CLAIMCHECK: MEASURING AND IMPROVING SPECIFICATION FAITHFULNESS

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification proves that a program satisfies its specification, but says nothing about whether the specification captures what the author meant. As LLMs increasingly write both code and specifications, a gap between the author's natural-language requirement and the specification becomes a live risk. A machine-checked proof of a property narrower than the one requested is worthless but passes every check. Closing this gap is solving the spec formalization problem: given a natural-language requirement, produce a formal artifact whose meaning matches it.

To measure the gap, we curate an end-to-end benchmark for spec formalization spanning four formal languages: Dafny contracts and Lean 4, Isabelle and Rocq theorem statements. Each of its 3,234 rows pairs a natural-language requirement with a formal artifact and a label established by execution, by proof, or by the benchmark it is drawn from. A third of the rows are meaning-preserving rewrites of the artifact that keep the faithful label, so a judge cannot score by detecting that a row was edited.

We introduce `claimcheck`, a tool that detects incorrect formalizations: the formal artifact is translated back into plain English by a model that never sees the requirement, and a separate model, the judge, decides whether the informalization and the requirement express the same guarantee. On all four languages it gains 8.2 to 9.8 points of macro recall over a single call shown the requirement and the artifact, and 4.2 to 6.0 over that call handed the same list of meaning-preserving rewrites. Showing the judge the artifact as well costs 1.4 to 4.5 points. Across four judges the gain shrinks as the judge gets stronger, to within a point of the informed single call for the strongest.

1 INTRODUCTION

As models get better at writing code, the programmer's job shifts toward saying *what* a program should do rather than *how* it should do it. Spec-driven development already works this way: the developer states the requirement in English, and the implementation is generated from it. The hard part moves from writing the code to knowing whether it does what was asked. For most of what a model produces there is no automatic way to tell, and checking falls to a human or to another model (Zheng et al., 2023). Formal languages are the exception. Here the requirement is first written down as a specification: a precise statement of what the program must guarantee, in a form a machine can check. A verifier then proves that the code meets that specification on every possible input. But the verifier only ever sees the specification. It never sees the natural-language requirement that specification was written from, so it cannot tell whether the specification asked for the right thing.

Suppose the requirement is to sort a list. The specification says the output must be in increasing order, and the code is proved to satisfy it. Now take a program that ignores its input and returns an empty list. Nothing in an empty list is out of order, so that program meets the specification too, and the same proof would go through. The specification never said the output has to contain the same items as the input. This is the part no verifier covers. In our Dafny (Leino, 2010) slice every row we labelled defective still passes verification, so a verified artifact can carry an assurance it has not earned. We call the task of getting it right *spec formalization*: given a natural-language requirement, produce a formal artifact whose meaning matches it.

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.

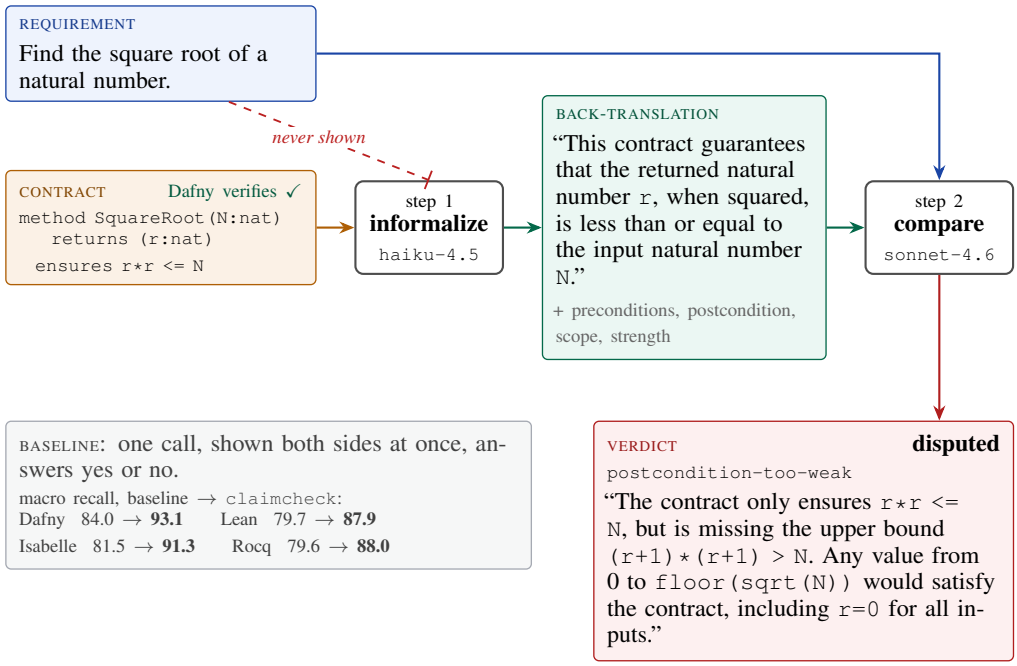


Figure 1: One `claimcheck` run, on a Dafny contract from CloverBench whose postcondition was weakened (class `C2`). Like the sorting example, the contract still verifies and is satisfied by returning zero. Step 1 translates the contract into English without ever seeing the requirement, so the informalization can only report what the contract says. Step 2 compares the informalization with the requirement and names the discrepancy; it never sees the contract. Colours follow the data: blue is the natural-language requirement, amber the formal artifact, green its informalization. A single-call baseline shown both sides at once scores lower on all four languages (Table 2).

