WITNESS BEFORE YOU JUDGE: WHERE TEST-TIME PROGRAM SELECTION FAILS AND HOW TO FIX IT

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time scaling for code generation samples many candidate programs and spends LLM calls to pick one. When the pick is wrong, did the judge misread the evidence, or was the evidence never found? We study selection through *witnesses*, inputs on which two candidates behave differently. On a development pool of LiveCodeBench tasks, a pairwise LLM judge picks the correct program in all 60 comparisons in which it is shown a witness, yet 41 comparisons end without one. Time-outs account for 18 of them, and for 14 of these the first failing hidden test is longer than the inputs a judge is shown. We therefore separate evidence construction from judgement. WITNESSFIRST asks the LLM to propose *objective* witnesses, constraint-scale inputs from model-written generators, checked by a model-written validator, on which execution alone rejects programs that crash or time out; a judge then decides the disagreements that remain, reading the code when no witness is found. These eliminations are sound whenever the validator is and execution is reproducible, so each disagreement with a benchmark label is an invalid input, an unreproducible outcome, or a gap in the benchmark's tests. Across eight held-out candidate pools from LiveCodeBench and CodeContests+ and four judges, placing the objective stage in front of each of four selectors improves it with open-weight judges, by 3.2 to 5.7 points under the benchmarks' labels and 5.2 to 9.3 under audited labels; behind it, a six-call vote does at least as well as a 36-call one, and with Qwen3-32B the full WITNESSFIRST is the most accurate of the main selectors. In our pools, 1.9% of the benchmark-correct LiveCodeBench programs fail on legal inputs, spanning at least 13 of the 342 problems. With a frontier judge, reading the code alone is the most accurate selector, and objective witnesses still help the selectors that do not read code.

1 INTRODUCTION

Sampling many programs and selecting one is a standard way to turn test-time compute into code-generation accuracy (Chen et al., 2021; Li et al., 2022; Brown et al., 2024; Snell et al., 2025). Candidates are filtered by the public examples of a task, clustered or voted on by their behaviour on generated inputs (Chen et al., 2023; Shi et al., 2022; Li et al., 2022), scored by trained verifiers (Cobbe et al., 2021; Ni et al., 2023; Inala et al., 2022), or compared pairwise by an LLM judge that sees generated tests and their outputs (Li et al., 2025). Papers report the accuracy of the whole pipeline. What they rarely report is why a selection goes wrong: whether the verifier made a wrong call, or never had the evidence to make the right one.

We study this question at the level of *witnesses*. A witness for two candidate programs is an input on which they behave differently; a verifier can only prefer one program over another for a reason that shows up on some input. An *objective* witness settles the comparison without judgement: a public example whose expected output one program fails to reproduce, or a valid input on which one program crashes or exceeds the time limit while the other finishes. A *subjective* witness is an input on which both programs finish but print different answers; deciding which is right requires a judge. The distinction suggests a principle that the rest of the paper tests: let the model *propose* evidence, let execution *check* what it can, and reserve judgement for what execution cannot settle.

Tracing knockout decisions on a development pool of 272 LiveCodeBench tasks (Jain et al., 2025) shows that they fail for want of evidence rather than judgement (Section 3). A pairwise judge was

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.

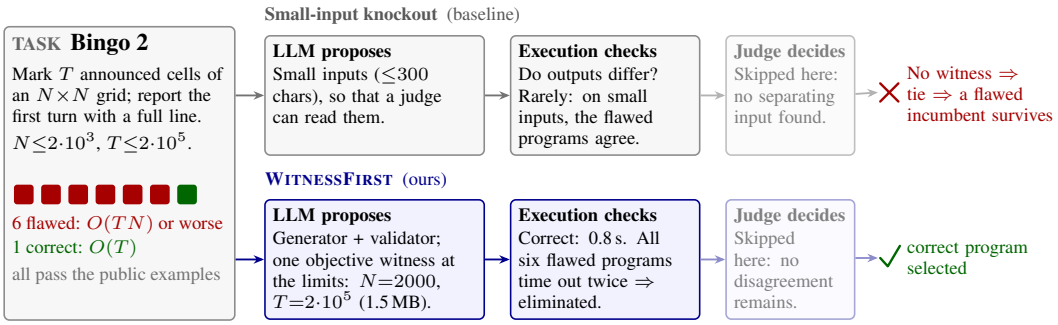

Figure 1: **Execution at the scale of the constraints eliminates failures that small-input search misses.** On a development task, six flawed public passers re-scan rows, columns, or the grid on every turn: right answers, but too slow at the constraint limits. *Top:* inputs small enough for a judge to read rarely separate them from the correct program, and in one knockout run the resulting tie kept a flawed incumbent. *Bottom:* one generated input at the constraint limits makes all six time out while the correct program finishes in 0.8 s, and execution eliminates them.

reliable whenever it had a witness: in a knockout with GPT-5.5, all 60 comparisons between a correct and a flawed program that were decided on a generated witness went to the correct program. Another 41 such comparisons ended without any separating input. For 14 of the 18 time-outs among them, the first hidden test that exposes the flawed program is longer than the 300 characters a judge is shown; the 18 wrong answers are exposed by short inputs that the generator never proposed.

This exposes a tension. A subjective witness must be small because a judge has to read it, but many bugs show up at the scale of the constraints. An objective witness can be as large as the constraints allow, because nobody needs to read it. WitnessFirst (Section 4) therefore asks the model to *construct* witnesses rather than to issue verdicts: generators of constraint-scale inputs and a validator derived from the statement, whose execution eliminates programs that crash or time out while others finish, before a pairwise knockout on small subjective witnesses that falls back to reading the code when none is found. Because execution rather than the model decides, these eliminations are sound whenever the validator is and execution is reproducible (Proposition 2), and every disagreement with a benchmark label becomes checkable: the input was invalid, the outcome did not reproduce, or the benchmark's tests miss a failure.

We evaluate on eight held-out candidate pools from LiveCodeBench v5/v6 and CodeContests+ with four judges, Qwen3 models from 4B to 32B parameters and GPT-5.5 (Section 5). Placed in front of each of the four selectors we test, the objective stage improves it: with the Qwen3 judges by 3.2 to 5.7 points under the benchmarks' labels and by 5.2 to 9.3 under audited labels, for about two extra calls, so that a six-call vote does at least as well as a 36-call vote. The best of the main selectors depends on the judge: the vote behind the objective stage with Qwen3-4B and Qwen3-8B, WitnessFirst with Qwen3-32B, where it turns the knockout from the weakest LLM-based selector into the most accurate one, and reading the code alone with GPT-5.5. The audit also shows that at least 1.9% of the programs that pass LiveCodeBench's hidden tests in our pools crash or time out on a legal input; they come from 13 of the 342 problems.

In summary, our contributions are:

- **A witness-level account of test-time program selection:** a lemma that makes the failure condition of knockout selection explicit, and a diagnosis showing that missing witnesses, not wrong verdicts, cause its errors (Sections 2–3).
- **WitnessFirst,** which uses the LLM to construct objective witnesses at the scale of the constraints; its eliminations are sound given a correct validator and reproducible execution, with no verdict from the model (Section 4).
- **A held-out evaluation and benchmark audit:** separating false eliminations from gaps in the benchmarks' tests, the audit finds such gaps in at least 13 of the 342 LiveCodeBench problems; at least two judges find gaps on 12 of them. The evaluation spans eight pools, two benchmarks,

and four judges, with 36-call baselines and ablations (Section 5). We release code, pools, and all traces.

2 SETUP: SELECTION, WITNESSES, AND WHERE SELECTION CAN FAIL

Selection. A task consists of a statement, a set $\mathcal{U}$ of valid inputs described by the statement's format and constraints, and a time limit τ. A pool $\mathcal{C} = \{c_1, \ldots, c_N\}$ of candidate programs is sampled once and frozen. Running c on $u \in \mathcal{U}$ yields an outcome $c(u)$ that is either an output, a crash, or a time-out. The selector sees the statement, the pool, and the public examples $T_0 = \{(u_j, y_j)\}$ with their expected outputs; hidden tests are used only to score the returned program. A program is *correct* if it reproduces every public example output and passes all hidden tests within τ, and *flawed* otherwise. The selector may call an LLM M that generates test inputs and judges comparisons; cost is the number of calls.

Definition 1 (Witness)**.** *An input $u \in \mathcal{U}$ is a witness for a pair (a, b) if $a(u) \neq b(u)$. It is* objective against a *if the verdict needs no judge: either u is a public input and $a(u)$ differs from its expected output, or a crashes or times out on u while b returns an output. Otherwise, when both return different outputs, it is* subjective*, and a judge must decide which output is correct.*

An objective witness is sound only if u is valid, since a correct program may crash on an input that the statement rules out (Section 4); a subjective witness is useful only if the judge can tell which output is right, which in practice requires an input small enough to reason about.

Public passers. Let $P \subseteq \mathcal{C}$ be the programs whose outputs match every public example. Every correct program is a public passer, so a task is solvable only if P contains a correct program. We call a solvable task *all-correct* if every member of P is correct and *mixed* otherwise. The strata use hidden labels and serve only for analysis.

Proposition 1 (Where verification can matter)**.** *Any selector that returns a member of P whenever $P \neq \emptyset$ is correct on every all-correct task, so on solvable tasks its errors are confined to mixed tasks.*

Knockout selection. A common way to spend judge calls on the public passers is a sequential knockout (Li et al., 2025; Chen et al., 2025). The first passer is the incumbent, and the remaining passers challenge it one at a time in their original order. A comparison ends either with a verdict on a witness or with a tie; a verdict drops the loser, and a tie keeps both programs and records the pair. The knockout ends when the incumbent has been compared with every surviving program. Call a comparison *discriminative* if exactly one of the two programs is correct.

