COUNT THE CONSTRAINTS, NOT THE TESTS: VERIFIER DIVERSITY GOVERNS REINFORCEMENT LEARNING FROM EXECUTION FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning from verifiable rewards (RLVR) trains a code policy against a test suite, so what it can learn is bounded by the distinctions that suite draws. Practice summarises a suite by the number of tests it holds. If no sampled program separates two tests, they state the same constraint, and a reward computed from twenty of them still takes only two values. We measure the distinctions instead, grouping a suite's tests by the pass and fail column they produce over twenty programs from a fixed checkpoint, and counting the distinct columns, the effective constraint count. The gap is wide in practice: among generations that pass at least one test of a five test suite, three quarters pass all five, a share that barely moves across three different models. We then hold the tasks and the test count fixed and vary only whether the three tests the reward reads come from three clusters or from one, over $9{,}535$ tasks with $1{,}907$ held out and three seeds per arm. The pass@1 rises from 7.1% to 10.0%, with every diverse seed above every redundant seed, while pass@20 moves by 0.7 points, so the effect is on reliability, not reach. The count needs only samples from a checkpoint that already exists, so a suite can be judged before compute is spent on it. Count the constraints, not the tests.

1 INTRODUCTION

Reinforcement learning from verifiable rewards is the standard way to post-train models for reasoning and coding (Shao et al., 2024; Guo et al., 2025). When a policy is trained against a verifier, it can learn only what that verifier can tell apart. In code generation, the verifier is a test suite, and the reward depends on which tests pass, which makes the suite the place to ask what the verifier itself contributes. Previous work in the area selects the training tests by their number, by the coverage they reach or by their source. In every case the suite is judged against a reference fixed before training, so how many independent constraints the reward actually reads is left open.

In our pipeline a quantised 7B teacher writes the test specifications with the derived task descriptions, and a deterministic renderer turns each specification into a JUnit suite. An implementation is written by the teacher, but only to prove the satisfiability of the suite, and is then discarded. A 1.5B student model trains against the generated suites without seeing a solution, so the supervision delivers a specification and a grader instead of an answer. The student reaches a 66.6% pass@1 against the teacher's 29.1%, matching it at 77.6% against 76.4% on pass@20 (Table 2). If the teacher's ability to solve tasks were the binding constraint, the student could not outperform it. The student inherits the teacher's capability and exceeds its reliability, making the suite the bottleneck.

The intuitive initial approach to measure how much a suite constrains an implementation is the number of tests it contains. We started with this approach, which proved to be a weak measure, since the tests can repeat one another: five tests can state the same constraints five times but the last four carry no new information. Therefore, we measure the constraints instead of tests: sampling generations from a fixed checkpoint and recording which tests each one passes, and every task yields a binary matrix. Two tests whose columns match cannot separate any two of those generations, so the constraint is written twice. Thus, the number of distinct columns is the effective constraint count. Measured this way, the original suites, with five tests per task, behave as a single constraint, so we

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

rebuilt the generator to produce twenty tests per task over classes whose methods branch. Across the 27,139 redesigned suites the median suite imposes three constraints, and only 40.6% of suites show four or more distinct codewords (Section 4). The measurement requires only the samples from a previous checkpoint, so it runs before any training compute is spent.

Establishing a correlation is not enough since tasks with many clusters differ from tasks with one in ways out of our control: difficulty and empty suites (Section 5). Therefore, we vary the diversity inside each task instead. Both arms train on three tests: in the diverse arm the tests come from three different clusters; in the redundant arm all the tests come from the same cluster; and one test is shared between both, differing only in whether the other two probe anything new. With the same tasks, warm start, hyperparameters and step counts, and three seeds per arm, pass@1 on 1,907 held out tasks rises from 7.1% to 10.0% against a start of 4.1%, with every diverse seed above every redundant seed, while pass@20 moves by 0.7 points. Diverse constraints allow partial credit, reducing the share of GRPO groups with no advantage from 50.8% to 43.4%, making the reward carry a gradient more often.

Contributions:

1. A verification only setting in which a 1.5B student trained exclusively against a 7B teacher's tests surpasses the teacher at pass@1 and matches it at pass@20, which leaves the suite, and not the teacher's ability to solve, as the binding component of the pipeline.
2. A measurement of how many independent constraints a suite imposes on a given policy, obtained by grouping its tests by the pass and fail column they produce over that policy's own samples. The relation is the indistinguishability of Ammann et al. (2014) computed against sampled programs instead of mutants, and with twenty tests per suite the median is three.
3. A controlled experiment on identical tasks with the same number of training tests, in which drawing those tests from distinct constraint clusters raises pass@1 by 2.9 points, with every diverse seed above every redundant seed, while pass@20 moves by 0.7 points.
4. An account of the mechanism, where tests from one cluster make the test reward binary and tie the GRPO groups, and partial credit lowers the share of groups with zero advantage from 50.8% to 43.4%.

2 Background and related work

The main objective of execution feedback is to convert execution results into usable gradients. Previous work tries to achieve this by credit assignment, learning a critic over unit test results (Le et al., 2022) and focusing on why the test fails instead of whether the test fails (Liu et al., 2023). Later work focused on the feedback channel, adding compiler errors, shortening the exploration problem with a curriculum of code completion subtasks and crediting only the lines the tests actually execute (Dou et al., 2024), and providing results across turns so the policy can repair (Gehring et al., 2025). GRPO later removed the critic and made a rule-based pass or fail reward the dominant formulation (Shao et al., 2024; Guo et al., 2025). The verifier is the one constant, human authored, fixed, and assumed adequate. In this work, we leave the optimizer as it is, focusing on varying the suite, since what a policy can learn is bounded by the differences its suites can make.

Synthetic data generation emerges to remove the need for human training data. Prior work establishes execution as a sufficient filter, using generated tests to accept instruction data, to verify the solution and test triples and to rank candidate programs (Wei et al., 2024b;a; Xu et al., 2025; Chen et al., 2023).

