JUDGEGYM: VERIFIER-ANCHORED ENVIRONMENTS FOR ATTRIBUTING LLM JUDGE FAILURES

Anonymous authors
Paper under double-blind review

ABSTRACT

A key strength of LLM judges is that they scale evaluation. A judge scores answers no human has time to check, and the same judge serves as the reward model a policy is trained against. However, we find that the labels used to train and test judges are a source of blind spots. Judges are graded on preference labels, and a preference label says which answer was better and nothing about why, so a judge that misses a flipped sign scores the same as one that favors the longer answer. We identify two failure modes of this practice. First, an asserted label cannot attribute a failure to a mechanism, so training judges harder raises aggregate scores without removing the flaw. Second, the oracle that grades answers is lenient by design, and its tolerance silently deletes the test cases that would expose scale errors. We introduce JUDGEGYM, which builds judge tests from benchmarks whose expert-written (gold) answers come with the short program that computes them. JUDGEGYM corrupts a gold program in one known way, re-executes it, and keeps the pair when the benchmark's matcher says the answer changed. Both right verdicts are therefore computed rather than asserted. Each kind of corruption tests one invariant, and a control that changes nothing must always be discarded. Five kinds of change over 7,095 FinQA programs give 24,875 verified pairs, about a quarter of them with the sign flipped, and the control is discarded all 7,095 times. The benchmark's matcher treats 0.125 and 12.5 as the same percentage. That leniency hides 60% of the changes to a constant's scale, and a strict matcher recovers 99%. We prove that this hidden share caps what any model trained against the matcher can gain, at any sampling budget. Since every right response is known, the pairs are an exact training reward as well as a test. We compare JUDGEGYM with nine judge and reward-model benchmarks, specify a one-GPU training recipe and a finance-to-tax-law transfer experiment, and release the engine, the pairs, the harness, and every number as a recomputable artifact. An executed label, not a larger judge, is therefore what makes a judge's failure attributable.

1 INTRODUCTION

Reinforcement learning against a learned judge has become a standard post-training step for large language models, and the judge is often the same model family being trained (Shao et al., 2024; Liu et al., 2025b; Chen et al., 2025; Whitehouse et al., 2026; Xu et al., 2026b;a). The appeal is scale. A finance agent reads a 10-K, subtracts the wrong pair of line items, and reports a clean 3.2 percentage-point margin improvement with a confident citation to a table that does not contain it. On the Finance Agent Benchmark's 537 expert-written questions the best model reaches 46.8% accuracy at $3.79 per query (Bigeard et al., 2025). Wrong is the common case, and no human can check every answer. An LLM judge can. Its verdicts then become the reward the next model is trained to satisfy, so every mistake the judge makes becomes something the model learns to repeat.

The labels that train and test the judge can also become a source of blind spots. Even on JudgeBench's hard pairs frontier judges barely beat random guessing (Tan et al., 2025), and we observe two failure modes in how judges are graded that explain why training them harder does not help.

Preference Labels Carry No Mechanism. Judges are trained and evaluated on preference pairs, and a preference pair records which answer was preferred, not why. When a judge fails on one,

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



Figure 1: JUDGEGYM answers three weaknesses of judge evaluation with preference labels (top row) with three measured changes (bottom row). Left, verdicts are computed by re-executing a corrupted gold program, giving 24,875 pairs and a no-change control that is never kept; middle, each operator family occupies its own region of the difficulty map, so a failure is attributed to a family; right, the production matcher absorbs 348 of 579 scale corruptions, the strict matcher recovers them, and Proposition 2 turns the absorbed share a into a ceiling on best-of-N accuracy.

nothing says whether it missed a sign, a scale, or a miscited cell, or simply preferred the longer answer (Figure 1, top left and top middle). Training judges harder with RL on judgment rewards raises aggregate scores without removing this flaw (Chen et al., 2025; Whitehouse et al., 2026; Xu et al., 2026b;a). A judge trained on labels whose failure modes nobody can list becomes a reward with the same unlisted failure modes. Optimizing against such a reward degrades a model abruptly (Gao et al., 2023; Pan et al., 2022; Skalse et al., 2022). More human labels do not close the gap either. Expert annotation in finance and law is scarce, and even the labels it produces are not firm ground, because experts disagree. MAST, a benchmark of expert-labeled agent failures, released its annotators' individual labels (Cemri et al., 2025). We did not annotate anything ourselves. Recomputing agreement from those matrices gives a pooled Fleiss κ of 0.868 inside the annotators' domain and 0.684 outside it. Behavioral test suites come closest (Ribeiro et al., 2020; Kaushik et al., 2020), but a person still writes and labels each perturbation, so they cover only the errors a person thinks to write down.

A Lenient Oracle Deletes the Tests. Benchmarks whose answers are computed by programs come with an executor and a matcher, the machinery that grades agents. That matcher is lenient by design. It must accept 0.125 and 12.5 as the same percentage so that a correct agent is not penalized for formatting. We show that the same leniency hides the corruptions that change a constant by a factor of one hundred. It deletes 60% of them before any judge sees them (Figure 1, top right). We also prove, in Section 4.3, that this hidden share is a ceiling on what any model trained against that matcher can gain, at any sampling budget.

In this paper, we introduce JUDGEGYM, which turns these two failures into environments for testing and training judges through three changes. Figure 1 connects each failure to its change. First, the labels are executed rather than asserted. Take a FinQA program (Chen et al., 2021) that computes revenue growth as this year's revenue minus last year's, divided by last year's. Swap the two operands of the subtraction and run it again. The executor reports the exact wrong answer with its sign flipped, and no person had to decide that it is wrong. JUDGEGYM applies one of five kinds of corruption, which we call operator families, to each gold program. It keeps the corrupted program only when the matcher says its answer differs from the original's. The two programs then form a verified pair, and a judge shown each answer on its own must accept the original and reject the corruption. Second, each operator family tests one invariant, a rule a correct judge must obey no matter what. A control corruption that changes nothing runs alongside the five and must always be filtered out. A judge's failures are therefore attributed to a family rather than summed into one score. Third, the matcher's strictness is a setting of every environment, reported in both modes. Each combination of program set, operator family, and matcher is one environment. We say environment rather

than benchmark because the exact verdicts double as a training reward. The same structure exists in statutory tax law, where SARA's cases ship with Prolog programs that solve them (Holzenberger et al., 2020). Whether what a judge learns on program-graded pairs carries over to tasks with no program behind them is answered by the transfer experiment of Section 3.5, not assumed.

Our measurements over 7,095 executable FinQA programs show that the construction is nearly free and that its labels are sound. Four of the five families convert 96.7% to 99.4% of the programs they apply to into verified pairs, 24,875 in all. The identity control is discarded all 7,095 times. The scale family yields 39.9% of its programs under the production matcher against 99.1% under the strict one. The families test different skills. In 45.2% of the operand swaps only the sign differs, while 64.0% of the operation substitutions move the answer more than tenfold. Our contributions follow.