Lemma 1 (Failure condition of knockout selection)**.** *Suppose every witnessed discriminative comparison is decided correctly. Then a knockout over P returns a flawed program if and only if the initial incumbent is flawed and every comparison between a flawed incumbent and a correct challenger ends without a witness.*

The proof (Appendix A) rests on the invariant that a correct incumbent is never replaced by a flawed one. The lemma separates two failure sources that aggregate accuracy conflates, wrong or inconsistent verdicts on witnessed comparisons and missing witnesses; on our development pool the first never occurs (Section 3).

3 DIAGNOSIS: SELECTION FAILS FOR WANT OF WITNESSES

Development pool. We use 272 LiveCodeBench tasks from release_v2 (108 stdin, 164 functional) on which at least one of eight Qwen2.5-Coder-7B-Instruct samples is correct. Hidden tests run in an isolated sandbox (6 s CPU, 512 MiB), and each flawed program is labelled by how it fails its first failing test: wrong answer, time-out, or crash. GPT-5.5 generates tests and judges. For each comparison it is asked, up to four times, for one small input (at most 300 characters for stdin tasks) that actually separates the two programs; the judge sees the task, both programs, the input, and both outputs in both orders, and must agree with itself or the comparison ties. This pool serves only for diagnosis and design; all conclusions about WITNESSFIRST are drawn on held-out tasks.

Public outputs do most of the work. Keeping the first candidate that reproduces all public example outputs selects a correct program on 251 of the 272 tasks, against 210 for majority clustering on the

Table 1: **The judge-based selectors err only on time-outs and crashes.** Failure types on the 54 mixed development tasks: flawed public passers; discriminative knockout comparisons without a witness, by the failure of the flawed program; tasks each selector (with GPT-5.5) gets wrong, by the failure of the program it returns.

	Flawed	Unwitnessed	Tasks selected wrongly		
Failure type	passers	comparisons	S*-style vote	Knockout	Code-only judge
Wrong answer	74	18	11	0	0
Time-out	26	18	4	2	0
Crash	11	5	3	2	3
Total	111	41	18	4	3

public inputs (Li et al., 2022); neither makes a model call. By the hidden labels of the public passers, 218 tasks are all-correct and 54 mixed. By Proposition 1, verification can change the outcome of a public-output-gated selector only on the 54 mixed tasks, and all remaining analysis concerns them.

Voting fails under correlated errors. Voting assumes that correct programs agree with each other and outnumber any single wrong behaviour, but flawed programs sampled from the same model often share a bug. A majority vote over a shared suite built with 15 generator calls, in the style of S* (Li et al., 2025), solves 29 of the 30 mixed tasks on which correct passers are the majority but only 3 of the 15 on which they are the minority, below the 4 of simply taking the first passer (Table 3 in Appendix B). Pairwise judging does not depend on the majority: a knockout with a judge and a code-only judge each solve 13 of those 15 tasks.

The judge is right when it has a witness. We label every knockout comparison with the hidden correctness of both programs: 60 discriminative comparisons were decided on a generated witness, and the judge picked the correct program in all 60. The other 41 ended without any separating input, so the comparison tied and the incumbent stayed. By Lemma 1, these unwitnessed comparisons account for every wrong selection of the knockout. The same holds for a variant that replays accepted witnesses across comparisons (Appendix B).

Unwitnessed bugs are scale bugs or search failures. Time-outs make up 23% of the flawed passers but 44% (18 of 41) of the unwitnessed comparisons, and crashes another 12% (Table 1). For the time-outs the failure is one of scale: in 14 of the 18, the first hidden test that exposes the flawed program is longer than the 300 characters allowed for a pair-specific input (median 293,340 characters), a mismatch between where the failure shows and where the generator searches. The 18 wrong answers are different: every one is exposed by a hidden test of at most 300 characters (median 6), so a readable witness existed and the generator did not find it. Accordingly, the S*-style vote mostly returns wrong answers, which correlated outputs outvote; the knockout returns time-outs and crashes that it never witnessed; and the code-only judge, which needs no witness and can recognise an inefficient algorithm by reading it, errs on three programs that crash.

> **Takeaway.** A judge that reads code can spot scale bugs, but its verdict comes with no evidence to check; a witness can be checked, but the witnesses a judge can read are usually too small for scale bugs. Objective witnesses resolve the tension, because an input on which one program crashes or times out while another finishes needs no reader. Wrong answers on small inputs remain a search problem for readable witnesses.

4 WitnessFirst: construct witnesses, then judge

WitnessFirst orders evidence by how much judgement it needs (Algorithm 1): public outputs first, then objective witnesses that the model constructs by reading the surviving programs, and only then subjective witnesses decided by a judge. The LLM M plays three roles, witness constructor, validator author, and judge, and every role is a separate, logged call.

Constructing objective witnesses. The model receives the statement and the distinct surviving programs, is asked which of them are likely to exceed the time limit or crash on inputs the public examples do not cover, and writes up to four deterministic generators, each printing one input designed to trigger such a failure; at least one must reach the constraint limits. Generators run in the

Algorithm 1 WITNESSFIRST. M is the LLM; τ the time limit; b the subjective-witness budget per comparison.

Require: statement, pool $\mathcal{C}$, public examples T_0, LLM M
$S \leftarrow \{c \in \mathcal{C} : c(u_j) = y_j \ \forall (u_j, y_j) \in T_0\}$; **if** $S = \emptyset$ **then** $S \leftarrow \mathcal{C}$ ▷ public witnesses
$G_1, \ldots, G_k \leftarrow M(\text{statement}, S)$; $V \leftarrow M(\text{statement})$ ▷ objective witnesses
for $i = 1, \ldots, k$ **do**
  $u \leftarrow G_i()$; **if** u fails to run or $V(u) \neq$ valid **then continue**
  $F \leftarrow \{f \in S : f$ returns an output on $u\}$; **if** $F = \emptyset$ **then continue**
  drop from S each program that crashes on u
  if some $f \in F$ ran in at most $\tau/2$ **then** drop from S each program that times out twice on u
$c^\star \leftarrow$ first member of S ▷ subjective witnesses
for each survivor $q \neq c^\star$ not yet tied with $c^\star$, in order **do**
  find a small input u with $c^\star(u) \neq q(u)$ in at most b requests to M
  if found **then** M judges $(c^\star, q)$ on u in both orders; the winner becomes $c^\star$ if they agree
  else a code-only comparison by M decides, and the decision is flagged as unwitnessed
return $c^\star$

candidates' sandbox and may print up to 16 MiB, far more than any judge could read. Reading the programs lets the model target the actual weakness, for example a nested loop over $n \leq 2 \cdot 10^5$ items; a program-agnostic variant sees only the statement, with its own generator instructions (Appendix E).

Validity and soundness. A crash on an invalid input is not evidence against a program. A second call therefore asks the model for a validator, a program that accepts an input if and only if it satisfies every format rule and constraint of the statement. The validator sees the statement, not the candidates or the generators, which limits direct dependence on them but not a shared misreading of the statement (Section 7); only accepted inputs are used. Two guards protect correct programs further: an input on which every survivor fails is discarded, and a time-out counts only if it recurs on re-execution while another survivor finished within half the limit. Call a program *semantically correct* if it returns the right output within τ on every valid input; a benchmark asks this only on its hidden tests.

Proposition 2 (Objective eliminations are sound)**.** *Suppose the validator accepts only valid inputs, and every execution of a program on an input has the same outcome (an output, a crash, or a time-out). Then, whatever generators the model writes, the objective stage removes only programs that are not semantically correct; in particular, every semantically correct program in the pool survives.*

The guarantee (proof in Appendix A) involves no judgement by the model, only two conditions that can be audited without it. It also turns a disagreement with the benchmark into a diagnosis: if the objective stage removes a program that passes every hidden test, then the validator accepted an invalid input, the outcome was not reproducible, or the hidden tests miss a failure; Section 5 checks the first two and attributes the rest to the tests.

Subjective witnesses and judging. The survivors enter the knockout of Section 2 in their original order. For each comparison, the model is asked for a small input on which the two programs differ, up to b times; the sandbox checks that the input actually separates them. The judge sees the task, both programs, the input, and both outputs, and is queried in both orders; disagreement counts as a tie, and a tie keeps the incumbent. If no separating input is found, WITNESSFIRST does not keep the incumbent by default: it asks the model to compare the two programs by reading them and flags the decision as unwitnessed, so that every selection reports how many of its eliminations were witnessed.

A judge-free filter. The objective stage never asks the model for a verdict: the model writes programs, and execution decides. It can therefore precede any selector, which separates the value of objective witnesses from that of the judge that follows them, and it costs two calls per task before retries, none when a single public passer remains (Appendix C).

5 EXPERIMENTS

We organise the evaluation around three questions:

Table 2: **The objective stage improves each selector it is placed in front of.** Mixed (pool, task) units solved (benchmarks' labels) over all held-out pools. Shaded rows put the objective stage in front of the selector above; WITNESSFIRST adds the code-only fallback to the knockout. Δ: a shaded row minus the nearest unshaded row above, an ablation minus WITNESSFIRST, in points over the Qwen3 judges; bold if the 95% bootstrap interval excludes zero (Table 9). *calls*: mean per mixed unit (Qwen3 judges). Bold counts: column best above the line; –: not run or incomplete.

		Qwen3 judges				Δ (points)	
Method	calls	4B	8B	32B	GPT-5.5	benchmark	audited
First passer	0	62	62	62	62		
Majority clustering	0	41	41	41	41		
Vote, 6 calls	6	70	68	73	78		
+ objective	8	**72**	**80**	79	85	**+5.7**	**+9.3**
Vote, 36 calls	36	**72**	74	76	–		
CodeT, 6 calls	6	68	68	72	86		
+ objective	8	70	73	78	93	+3.7	**+7.9**
CodeT, 36 calls	36	67	70	76	–		
Code-only judge	15	64	67	78	**108**		
+ objective	17	66	73	81	–	+3.2	**+5.2**
Knockout	25	61	64	65	102		
+ objective	25	62	70	70	–	+3.4	**+7.7**
WITNESSFIRST	42	65	76	**82**	106	**+9.5**	**+13.9**
WITNESSFIRST, agnostic generators	42	65	74	81	–	–0.9	–0.8
WITNESSFIRST, no validator	40	71	74	83	–	+1.4	**+2.7**
units n		116	116	116	116		

- **RQ1** (Section 5.1): does the objective stage improve every selector, and which suits each judge?
- **RQ2** (Section 5.2): are removed programs real failures, and what do they say about the tests?
- **RQ3** (Section 5.3): where do the gains come from, and what do they cost?

