Making Execution Time a Trainable Reward for Code Generation

Pierre Chambon [1 2] **Kunhao Zheng** [1 3] **Juliette Decugis** [1 2] **Benoit Sagot** [2] **Gabriel Synnaeve** [4]

Abstract

Reinforcement learning for code generation usually treats execution as a correctness verifier: a program receives reward when it passes hidden tests. Using execution time as reward is harder, because weak tests, noisy timings, and sparse rewards can make a fast-but-wrong program look attractive. We study a controlled setting where competitive-programming problems are augmented with generated large-input tests, so correct human reference solutions have enough runtime spread to support timing-based comparisons. We then turn execution time into a reward by first checking correctness and only then comparing runtimes to human references, either as a graded quality-percentile score or as a collapsed binary top-percentile event. On Qwen 2.5 7B, the best binary percentile reward improves strict speed-conditioned pass@1 from 18.1% to 27.7%, while preserving most of the pure correctness score.

1. Why Execution Time Is Hard to Reward

Training code models with executable feedback is now a standard recipe: sample a program, run it against tests, and reward it if the tests pass (Le et al., 2022; Shao et al., 2024; Guo et al., 2025; Yu et al., 2025). This recipe does not make generated code reliably fast. Efficiency benchmarks show a gap between producing a correct answer and matching strong human solutions: recent repository and competitive-programming evaluations report high correctness but much weaker speed or complexity scores (Du et al., 2024; Huang et al., 2024; Liu et al., 2024; Qiu et al., 2025; Peng et al., 2025b; Qing et al., 2025; He et al., 2025; Ma et al., 2025; Chambon et al., 2025). The missing signal is not obtained simply by adding more training data or scaling compute. For code optimization, execution must also decide whether one correct program is faster than another, and that decision can be corrupted by small inputs, backend noise, or rewards that mix speed with correctness too early.

We study the problem of turning execution time into a trainable reward for one-shot code generation. Let p_τ denote a speed-conditioned pass@1 metric: a sampled program counts only if it is correct and its runtime score lands in the top $\tau\%$ of a problem-specific human-reference leaderboard, with smaller τ imposing a stricter speed requirement. The usual correctness metric is recovered at p_{100}, while p_{20} and p_{10} ask whether the model produces solutions competitive with faster references. This separation matters because a training run can improve correctness while failing to improve strict speed, or can reward shorter runtimes while damaging correctness.

Our contribution is a training recipe for making execution time a reward. First, we augment competitive-programming problems with generated large-input tests, keeping only cases where correct human references exhibit enough runtime spread to make timing comparisons meaningful. Second, all candidate programs are evaluated in a fixed shared execution harness, so reward and evaluation use the same timed records. Third, the reward checks functional correctness before it assigns any timing score. The scope of this paper is deliberately one-shot generation: the model receives a problem and must emit a complete program before observing any runtime feedback. This differs from iterative repair or search settings, where an existing program can be profiled and edited over multiple rounds (Romera-Paredes et al., 2023; Shypula et al., 2024; Peng et al., 2025a; Gao et al., 2025; Du et al., 2025; Novikov et al., 2025).

2. Making Runtime Measurable

Execution time is a poor reward when the tests are too small. Runtime is measured inside a sandbox, so the execution backend becomes part of the signal: local sandboxing can contend with model inference and rollout orchestration, while remote sandboxing still adds dispatch and service overhead. On short tests, a fixed tens-of-milliseconds perturbation can therefore be comparable to the whole measured duration, so rankings can depend more on the execution substrate than on algorithmic or implementation differences. We therefore change the test regime before changing the

[1]FAIR at Meta [2]Inria [3]Paris Dauphine University [4]Work done while at Meta. Correspondence to: Pierre Chambon <pchambon@meta.com>.

Accepted to the Reinforcement Learning from World Feedback (RLxF) Workshop.

