment relies only on duration-based post-execution ranking, we set $\rho = 1$, so $g^{\rho}_{\mathrm{to}} \equiv 1$, and use infinite intended optimization timeouts so that $q_{\mathrm{to}} \equiv 0$ and $q = q_{\mathrm{dur}}$. In practice, each test execution is still capped at 10s for systems reasons, and timeout durations are recorded at that cap before being ranked. **More concretely, what post-execution ranking does is:**

(a) Run the retained optimization tests under the 10s service cap and record one duration $\delta_t(y)$ for each executed test; if the run reaches the cap, record $\delta_t(y) = 10\,\mathrm{s}$.

(b) For each usable test, insert $\delta_t(y)$ into the recorded reference durations D_t and compute the per-test percentile $p_t(y) \in [0, 1]$.

(c) Example: the mean per-test percentile scalar averages these percentiles into $q_{\mathrm{test}}(y) = a(y)$. The leaderboard percentile scalar ranks the same $a(y)$ against the reference aggregates to compute $q_{\mathrm{leader}}(y)$; with duration threshold $p = 0.3$, the duration gate passes candidates that rank in the top 30% under the selected ranked scalar.

For the top-$X\%$ leaderboard-percentile rows of Table 1, the binary optimization gate is $g = g^{X/100}_{\mathrm{dur,leader}}$, that is, the candidate position must be inside the top $X\%$ of this leaderboard. Optimization timeouts are tolerated for status, but their durations still affect ranking. The two-gate bucketed rows apply no duration gate ($g_{\mathrm{dur}} \equiv 1$) and use $q = q_{\mathrm{leader}}$ or $q = q_{\mathrm{test}}$.

Post-execution ranking applies optimization pressure by making the candidate compete against a reference leaderboard after execution, instead of choosing a workload cutoff before execution or changing the sandbox budget during execution. Relative to pre- and intra-execution families, this moves the strictness choice out of workload construction: the environment does not need to tune a, L, r, or a per-test deadline λ_t to make speed visible, although a downstream threshold p can still turn the ranked scalar into a hard gate. In practice, the model is rewarded for moving upward in the reference ordering on the same problem, which is close to how many code-optimization benchmarks are evaluated as ranked speed comparisons. The price is that the reference distribution becomes part of the environment. If $H(x)$ is weak, too narrow, or biased toward one implementation family, then a generated solution near the reference pack and a much faster solution may not be separated enough, or the model may overfit to the observed reference trade-offs. Note that this does not make this type of environment less generalizable, or stuck in this potential pitfall: the reference distribution could be generated, learned, refreshed adversarially against the current model, or augmented with past rollout positions. We keep it fixed here for the sake of simplicity, as stated in the limits of our study at the beginning of this subsection.

The mean-per-test-percentile/leaderboard-percentile split is the main scalarization choice once post-execution ranking is selected. The mean per-test percentile keeps the average $a(y)$ itself, so it rewards broad movement across the retained tests and makes the per-test-to-problem aggregation choice explicit; the displayed definition uses a mean, while Appendix D.3 compares this family to median, trimmed, and filtered variants. The leaderboard percentile takes the same aggregate $a(y)$, sorts the candidate among the references, and converts that position into a problem-level leaderboard percentile. The tradeoff is dynamic range. If different reference solutions are fast on different tests and slow on others, the raw average $a(y)$ can compress many candidates into a narrow interval; the leaderboard percentile spreads this aggregate back into leaderboard positions, making more relative positions visible to the reward. The cost is saturation at the extreme: once a candidate is best on the existing ladder, the leaderboard percentile remains at the best position, while the mean per-test percentile can still decrease if the candidate keeps improving across tests beyond any pre-existing reference. Appendix D.3 therefore studies which ranked scalar has low repeated-CES noise, enough within-problem discrimination, and stable ordering after calibration, before the reward-composition subsection decides how to consume g and q.

The final optimization interface multiplies the status-side and duration-side gate factors,

$$\begin{aligned} g(x,y) &= g^{\rho}_{\mathrm{to}}(x,y)\, g^{p}_{\mathrm{dur}}(x,y), \\ q(x,y) &= 1 - \big(1 - q_{\mathrm{to}}(x,y)\big)\big(1 - q_{\mathrm{dur}}(x,y)\big) \\ &= \begin{cases} q_{\mathrm{dur}}(x,y), & \text{for active post-execution ranking,} \\ q_{\mathrm{to}}(x,y), & \text{otherwise.} \end{cases} \end{aligned} \tag{33}$$

where inactive gate factors are set to 1 and inactive graded scalars are set to 0. Thus the environment exposes one hard optimization gate g and one graded optimization scalar q, after separating the correctness prerequisite $\tilde{c}_{\mathrm{cor}}$.

Evaluation metric p_τ. When using a leaderboard-percentile post-execution environment as an evaluation, this directly defines the p_τ family of pass@k metrics used throughout the paper in many tables and figures. Let $q_{\mathrm{lead}}(x,y) \in (0,1]$ be the leaderboard percentile of a generated solution y, that is, its one-indexed position divided by the number of entries after inserting it into the calibrated human reference leaderboard for problem x (unlike the zero-indexed graded scalar q_{leader}), with lower values faster. For an evaluation set $\mathcal{X}$ and n sampled solutions $y_1, \ldots, y_n$ per problem, define

$$m_\tau(x) = \sum_{i=1}^{n} \mathbb{I}\Big\{ \tilde{c}_{\mathrm{cor}}(x,y_i) = 1 \ \wedge \ q_{\mathrm{lead}}(x,y_i) \le \frac{\tau}{100} \Big\}, \tag{34}$$

$$\mathrm{pass@}k(p_\tau) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \left(1 - \frac{\binom{n - m_\tau(x)}{k}}{\binom{n}{k}} \right). \tag{35}$$

Here $\tilde{c}_{\mathrm{cor}}$ is the strict correctness prerequisite: it checks that all base and generated correctness tests succeed, and that no optimization test has a hard failure (but timeouts are not penalized). Thus p_{100} pass@1 does not have any leaderboard-ranking constraint and is equivalent to a pure-correctness score, while smaller τ requires the solution to both satisfy the strict correctness prerequisite and

rank in the top $\tau\%$ of the human reference leaderboard. The top-$X\%$ leaderboard-percentile training environments use the same correctness and leaderboard gate with $\tau = X$, under the training-time duration calibration. Evaluation ranks each sample with mean 1-based per-test ranks, which orders entries exactly like mean per-test percentiles when every entry has a duration.

Table 20 lists the exact operator, gate, and reward behind each row family of Table 1; the reward families are defined in Appendix D.4.

Table 20: **Exact environment operators behind the Qwen 2.5 7B environment sweep.** All rows use the same prompt family, rollout budget, and execution backend; the table only lists the environment and reward differences. Rows below the large-timeout row use $T^{\mathrm{C}} = T_0 \cup T_+$ with a 10 s correctness cap and include the optimization tests.

Rows in Table 1	Optimization operator	Gate and reward
Standard RLVR	$T^{\mathrm{C}} = T_0$ $T^{\mathrm{O}}_{\mathrm{used}} = \emptyset$.	Correctness only R_{corr}.
More-correctness tests	$T^{\mathrm{C}} = T_0 \cup T_+$ $T^{\mathrm{O}}_{\mathrm{used}} = \emptyset$.	R_{corr}.
More-correctness tests + large 10 s timeout	Same with a 10 s cap on correctness tests.	R_{corr}.
More-correctness + optimization tests	$T^{\mathrm{O}}_{\mathrm{used}} = T^{\mathrm{O}}$ 10 s optimization cap.	Collapsed binary with $g = g^{0}_{\mathrm{to}}$: all optimization tests must succeed.
Absolute-duration filter	$T^{\mathrm{O}}_{\mathrm{used}} = F^{a}_{\mathrm{abs}}(T^{\mathrm{O}})$ $a \in \{0.1, 0.5, 2\}$ s $\lambda_t = 0.5$ s.	Collapsed binary with $g = g^{0.1}_{\mathrm{to}}$.
Character-length filter	$T^{\mathrm{O}}_{\mathrm{used}} = F^{L}_{\mathrm{len}}(T^{\mathrm{O}})$ $L \in \{10^3, 10^5, 10^7\}$ characters $\lambda_t = 0.5$ s.	Collapsed binary with $g = g^{0.1}_{\mathrm{to}}$.
Relative-duration filter	$T^{\mathrm{O}}_{\mathrm{used}} = F^{r}_{\mathrm{rel}}(T^{\mathrm{O}})$ $r \in \{30, 50, 80\}$ dropping the slowest $r\%$ of tests $\lambda_t = 0.5$ s.	Collapsed binary with $g = g^{0.1}_{\mathrm{to}}$.
Absolute timeout	No pre-filter $\lambda_t = T$ $T \in \{0.5, 2\}$ s.	Collapsed binary with $g = g^{0.1}_{\mathrm{to}}$.
Ranked-worst timeout	No pre-filter $\lambda_t = \lambda^{\mathrm{ranked}}_t(p, \max)$ $p \in \{20, 50, 80\}$ tests without a usable D_t are skipped.	$p \in \{20, 80\}$: collapsed binary with $g = g^{0.1}_{\mathrm{to}}$. $p = 50$: two-gate bucketed with $q = q_{\mathrm{to}}$.
Relative timeout (5k steps Table 36)	No pre-filter $\lambda_t = \lambda^{\mathrm{rel}}_t(p)$ $p \in \{30, 50, 80\}$ tests without a usable D_t are skipped.	Collapsed binary with $g = g^{0.1}_{\mathrm{to}}$.
Leaderboard percentile / mean per-test percentile two-gate bucketed	No pre-filter 10 s optimization cap post-execution ranking against D_t no duration gate.	Two-gate bucketed with $q = q_{\mathrm{leader}}$ or $q = q_{\mathrm{test}}$.
Leaderboard percentile top $X\%$	No pre-filter 10 s optimization cap post-execution leaderboard gate $X \in \{80, 50, 30\}$.	Collapsed binary with $g = g^{X/100}_{\mathrm{dur,leader}}$. Optimization timeouts are tolerated for status, but their durations still affect ranking.

D.3 Ranking metric screening

The graded post-execution branch needs a ranking metric that is stable under repeated CES measurements, aligned with fresh-CES ordering after calibration, and still discriminative. We screened

17 candidates: mean, median, trimmed-mean, and log-space percentiles; top-25 and top-50 shares; total-duration percentile; slope-based percentiles with and without intercept; win rate against the human mean and median; and filtered percentile variants that discard easiest or hardest tests.

Several families fail one of these requirements. Win-rate metrics can have zero within-problem variance and fail to distinguish solutions. Median-based metrics have cross-validation alignment standard deviations around 10–18 percentile points. Top-k shares become less aligned after correction, with cross-validation differences of $+20.1$ for top-25 and $+6.6$ for top-50. Total-duration percentile has mean coefficient of variation around 20%, substantially noisier than percentile means. The mean-based percentile family is the only family that remains both stable and discriminative.

Table 21: **Finalists inside the mean-based percentile family.** The table reports the cross-validation alignment standard deviation between corrected-stored rankings and fresh-CES gold rankings, together with within-problem standard deviation across reruns. Lower is better. Values are percentile points.

Metric	CV std (affine)	CV std (poly)	Within-problem std
Mean percentile	**3.8**	**2.9**	3.0 ± 1.0
Trimmed mean percentile	5.1	4.0	3.5 ± 1.2
Mean percentile, discard fastest 10%	4.2	3.3	2.8 ± 1.0
Mean percentile, discard slowest 25%	4.2	3.3	3.7 ± 1.1
Mean percentile, discard slowest 50%	4.8	4.0	5.3 ± 1.7

Mean percentile is retained because it has the lowest cross-validation dispersion, comparable rerun noise to filtered variants, and a 47-point spread between strong and weak solutions without adding trimming hyperparameters. The leaderboard percentile used in some reward variants is derived from the same aggregate ranking; it gives a more interpretable competitive scale but preserves almost the same ordering.

D.4 Reward composition families

Once an environment is reduced to (c, g, q), the reward ablation is a composition rule. We use the implementation-scale convention $q \in [0, 100]$, where $q = 0$ is best. Let c_{oc} be the correctness-test pass indicator, let h be the no-hard-error guard, and let $c = c_{\mathrm{oc}}h$ be the strict correctness gate ($c_{\mathrm{oc}} = c_{\mathrm{cor}}$ and $c = \tilde{c}_{\mathrm{cor}}$ in Appendix D.2) used by the reward families that must not give efficiency credit to wrong or crashing programs. The binary optimization signal is $g \in \{0, 1\}$, either a timeout-tolerance gate or a top-percentile ranking gate. For graded rewards we use

$$f_{\mathrm{B}}(q) = \begin{cases} 1, & q = 0, \\ \max\{0,\ 0.9 - 0.1\lfloor (q - 10^{-4})/10 \rfloor\}, & q > 0, \end{cases} \quad f_{\mathrm{C}}(q) = \mathrm{clip}\left(1 - \frac{q}{100}, 0, 1\right), \quad (36)$$

where f_{B} is the ten-bin map used in the main reward grid and f_{C} is the continuous variant. The reward-composition grid instantiates these formulas on two environment sources. In the duration-filtered source, $T^{\mathrm{O}}_{\mathrm{used}} = F_{\mathrm{abs}}(2\mathrm{s})$ and the intended optimization cap is 0.5s; binary cells use $g = 1\{\phi \le 0.1\}$, while graded cells set $q = 100\phi$ and disable the hard optimization gate with tolerance $\rho = 1$. In the ranked source, optimization tests are kept intact under a 10s cap; binary cells use $g = 1\{100\, a(y) < 30\}$, except the collapsed binary cell, which is the top-30% leaderboard-percentile run, and graded cells use $q = 100\, a(y)$ with $p = 100$, where $a(y)$ is the mean per-test percentile from Appendix D.2.

The distinction between c_{oc} and $c = c_{\mathrm{oc}}h$ matters. Collapsed and two-gate rewards require both correctness and the hard-error guard before giving any efficiency credit, so timeouts may be tolerated only through the environment's explicit timeout-tolerance rule, while failures, exceptions, and missing executions stay at -1. Optimization-only, blend, and multitask variants were included as negative controls precisely because they decouple speed from full correctness in different ways.

All graded families keep the global reward range in $[-1, 1]$. The simple continuous map sends passing solutions to $1 - q/100$. The bucketed map discretizes the same interval into ten bins. The balanced and correctness-emphasis variants compress the positive range to $[1/3, 1]$ and $[1/2, 1]$, respectively, so that the gap between failure and the worst passing solution is larger than the optimization range among passing solutions. These mappings are ablations around the same correctness-gated principle used in the main experiments.