We introduce `claimcheck`, a tool that checks whether a formal artifact means what its natural-language requirement asks for. The direct way to check this is to show a model both the requirement and the specification and ask whether they agree (Cabral et al., 2026; Zhang et al., 2026; Lu et al., 2025). The trouble is that the requirement gives the answer away. A model that has just read *sort a list* expects the specification to describe sorting, and is biased toward agreeing that it does. Telling the model not to do this is an instruction it can fail to follow, so `claimcheck` separates reading the specification from judging it. A first model, the informalizer, puts the specification into plain English — its preconditions, its postcondition, its scope — without ever seeing the requirement, so what it writes can only report what the specification says. We call this the informalization (Wu et al., 2022; Amrollahi et al., 2026). A second model, the judge, then compares the informalization against the requirement (Figure 1). What it compares is two English texts, one saying what was asked for and the other what the artifact guarantees, and the one it reads was written before the requirement could shape it. We also report the variant that hands the judge the artifact alongside the informalization; it is the weaker of the two on all four of our languages, by 1.4 to 4.5 points of macro recall.

1.1 Contributions

Our paper's main contributions are:

1. We curate a test suite for end-to-end formalization of natural-language requirements, spanning four formal languages: Dafny contracts and Lean 4, Isabelle and Rocq theorem statements. Its 3,234 rows pair a requirement with a formal artifact and a label established by execution, by proof, or by the benchmark it is drawn from, and are scored by a standardized grader (Section 2). Benchmarks of this kind are built by mutation, so their labels record whether a row was edited rather than what it now means. A third of our rows are control mutations, which rewrite the formal artifact while preserving its meaning and stay labelled faithful, so detecting edits is no longer enough to score.

2. We measure baseline scores for frontier models. A single-prompt judge shown the requirement and the artifact reaches between 79.6 and 84.0 macro recall across the four languages, and errs in both directions: on Dafny it disputes 20% of the faithful contracts while missing a quarter of the rows whose requirement carries the defect. Handing that same single call the list of meaning-preserving rewrites and the error vocabulary `claimcheck`'s judge is given raises it by 3 to 5 points on all four languages, so we report it as a second baseline throughout: it is what separates the pipeline from the information in its prompt (Section 5).
3. We introduce `claimcheck`, the two-step tool of Figure 1. Comparing the two English texts gains 4.2 to 6.0 points of macro recall over that informed baseline on all four, most of it on the faithful rows, where the baselines dispute controls that preserve meaning. Three ablations locate the gain: removing the back-translation costs 2.8 to 6.0 points, giving the informalizer the requirement it is otherwise denied costs 0.8 to 3.0, and holding rewrites out of the judge's list moves macro recall by at most a point, so that list is not acting as an answer key for the benchmark's controls. Varying the judge shows the gain is substitutive rather than additive: it is 27 points for the weakest model we test and zero for the strongest (Section 5).

2 Benchmark Design

Each row is a natural-language requirement, a formal artifact meant to express it, and the context needed to read that artifact, and its label says whether the artifact means what the requirement asks for: `confirmed` or `disputed`. The benchmark is a union of four slices: 362 Dafny rows derived from CloverBench (Sun et al., 2024), and 1,296 Lean 4, 920 Isabelle and 656 Rocq rows derived from PutnamBench (Tsoukalas et al., 2024).

Meaning-preserving controls. A faithfulness benchmark needs pairs whose two sides are known to disagree, and the usual way to get them is mutation. Take a correct pair, damage one of its two sides, and label the result `disputed`. Both sides get damaged here. Clover mutates the contract in two of its three variant classes and the docstring in the third; the PutnamBench slices mutate the theorem statement. The construction has a defect that is easy to miss. Suppose a judge ignores both sides of every row and answers `confirmed` whenever the artifact looks untouched. It gets the entire confirmed half right, because the label tracks where a row came from rather than what it means. A high score on such a benchmark can therefore reflect nothing but good mutation detection. We add a third family to close that hole. A *control mutation* rewrites the formal artifact while preserving its meaning, and it stays labelled `confirmed`. Every slice carries one, and together they are 1,029 of the 1,854 confirmed rows, so a judge betting on provenance now loses most of the confirmed half either way. Controls carry a burden that semantic mutations do not: a rewrite has to be *shown* to preserve meaning, not merely intended to, and the two subsections below set out how each slice discharges it.