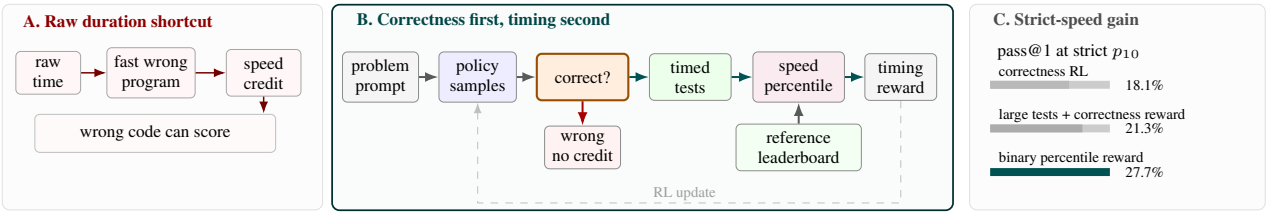


Figure 1. **Correctness must be checked before timing is scored.** *A:* Raw duration can give speed credit to fast wrong programs. *B:* The controlled reward first rejects incorrect samples; only correct samples are timed on generated large tests, inserted into problem-specific reference leaderboards, and converted into the event or score used for the policy update. *C:* Strict p_{10} rises from 18.1% under correctness RL to 21.3% with large tests and a correctness reward, then to 27.7% with the binary percentile reward.

reward: for each problem, generated large-input tests are added to the original correctness tests, following the broader use of competitive-programming tasks and generated tests in code evaluation (Li et al., 2022; Wang et al., 2025), and timing-based training is restricted to problems where verified correct human references have non-trivial runtime spread. This controlled setting asks whether a timing reward can be learned at all before moving to broader workloads.

We use the following simple noise model to make this timing-measurement issue explicit:

$$d_i^{\text{obs}} = d_i^{\text{true}}\left(1 + \epsilon_i^{\text{mult}}\right) + \epsilon_i^{\text{add}}. \tag{1}$$

The multiplicative term covers workload-dependent variation such as CPU allocation or cache behavior, while the additive term represents fixed sandbox overheads such as startup, scheduling jitter, or request dispatch. The same additive perturbation is much less damaging when the measured workload is large enough that the true duration dominates the overhead.

This model motivates the protocol used below. First, the timed workload should be large enough that fixed overhead is small relative to real execution. Second, speed should be evaluated only after correctness, because a short failed run is not evidence of efficient code. Accordingly, correctness tests remain the first step. Large-input tests are used to measure runtime only after a program has passed the correctness checks. For each problem, verified human references define per-test runtime leaderboards; model samples are inserted into those leaderboards to compute p_τ. This avoids a single global timeout as the primary score: the same absolute time budget can be too loose for one problem and too strict for another, while a top-percentile cutoff remains problem-relative.

Let $\mathcal{H}_x$ be the verified human reference pool for problem x, and let $\mathcal{T}_x$ be the generated large tests. For each test t, we insert the candidate duration into the calibrated reference-duration list, assign per-test ranks $r_{t,k}$ with rank 1 fastest and tied durations sharing a rank, and average ranks over tests to obtain $\bar{r}_k(x, y)$ for every participant $k \in \mathcal{H}_x \cup \{\star\}$, where $\star$ denotes the candidate. Let

Table 1. **Generated large tests move timing into a more usable tail.** Rows are generic test groups used in this controlled setting; statistics are pooled over tests within the retained training pool and do not expose problem counts or construction details.

Test group	Mean (s)	p_{95} (s)	p_{99} (s)	> 1s
Original tests	0.088	0.145	0.463	0.42%
Added correctness tests	0.137	0.407	1.378	1.54%
Generated large tests	0.334	1.296	3.710	6.87%

Table 2. **Controls used to make timing rewards trainable.** Each row states the control and the failure mode it prevents before the policy update sees a scalar reward.

Stage	Control
Measurement	Generate large tests; keep problems with measurable reference-runtime spread. Prevents overhead-dominated rankings.
Reward	Check correctness first; score runtime only for correct samples. Prevents fast wrong programs from earning timing reward.
Metric	Use per-problem human-reference percentiles. Avoids one global cutoff across heterogeneous tasks.
Evaluation	Replay the same timed records for every configuration. Prevents differences from changing execution samples.
Training	Use grouped rollouts per prompt. Gives the reward operator multiple candidates before each policy update.