Table 22: **Reward-composition families used in the ablation grid.** All rows consume the same environment outputs (c, g, q), but they differ in whether correctness is a hard prerequisite, whether speed can compensate for wrong code, and whether correctness and optimization are mixed inside one rollout or across separate task streams.

Family	Formula	Interpretation
Correctness only	$R_{\text{corr}} = 2c_{\text{oc}} - 1$	Standard RLVR control; optimization tests may be present as correctness tests, but no graded timing term is exposed.
Optimization only	$R_{\text{opt}-\text{bin}} = 2g - 1$, $R_{\text{opt}-\text{gr}} = 2f(q) - 1$	Removes the correctness signal and tests whether a pure duration-side objective can sustain code generation. It does not use c or h.
Collapsed	$R_{\text{coll}-\text{bin}} = 2cg - 1$, $R_{\text{coll}-\text{gr}} = 2cf(q) - 1$	Multiplies correctness and optimization into one scalar. A correct but slow graded solution can receive the same reward as an incorrect one.
Two-gate	$R_\eta = -1$ if $c = 0$, else $R_\eta = \eta + (1-\eta)f(q)$	Keeps correctness as an outer gate. The primary two-gate bucketed settings use $\eta = 0$; earlier compressed variants use $\eta = 1/3$ or $1/2$, and an optimization-emphasis variant with $\eta = -1/3$ was also defined but is not part of the primary two-gate results reported here.
Additive blend	$R_{\text{blend}} = \lambda(2c_{\text{oc}} - 1) + (1-\lambda)(2f(q) - 1)$, with binary g replacing $f(q)$ in binary cells	Tests whether direct weighted sums are enough. The primary 1:1 cells use $\lambda = 1/2$; 2:1 and 1:2 variants use $2/3$ and $1/3$. Because this uses c_{oc}, a wrong but fast program can recover partial reward.
Multitask	Sample either R_{corr} or R_{opt} from separate task streams	Alternates correctness-only and optimization-only rollouts instead of combining their rewards within a single rollout. The primary variants use a 1:1 task mixture, with 2:1 and 1:2 variants defined analogously.

D.5 Naive raw-duration baselines

The naive-duration baselines test the simplest alternative to reference-normalized timing: add raw seconds directly to the correctness reward. Let $\bar{d}_{\text{base}}$ be the mean duration on the original public, private, and generated correctness tests, excluding the additional correctness tests, and let $\bar{d}_{\text{opt}}$ be the mean duration on optimization tests. Timeouts, missing durations, and non-positive durations are replaced by the effective time limit for that branch before averaging; if a branch has no tests, the same effective limit is used. The four baselines choose $s \in \{\text{base}, \text{opt}\}$ and one of two maps,

$$m_{\text{lin}}(\bar{d}_s) = \text{clip}\left(1 - \frac{\bar{d}_s}{10}, 0, 1\right), \tag{37}$$

$$m_{\log}(\bar{d}_s) = \text{clip}\left(\frac{1 - \log_{10}(\max\{\bar{d}_s, 10^{-6}\})}{3}, 0, 1\right), \tag{38}$$

then set

$$R_{\text{naive}}(x, y) = c_{\text{oc}}(x, y) + m(\bar{d}_s(x, y)) - 1. \tag{39}$$

Thus a correct 10s average solution receives 0, a correct 1s linear solution receives 0.9, and an incorrect but very fast solution can receive a reward close to 0 rather than -1. This is the behavior the reference-normalized families avoid.

These baselines are useful because they isolate the failure mode of absolute timing. A 0.1s program receives nearly the same raw-duration credit whether the problem's reference solutions run in 0.05s or 5s, so the reward is not normalized by problem difficulty or by the attainable speed distribution. Appendix F.5 reports the corresponding training results.

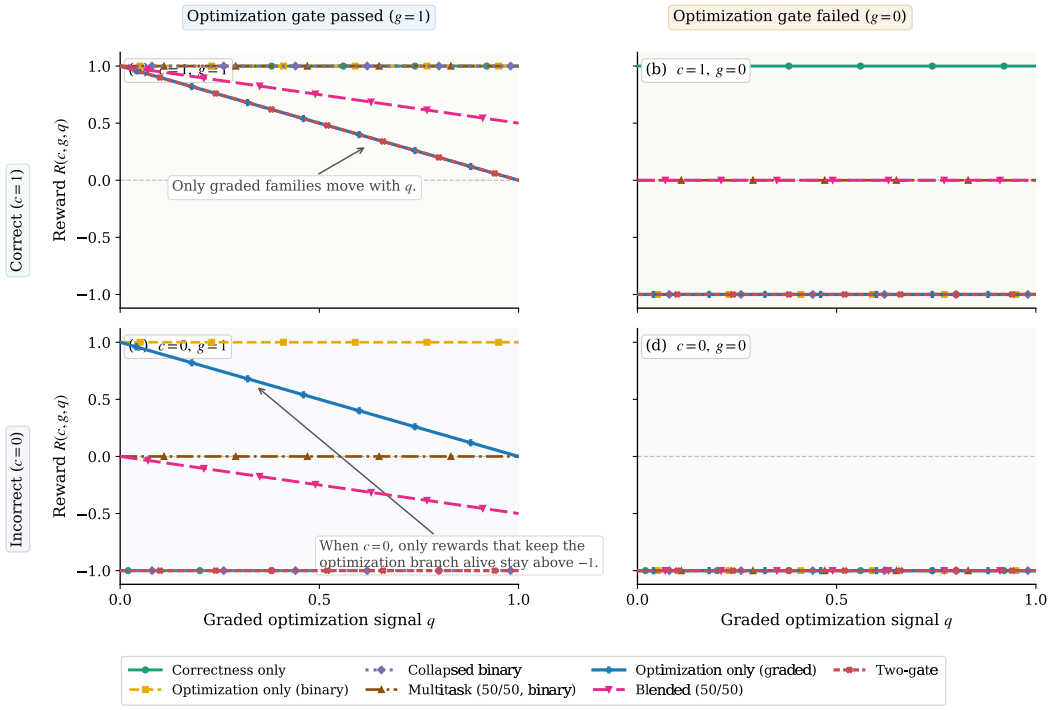


Figure 26: **Reward composition families in the shared (c, g, q) interface.** Rows fix the correctness gate c, columns fix the hard optimization gate g, and each trace shows the resulting reward $R(c, g, q)$ as the graded optimization signal q moves from best (0) to worst (1). The graded traces use the canonical linear map $f(q) = 1 - q$, and the graded optimization branch is clamped to -1 when $g = 0$. The seven traces correspond to correctness-only, binary optimization-only, collapsed binary, a 50/50 binary multitask variant, graded optimization-only, 50/50 additive blending, and the two-gate reward.

Table 23: **Naive-duration baselines.** The variants differ only in which tests provide the raw average duration and how that duration is mapped to $[0, 1]$. None compares the candidate to per-problem human references.

Variant	Duration source	Duration map
Base-linear	$\bar{d}_{\text{base}}$	m_{lin}
Base-log	$\bar{d}_{\text{base}}$	m_{log}
Optimization-linear	$\bar{d}_{\text{opt}}$	m_{lin}
Optimization-log	$\bar{d}_{\text{opt}}$	m_{log}

D.6 Taxonomy of optimization/efficiency definitions in previously published papers

The taxonomy built above tries to cover the different works already published around code optimization: they differ by how they define the optimization problem, and whether they train for it, evaluate on it, or both. A benchmark can construct hard inputs before execution, enforce time limits during execution, or rank a candidate after execution, though this does not necessarily make it an online RL environment. Similarly, an inference-time search system may use execution feedback as a fitness score without updating the model policy. Tables 24 and 25 therefore go over previously published studies on topics close to our work, and describe what they mainly do in light of the environment taxonomy and reward families defined above. The comparison remains high-level. Some systems use memory, instruction counts, complexity classes, or task-specific evaluator scores rather than wall-clock time, and small implementation choices can matter a lot.

We notice two things: first, benchmark papers often solve measurement but stop before the online reward question: they can use stress tests, hardware counters, right-censoring, or curve fitting that are carefully refined for a specific test set, including with heavy computations to make scores as reliable as possible, but can then become too expensive or too indirect to be evaluated on every GRPO rollout

Table 24: **Benchmark papers adjacent to our environment taxonomy.** Here we classify benchmark work by where efficiency enters the evaluation routine. **Pre-execution** refers to workload or test construction before the candidate is run; **intra-execution** refers to time limits or right-censoring that are part of the scoring rule; and **post-execution** refers to ranking or scoring a candidate after running it.

Paper	Kind	What it studies	Fit to our environment taxonomy	Training or reward analogue
MERCURY (Du et al., 2024)	Benchmark + SFT/DPO	Python efficiency benchmark with human/runtime reference distributions and the BEYOND percentile metric.	Mostly **post-execution** reference-runtime percentile scoring; timeout is used as an execution guard rather than a learned timing environment.	SFT on fast solutions and DPO over faster/slower correct solutions. The benchmark metric is correctness-gated, but the training signal is pairwise preference rather than one of our scalar reward compositions.
EFFIBENCH (Huang et al., 2024)	Benchmark	LeetCode-style benchmark measuring execution time and memory of generated Python code against canonical efficient solutions.	**Post-execution** normalized runtime and memory scoring; generated tests are benchmark construction, not train-time **pre-execution** filtering.	No training reward.
EFFIBENCH-X (Qing et al., 2025)	Benchmark	Multi-language extension of EFFIBENCH with expert references and relative execution-time and memory metrics.	**Post-execution** expert-reference-normalized time and memory scoring; translation and test generation happen before evaluation.	No training reward.
EVALPERF (Liu et al., 2024)	Benchmark	Creates performance-exercising inputs and reference-solution clusters, then evaluates candidates with profiled execution time.	**Pre-execution** benchmark construction for hard inputs, followed by **post-execution** runtime ranking against reference clusters.	No training reward; the benchmark improves input selection and runtime comparison, but remains an offline evaluation protocol rather than an online reward.
ENAMEL (Qiu et al., 2025)	Benchmark	HumanEval-derived benchmark with expert algorithms, level-based strong inputs, right-censored execution, and $\mathrm{eff@}k$.	Mixed **intra-execution** and **post-execution** design: reference-derived time limits and right-censoring are part of scoring, then candidates receive normalized efficiency scores.	No training reward.
COFFE (Peng et al., 2025b)	Benchmark	Function- and file-level code-efficiency benchmark using stress tests and stable CPU instruction-count measurements.	**Pre-execution** stress-test generation and selection, then **post-execution** correctness-gated instruction-count scoring.	No training reward.
SWE-PERF (He et al., 2025)	Benchmark	Repository-level benchmark from real performance-improving pull requests, with performance tests and expert patches.	**Post-execution** repository workload timing with a correctness gate and statistical performance-gain check; target functions and tests are benchmark curation.	No training reward.
SWE-FFICIENCY (Ma et al., 2025)	Benchmark	Repository-level workload benchmark where models patch real Python projects and are scored by captured expert speedup.	**Post-execution** workload timing normalized to expert speedup, with correctness and anti-cheating checks.	No training reward.
BIGO(BENCH) (Chambon et al., 2025)	Benchmark	Evaluates whether models can generate code with controlled time and space complexity through scaled execution and curve fitting.	**Post-execution** empirical complexity inference and ranking; complexity constraints may be requested in the prompt, but the evidence comes from measured growth curves.	No training reward.

or provide usable feedback to update the policy in the right direction. Second, training and search papers that do use execution feedback often work in an easier regime than one-shot generation, either by editing an existing program or selecting among candidates at inference time. Our environment definitions cover the main places where timing can enter such systems, and our retained training setting tries to avoid any form of simplification: one-shot generation from scratch, based on the problem description only, with no execution feedback, evaluated with a live execution sandbox that gives reward based on execution results.

Table 25: **Training and inference-time search papers.** The table separates online policy training from supervised training, preference optimization, and inference-time search. The taxonomy column highlights whether the execution feedback is mainly **pre-execution**, **intra-execution**, or **post-execution**; the final column describes the closest reward analogue.

Paper	Kind	What it studies	Fit to our environment taxonomy	Training or reward analogue
Performance-Aligned LLMs (Nichols et al., 2024)	Training	Uses supervised data, preference data, reward modeling, and PPO/RLPF-style optimization to generate faster code.	**Post-execution** runtime measurement on executable coding tasks; learned reward modeling is used when direct execution is not available.	For executable tasks, a correctness-gated continuous speedup reward: incorrect code gets a failure reward, correct faster code receives larger reward. This is close to gated continuous q, but without a separate hard optimization gate g.
SUPERCODER (Wei et al., 2025a)	Benchmark + RL	Assembly superoptimization against `gcc -O3`, with RL improving both correctness and speedup.	**Post-execution** compilation, testing, and speedup measurement for generated assembly.	Online RL uses a correctness-gated continuous reward: zero unless all tests pass, then speedup. This is closest to a two-gate-style reward with q as speedup, but no explicit ranked gate.
HPC online RL (Mikasa et al., 2026)	Training	Online GRPO for matrix multiplication on real machines, using measured GFLOPS as performance feedback.	**Post-execution** real-machine performance reward; staged restrictions on allowed optimization techniques are a curriculum-like **pre-execution** constraint.	Status-penalty ladder for format, compile, timeout, runtime, and verification failures, with continuous GFLOPS on success. This is not a strict two-gate or collapsed reward; it is closer to ordinal failure penalties plus continuous success quality.
AFTERBURNER (Du et al., 2025)	Training / refinement	Iterative code-efficiency refinement with SFT, DPO, and GRPO using runtime, memory, and integral feedback.	**Post-execution** feedback in a refinement environment: the model edits an existing solution after observing prior metrics, rather than solving one-shot from scratch.	GRPO uses an additive blend of format, correctness, and efficiency rewards. The efficiency term is gated by output correctness, but the final scalar is still a weighted blend rather than a hard correctness-preserving two-gate reward.
EFFICODER (Huang et al., 2025)	SFT + preference ablations	Builds efficient-code fine-tuning data by selecting candidates with good execution time and memory.	Offline **post-execution** selection and curation of efficient correct code.	No online RL reward; the closest analogues are supervised learning on fastest or lowest-memory correct samples and offline preference objectives over efficient candidates.
PIE (Shypula et al., 2024)	Dataset + SFT	Slow-fast C++ edit dataset with measured performance-improving code edits.	**Post-execution** measured edit improvement in a code-to-code setting.	Supervised and edit-pair training rather than scalar RL. Evaluation is correctness-gated speedup-like, but the model is not trained with our reward families.
PERFCODEGEN (Peng et al., 2025a)	Search	Inference-time iterative refinement using execution and profiling feedback to find faster generated code.	**Post-execution** profiling, repair, and fastest-correct selection inside an inference-time loop.	No policy-update reward; the search objective resembles correctness-gated runtime feedback.
SBLLM (Gao et al., 2025)	Search	Search-based LLM optimization with iterative candidate scoring, retrieval, and revision.	**Post-execution** execution-based ranking inside search; retrieval and prompting are search machinery rather than an RL environment operator.	No train-time scalar reward for the model policy; the search fitness is closest to correctness-gated speedup or ranking.
MAXCODE (Ou et al., 2026)	Search + value model	Frames code optimization as max-reward inference-time search with execution feedback, critique, and a reward-to-go model.	**Post-execution** iterative search over execution feedback with a frozen code-proposal policy.	Uses bucketed speedup labels for search/value guidance. This is adjacent to a correctness-gated bucketed quality signal, but it is not online RL on the code-generation policy with our scalar reward.
FUNSEARCH (Romera-Paredes et al., 2023)	Search / evolution	LLM-guided evolutionary program search for mathematical and combinatorial objectives.	Mostly outside runtime-efficiency training: candidates receive **post-execution** task-specific evaluator scores, and timeouts are guardrails.	No policy-training reward; selection uses task-specific evaluator scores rather than code-runtime reward composition.
ALPHAEVOLVE (Novikov et al., 2025)	Search / evolution	General evolutionary coding agent for scientific and algorithmic discovery, sometimes optimizing runtime or resource objectives.	Broad **post-execution** black-box evaluator. Runtime/resource cases fit **post-execution** scoring, but the paper is not limited to code-efficiency RL.	No policy-training reward; evolutionary selection may use multiple evaluator scores rather than collapsed, two-gate, or blended RL rewards.