Limited suites inflate measured performance and distort the reward under RLVR (Ma et al., 2025), verifier precision and recall change what the trained model learns (He et al., 2025). The tests can be more difficult, with less redundancy, by evolution (Ruan et al., 2026) or by synthesis against near correct faulty code, where candidate tests are clustered by their behaviour on those programs before representatives are kept (Zhang et al., 2026). Zhang (2026) developed the closest experiment to ours and reported that a hardened suite reduces the inflation but gets little capability at that scale. The key differentiation is which tests a suite contains. Where behaviour is used to group, it is the programs that are grouped (Li et al., 2022; Chen et al., 2023), and where independent constraints are counted,

they are counted by matrix rank against a corpus of wrong submissions fixed in advance (Luo et al., 2026). In this work, we group the tests instead, and we hold the number of tests fixed while varying how many constraints they impose.

The main concern about RLVR is whether it adds capability or reallocates what a model already has. Prior work indicates that pass@1 rises while pass@k at large k does not (Yue et al., 2025). Shao et al. (2025) show that, on some base models, gains survive rewards that carry no information about correctness, which breaks the inference from a better score to a better model. The boundary is contested, since longer training does move pass@k (Liu et al., 2025); our arms are matched on compute, focusing on comparing the ceiling at a fixed budget.

In the software engineering area, our question is called test suite adequacy. Multiple answers appeared through time such as the usage of coverage criteria (Zhu et al., 1997), mutation score as proxy for fault detection (DeMillo et al., 1978), and the removal of tests that covered nothing the remainder already covered (Harrold et al., 1993). Furthermore, the same literature establishes the limits of its own proxies, since coverage correlates weakly with effectiveness (Inozemtseva & Holmes, 2014) and redundancy among mutants increases the score a technique appears to earn (Papadakis et al., 2016). Ammann et al. (2014) call two tests *indistinguished* when they kill precisely the same mutants, which is the equivalence relation we use. In this work, the reference set is the samples of the policy being trained, and the partition is what we measure and then vary at a fixed test count.

GRPO stops learning when the samples in a group score alike. Since advantages are standardised within a group, such a group yields no gradient. The degeneracy is measured in 28 to 45% of training batches across five model scales on mathematical reasoning (He et al., 2026), and it sharpens as the group shrinks, achieving 0.69 at four samples (Nie et al., 2026). Prior work answers at the level of the algorithm by discarding the groups that carry no contrast and oversampling to replace them (Yu et al., 2025), or adding a continuous term to split the same outcome groups (Deng et al., 2026).

Li et al. (2026) state that pass rate shaping does not reliably improve the final performance, and Wang et al. (2026) show that weighting by a number of tests passed biases the update toward easiest tests. Therefore, a denser test reward is not by itself an answer. With this work, we show the degeneracy also depends on the verifier, since constraints that no sample separates make the test reward binary. Section 6 measures the gap between our arms.

3 SETTING

3.1 VERIFICATION ONLY TRAINING

The 7B parameter teacher model is quantised to 4-bits and generates tasks and TestSpecs in JSON. This acts like a DSL and is then deterministically rendered into a valid JUnit suite to eliminate compilation errors. Furthermore, to ensure that the tests are possible to pass, the task is only kept if an implementation written by the teacher compiles and passes every test. The implementation generated is then discarded and only the suite survives.

Based on those suites, we derive the description from the required implementation API signature and that is used to train the student, `Qwen2.5-Coder-1.5B-Instruct`, with GRPO and a LoRA adapter. For GRPO, reward is given by

$$r = 0.7 \cdot r_{\text{pass}} + 0.3 \cdot r_{\text{style}}, \tag{1}$$

where r_{pass} is the fraction of the k training tests the candidate passes, r_{style} is a style score in $[0, 1]$ from PMD, and a candidate that does not compile receives $r = -1$ instead.

π_0 is the student checkpoint obtained after training on the original suites from Section 4 and every run in Section 5 starts from it. This initial policy has a pass@1 of 4.1% and a compile rate of 76.9% using the compilation regime of Section 4.

3.2 CONSTRAINTS, SIGNATURES AND EFFECTIVE CONSTRAINT COUNT

Consider a task T with tests $t_1, \ldots, t_m$. Sample twenty programs for it from the policy π_0 before training, keep those that compile, and record which tests each one passes in a signature matrix S

where rows correspond to the programs and columns are tests (Figure 1, right). For example, if task T is "invert the input string", three of the twenty programs compile and the suite has four tests, then

$$S = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}, \tag{2}$$

where program 1 passes t_1 and t_4 and fails t_2 and t_3.

Two tests with equal columns are redundant, since no sampled program separates them, and they form a cluster, the indistinguished tests of Ammann et al. (2014) with sampled programs in place of mutants. We define the effective constraint count c of a set of tests as its number of distinct columns, and a codeword as a distinct row. In the example t_1 and t_4 share the column $(1, 1, 1)$, so the suite has four tests but $c = 3$, and programs 1 and 3 share a row, so there are two codewords. Adding a program can split a cluster but never merge two, so c can only grow with the sample and every measured value is a lower bound on what a larger sample would give. The count matters because training does not read the whole suite. It reads a subset T' of k tests per task, with k fixed in advance, and the test term of the pass reward of program i is given by

$$r_{\text{pass}} = \frac{1}{k} \sum_{t_j \in T'} S_{ij}, \tag{3}$$

the fraction of the k tests it passes (the full training reward adds a compilation penalty and a style term, Section 3.1). If the k tests of T' all lie in one cluster, every program passes all of them or none, so r_{pass} is 0 or 1. If instead each of the k tests comes from a different cluster, r_{pass} can take any of the $k + 1$ values $(0, \frac{1}{k}, \ldots, 1)$. In the example, take $k = 2$. With $T' = \{t_1, t_4\}$, both tests in the same cluster, programs 1, 2 and 3 each pass both tests. So r_{pass} is 1 and the three programs are indistinguishable to the reward. With $T' = \{t_1, t_2\}$, one test from each of two clusters, programs 1, 2 and 3 have corresponding r_{pass} of $\frac{1}{2}$, 1 and $\frac{1}{2}$ and the reward now separates program 2 from the other two. Same k, same task, same programs, and only the second choice of T' carries information. The resolution of r_{pass} is set by the number of clusters among the k tests, not by k.