Benchmarks and pools. Our main benchmark is the 342 LiveCodeBench problems added in releases v5 and v6 (contests from September 2024 to April 2025; 217 stdin, 125 functional), none of which appear in the development pool. For each problem we sample 16 candidates from each of four open-weight generators with the prompt and sampling settings of the development pool: Qwen2.5-Coder-7B and -32B (Hui et al., 2024), DeepSeek-Coder-V2-Lite (Zhu et al., 2024), and Qwen3-8B (Yang et al., 2025). As a control with strong hidden tests we add 181 Codeforces problems whose tests come from CodeContests+ (Wang et al., 2025), built with problem-specific generators, validators, and accepted solutions; the same generators build pools on them (Appendix C). A program is correct if it reproduces every public example output and passes every hidden test. We evaluate on the mixed tasks of each pool, 116 (pool, task) units in total; by Proposition 1, every method except majority clustering, which ignores the expected outputs, is correct on all other solvable tasks. The core design of WITNESSFIRST was fixed on the development pool before any held-out run; Appendix C discloses two later changes that apply to every method.

Judges and baselines. Within a run one model writes generators and validators, proposes inputs, and judges. Three open-weight judges, Qwen3-4B, -8B, and -32B in non-thinking mode, run on every pool, and GPT-5.5, the development-pool model, runs on the mixed tasks as a strong reference. The selectors are the first public passer, majority clustering on public inputs (Li et al., 2022), an S*-style vote over a shared generated suite (Li et al., 2025), CodeT (Chen et al., 2023), the code-only judge, and the knockout whose ties keep the incumbent; we run the last four alone and behind the objective stage. The vote and CodeT use six generator calls, and also 36, close to the mean cost of WITNESSFIRST. Results are pooled over pools; because a problem appears in several pools, 95% intervals for differences come from a bootstrap over problems.

Audited labels. When the objective stage removes a program that the benchmark labels correct, a reference validator checks the eliminating input: one we wrote from the statement for each LiveCodeBench problem, and the official one for CodeContests+. Ours accept all public and hidden inputs, which tests acceptance of valid inputs rather than rejection of invalid ones. Given repro-

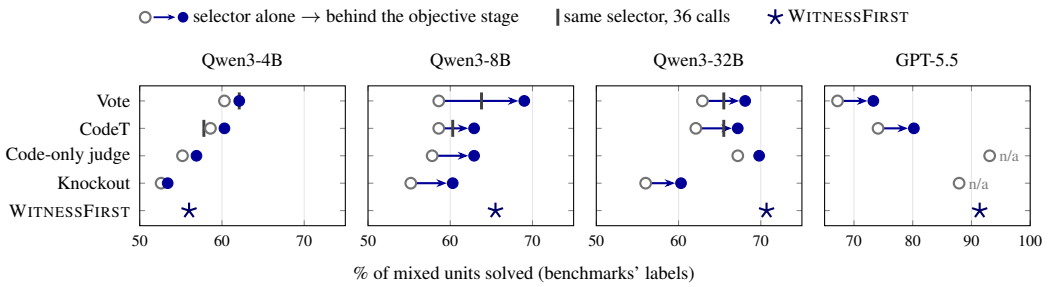

Figure 2: **The objective stage improves every selector in every reported combination.** Mixed units solved (%, benchmarks' labels, pooled as in Table 2); the GPT-5.5 panel has its own axis; n/a: not run or incomplete.

ducible execution, this tells an invalid input from a gap in the hidden tests (Proposition 2). A crash or time-out on a valid input is a failure whatever the expected output, so the audited labels count such programs as flawed; the labels for a judge's runs use only inputs found by the other judges, so no selector is scored with its own witnesses, and the mixed tasks are recomputed under each label set. This cannot rule out a bias shared by all generators or by our validators, so we report both label sets throughout.

5.1 OBJECTIVE WITNESSES IMPROVE EVERY SELECTOR

The gains are consistent and cheap. Put in front of any of the four selectors, the objective stage raises the number of mixed units solved (Table 2, Figure 2). Pooled over the three Qwen3 judges, every gain has an interval that excludes zero under the audited labels, but only the vote's does under the benchmarks' labels. The stage adds about two calls per task: with it, a six-call vote scores at least as high as a 36-call vote under the benchmarks' labels (+2.6 points, an interval that includes zero) and beats it under the audited labels (+5.1), and so does a six-call CodeT (+2.3 and +5.7).

The best partner depends on the judge. Under its premise, Lemma 1 locates the knockout's failures in unwitnessed comparisons; WITNESSFIRST removes scale bugs before any comparison and decides the comparisons that still lack a witness by reading the code. Together, filtering and fallback add 9.5 points to the knockout (13.9 under the audited labels) and 4.0 over the code-only judge, and WITNESSFIRST is the most accurate of the main selectors with Qwen3-32B. With Qwen3-4B and Qwen3-8B, whose generators rarely find a witness and whose verdicts often flip when the programs are swapped (Section 5.3), the vote behind the objective stage leads the main selectors, although some time-out and crash errors remain (Figure 3). Voting still fails where the development pool predicts: on the units whose correct programs are a minority of the public passers, the vote solves 27 of 111, fewer than the 30 of the first passer, and WITNESSFIRST solves 47.

Stronger judges read code well. With GPT-5.5, the code-only judge is the most accurate selector: it solves 108 of the 116 units under the benchmarks' labels, against 106 for WITNESSFIRST, and under the audited labels the interval of their difference includes zero. A frontier model often recognises a slow or wrong program by reading it. The objective stage still improves the selectors that do not read code, by 11.7 points for GPT-5.5's vote and 11.7 for its CodeT under the audited labels.

> **Takeaway.** The objective stage is a composable filter: it costs about two calls and improves every selector we tested. The best selector depends on the judge: the vote behind the objective stage for the two small judges, WITNESSFIRST for Qwen3-32B, and reading the code alone for GPT-5.5.

5.2 WHAT THE OBJECTIVE STAGE REMOVES

Most removed "correct" programs are real failures. The objective stage removes 17 of the 112 flawed passers that time out on the hidden tests with Qwen3-4B, 34 with Qwen3-8B, and 32 with Qwen3-32B, and 58 of 109 with GPT-5.5, together with 9 to 17 crashing programs; the program-agnostic variant, which sees only the statement, is within one point of WITNESSFIRST (Table 2).

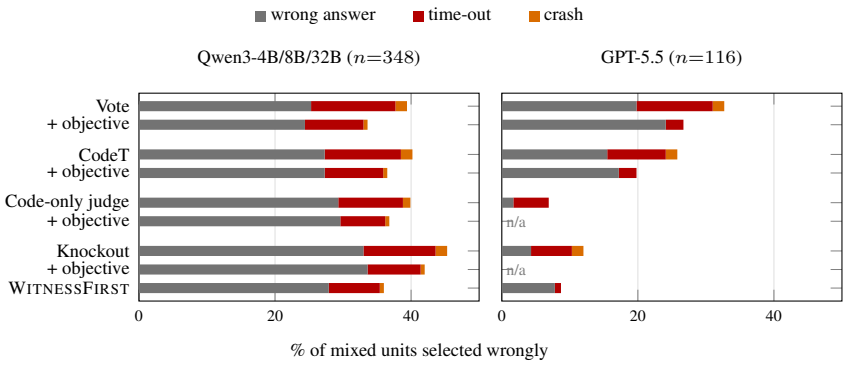

Figure 3: **The objective stage cuts time-out and crash errors; wrong answers change little or rise.** How the selected program fails on the hidden tests, in percent of all mixed (pool, judge, task) units: each selector alone and behind the objective stage (for the knockout, WitnessFirst without fallback), and WitnessFirst (n/a: not run or incomplete).

The stage also removes programs that the benchmarks label correct, and on 1 to 4 of the 116 mixed tasks it removes every correct passer (Appendix D). Across all solvable tasks, the reference validators confirm most such eliminations for the stronger models: they accept the eliminating input for 62 of 63 benchmark-correct programs removed with Qwen3-32B and 39 of 63 with Qwen3-8B. With Qwen3-4B the balance reverses (9 valid, 55 invalid): its validators accept inputs that violate the constraints, the case that Proposition 2 rules out by assumption. The validator buys safety rather than accuracy: without it, Qwen3-4B and Qwen3-8B remove 25 and 25 correct programs instead of 14 and 12, yet this variant solves 1.4 points more units, because the extra inputs also remove flawed programs and a pool usually keeps another correct one. We keep it because Proposition 2 and the audit rest on valid inputs.

The hidden tests miss scale bugs. Pooling the valid eliminations of all runs, at least 86 of the 4,645 programs that pass LiveCodeBench's hidden tests in our pools (1.9%) crash or time out on a legal input, against 2 of 318 (0.6%) on CodeContests+; they span 13 of the 342 LiveCodeBench problems and 1 of the 181 CodeContests+ ones, and for 12 of the 13 LiveCodeBench problems the objective stages of at least two judges found such an input in separate runs. In abc386_f, for instance, the longest hidden test has 947 characters, and benchmark-correct programs run out of memory on a valid 1 MB input (Appendix D). The contrast matches the tests: the longest hidden input of a LiveCodeBench problem has a median of about 4,900 characters and is shorter than 1,000 characters for 40% of the problems, whereas 79% of our CodeContests+ problems have a test input longer than 10,000 characters. Relabelling lowers pass@1 by at most 0.5 points. On the inputs that removed a program by time-out, the fastest program finishes in a median of 0.12 s (90th percentile 1.8 s) against the 6 s limit, and in 82% of them within a tenth of the limit; these margins support the fast-finisher guard, and the removals barely depend on the half-limit rule (Table 7).