E RL simulator details

This appendix preserves the simulator details compressed out of Section 4. The simulator is a pruning tool for environment and reward choices: it keeps the same environment computation as online RL, but replaces model generation and live execution with sampled human solutions and stored calibrated durations.

E.1 Simulator mechanics

Figure 27 shows the approximation. In online training, the model generates code, the sandbox executes it, the environment computes gates and rewards, and GRPO updates the model. In the simulator, a human solution is sampled from a quality band, stored durations replace sandbox calls, and the same environment/reward path produces a pass-rate observation. Moving the quality band from weaker to stronger reference solutions yields one simulated pass-rate curve per environment configuration.

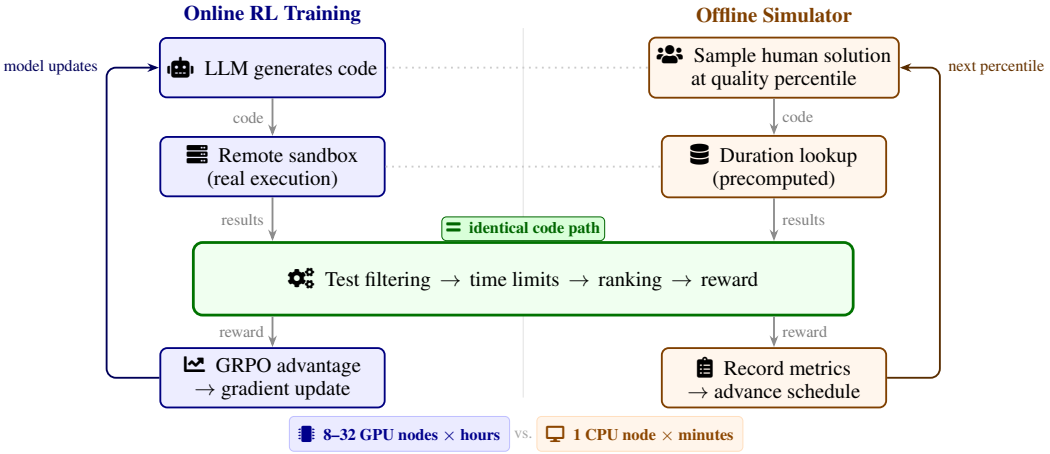


Figure 27: **Online RL training loop vs. offline simulator.** The simulator evaluates candidate environment configurations by replacing LLM inference and sandbox execution with human solution sampling and precomputed duration lookups. The environment and reward computation run the same code in both paths: tests are filtered, time limits are enforced, and solutions are ranked against human references identically. Online training uses the reward for GRPO updates; the simulator records curve diagnostics and advances the sampled quality percentile.

E.2 Curve diagnostics

Let $c_i \in [0, 1]$ be the simulated pass rate at normalized step $x_i = i/(n-1)$, and let $\tilde{c}_i$ be a smoothed version of the curve. We use AUC, $\frac{1}{n}\sum_i \tilde{c}_i$, to measure reward density; deviation from the diagonal, $\frac{1}{n}\sum_i |c_i - x_i|$, to penalize sparse and saturated settings; Spearman monotonicity between x_i and c_i to measure ordering; steepness $S = a/(1 + |a|)$, where a comes from the clamped-linear fit $\hat{c}(x) = \text{clip}(ax + b, 0, 1)$, to measure the transition slope after bounding it to $(-1, 1)$; and variance around trend to measure curve irregularity. The noise diagnostic is a curve-wobble metric, not a direct CES-noise estimate.

E.3 Comparison against online training

We compare the simulator to the final Qwen 2.5 7B environment sweep, containing 18 online runs across pre-execution, intra-execution, and post-execution families. Each run is evaluated by resimulation on the DMC-Optim test split at p_{100}, p_{80}, p_{50}, and p_{30}. The main statistic is Spearman correlation between a simulator diagnostic and the best online pass@1 reached by that run.

This comparison supports the intended use of the simulator. It is not a predictor of generic correctness learning: at p_{100}, all correlations are weak and none is significant. As the evaluation threshold becomes stricter, ordering diagnostics become predictive. By p_{30}, the strongest signals are deviation

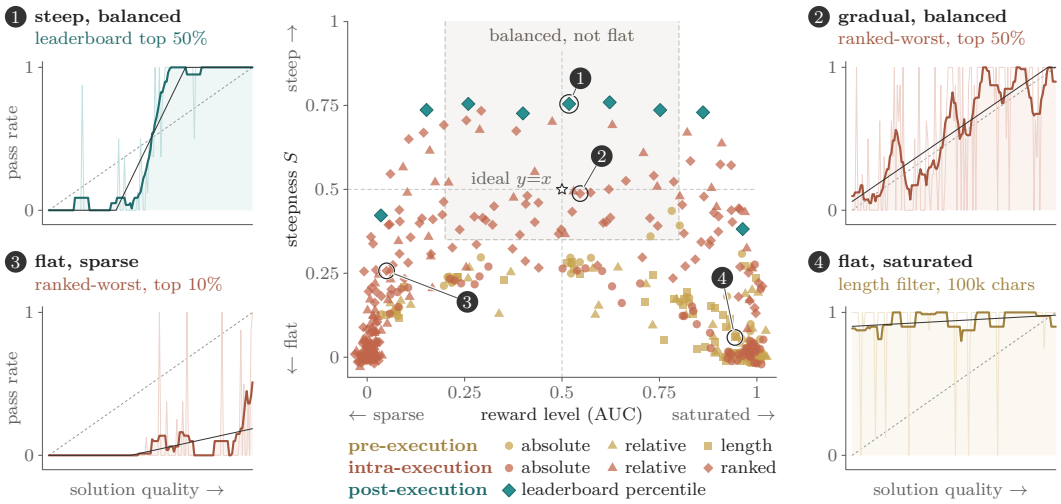


Figure 28: **Offline simulator screening of optimization-aware environments.** Each point is one environment configuration, positioned by two simulator diagnostics: AUC, the area under the simulated pass-rate curve as sampled solution quality increases, and steepness $S = a/(1 + |a|)$, the bounded slope of a clamped-linear fit to that curve. The AUC axis separates sparse from saturated settings, while steepness shows whether the transition from weak to strong sampled solutions is flat or usable. Color marks the intervention point (pre-execution filtering, intra-execution time limits, or post-execution ranking) and shape marks the subclass; insets show representative curves. In the 18-run Qwen 2.5 7B test sweep, strict p_{30} performance correlates most with ordering diagnostics: deviation from $y = x$ gives $r_s = -0.832$ ($p = 2 \times 10^{-5}$), raw monotonicity $r_s = 0.787$ ($p = 10^{-4}$), and steepness $r_s = 0.723$ ($p = 7 \times 10^{-4}$).

Table 26: **Spearman correlation between simulator diagnostics and best online pass@1 on the 18-run Qwen 2.5 7B test sweep.** The simulator is weak on pure correctness (p_{100}) but becomes more predictive as evaluation asks for stronger optimization quality. Lower deviation from $y = x$ is better, hence the negative correlations; the p columns report the two-sided Spearman p-values.

	p_{100}		p_{80}		p_{50}		p_{30}	
Metric	r_s	p	r_s	p	r_s	p	r_s	p
AUC (smoothed)	0.176	0.486	-0.197	0.433	-0.389	0.110	-0.448	0.062
AUC (raw)	0.190	0.450	-0.193	0.443	-0.383	0.117	-0.448	0.062
Monotonicity (smoothed)	0.262	0.293	0.625	0.006	0.716	0.001	0.752	0.0003
Monotonicity (raw)	0.353	0.151	0.660	0.003	0.763	0.0002	0.787	0.0001
Variance around trend	-0.337	0.172	-0.179	0.478	-0.059	0.817	-0.002	0.994
Deviation from $y = x$	-0.291	0.241	-0.652	0.003	-0.788	0.0001	-0.832	0.00002
Steepness S	0.382	0.118	0.633	0.005	0.720	0.0008	0.723	0.0007

from $y = x$ ($r_s = -0.832$, $p = 2 \times 10^{-5}$), raw monotonicity ($r_s = 0.787$, $p = 10^{-4}$), and steepness ($r_s = 0.723$, $p = 7 \times 10^{-4}$), while curve noise remains uninformative ($r_s = -0.002$, $p = 0.994$).

E.4 Matched simulator and training curves

Table 26 compares the simulator against held-out DMC-Optim test resimulation pass@1. A complementary check is to compare the simulated pass-rate curve for an environment with the online training pass rate logged while the model is trained on that same environment. Figure 29 shows this comparison for 17 Qwen 2.5 7B configurations for which both artifacts are available. This view is not the held-out evaluation used in the correlation table; it is a training-signal diagnostic that shows whether the simulator curve has the same qualitative shape as the online reward stream.

The paired curves support the same interpretation as the rank correlations. Post-execution ranking environments have high steepness ($S = 0.73$–0.75) and the clearest improving training trajectories. Saturated pre-execution filters, such as settings with AUC near 1 and steepness near 0, produce nearly flat simulator curves and provide little ordering signal during training. The ranked-worst

timeout from the top 20% setting shows the opposite failure mode: its steepness is moderate ($S = 0.41$), but its AUC is only 0.17, so the reward is too sparse and the training pass rate collapses after several thousand steps. The combined view therefore clarifies why the simulator screen uses both reward level and shape, while the downstream correlation favors ordering diagnostics.

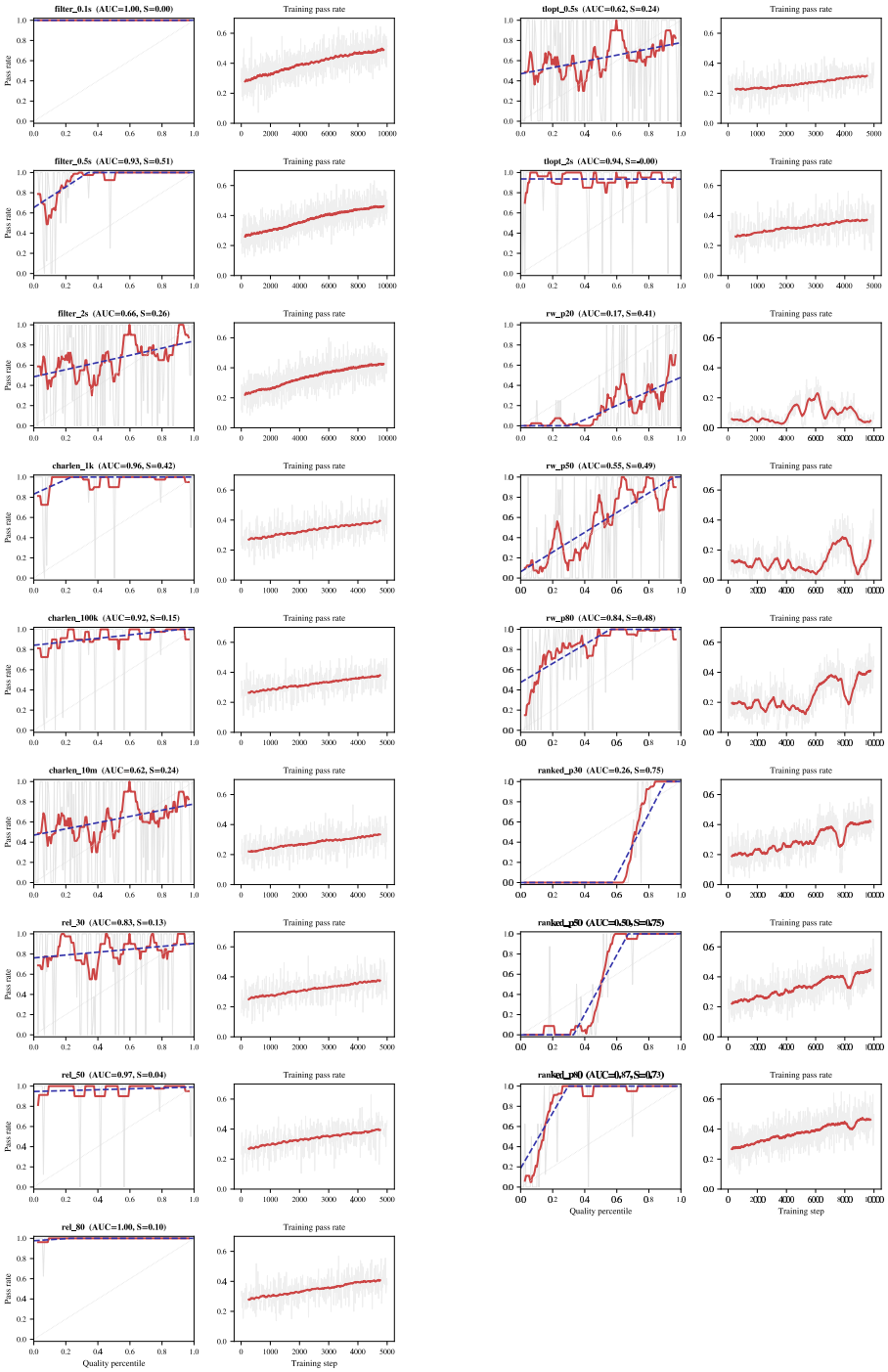


Figure 29: **Offline simulator curves compared with online training pass rates for matched Qwen 2.5 7B environments.** Each row pairs one environment configuration with its simulator curve on the left and the online training pass rate recorded on the same training environment on the right. Simulator panels show the raw pass-rate curve in gray, a smoothed curve in red, the clamped-linear fit in blue, and the $y = x$ reference in light gray; titles report simulator AUC and steepness S. Training panels show raw logged pass rates in gray and a smoothed curve in red. These online curves are not the held-out resimulation pass@1 values used in Table 26; they provide a qualitative view of the reward stream seen during training. Steep post-execution ranking curves lead to the clearest improving trajectories, saturated filters remain flat, and the low-AUC ranked-worst timeout from the top 20% case illustrates that steepness alone is insufficient when positives are too sparse.