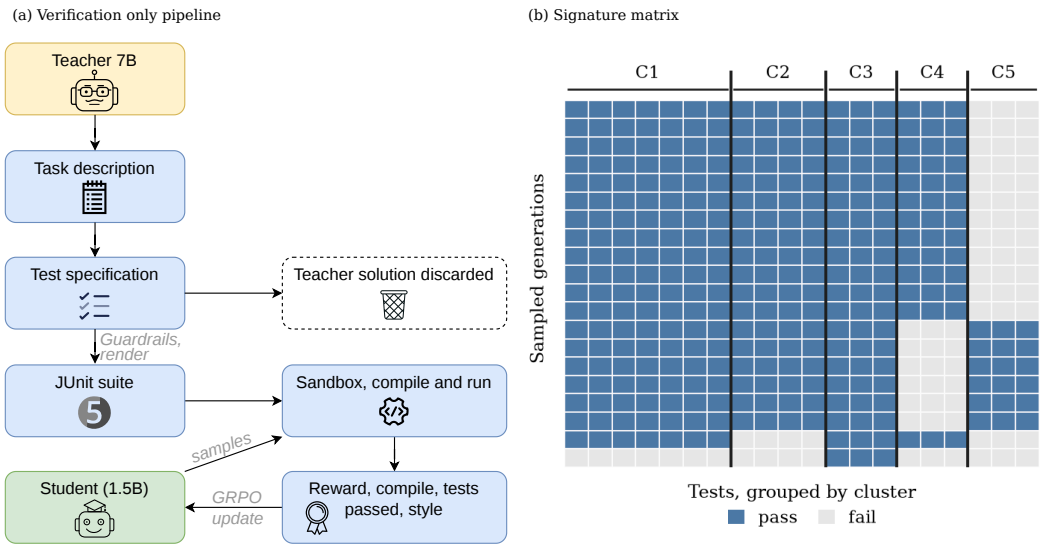


Figure 1: Left, the verification only pipeline. A teacher writes tasks and test specifications, its own solutions are discarded, and a student is trained with GRPO against the rendered suites. Right, the signature matrix of one task, rows are sampled generations, columns are tests. Tests with identical columns form one constraint cluster.

4 TEST COUNT IS NOT CONSTRAINT COUNT

A suite of five tests is not the same thing as five constraints on the implementation. This means that if the tests all behave similarly, the suite will score the candidate on a single property, and the remaining four tests add nothing the first one did not already say. Figure 2 illustrates this on the original task distribution. By plotting the share of the $8{,}300$ generations that pass at least $X\%$ of their suite for each of the three models, we discover that the curve drops at the origin and stays flat across the remaining range except for the final $X = 100\%$ step. This indicates that almost no generation lands in between.

If we then condition on the generations that pass at least one test, the share that passes the entire suite is 70.5% for the untrained base model, 74.7% for the teacher, and 74.7% for the trained student. The three models differ by more than an order of magnitude in compilation rates (3.3%, 40.0% and 92.1% respectively) and by a factor of thirty in pass@1 (Table 2). Despite these disparities, they agree to within four points on conditional quality, which shows that what the suite does once a candidate compiles is a property of the suite not of the model under test. The same reading holds in the redesigned suites where redundancy was removed, see Figure 2(b) where the curve descends in steps rather than sitting flat.

Whether the structure of the suite reaches the optimizer depends on the reward being piecewise. A candidate that fails to compile receives -1.0, and a candidate that produces an answer is scored with $0.7 \cdot r_{\text{pass}} + 0.3 \cdot r_{\text{style}}$. When most generations in a group fail to compile the compilation term dominates, no advantage is created within the group and the suite composition is never observed. Once most generations compile, the test term is what varies within a group and the suite is what the optimizer sees.

The redesigned dataset replaces single-expression utilities with classes whose methods branch and asks the generator for tasks that state boundaries, empty and missing cases and state transitions. Void methods are also asserted through a non-void observer, duplicate tasks are rejected by a normalised description signature, duplicate cases within a suite are rejected by both an exact scenario key and a literal-normalised behaviour key, an AST gate removes trivial and tautological assertions, and every suite must compile against a signature-correct stub before it is accepted. Out of the initial $34{,}631$ suites produced, $6{,}949$ whose entire API is void were removed, leaving $27{,}682$ for analysis.

To measure how many constraints a suite imposes and thus get a measure of its quality, we sample twenty implementations per task, and for each one, the binary vector of which tests it passes forms a codeword. Suites that have more distinct codewords are able to separate candidates along several independent axes and a suite that admits one codeword does not separate any two sampled programs, whatever its test count. Thus, this process allows us to quantify redundancy in suites.

Out of $27{,}682$ suites, $27{,}139$ suites produced at least one complete implementation. In Figure 3, we have the result of suites that have at least two distinct codewords (80.0%) and that number drops to 40.6% for suites with at least four distinct codewords. The curve statistics show that the median suite admits three codeword and has a maximum of seventeen codewords in a suite out of the theoretical maximum of twenty codewords.

Table 1: Effective constraint structure of the original and the redesigned task distributions.