2.1 Dafny

CloverBench ships 62 hand-written textbook programs, each a docstring, which serves as the requirement, paired with a contract, together with a taxonomy of incorrect variants. We use three of its four released classes: the 62 originals are labelled `confirmed`, and `C1`, which mutates the docstring (62 rows), `C2`, which mutates the contract (60), and `C6`, which mutates the contract and the body together (60), are `disputed`. All 182 defective rows still verify in Dafny, which the benchmark states as a design requirement: the domain has a formal oracle and it reports success on every defective row, so an autoformalization has to be checked for faithfulness even where verification is available.

We add a fourth family of 118 control mutations over 45 of the 62 programs, which rewrite the contract while preserving what it guarantees and so keep the label `confirmed`. A candidate is kept only if Dafny proves the rewritten contract equivalent to the original under its preconditions, and only if it cannot also prove that postcondition false, which would make the equivalence vacuous. The slice is 362 rows, 180 of them `confirmed`.

The formal artifact is the contract alone: declaration, `requires` and `ensures`, with the body stripped. The body is what a contract is checked *against* rather than part of what it claims, and a judge

given it could infer the intent from the implementation instead of reading the contract. Extraction is exact because no Clover program declares a helper function, predicate or datatype. Four rows are dropped where the mutated contract retains only framing clauses such as `modifies s`, which guarantee nothing and leave nothing to compare, so `C2` and `C6` hold 60 rows each (Appendix B).

2.2 LEAN, ISABELLE AND ROCQ

PutnamBench (Tsoukalas et al., 2024) formalizes the same competition problems in all three languages, and the natural-language statement lives in one file keyed by problem name, independent of the language. A row in these slices therefore pairs the *same* requirement with a different formal artifact, which is what makes the three comparable: a difference between them is a difference in the language, not in the question. The artifact is the theorem statement and the requirement is the problem statement that sits above it in the source, so the expression that extracts the one removes the other. PutnamBench writes a problem's answer into a separate constant, which we substitute back into the context before splitting the theorem, since a statement about an opaque constant reads as vacuous whatever it says.

Five rewrite rules preserve meaning and four change it (Table 1), and the two families are warranted differently. For the controls, Lean discharges the obligation `(original)` $\leftrightarrow$ `(mutated)` per row in PutnamBench's own project under a fixed tactic list, and a candidate that fails is dropped. In Isabelle and Rocq the same five rules are propositional schemas — $\top \wedge P \leftrightarrow P$, $P \vee \bot \leftrightarrow P$, $\neg\neg P \leftrightarrow P$, a universal through its dual, an iff with its sides swapped — valid for every P, so the argument is made once per rule rather than once per row, and no Isabelle or Rocq installation enters the construction. Neither form of warrant is a weakened version of the other: a per-row obligation catches a rewriter that misfires on a single statement, and a per-rule argument holds for every instance without depending on a tactic finding the proof.

The semantic rules are labelled by their construction. `negate` replaces the statement by its negation, which contradicts it for every P. `forall_to_exists` yields a strictly weaker statement, which differs from the original unless the domain is a singleton. `bump_const` moves a numeral and `strictness` loosens or tightens an inequality; both are Lean-only, because they pick their target by position and the three formalizations were written independently over different libraries, so the same rule lands on a different subterm in each language. Over the problems Lean and Isabelle share, 63% differ in how many numerals their statement contains.

Table 1: The PutnamBench slices. Controls are labelled `confirmed`, semantic mutations `disputed`. A dash marks a rule that is not applied in that language; see the text.

	rule	Lean	Isabelle	Rocq
control	`true_and`	98	94	62
	`or_false`	98	96	66
	`double_neg`	97	95	57
	`forall_not_exists`	24	22	38
	`iff_comm`	21	18	25
semantic	`negate`	293	285	185
	`bump_const`	229	—	—
	`strictness`	114	—	—
	`forall_to_exists`	29	25	38
	untouched originals	293	285	185
	total	1,296	920	656

3 HOW CLAIMCHECK WORKS

`claimcheck` is a tool for checking whether a formal artifact means what its natural-language requirement asks for. Given the pair, it returns a verdict, `confirmed` or `disputed`, and names the discrepancy where it finds one. Figure 1 and Figure 2 show one run each, on two Dafny contracts.

A run is two model calls. *Informalize* receives the artifact and whatever context is needed to read it, and writes an informalization: a plain-English account of what the artifact guarantees. It is not given the requirement. *Compare* then receives two descriptions of the same thing in the same language, the requirement and that informalization, and returns the verdict. Both figures set the two calls out in order, with what each one reads.

Comparing the two English descriptions is the idea. One says what was asked for, the other says what the artifact actually guarantees, and a difference in meaning surfaces as a difference between two sentences rather than between a sentence and a block of code. The contract in Figure 1 keeps every word of its requirement and is satisfied by returning zero, and the contract in Figure 2 has exactly the right shape for its requirement and one wrong index inside the slice. The informalization is what makes that comparison possible, and it can only do the job because it was written blind. An informalization produced with the requirement in view would tend to echo it, and a near miss would read as a hit.