$m_\star(x, y)$ be the zero-based position of the candidate after sorting participants by increasing $\bar{r}_k$, and let $M(x, y)$ be the number of ranked participants. The top-τ% cutoff is $b_\tau(x, y) = \max(1, \lfloor \tau M(x, y)/100 \rfloor)$. The speed-conditioned indicator is

$$p_\tau(y, x) = \mathbb{I}[y \text{ is correct } \wedge m_\star(x, y) < b_\tau(x, y)]\,. \tag{2}$$

We report averages over problems for one sampled program per problem. The case p_{100} is a lenient correctness-oriented score, while p_{20} and p_{10} require increasingly fast correct solutions.

3. Correctness-First Timing Rewards

Reliable durations are still not a reward. A scalar reward must decide how completed durations are compared and how to prevent wrong programs from receiving timing reward. We use the following interface:

$$c(x, y) \in \{0, 1\}, \qquad g(x, y) \in \{0, 1\}, \qquad q(x, y) \in [0, 1],$$

where x is the problem, y is the generated program, c records whether the program is correct, g records whether the timing branch completed without a hard execution failure, and q is a duration-derived quality score. Lower q means faster relative to references.

We focus here on post-execution ranking. After a candidate has run on the generated large tests, its per-test duration records are inserted into the human-reference leaderboards for the same problem and converted into a quality percentile. This gives a problem-relative speed scale: a solution can be fast for its own prompt without being compared to unrelated problems that have different input sizes or algorithms. The same post-execution comparison can then be used in two ways. A graded quality-percentile reward gives more credit as q improves, while a collapsed binary reward gives credit only if the correct program lands inside a fixed top-percentile region.

We compare both post-execution reward operators:

$$\begin{aligned} r_{\text{grade}}(x, y) &= \begin{cases} -1, & c(x, y)g(x, y) = 0, \\ 1 - q(x, y), & c(x, y)g(x, y) = 1, \end{cases} \\ r_{\text{bin},\rho}(x, y) &= \mathbb{I}[c(x, y) = 1 \wedge g(x, y) = 1 \\ &\qquad \wedge\, q(x, y) \leq \rho]\,. \end{aligned} \tag{3}$$

Both forms are conservative. A wrong program receives no efficiency credit even if it exits quickly; in the graded reward it receives the failure value -1. A correct program receives timing reward only after the timing branch produces a valid comparison against human-reference runtimes, at which point the graded value lies in $[0, 1]$. Naive duration rewards violate this separation: they can assign a useful-looking scalar to executions whose correctness status is ambiguous, or they can make the optimization signal too sparse when most samples fail early.

We compare four representative training configurations on Qwen 2.5 7B in Table 3. The first row is correctness-only RL. The second row adds the large tests but does not use their runtime as the main reward. The third row uses the graded human-reference comparison r_{grade}. The final row uses the collapsed binary percentile reward $r_{\text{bin},0.30}$. All rows are evaluated with the same replayed timed records, and values are pass@1 percentages.

Table 3. **Generated large tests help, but the reward still matters.** The correctness-oriented score is p_{100}; p_{20} and p_{10} impose increasingly strict top-percentile cutoffs relative to verified human references. The third row uses a graded quality-percentile reward from comparison to human references; the final row collapses the same comparison to a binary top-30% reward. The last column measures strict-speed gain over the correctness-RL baseline.

Training setup	p_{100}	p_{20}	p_{10}	Δp_{10}
Correctness RL	43.0	21.9	18.1	0.0
+ large tests, correctness reward	46.9	25.0	21.3	+3.2
+ comparison to human references	**47.3**	30.4	25.8	+7.7
+ binary percentile reward	46.1	**31.8**	**27.7**	**+9.6**