F Training and evaluation support

This appendix keeps setup details and diagnostic displays that would otherwise crowd the main Section 5 results.

F.1 Model, training, and evaluation setup

Table 27: **Model configurations used in the optimization-RL experiments.** Two model families provide complementary starting points for RL. CWM 32B is used as a released public SFT checkpoint, while Qwen 2.5 lets us control the entire reasoning-SFT stage.

Configuration	Params	Base model	SFT provenance	SFT data note
CWM 32B	32B	CWM mid-trained	Released (FAIR CodeGen team et al., 2025)	Broader released post-training mix; authors report LCB decontamination
Qwen 2.5 32B	32B	Qwen 2.5 base	Ours	Public reasoning-only mix; decontaminated against DMC, LCB, and BigO(Bench)
Qwen 2.5 7B	7B	Qwen 2.5 base	Ours	Public reasoning-only mix; decontaminated against DMC, LCB, and BigO(Bench)

Decontamination. We use two model families across two model sizes: Qwen 2.5 (Yang et al., 2025) at 7B and 32B scale, and CWM 32B (FAIR CodeGen team et al., 2025). We choose Qwen 2.5 because the released checkpoints are pre-reasoning base models, which lets us control the reasoning trajectories shown to both sizes before optimization RL: we apply one epoch of SFT on a public reasoning-only mix built from OpenCodeReasoning-2 and OpenMathReasoning (NVIDIA, 2025; Moshkov et al., 2025), and decontaminate the SFT data against DMC, LCB, and BigO(Bench) (Li et al., 2022; Jain et al., 2024; Chambon et al., 2025). This decontamination includes the DMC-Optim test set we build and the training prompts used for RL. Therefore, during RL, the Qwen models should be reasoning about these problems for the first time (in the sense that they were not SFT on these problems, though examples could still appear somewhere in their broader internet pretraining data) and exploring possible optimizations from the problem statement only, rather than searching inside a prior supervised exposure.

Things are different for CWM 32B, which starts from a released SFT checkpoint already prepared for reasoning RL, so we do not control its SFT data. The CWM authors report decontamination against LCB, but we cannot make the same claim for DMC-Optim test or for the RL training prompts used here. Since CWM SFT uses one pass over OpenCodeReasoning, any overlapping problems likely appear with verified R1 generations rather than with the human reference solutions that define our DMC-Optim timing rankings. Their midtraining data also includes tracing data built on top of DMC problems, using Llama 3.1 70B generations traced with some of the base DMC tests. CWM 32B therefore probably did not see the ground-truth human solutions used for DMC-Optim evaluation and ranking (in the sense that we do not expect those human references to be part of its SFT or midtraining data, though examples could still appear somewhere in broader internet pretraining data), and did not receive optimization feedback. This means that RL with CWM 32B instead asks whether a model that may already know some problem statements and was exposed to sampled generated solutions, though unordered and without optimization-signal supervision, can learn to search this space with an optimization criterion, and potentially discover new optimization tricks not present in the exposed data.

Reasoning SFT. For the Qwen SFT stage, all GPUs are trainers: the 7B run uses 16 H100 nodes (128 GPUs, tensor parallelism 4) and the 32B run uses 32 H100 nodes (256 GPUs, tensor parallelism 8), with data parallelism 32 in both cases. Both use sequence length 32,768, per-data-parallel-rank batch size 2, no gradient accumulation, and learning rate 8.6×10^{-6} after a 1k-step warmup, giving $32 \times 2 \times 32{,}768 = 2{,}097{,}152$ tokens per optimizer step. Over 24,829 steps, this is 52.1B packed training tokens, corresponding to 43.5B raw unpadded tokens from the reasoning-only mix.

GRPO objective. For a prompt x, workers generate a same-prompt group of G rollouts $\{y_i\}_{i=1}^{G}$. After unpacking them into training contexts, the trainer centers each return against a prompt-level token-weighted mean:

$$\hat{A}_i = R_i - \mu_x, \qquad \mu_x = \frac{\sum_{j=1}^{G} L_j R_j}{\sum_{j=1}^{G} L_j}, \tag{40}$$

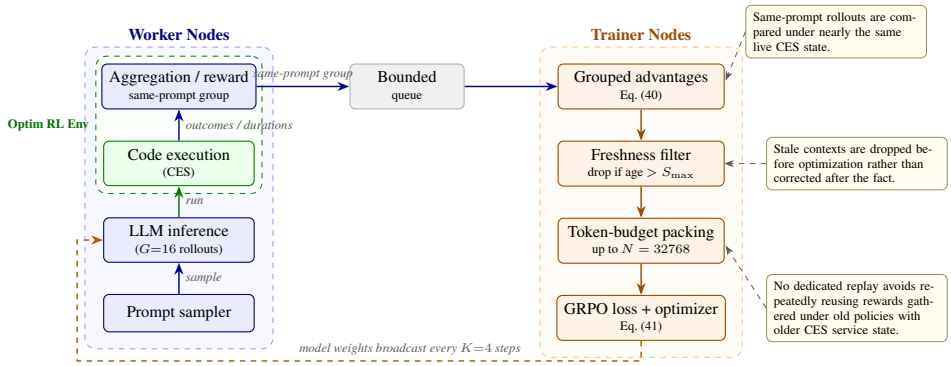


Figure 30: **Asynchronous trainer-worker architecture for efficiency RL.** Workers sample one prompt, generate $G = 16$ rollouts, execute them on CES, aggregate the resulting outcomes into a same-prompt reward group, and send it to the trainers through a bounded queue. The trainers compute grouped advantages using Eq. (40), discard contexts older than $S_{\max} = 30$ steps, pack the remaining contexts up to $N = 32768$ tokens, and apply the clipped GRPO loss of Eq. (41). Updated model weights are broadcast every $K = 4$ optimizer steps.

where L_j is the number of model-generated tokens in rollout j. We then optimize the clipped surrogate objective

$$\mathcal{J}(\theta) = \frac{1}{N} \sum_{y_i \in \mathcal{B}} \sum_{t=1}^{|y_i|} M_{i,t} \min\Big[\rho_{i,t}(\theta)\hat{A}_i,\ \mathrm{clip}\big(\rho_{i,t}(\theta), 1-\varepsilon_{\mathrm{low}}, 1+\varepsilon_{\mathrm{high}}\big)\hat{A}_i\Big], \tag{41}$$

with a fixed token horizon N rather than the realized response length.

Table 28: **Training choices that matter once the reward becomes sparse or noisy.** Several rows move in the same direction as recent math and code RL variants, but the motivation changes here because the signal comes from timed execution rather than reasoning accuracy alone.

Component	Standard GRPO	Ours	Why it helps here
Advantage scaling	Mean and standard deviation normalization	Center only, no standard-deviation division	Preserves relative ordering without amplifying small timing differences or damping large efficiency gaps
Baseline	Unweighted group mean	Token-weighted prompt mean	Aligns the sequence-level advantage with the token-level loss
Zero-signal handling	Keep all contexts	Drop zero-advantage contexts	Avoids spending updates on all-fail, all-pass, or otherwise uniform outcomes
Clipping	Symmetric $(0.2, 0.2)$	Asymmetric $(0.2, 0.25)$	Keeps some upside freedom without treating every favorable fluctuation as strong evidence
Length normalization	Normalize by realized response length	Divide by a fixed token horizon N	Keeps the update scale comparable across very different code lengths
Freshness	Usually synchronous or weakly controlled async	Async with stale cutoff and no dedicated replay	Bounds off-policy drift and avoids repeatedly reusing rewards gathered under older CES states

Table 29: **Evaluation tasks used in Section 5.** DMC-Optim is the main in-domain optimization benchmark. LCB I/O is the main out-of-distribution optimization-sensitive benchmark because it preserves dumped CES timings and avoids call-signature artifacts under optimization RL.

Benchmark	**Role**	**Primary metric**	**Dump / replay**
DMC-Optim test	In-domain optimization	pass@k over percentile sweep (p_{100}–p_{10})	Yes / Yes
LCB I/O	OOD optimization-sensitive transfer	pass@1 and speed win rates against standard RLVR	Yes / Yes

F.2 GRPO recipe ablations

The offline simulator in Section 4 shows that even after the sandbox and timing tool improvements, our environment candidates remain tied to noisy timing measurements and give sparse rewards.

Compared with correctness-only RLVR, correctness-plus-optimization rewards have fewer useful rollouts and more zero-advantage groups (when all rollouts get the same reward, either all 1 when the RL problem is too easy, or more frequently in our case all -1 when not a single rollout manages to solve it, at least partially); on a Qwen 2.5 7B run, for example, the zero-advantage-context rate at the beginning of training can be around 40–50% for binary optimization rewards like the ones we evaluate. The sandbox state can also vary under concurrent CES load, and therefore the ranges of the timing measurements that are performed can shift, so two rewards collected at different times are not perfectly comparable even if the model policy has not changed: we want to avoid giving more reward to a rollout that is in theory less optimized than another one, simply because the sandbox state was more favorable for the first rollout and made it run faster.

For these reasons, all optimization RL runs below use the same stabilized training recipe. We keep a standard async-GRPO structure (that we detail in Appendix F.1), but improve the following points:

1. We first increase the number of same-prompt rollouts on the worker side: this helps with zero-advantage problems and makes runs reach useful updates faster while dropping fewer batches of data; it also improves the quality of the Monte Carlo estimates used in the advantage computation, which is helpful when individual rewards can be noisy. This reduces the number of problems seen per global trainer batch, but decreases the wall-clock duration of the RL runs.
2. On the trainer side, we also increase the training batch size, improving pure-correctness performance in RL runs by 10–25% across different optimization RL environments, optimization performance at p_{50} by up to 35%, and optimization performance at p_{30} by up to 60%. This is consistent with the fact that batches that are too small become prone to unstable gradient estimates under timing noise, and enlarging these batches helps smooth it out.
3. Following the Dr. GRPO analysis that standard-deviation normalization can introduce a difficulty bias (Liu et al., 2025), we center returns but do not divide by the group standard deviation. This matters more for timing-based optimization rewards: we do not want to overweight prompts where the model does not find optimization tricks, or prompts where a single timing fluctuation creates an isolated reward difference, relative to prompts where rollouts expose a variety of optimization tricks and therefore carry a more useful teaching signal.
4. At the problem level, we use a token-weighted prompt mean for the advantage baseline and normalize the loss by a fixed token horizon $N = 32768$, rather than by each rollout length. The token-weighted mean keeps the global batch gradient neutral, so total positive and negative gradients cancel. The fixed horizon corrects the bias of per-rollout length normalization, which would not penalize long incorrect trajectories enough and would underreward successful deep-reasoning trajectories. This is especially important in our setting, where rewards can be very sparse and the model can get lost in indefinite incorrect reasoning. With this setting, by the end of training, models tend to generate around 9k tokens on easy problems, while hard problems average around 16k. We do not observe a particular difference between RLVR training and optimization training in reasoning length.
5. Finally, to limit stale timing comparisons, we first tune the worker/trainer ratio so generated batches are consumed promptly: we prefer mild trainer-side staleness over worker-side staleness, although this slightly hurts GPU utilization, because queued worker-side batches can become stale and be mixed with batches collected under different sandbox states. On top of this, we enforce an additional hard filtering of contexts getting too old, discarding contexts older than $S_{\max} = 30$ optimizer steps, which gives a 5% score gain. Finally, we avoid a replay buffer that would reuse rewards from older sandbox states.

F.3 Run-label abbreviations

Several figures and appendix displays use short labels because the same training configurations recur across Qwen 2.5 7B, Qwen 2.5 32B, CWM 32B, LCB transfer, train-by-eval cross-evaluation, and rerun diagnostics. Table 30 expands these labels; Appendix D.2 gives the corresponding environment definitions and reward equations.

Table 30: **Abbreviations and compact labels for model, metric, and training-result rows.** Rows define the short labels used in figure legends and appendix displays. The same label denotes the same environment/reward family across model sizes unless the table explicitly fixes a training threshold or model size.

Label	Expansion	Meaning in this paper
Q7B / Q32B / CWM	Qwen 2.5 7B / Qwen 2.5 32B / CWM 32B	Model shorthand used in compact result tables. CWM always denotes the 32B released SFT checkpoint used for RL.
$p_{100}, p_{50}, p_{30}, p_{10}$	Evaluation percentile threshold	p_{100} is the lenient correctness-style threshold; smaller percentiles require the generated solution to match stricter human-speed percentiles.
WR_b / WR_m	Best-sample / median-sample win rate	LCB speed win rate against standard RLVR: the fastest or median passing sample of each model among 20 generations is compared on each task, a model with the only passing sample wins, and draws are excluded (Appendix G.1).
RS	Random-sample win rate	Share of all sample pairs won by the trained model (A) or by standard RLVR (B), with draws reported; Δ is A minus B (Appendix G.1).
Standard RLVR	Standard RL with verifiable rewards	Correctness-only RLVR on the standard correctness-test branch; no optimization-test reward is exposed.
MC	More-correctness tests	Adds generated correctness tests to the correctness branch.
TLoc	Correctness-test time limit	Adds the 10s timeout cap on correctness tests in the MC setting.
MO	Optimization tests	Adds generated optimization tests as a binary pass gate, without a graded speed-ranking reward.
Filter abs τ	Absolute-duration filter	Pre-execution environment that keeps only optimization tests whose calibrated human-reference duration is below τ seconds.
Charlen L	Character-length filter	Pre-execution environment that filters optimization tests by input-output character count.
Rel filter r, Rel p_r, rel_r	Relative-duration filter	Pre-execution environment that drops the slowest fraction of optimization tests according to calibrated human-reference duration.
TL abs T, tlopt_T	Absolute timeout	Intra-execution environment with a fixed optimization-test timeout T.
RW p	Ranked-worst timeout	Intra-execution environment whose per-test timeout is set from the slowest reference solution among the top-$p\%$ ranked human references.
Baseline	Standard RLVR	Figure-legend label of the standard-RLVR baseline.
Abs τ, Filter τ	Absolute-duration filter	Same as Filter abs τ.
TL rel p, Rel timeout p	Relative timeout	Intra-execution environment whose per-test timeout is the p-th percentile of the calibrated reference durations on that test (Table 36).
Ranked worst p	Ranked-worst timeout	Same as RW p.
Coll. graded filter / ranked	Collapsed graded reward	Reward-composition cell on the 2 s filter or post-execution ranking environment (Appendix D.4).
QP	Leaderboard percentile, two-gate bucketed	Post-execution ranking reward that inserts the generated solution into the calibrated human-reference leaderboard and grades each correct solution by its leaderboard percentile with ten buckets.
QAR	Mean per-test percentile, two-gate bucketed	Post-execution ranking reward that grades each correct solution by its mean per-test percentile directly rather than converting it to a leaderboard position.
QP p_x, QP train p_x, Ranked p_x, Perf ranking p_x, p_{30} in training legends	Leaderboard percentile, top $x\%$	Binary post-execution training gate: a strictly correct solution passes the optimization branch only if its position is within the top $x\%$ of the leaderboard formed by the calibrated human references and the solution.
Base filtered / mixed / non-filterable	Training-data ablation labels	Base filtered is the duration-filterable training pool used for the main runs; mixed and non-filterable alter that pool while keeping evaluation fixed.
Collapsed / additive / multitask	Reward-composition ablation labels	Appendix-only labels for how correctness and optimization credit are combined before the scalar GRPO reward.
Naive base / naive opt.	Raw-duration reward controls	Appendix-only negative controls that reward unnormalized execution duration relative to either the base or optimization branch.