• **We propose a way to build verified evaluation environments for LLM judges automatically from the executable structure of existing benchmarks.** A gold program is altered and run again. The new answer is wrong by construction, so a judge is tested with no human labels (Section 3).

• **We instantiate the environments in finance and specify them in tax law.** From 7,095 FinQA programs we release 24,875 verified pairs, with the identity control discarded all 7,095 times (Table 1), and we specify the same procedure for SARA's Prolog programs (Appendix D).

• **We characterize what the environments test and show that the grading oracle is part of the experiment.** In 45.2% of the `swap_args` pairs only the sign differs, 64.0% of the `op_subst` pairs move the answer more than tenfold (Figure 3), and the matcher absorbs 348 of 579 scale corruptions (Table 1), a share we prove caps what any model trained against it can gain.

• **We release a pre-registered protocol and harness for scoring and training judges on the environments.** Three metrics, an exact reward, and runs R1 to R4 are fixed in advance (Section 5). Two deterministic reference judges already show that the environments separate strategies by family. No model judge is evaluated in this version.

2 Related Work

Four lines of work bear on JudgeGym, and we place it against the closest member of each. **RL-trained judges** such as JudgeLRM, J1, J4R, SPCT and TIR-Judge train the judge with judgment rewards (Chen et al., 2025; Whitehouse et al., 2026; Xu et al., 2026a; Liu et al., 2025b; Xu et al., 2026b), and TIR-Judge puts a code executor inside the judge's loop and matches 96% of Claude-Opus-4's listwise performance with 8B parameters (Xu et al., 2026b). The difference is where the executor sits; TIR-Judge's executor helps the judge answer, ours writes the judge's exam. J4R's GRPO groups differ in order, permutations of one pair, whereas ours differ in content, a program and its corruption (Xu et al., 2026a); to our knowledge none of these mines executed mutations, measures oracle tolerance, or tests cross-domain transfer. **Judge and reward-model benchmarks** from MT-Bench and LLMBar through JudgeBench, RewardBench 2, RM-Bench, PPE and Process-Bench (Zheng et al., 2023; Zeng et al., 2024; Tan et al., 2025; Malik et al., 2026; Liu et al., 2025a; Frick et al., 2025; Zheng et al., 2025) assert their labels by human vote, expert curation, a source dataset, hand edits, or model judges; JudgeBench is the closest in spirit, and its pairs come from model responses graded against a source answer, whereas JudgeGym corrupts the derivation itself and lets the executor compute the label. Section 4.5 places the nine resources side by side. **Behavioral testing and mutation** supply the roots. CheckList's invariance tests are the hand-built precedent for our identity control (Ribeiro et al., 2020), counterfactually augmented data showed that minimal edits sharpen a decision boundary (Kaushik et al., 2020), GSM-Symbolic and counterfactual tasks perturb problems to test solvers (Mirzadeh et al., 2025; Wu et al., 2024), and mutation and metamorphic testing have executed small program edits for decades (DeMillo et al., 1978; Jia & Harman, 2011; Segura et al., 2016) without a judge as the system under test. Self-rewarding, meta-rewarding, and self-taught evaluators judge their own outputs or synthetic contrasts (Yuan et al., 2024; Wu et al., 2025; Wang et al., 2024b); those corruptions are judged plausible by the generating model, ours are executed, and the difference matters most when the generator shares the judge's blind spots (Shumailov et al., 2024). Verifier false positives cap what resampling can reach at any budget (Stroebl et al., 2026), the result behind Proposition 2. **Domain anchors and taxonomies.** On the domain side, FinQA, ConvFinQA, and TAT-QA supply executable finance programs (Chen et al., 2021; 2022; Zhu et al., 2021) and FinanceBench the judgment-only side (Islam et al., 2023); SARA

is the executable legal core (Holzenberger et al., 2020), LexGLUE and CUAD its judgment-heavy shell (Chalkidis et al., 2022; Hendrycks et al., 2021); MAST and TRAIL supply expert-annotated failure taxonomies (Cemri et al., 2025; Deshpande et al., 2025). Appendix E discusses each of these lines in full.

3 THE JUDGEGYM FRAMEWORK

3.1 PRELIMINARIES

Five terms recur throughout the paper, and we define them here. A *gold program* p is the executable derivation of a benchmark answer written by the benchmark's annotators, a FinQA arithmetic program or a SARA Prolog program. An *anchor* (O, M) is the benchmark's own grading machinery, an executor O that runs a program over the question's table and text and returns its answer, and a matcher M that decides whether two answers count as the same. A *corruption* $\tilde{p}$ is a program obtained from p by one small change, and a *verified pair* is a pair $(p, \tilde{p})$ whose two answers the matcher tells apart. An *environment* fixes a program set, one kind of corruption, and one matcher mode, and serves the resulting pairs to a judge. Figure 1 shows how the five fit together.

3.2 ANCHORS

The finance anchor is an executor we wrote for FinQA's program language, which chains arithmetic operations over literals, table cells, constants such as `const_100`, and references to earlier steps such as `#0`. We trust it because it reproduces the released gold answer on 98.9% of the training programs (6,182 of 6,251) and 98.2% of the development programs (867 of 883), and because the residual is localized. Programs that only do arithmetic match on 99.5% of 6,053 items with none failing to execute; programs that aggregate over table rows match on 81.3% of 198 items and fail to execute on 15.7%, so we exclude that group from the environments served to judges and track it separately. The engine itself still runs over every program the executor can run, 6,220 in train and 875 in dev, so the identity control of Section 4 covers the full executable set.[1]

The legal anchor is SARA, nine sections of the US tax statute with 376 cases written by the dataset's authors, each with a Prolog program that solves it (Holzenberger et al., 2020).[2] Its executor is SWI-Prolog and its matcher is exact on dollar amounts and on entailment labels, so the tolerance question of Section 4.3 does not arise there (Appendix D). Its size matters less than it appears; Section 4 measures 3.5 verified pairs per gold program on FinQA (3.4 to 4.1 by program length), and two rule-level Prolog corruptions, negating one statutory condition and shifting one threshold constant, multiply the legal cases the same way.

3.3 THE MUTATION ENGINE

For each gold program and each kind of corruption, the engine makes one change, runs the changed program with the same executor that produced the gold, and asks the matcher whether the answer changed. Five kinds of corruption, which we call operator families, are used. `swap_args` exchanges the operands of the first subtraction or division, `op_subst` replaces that operation, `digit_shift` scales one literal by ten, `cell_swap` cites a different cell of the same table, and `const_break` divides a constant by one hundred; each rewrites exactly one span of the program text. Table 3 and Algorithm 1 in Appendix A give the operators as the released engine implements them and the loop it runs. The pair is kept if and only if $M(O(\tilde{p}), O(p))$ is false, that is, only when the matcher can tell the two answers apart; a corruption that changes the program but not its answer is an equivalent mutant in the language of mutation testing (Jia & Harman, 2011) and is discarded.