> **Takeaway.** Objective eliminations are sound when inputs are valid and execution is reproducible: with Qwen3-32B nearly every removed benchmark-correct program fails on a valid input, which exposes scale bugs that LiveCodeBench's hidden tests miss.

5.3 Where the gains come from

Objective witnesses remove scale bugs. With the open judges, the objective stage removes about a third of the time-out and crash errors of every selector, for example from 14.1% to 9.2% of the units for the vote, and with GPT-5.5 three quarters or more; with the open judges, wrong answers stay near their level without it (Figure 3). The remaining errors of the open judges are mostly wrong answers, the search problem that objective witnesses do not address.

Subjective witnesses rarely materialise with open judges. Compared with the knockout, WitnessFirst wins 57 units and loses 24 with the Qwen3 judges. Of the wins, 18 come from objective eliminations and 39 from the code-only fallback on unwitnessed comparisons; of the losses, 7 come from false eliminations and 17 from wrong fallback decisions. The fallback matters because wit-

nesses are scarce: the Qwen3 generators find a separating small input for only 8% to 12% of the discriminative comparisons, against 45% for GPT-5.5, and removing the fallback costs 6.0 points. When a witness is found, GPT-5.5 judges it correctly in 187 of 192 comparisons, whereas the small judges mostly contradict themselves when the two programs are swapped, which counts as a tie (Appendix D).

Cost. WITNESSFIRST costs 42 calls per mixed task with the Qwen3 judges (Table 2); most of it goes to the knockout's witness search, which the open models seldom complete. Behind the objective stage, the code-only judge comes within 0.9 points of WITNESSFIRST at 17 calls, and the vote scores higher at about 8; both need fewer LLM calls when the judge is a small model.

6 RELATED WORK

Selecting among sampled programs. Repeated sampling raises the chance that a pool contains a correct program (Chen et al., 2021; Brown et al., 2024; Snell et al., 2025). Selectors cluster or vote on behaviour over public or generated inputs (Li et al., 2022; Chen et al., 2023; Shi et al., 2022; Liu et al., 2026), score candidates with trained verifiers (Cobbe et al., 2021; Ni et al., 2023; Inala et al., 2022; Zhang et al., 2023; Ma et al., 2025), or compare them pairwise on generated tests (Li et al., 2025); knockout tournaments scale when the judge beats chance on each comparison (Chen et al., 2025). Our lemma links knockout errors to witness-search failures when witnessed discriminative comparisons are decided correctly.

Judges, generated tests, and counterexamples. Pairwise LLM judging is standard for open-ended outputs (Zheng et al., 2023; Jiang et al., 2023) and has been applied to code without execution (Tong & Zhang, 2024); self-repair uses execution feedback to fix rather than to select programs (Chen et al., 2024; Olausson et al., 2024). LLM-written tests are often invalid and benchmark tests often weak (Liu et al., 2023), so recent pipelines build suites offline with generators, validators, and oracle solutions (He et al., 2025; Wang et al., 2025; Ridnik et al., 2024) or train test generators with reinforcement learning (Lee et al., 2026); our audit finds such gaps from the other side, with witnesses that a selector constructs at test time without a reference solution. Performance testing (Liu et al., 2024), worst-case fuzzing (Petsios et al., 2017; Lemieux et al., 2018), differential and property-based testing (McKeeman, 1998; Claessen & Hughes, 2000), and counterexample-guided synthesis (Solar-Lezama et al., 2006) also search for inputs that expose failures. Generators, validators, and stress tests are not new; what we add is their use at test time without a reference solution, as a judge-free filter that composes with any selector, and an account of selection at the level of witnesses. Objective witnesses make verification less dependent on the verifier's strength, a concern shared with scalable oversight (Burns et al., 2024).

7 LIMITATIONS AND CONCLUSION

Limitations. We condition on pools containing a correct program, and objective witnesses depend on the time limit and hardware. A validator written by the selector's model can share its misreading of the statement: with Qwen3-4B most removed correct programs are false eliminations. The audit relies on our reference validators and 6 s limit. The judges may have seen these public problems in training, which could favour code-only judging. CodeContests+ has few mixed tasks and serves as a control, and 10% of the Qwen2.5-Coder-32B samples on LiveCodeBench reach the token limit.

Conclusion. On the development pool, knockout errors arise from missing witnesses, either because the bug appears at the scale of the constraints or because a small witness exists and the generator does not find it; held-out traces also contain incorrect witnessed verdicts. WITNESSFIRST asks the model to construct witnesses and eliminates programs on objective evidence before any judgement. On held-out pools, objective witnesses improve every selector they precede, make the knockout, together with a code-only fallback, the most accurate of the main selectors with Qwen3-32B, and audit the benchmark labels themselves, exposing programs that pass the hidden tests yet fail on legal inputs. The broader lesson is to ask a model for evidence that a trusted procedure can check, and to reserve its judgement for what cannot be checked; whether this helps LLM judges and reward models beyond code is left to future work.

AI USE STATEMENT

We used LLM-based coding assistants to help implement and modify parts of the experimental code; all code was reviewed and tested by the authors. LLMs are also components of the studied methods, where they generate test inputs, input generators and validators, and act as judges (Sections 3 to 5). The authors take full responsibility for the content.

ETHICS STATEMENT

This work studies how to select among automatically generated programs. All candidate programs are executed in a network-isolated sandbox with CPU, memory, and file-descriptor limits. We use public benchmark problems and open-weight models; no human subjects or personal data are involved.

REPRODUCIBILITY STATEMENT

We release the candidate pools, hidden-test labels with per-test failure types, all selection traces (every model call with its prompt, response, and token counts), the sandbox protocol, and the scripts that build every table from these traces. Algorithm 1 and Appendix E give the full procedure and prompts; Appendix C lists sampling settings, time limits, and budgets.

REFERENCES

Bradley Brown et al. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Collin Burns et al. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Bei Chen et al. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023.

Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. Provable scaling laws for the test-time compute of large language models. In *Advances in Neural Information Processing Systems*, 2025.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 268–279, 2000.

Karl Cobbe et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

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. *arXiv preprint arXiv:2505.24098*, 2025.

Binyuan Hui et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Jeevana Priya Inala et al. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*, 2022.

Naman Jain et al. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. LLM-Blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, Toronto, Canada, 2023.

Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning. In *International Conference on Learning Representations*, 2026.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 254–265, 2018.

Dacheng Li et al. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, Suzhou, China, November 2025. Association for Computational Linguistics.

Yujia Li et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *Conference on Language Modeling*, 2024.

R.-B. Liu, J.-T. Xue, C.-Z. Ma, H. Sun, X.-Y. Li, and M. Li. Dynamic-static synergistic selection method for candidate code solutions with generated test cases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 32096–32104, 2026. doi: 10.1609/aaai.v40i38.40481.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

Ansong Ni et al. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*, 2024.

Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2155–2168, 2017.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3533–3546, Abu Dhabi, United Arab Emirates, 2022.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 404–415, 2006.

Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

Zihan Wang et al. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025.

An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Lianmin Zheng et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

A PROOFS

Proposition 1. A correct program reproduces every public example output, so it belongs to P. A solvable task therefore has $P \neq \emptyset$ and is either all-correct or mixed. On an all-correct task every member of P is correct, so any member the selector returns is correct. □

Lemma 1. Let c_0 be the initial incumbent and c_t the incumbent after the t-th comparison. We first show that a correct incumbent stays correct. Suppose c_t is correct and faces challenger q. If q is correct, both outcomes leave a correct incumbent. If q is flawed, the comparison is discriminative: if a witness is found, the comparison is decided for c_t by assumption; otherwise it ties and c_t stays. Hence c_{t+1} is correct, and by induction every later incumbent is correct.

($\Leftarrow$) If c_0 is flawed and every comparison of a flawed incumbent against a correct challenger is unwitnessed, then each such comparison ties and keeps the flawed incumbent, while comparisons of a flawed incumbent against a flawed challenger leave a flawed incumbent whatever the outcome. The incumbent is therefore flawed after every comparison, including the last.

($\Rightarrow$) If c_0 is correct, the returned program is correct by the invariant. If some comparison of a flawed incumbent c_t against a correct challenger q finds a witness, it is decided for q by assumption, so $c_{t+1} = q$ is correct and, by the invariant, so is the returned program. Hence a flawed return requires a flawed c_0 and no witnessed comparison of a flawed incumbent against a correct challenger. □

The lemma assumes that ties keep the incumbent, as in the knockout of Section 3. Without its premise, a witnessed comparison can also tie when the judge contradicts itself across the two orders, which keeps a flawed incumbent. WITNESSFIRST replaces some ties with a code-only decision, which can turn an unwitnessed comparison into a correct or an incorrect elimination; Section 5 reports how often such decisions occur and how often they are right.

Proposition 2. Let the objective stage remove a program c on an input u. The validator accepted u, so u is valid. The stage removes c only if c crashed on u, or timed out on u on two executions. By the reproducibility assumption, this is the outcome of every execution of c on u, so c does not return an output within τ on the valid input u and is not semantically correct. A semantically correct program returns an output within τ on every valid input and is therefore never removed; if the pool contains one, it survives. □

The two guards of the objective stage, discarding inputs on which every survivor fails and requiring a fast finisher for a time-out, are not needed for the proof. They limit the damage when its assumptions fail: an invalid input on which every program fails is discarded, and a program that is merely slow near the limit is spared unless another survivor finishes within half the limit.

Table 3: Tasks solved on the mixed tasks, grouped by the share of correct programs among the public passers (hidden labels, analysis only).