F.4 Compute resources for result runs

The result tables reuse the same RL checkpoints across DMC-Optim resimulation, train-by-eval replay, calibration sweeps, and LCB transfer. We count one NVIDIA H100 node as eight GPUs. Qwen 2.5 7B runs use 64 GPUs, 32 of which are trainers with DP$\times$TP $= 16\times2$, for about 24 hours, while 32B runs use 256 GPUs, 128 of which are trainers with DP$\times$TP $= 16\times8$, for about 36 hours.

Displayed rows average several reruns when available (up to eight for the baselines and the most promising configurations), and the project also used compute for preliminary and degenerate configurations, appendix-only ablation and calibration-sweep runs, CES re-executions, and simulator runs.

Prompt budget and training dynamics. All runs above, whether standard RLVR or one of the optimization-aware RL environments, train on the same underlying DMC-Optim seed set of roughly 1,000 prompts and use the same 10,000 optimizer-step budget. This is a small prompt budget compared with larger code-RL runs: the CWM 32B report uses 81,000 code-problem prompts for RL (FAIR CodeGen team et al., 2025). We therefore read these experiments as a controlled small-scale comparison rather than as a saturated training recipe. Given the effect sizes observed here, we expect larger prompt pools to help optimization RL further: the main claims are not 5% gains that might need scale to become visible, but roughly 100% relative improvements, or more, at stricter percentiles already in this limited setup.

The training dynamics point in the same direction. For Qwen 2.5 32B, the standard-RLVR baseline starts after 1k steps at 49.2% p_{100} pass@1 on DMC-Optim test and finishes at 54.8%, whereas the top-30% leaderboard-percentile run starts lower, at 46.1%, but catches up within 10k steps and reaches 55.7%. At stricter optimization thresholds, standard RLVR simply does not learn optimization capabilities: p_{50} pass@1 moves from 22.2% after 1k steps to 21.1% after 10k, and p_{30} from 10.7% to 9.4%. This suggests that pure correctness RL may even forget some occasional efficient samples inherited from the more diverse post-SFT model. By contrast, the top-30% optimization run moves from 23.4% to 39.6% at p_{50}, with the same direction at stricter percentiles. We observe the same findings with the other models.

F.5 Reward and data ablation support

The main text uses these ablations only as support for the interpretation of Table 1, rather than as additional main-paper displays. We report Qwen 2.5 7B and CWM 32B pass@1 under calibrated DMC-Optim resimulation. Trainer-hyperparameter variants are omitted here because they tune the GRPO mechanics rather than the reward/environment semantics.

The grid separates three effects. First, optimization-only rewards collapse to 0.0% pass@1 at every threshold, showing that speed without correctness does not sustain working-code generation. Second, collapsed binary rewards are the strongest cells in this grid at strict thresholds, reaching 29.2/17.4/5.7% at $p_{50}/p_{30}/p_{10}$ for the duration-filtered environment and 31.3/19.1/6.0% for the top-30% leaderboard-percentile gate. Third, additive and multitask formulations are weaker because they let the model receive some speed signal without solving the full correctness-preserving optimization problem.

Table 32 also reports CWM 32B reward-composition variants on the most successful environments. It confirms the Qwen 2.5 7B finding that collapsed binary rewards are the best balanced choice, although the CWM 32B differences are smaller and additive binary rewards become competitive in the strictest cells, while still giving up more pure-correctness score.

Raw duration hurts lenient correctness and does not recover the strict-threshold gains from reference-normalized timing. The four naive variants reach only 42.5–44.9% at p_{100}, two to four points below the more-correctness and optimization-test control, and 2.3–3.6% at p_{10}, at least 2.4 points below the best reference-normalized cells. The best naive strict score, optimization-linear at 3.6%, is only 0.9 points above that control and still well below the 2 s filter and the top-30% leaderboard-percentile run.

Table 31: Reward-composition sweep on Qwen 2.5 7B. DMC-Optim test pass@1 after 10k of RL training. Columns compare training with the 2 s pre-execution absolute filter with the post-execution ranking environment, each evaluated with a post-execution ranking environment. For the pre-execution environment, the binary optimization signal uses a 10% timeout-tolerance gate, while non-binary variants use the proportion of timeouts. For the post-execution environment, the collapsed binary reward uses the top-30% leaderboard-percentile gate, the other binary variants require a mean per-test percentile below 30, and non-binary variants use the mean per-test percentile itself. The continuous 50/50, 66/33, and 75/25 rows are two-gate ranked variants that set the reward range of correct solutions to respectively $[0, 1]$, $[0.33, 1]$, and $[0.5, 1]$, while keeping incorrect-solution reward at -1.

Training config.	**Pre-exec abs. filter 2 s**				**Post-exec ranking**			
Eval. percentile	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
COLLAPSED REWARD								
Binary	46.9	29.2	17.4	5.7	46.2	31.3	19.1	6.0
Bucketed	46.2	24.8	13.6	4.3	45.6	30.1	18.6	5.7
TWO-GATE REWARD								
Binary	45.7	26.1	14.9	4.9	47.6	29.8	17.8	5.8
Bucketed	46.6	22.1	10.6	3.4	47.1	27.5	15.5	4.9
Continuous, 50/50 split	–	–	–	–	47.6	24.3	12.7	4.2
Continuous, 66/33 split	–	–	–	–	47.1	21.9	10.7	3.3
Continuous, 75/25 split	–	–	–	–	46.4	21.4	10.4	3.2
ADDITIVE BLEND								
Binary	39.1	24.4	13.7	5.2	35.8	25.4	15.1	5.2
Bucketed	42.2	21.9	11.3	3.6	39.5	25.4	15.0	5.0
OPTIMIZATION ONLY								
Binary	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Bucketed	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MULTITASK								
Binary	38.0	22.7	12.5	4.4	30.3	21.8	13.6	4.8
Bucketed	41.2	20.5	10.2	3.4	39.8	26.7	16.0	5.5

Table 32: Reward-composition sweep on CWM 32B. DMC-Optim test pass@1 at step 10k. Columns compare the same 2 s pre-execution absolute filter and post-execution ranking environments as in Table 31. The table includes the reward-composition cells that were also run for Qwen 2.5 7B.

	Pre-exec abs. filter 2 s				**Post-exec ranking**			
Reward variant	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
COLLAPSED REWARD								
Binary	70.5	49.1	30.1	9.5	70.2	50.4	30.9	9.8
Bucketed	71.3	46.7	27.8	8.5	70.2	49.3	29.8	9.7
TWO-GATE REWARD								
Binary	70.5	47.2	28.7	9.0	70.4	48.2	29.1	9.1
Bucketed	70.8	40.0	21.7	6.8	70.1	46.6	28.0	8.6
ADDITIVE BLEND								
Binary	68.7	48.5	29.5	9.2	68.3	49.9	31.0	10.0
Bucketed	69.2	45.3	27.4	8.8	69.4	48.1	29.0	8.8

The data ablation points in the same direction as the reward ablation. Non-filterable training can dilute the strict optimization signal; for the top-30% leaderboard-percentile gate, it lowers p_{30} by 43.5% relative. Mixed-proportion training is closer to neutral, with strict-threshold changes from -17.4% to -1.1%. These controls support the main interpretation that the gain comes from exposing a correctness-preserving, problem-relative timing signal, not from adding more raw execution evidence or raw duration to RLVR.

The data ablation isolates problem selection: we compare three equal-sized training pools of 1,000 problems under the same reward and environment. The pools are the duration-filterable pool used in the main runs; a mixed-proportion pool with 478 duration-filterable and 522 non-duration-filterable

Table 33: Qwen 2.5 7B baseline RLVR and naive-duration rewards on DMC-Optim test. Rows compare standard correctness-only RLVR with base tests, with our enriched correctness tests and optimization tests, and raw-duration rewards on either base tests or optimization tests as well. Columns report pass@1 under the leaderboard-percentile post-execution environment at different percentile levels. Here, p_{100} is equivalent to correctness-only pass@1, whereas p_{50} requires solutions to be both correct and rank in the top 50% of the human reference solutions for the same problem.

	pass@1			
RL training environment	p_{100}	p_{50}	p_{30}	p_{10}
RLVR REFERENCES				
Standard RLVR	43.5	18.0	7.7	1.9
More-correctness + optimization tests	46.9	20.6	9.3	2.7
NAIVE RAW-DURATION REWARD				
Base tests, linear	44.9	18.9	8.3	2.3
Base tests, log	43.2	18.0	8.3	2.3
Optimization tests, linear	43.1	21.7	10.9	3.6
Optimization tests, log	42.5	21.2	11.1	3.5

Table 34: **Training-pool composition controls at fixed dataset size.** Entries are pass@1 at evaluation percentile p_{30} after 5k RL steps; values in parentheses are relative changes against the same environment trained on the duration-filterable pool. The comparison isolates the problem-selection step: the added-test family, reward definition, RL environment, evaluation set, and number of training problems are held fixed within each row. The character-length filter 100k row filters optimization tests above 100k serialized input-output characters and applies a 0.5 s optimization gate with 10% timeout tolerance.

Training environment	**Filterable pool**	**Mixed pool**	**Non-filterable pool**
More-correctness + optimization tests (correctness gate)	9.3	8.9 (-4.3%)	8.3 (-10.8%)
2 s absolute filter (timing gate)	10.4	9.6 (-7.7%)	9.3 (-10.6%)
Character-length filter 100k + 0.5 s gate	9.5	9.4 (-1.1%)	9.4 (-1.1%)
Top-30% leaderboard-percentile gate	16.1	13.3 (-17.4%)	9.1 (-43.5%)

problems, matching the full-corpus ratio $1{,}302/2{,}723$; and a non-filterable pool sampled from the $1{,}421$-problem complement whose optimization tests fail the duration-filterability criterion.

The decisive comparison is therefore the top-30% leaderboard-percentile reward. This reward uses the human-reference timing distribution to rank correct solutions by speed, so it needs a real within-problem duration profile. When duration-filterable problems are replaced by mixed or purely non-filterable pools, the top-30% leaderboard-percentile reward falls from 16.1% to 13.3% and then 9.1%, ending only 0.8 points above the more-correctness and optimization-test control on the same non-filterable pool. This is a bad regime for a ranking reward: if the optimization-test duration profile is nearly flat, the reward still forces a speed ordering, but that ordering is weakly supported by the tests and mostly adds noise to the reward. The conclusion is not that non-filterable problems are useless, but that duration filterability predicts where problem-relative timing rewards have a usable training signal.

F.6 Training re-run variability

The previous results show clear gains for the best optimization RL runs over standard RLVR. Nevertheless, even with our data, sandbox, RL-environment, GRPO, and evaluation contributions we described above, these trainings and evaluation tasks remain particularly prone to noise. We therefore try to estimate how much variability can appear during training and during evaluation. Table 35 reports training-time and evaluation-time 95% confidence-interval half-widths, to give an order of magnitude of the noise one can expect around the reported scores.

During training, even when we try to fix random seeds, train on the same prompt set, and preserve the same prompt ordering, the async RL setup remains strongly stochastic: rollout sampling, zero-advantage batch dropping, CES timing measures, among else, can all differ across launches. On the evaluation side, variability comes from temperature sampling, which is partly controlled by sampling many candidates and using the unbiased pass@k estimator of Chen et al. (2021), and

Table 35: Training-time and test-time uncertainty for DMC-Optim evaluation. Values are 95% confidence-interval half-widths in pass@1 points for a single training run's score. Training-time rows estimate how much a single run's score may vary, by pooling deviations across repeated launches and building a shared robust CI from the variance of deviations across all trainings. Test-time rows use a classic bootstrapping approach of the 302 DMC-Optim problems, with 1,000 resamples. For pairwise comparisons between two single-run configurations on the same test set, to know whether their difference is statistically meaningful with 95% confidence, compare the difference with the corresponding training-time CI half-width multiplied by $\sqrt{2}$ (though in practice paired differences tend to remove the $\sqrt{2}$ penalty according to Wang (2026)); test-time CIs give the additional uncertainty when the evaluation set is not fixed.

	pass@1				**pass@10**			
Source and step	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
TRAINING-TIME								
Qwen 2.5 7B								
After 5k RL steps	±1.9	±1.7	±1.3	±0.8	±1.9	±1.7	±3.1	±1.7
After 10k RL steps	±1.4	±1.4	±1.1	±0.5	±1.5	±2.4	±2.8	±1.7
CWM 32B								
After 5k RL steps	±1.1	±4.4	±4.2	±1.4	±1.6	±2.5	±5.5	±3.4
After 10k RL steps	±1.1	±1.4	±1.5	±0.8	±1.4	±1.3	±1.9	±2.5
TEST-TIME								
CWM 32B								
After 5k RL steps	±4.4	±4.2	±3.5	±2.1	±4.0	±4.6	±5.0	±4.3
After 10k RL steps	±4.3	±4.3	±3.8	±2.3	±3.9	±4.5	±5.1	±4.5

more importantly from the measured durations used to assign optimization scores and from the finite DMC-Optim test set itself. Most previous tables report averaged scores over multiple training runs and evaluations when available; here we report CIs for single training-run reported scores, so averaged scores would have less variance than these CIs, but this gives a good worst-case estimate of having a single run.