Why is a kept pair a valid test without a human label? One argument suffices, and we state it once. **Proposition 1 (necessary condition).** Let $(p, \tilde{p})$ be a verified pair, so $M(O(\tilde{p}), O(p))$ is false, and let $O(p)$ agree with the released gold. Then a judge that accepts $\tilde{p}$'s answer, or rejects p's, is wrong

[1] Executability admits a program to the engine; mutation needs only that the program runs, and 6,182 of the 6,220 executable training programs also match the released gold.

[2] SARA is MIT-licensed within LegalBench and runs under SWI-Prolog. Our artifact pins the interpreter version and fetches rather than redistributes the cases.

on that episode. The proof (Appendix F) uses only that the gold program is the accepted derivation of the released answer and that M is the anchor's own notion of sameness; the one exception is a gold answer that is itself wrong, which Assumption A2 below bounds. The converse fails; passing every verified pair does not certify a judge, because the operator families express only some errors.

The identity operator, `noop`, runs alongside the five as a permanent control. It changes nothing, so every `noop` must be discarded, and any survivor means the executor or the matcher is broken. The control earned its keep early, catching a span bug in our first engine within one run (Appendix A).

3.4 Environments, metrics, and reward

An episode presents the judge with the question, the context, and one of the two answers, p's or $\tilde{p}$'s, together with its stated reasoning. The judge outputs accept, reject, or abstain, with a confidence. Both answers are rendered from their programs and differ only in the mutated span, so length and style are held fixed; showing one answer at a time removes position bias by construction, and the harness runs two prompt templates so that no result depends on one wording.

Three assumptions carry the labels. **A1.** The executor reproduces the released gold; testable, and tested at 98.9% on train and 98.2% on dev (Section 3.2). **A2.** The released gold is the correct answer to its question. This is the one exception Proposition 1 allows, and the 1.1% of training programs on which the executor and the released gold disagree bounds how often it can occur. **A3.** Pairs from one program are dependent, so every interval resamples programs. Programs from one filing may also be correlated; the 6,005 training programs with pairs come from 741 filings (8.1 per filing on average, at most 78), and resampling by filing widens the `swap_args` sign-flip interval from [45.5, 48.1] to [44.6, 48.9] and `const_break` from [27.1, 40.6] to [25.4, 42.0]; the harness reports both.

We report three metrics per environment. The *violation rate* is the fraction of pairs on which the judge accepts the corruption or rejects the original. *Sign-flip sensitivity* is the violation rate restricted to sign flips, corruptions whose answer has the opposite sign from the original, the errors a finance user least forgives. *Pairwise consistency* asks whether the judge accepts exactly one of the two answers. Every metric carries a 95% bootstrap interval from 5,000 resamples over gold programs.

The training reward is defined per verdict and averaged per pair. A verdict scores $+1$ when correct, -1 when wrong, and $-c$ when the judge abstains, with a small $c > 0$, so a pair scores $+1$, 0, or -1, and abstaining beats guessing only when the judge's confidence in a verdict falls below $(1 - c)/2$. Because the labels are executed the reward is exact, which matters for GRPO (Shao et al., 2024), the training method of Section 3.5, since it ranks several attempts at the same episode by their rewards and learns from the ranking.

3.5 Training recipe and pre-registered hypotheses

We train 1.5B to 8B judges with LoRA GRPO on the episodes above, eight sampled verdicts per episode with the reward averaged per pair, mixing operator families uniformly and holding one family out to test generalization within the domain. The recipe is deliberately single-GPU, with rollouts served by paged attention (Kwon et al., 2023). Three hypotheses are registered before any training run. **H1. Transfer.** Invariances about direction, magnitude, and citation survive the move from finance programs to statutory Prolog better than surface features of the question text do; the test is the matrix of train $\in$ {finance, legal, joint} against eval $\in$ {finance, legal, held-out family}. **H2. Label efficiency.** Anchor-filtered LLM-judge labels at 50% of the budget match or beat unfiltered labels at 100%, over mixtures of verified, human, and LLM labels at 12.5% to 100% of budget, with predicted non-monotonicity (Gao et al., 2023; Shumailov et al., 2024). **H3. The downstream probe.** A small policy trained against each judge variant and scored by the executor detects reward hacking that the judge's own agreement metrics miss.

4 Measured Environment Properties

The introduction claimed four things about the environments, that mutation is nearly free, that the labels are computed rather than guessed, that the families test different skills, and that the matcher decides which tests exist. This section measures each. Every number is recomputed by the artifact from the released mutation outputs; Table 1 covers all 7,095 executable programs, and the composi-

Table 1: Verified-pair yields over the 7,095 executable FinQA programs, with the strict row rescoring the same 579 `const_break` mutations.

family	tried	kept	yield	sign flips	flip share
`swap_args`	6,021	5,982	99.4%	2,809	47.0%
`op_subst`	6,790	6,722	99.0%	1,093	16.3%
`digit_shift`	6,909	6,768	98.0%	1,107	16.4%
`cell_swap`	5,346	5,172	96.7%	957	18.5%
`const_break`, production M	579	231	39.9%	82	35.5%
`const_break`, strict M	579	574	99.1%		
`noop` (control)	7,095	**0**	0.0%	0	
total, production M		24,875		6,048	24.3%

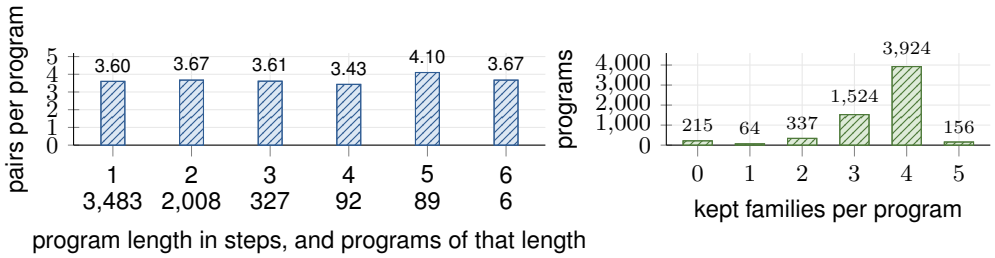

Figure 2: Spread of the 21,786 training pairs over the 6,220 training programs, by program length (left) and by the number of families that produce a pair (right).

tion analyses use the 6,220 training programs and their 21,786 kept pairs, the split the environments train on.

4.1 Yields and the identity control

Table 1 reports what the engine produced, and the first claim holds; verified mutation is close to free. Four of the five families convert 96.7% to 99.4% of the programs they apply to into labeled test cases, and the whole set yields 3.5 verified pairs per program. The families fire on different programs, which is why the tried counts differ and why the scale-convention family is a tenth the size of the others. What is lost is small and listed per family in Appendix A; the 300 equivalent mutants of `const_break` are the subject of Section 4.3. The `noop` row is the second claim. The identity operator was tried on all 7,095 programs and kept zero times, so the pipeline never labels an unchanged answer as wrong, the same guarantee CheckList's invariance tests give by hand (Ribeiro et al., 2020) and here given by execution. Yields are population counts over every executable program and carry no sampling interval; the composition shares of Section 4.2 do, resampled over programs because pairs from one program are not independent.