4. Training and Replayed Evaluation

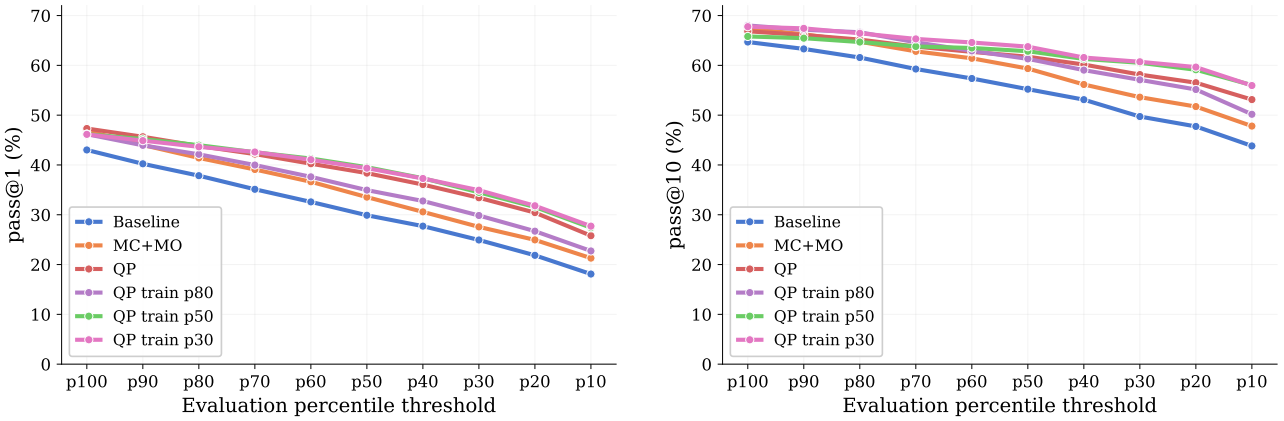
The training loop keeps the broad shape of correctness RL with executable feedback and grouped on-policy updates (Le et al., 2022; Schulman et al., 2017; Shao et al., 2024; Liu et al., 2025). For each prompt, the policy samples a group of candidate programs, the environment executes the correctness branch and then the timing branch, and the on-policy update uses the resulting scalar rewards. The grouped structure matters because timing rewards are sparse: many prompts have all samples fail correctness early, producing no useful within-prompt advantage for speed. We therefore treat zero-advantage prompt groups as uninformative for the update and keep the generation budget fixed across reward variants.

Evaluation is replayed rather than redefined per model. Each reported checkpoint samples 20 candidate programs per problem, all samples are executed in a shared timing pass, and the stored verdicts and durations are then replayed into p_{100}, p_{20}, and p_{10}. This separation lets us compare reward choices under the same timed records, while avoiding the ambiguity that a model sampled later might face a different execution state.

5. Results and Scope

The main result is that generated large tests are necessary but not sufficient. Adding them to correctness-oriented RL improves strict p_{10} pass@1 from 18.1% to 21.3%, but this still leaves most of the speed gap untouched. Post-execution comparison to human references exposes a stronger signal: the graded quality-percentile reward reaches 25.8% at p_{10}, and the collapsed binary top-30% reward reaches 27.7%. At the same time, p_{100} remains close to the large-test correctness baseline, suggesting that the gain is not obtained by sacrificing most correctness. Other binary top-percentile cutoffs follow the same pattern but are weaker under strict evaluation: top-80% reaches 22.7% at p_{10}, and top-50% reaches 27.4%.

Figure 2 shows the same pattern across the full percentile



(a) pass@1 *(b)* pass@10

Figure 2. **Post-execution reference comparisons improve the full top-percentile profile.** Each curve reports the same Qwen 2.5 7B checkpoint family as the evaluation cutoff tightens from p_{100} to p_{10}. QP denotes the graded quality-percentile reward, while QP train p_{80}, p_{50}, and p_{30} collapse the same post-execution reference comparison to binary top-percentile rewards. Relative to correctness RL, the best binary reward improves the strict endpoint from 18.1% to 27.7% at pass@1 and from 43.8% to 55.9% at pass@10. The pass@10 gain indicates that training improves the candidate set itself, not only the alignment between the first sample and the faster tail.

Table 4. **Raw duration is a weak timing reward.** Naive variants map raw seconds directly into reward, without comparing a candidate to problem-specific reference runtimes.

Training setup	p_{100}	p_{10}
Large-test correctness reward	46.9	21.3
Graded human-reference reward	**47.3**	25.8
Binary percentile reward	46.1	**27.7**
Raw duration rewards	42.4–44.8	18.9–21.7

profile. The binary p_{30} cutoff is strongest under strict evaluation, while p_{50} is nearly tied at p_{10} and the looser p_{80} cutoff loses much of the strict-speed gain. The pass@10 curves matter because they test whether the training run creates more fast correct candidates, rather than only moving probability mass toward one fast first sample.