Training-time rows in Table 35 pool the within-recipe deviations of repeated launches, combining them across all training recipes to compute a single, more robust variance estimate and shared CI. Test-time rows use a classic bootstrapping approach of the 302 DMC-Optim problems: we resample problems with replacement and recompute the aggregated scores. These quantities can be used differently depending on the comparison we are trying to make. If we compare two runs on a fixed test set and fixed evaluation round, the training-time absolute half-width from Table 35 multiplied by $\sqrt{2}$ gives the half-width of the difference CI; therefore, if the two models' score difference exceeds that value, it is significant at 95%. If the checkpoints are fixed and we want to ask how they compare on a general distribution of DMC-like problems, to be able to say "checkpoint A is better than checkpoint B on general DMC-like problems", the evaluation-time interval is the relevant scale. If we want to know whether one training method is better than another in general, a conservative approach is to combine both CIs. Our goal remains to simply give an order of magnitude of the variations while staying pessimistic: we do not compute paired bootstrap intervals, nor one-sided tests that would assume in advance which run should be better.

Even under a conservative combination of training-time and test-time CIs (by simply adding the variances), the half-width is 4.7 points at p_{50}, so the p_{50} gaps of the CWM 32B top-30% run over standard RLVR and optimization-test RLVR (19.7 and 14.8 points, Section 5) remain $4.2\times$ and $3.1\times$ above it. At p_{30}, the corresponding gaps are 17.2 and 13.8 points, which is again well over significance levels. The smaller optimization-test RLVR gain over standard RLVR is more borderline under test-set uncertainty, but if we fix the test set, gains remain above training-time variability.

We also include uncertainties after only 5k RL steps. They are larger than the final 10k intervals, especially for CWM 32B at strict thresholds, suggesting that performance of RL reruns varies more significantly at intermediate training steps, though they converge toward a more consistent performance level as training proceeds. We also note the test-time variance at p_{100}: since LCB and DMC-Optim have similar test-set sizes, this variance should be of similar order on LCB, and this also mitigates the actual significance of the 4–8% pure-correctness decrease observed for some optimization RL configurations.

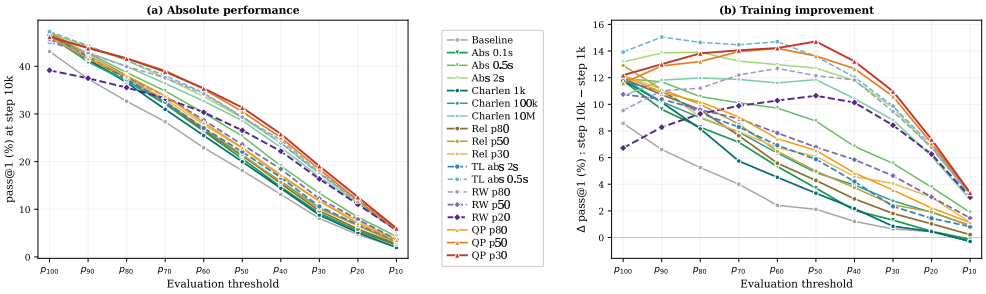


Figure 31: **Performance and training gains by evaluation threshold.** Left: pass@1 at step 10k as the human-speed requirement tightens. Right: gain from step 1k to 10k. Balanced optimization environments retain gains deeper into the percentile sweep than correctness-focused training. Legend labels follow Table 30; QP p_x is the top-x% leaderboard-percentile run.

F.7 Additional result displays

This section collects result displays that support the main text. Figure 33 breaks CWM 32B DMC-Optim pass@1 down by difficulty for the top-30% leaderboard-percentile run, which also gives the per-difficulty numbers in Section 6.

Table 36: **Relative-timeout ablation on Qwen 2.5 7B (DMC-Optim test).** Rows marked * are timeout-relative ablations stopped at 5k because they produced too many zero-advantage groups and training became too slow; gray rows are 5k references from nearby configurations, not additional ablations. Columns as in Table 1.

Configuration	pass@1				pass@10			
	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
ref. abs filter 2s @5k	42.9	21.1	10.4	3.4	65.4	48.3	31.1	11.5
ref. lead. per. 30% @5k	40.9	26.8	16.1	5.2	65.0	53.7	40.5	15.2
top 30%*	31.0	19.4	10.8	3.9	56.6	44.4	31.2	12.5
top 50%*	35.5	21.5	12.6	4.4	60.0	47.1	35.1	15.6
top 80%*	35.1	21.4	12.2	4.7	59.2	48.2	33.4	14.8

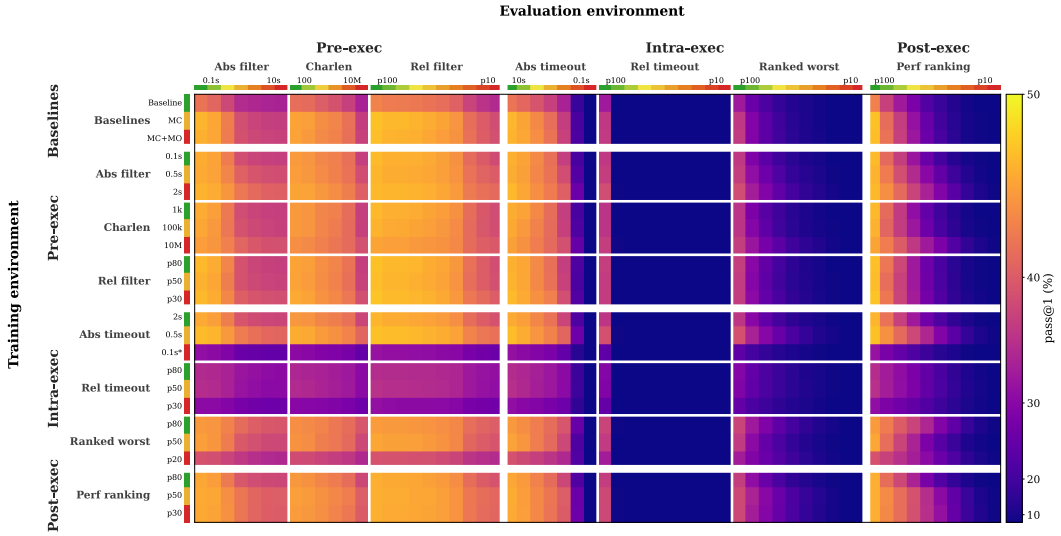
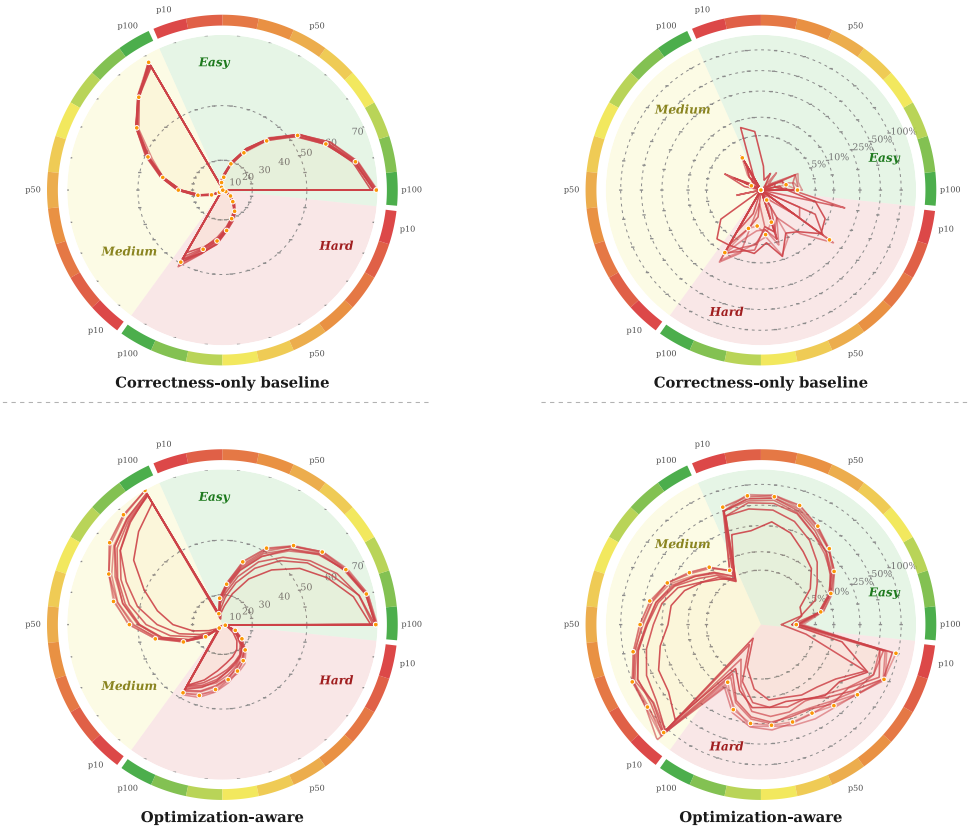


Figure 32: Train-by-eval cross-evaluation for Qwen 2.5 7B optimization RL. Rows are training environments and columns are 60 evaluation settings; each cell reports calibrated pass@1 at step 10k (4k for the 0.1 s* absolute timeout, 5k for the relative timeouts). Post-execution performance ranking best discriminates training environments, while the strongest training rows transfer broadly. Short row and column labels follow Table 30.

Table 37: Cross-model DMC-Optim scores. DMC-Optim test pass@1 under various evaluation percentiles. Columns report p_{100}, p_{50}, p_{30}, and p_{10} for each model, Qwen 2.5 7B, Qwen 2.5 32B, and CWM 32B; – indicates that a configuration was not run for that model. Bold marks the best displayed value within each model and evaluation threshold. We used group-averages when possible, and further study variability in Appendix F.6.

	Qwen 2.5 7B				**Qwen 2.5 32B**				**CWM 32B**			
Configuration	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}	p_{100}	p_{50}	p_{30}	p_{10}
STANDARD RLVR												
Standard RLVR	43.5	18.0	7.7	1.9	54.8	21.1	9.4	2.4	69.8	30.7	13.7	3.3
+ OPTIMIZATION/MORE-CORRECTNESS TESTS												
More-correctness tests	46.7	19.8	8.7	2.4	56.4	24.6	11.3	3.0	**71.5**	35.5	17.6	5.0
More-correctness tests + large 10 s timeout	46.4	19.8	8.8	2.3	59.4	26.5	12.7	3.6	71.1	34.2	15.7	4.1
More-correctness + optimization tests	46.9	20.6	9.3	2.7	**60.0**	28.7	14.2	4.4	**71.5**	35.6	17.1	4.7
+ REWARD: PRE-EXECUTION (test filtering)												
Absolute-duration filter												
$\tau = 0.5$ s	46.5	25.3	13.3	4.3	–	–	–	–	70.9	48.3	28.6	8.7
$\tau = 2$ s	46.9	29.2	17.4	5.7	48.2	31.2	19.2	6.3	70.5	49.1	30.1	9.5
+ REWARD: INTRA-EXECUTION (timeout-based)												
Ranked-worst timeout												
from top 80% ranked	45.0	29.3	17.7	**6.0**	54.4	35.4	21.3	6.9	69.9	49.7	30.6	**9.8**
+ REWARD: POST-EXECUTION (ranking-based)												
Leaderboard percentile												
Two-gate bucketed	**47.1**	27.5	15.5	5.1	59.3	37.6	22.8	7.5	71.2	49.0	29.9	9.3
top 50%	46.2	30.5	18.4	5.7	55.5	39.2	**24.2**	8.3	71.4	48.9	29.4	9.4
top 30%	46.2	**31.3**	**19.1**	**6.0**	55.7	**39.6**	**24.2**	**8.5**	70.2	**50.4**	**30.9**	**9.8**
Mean per-test percentile												
Two-gate bucketed	46.5	25.4	13.8	4.4	59.6	36.3	21.8	6.8	70.1	46.6	28.0	8.6



(a) Absolute pass@1 at each training step

(b) Relative pass@1 of each training step compared to step 1k

Figure 33: Per-difficulty CWM 32B pass@1. Top: correctness-only; bottom: top-30% leaderboard percentile; rings are training steps from 1k to 10k. The top-30% leaderboard-percentile run expands p_{50}/p_{10} across difficulty while mostly preserving p_{100}.

G Win-rate definitions and additional LCB results

This appendix defines the LCB speed win rates of Table 2 and reports the breakdowns behind Section 5. All comparisons use the 287 LCB tasks (69 easy, 77 medium, 141 hard), 20 samples per task and model, and the standard-RLVR baseline of the same model family as model B.

G.1 Definitions

The best- and median-sample win rates are not directly conditioned on passing samples, but the aggregation (best or median) favors correct samples. These metrics are more comparable to pass@20 than to pass@1, but they do suffer if the model outputs incorrect samples.

More specifically, for each problem/task, we generate 20 samples from each model and proceed as follows:

1. Best sample win rate: select the fastest correct sample from each model, among the correct samples (if available)
2. Median sample win rate: order each model's correct samples by execution time and select the median one (if available; the mean of the two middle times when their number is even)
3. If only one model has a correct sample, that model wins. If both have a correct sample, the model whose selected sample is more than 1% faster, relative to the slower time, wins. If neither model succeeds, or if their execution times differ by at most 1%, the task is a draw.

A sample is correct if it passes all tests, and its execution time is its mean per-test duration.

These win rates remove draws. And they fix the sampling budget, whatever the success rate of the model is. Therefore, if a model outputs many fast programs that are rarely correct, it can lose against a model that consistently outputs correct code, even if it is less likely to be well-optimized. Nevertheless, the aggregation includes a form of correctness oracle, so this metric corresponds to the following use case: "with an equal sampling budget, we want to know which model may work best when combined with inference-time search methods or execution feedback". Let N_A and N_B be the numbers of tasks won by A and B. The win rate of A is:

$$w_A = \frac{N_A}{N_A + N_B}.$$

The random-sample (RS) win rate covers a different use case: what happens when one generation is selected randomly from each model, without any additional search or feedback? Which model is most likely to generate a correct and fast sample in one shot? It compares every sample from model A with every sample from model B on each task. With 20 samples per model, this gives $20 \times 20 = 400$ comparisons. Every comparison uses the same rule: a correct sample beats an incorrect one; when both are correct, the sample that is more than 1% faster wins (we ablate this threshold in Appendix G.4); otherwise the comparison is a draw.

For task t, let $W^A_{t,i,j}$, $W^B_{t,i,j}$, and $D_{t,i,j}$ respectively denote the binary indicators of an A win, a B win, and a draw when comparing samples i and j. The per-task estimator of the random-sample win rate of model A is

$$\widehat{p}_{A,t} = \frac{1}{20 \times 20} \sum_{i=1}^{20} \sum_{j=1}^{20} W^A_{t,i,j}.$$

With T evaluation tasks, we macro-average across tasks. The global probability of an A win is

$$\widehat{p}_A = \frac{1}{T} \sum_{t=1}^{T} \widehat{p}_{A,t}.$$

We obtain $\widehat{p}_B$ in the same way. And finally, we explicitly compute the draw probability, as draws are not discarded for this metric:

$$\widehat{p}_{\text{draw}} = \frac{1}{20 \times 20 \times T} \sum_{t=1}^{T} \sum_{i=1}^{20} \sum_{j=1}^{20} D_{t,i,j}.$$

This way, every sample pair contributes exactly one outcome: A wins, B wins, or the pair is a draw (and the three probabilities therefore cover the full outcome space, i.e., $\widehat{p}_A + \widehat{p}_B + \widehat{p}_{\text{draw}} = 1$).