Correct passers are	minority (n=15)	half (n=9)	majority (n=30)
First public passer	4	4	25
S*-style vote, 15-call suite	3	4	29
Knockout, judge on witnesses	13	7	30
Code-only judge	13	8	30

B Development pool: additional analyses

A knockout with witness reuse. We also traced a variant of the knockout that replays previously accepted witnesses before generating new ones and, when a new witness separates programs that had behaved identically so far, splits them into groups that are then compared group against group. With GPT-5.5 it decided 45 discriminative comparisons on a witness, all correctly, and left 16 unwitnessed; in 14 of these the flawed program times out on the hidden tests and in the other 2 it crashes. In this variant no unwitnessed comparison involves a wrong-answer bug; what remains are almost exclusively scale bugs.

The objective stage on the development pool. Of the 54 mixed development tasks' flawed public passers, 26 flawed public passers fail the hidden tests by timing out. With program-aware construction the objective stage removes 19 of them when GPT-5.5 writes the generators, 9 with Qwen3-32B, 4 with Qwen3-8B, and 1 with Qwen3-4B; program-agnostic construction removes 18, 13, 5, and 10. The two prompts differ in code access and in the generators and input kinds they request (Appendix E), so this comparison does not isolate the effect of reading the candidates. No open model removes a correct program on the development pool. GPT-5.5 removes three, all in one task, on a valid worst-case input on which they exceed the time limit while another program finishes in under 0.1 s: performance bugs that the hidden tests do not exercise.

C Experimental setup details

Candidate pools. LiveCodeBench held-out problems are the 342 problems first released in v5 and v6 (217 stdin, 125 functional). Every pool uses the prompt of the development pool, which asks for a single Python block that reads standard input or completes the given class, and samples 16 programs per problem with temperature 0.7 and top-p 0.95 using vLLM with bfloat16 weights and seed 0. Four generators build LiveCodeBench pools: Qwen2.5-Coder-7B-Instruct and Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024), and Qwen3-8B (Yang et al., 2025) in non-thinking mode, each generating at most 2,048 new tokens. The same generators also build CodeContests+ pools with the same prompt and settings.

Second benchmark: CodeContests+. We take the 282 Codeforces problems of the CodeContests test and validation splits (Li et al., 2022) (contests 1548 to 1623), of which 276 appear in CodeContests+ (Wang et al., 2025). Statements and public examples come from CodeContests. Hidden tests come from the 1x release of CodeContests+, whose inputs are written by problem-specific generators, checked by validators, and labelled with accepted solutions: a median of 25 tests per problem, and 79% of the problems have a test input longer than 10,000 characters, whereas the longest CodeContests test input on these problems has 503. Our executor compares outputs exactly after removing trailing whitespace, so we keep only problems whose official checker accepts a single answer up to whitespace (the standard integer, token, and line comparators of testlib). We drop problems with custom checkers (65), floating-point or case-insensitive yes/no checkers (27), and three whose statement allows any valid answer although CodeContests+ lists a standard comparator. This leaves 181 problems with Codeforces ratings from 800 to 3500 (quartiles 1300, 2000, 2600). As a check of the labels, the first Python 3 solution that CodeContests marks as accepted passes every test of 33 of 40 randomly chosen problems under our protocol; the other seven time out on inputs of 38 thousand to 5.7 million characters (5) or crash (2, one of them on the problem's own example), and none returns a different output.

Hidden-test labels. Every candidate is run twice on all hidden tests of its problem in a network-isolated sandbox (6 s CPU, 6.2 s wall clock, 512 MiB address space, 32 file descriptors), stopping at the first failing test. Candidates whose two runs disagree are re-run with fewer parallel jobs; a label is kept only when both runs agree. Because runs stop at the first failing test, a flawed candidate is labelled by how it fails that test: a different output (wrong answer), an exception or abnormal exit (crash), or exceeding the time limit (time-out). Following LiveCodeBench, a candidate is correct only if it also reproduces every public example output, checked with the selector's own public filter and confirmed by a second run. Hidden tests do not always contain the public examples: on the development pool 16 candidates in 12 tasks pass every hidden test but fail a public example, and they are labelled flawed, which leaves 4 tasks without a correct candidate. Table 4 gives the held-out counts per pool. Strata are computed from the public passers of this gate. The public filter of a selection run can disagree with it when a program runs close to the time limit on a public example; this happened in 13 of 2,050 (task, run) pairs, on 3 tasks, and we keep the gate's strata throughout.

Table 4: Candidate pools: samples, samples cut off at the token limit, samples without a code block, candidates relabelled by the public gate, solvable and mixed tasks, mixed tasks whose correct passers are a minority, and hidden-test failure types of the flawed public passers on mixed tasks (WA, T, C: wrong answer, time-out, crash).

								flawed passers		
Pool	samples	cut off	no code	gated	solvable	mixed	minority	WA	T	C
LiveCodeBench v5/v6										
Qwen2.5-Coder-7B	5,472	16	0	24	106	18	7	48	2	1
Qwen3-8B	5,472	257	74	48	117	33	12	87	29	0
DeepSeek-Coder-V2-Lite	5,472	3	0	21	100	13	5	49	9	2
Qwen2.5-Coder-32B	5,472	531	0	16	132	36	8	89	52	0
CodeContests+										
Qwen2.5-Coder-7B	2,896	9	0	5	10	3	1	4	5	0
Qwen3-8B	2,896	233	213	2	22	7	2	10	11	0
DeepSeek-Coder-V2-Lite	2,896	1	0	0	13	3	2	6	2	3
Qwen2.5-Coder-32B	2,896	1,037	1	0	14	3	0	5	0	0

Judges. The selector model plays all roles within a run. Open-weight models are served with vLLM in non-thinking mode (context 32,768 tokens for Qwen3-4B and Qwen3-8B, 20,480 for Qwen3-32B). The judge answers with a single token at temperature 0 and is asked twice with the programs in swapped positions; a comparison is decided only if both answers agree. A witness is shown to the judge only if the input and both outputs together fit in 20,000 characters; a longer witness cannot be read and counts as a tie, and the search continues. Pair-specific input generation uses at most 400 tokens; each of the $b{=}4$ requests per comparison may make up to three proposals, and a proposal counts only if the sandbox confirms that it separates the pair. Pair-specific inputs and shared suites are sampled at temperature 0.7 with the open-weight models, whose recommended non-thinking setting this is; at the temperature of 0.2 requested from GPT-5.5 on the development pool, vLLM returns the same input for most repeated requests, so the retries would add nothing. The objective-stage calls decode at temperature 0, like the judges.

Objective stage. One call writes at most four generators, at least one targeting the constraint limits; generators are asked to print one input within 5 s (enforced at 6 s) and may print at most 16 MiB. A second call writes the validator from the statement alone; a response that cannot be parsed is requested once more. In the knockout, a comparison costs at most $3b$ generator calls (each of the b requests may retry twice), two judge calls per witness, and two calls for the fallback; objective eliminations shorten the knockout, and tasks with a single public passer need no calls. Candidates are run on each accepted input under the hidden-test limits. A crash eliminates a program if at least one other survivor returns an output. A time-out, including a kill at the CPU limit, eliminates it only if it recurs on re-execution and another survivor finished within half the limit. Inputs on which all survivors fail are discarded.

Baselines. The first public passer and majority clustering on public inputs make no calls. The S*-style vote runs all public passers on inputs accumulated from six generator calls and returns the first program whose outputs agree most often with the others, as in the majority mode of S* (Li et al.,

2025). CodeT (Chen et al., 2023) uses the same six calls, each asking for five new tests with their expected outputs; it groups the public passers by the set of generated tests they pass and returns the lowest-indexed program of the group with the largest product of group size and number of passed tests. Tests are deduplicated as input–output pairs, so the same input with two different expected outputs counts as two tests, as in the original method. As higher-call baselines we also run the vote and CodeT with 36 generator calls, close to the mean number of LLM calls of WITNESSFIRST with open-weight judges. The code-only judge and the knockout with subjective witnesses use the knockout of Section 2; the former sees only the task and the two programs. The compositions run the objective stage of WITNESSFIRST (program-aware, validated) and then the code-only judge, the vote, CodeT, or the knockout on the surviving programs; the last is WITNESSFIRST without its code-only fallback. They reuse the objective stage of the same run, so they add only the calls of the second selector. CodeT and its composition were added after the first held-out runs.

Auditing eliminations. When the objective stage removes a program that the benchmark labels correct, we check whether the input that removed it satisfies the problem's input constraints. For LiveCodeBench problems we wrote a reference validator per problem from the statement alone; each accepts every public and hidden input of its problem. For CodeContests+ problems we compile the official testlib validators, after normalising trailing whitespace; 178 of the 181 accept every official input, and the remaining three reject all inputs because their patterns use syntax that the current testlib does not support, so eliminations on those problems stay unaudited. A crash or time-out on a valid input is a failure regardless of the expected output, so for the audited labels we relabel such programs as incorrect and recompute every selector's result. In the leave-one-judge-out variant, the labels used to score a judge's runs are built only from inputs found in runs of the other judges, so no selector is scored with inputs it generated itself.

Statistics. Within one pool and judge, paired differences in tasks solved use exact two-sided McNemar tests on discordant tasks, and differences in calls use 95% intervals from 5,000 paired bootstrap resamples of tasks. Pooled comparisons count each (pool, judge, task) selection once; because the same problem appears in several pools, their 95% intervals come from 10,000 bootstrap resamples of problems (cluster bootstrap), and the McNemar p values are reported for reference only.

Changes after the first held-out runs. The prompts and core settings of WITNESSFIRST (including $b{=}4$, the half-limit rule, and the 16 MiB cap) were fixed on the development pool and not changed. After the first held-out selection runs we made two changes that apply to every method: open-weight models sample generated inputs at temperature 0.7 rather than 0.2 (see Judges), and a judge answer that violates the single-token format counts as a tie, like a disagreement between the two orders (10 answers in all held-out runs). We also added comparisons: the objective stage composed with the vote, CodeT and its composition, the 36-call vote and CodeT, the Qwen2.5-Coder-32B and DeepSeek-Coder-V2-Lite pools, the CodeContests+ pools, the GPT-5.5 reference on all pools, and the audit of eliminations. For GPT-5.5, the composition of the objective stage with the code-only judge is not reported: on two CodeContests+ tasks the public filter admitted a different set of programs than the logged objective stage, and the runner refuses such mismatches. Every added comparison uses the same pools, tasks, and labels as the original ones.