4.2 Difficulty composition

A kept pair certifies that the corruption is wrong, not how wrong, so the third claim, that the families test different skills, needs a measurement of how far each family moves the answer. Figure 3 bins the ratio of corrupted to gold magnitude, and the families occupy different regions. `swap_args` lives in the near band; 45.2% of its pairs stay within 5% of the gold magnitude, and 2,319 of those 2,366 pairs are exact negations, because swapping the operands of a subtraction flips the sign and nothing else. These pairs cannot be caught by asking whether the size of the answer is plausible. They can be caught only by reading the question. In at least 81.0% of the `swap_args` sign flips (1,982 of 2,448) the question names a direction. A rule that mirrors that wording still catches only 21.8% of them (Section 5). So the pair tests whether the judge grounds the sign in the question and the two cited cells, not in a keyword. `op_subst` sits at the opposite end, with 64.0% of its pairs beyond a factor of ten and a median ratio of 21.5, because replacing a division with a multiplication squares the operands' scale. `digit_shift` is bimodal at exactly 10 and 0.1, its lower quartile 0.1 and

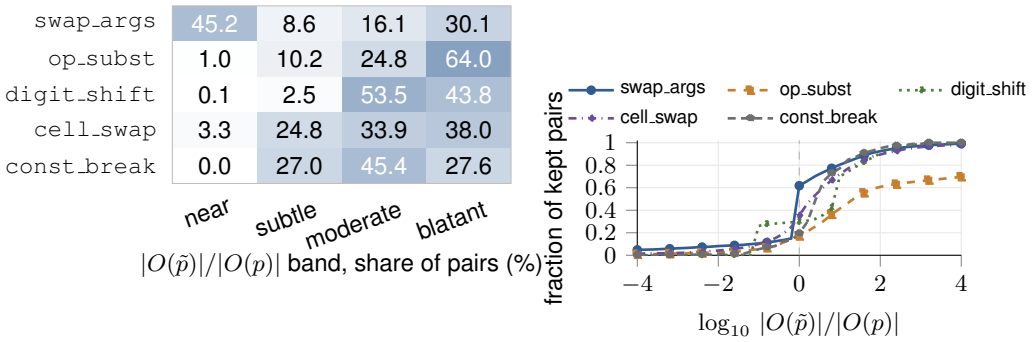

Figure 3: Difficulty composition of the training pairs by magnitude band (left) and as a cumulative log-ratio (right), where near means within 5% of the gold magnitude, subtle up to a factor of 2, moderate 2 to 10, and blatant beyond 10.

its median 10.0, which puts 53.5% of it in the moderate band, and `cell_swap` is the most evenly spread, 3.3%, 24.8%, 33.9%, and 38.0% across the four bands, because the alternative cell is a real number from the same table (quantiles in Appendix G). Magnitude is therefore a strong cue for two families and no cue for half of a third, so a per-family violation profile is a diagnostic rather than a leaderboard, in the same way RM-Bench's difficulty levels reorder reward models that a single score would tie (Liu et al., 2025a). Sign-flip shares carry 95% bootstrap intervals over programs of [45.5, 48.1] for `swap_args`, [15.2, 17.1] for `op_subst`, [15.2, 17.1] for `digit_shift`, [17.4, 19.7] for `cell_swap`, and [27.1, 40.6] for `const_break`, wide because only 187 training pairs survive the production matcher; ratios are undefined for 251 of the 21,786 pairs, 159 of them `cell_swap`, where an answer is zero.

4.3 Oracle tolerance

The two `const_break` rows of Table 1 carry the fourth claim and the paper's cautionary result. The production matcher accepts 0.125 and 12.5 as the same answer because, when grading agents, both percent conventions are correct. Under that matcher 348 of the 579 scale-convention corruptions, 60.1%, produce an answer the oracle cannot distinguish from gold, and the family's yield collapses to 39.9%. Disable the percent equivalence and the identical corruptions yield 99.1%, 574 of 579; strict mode compares exactly, dropping both the equivalence and the production tolerance of 1% of the gold, and the five mutations it still filters are ones whose answer does not change at all, because the constant cancels or a comparison keeps its truth value. The family rewrites the first constant containing `const_100`, so `const_1000` becomes `const_10` as well, and the equivalence absorbs those conventions too. Nothing about the mutations, the programs, or the seed changed between the two runs, and the identity control is filtered under both; only the oracle's definition of sameness changed. Mutation testing knows this as the equivalent-mutant problem (Jia & Harman, 2011; Segura et al., 2016), but there the question is whether a mutant is equivalent, whereas here the matcher's tolerance decides it, and a tolerance chosen to grade agents fairly deletes the test cases a judge needs. We therefore make matcher mode an environment parameter, strict by default for scale invariance, and note that the same 1% tolerance sets a floor under every family, since a corruption closer than that to the gold is never kept.

The absorbed pairs matter beyond this benchmark, because the same matcher is the verifier a policy is graded against, and a wrong answer the matcher accepts is a free pass. **Proposition 2 (tolerance ceiling).** Let a policy draw answers independently, correct with probability c and wrong with probability w, and let the matcher M, which accepts every correct answer, absorb a fraction a of the wrong ones. Best-of-N selection over answers M accepts returns a wrong answer with probability at least $aw/(c + aw)$ as N grows, and no budget lowers it, because absorbed answers pass with probability one (Stroebl et al., 2026). The proof is in Appendix F. If the matcher also rejects a fraction $1 - r$ of correct answers, the bound becomes $aw/(rc + aw)$. At the measured $r = 0.989$ the change is below one point. On the scale-convention corruptions of this environment the absorption rate is 0.601 under the production matcher and 0.009 under the strict one; a policy enters the bound with its own rate, and one whose errors are scale errors keeps 60.1% of them invisible to the pro-

Table 2: Nine judge and reward-model resources by how their labels are made.

resource	size	label source	objective	executed	identity	reward
MT-Bench (human)	3,355 votes	human votes	✗	✗	✗	✗
LLMBar	419 pairs	expert curation	✓	✗	✗	✗
RewardBench	2,985 trios	human, verifiable subsets	partly	✗	✗	✗
JudgeBench	350 pairs	source-dataset answers	✓	✗	✗	✗
RM-Bench	1,327 items	hand-edited responses	✓	✗	✗	✗
RewardBench 2	1,865 prompts	verifiers, LLM judges, humans	partly	partly	✗	✗
TRAIL	148 traces	expert annotation	✓	✗	✗	✗
ProcessBench	3,400 cases	expert step annotation	✓	✗	✗	✗
JUDGEGYM (ours)	24,875 pairs	re-execution	✓	✓	✓	✓

duction verifier at any budget. The oracle can be audited rather than trusted, since the artifact ships a reference executor that reproduces every value in the released outputs and documents both modes (Appendix B).