Each win is a correctness win (the winner is correct and the other sample is not) or a speed win (both are correct and the winner is more than 1% faster), and each draw is a pair where both samples fail or a speed tie. A's share of correctness wins minus B's equals the pass@1 difference between A and B, so the random-sample margin $\widehat{p}_A - \widehat{p}_B$ adds the speed wins to that correctness difference. We use bootstrapping to obtain confidence intervals: we resample the 287 tasks with replacement in 2,000 replicates (seed 0) and report 95% percentile intervals. We use paired bootstrapping on the differences: the margin is computed inside each replicate. A margin is significant when its interval excludes 0, and a best- or median-sample win rate when its interval excludes 50%; both tests are two-sided, for conservativeness. For paper rows that average several reruns, tables report the mean of the rerun point estimates, marked as significant when every rerun's interval is significant.

G.2 Random-sample win rates and significance across reruns

Table 38 decomposes the random-sample win rate of every LCB configuration with optimization tests against the standard-RLVR baseline of its model family, and Table 39 compares the random-sample margin with the best- and median-sample win rates. With the random-sample metric, because speed gains are limited on easy problems and many draws are due to both samples failing (around 40% of the pairs on CWM 32B, and up to about half on Qwen 2.5 7B), the win rates are more compressed. In every configuration, standard RLVR wins more pairs on correctness alone, and the optimization margin comes from speed wins.

Table 38: **Random-sample win rates on LCB at a 1% tie threshold.** A is the configuration and B the standard-RLVR baseline of the same model family. Wins are split into correctness wins and speed wins, and draws into pairs where both samples fail and speed ties. Rows with several reruns show the mean over reruns. The last column gives the margin Δ (A minus B), and * marks a margin whose paired 95% interval excludes 0 in every rerun. Components and Δ are computed before rounding.

Model	A win (corr. + speed)	B win (corr. + speed)	Draw (both fail + tie)	Δ
More-correctness + optimization tests				
Qwen 2.5 7B	20.6 (5.4 + 15.2)	19.9 (7.5 + 12.3)	59.6 (50.4 + 9.2)	+0.7
Qwen 2.5 32B	25.7 (5.1 + 20.6)	21.2 (5.9 + 15.3)	53.1 (42.2 + 10.9)	+4.4*
CWM 32B	26.7 (4.3 + 22.4)	22.3 (5.6 + 16.7)	51.0 (39.2 + 11.8)	+4.4*
Absolute-duration filter, $\tau = 0.5\,s$				
Qwen 2.5 7B	22.3 (5.1 + 17.2)	18.7 (7.6 + 11.1)	59.0 (50.7 + 8.3)	+3.6*
CWM 32B	29.3 (4.2 + 25.0)	20.6 (6.3 + 14.3)	50.1 (39.3 + 10.8)	+8.7*
Absolute-duration filter, $\tau = 2\,s$				
Qwen 2.5 7B	22.7 (5.3 + 17.4)	18.8 (8.4 + 10.4)	58.5 (50.5 + 8.0)	+3.9*
Qwen 2.5 32B	24.2 (3.2 + 21.0)	22.3 (10.3 + 12.0)	53.5 (44.1 + 9.4)	+1.9
CWM 32B	29.2 (4.0 + 25.2)	20.6 (6.4 + 14.2)	50.2 (39.5 + 10.6)	+8.6*
Ranked-worst timeout, from top 80% ranked				
Qwen 2.5 7B	21.3 (4.4 + 16.9)	19.2 (8.9 + 10.3)	59.4 (51.3 + 8.1)	+2.1
Qwen 2.5 32B	27.0 (4.5 + 22.6)	20.7 (7.8 + 12.9)	52.3 (42.8 + 9.5)	+6.4*
CWM 32B	29.4 (3.9 + 25.6)	20.4 (7.1 + 13.3)	50.1 (39.6 + 10.4)	+9.1*
Leaderboard percentile, two-gate bucketed				
Qwen 2.5 7B	22.6 (5.3 + 17.4)	18.5 (7.6 + 10.9)	58.8 (50.5 + 8.4)	+4.2*
Qwen 2.5 32B	28.8 (5.5 + 23.3)	19.6 (5.9 + 13.8)	51.6 (41.8 + 9.8)	+9.2*
CWM 32B	28.8 (3.8 + 25.0)	20.7 (6.2 + 14.5)	50.5 (39.8 + 10.8)	+8.1*
Leaderboard percentile, top 50%				
Qwen 2.5 7B	22.7 (5.0 + 17.7)	18.3 (8.1 + 10.2)	59.0 (50.8 + 8.2)	+4.3*
Qwen 2.5 32B	26.8 (4.4 + 22.4)	20.8 (8.3 + 12.5)	52.4 (43.0 + 9.5)	+5.9*
CWM 32B	29.3 (4.1 + 25.2)	20.5 (6.3 + 14.2)	50.1 (39.5 + 10.7)	+8.8*
Leaderboard percentile, top 30%				
Qwen 2.5 7B	22.1 (4.8 + 17.2)	18.9 (8.4 + 10.5)	59.0 (50.9 + 8.1)	+3.1*
Qwen 2.5 32B	27.4 (4.5 + 22.9)	20.4 (8.0 + 12.4)	52.2 (42.8 + 9.4)	+7.0*
CWM 32B	29.4 (4.0 + 25.5)	20.4 (6.5 + 13.8)	50.3 (39.6 + 10.6)	+9.1*
Mean per-test percentile, two-gate bucketed				
Qwen 2.5 7B	21.9 (5.2 + 16.7)	19.1 (7.8 + 11.3)	59.0 (50.5 + 8.4)	+2.8
Qwen 2.5 32B	29.0 (5.3 + 23.7)	18.9 (5.6 + 13.3)	52.1 (42.0 + 10.1)	+10.1*
CWM 32B	28.4 (3.8 + 24.7)	21.1 (6.4 + 14.6)	50.5 (39.8 + 10.7)	+7.4*

Table 39: **Comparison of the three LCB win rates at a 1% tie threshold.** RS Δ is the random-sample margin (A minus B); WR_b and WR_m are the best- and median-sample win rates. Rows with several reruns show the mean over reruns, and * marks a value whose 95% task-bootstrap interval excludes 0 (margin) or 50% (win rate) in every rerun.

Model	RS Δ	WR_b	WR_m
More-correctness + optimization tests			
Qwen 2.5 7B	+0.7	55.4	62.1
Qwen 2.5 32B	+4.4*	52.3	72.5*
CWM 32B	+4.4*	60.4*	69.3*
Absolute-duration filter, $\tau = 0.5\,s$			
Qwen 2.5 7B	+3.6*	62.2*	65.3*
CWM 32B	+8.7*	64.0*	83.0*
Absolute-duration filter, $\tau = 2\,s$			
Qwen 2.5 7B	+3.9*	63.2*	69.5*
Qwen 2.5 32B	+1.9	57.7	70.9*
CWM 32B	+8.6*	63.0	78.0*
Ranked-worst timeout, from top 80% ranked			
Qwen 2.5 7B	+2.1	59.5*	69.5*
Qwen 2.5 32B	+6.4*	57.4	76.7*
CWM 32B	+9.1*	63.7*	78.5*
Leaderboard percentile, two-gate bucketed			
Qwen 2.5 7B	+4.2*	62.7*	70.7*
Qwen 2.5 32B	+9.2*	61.4*	74.6*
CWM 32B	+8.1*	61.7*	79.3*
Leaderboard percentile, top 50%			
Qwen 2.5 7B	+4.3*	61.4*	73.0*
Qwen 2.5 32B	+5.9*	52.5	73.4*
CWM 32B	+8.8*	68.1*	80.7*
Leaderboard percentile, top 30%			
Qwen 2.5 7B	+3.1*	60.3*	68.5*
Qwen 2.5 32B	+7.0*	61.9*	72.9*
CWM 32B	+9.1*	66.3*	82.9*
Mean per-test percentile, two-gate bucketed			
Qwen 2.5 7B	+2.8	59.8	65.0*
Qwen 2.5 32B	+10.1*	66.7*	77.8*
CWM 32B	+7.4*	63.5*	76.6*

G.3 Breakdown by difficulty

On easy problems (Table 40), the gains are not necessarily significant (using two-sided tests for conservativeness), and wins mostly come from speed differences. On medium problems, every pre-, intra-, and post-execution environment significantly outperforms standard RLVR, with similar numbers of correctness-based wins on CWM 32B (medium pass@1 within 2.2 points of standard RLVR), while Qwen 2.5 7B and Qwen 2.5 32B lose up to 5.1 and 3.9 points of medium pass@1 (9.7 for the Qwen 2.5 32B 2 s filter), and with more than twice as many wins due to speed on CWM 32B and Qwen 2.5 32B. On hard problems, standard RLVR has more correctness wins than optimization RL has, but the latter still achieves more speed wins, which outweigh, in point estimates, the diminution in correctness on CWM 32B and most Qwen 2.5 32B configurations (significantly in 3 of the 20 pre-, intra-, and post-execution configurations), but not on Qwen 2.5 7B. Table 41 reports the CWM 32B pass@1 and median-sample win rate by difficulty; at a 1% tie threshold, every best- and median-sample cell rests on at least 22 decided tasks, except the median-sample easy cells (at least 10, and 8 to 9 for the three marked with a dagger).

Table 40: **Random-sample margins on LCB by difficulty at a 1% tie threshold.** * marks a margin Δ (A minus B) whose paired 95% interval excludes 0 in every rerun. For medium and hard tasks, Δ is split into net correctness wins (A's correctness wins minus B's, which equals the pass@1 difference) and net speed wins. The easy, medium, and hard subsets have 69, 77, and 141 tasks.

Model	Easy Δ	Medium Δ	Medium net corr. / speed	Hard Δ	Hard net corr. / speed
More-correctness + optimization tests					
Qwen 2.5 7B	+1.8	+3.5	−2.4 / +5.9	−1.4	−3.1 / +1.8
Qwen 2.5 32B	+1.0	+13.1*	+0.6 / +12.5	+1.4	−1.8 / +3.2
CWM 32B	+2.8	+11.2*	+0.4 / +10.8	+1.6	−2.9 / +4.5
Absolute-duration filter, $\tau = 0.5\,s$					
Qwen 2.5 7B	+5.0*	+10.5*	−3.6 / +14.1	−0.8	−3.5 / +2.7
CWM 32B	+2.9	+20.8*	−1.5 / +22.3	+4.8*	−3.4 / +8.2
Absolute-duration filter, $\tau = 2\,s$					
Qwen 2.5 7B	+3.0	+14.3*	−3.6 / +17.9	−1.4	−4.7 / +3.3
Qwen 2.5 32B	+1.9	+10.3*	−9.7 / +20.0	−2.6	−7.4 / +4.8
CWM 32B	+3.8	+21.5*	−2.0 / +23.6	+3.9	−3.8 / +7.7
Ranked-worst timeout, from top 80% ranked					
Qwen 2.5 7B	−0.4	+13.1*	−5.1 / +18.2	−2.7	−5.7 / +3.0
Qwen 2.5 32B	+3.4	+19.2*	−2.2 / +21.4	+0.8	−4.8 / +5.6
CWM 32B	+5.8*	+24.0*	−2.2 / +26.2	+2.6	−5.2 / +7.9
Leaderboard percentile, two-gate bucketed					
Qwen 2.5 7B	+3.6	+13.4*	−2.7 / +16.1	−0.7	−3.6 / +2.9
Qwen 2.5 32B	+3.2	+25.6*	+2.8 / +22.8	+3.1	−2.0 / +5.1
CWM 32B	+2.7	+22.9*	−0.9 / +23.8	+2.6	−4.4 / +7.0
Leaderboard percentile, top 50%					
Qwen 2.5 7B	+2.9	+15.1*	−3.5 / +18.6	−0.8	−4.5 / +3.7
Qwen 2.5 32B	+1.1	+19.7*	−3.8 / +23.5	+0.8	−5.1 / +5.9
CWM 32B	+3.1	+22.6*	−0.8 / +23.4	+4.1*	−4.1 / +8.2
Leaderboard percentile, top 30%					
Qwen 2.5 7B	+2.9	+12.4*	−4.3 / +16.7	−1.8	−4.9 / +3.1
Qwen 2.5 32B	+3.4	+20.7*	−3.9 / +24.6	+1.3	−4.7 / +6.0
CWM 32B	+3.7	+22.7*	−2.0 / +24.7	+4.3	−4.0 / +8.4
Mean per-test percentile, two-gate bucketed					
Qwen 2.5 7B	−0.5	+11.7*	−3.2 / +14.9	−0.5	−3.7 / +3.2
Qwen 2.5 32B	+6.9*	+22.1*	+0.4 / +21.7	+5.1*	−0.5 / +5.6
CWM 32B	+2.7	+20.7*	−1.1 / +21.8	+2.4	−4.7 / +7.1

For CWM 32B, on Easy and Medium, we do not diagnose large correctness differences. The top-50% post-execution run has the same Easy pass@1 as standard RLVR, and is within one point on Medium. Only on Hard problems do we observe a real solve-rate difference: top-50% and top-30% post-execution training reduce pass@1 from 29.8% to about 25.7–25.8%, a roughly 13% relative drop. The optimization levels are nevertheless very large, reaching 86.0% WR_m on Hard problems. We believe this degradation is partly due to LCB hard problems, which are much harder than DMC-Optim, and so the task of solving them can conflict with producing optimized code at the same

Table 41: CWM 32B LCB transfer by difficulty. Rows use the same CWM 32B configurations as Table 43. The overall column reports pass@1 on all LCB tasks. Per-difficulty columns report pass@1 and WR_m within the easy, medium, and hard subsets; WR_m uses the median passing samples of an optimization RL configuration and the baseline RLVR, and compares them to determine which one is faster. Bold marks the best displayed value within each column. † Fewer than 10 decided tasks in at least one rerun; not informative.