D ADDITIONAL HELD-OUT RESULTS

Tables 8 and 13 split Table 2 by benchmark and by pool. Table 9 gives all pooled paired differences, including those for GPT-5.5 and for the objective stage placed in front of other selectors. Table 5 details what the objective stage removes, Table 6 how the valid eliminations relabel each pool, and Table 7 how the removals depend on the fast-finisher threshold of the time-out rule. Table 10 traces the discriminative comparisons of the knockout, Table 11 repeats the minority analysis of Section 3 on the held-out pools, and Table 12 compares the benchmarks' labels with the audited ones. Figure 4 plots accuracy against cost.

Timing. The time-out rule removes a program only if it times out twice while another survivor finishes within half the limit. Recomputing the removals offline from the logged executions, moving this fast-finisher threshold from its default to anywhere between a quarter of the limit and the full limit changes the number of removed benchmark-correct programs by at most 2 and the number of tasks that lose every correct passer by at most 1 for every judge, while the number of flawed programs removed moves by at most 5 (Table 7).

Table 5: The objective stage (program-aware construction) on mixed tasks. Flawed and correct public passers removed by a time-out (T) or crash (C); time-out recall over flawed passers that time out on the hidden tests; *lost*: tasks on which every correct passer was removed. The last two columns repeat the counts for inputs used without the validator. Correct follows the benchmark labels; Section 5.2 audits the removed correct programs.

		flawed removed			correct removed			no validator	
Judge	tasks	T	C	T recall	T	C	lost	correct	lost
development pool									
Qwen3-4B	54	1	1	1/26	0	0	0	4	0
Qwen3-8B	54	5	1	4/26	0	0	0	0	0
Qwen3-32B	54	9	1	9/26	0	0	0	0	0
GPT-5.5	54	21	4	19/26	3	0	0	3	0
held-out pools, both benchmarks									
Qwen3-4B	116	18	9	17/112	2	12	1	25	2
Qwen3-8B	116	28	16	34/112	11	1	1	25	2
Qwen3-32B	116	35	11	32/112	20	14	4	35	4
GPT-5.5	116	61	17	58/109	16	15	4	31	4

Table 6: Benchmark labels against valid objective eliminations pooled over all runs (a union, unlike the leave-one-judge-out labels of Table 2), per pool: samples, samples that pass the benchmark's public and hidden tests (*correct*), those among them that some run removed on an input the reference validator accepts (*relabelled*, a lower bound, since the objective stage runs only on tasks with at least two public passers), the problems they belong to, and pass@1 under both label sets. Across pools, the relabelled programs belong to 13 LiveCodeBench problems (3612, abc374_c, abc378_d, abc381_d, abc386_c, abc386_f, abc393_d, abc395_c, abc397_c, abc397_e, abc398_c, abc399_f, arc195_a) and 1 CodeContests+ problem (cf1582_G).

					pass@1 (%)	
Pool	samples	correct	relabelled	problems	benchmark	audited
LiveCodeBench v5/v6						
Qwen2.5-Coder-7B	5,472	987	17	6	18.0	17.7
Qwen3-8B	5,472	1,259	14	6	23.0	22.8
DeepSeek-Coder-V2-Lite	5,472	962	27	5	17.6	17.1
Qwen2.5-Coder-32B	5,472	1,437	28	7	26.3	25.7
CodeContests+						
Qwen2.5-Coder-7B	2,896	39	0	0	1.3	1.3
Qwen3-8B	2,896	133	0	0	4.6	4.6
DeepSeek-Coder-V2-Lite	2,896	82	0	0	2.8	2.8
Qwen2.5-Coder-32B	2,896	64	2	1	2.2	2.1

Case studies. The audit separates two kinds of removed "correct" programs. In abc386_f (two strings of length up to $5 \cdot 10^5$, at most 20 edits), benchmark-correct programs of the Qwen2.5-Coder-7B pool fill a dynamic-programming table over both strings. The longest hidden test has 947 characters, so they pass; on a valid 1 MB input written by the objective stage they run out of memory, while another program finishes in under 0.1 s. The objective stages of Qwen3-4B, Qwen3-32B, and GPT-5.5 all found such an input. In abc390_b, by contrast, the generator of Qwen3-8B produced a sequence of length 1, which the statement rules out; its validator accepted the input, and 8 correct programs that read the second element crashed. In abc395_e the statement guarantees that vertex N is reachable, a semantic constraint that none of the non-thinking Qwen3 validators checked; generated graphs that violate it make correct programs run until the time limit.

Table 7: Sensitivity of the objective stage to the fast-finisher threshold of the time-out rule (default $\tau/2$), recomputed offline from the logged executions on the official mixed tasks: flawed / benchmark-correct programs removed / tasks that lose every correct passer. Crash eliminations do not depend on the threshold. Below $\tau/2$ the counts are lower bounds, because programs removed in the original run were not executed on later inputs.

Judge	tasks	$\tau/4$	$\tau/2$	$3\tau/4$	τ
Qwen3-4B	116	27/14/1	27/14/1	27/14/1	27/14/1
Qwen3-8B	116	42/12/1	44/12/1	44/12/1	44/12/1
Qwen3-32B	116	43/34/4	46/34/4	46/34/4	50/36/4
GPT-5.5	116	75/30/4	78/31/4	78/31/4	83/32/5

Table 8: Mixed units solved by benchmark and judge, pooled over the pools of each benchmark. First: first passer; Maj.: majority clustering; Code: code-only judge; KO: knockout; bold: best in the row.

Judge	n	First	Maj.	Vote	CodeT	Code	KO	WITNESSFIRST
LiveCodeBench v5/v6, 4 pools								
Qwen3-4B	100	55	36	**62**	61	57	54	58
Qwen3-8B	100	55	36	58	61	58	56	**62**
Qwen3-32B	100	55	36	62	62	65	58	**68**
GPT-5.5	100	55	36	66	71	**92**	87	90
CodeContests+, 4 pools								
Qwen3-4B	16	7	5	**8**	7	7	7	7
Qwen3-8B	16	7	5	10	7	9	8	**14**
Qwen3-32B	16	7	5	11	10	13	7	**14**
GPT-5.5	16	7	5	12	15	**16**	15	**16**

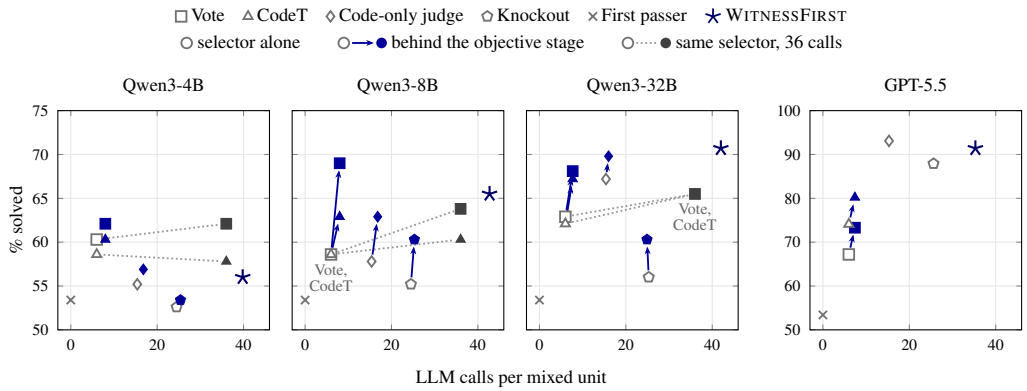

Figure 4: **Accuracy against cost.** Mixed units solved (%, benchmarks' labels) against mean LLM calls per mixed unit, pooled over all held-out pools as in Table 2; the Qwen3 judges share a vertical axis and GPT-5.5 has its own. Missing points were not run or are incomplete: for GPT-5.5, the 36-call selectors and the objective stage in front of the code-only judge and the knockout.

Table 9: Pooled paired differences in percentage points (row method minus the method it is compared with; WF: WITNESSFIRST), with 95% intervals from a bootstrap over problems; bold: interval excludes zero. *Audited* uses the leave-one-judge-out audited labels. Top block: the objective stage in front of each selector and against the 36-call baselines; middle: WITNESSFIRST against the baselines; bottom: ablations. –: not run (GPT-5.5 with the 36-call baselines, the knockout without fallback, and the no-validator ablation), or not covering all mixed units (GPT-5.5 with the code-only judge behind the objective stage and with agnostic generators; Appendix C), following the rule of Table 2.