We also report a variant in which the judge is handed the artifact alongside the informalization, so that it can check the account against the source where that account is unclear or looks wrong. It is the weaker of the two on every language we test, by 1.4 to 4.5 points of macro recall, and Section 5 reports both.

Dafny `remove_front`, the slice takes the wrong end

REQUIREMENT

Removes the front element from the array `a` and returns the resulting array `c`.

CONTRACT

```
method remove_front(a:  array<int>) returns (c:  array<int>)
requires a.Length > 0
ensures a[0..a.Length - 1] == c[..]
```

STEP 1 INFORMALIZE *reads the contract, never the requirement*

"The returned array `c` contains the elements of array `a` from index 0 to `a.Length - 1` (i.e. all elements except the last one)."

STEP 2 COMPARE *reads the requirement and the informalization*

disputed, `wrong-property`

"The requirement says to remove the FRONT (first) element, but the back-translation describes removing the LAST element (slice from index 0 to `a.Length-1` excludes the last element, not the first)."

Figure 2: An example `claimcheck` run on a Dafny contract. The contract has the right shape and one wrong index inside the slice: `a[0..a.Length - 1]` drops the last element where `a[1..]` would drop the first, so the method it describes returns the array with the wrong end removed. The contract verifies against a body that satisfies it.

4 Experimental Setup

Baseline. The baseline gets one row per call: the requirement, the formal artifact and its context. It replies through a tool call whose only field is `YES` or `NO`. The prompt contains no system message, worked examples, reasoning instruction or failure vocabulary. Each of those is itself a reasoning aid, so giving them to the baseline would make the comparison measure the prompt rather than the tool. *Baseline + list* is the one exception, and it is a control: the same single call, given the judge's list of rewrites that leave a claim unchanged and of the kinds of mismatch to look for. That block is taken verbatim from the judge's own tool rather than written for this arm, so no wording in it is ours to choose; the cost is that descriptions written for a tool with more fields than a bare `YES`/`NO` one mention fields this arm does not have, which makes it a conservative control. It still answers `YES` or `NO`, so what it holds fixed against `claimcheck` is the information in the prompt. What separates the two-step pipeline from the output schema it answers through is the no-back-translation ablation below.

`claimcheck`. Two steps. *Informalize*: a model, the informalizer, sees the formal artifact and its context, but not the requirement, and writes an informalization, a plain-English account of what the

artifact guarantees. *Compare*: a judge reads that informalization against the requirement and decides whether the two express the same guarantee. What the tool separates is the reading of the artifact from the judging of it: the informalization the judge works from was written before the requirement could shape it. The version we evaluate adds three things to the prompts the tool shipped with. The informalizer is given a short list of structure that carries no meaning, such as a double negation or a conjunction with an always-true clause, and describes the artifact with it discarded, so two artifacts that say the same thing get the same informalization. The judge's tool lists the rewrites that leave a claim unchanged in each language (in Dafny, for example, two `ensures` clauses merged into one with `&&`, which changes nothing because Dafny conjoins them by definition) and names the kinds of mismatch to look for, such as a wrong constant or a missing condition. And the judge is asked plainly whether the two express the same guarantee, where the original wording told it to be strict and to prefer flagging a mismatch. The variant we also report hands the judge the formal artifact alongside the informalization; everything else, the tool schema included, is unchanged.

Settings. Informalizer `claude-haiku-4.5`, judge `claude-sonnet-4.6`, 8192 max tokens, temperature 0. Both return their output through a tool call, and the judge's verdict is a single boolean. The rewrite list and the error vocabulary are per language, following each slice's own construction: the judge answers through a contract-shaped taxonomy on Dafny, whose categories name which clause moved and in which direction, and a proposition-shaped one on Lean, Isabelle and Rocq. Both steps get one row per call, as the baseline does, so the two see the same input and any difference comes from the tool. Neither prompt carries the row's name, which in earlier runs could reveal how a row was built (Appendix C).

Ablations and sweep. Each ablation changes one part of the tool. *Held-out rules* removes some rewrites from the lists, to test whether they act as an answer key for the benchmark's controls: the always-true conjunction and always-false disjunction on Lean, Isabelle and Rocq, and a shifted bound, a swapped equality and a contrapositive on Dafny. *No back-translation* skips the informalize step, so the judge compares the requirement with the artifact directly, under the same prompt otherwise. *Informalizer sees requirement* gives the informalizer the requirement it is otherwise denied. The first and last are run on `claimcheck` as reported; no back-translation is run on the variant that sees the artifact, since a judge with neither the artifact nor an informalization would have nothing to read. The sweep repeats the baseline, baseline + list and `claimcheck` with a different judge, the informalizer fixed at `claude-haiku-4.5`: `gemini-3.7-flash`, `gpt-5.6-luna` and `deepseek-v4-pro`, reached through OpenRouter at temperature 0. `deepseek-v4-pro` runs with its reasoning mode off, since it rejects a forced tool call otherwise. The last row of Table 4 (Appendix A) uses `gemini-3.7-flash` in both roles.