4.4 COVERAGE AND PROGRAM SHAPE

An environment that drew most of its pairs from a few programs would test those programs rather than the judge, so Figure 2 asks how the pairs are spread. Of the 6,220 training programs, 3,924 yield pairs from four families, 1,524 from three, and 156 from all five, while 215 yield nothing, typically programs with no `subtract` or `divide`, no table-sourced literal, and a literal whose scaling the matcher tolerates (right panel). Program length does not move the multiplier (left panel); one-step programs yield 3.60 pairs each, two-step programs 3.67, three-step programs 3.61, and the 89 five-step programs 4.10, so yield does not decay with reasoning depth. Negative gold answers appear in 2,753 of the 21,786 kept pairs, so a judge cannot learn to reject every negative number. At the measured multiplier each numerical SARA case would yield 3 to 4 verified pairs before rule-level mutation, a projection to be replaced once the Prolog mutations run.

4.5 HOW JUDGEGYM COMPARES WITH EXISTING RESOURCES

Where does JUDGEGYM sit among the resources a reviewer would compare it with? Table 2 lists nine judge and reward-model evaluation resources (Zheng et al., 2023; Zeng et al., 2024; Lambert et al., 2025; Tan et al., 2025; Liu et al., 2025a; Malik et al., 2026; Deshpande et al., 2025; Zheng et al., 2025) by four properties. Objective means the label has a single correct answer; executed means a program computes the label; identity means the resource ships a no-change control; reward means the labels are meant to be trained against. Every prior resource asserts its labels, by human vote, expert curation, a source dataset, hand edits, or model judges. Asserted labels can be objective, as LLMBar's 94% expert agreement shows, yet their supply is bounded by annotator time, which is why the objective resources are small, and their perturbations by what a person can foresee, which is why none ships a no-change control. JUDGEGYM inverts both constraints; its 24,875 pairs cost no labeling, its identity control checks every program, and its computed reward makes it the only resource in the table meant to be trained against. It covers a narrower slice than the others, arithmetic and statutory computation, exactly and at scale, so a judge should be measured on both kinds of resource.

5 JUDGE EVALUATION PROTOCOL

The environments are cheap to build and cheap to run a judge on. The engine processes the 6,251 FinQA training programs in under four seconds on one CPU core, and a full development-split run for one judge is 6,182 episodes, about 15 million input tokens with both templates and the recall probe. Table 5 in Appendix C fixes the protocol the released harness runs. Every results table carries two analytic baselines, accept-everything, whose pairwise consistency is zero, and coin-flip, whose expected consistency is 50% and whose pair reward is zero; any policy that gives the same verdict to every episode earns pair reward zero, so the reward has no degenerate optimum.

R0. Reference judges. Before any model runs, two deterministic judges test whether the environments separate strategies. The first is a sign rule. It reads the direction the question names and rejects an answer whose sign contradicts it. On the development split it catches 21.8% of the `swap_args` sign flips and wrongly rejects 123 of the 3,091 originals. The reason is that FinQA states many decreases as positive percentages, so mirroring the question's wording does not recover the sign. The second is a magnitude rule. It rejects an answer larger than ten times the largest value in the table. It catches 23.9% of the `op_subst` pairs and 19.9% of the blatant band, and it wrongly rejects 402 originals (13.0%). Neither shortcut solves any family. Their violation rates, 95.3% [94.2, 96.3] and 89.7% [88.6, 90.8], sit between the coin-flip baseline at 75% and accept-everything at 100%. Their per-family profiles differ where the difficulty map says they should.

R1. Prompted judges. Three small open models and two frontier models run zero-shot across every environment. Scalar reward models enter by thresholding an answer's score at the midpoint of its pair's two scores, and fine-tuned judges such as JudgeLM run unchanged. No judge is tuned per environment, every judge runs both shipped templates, the abstention cost is fixed at $c = 0.1$, and an executor-backed oracle judge that re-runs the candidate program is reported as the ceiling of every table. R1 reports the three metrics of Section 3.4 per family and per magnitude band (Section 4.2), with intervals, and compares judges by paired bootstrap over gold programs. Memorization cuts the other way here. A judge that has memorized FinQA's answers rejects every corruption, since corrupted values appear in no corpus. Recall can therefore inflate only the acceptance of originals. A recall probe asks the judge for the answer with no candidate shown and reports separately the programs whose gold it reproduces.

R2. GRPO judges. The recipe of Section 3.5 with the held-out family protocol, reporting per-family violation and consistency before and after training over three seeds. **R3. Transfer.** The H1 matrix on SARA and the judgment-only sets. **R4. Labels and hacking.** The H2 mixtures and the H3 probe, reporting violation against label budget and executor-scored policy accuracy per judge variant.

6 Discussion and Limitations

Limitations. Five limits bound the measurements, stated as non-claims. First, the only judges run are the two deterministic rules of R0. No model judge has been evaluated, and the composition analysis of Section 4.2 covers the training split only. Second, the pairs are verified program-semantic flips, not adversarial pairs in LLMBar's sense, wrong answers written to look attractive. Third, invariance is necessary, not sufficient; a judge can pass every operator family and still fail on errors no mutation expresses, so JudgeGym lower bounds a judge's failures rather than certifying their absence. Fourth, the verified reward stops where the gold programs stop, by design, and until R3 runs nothing here says whether an invariance learned on FinQA holds on FinanceBench. Fifth, the legal set is small, 376 cases, so legal conclusions will carry wider intervals.

Two boundary conditions matter for reuse. The operators assume a program representation that exposes literals, operators, and step references as spans; equation traces, SQL, and unit-tested code expose the same, though we have not run the engine on them. And strict matching would punish legitimate conventions in agent grading, so the two matcher modes are complements rather than a correction.

7 Conclusion

Judges are graded today by labels someone asserted; JudgeGym grades them by labels something computed. From 7,095 public programs it produces 24,875 verified pairs, a composition that says which families test direction and which test magnitude, a measured warning that the oracle's tolerance is part of the experiment, and a reward that turns the same signal into training. The executable side of finance and law has subsidized agent evaluation for years; the same executors can subsidize judge training.

Reproducibility statement

The artifact contains the mutation engine, the FinQA executor, the strict and production matchers, the per-split mutation outputs (`judgegym_mutations_{train,dev}.json`) of the run described in Section 4, the analysis script that regenerates every number in Table 1, Figures 2 and 3, and Section 4 from those outputs and writes every figure's data file, and the judge harness of Section 5, with seed 13 for operator sampling and seed 7 for the 5,000 bootstrap resamples. Regenerating Table 1 from scratch requires the FinQA release, which the script fetches, and the executor with both matcher modes, which the artifact includes. The shipped reference executor reproduces every original and corrupted value in the released outputs and executes more table-aggregation programs than the run's executor did, so a fresh run covers 6,251 rather than 6,220 training programs, a difference the artifact documents. Datasets are fetched from their original repositories under their original licenses and are not redistributed. An anonymized repository accompanies the submission.