The negative control is also informative. Naive raw-duration rewards on the same model reach only 18.9–21.7% at p_{10} and 42.4–44.8% at p_{100}, depending on the reward-range mapping. Simply adding runtime to the scalar reward therefore does not reliably improve efficient code generation. The difference is the ordering of operations: first make runtimes measurable, then reject wrong programs, and only then compare durations among correct solutions.

These rows rule out simpler explanations. The gain is not merely the result of adding more tests: relative to correctness RL, the large-test correctness row adds 3.2 points at strict p_{10}, while the binary percentile reward adds 9.6 points. The gain is also not explained by rewarding short executions in isolation, since the raw-duration variants underperform both in strict speed and in the lenient correctness-oriented score. What changes is that timing enters only after a program is correct, and the speed score is normalized by references for the same problem rather than by a global number of seconds.

The graded and binary post-execution rows are useful for different reasons. The graded quality-percentile reward keeps more ordering information from the human-reference comparison, and it is the most direct version of the idea that faster correct programs should receive more credit. The collapsed binary reward asks a stricter question: did the correct program land inside the top part of the reference-ranked distribution? The stronger strict-speed result comes from this binary version, not from a more finely graded scalar. This suggests that the main step is creating a measured, correctness-first timing channel; in this run, the best use of that channel is a sparse but clean top-percentile event.

The evaluation metric also avoids turning the result into a timeout contest. Absolute seconds are not comparable across problems: a slow solution to a tiny task can finish faster than a good solution to a large one. The reference-leaderboard metric makes the comparison local to each problem, so p_{10} asks whether the model reaches the faster tail of known correct solutions for that same prompt. This is why p_{100} remains in the table. It is not the main optimization target, but it checks that the timing reward has not simply traded correctness for speed-looking failures.

Practical implications. Runtime feedback should be treated as a measurement problem before it is treated as a reward-design problem. The domain must provide large enough inputs, a correctness oracle strong enough to reject invalid fast programs, and reference or historical executions

that define a local speed scale. Without these pieces, a timing reward may be easy to compute but not faithful. This setting remains limited to single-file competitive-programming problems and repeated timed execution, but it shows that stable measurement plus checking correctness before timing can shift a policy toward faster correct samples without collapsing the lenient correctness-oriented score.

Impact Statement

This paper studies how execution time can be used as reinforcement learning feedback for code generation. It does not include a model release or a dataset release, and therefore does not directly expand access to model capabilities or benchmark data. The work nevertheless relies on executing model-generated code; such executions should be run only inside isolated sandboxes to prevent side effects from incorrect or harmful programs. The same design choices also affect reproducibility. The timing measurement protocol is described in Section 2, the correctness-first reward operators in Section 3, and the replayed evaluation protocol in Section 4. Sections A to C give the metric aggregation, timing protocol, reward operators, and negative controls, while Section D documents the computational infrastructure and experimental setup, including the base model, training length, evaluation sampling, and coarse compute scale. Section E states the boundary conditions of the controlled setting.

Acknowledgements

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 Mathurin Videau for helpful discussions. We thank Andrew Hamiel, Don Landrum, and Eslam Elnikety for their support in using the remote execution service.

References

Chambon, P., Roziere, B., Sagot, B., and Synnaeve, G. BigO(Bench) – can LLMs generate code with controlled time and space complexity? In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Du, M., Tuan, L. A., Ji, B., Liu, Q., and Ng, S.-K. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

Du, M., Tuan, L. A., Liu, Y., Qing, Y., Huang, D., He, X., Liu, Q., Ma, Z., and Ng, S.-K. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *arXiv preprint arXiv:2505.23387*, 2025.

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

Guo, D., Zhang, H., Song, J., Wang, P., Zhu, Q., Xu, R., Zhang, R., Ma, S., Bi, X., Zhang, X., et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., Yuan, Z., and Ma, Z. SWE-Perf: Can language models optimize code performance on real-world repositories? *arXiv preprint arXiv:2507.12415*, 2025.

Huang, D., Qing, Y., Shang, W., Cui, H., and Zhang, J. EffiBench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35, 2022.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Liu, J., Xie, S., Wang, J., Wei, Y., Ding, Y., and Zhang, L. EvalPerf: Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*, 2024.