5 Results

5.1 Main results

Table 2: Main results. Informalizer `claude-haiku-4.5`; judge and baseline `claude-sonnet-4.6`. Macro recall (mean of the `confirmed` and `disputed` recalls) ± its 95% interval. Per-row outputs: anonymized release.

	Lean	Isabelle	Rocq	Dafny
zero-shot baseline	79.7 ±2.1	81.5 ±1.9	79.6 ±2.3	84.0 ±3.8
baseline + list	83.2 ±1.9	85.3 ±1.8	83.0 ±2.3	88.9 ±3.2
claimcheck, judge sees artifact	84.8 ±1.9	87.2 ±1.8	83.5 ±2.3	91.7 ±2.8
claimcheck	87.9 ±1.8	91.3 ±1.7	88.0 ±2.1	93.1 ±2.6

`claimcheck` is ahead of both baselines on all four languages: by 8.2 to 9.8 points of macro recall over the zero-shot baseline, and by 4.2 to 6.0 over the same call handed the rewrite list (Table 2). Most of the gain is on faithful rows. Confirmed recall rises from 66.5–85.6% with the list to 77.8–91.7%, on the untouched originals as well as on the meaning-preserving controls: the list baseline accepts 69.7–91.9% of the originals and 64.1–82.2% of the controls, `claimcheck` 78.4–95.2%

and 77.4–89.8%. Disputed recall moves less. It rises on Dafny, from 92.3 to 94.5%, falls by at most 3.3 points on Isabelle and Rocq, and falls on Lean, from 95.9 to 88.4%.

Paired on the same rows, `claimcheck` fixes more of the list baseline's errors than it introduces on every language: 148 against 91 on Lean (exact McNemar $p = 3 \times 10^{-4}$), 128 against 46 on Isabelle ($p = 4 \times 10^{-10}$), 83 against 37 on Rocq ($p = 3 \times 10^{-5}$) and 24 against 9 on Dafny ($p = 0.014$).

Handing the judge the artifact as well costs 1.4 to 4.5 points. That variant disputes more of the faithful rows: its confirmed recall is 9.9 to 12.0 points lower on Lean, Isabelle and Rocq, and 2.8 lower on Dafny.

5.2 Ablations

Table 3: Ablations, with the models of Table 2. Macro recall (mean of the `confirmed` and `disputed` recalls) ± its 95% interval.

	Lean	Isabelle	Rocq	Dafny
held-out rules	88.0 ±1.8	91.8 ±1.7	87.5 ±2.3	92.3 ±2.8
no back-translation (judge sees artifact)	82.0 ±2.0	81.9 ±2.0	77.5 ±2.6	86.7 ±3.4
informalizer sees requirement	85.3 ±1.9	89.9 ±2.1	87.2 ±2.6	90.1 ±3.1

Without the back-translation, the judge compares the requirement with the artifact directly and loses 2.8 to 6.0 points against the variant it is run on (Table 3); on every language it ends below the single-call baseline with the list. Giving the informalizer the requirement costs 0.8 to 3.0 points, and the loss is on the disputed side: disputed recall falls by 6.0 to 8.8 points while confirmed recall holds or rises. That is the direction Section 3 predicts for an informalization written with the requirement in view. Holding some of the rules that generate the benchmark's controls out of the lists moves macro recall by at most 0.8 points, inside every interval, so the lists are not acting as an answer key for those rules.

5.3 Model sweep

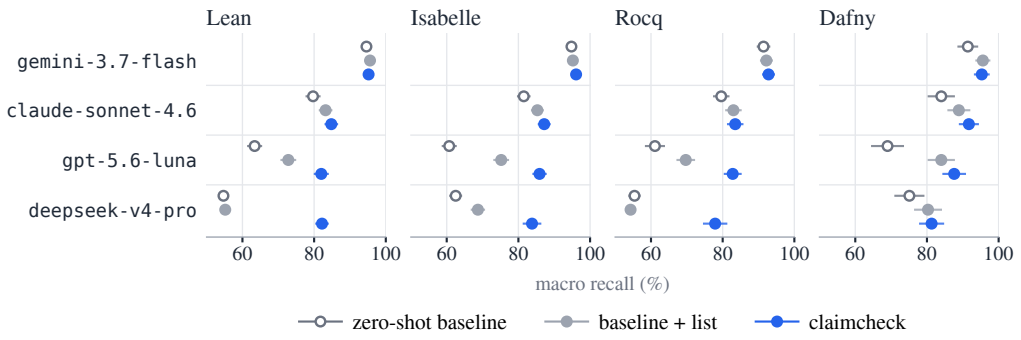


Figure 3: The model sweep, with 95% intervals; values in Table 4. As there, `claimcheck` is the variant whose judge also sees the artifact, and the `claude-sonnet-4.6` row is Table 2's judge under that variant.