Ethics statement

This work involves no human subjects, no human annotation, and no private data. FinQA and SARA are public benchmarks used under their licenses; the released pairs contain program text and computed values only, not the FinQA text or tables, which the artifact fetches from the original repository. The mutation engine produces wrong answers on purpose, and we release them only as paired test cases whose correct verdicts are known, with a manifest that marks every corrupted value as such, so a crawler that ingests the artifact cannot mistake corruptions for gold. Judges trained on these environments inherit the coverage of the two domains, arithmetic and statutory computation rather than the judgment those domains also require, and a judge that passes every operator family is not thereby certified correct. Section 6 states this limit, and we ask users of the suite to restate it.

AI use statement

We used a large language model as an assistant for drafting and editing the text, for drafting the TikZ and PGFPlots figure code, the analysis script, the reference executor, and the judge harness from our own data and specifications. The research idea, the mutation engine, the executor used for the reported runs, and all runs are the authors' own. Every reported number is recomputed by the released scripts from the released outputs, and the authors reviewed all AI-assisted text and code. We take full responsibility for the content of this paper.

References

Antoine Bigeard, Langston Nashold, Rayan Krishnan, and Shirley Wu. Finance agent benchmark: Benchmarking LLMs on real-world financial research tasks. *arXiv preprint arXiv:2508.00828*, 2025.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Ilias Chalkidis, Abhik Jana, Dirk Hartung, Michael Bommarito, Ion Androutsopoulos, Daniel Katz, and Nikolaos Aletras. LexGLUE: A benchmark dataset for legal language understanding in English. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4310–4330, 2022. doi: 10.18653/v1/2022.acl-long.297.

Nuo Chen, Zhiyuan Hu, Qingyun Zou, Jiaying Wu, Qian Wang, Bryan Hooi, and Bingsheng He. JudgeLRM: Large reasoning models as a judge. *arXiv preprint arXiv:2504.00050*, 2025.

Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, Hanghang Tong, and Heng Ji. RM-R1: Reward modeling as reasoning. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv:2505.02387.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. FinQA: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3697–3711, 2021. doi: 10.18653/v1/2021.emnlp-main.300.

Zhiyu Chen, Shiyang Li, Charese Smiley, Zhiqiang Ma, Sameena Shah, and William Yang Wang. ConvFinQA: Exploring the chain of numerical reasoning in conversational finance question answering. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 6279–6292, 2022. doi: 10.18653/v1/2022.emnlp-main.421.

Richard A. DeMillo, Richard J. Lipton, and Frederick 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.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. TRAIL: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Evan Frick, Tianle Li, Connor Chen, Wei-Lin Chiang, Anastasios N. Angelopoulos, Jiantao Jiao, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. How to evaluate reward models for RLHF. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.14872.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning (ICML), PMLR 202*, pp. 10835–10866, 2023.

Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. CUAD: An expert-annotated NLP dataset for legal contract review. In *Advances in Neural Information Processing Systems 34 (NeurIPS), Datasets and Benchmarks Track*, 2021.

Nils Holzenberger, Andrew Blair-Stanek, and Benjamin Van Durme. A dataset for statutory reasoning in tax law entailment and question answering. In *Proceedings of the Natural Legal Language Processing Workshop 2020 (NLLP@KDD), CEUR Workshop Proceedings Vol-2645*, pp. 31–38, 2020. arXiv:2005.05257.

Pranab Islam, Anand Kannappan, Douwe Kiela, Rebecca Qian, Nino Scherrer, and Bertie Vidgen. FinanceBench: A new benchmark for financial question answering. *arXiv preprint arXiv:2311.11944*, 2023.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62.

Divyansh Kaushik, Eduard Hovy, and Zachary C. Lipton. Learning the difference that makes a difference with counterfactually-augmented data. In *The Eighth International Conference on Learning Representations (ICLR)*, 2020. arXiv:1909.12434.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 4334–4353, 2024. doi: 10.18653/v1/2024.emnlp-main.248.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1755–1797, 2025. doi: 10.18653/v1/2025.findings-naacl.96.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. RM-Bench: Benchmarking reward models of language models with subtlety and style. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2410.16184.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*, 2025b.

Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A. Smith, Hannaneh Hajishirzi, and Nathan Lambert. RewardBench 2: Advancing reward model evaluation. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv:2506.01937.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.05229.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *The Tenth International Conference on Learning Representations (ICLR)*, 2022.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4902–4912, 2020. doi: 10.18653/v1/2020.acl-main.442.

Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi: 10.1109/TSE.2016.2532875.

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

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. AI models collapse when trained on recursively generated data. *Nature*, 631:755–759, 2024. doi: 10.1038/s41586-024-07566-y.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022. arXiv:2209.13085.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. The limits of inference scaling through resampling. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv:2411.17501.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. JudgeBench: A benchmark for evaluating LLM-based judges. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.12784.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9440–9450, 2024a. doi: 10.18653/v1/2024.acl-long.511.

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024b.

Chenxi Whitehouse, Tianlu Wang, Ping Yu, Xian Li, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. J1: Incentivizing thinking in LLM-as-a-judge via reinforcement learning. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv:2505.10320.

Tianhao Wu, Weizhe Yuan, Olga Golovneva, Jing Xu, Yuandong Tian, Jiantao Jiao, Jason E. Weston, and Sainbayar Sukhbaatar. Meta-rewarding language models: Self-improving alignment with LLM-as-a-meta-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 11537–11554, 2025. doi: 10.18653/v1/2025.emnlp-main.583.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1819–1862, 2024. doi: 10.18653/v1/2024.naacl-long.102.

Austin Xu, Yilun Zhou, Xuan-Phi Nguyen, Caiming Xiong, and Shafiq Joty. J4R: Learning to judge with equivalent initial state group relative policy optimization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1492–1511, 2026a. doi: 10.18653/v1/2026.acl-long.67.

Ran Xu, Jingjing Chen, Jiayu Ye, Yu Wu, Jun Yan, Carl Yang, and Hongkun Yu. Incentivizing agentic reasoning in LLM judges via tool-integrated reinforcement learning. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026b. arXiv:2510.23038.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML), PMLR 235*, 2024. arXiv:2401.10020.

Zhiyuan Zeng, Jiatong Yu, Tianyu Gao, Yu Meng, Tanya Goyal, and Danqi Chen. Evaluating large language models at evaluating instruction following. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.07641.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. ProcessBench: Identifying process errors in mathematical reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1009–1024, 2025. doi: 10.18653/v1/2025.acl-long.50.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*, 2023. arXiv:2306.05685.

Yilun Zhou, Austin Xu, Peifeng Wang, Caiming Xiong, and Shafiq Joty. Evaluating judges as evaluators: The JETTS benchmark of LLM-as-judges as test-time scaling evaluators. In *Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267*, pp. 79436–79471, 2025. arXiv:2504.15253.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3277–3287, 2021. doi: 10.18653/v1/2021.acl-long.254.

Lianghui Zhu, Xinggang Wang, and Xinlong Wang. JudgeLM: Fine-tuned large language models are scalable judges. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. Spotlight; arXiv:2310.17631.