Liu, Z., Chen, C., Li, W., Qi, P., Pang, T., Du, C., Lee, W. S., and Lin, M. Understanding R1-Zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Ma, J. J., Hashemi, M., Yazdanbakhsh, A., Swersky, K., Press, O., Li, E., Reddi, V. J., and Ranganathan, P.

SWE-fficiency: Can language models optimize real-world repositories on real workloads? *arXiv preprint arXiv:2511.06090*, 2025.

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

Peng, Y., Gotmare, A. D., Lyu, M. R., Xiong, C., Savarese, S., and Sahoo, D. PerfCodeGen: Improving performance of LLM generated code with execution feedback. In *IEEE/ACM International Conference on AI Foundation Models and Software Engineering*, pp. 1–13, 2025a. doi: 10.1109/Forge66646.2025.00008.

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

Qing, Y., Zhu, B., Du, M., Guo, Z., Zhuo, T. Y., Zhang, Q., Zhang, J. M., Cui, H., Yiu, S.-M., Huang, D., Ng, S.-K., and Tuan, L. A. EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code. *arXiv preprint arXiv:2505.13004*, 2025.

Qiu, R., Zeng, W. W., Ezick, J., Lott, C., and Tong, H. How efficient is LLM-generated code? A rigorous & high-standard benchmark. *International Conference on Learning Representations*, 2025. arXiv:2406.06647.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2023. doi: 10.1038/s41586-023-06924-6.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., and Yazdanbakhsh, A. Learning performance-improving code edits. *International Conference on Learning Representations*, 2024. arXiv:2302.07867.

Wang, Z., Liu, S., Sun, Y., Li, H., and Shen, K. CodeContests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025.

Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

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

A. Metric and Aggregation Details

The main text reports pass@1 under problem-relative top-percentile cutoffs. This appendix gives the exact leaderboard aggregation used for those scores and for the negative controls. For each problem x, let $\mathcal{H}_x$ be the verified human reference pool and $\mathcal{T}_x$ the generated large tests. Each test $t \in \mathcal{T}_x$ has a calibrated duration list from the historical human references. For a sampled program y, we first append its duration on test t to that list and sort all participants by duration, fastest first. The per-test rank $r_{t,k}$ is one for the fastest participant, and ties receive the same rank. Ranks are then averaged across tests:

$$\bar{r}_k(x, y) = \frac{1}{|\mathcal{T}_x|} \sum_{t \in \mathcal{T}_x} r_{t,k}. \tag{4}$$

Sorting $\mathcal{H}_x \cup \{\star\}$ by increasing $\bar{r}_k$ gives the final problem-level leaderboard, where $\star$ denotes the generated program. Let $m_\star(x, y)$ be the zero-based position of $\star$ in this sorted list, and let $M(x, y)$ be the number of ranked participants. Let $b_\tau(x, y) = \max(1, \lfloor \tau M(x, y)/100 \rfloor)$ be the top-$\tau\%$ cutoff. For a sampled program y, the speed-conditioned correctness indicator is

$$p_\tau(y, x) = \mathbb{I}[y \text{ passes correctness tests} \wedge m_\star(x, y) < b_\tau(x, y)]\,. \tag{5}$$

The reported pass@1 value averages $\mathbb{E}_x[p_\tau(y_1, x)]$, where y_1 is the first sampled program for problem x. The evaluation run stores 20 candidate programs per problem, but the main tables use pass@1 to keep the result tied to one-shot generation rather than search. When a program fails correctness, its timing branch cannot make it count for any p_τ. When timing execution fails for reasons that prevent a leaderboard comparison, the sample is treated as not satisfying the speed-conditioned indicator.

The interpretation of τ is reversed relative to a timeout budget: smaller τ means the program must land higher in the human-reference leaderboard. Thus p_{100} is lenient and mostly checks correctness preservation, p_{20} is a moderate speed cutoff, and p_{10} is the strict metric used to test whether training increases the share of fast correct solutions. The Δp_{10} column in Table 3 subtracts the correctness-RL row, so it reports the total strict-speed gain over the standard executable-feedback baseline.

Table 5. **Roles of the three cutoffs reported in the paper.** The same stored verdicts and durations are replayed into each top-percentile cutoff.