	Qwen3 judges (4B, 8B, 32B)		GPT-5.5	
Comparison	benchmark	audited	benchmark	audited
Obj.+Vote vs. Vote	**+5.7** [1.5, 10.3]	**+9.3** [5.1, 14.0]	+6.0 [−0.9, 13.6]	**+11.7** [4.7, 19.6]
Obj.+CodeT vs. CodeT	+3.7 [−0.3, 8.2]	**+7.9** [3.6, 12.9]	+6.0 [−1.0, 13.6]	**+11.7** [4.6, 19.4]
Obj.+Code vs. Code	+3.2 [0.0, 6.7]	**+5.2** [2.5, 8.2]	–	–
Obj.+KO vs. KO	+3.4 [−0.3, 7.4]	**+7.7** [3.9, 11.9]	–	–
WF vs. KO	**+9.5** [3.3, 15.6]	**+13.9** [7.7, 20.4]	+3.4 [−3.2, 10.3]	**+8.1** [1.8, 14.9]
Obj.+Vote vs. Vote@36	+2.6 [−1.4, 6.7]	**+5.1** [1.2, 9.3]†	–	–
Obj.+CodeT vs. CodeT@36	+2.3 [−2.1, 7.0]	**+5.7** [1.8, 10.1]†	–	–
WF vs. First	**+10.6** [3.7, 17.6]	**+14.2** [7.2, 21.2]	**+37.9** [28.4, 47.5]	**+43.2** [34.3, 52.2]
WF vs. Maj.	**+28.7** [20.5, 37.4]	**+29.8** [21.5, 38.2]	**+56.0** [46.6, 65.3]	**+60.4** [51.6, 69.2]
WF vs. Vote	+3.4 [−3.9, 11.4]	**+8.5** [1.0, 16.3]	**+24.1** [15.1, 33.3]	**+30.6** [22.0, 39.6]
WF vs. CodeT	+4.3 [−2.5, 11.3]	**+9.8** [2.5, 17.3]	**+17.2** [8.1, 26.7]	**+23.4** [14.7, 32.7]
WF vs. Code	**+4.0** [0.3, 8.0]	**+5.7** [2.6, 9.2]	−1.7 [−7.2, 3.5]	+3.6 [−0.9, 8.7]
WF vs. Vote@36	+0.3 [−7.1, 8.3]	+2.7 [−4.9, 10.7]†	–	–
WF vs. CodeT@36	+2.9 [−3.9, 9.9]	+6.3 [−0.9, 13.7]†	–	–
WF vs. Obj.+Vote	−2.3 [−8.5, 4.3]	−0.8 [−7.0, 5.4]	**+18.1** [11.1, 25.4]	**+18.9** [11.6, 26.5]
WF vs. Obj.+Code	+0.9 [−1.5, 3.3]	+0.5 [−1.6, 2.9]	–	–
WF agn. vs. WF	−0.9 [−3.9, 2.1]	−0.8 [−3.4, 1.6]	–	–
WF no val. vs. WF	+1.4 [−1.1, 4.3]	**+2.7** [0.3, 5.6]	–	–
Obj.+KO vs. WF	**−6.0** [−11.2, −0.9]	**−6.3** [−11.2, −1.2]	–	–
units n	348	366	116	111

† On the units both methods cover (Qwen3 audited: $n = 333$); the 36-call baselines were run on the official mixed tasks only.

Table 10: Discriminative comparisons, pooled over pools, from two separate traces: those of the knockout, with how many had a witness and how the judge decided them, and the unwitnessed ones of WITNESSFIRST, with how its code-only fallback decided them. A tie means the two orders disagreed or a verdict violated the answer format.

		witnessed				code fallback		
Judge	disc.	n	right	wrong	tie	right	wrong	tie
LiveCodeBench v5/v6, 4 pools								
Qwen3-4B	348	27	0	1	26	9	0	310
Qwen3-8B	352	29	11	3	15	47	31	245
Qwen3-32B	353	46	24	10	12	83	40	159
GPT-5.5	388	168	163	5	0	108	0	64
CodeContests+, 4 pools								
Qwen3-4B	45	5	0	0	5	0	0	36
Qwen3-8B	44	2	2	0	0	10	4	25
Qwen3-32B	45	1	0	0	1	22	2	23
GPT-5.5	39	24	24	0	0	8	0	1

Table 11: Held-out mixed units solved / in the group, grouped by the share of correct programs among the public passers.

	Qwen3 judges			GPT-5.5		
Correct passers are	minority	half	majority	minority	half	majority
First	30/111	24/42	132/195	10/37	8/14	44/65
Vote	27/111	27/42	157/195	10/37	10/14	58/65
CodeT	35/111	22/42	151/195	17/37	12/14	57/65
Code	40/111	27/42	142/195	34/37	12/14	62/65
KO	33/111	24/42	133/195	32/37	12/14	58/65
WITNESSFIRST	47/111	30/42	146/195	34/37	13/14	59/65

Table 12: Mixed units solved under the benchmarks' labels / the audited (leave-one-judge-out) labels. The audit relabels benchmark-correct programs that crash or time out on a valid input, which changes both the set of mixed units (n) and the outcomes. Abbreviations as in Table 8.

Judge	n	Vote	CodeT	Code	KO	WITNESSFIRST
LiveCodeBench v5/v6, 4 pools						
Qwen3-4B	100/106	62/62	61/61	57/58	54/55	58/61
Qwen3-8B	100/106	58/58	61/60	58/64	56/56	62/69
Qwen3-32B	100/104	62/60	62/59	65/66	58/56	68/73
GPT-5.5	100/95	66/59	71/64	92/85	87/81	90/89
CodeContests+, 4 pools						
Qwen3-4B	16/17	8/8	7/7	7/8	7/7	7/8
Qwen3-8B	16/16	10/10	7/7	9/9	8/8	14/14
Qwen3-32B	16/17	11/11	10/10	13/14	7/7	14/15
GPT-5.5	16/16	12/12	15/15	16/16	15/15	16/16

Table 13: Mixed tasks solved per pool and judge; abbreviations and bold as in Table 8.

Pool	Judge	n	First	Maj.	Vote	CodeT	Code	KO	WITNESSFIRST
LiveCodeBench v5/v6									
Qwen2.5-Coder-7B	Qwen3-4B	18	9	6	**12**	**12**	9	9	8
	Qwen3-8B	18	9	6	13	13	11	11	**14**
	Qwen3-32B	18	9	6	12	13	**14**	13	12
	GPT-5.5	18	9	6	12	14	**18**	17	15
Qwen3-8B	Qwen3-4B	33	18	10	19	**21**	19	18	**21**
	Qwen3-8B	33	18	10	16	20	20	18	**21**
	Qwen3-32B	33	18	10	18	19	23	19	**27**
	GPT-5.5	33	18	10	20	20	**31**	28	30
DeepSeek-Coder-V2-Lite	Qwen3-4B	13	7	4	**8**	6	**8**	6	**8**
	Qwen3-8B	13	**7**	4	6	6	5	6	6
	Qwen3-32B	13	7	4	**8**	6	6	5	6
	GPT-5.5	13	7	4	9	11	**13**	11	12
Qwen2.5-Coder-32B	Qwen3-4B	36	21	16	**23**	22	21	21	21
	Qwen3-8B	36	21	16	**23**	22	22	21	21
	Qwen3-32B	36	21	16	**24**	**24**	22	21	23
	GPT-5.5	36	21	16	25	26	30	31	**33**
CodeContests+									
Qwen2.5-Coder-7B	Qwen3-4B	3	0	0	**1**	0	0	0	0
	Qwen3-8B	3	0	0	2	1	2	1	**3**
	Qwen3-32B	3	0	0	**2**	**2**	**2**	0	**2**
	GPT-5.5	3	0	0	2	**3**	**3**	**3**	**3**
Qwen3-8B	Qwen3-4B	7	**4**	3	**4**	**4**	**4**	**4**	**4**
	Qwen3-8B	7	4	3	4	4	4	4	**6**
	Qwen3-32B	7	4	3	5	4	**6**	4	**6**
	GPT-5.5	7	4	3	5	**7**	**7**	**7**	**7**
DeepSeek-Coder-V2-Lite	Qwen3-4B	3	**2**	**2**	**2**	1	**2**	**2**	**2**
	Qwen3-8B	3	**2**	**2**	**2**	1	1	**2**	**2**
	Qwen3-32B	3	2	2	2	2	2	2	**3**
	GPT-5.5	3	2	2	2	2	**3**	2	**3**
Qwen2.5-Coder-32B	Qwen3-4B	3	1	0	1	**2**	1	1	1
	Qwen3-8B	3	1	0	2	1	2	1	**3**
	Qwen3-32B	3	1	0	2	2	**3**	1	**3**
	GPT-5.5	3	1	0	**3**	**3**	**3**	**3**	**3**

E PROMPTS

Prompts are reproduced verbatim. Angle-bracketed placeholders stand for the task statement, the public examples, the programs, and the input and outputs of a comparison; the public examples and inputs are inserted as JSON.

CANDIDATE SAMPLING (STANDARD I/O)

```
You are an expert Python programmer. Solve the problem.
The program must read from stdin and print the answer to stdout.
Define the solution then call it. Return only a Python markdown
↪ block.

<STATEMENT>
```

CANDIDATE SAMPLING (FUNCTIONAL)

```
You are an expert Python programmer. Solve the LeetCode-style
↪ problem.
Complete class Solution. Do not read stdin. Return only a Python
↪ markdown block.

<STATEMENT>

Starter code:
<STARTER CODE>
```

PAIR-SPECIFIC INPUT GENERATION (STANDARD I/O)

```
Two Python programs solve the same competitive-programming problem
↪ (stdin -> stdout). Write ONE new stdin test on which they are
↪ likely to print different outputs, so that a judge can tell
↪ which one is wrong. Requirements: the input MUST satisfy every
↪ constraint in the statement (format, value ranges, N limits);
↪ keep it small (at most 300 characters); prefer boundary
↪ values, duplicates, minimal N, and cases the public examples
↪ do not cover.
Return ONLY a JSON object {"mode": "stdin", "input": "<stdin text
↪ with \n newlines>"}.
Public examples: [{"input":"<PUBLIC EXAMPLE>","mode":"stdin"}]
Problem:
<STATEMENT>

Program A:
<PROGRAM A>

Program B:
<PROGRAM B>
```

PAIR-SPECIFIC INPUT GENERATION (FUNCTIONAL)

```
Two Python programs implement the same method for one problem.
↪ Write ONE new argument list on which they are likely to return
↪ different values, so that a judge can tell which one is wrong.
↪ The arguments MUST satisfy every constraint stated in the
↪ problem (types, value ranges, length limits); keep them small;
↪ prefer boundary cases the public examples do not cover.
Return ONLY a JSON object of exactly this shape, with one entry in
↪ "args" per parameter of the method:
```

```
{"args":["<PUBLIC
 ↪  EXAMPLE>"],"func_name":"solve","mode":"leetcode"}
Problem:
<STATEMENT>

Program A:
<PROGRAM A>

Program B:
<PROGRAM B>
```