A THE MUTATION ENGINE AS RELEASED

Figure 4 runs all six operators on one program and Algorithm 1 is the loop the released engine runs over each gold program. Each operator rewrites one span of the program text, and the rewrite is span-aware, so a step reference such as `#10` is never touched by a literal rewrite; our first engine used plain string replacement, corrupted such a reference while aiming at a literal, and the identity control and the executor's crash caught it within one run. Table 3 defines the operators. `swap_args` exchanges

`subtract(84, 80), divide(#0, 80), multiply(#1, const_100)` $O(p) = 5.0$

operator	mutated step $\tilde{p}$	$O(\tilde{p})$	matcher decision
swap_args	`subtract(80, 84)`	-5.0	✓ kept, sign flip, magnitude unchanged
op_subst	`add(84, 80)`	205.0	✓ kept, a factor of 41
digit_shift	`subtract(840, 80)`	950.0	✓ kept, a factor of 190
cell_swap	`subtract(84, 76)`	10.0	✓ kept, a factor of 2
const_break	`multiply(#1, const_1)`	0.05	✗ production M filters, strict M keeps
noop	`unchanged`	5.0	✗ filtered, the control

Figure 4: One gold program and its six corruptions with illustrative values, where the scale-convention corruption survives only under the strict matcher.

the two operands of the first `subtract` or `divide`. `op_subst` replaces the first `subtract` with `add`, or the first `divide` with `multiply`, or the reverse. `digit_shift` multiplies one randomly chosen numeric literal by ten (seed 13). `cell_swap` replaces a literal that appears in the table with a different value from the same table. `const_break` rewrites the first constant containing `const_100`, dividing it by one hundred; of the 187 training pairs it keeps under the production matcher, 152 rewrite `const_100`, 32 rewrite `const_1000`, and 3 rewrite `const_1000000`. Table 4 gives the per-split counts behind Table 1; the dev columns are the combined counts of Table 1 minus train, because the released mutation output covers the training split.

Table 3: Operator families as implemented in the released engine, each rewriting one span and firing only where its pattern exists, with counts over the 6,220 training programs.

family	rewrite	fires on	error it models
`swap_args`	$(a, b) \to (b, a)$ in the first `subtract` or `divide`	5,265	direction, ratio inversion
`op_subst`	first `subtract`↔`add` or `divide`↔`multiply`	5,951	wrong operation
`digit_shift`	one numeric literal $x \to 10x$	6,064	unit and scale slip
`cell_swap`	a table-sourced literal → another cell of the same table	4,685	evidence miscitation
`const_break`	first constant containing `const_100`, divided by 100	487	scale convention
`noop`	identity	6,220	control, must be filtered

Algorithm 1 The mutation engine for one gold program p with table T, anchor (O, M), operator families F and control `noop`, as released.

$g \leftarrow O(p, T)$; **if** g is undefined **then return** $\emptyset$ ▷ outside the anchor
for $f \in F \cup \{\texttt{noop}\}$ **do**
  $\tilde{p} \leftarrow f(p, T)$; **if** f does not fire on p **then continue**
  $c \leftarrow O(\tilde{p}, T)$; **if** c is undefined **then** record unexecuted; **continue**
  if $M(c, g)$ **then** record no-delta, an equivalent mutant ▷ `noop` must always land here
  else emit $(p, \tilde{p}, g, c)$, flag a sign flip if $c \cdot g < 0$
  end if
end for

B MATCHER SEMANTICS AND THE FIVE SURVIVING EQUIVALENT MUTANTS

The production matcher accepts a prediction x for a gold g when $|x - g| \leq 0.01\,|g|$, or when $100x$ or $x/100$ satisfies the same test, which is the percent equivalence of Section 4.3. The strict matcher accepts only $|x - g| \leq 10^{-9}$. The tolerance is the one the released outputs imply; no kept pair differs from its gold by less than 1.01%, and the closest kept pair differs by exactly 1.0101%. The five `const_break` mutations the strict matcher still filters are two programs whose rewritten constant sits in an intermediate step the final answer does not use, and three programs whose final

Table 4: Per-split counts behind Table 1 under the production matcher, with train counts from the released mutation output and dev counts derived as combined minus train.

	train (6,220 programs)					dev (875 programs)		
family	tried	kept	unexecuted	no-delta	sign flips	tried	kept	sign flips
`swap_args`	5,265	5,231	3	31	2,448	756	751	361
`op_subst`	5,951	5,895	0	56	951	839	827	142
`digit_shift`	6,064	5,939	21	104	956	845	829	151
`cell_swap`	4,685	4,534	47	104	840	661	638	117
`const_break`	487	187	0	300	63	92	44	19
`noop`	6,220	0	0	6,220	0	875	0	0
total		21,786			5,258		3,089	790

step is a `greater` comparison that keeps its truth value; in all five the answer does not change at all. The reference executor in the artifact reproduces all 21,786 original and all 21,786 corrupted values of the released training output and separates every kept pair under the production matcher.

C JUDGE HARNESS

Table 5: The judge evaluation and training protocol shipped with the artifact and fixed in its configuration files before any judge is run.

component	setting
episode and output	question, context, one answer with its reasoning; verdict accept, reject, or abstain
prompted judges (R1)	Llama-3.1-8B-Instruct, Mistral-7B-Instruct, Qwen2-7B-Instruct, GPT-4o, Claude 3.5 Sonnet
trained judges (R2)	1.5B, 3B, 8B backbones, LoRA rank 16, learning rate 2×10^{-4}, batch 32, seeds 42, 43, 44
GRPO	8 rollouts per episode, 1,000 steps, operator families mixed uniformly, one family held out per run
reward and metrics	$+1$ correct, -1 wrong, $-c$ abstain per verdict, averaged per pair; violation rate, sign-flip sensitivity, pairwise consistency
intervals and splits	95% bootstrap, 5,000 program resamples, seed 7; FinQA train for mutations, dev for evaluation, SARA case-wise

Table 5 lists the protocol; this section gives what the table cannot. An episode presents the question, the report text, the table (one row per line), and one candidate answer with its reasoning rendered from the program as numbered steps with intermediate values. Template A asks for a verdict directly; template B asks the judge to identify the needed cells, check the operation and the scale, and then give a verdict. Both end with the same two lines, a verdict in $\{$accept, reject, abstain$\}$ and a confidence in $[0, 1]$. The recall probe presents the question, text, and table with no candidate and asks for the numeric answer; a program whose gold the judge reproduces is reported separately. The harness parallelizes requests, logs input and output tokens and wall clock per judge, stores per-pair verdicts, and reports the metrics of Section 3.4 per family and per magnitude band with 95% percentile bootstrap intervals over 5,000 program resamples, beside the accept-all and coin-flip baselines and the executor-backed oracle judge. Beyond the table, the training recipe uses LoRA alpha 32 and dropout 0.05 on the attention projections, an effective batch of 32 reached by gradient accumulation, and $c = 0.1$ in the reward.