	Original suites	Redesigned suites
Suites analysed	415	27,139
Tests per suite (median)	5	20
Suites with ≥ 2 distinct codewords	39.7%	80.0%
Suites with ≥ 4 distinct codewords	6.3%	40.6%
Median effective constraints (columns)	1	3
Suites with ≥ 2 distinct constraints	44.6%	86.2%
Suites with ≥ 4 distinct constraints	4.8%	45.4%

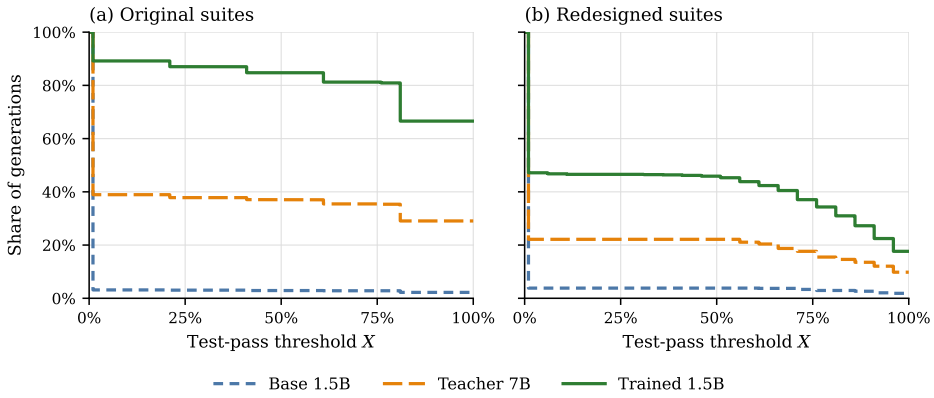


Figure 2: Share of generations passing at least $X\%$ of the suite, for the base, teacher and trained models. (a) Original suites, 5 tests per suite and 8,300 generations per model: a cliff followed by a plateau, so the tests are near redundant; among the trained model's generations that pass at least one test, 75% pass all of them. (b) Redesigned suites, 20 tests per suite and 1,020 generations per model: a staircase, each test eliminating generations the previous ones let through; the same share falls to 37%.

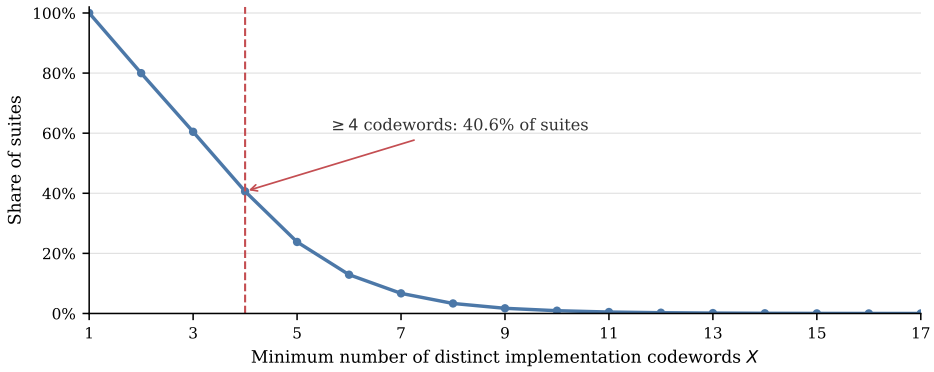


Figure 3: Share of redesigned suites with at least X distinct implementation codewords under the checkpoint used to select tasks (27,139 suites). The dashed line marks the diversity threshold of at least four codewords, met by 40.6% of suites; the controlled experiment selects on the stricter column criterion described in the text, which leaves 9,535 tasks.

5 Constraint diversity causes reliability gains

5.1 Why a between population comparison is confounded

The simplest way to test the importance of constraint diversity is to compare training on tasks whose suites admit many codewords to training on tasks whose suites only admit one. However that comparison is confounded since the two populations are not two levels of a treatment applied to comparable tasks, so the two populations have different task distributions and are separable before any model runs. Taking the distribution as generated with the all-void suites in place, we can train a logistic regression on surface features of the suite alone that separates the 11,026 suites admitting at least four codewords from the 17,680 admitting at most two at a five-fold cross-validated AUC of 0.86.

The features for the logistic regression only contain the number of declared methods, the fraction of void methods, the use of setup calls, the mix of assertion kinds, and the declared scope. The strongest single feature is the number of distinct assertion kinds, with an AUC of 0.78, followed

Table 2: Base, teacher and student on the original suites, 415 tasks, 20 generations per task.

	pass@1 (%)	pass@20 (%)	Compile (%)
Base 1.5B	2.2	29.6	3.3
Teacher 7B	29.1	76.4	40.0
Student 1.5B	66.6	77.6	92.1

by the share of test cases carrying setup calls with 0.73 AUC and the number of distinct return types (0.70 AUC). Stripping the all-void suites removes some but not all of the separability as on the surviving population the same classifier still reaches 0.77. The main problem with this dataset is that the low diversity group is not only easy. It contains 39.3% all-void APIs and 95.5% of all assertions are `assertDoesNotThrow` making it such that any implementation that runs without raising an exception passes every test by construction. This is what motivates holding the task fixed and varying only which of its tests the reward reads (Appendix B).

5.2 Within task design

The population we study is built by sampling twenty implementations per task from an initial checkpoint, grouping the tests of each suite by their exact pass/fail column across those implementations, and retaining the tasks whose suites contain at least three such columns each holding at least three tests. From the original $27{,}682$ tasks, only $9{,}535$ survive this filter. Tasks were then divided $80/20$ by a seeded shuffle of the sorted task identifiers into $7{,}628$ training and $1{,}907$ validation tasks. Both arms train on the same $7{,}628$ tasks with $k = 3$ tests per task and are evaluated on the $1{,}907$ held out tasks against the full twenty test suite.

We then sampled tests from the clusters. The redundant arm sampled all from a cluster with at least three tests, drawn at random with a fixed seed. The diverse arm keeps one of those three and adds one test from each of two other clusters, also at random. As such, the two arms share exactly one test per task, and the number of tests, the tasks, the optimizer, the hyperparameters and the number of steps are all held fixed at $11{,}442$ optimizer steps adding up to exactly three epochs, in every run. Thus the only variable is the behavioural cluster difference.

At the end, every trained arm is evaluated on the validation tasks, all twenty tests, sampling each implementation twenty times. To separate the effect from run to run variation, each arm is trained three times, with seeds 42, 104171 and 123456789, producing six distinct LoRA adapters whose hashes are listed in Appendix D. Uncertainty is quantified by a bootstrap paired at the level of the task, with $10{,}000$ resamples, as described in Appendix E.