The sweep's `claimcheck` rows are the variant whose judge also sees the artifact, and how much they gain depends on the judge (Figure 3; the values are in Table 4, Appendix A). For the two weaker judges the gain is large: over the zero-shot baseline, `gpt-5.6-luna` gains 18.6 to 25.2 points and `deepseek-v4-pro` 6.2 to 27.5, while the list alone adds at most 15.0. `gemini-3.7-flash` already reaches 91.4 to 94.8 in a single call; it gains 0.6 to 3.9 over that, and is within a point of its list baseline on every language. Using it as the informalizer too changes macro recall by at most 1.1 points. `claimcheck` narrows the gap between judges: the spread across the three falls from 22.4–39.9 points for the baseline to 12.3–14.9.

6 Related Work

Specifications that verify but say the wrong thing. Boehm's distinction between verification, building the product right, and validation, building the right product, is forty years old (Boehm, 1984), and a verifier does only the first. Tools that write specifications with a model mostly take the verifier's word: SpecGen (Ma et al., 2025) and AutoSpec (Wen et al., 2024) keep whichever candidate contract the verifier accepts, which says nothing about whether the accepted candidate is the one the author meant. Where the specification is scored at all, it is scored against code. nl2postcond asks whether the reference implementation passes a generated postcondition and how many buggy versions of that implementation the postcondition rejects (Endres et al., 2024), and Richter and Wehrheim do the same for full contracts (Richter & Wehrheim, 2025). Benchmarks for verified code generation have met the gap from the other side. The authors of the vericoding benchmark read their specifications by hand and found about a tenth too weak and a further sixth poorly translated from the task (Bursuc et al., 2025). AlphaVerus adds an exploit step that tries to write a trivial program satisfying the specification, because a model rewarded for verified code will otherwise learn to weaken the specification (Aggarwal et al., 2025). Vacuity detection in model checking is the older form of the same worry: a temporal property can hold of a system for a reason that has nothing to do with what it was written to say (Beer et al., 2001; Kupferman & Vardi, 2003). All of these check the specification against code or traces. `claimcheck` checks it against the requirement.

Checking against a reference formalization. When a trusted formalization exists, faithfulness becomes equivalence between two formal objects, which can be decided or at least attempted. CLEVER asks the model to prove its specification equivalent to a hidden one (Thakur et al., 2025); Verina scores soundness and completeness against a ground-truth specification, by proof where it goes through and by tests where it does not (Ye et al., 2025). In mathematics, Poiroux et al. check derivability in both directions in Lean (Poiroux et al., 2025), LeanEuclid hands the equivalence to an SMT solver (Murphy et al., 2024), and GTED compares operator trees (Liu et al., 2025). These are precise where they apply, and they apply only once someone has written the answer down. Our rows carry no reference beyond the requirement. The controls of Section 2 do the one job that can be done without one, which is to certify that a rewrite of the artifact preserved its meaning: by proof where a prover decides it, and by an argument about the rule where none runs.

Judging against the requirement directly. Without a reference the judge has to read the natural language, and the usual design shows a model both sides and asks. ProofFlow scores faithfulness this way, clause by clause (Cabral et al., 2026); Zhang et al. take the agreement of two such judges as a stand-in for human review of Lean statements (Zhang et al., 2026). The record of the direct judge is mixed. A GPT-4 judge shown an informal statement and its formalization had far lower precision than FormalAlign's trained scorer (Lu et al., 2025), and Verus-SpecGym finds that a judge misses a quarter of the specification failures its executable evaluator catches (Agarwal et al., 2026). The literature on model judges gives reasons to expect this. What a judge sees next to the thing it is grading shapes the verdict: the first candidate is favoured (Wang et al., 2024), as is the judge's own output (Panickssery et al., 2024); a prior score shown as metadata anchors the grade even when it is uninformative by construction (Kapetanovic et al., 2026); and a supplied reference answer is weighed against what the judge already believes rather than taken as given (Lee et al., 2026). Our baseline is this direct judge with nothing else in the prompt, and the requirement is the anchor `claimcheck` withholds from the informalizer.

Informalization and round trips. The step `claimcheck` rests on is not new. Wu et al. noticed that Codex translated Isabelle into English far more reliably than the other way and proposed back-translation as a source of training data (Wu et al., 2022), which MMA then produced at scale (Jiang et al., 2024). Lean Workbook uses it as a filter: each formal statement is translated back, and a model is asked whether the two English texts pose the same problem (Ying et al., 2024). Li et al. replace the model with embedding similarity between the original and its back-translation (Li et al., 2024). Two recent systems are close. Amrollahi et al. formalize a statute, back-translate it, formalize it again, and let an SMT solver decide whether the two formalizations agree, calling a model only to diagnose which step failed (Amrollahi et al., 2026). Monty, for JML assertions, has one model describe the generated assertion and another compare that description with the original clause by clause (Liu et al., 2026). Outside program verification, nl2spec and ARTEMIS put the round trip in front of the user for