PAIRWISE JUDGE WITH EXECUTION WITNESS

```
Problem:
<STATEMENT>
Test input JSON: "<INPUT>"
[Sample 0] code:
<PROGRAM A>
output: <OUTPUT A> error=ok
[Sample 1] code:
<PROGRAM B>
output: <OUTPUT B> error=ok
Which sample is more likely correct? Reply with ONLY one
 ↪  character: 0, 1, or 2.
```

CODE-ONLY PAIRWISE JUDGE

```
Problem:
<STATEMENT>
[Sample 0] code:
<PROGRAM A>
[Sample 1] code:
<PROGRAM B>
Which sample is more likely correct? Reply with ONLY one
 ↪  character: 0, 1, or 2.
```

SHARED TEST SUITE GENERATION (STANDARD I/O)

```
Write 5 new stdin test inputs for this competitive-programming
 ↪  problem. Every input MUST satisfy all constraints in the
 ↪  statement (format, value ranges, N limits); keep each small
 ↪  (at most 300 characters); cover boundary values and cases the
 ↪  public examples do not cover.
Return ONLY a JSON array of objects {"mode": "stdin", "input":
 ↪  "<stdin text with \n newlines>"}.
Public examples: [{"input":"<PUBLIC EXAMPLE>","mode":"stdin"}]
Problem:
<STATEMENT>
```

SHARED TEST SUITE GENERATION (FUNCTIONAL)

```
Write 5 new argument lists for the method in this problem. Every
 ↪  argument list MUST satisfy all constraints stated in the
 ↪  problem (types, value ranges, length limits); keep them small;
 ↪  cover boundary cases the public examples do not cover.
Return ONLY a JSON array whose elements have exactly this shape,
 ↪  with one entry in "args" per parameter of the method:
{"args":["<PUBLIC
 ↪  EXAMPLE>"],"func_name":"solve","mode":"leetcode"}
```

```
Problem:
<STATEMENT>
```

CODET TEST GENERATION (STANDARD I/O)

```
Write 5 new stdin test inputs for this competitive-programming
 ↪  problem. Every input MUST satisfy all constraints in the
 ↪  statement (format, value ranges, N limits); keep each small
 ↪  (at most 300 characters); cover boundary values and cases the
 ↪  public examples do not cover.
Return ONLY a JSON array of objects {"input": "<stdin text with \n
 ↪  newlines>", "output": "<exact expected stdout>"}.
Public examples: [{"input":"<PUBLIC EXAMPLE>","mode":"stdin"}]
Problem:
<STATEMENT>
```

CODET TEST GENERATION (FUNCTIONAL)

```
Write 5 new argument lists for the method in this problem. Every
 ↪  argument list MUST satisfy all constraints stated in the
 ↪  problem (types, value ranges, length limits); keep them small;
 ↪  cover boundary cases the public examples do not cover.
Return ONLY a JSON array whose elements have exactly this shape,
 ↪  with one entry in "args" per parameter of the method, like
 ↪  {"args": [...], "output": <expected return value as JSON>}:
{"args":["<PUBLIC EXAMPLE>"],"output":"<expected return value as
 ↪  JSON>"}
Problem:
<STATEMENT>
```

PROGRAM-AGNOSTIC STRESS WITNESS GENERATOR (STANDARD I/O)

```
Write at most 3 deterministic Python input-generator programs,
 ↪  using only the standard library. Use a fixed seed (e.g.
 ↪  random.seed(0)) for any randomness. Each program must print
 ↪  EXACTLY ONE input satisfying ALL statement format rules and
 ↪  constraints. Finish within 5 seconds; stdout must be at most
 ↪  16 MiB in UTF-8. Do not print explanations or expected
 ↪  outputs. max: maximum permitted size and values, worst case
 ↪  for naive/inefficient algorithms. max2: another maximum-size
 ↪  structure (equal, sorted, extreme values, etc.). min: smallest
 ↪  permitted size and boundary values. Use each kind at most
 ↪  once. Print the exact stdin text.
Return ONLY JSON: {"generators": [{"kind": "max|max2|min", "code":
 ↪  "..."}]}.
Public example input: {"mode": "stdin", "input": "<PUBLIC
 ↪  EXAMPLE>"}
Problem:
<STATEMENT>
```

PROGRAM-AGNOSTIC STRESS WITNESS GENERATOR (FUNCTIONAL)

```
Write at most 3 deterministic Python input-generator programs,
 ↪  using only the standard library. Use a fixed seed (e.g.
 ↪  random.seed(0)) for any randomness. Each program must print
 ↪  EXACTLY ONE input satisfying ALL statement format rules and
 ↪  constraints. Finish within 5 seconds; stdout must be at most
 ↪  16 MiB in UTF-8. Do not print explanations or expected
 ↪  outputs. max: maximum permitted size and values, worst case
 ↪  for naive/inefficient algorithms. max2: another maximum-size
 ↪  structure (equal, sorted, extreme values, etc.). min: smallest
 ↪  permitted size and boundary values. Use each kind at most
 ↪  once. Print one JSON list, one item per method argument,
 ↪  shaped like the example args.
Return ONLY JSON: {"generators": [{"kind": "max|max2|min", "code":
 ↪  "..."}]}.
Public example input: {"mode": "leetcode", "func_name": "solve",
 ↪  "args": ["<PUBLIC EXAMPLE>"]}
Problem:
<STATEMENT>
```

PROGRAM-AWARE STRESS WITNESS GENERATOR (STANDARD I/O)

```
All programs below passed the public examples, but some may be too
 ↪  slow or crash on inputs allowed by the problem statement. Read
 ↪  them and identify risks: time complexity too high for the
 ↪  constraints, excessive recursion depth, out-of-bounds
 ↪  indexing, or unhandled boundary cases. Write 14 deterministic
 ↪  Python input-generator programs using only the standard
 ↪  library. Use a fixed seed (e.g. random.seed(0)) for any
 ↪  randomness. Each must print EXACTLY ONE input satisfying ALL
 ↪  statement format rules and constraints, aiming to make at
 ↪  least one program time out or crash. At least one generator
 ↪  must have kind scale and produce an input at the maximum
 ↪  permitted size; use kind edge for boundary cases. Each
 ↪  generator must finish within 5 seconds; stdout must be at most
 ↪  16 MiB in UTF-8. Do not solve the problem. Do not print
 ↪  explanations or expected outputs. Print the exact stdin text.
Return ONLY JSON: {"generators": [{"kind": "scale|edge", "target":
 ↪  "short description, e.g. P2 O(n^2) loop", "code": "..."}]}.
Public example input: {"mode": "stdin", "input": "<PUBLIC
 ↪  EXAMPLE>"}
Problem:
<STATEMENT>
Public-passing programs (labels and exact code, JSON):
[{"label": "P0", "code": "<PROGRAM P0>"}, {"label": "P1", "code":
 ↪  "<PROGRAM P1>"}]
```

PROGRAM-AWARE STRESS WITNESS GENERATOR (FUNCTIONAL)

```
All programs below passed the public examples, but some may be too
 ↪  slow or crash on inputs allowed by the problem statement. Read
 ↪  them and identify risks: time complexity too high for the
 ↪  constraints, excessive recursion depth, out-of-bounds
 ↪  indexing, or unhandled boundary cases. Write 14 deterministic
 ↪  Python input-generator programs using only the standard
 ↪  library. Use a fixed seed (e.g. random.seed(0)) for any
 ↪  randomness. Each must print EXACTLY ONE input satisfying ALL
 ↪  statement format rules and constraints, aiming to make at
 ↪  least one program time out or crash. At least one generator
 ↪  must have kind scale and produce an input at the maximum
 ↪  permitted size; use kind edge for boundary cases. Each
 ↪  generator must finish within 5 seconds; stdout must be at most
 ↪  16 MiB in UTF-8. Do not solve the problem. Do not print
 ↪  explanations or expected outputs. Print one JSON list, one
 ↪  item per method argument, shaped like the example args.
Return ONLY JSON: {"generators": [{"kind": "scale|edge", "target":
 ↪  "short description, e.g. P2 O(n^2) loop", "code": "..."}]}.
Public example input: {"mode": "leetcode", "func_name": "solve",
 ↪  "args": ["<PUBLIC EXAMPLE>"]}
Problem:
<STATEMENT>
Public-passing programs (labels and exact code, JSON):
[{"label": "P0", "code": "<PROGRAM P0>"}, {"label": "P1", "code":
 ↪  "<PROGRAM P1>"}]
```

SYNTHESIZED INPUT VALIDATOR (STANDARD I/O)

```
Write one Python input validator, using only the standard library.
 ↪  stdin contains the candidate stdin text. Read all input. Check
 ↪  ALL statement format rules, types, ranges, size limits and
 ↪  structural constraints. Exit with code 0 iff valid; otherwise
 ↪  exit with code 1. Do not solve the problem. Finish within 5
 ↪  seconds.
Return ONLY JSON: {"code": "..."}.
Public example input: {"mode": "stdin", "input": "<PUBLIC
 ↪  EXAMPLE>"}
Problem:
<STATEMENT>
```

SYNTHESIZED INPUT VALIDATOR (FUNCTIONAL)

```
Write one Python input validator, using only the standard library.
 ↪  stdin contains json.dumps(args): a JSON list, one item per
 ↪  method argument. Read all input. Check ALL statement format
 ↪  rules, types, ranges, size limits and structural constraints.
 ↪  Exit with code 0 iff valid; otherwise exit with code 1. Do not
 ↪  solve the problem. Finish within 5 seconds.
Return ONLY JSON: {"code": "..."}.
Public example input: {"mode": "leetcode", "func_name": "solve",
 ↪  "args": ["<PUBLIC EXAMPLE>"]}
Problem:
<STATEMENT>
```