5.3 Results

Each diverse run produces a higher pass@1 than every one of the three redundant runs. The weakest diverse run, at 9.6%, still stands 2.2 points above the strongest redundant seed, at 7.4%. Over the three seeds, diversity moves pass@1 from 7.1% to 10.0%, a gain of 2.9 points or a factor of 1.4. Pairing the two arms task by task and resampling tasks with replacement, seed 42 gives a difference of 3.4 points and a 95% interval of $[2.8, 4.0]$. The interval excludes zero, and so does the corresponding interval for each of the other two seeds, at $[2.3, 3.5]$ and $[1.7, 2.8]$.

Furthermore, pass@20 moves by 0.7 points between the arms (14.2 against 15.0, difference on unrounded means, Table 3), and compilation is indistinguishable at roughly 98% in all six runs. This means that both arms can reach a correct implementation about equally often when given twenty attempts with compilation not being a problem. The gain in mean test pass rate, 3.2 points, is of the same size as the gain in pass@1, which is consistent with the diverse arm satisfying more of the suite on a typical attempt rather than solving a different set of tasks outright.

6 Mechanism: diverse constraints densify the reward

Both arms read three tests per task and compute the reward with the same formula with the only difference being how many values that reward can take. When the three tests fall in a single cluster

Table 3: Within task controlled experiment on 1,907 held out tasks, 20 generations per task. Both arms train on the same tasks with three tests per task and differ only in whether the tests come from distinct constraint clusters or from a single cluster. Each cell is the mean over three training seeds, with the minimum and maximum across seeds in brackets. Differences are computed from unrounded means. The paired bootstrap 95% interval on the pass@1 difference is $[2.8, 4.0]$, $[2.3, 3.5]$ and $[1.7, 2.8]$ for the three seeds.

	pass@1 (%)	pass@20 (%)	Compile (%)	Mean test pass (%)
Initial checkpoint	4.1	12.1	76.9	40.0
Redundant (same cluster)	7.1 [6.7, 7.4]	14.2 [14.0, 14.4]	98.1 [98.0, 98.1]	62.1 [61.7, 62.3]
Diverse (distinct clusters)	10.0 [9.6, 10.2]	15.0 [14.9, 15.0]	98.2 [98.2, 98.3]	65.3 [65.1, 65.7]
Diverse minus redundant	+2.9	+0.7	+0.2	+3.2

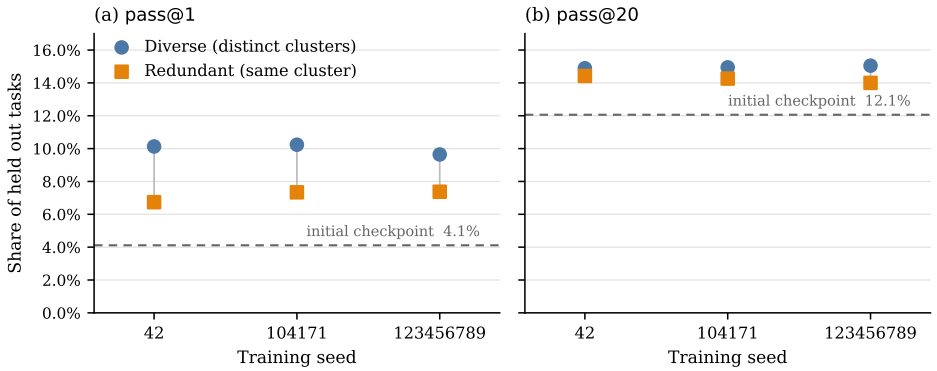


Figure 4: pass@1 (left) and pass@20 (right) for each training seed of both arms, with the initial checkpoint as a horizontal reference. Both panels share a vertical scale, so the two gaps are directly comparable. Diversity moves pass@1 by 2.9 points and pass@20 by 0.7.

they behave similarly, meaning, they pass and fail together, so the test term is close to a binary reward. When the tests come from different clusters, a sample can satisfy one or more independently and the reward separates degrees of partial correctness.

With a binary test term a GRPO group either ties or spreads by the full 0.7, so the within group standard deviation is high and ties are frequent. Partial credit makes the differences smaller and more frequent, which lowers the standard deviation and cuts the ties. Averaged over training, the within group standard deviation is 0.433 for the diverse arm against 0.514 for the redundant one, with the three diverse runs in $[0.426, 0.442]$ and the three redundant runs in $[0.507, 0.518]$, and the gap widens to 0.378 against 0.474 over the last two thousand steps (Figure 5b). The share of groups with zero advantage, where no gradient flows, falls from 50.8% to 43.4%. Neither number measures quality, since the two rewards have different effective scales; they measure how often the optimizer receives a gradient.

7 Discussion and Limitations

Measuring a suite costs less than training against it. The measurement reads twenty samples from a checkpoint that already exists and returns how many constraints the suite imposes on that policy. On the redesigned suites the median is three constraints for twenty tests, and on the original suites it is one constraint for the five tests. Therefore, a suite can grow by a factor of four and give the reward no extra resolution. For a practitioner this means the constraint count should be checked before the compute is committed, and the generation budget should be spent on cases that disagree. It also lowers what is asked of the teacher, since a model that cannot solve a task can still write a suite that separates attempts. In our setting the teacher supplies a specification and a grader, and its own implementations are discarded.

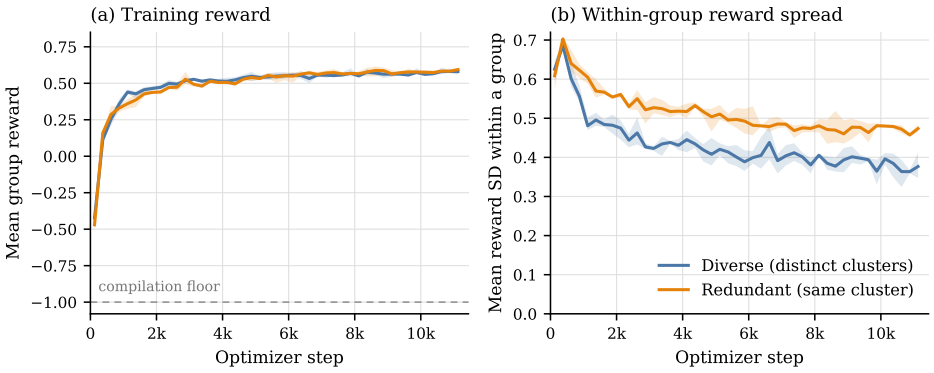


Figure 5: Training dynamics of both arms, averaged over the three training seeds with the band spanning the minimum and maximum across seeds. Left, mean group reward. Right, mean standard deviation of the reward within a GRPO group, which measures the size of the reward spread within a group. Panel (a) does not compare quality across arms, since the two rewards have different effective scales.