D THE LEGAL ENVIRONMENT AS SPECIFIED

Statutory reasoning in SARA applies one shared Prolog encoding of the statute to many cases, each a file of facts and a query, so rule-level mutation edits the statute and re-executes every case. The released legal engine implements four families and runs them through SWI-Prolog. `negate_condition` wraps one body goal of one statutory rule in negation as failure; `shift_threshold` multiplies one numeric constant in an arithmetic comparison by ten;

`swap_comparison` replaces the first > by <, or >= by =<, and the reverse; `noop` is the identity control, which the engine asserts is filtered on every run. A (case, mutation) pair is kept when the case's answer changes under exact matching on dollar amounts or on the truth value of an entailment query; a mutated statute that derives no answer is recorded as unexecuted, not as a wrong answer. The engine has been exercised on a synthetic statute with the same structure; the runs on SARA's own encodings have not been performed, and the projection of Section 4.4 is to be replaced by their measurement.

E Extended related work

RL-trained and reasoning judges. JudgeLRM trains 3B and 7B judges with judgment-outcome rewards (Chen et al., 2025) and J1 casts judging prompts into a verifiable-reward format for RL (Whitehouse et al., 2026). J4R groups GRPO rollouts by equivalent initial state to remove position bias (Xu et al., 2026a; Wang et al., 2024a); its groups differ in order, permutations of one pair, whereas ours differ in content, a program and its corruption. SPCT scales generative reward models at inference time (Liu et al., 2025b), on the fine-tuned-judge foundation JudgeLM laid (Zhu et al., 2025). Closest to us, TIR-Judge puts a code executor inside the judge's RL loop and matches 96% of Claude-Opus-4's listwise performance with 8B parameters (Xu et al., 2026b). The difference is where the executor sits. TIR-Judge's executor helps the judge answer. Ours writes the judge's exam. To our knowledge none of them mines executed mutations, measures oracle tolerance, or tests cross-domain transfer.

Self-improving evaluators. Self-rewarding and meta-rewarding models judge their own outputs (Yuan et al., 2024; Wu et al., 2025) and self-taught evaluators learn from synthetic contrasts (Wang et al., 2024b); those corruptions are judged plausible by the generating model, ours are executed, and the difference matters most when the generator shares the judge's blind spots (Shumailov et al., 2024).

Judge and reward-model benchmarks. MT-Bench established LLM-as-a-judge against human votes (Zheng et al., 2023); LLMBar hand-curated 419 pairs with objective preferences and showed evaluators diverge on adversarially appealing wrong answers (Zeng et al., 2024); JudgeBench converts hard datasets into response pairs with objective correctness labels (Tan et al., 2025); RewardBench and RewardBench 2 evaluate reward models on prompt-completion sets, the second on mostly unseen human prompts in a best-of-four format (Lambert et al., 2025; Malik et al., 2026); RM-Bench tests sensitivity to subtle edits and style (Liu et al., 2025a); PPE ties reward-model metrics to post-RLHF preference (Frick et al., 2025), and ProcessBench asks judges to locate the first wrong step in 3,400 expert-annotated solutions (Zheng et al., 2025). Trained evaluators such as Prometheus 2 (Kim et al., 2024) and reasoning reward models such as RM-R1 (Chen et al., 2026) learn from feedback written by humans or stronger models, and JETTS evaluates judges as test-time evaluators on such labels (Zhou et al., 2025). Every one of these labels is asserted by a human, a source dataset, or a model; JudgeLM, RM-Bench and JudgeBench evaluate judges on curated sets, and JudgeBench's pairs come from model responses graded against a source answer, whereas JudgeGym generates verified pairs from an executable oracle by corrupting the derivation itself, and Section 4.5 places them side by side.

Behavioral testing, mutation, and contrast sets. CheckList's invariance and directional tests are our methodological ancestor (Ribeiro et al., 2020), and its invariance test is the direct precedent for our identity control, and counterfactually augmented data showed that minimal human edits sharpen a model's decision boundary (Kaushik et al., 2020). Mutation testing supplied the original insight that small syntactic corruptions of a program, executed rather than imagined, reveal what a test suite misses (DeMillo et al., 1978; Jia & Harman, 2011), and metamorphic testing replaced exact oracles with relations across executions (Segura et al., 2016). Both fields have lived with the equivalent-mutant problem that Section 4.3 meets in a new form. Verifier false positives cap what resampling can reach at any budget (Stroebl et al., 2026); Proposition 2 applies that ceiling to the matcher.

Failure taxonomies and domain anchors. MAST taxonomizes multi-agent failures into 14 modes (Cemri et al., 2025), and TRAIL's 841 annotated errors defeat even Gemini-2.5-Pro, at 11% joint accuracy (Deshpande et al., 2025). FinQA, ConvFinQA, and TAT-QA supply executable finance programs (Chen et al., 2021; 2022; Zhu et al., 2021) and FinanceBench the judgment-only

side (Islam et al., 2023); SARA is the executable legal core (Holzenberger et al., 2020), LexGLUE and CUAD its judgment-heavy shell (Chalkidis et al., 2022; Hendrycks et al., 2021).

F PROOFS

Proposition 1. Let $(p, \tilde{p})$ be a verified pair, so $M(O(\tilde{p}), O(p))$ is false, and let $O(p)$ agree with the released gold. The gold program is the accepted derivation of the released answer, so $O(p)$ is the correct answer to the question. M is the anchor's own notion of sameness, so $M(O(\tilde{p}), O(p))$ being false means $O(\tilde{p})$ is not that answer, and therefore wrong for the question. A judge that accepts $\tilde{p}$'s answer accepts a wrong answer, and one that rejects p's rejects the correct answer; both are wrong on the episode, and neither verdict needed a human label. The only way the conclusion fails is for $O(p)$ to disagree with the true answer, that is, for the gold to be wrong, which Assumption A2 bounds by the measured gold disagreement. □

Proposition 2. Let answers be drawn independently, correct with probability c and wrong with probability w, let M accept every correct answer, and let a fraction a of wrong answers be absorbed, meaning M accepts them. Each draw is accepted with probability $c + aw$ and is wrong given acceptance with probability $aw/(c + aw)$. Best-of-N selection that returns an accepted answer chosen uniformly among the accepted draws returns a wrong answer with probability $aw/(c + aw)$ whenever at least one draw is accepted, and the probability that none is accepted vanishes as N grows. Increasing N therefore cannot lower the wrong-answer probability below $aw/(c + aw)$, which is the ceiling of Stroebl et al. (2026) written for a verifier with completeness one. □

G MAGNITUDE RATIO QUANTILES BY FAMILY

Table 6 gives the quantiles of $|O(\tilde{p})|/|O(p)|$ on the 21,786 training pairs behind Figure 3.

Table 6: Quantiles of the corrupted-to-gold magnitude ratio per family on the training split, excluding the 251 pairs with a zero answer.

family	pairs	p10	p25	median	p75	p90
`swap_args`	5,229	0.07	1.00	1.00	4.41	51.1
`op_subst`	5,869	0.35	2.33	21.5	3.0×10^5	1.2×10^9
`digit_shift`	5,877	0.10	0.10	10.0	17.0	92.8
`cell_swap`	4,375	0.10	0.63	2.02	13.1	113
`const_break`	185	0.32	1.43	2.54	7.61	32.3