	Overall	Per difficulty					
		Easy		Medium		Hard	
Config.	**p@1**	**p@1**	**WR_m**	**p@1**	**WR_m**	**p@1**	**WR_m**
STANDARD RLVR							
Standard RLVR	**56.4**	98.3	–	67.7	–	**29.8**	–
+ OPTIMIZATION/MORE-CORRECTNESS TESTS							
More-correctness tests	55.3	98.1	50.0†	68.1	82.5	27.3	69.2
More-correctness tests + large 10 s timeout	54.7	**98.5**	66.7†	66.6	67.6	26.7	69.2
More-correctness + optimization tests	55.2	**98.5**	36.5	**68.2**	74.4	26.9	72.5
+ REWARD: PRE-EXECUTION (test filtering)							
Absolute-duration filter							
$\tau = 0.5$ s	54.4	98.2	63.6	66.2	**93.4**	26.4	79.4
$\tau = 2$ s	54.1	98.2	58.3	65.7	88.8	26.0	75.4
+ REWARD: INTRA-EXECUTION (timeout-based)							
Ranked-worst timeout							
from top 80% ranked	53.2	98.3	63.4	65.5	86.3	24.6	76.0
+ REWARD: POST-EXECUTION (ranking-based)							
Leaderboard percentile							
Two-gate bucketed	54.0	98.3	54.7†	66.8	91.8	25.4	75.0
top 50%	54.2	98.3	56.2	66.9	82.6	25.7	**86.0**
top 30%	53.9	98.2	**70.2**	65.8	88.7	25.8	81.4
Mean per-test percentile							
Two-gate bucketed	53.8	98.1	51.1	66.6	88.7	25.1	73.9

time. LCB can show lower pure-correctness numbers for this specific setting: the benchmark is out-of-distribution, the hard split is more difficult than DMC-Optim, and the pass@1 on this specific dataset only uses small tests with infinite timeout, which simply gives no competitive advantage to any form of optimization. Looking instead at hard-problem pass@10 after 10k RL steps, the top-50% run keeps the same 44.7 score as standard RLVR, while the top-30% run decreases by only about 3%. Maybe this is too optimistic, but we believe that this shows that further RL training should help translate more of this preserved pass@10 capacity back into pass@1, and the fact that the underlying multi-sample solving capacity is moving as much as it does under standard RLVR is a good signal (something that inference-time search methods can also exploit).

G.4 Tie threshold

At a 5% tie threshold (Table 42), the number of draws increases, and the relative win rate of optimization RL over standard RLVR also increases in most configurations, while its significance is unchanged on CWM 32B and Qwen 2.5 32B and weaker on Qwen 2.5 7B. The easy subset is omitted at 5%: in every configuration only 0 to 9 easy tasks are decided, so its win rates are not informative.

Table 42: **LCB win rates at a 5% tie threshold.** Random-sample (RS) columns are over all tasks: A / B / draw shares, the margin Δ at 5% and at 1%, and the share A/(A+B) of decided pairs won by A at 1% and 5%. WR_b and WR_m are the best- and median-sample win rates on all, medium, and hard tasks. * marks a value that is significant in every rerun. The easy subset is omitted.

	RS, all tasks				All tasks		Medium		Hard	
Model	A / B / draw	Δ at 5%	Δ at 1%	A/(A+B) 1% → 5%	WR_b	WR_m	WR_b	WR_m	WR_b	WR_m
More-correctness + optimization tests										
Qwen 2.5 7B	10.5 / 10.5 / 78.9	−0.1	+0.7	50.8 → 49.9	57.2	64.2	58.2	68.3	54.6	63.4
Qwen 2.5 32B	14.0 / 10.1 / 75.8	+3.9*	+4.4*	54.8 → 58.1	57.6	70.7*	66.7	87.5*	50.0	60.6
CWM 32B	13.4 / 10.6 / 76.0	+2.9*	+4.4*	54.5 → 55.7	70.5*	79.0*	79.3*	76.8*	63.6	81.7*
Absolute-duration filter, $\tau = 0.5\,s$										
Qwen 2.5 7B	12.2 / 10.1 / 77.7	+2.1	+3.6*	54.4 → 54.7	56.7	66.2*	66.7	79.3*	50.0	61.5
CWM 32B	16.1 / 9.7 / 74.2	+6.5*	+8.7*	58.7 → 62.5	77.4*	88.3*	83.7*	94.4*	72.9	83.4*
Absolute-duration filter, $\tau = 2\,s$										
Qwen 2.5 7B	12.8 / 10.4 / 76.8	+2.5	+3.9*	54.7 → 55.2	64.7*	76.4*	79.3*	87.5*	51.4	68.4*
Qwen 2.5 32B	12.9 / 13.0 / 74.1	−0.2	+1.9	52.0 → 49.8	61.4	70.8*	89.5*	92.6*	44.4	56.8
CWM 32B	16.2 / 9.8 / 74.0	+6.5*	+8.6*	58.6 → 62.4	72.5*	82.3*	78.7*	92.1*	68.0	74.8*
Ranked-worst timeout, from top 80% ranked										
Qwen 2.5 7B	11.6 / 10.8 / 77.6	+0.8	+2.1	52.6 → 51.8	64.3*	75.0*	69.6	83.3*	61.3	69.4*
Qwen 2.5 32B	15.6 / 10.7 / 73.7	+5.0*	+6.4*	56.6 → 59.3	64.3*	78.4*	88.9*	93.5*	50.0	65.9*
CWM 32B	16.2 / 10.2 / 73.7	+6.0*	+9.1*	59.1 → 61.4	74.5*	81.6*	81.0*	88.1*	68.7	77.2*
Leaderboard percentile, two-gate bucketed										
Qwen 2.5 7B	12.7 / 9.9 / 77.5	+2.8	+4.2*	55.0 → 56.1	63.5*	72.5*	71.2	80.2*	54.5	69.3*
Qwen 2.5 32B	16.8 / 9.4 / 73.9	+7.4*	+9.2*	59.5 → 64.1	69.5*	78.4*	85.7*	93.3*	58.3	69.0*
CWM 32B	15.8 / 9.8 / 74.4	+6.0*	+8.1*	58.2 → 61.7	69.8*	81.4*	76.0*	92.4*	65.5	74.6*
Leaderboard percentile, top 50%										
Qwen 2.5 7B	12.9 / 9.8 / 77.3	+3.1*	+4.3*	55.4 → 56.8	65.6*	79.7*	71.4*	84.8*	58.6	73.5*
Qwen 2.5 32B	15.3 / 11.1 / 73.6	+4.2*	+5.9*	56.3 → 58.0	60.7	77.1*	87.0*	93.1*	41.7	63.2
CWM 32B	16.0 / 9.6 / 74.4	+6.4*	+8.8*	58.8 → 62.5	78.3*	87.9*	79.2*	88.5*	79.4*	89.5*
Leaderboard percentile, top 30%										
Qwen 2.5 7B	12.5 / 10.3 / 77.2	+2.2	+3.1*	53.9 → 54.8	59.0	73.1*	71.4*	85.7*	43.3	62.2
Qwen 2.5 32B	16.3 / 10.6 / 73.2	+5.7*	+7.0*	57.3 → 60.6	68.8*	77.3*	91.3*	93.5*	53.8	63.4
CWM 32B	16.4 / 9.6 / 74.0	+6.8*	+9.1*	59.1 → 63.2	74.9*	85.7*	77.5*	91.3*	72.5*	81.5*
Mean per-test percentile, two-gate bucketed										
Qwen 2.5 7B	12.1 / 10.1 / 77.8	+2.0	+2.8	53.3 → 54.5	62.7	70.5*	71.2	78.5*	55.0	67.1
Qwen 2.5 32B	16.5 / 8.9 / 74.6	+7.6*	+10.1*	60.5 → 65.0	66.7*	79.5*	90.5*	93.1*	50.0	69.0*
CWM 32B	15.4 / 10.1 / 74.6	+5.3*	+7.4*	57.4 → 60.4	71.7*	81.2*	78.4*	90.0*	66.7	75.4*

G.5 Full LCB transfer table

Table 43 reports LCB pass@1 and best- and median-sample win rates for every configuration of Table 37.

Table 43: Cross-model LCB transfer. LCB pass@1 and speed win rates versus each model-family standard RLVR baseline. WR_b/WR_m use the fastest/median passing sample and exclude draws; if only one model has a correct sample, that model wins (Appendix G.1). Rows mirror the DMC-Optim best configurations in Table 37, where available; – indicates that a configuration was not run for that model. Bold marks the best displayed value within each model and metric.

	Qwen 2.5 7B			Qwen 2.5 32B			CWM 32B		
Config.	**p@1**	**WR_b**	**WR_m**	**p@1**	**WR_b**	**WR_m**	**p@1**	**WR_b**	**WR_m**
STANDARD RLVR									
Standard RLVR	**44.2**	–	–	**52.7**	–	–	**56.4**	–	–
+ OPTIMIZATION/MORE-CORRECTNESS TESTS									
More-correctness tests	42.6	51.2	57.7	50.7	51.5	68.6	55.3	62.1	73.0
More-correctness tests + large 10 s timeout	43.1	56.6	57.3	51.5	53.7	71.2	54.7	59.3	68.4
More-correctness + optimization tests	42.1	55.4	62.1	51.9	52.3	72.5	55.2	60.4	69.3
+ REWARD: PRE-EXECUTION (test filtering)									
Absolute-duration filter									
$\tau = 0.5$ s	41.7	62.2	65.3	–	–	–	54.4	64.0	**83.0**
$\tau = 2$ s	41.1	**63.2**	69.5	45.6	57.7	70.9	54.1	63.0	78.0
+ REWARD: INTRA-EXECUTION (timeout-based)									
Ranked-worst timeout									
from top 80% ranked	39.7	59.5	69.5	49.4	57.4	76.7	53.2	63.7	78.5
+ REWARD: POST-EXECUTION (ranking-based)									
Leaderboard percentile									
Two-gate bucketed	41.9	62.7	70.7	52.3	61.4	74.6	54.0	61.7	79.3
top 50%	41.1	61.4	**73.0**	48.7	52.5	73.4	54.2	**68.1**	80.7
top 30%	40.6	60.3	68.5	49.1	61.9	72.9	53.9	66.3	82.9
Mean per-test percentile									
Two-gate bucketed	41.6	59.8	65.0	52.4	**66.7**	**77.8**	53.8	63.5	76.6

H Licenses and asset provenance

This appendix records the licenses or public terms of use for the upstream assets directly used in the experiments or in the preparation of model and evaluation data. For replayed variants such as LCB I/O, we report the upstream benchmark license. DMC-Optim itself is a derived benchmark built from the upstream assets listed below together with the generated tests described in Appendix B; this appendix therefore records the upstream provenance used in this paper.

Table 44: **Licenses and terms of use for upstream assets used in this paper.** The table covers the source corpus used to build DMC-Optim, the main evaluation benchmark, the SFT datasets used for the Qwen 2.5 starts, the base and judge models, and the external benchmark used for decontamination. For replay-only variants such as LCB I/O, we list the license of the underlying upstream asset.

Asset	Use in paper	License / terms
DeepMind Code Contests (CodeContests) / Codeforces human references	Source corpus for DMC-Optim and human-speed reference solutions	CC BY 4.0 [link]
LiveCodeBench (LCB, including LCB I/O)	Out-of-distribution evaluation benchmark	MIT [link]
OpenCodeReasoning-2	Part of the Qwen 2.5 reasoning-only SFT mix	CC BY 4.0 [link]
OpenMathReasoning	Part of the Qwen 2.5 reasoning-only SFT mix	CC BY 4.0 [link]
Qwen 2.5 7B / 32B base checkpoints	Base models for the Qwen optimization-RL runs	Apache 2.0 [link]
CWM 32B	Released SFT checkpoint used for optimization RL	FAIR Noncommercial Research License [link]
GPT-OSS 120B	LLM-as-judge model for the code-pair analysis in Section 6	Apache 2.0 [link]
BigO(Bench)	External benchmark used for decontamination and motivation	CC BY-NC 4.0 [link]

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: We cover the three main points directly: point 1 in Section 3 and Appendices B and C; point 2 in Sections 4 and 5 and Appendices D to F; and point 3 in Sections 5 and 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: We discuss this in the limitations paragraph of Section 6, including the online-RL scope, the cost of the pipeline, and the fact that the data are not released. The dataset and timing scope behind those limitations is also documented in Sections 3 and 5 and Appendices B and C.

   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: We do not state theorem-style theoretical results. The equations used as method definitions or simulation setup are given where they are used, mainly in Fig. 3 and Appendices A and D to G, with each row of Table 1 defined in Table 20.

   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: We give the dataset construction and splits in Section 3 and Appendix B, the reward and environment definitions in Section 4, Fig. 3, and Appendix D.2, the timing/replay setup and calibration in Section 5 and Appendix C, and the training setup and GRPO recipe in Section 5 and Appendices F and F.2.

   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 are not releasing code or data with this paper. We do provide the details needed to reproduce the setup and main comparisons in Sections 3 to 5 and Appendices B to D and F.

   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: We give the main experimental setup in Section 5 and the supporting details in Appendices B, D and F.

   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: We report training-time 95% confidence-interval half-widths pooled over repeated RL launches and test-time intervals from 1,000 bootstrap resamples of the DMC-Optim test problems in Appendix F.6 (Table 35), and 95% task-bootstrap intervals with

paired tests for the LCB win rates in Table 2, with the resulting significance marks in Appendix G. We also study timing/sandbox variability in Fig. 20, the CES/local timing analysis in Section 3 and Appendix C, the noise simulation in Appendix A, and the ranking-metric stability checks in Appendix D.3.

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: We give the reported-run GPU counts and hours in Appendix F.4, which also states that reruns, preliminary runs, appendix-only ablation runs, re-executions, and simulator runs used additional compute. The online-vs-simulator compute contrast and shared CES evaluation setup are in Fig. 27, Section 5, and Appendix C.

   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: We checked the work against the NeurIPS Code of Ethics. The experiments study code-generation training dynamics on public datasets and benchmarks, without human-subject experiments, private data, or deceptive procedures.

   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: The work is methodological and studies how RL changes code-generation models under timing-based rewards. Apart from the broad dual-use issues that apply to code-generation models in general, we do not see a direct negative societal application specific to this paper.

    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: No release falls under this question: the paper does not make new model checkpoints, datasets, or other artifacts available.

    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: We cite the upstream datasets, benchmarks, and models where they are used. Their role in the paper and license or public terms are listed in Appendix H and Table 44.

    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: This paper does not make any new dataset, model checkpoint, or code package available.

    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: We do not run crowdsourcing studies or human-subject experiments in this work.

    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: There are no human-subject experiments in the paper, so IRB-style approval does not apply.

    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: We use GPT-OSS 120B for the LLM-as-judge study. The blinded code-pair setup, repeated rollouts, category analysis, and judge limitations are described in Section 6 and Fig. 4. A CWM 32B checkpoint generated candidate input-generator programs for DMC-Optim tests; every retained test was validated through execution against verified human solutions and the filtering pipeline in Appendix B, and an instruction-tuned model helped identify malformed, missing, or ambiguous problem statements during dataset cleaning.

    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.