The experiment reproduces a pattern already reported for RLVR. pass@20 separates the arms by only 0.7 points while pass@1 moves from 7.1% to 10.0%, which is the evidence used to argue that RLVR redistributes probability inside a fixed reach (Yue et al., 2025). That literature describes the pattern; our arms isolate one cause of it, the constraint count the reward reads. Our arms differ by 2.9 points of pass@1 at equal compute, so how much of the capability already present becomes first attempt reliability depends on the verifier. The base model still sets the ceiling and the suite sets how often the policy reaches it on a single attempt.

This work uses a single model family, one language and synthetic tasks, and evaluates on the distribution it trains on. Thus, it does not show that a diverse suite produces better code in a real world codebase. The effect is small in absolute terms, 2.9 points of pass@1, though every diverse seed is above every redundant seed and the paired intervals exclude zero (Appendix E). Furthermore, the partition also depends on the checkpoint that produced it. Clusters are computed from twenty samples of π_0 and can only split as the sample grows, so every count reported here is a lower bound. The population is restricted as well, since only the 9,535 of the 27,682 tasks whose suites hold at least three clusters of at least three tests are used. Appendix B examines the suites outside that population. Finally, nothing about the suite reaches the optimizer while compilation still controls the reward (Section 4).

8 Conclusion

What a policy learns from execution feedback is bounded by what its verifier can tell apart, and the number of tests does not measure that. The count that does is the number of distinct pass and fail columns the tests of a suite produce over twenty programs sampled from a fixed checkpoint. On the redesigned suites the median constraint count is three, over twenty tests; on the original suites it is one, over five. We hold the task and the number of training tests fixed, and vary only whether the three tests the reward reads come from three clusters or from one. pass@1 moves from 7.1% to 10.0%, every diverse seed stands above every redundant seed, and pass@20 moves by 0.7 points. The count needs only samples from a checkpoint that already exists, so a suite can be judged before any compute is spent training against it. Count the constraints, not the tests.

AI use statement

AI models are part of the method. The dataset is generated by a model, and Section 3.1 describes it in full. The teacher model wrote every test specification and also wrote the blind implementations used to check that each suite can be satisfied, which were then discarded. No task, suite or implementation on the dataset was written by a human.

We used AI assistants for three further purposes, and the authors checked the output of each. We used them for retrieval and discovery, to find candidate related work and to locate the published version of each paper we cite; every reference was then read by the authors, and each claim in Section 2 was checked against a quotation from the cited paper. We used them to draft initial versions of some sections, which the authors rewrote, and to polish writing throughout, for grammar and consistency of terminology.

We did not use AI assistants for research ideation or execution. The research question, the experimental design, the training and the evaluation code, the runs, and the derivation in Section 3.2 are the authors' own work.

The authors verified and tested all code, confirmed every number in the paper against the experimental artifacts, and take responsibility for the content.

Ethics statement

This work involves no human subjects and no personal data. Every test suite and implementation in the dataset was written by a model (Section 3.1), so the data is not scraped from repositories and contains no user content. The released dataset holds task descriptions and test suites, never a solution.

Training and evaluation compile and run model generated code, which is untrusted. Each completion is compiled and executed in an isolated temporary directory that is removed once the run finishes (Appendix D).

Reproducibility statement

The paper carries what is needed to repeat the experiment. Appendix C describes the generation prompts, the guardrails that reject a task and the retention count at each stage. Appendix D gives the adapter configuration, the optimizer settings, the step count, the three training seeds and the SHA256 of each of the six adapters. Appendix E gives the pass@k estimator, the paired bootstrap, the number of resamples and the percentiles we report.

An anonymous repository holds the generation pipeline, the training and evaluation scripts, the bootstrap and the codeword analysis. The repository is available at `https://anonymous.4open.science/r/autotesting-829E/`

References

Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, pp. 21–30. IEEE, 2014. doi: 10.1109/ICST.2014.13. URL `https://doi.org/10.1109/ICST.2014.13`.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The 11th International Conference on Learning Representations (ICLR 2023)*. OpenReview.net, 2023. URL `https://openreview.net/forum?id=ktrw68Cmu9c`.

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`.

R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136. URL https://doi.org/10.1109/C-M.1978.218136.

Zeyun Deng, Yuzhe Lu, Yawei Wang, Linbo Liu, Qing Ping, Han Ding, Guande Wu, Panpan Xu, and Jun Huan. Prism-GRPO: Faster VLA policy optimization via splitting same-outcome groups, 2026. URL https://arxiv.org/abs/2608.17423.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024)*, pp. 4571–4585. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.251. URL https://aclanthology.org/2024.acl-long.251/.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 19034–19055. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

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

M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993. ISSN 1049-331X. doi: 10.1145/152388.152391. URL https://doi.org/10.1145/152388.152391.

Xixiang He, Qiyao Sun, Ao Cheng, Xingming Li, Xuanyu Ji, Hailun Lu, Runke Huang, and Qingyong Hu. Advantage collapse in group relative policy optimization: Diagnosis and mitigation. In *Proceedings of the 43rd International Conference on Machine Learning (ICML*

2026), volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. URL https://arxiv.org/abs/2605.21125.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. HardTests: Synthesizing high-quality test cases for LLM coding, 2025. URL https://arxiv.org/abs/2505.24098.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pp. 435–445. ACM, 2014. doi: 10.1145/2568225.2568271. URL https://dl.acm.org/doi/10.1145/2568225.2568271.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328. Curran Associates, Inc., 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf.