Metric	What it asks	Main use
p_{100}	Is the program correct and rankable on the large tests?	Correctness preservation
p_{20}	Is the correct program in the top fifth of the reference leaderboard?	Moderate speed
p_{10}	Is the correct program in the top tenth of the reference leaderboard?	Strict speed

B. Timing Protocol

The timing protocol separates two questions that correctness-only RL often merges. First, a program must pass correctness tests that are intended to reject wrong behavior. Second, only after correctness is established, generated large tests are used to compare runtime against verified human references. This ordering prevents a short failed execution from being confused with an efficient solution.

The noise model in Equation (1) abstracts the sandbox effects that motivate the large-test choice. If d^{true} is small, the relative additive burden $|\epsilon^{\text{add}}|/d^{\text{true}}$ can be large enough to reorder correct programs. Increasing the workload does not remove timing noise, but it makes the useful duration signal larger relative to fixed overheads. The pooled timing statistics in Table 1 show this intended shift: the median regime remains short, but the upper tail grows enough that reference solutions can be separated by runtime on a non-trivial part of the suite.

This paper treats test generation as a measurement device rather than as the main contribution. For the reward question studied here, the important fact is that generated large tests create a controlled timing regime in which correct reference programs can be ranked. The exact prompt templates, filtering thresholds, and larger validation suite are separate engineering details.

Table 6. **Measurement requirements before using runtime as reward.** Each requirement is checked or enforced before the scalar reward in Equation (3) is computed.

Requirement	Reason	Consequence in this work
Correctness first	Runtime of wrong code is not meaningful speed evidence	Timing credit is unavailable unless $c = 1$
Problem-relative leaderboards	Absolute seconds are not comparable across tasks	Scores use human-reference ranks
Shared timed records	Re-running each ablation can change the timing state	Metrics replay stored verdicts and durations
Runtime spread within problem	A timing reward needs more than one indistinguishable cluster	Training uses problems where references separate by duration

C. Reward Operators

The reward interface separates execution into three objects. The correctness indicator c is one only when a program passes the standard correctness tests. The timing-completion indicator g is one only when the program completes the large-test timing branch without a hard execution failure. The quality score q compares the program's aggregated per-test ranks to the verified human reference leaderboard for the same problem. The reward operators in Equation (3) use this quality score only after both conditions are satisfied.

The post-execution comparison can be written abstractly as

$$\{(s_t(y), \delta_t(y))\}_{t \in T_{\text{large}}(x)} = E\big(y, T_{\text{large}}(x)\big), \tag{6}$$

$$q(x, y) = Q_{\text{post}}\big(\{(s_t(y), \delta_t(y))\}_{t \in T_{\text{large}}(x)}, \mathcal{H}_x\big). \tag{7}$$

E records statuses and durations on generated large tests, and Q_{post} inserts the candidate into the human-reference ranking for the same problem. The main result compares the graded quality-percentile version of this comparison to a collapsed binary top-30% version.

Table 7. **Reward families considered in the controlled setting.** Rows differ in whether correctness is checked before speed is scored.

Family	Reward source	Main failure mode
Correctness only	Binary pass/fail on correctness tests	No direct incentive to be fast
Raw duration	Timing branch without problem-relative references	Fast wrong programs or easy short workloads can receive useful-looking reward
Graded reference comparison	Check correctness and timing completion, then assign a quality-percentile speed score	Preserves ordering information but can be noisier as a scalar reward
Binary percentile reward	Check correctness, then assign a top-ρ reference-ranked event	Can be sparse when few samples are both correct and fast

The naive raw-duration baselines instantiate the speed-only intuition with raw seconds. Let $\bar{d}_s(x, y)$ be the mean observed duration on source s, where s is either the original correctness-style tests or the generated large tests. Two maps are used:

$$m_{\text{lin}}(\bar{d}_s) = \text{clip}\left(1 - \frac{\bar{d}_s}{10}, 0, 1\right), \tag{8}$$

$$m_{\log}(\bar{d}_s) = \text{clip}\left(\frac{1 - \log_{10}(\max\{\bar{d}_s, 10^{-6}\})}{3}, 0, 1\right). \tag{9}$$

The raw-duration reward is then $R_{\text{raw}} = c + m(\bar{d}_s) - 1$. This means an incorrect but very fast program can be less strongly penalized than a slower incorrect program, which is why raw duration is used only as a negative control.