temporal logic, showing sub-translations or distinguishing traces to accept or reject (Cosler et al., 2023; Mendoza et al., 2026). Each of these uses the round trip inside its own system and reports the system's end-to-end gain. Clover is the closest, and it is where our Dafny slice comes from: it regenerates a docstring from the annotations without seeing the original and asks GPT-4 whether the two match (Sun et al., 2024). It scores that check alone, accepting all 60 originals and rejecting 40 to 70% of each incorrect class, but never against a model shown both sides; and on its rows, where every faithful row is untouched, detecting edits would score the whole faithful half. Mathesis makes that comparison, with DeepSeek-V3 throughout, on model-written Lean statements labelled by hand, and the round trip lost on F1, 0.65 to 0.85 (Yu et al., 2025). Our controls remove the edit signal, and Section 5 runs the comparison across four languages and four judges, against a single call given the same information.

Benchmarks with labelled mismatches. Labelled pairs of requirement and formalization are scarce, and the ones that exist are made by damage. CloverBench mutates the docstring, the annotations or the code (Sun et al., 2024); ProofNetVerif labels model-written formalizations against a reference (Poiroux et al., 2025); DriftBench plants controlled drift in Lean statements (Mohammad & Sheikh, 2026). FaithformBench scores the unperturbed rows next to the perturbed ones (Cornish et al., 2026), but its unperturbed rows are untouched, and ours are rewritten. The construction descends from mutation testing, which has a known hazard: the equivalent mutant, an edit that changes the text and not the behaviour, which must be found and discarded or it corrupts the score (Jia & Harman, 2011). We keep ours and label them faithful. Certifying one is an equivalence question, and our four slices answer it differently: the Dafny verifier decides it per row, a fixed tactic list decides it per row in Lean, and in Isabelle and Rocq the rules kept are propositional schemas whose correctness is argued once for the rule rather than once for the row.

7 Limitations

Our benchmark may not be hard enough to separate systems clearly: a single call already reaches between 79.6 and 84.0 macro recall, and between 83.0 and 88.9 once it is handed the rewrite list, so the room a two-step tool can occupy is small and differences between systems compress into a few points. Its composition narrows that room further. In Isabelle and Rocq `negate` is 92% and 83% of the disputed rows, and asserting the negation of a statement is the easiest defect in the benchmark to catch, so what separates systems on those two slices is almost entirely the confirmed side; and there the three trivial wrappers are about 45% of the rows, which tests whether a judge is disturbed by structure carrying no meaning rather than whether it can read a specification. Lean and Dafny carry defects of several strengths and are where the disputed half discriminates.

The labels also carry different warrants. Every control was shown to preserve meaning, by the Dafny verifier, by a tactic list in Lean, or by a propositional argument in Isabelle and Rocq, but the semantic mutations rest on the rule that produced them, and for `bump_const` and `strictness` that is intent alone, since a moved numeral or a loosened bound will in rare cases leave a statement equivalent to the one it replaced. And all four slices are single contracts or single theorem statements, short enough that a requirement and its formalization can be held in view at once. Artifacts whose meaning lives in their structure rather than in a proposition, and developments too long to read in one pass, are outside what we measure; finding such tasks is the main thing we would change before making a stronger claim.

8 Conclusion

`claimcheck` checks whether a formal artifact means what the natural-language requirement it was written from asks for, which is the question a verifier never asks. It does so in two model calls: one informalizes the artifact without seeing the requirement, the other compares the two English texts. Across Dafny contracts and Lean, Isabelle and Rocq theorem statements it gains 8.2 to 9.8 points of macro recall over a direct judge, and 4.2 to 6.0 over the same judge handed the same list of meaning-preserving rewrites in a single call. Undoing any of its three choices lowers the score: removing the back-translation costs 2.8 to 6.0 points, letting the informalizer see the requirement up to 3.0, and showing the judge the artifact up to 4.5.

AI USE STATEMENT

Generative AI models are an object of study in this work: they serve as the informalizer and the judge under evaluation, as described in Section 4. Separately, generative AI tools were used in writing and refactoring parts of the experimental harness, which was initially prototyped during a Claude Code hackathon. All harness code that produces reported numbers was read and tested by the authors. We have reviewed all AI-assisted work and take responsibility for the final content of this paper, including its text, claims and artifacts.

ETHICS STATEMENT

This work evaluates whether formal specifications faithfully capture natural-language requirements. It introduces no human-subjects data. The benchmark is derived from existing public datasets, whose provenance and our mutation procedure are described in Section 2. One foreseeable risk is misplaced confidence: a tool that reports a specification as faithful may encourage less human review of that specification, and our measured recall is well short of perfect. We report both class recalls rather than a single accuracy figure so that this failure mode stays visible.