Xin-Ye Li, Ren-Biao Liu, Yun-Ji Zhang, Hui Sun, Zheng Xie, and Ming Li. Exploring pass-rate reward in reinforcement learning for code generation, 2026. URL https://arxiv.org/abs/2605.02944.

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

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17998–18031. Curran Associates, Inc., 2025. doi: 10.52202/085713-0608. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/1a22b912945fb7c0bdd079e792b31b6f-Paper-Conference.pdf.

Xianzhen Luo, Jinyang Huang, Wenzhen Zheng, Qingfu Zhu, Mingzheng Xu, Yiheng Xu, Yuantao Fan, and Wanxiang Che. How many code and test cases are enough? evaluating test cases generation from a binary-matrix perspective. In *The 14th International Conference on Learning Representations (ICLR 2026)*. OpenReview.net, 2026. URL https://openreview.net/forum?id=RomWar2kVN.

Zihan Ma, Taolin Zhang, Maosongcao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. Rethinking verification for llm code generation: From generation to testing. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 137549–137582. Curran Associates, Inc., 2025. doi: 10.52202/085713-4591. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c8ec1867837f5cdaa9e61d40a8b8bb4a-Paper-Conference.pdf.

Wenhua Nie, Jianan Wu, Junlin Liu, Ziwei Li, Zheng Lin, Zijian Zhang, Yilong Fan, Haoran Zheng, and Jyh-Shing Roger Jang. Gradient starvation in binary-reward GRPO: Why group-mean centering fails and why the simplest fix works, 2026. URL https://arxiv.org/abs/2605.07689.

Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, pp. 354–365. ACM, 2016. doi: 10.1145/2931037.2931040. URL https://dl.acm.org/doi/10.1145/2931037.2931040.

Chi Ruan, Dongfu Jiang, Huaye Zeng, Ping Nie, and Wenhu Chen. EvolveCoder: Evolving test cases via adversarial verification for code reinforcement learning, 2026. URL https://arxiv.org/abs/2603.12698.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, Yulia Tsvetkov, Hannaneh Hajishirzi, Pang Wei Koh, and Luke Zettlemoyer. Spurious rewards: Rethinking training signals in RLVR, 2025. URL https://arxiv.org/abs/2506.10947.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Longwen Wang, Yirui Liu, Xuan'er Wu, Xiaohui Hu, Yuankai Fan, Kaidong Yu, Qizhen Weng, Wei Xi, and Xuelong Li. Beyond binary: Turning partial success into dense verifiable rewards for reinforcement learning in code generation, 2026. URL https://arxiv.org/abs/2601.03525.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62787–62874. Curran Associates, Inc., 2024a. doi: 10.52202/079017-2008. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/72da102da91a8042a0b2aa968429a9f9-Paper-Conference.pdf.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wei24h.html.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics (ACL 2025)*, pp. 6980–7008. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.365. URL https://aclanthology.org/2025.findings-acl.365/.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, juncai liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 113222–113244. Curran Associates, Inc., 2025. doi: 10.52202/085713-3775. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 57654–57689. Curran Associates, Inc., 2025. doi: 10.52202/

085713-1933. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/537d5aa768c2d534016a4d06f87bc8fb-Paper-Conference.pdf.

Chuyifei Zhang. When the reward suite is leaky: A preregistered causal contrast of natural verifier false positives in RLVR, 2026. URL https://arxiv.org/abs/2607.11022.

Yiwen Zhang, Xiaodong Yan, Zhenyu Huang, Deng Zhao, Liang Jiang, Qing Cui, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Robust code RL via faulty-code-driven test case synthesis and dense reward shaping. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing (EMNLP 2026)*. Association for Computational Linguistics, 2026. URL https://arxiv.org/abs/2608.24135.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. doi: 10.1145/267580.267590. URL https://dl.acm.org/doi/10.1145/267580.267590.

A Pilot ablation on the number of training tests

Before running the main experiment, we ran a smaller ablation on the original task distribution. A fixed pool of single method utility tasks was split deterministically into nine training and seven held out tasks, and three students were trained for 45 optimizer steps each with rewards reading the first one, three or five cases of a five case suite. All three were then scored against the full suites. The three conditions are indistinguishable. pass@1 is 10.7% at $k = 1$, 11.8% at $k = 3$ and 10.7% at $k = 5$, with compilation near 17% in all three, while the untrained baseline sits at 5.7%.

B The between population confound

This appendix expands the argument of Section 5 that the two codeword populations cannot serve as treatment and control.

The contrast is between the suites admitting at most two codewords (G2) and those that admit at least four (G1), taking the distribution as generated, before the all-void suites are removed. G2 holds $17{,}680$ suites while G1 holds $11{,}026$, the $5{,}382$ suites admitting exactly three fall between the two and enter neither and the 543 with no compiling implementation are also excluded. Out of G2, $6{,}949$ suites, or 39.3%, declare an entirely void API, so the suite has no non-void method to observe a mutation through. That leaves the DSL with only two assertions: 95.5% of its assertions are `assertDoesNotThrow` and the remaining 4.5% are `assertThrows`, and 83.8% of these suites use `assertDoesNotThrow` exclusively. Those suites admit exactly one codeword by construction, since any implementation that runs without raising passes all twenty tests.

G2 suites use fewer distinct assertion kinds, rely more heavily on setup calls, and declare fewer distinct return types; individually these give AUCs of 0.78, 0.73 and 0.70. A logistic regression over the surface feature set reaches a five-fold cross-validated AUC of 0.86 and falls to 0.77 once the all-void suites are removed. Membership is thus largely predictable without executing anything, which is precisely what a valid control group must not permit.