Table 8. **Naive raw-duration variants used as negative controls.** The result range in Table 4 aggregates these four rows.

Variant	Duration source	Map	p_{100}	p_{10}
Base-linear	Original tests	Linear	44.8	20.0
Base-log	Original tests	Log	43.1	18.9
Large-test-linear	Generated large tests	Linear	43.0	21.7
Large-test-log	Generated large tests	Log	42.4	20.1

D. Training and Replayed Evaluation Details

The experiment in Table 3 starts from Qwen 2.5 7B (Yang et al., 2025). Each evaluation samples 20 candidate programs per problem and replays the stored verdicts and durations into the p_{100}, p_{20}, and p_{10} metrics. The RL runs use the same broad recipe as correctness RL with executable feedback: grouped rollouts per prompt, on-policy updates, and a fixed generation budget. We report the 10k-step checkpoint and use a coarse compute scale of several dozen H100 GPUs for about one day.

Table 9. **Training and evaluation choices kept fixed across the main comparison.** The table lists only the coarse choices needed to interpret Table 3.

Component	Setting
Base model	Qwen 2.5 7B
Training length	10k optimizer steps
Sampling	Grouped same-prompt rollouts
Evaluation samples	20 candidate programs per problem
Reported score	pass@1 replayed from stored verdicts and durations
Primary cutoffs	p_{100}, p_{20}, p_{10}
Compute scale	Several dozen H100 GPUs for about one day

The grouped rollout structure is retained because the reward is sparse in two ways. Some prompt groups contain no correct samples, so every candidate receives the same negative outcome and the group carries no useful within-prompt ordering. Other groups contain correct samples, but only a subset complete the timing branch and receive a duration-derived score. Keeping the prompt group fixed across reward variants makes the comparison in Table 3 about the reward signal rather than about a different sampling budget.

The replayed evaluation separates generation from metric definition. A checkpoint first produces candidate programs. Those programs are executed once under the shared timing protocol, creating stored statuses and durations. Only then are the records replayed into the lenient and strict top-percentile cutoffs. This is why Table 3 can compare correctness-only, graded human-reference, and binary percentile rows under the same evaluation records, even though the rewards used during training differ.

Run-to-run variability remains a limitation of this kind of experiment. In repeated launches from the same model family, strict p_{10} scores move more than p_{100} scores because they depend on both solving the problem and landing high in the reference leaderboard. In repeated 7B launches, strict-cutoff spreads were a few percentage points, smaller than the 9.6-point p_{10} gap between correctness RL and the best binary percentile reward in Table 3. This does not remove the need for broader repetition, but it suggests that the main effect is larger than the observed launch-to-launch noise in this controlled setting.

E. Scope and Boundary Conditions

The controlled result supports the claim that execution time can become a learnable reward when tests create measurable runtime spread and speed is scored only after correctness. It does not establish that the same recipe generalizes unchanged to every code domain, model family, or execution environment. We separate those boundary conditions explicitly because runtime rewards are unusually sensitive to the measurement protocol.

Table 10. **What the controlled result does and does not establish.**

Supported by this controlled result	Still outside this controlled setting
Generated large tests can make timing comparisons more informative than the original short-test regime.	Full benchmark construction and filtering details.
Correctness-first timing rewards improve strict speed-conditioned pass@1 for Qwen 2.5 7B.	Broader model families and out-of-domain validation.
Raw duration is not enough to recover the strict-speed gain.	Detailed execution-environment validation.
Replayed timed records make reward ablations comparable under one evaluation protocol.	Broader environment selection and qualitative behavior analysis.

The task family is controlled: single-file competitive-programming problems, generated large tests, and reference-relative speed cutoffs. This setup isolates the reward question, but it does not cover repository-scale workloads, multi-file performance engineering, or cases where speed depends on deployment-specific hardware. The reward is likewise not a cheap static oracle; it requires executing candidate programs and comparing them to reference timings.

The intended takeaway is direct. Correctness-only executable feedback leaves efficient-code generation under-specified. Adding generated large tests helps, but does not close the strict-speed gap. The stronger improvement comes from making timing measurable, rejecting wrong programs before speed is scored, and comparing correct programs relative to problem-specific human references. That is the piece this paper contributes, while leaving broader validation as future empirical work.