G2 is a population of suites that cannot discriminate since the all-void suites cannot fail a non-throwing implementation at all. Among the $10{,}731$ G2 suites that do declare a non-void method, 28.6% have every test passed by all twenty sampled implementations and 57.4% of their tests on average are never failed by any implementation, while a further 3.2% are uniformly failed, meaning no sampled implementation passed a single test. The equivalent values for G1 are 0% uniformly passed and 11.2% of tests never failed. Therefore, by training against G2 we would be training against a reward that is constant on a large share of its tasks.

Restricting G2 to suites that admit a single codeword leaves $12{,}382$ suites, of which 56.1% are all-void, raises the classifier to an AUC of 0.93, and raises the share of uniformly passed non-all-void members from 28.6% to 56.6%. We then report the looser threshold in the main text as the conservative choice.

We read this as a data quality finding rather than as an available control condition. A suite whose assertions cannot fail carries no constraint regardless of how many tests it contains, and the right

response is to exclude such suites and to vary diversity inside tasks that do constrain, which is the design of Section 5.

C Data generation and guardrails

The generator is asked for classes of two or three instance methods whose contracts state boundaries, empty and missing cases and state transitions, and which admit at least twenty behaviourally distinct tests. It outputs a test specification under a JSON schema that fixes the admissible types, assertions and number of cases, and a deterministic renderer turns it into a JUnit suite.

Each case is validated individually and it may only call methods named in the task, with the declared arity and literal inputs of the declared types. A void method must be observed through a declared non-void method, so an all-void API cannot be asserted. An AST pass rejects assertions that cannot fail, such as `assertTrue(true)` or `assertEquals(x, x)`. Duplicate tasks are rejected by a normalised description signature, and duplicate cases by an exact key and by a key that normalises literals, so renaming a value does not readmit a case. Cases accumulate over up to four attempts, the selection covers every declared method, and satisfiability is checked on every task: the suite must compile against a signature correct stub of the class.

Of $107{,}684$ candidate tasks, $34{,}631$ are accepted (32.2%). Most rejections, 88.1%, come from the generator running out of behaviourally distinct cases before reaching twenty while invalid JSON accounts for 7.7% and failed satisfiability for 4.2%. Removing the $6{,}949$ all-void tasks leaves the $27{,}682$ suites of Section 4, the split is detailed in Appendix B. Satisfiability guarantees the suite is callable, not that a program passes all of it: 52.0% of populated suites contain a test no sampled program passes. These tests form one cluster that no sampled program satisfies.

Compared with the original suites, which held five cases per task with no requirement that they differ, the redesigned suites raise the median effective constraint count from one to three and the share of suites with at least four codewords from 6.3% to 40.6% (Table 1).

D Training and evaluation details

All runs fine tune `Qwen2.5-Coder-1.5B-Instruct` starting from the LoRA adapter of π_0 with a rank of 16, $\alpha = 32$, dropout 0.05, applied to the attention projections, which leaves $4{,}358{,}144$ trainable parameters, or 0.28% of the model. Optimisation is GRPO with four generations per prompt, a per device batch of one, gradient accumulation of eight, learning rate 10^{-5}, KL coefficient $\beta = 0.04$, `bfloat16` and gradient checkpointing, with completions capped at 1024 tokens and no cap on the prompt. The reward is -1.0 for a candidate that fails to compile and $0.7 \cdot r_{\text{pass}} + 0.3 \cdot r_{\text{style}}$ otherwise. Every run covers three epochs of its $7{,}628$ task training set, which is $11{,}442$ optimizer steps, and this is identical across all six runs.

Evaluation loads the base model once under `vLLM` with the arm's adapter applied, generates twenty completions per held out task at temperature 0.6 with at most 1024 new tokens and a context of 2048, then compiles each completion against the task's full twenty test suite and runs it under `JUnit` in an isolated temporary directory. A completion counts as compiling only if both the implementation and the suite compile, and pass@1 is computed from the unbiased estimator over the twenty samples.

Development, the six training runs and all evaluations ran on an institutional cluster with the GPUs listed in Table 4. The two arms of a given seed were trained on the same GPU model, so the hardware difference between seeds cannot favour either arm.

E Statistical procedure

The pass@k is computed with the unbiased estimator of Chen et al. (2021), which for a task with n samples of which c pass all tests is $1 - \binom{n-c}{k} / \binom{n}{k}$, averaged over tasks. With $n = 20$ samples per task this avoids the upward bias of taking the best of k from a finite sample.

Uncertainty on the difference between arms is estimated by a bootstrap paired at the level of the task. Both arms are evaluated on the same $1{,}907$ held out tasks, so for each task we form the difference

Table 4: SHA256 of the LoRA adapter weights for each arm and training seed. All six reached $11{,}442$ optimizer steps, exactly three epochs.

Arm	Seed	GPU	SHA256 (first 16 hex)
Diverse	42	RTX PRO 5000 Blackwell	`666a3a20bac21647`
Diverse	104171	RTX PRO 5000 Blackwell	`cb11d4dc2b2e2b67`
Diverse	123456789	RTX A6000	`c74a56e3388f23da`
Redundant	42	RTX PRO 5000 Blackwell	`1b0119254103a94a`
Redundant	104171	RTX PRO 5000 Blackwell	`439408c55fbd171d`
Redundant	123456789	RTX A6000	`5c433d0ebff0628e`

between the arms' per task pass rates, resample the $1{,}907$ tasks with replacement $10{,}000$ times, take the mean difference within each resample, and report the 2.5 and 97.5 percentiles. Pairing at the task rather than the generation is what the design supports: the two arms share the evaluation tasks but not their samples, and generation i of one arm has no correspondence with generation i of the other, so pairing them would impose a match that does not exist and would understate the variance.

Applied to pass@1, the procedure gives $+3.4$ points with a 95% interval of $[2.8, 4.0]$ for seed 42, $+2.9$ with $[2.3, 3.5]$ for seed 104171, and $+2.3$ with $[1.7, 2.8]$ for seed 123456789. All three exclude zero. The task level differences are strongly concentrated: the diverse arm is ahead on 10.9% of tasks and behind on 3.1%, with the remaining 86.0% tied, which is expected when most held out tasks are solved by neither arm in any of the twenty samples.