REPRODUCIBILITY STATEMENT

The benchmark, the scripts that construct it, the evaluation harness, the prompts and tool schemas sent to every model, and the per-row output of every run reported here are released at `https://anonymous.4open.science/r/claimcheck-release-anonymized`. The construction scripts matter as much as the rows: they are what defines the labels, since a control mutation is kept only where the equivalence obligation it carries is discharged, and a reader who wants to check a label has to be able to re-run that obligation. The experimental settings, including the specific model versions used as informalizer and judge, are given in Section 4. The model sweep is in Figure 3, with its values in Table 4, and the row-name fix that separates these runs from our earlier ones is described in Appendix C. The benchmark construction procedure, including how meaning-preserving controls and semantic mutations are generated, is described in Section 2.

REFERENCES

Anmol Agarwal, Natalie Neamtu, Pranjal Aggarwal, Seungone Kim, Jannis Limperg, Cedric Flamant, Kanna Shimizu, Bryan Parno, and Sean Welleck. Verus-specgym: An agentic environment for evaluating specification autoformalization. *arXiv preprint arXiv:2605.26457*, 2026.

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. arXiv:2412.06176.

Daneshvar Amrollahi, Jerry Lopez, and Clark Barrett. Faithful autoformalization via roundtrip verification and repair. *arXiv preprint arXiv:2604.25031*, 2026.

Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A benchmark for vericoding: Formally verified program synthesis. *arXiv preprint arXiv:2509.22908*, 2025.

Rafael Cabral, Tuan Manh Do, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Xin Shen. Proofflow: A dependency graph approach to faithful proof autoformalization. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.15981.

Rob Cornish, Iacopo Ghinassi, Po-Hung Yeh, Shuqi Liu, Qiyuan Xu, Haoxuan Yin, Dominik Wagner, Wenda Li, Yee Whye Teh, and Luke Ong. Faithformbench: Benchmarking faithfulness of mathematical chain-of-thought autoformalisation. *arXiv preprint arXiv:2608.10916*, 2026.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *Computer Aided Verification (CAV)*, 2023. arXiv:2303.04864.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. arXiv:2310.01831.

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.

Albert Q. Jiang, Wenda Li, and Mateja Jamnik. Multi-language diversity benefits autoformalization. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. arXiv:2311.03755.

Ante Kapetanovic, Kemal Altwlkany, Andro Mercep, Tomislav Duricic, and Emanuel Lacic. Anchoring bias in LLM-as-a-judge systems: Prior scores compromise evaluation independence. *arXiv preprint arXiv:2608.25869*, 2026.

Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, 4(2):224–233, 2003.

Dongryeol Lee, Yerin Hwang, Taegwan Kang, Minwoo Lee, Younhyung Chae, and Kyomin Jung. Judging against the reference: Uncovering knowledge-driven failures in LLM-judges on QA evaluation. *arXiv preprint arXiv:2601.07506*, 2026.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Xian Zhang, Fan Yang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. arXiv:2410.20936.

Hongyi Liu, Madhusudan Parthasarathy, and Adithya Murali. Faithful autoformalization of natural language assertions. *arXiv preprint arXiv:2607.13303*, 2026.

Yuntian Liu, Tao Zhu, Xiaoyang Liu, Yu Chen, Zhaoxuan Liu, Qingfeng Guo, Jiashuo Zhang, Kangjie Bao, and Tao Luo. Generalized tree edit distance (gted): A faithful evaluation metric for statement autoformalization. In *AI for Math Workshop at ICML 2025*, 2025. arXiv:2507.07399.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. Formalalign: Automated alignment evaluation for autoformalization. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.10135.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025. arXiv:2401.08807.

Daniel Mendoza, Anastasia Mavridou, Andreas Katis, and Caroline Trippel. Automating requirements formalization: Using llms and low-complexity distinguishing traces for semantic validation. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026.

Noor Islam S. Mohammad and Tamim Sheikh. The faithfulness gap: Certifying semantic equivalence between natural-language and formal mathematical statements. *arXiv preprint arXiv:2606.16541*, 2026.

Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. Autoformalizing euclidean geometry. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024. arXiv:2405.17216.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. arXiv:2404.13076.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 17958–17980, 2025. arXiv:2406.07222.

Cedric Richter and Heike Wehrheim. Beyond postconditions: Can large language models infer formal contracts for automatic software verification? *arXiv preprint arXiv:2510.12702*, 2025.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *AI Verification (SAIV 2024)*, volume 14846 of *Lecture Notes in Computer Science*. Springer, 2024. arXiv:2310.17807.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025), Datasets and Benchmarks Track*, 2025. arXiv:2505.13938.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *arXiv preprint arXiv:2407.11214*, 2024.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, 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 (ACL)*, 2024. arXiv:2305.17926.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2024. arXiv:2404.00762.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022. arXiv:2205.12615.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean workbook: A large-scale lean problem set formalized from natural language math problems. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*, 2024. arXiv:2406.03847.

Xuejun Yu, Jianyuan Zhong, Zijin Feng, Pengyi Zhai, Roozbeh Yousefzadeh, Wei Chong Ng, Haoxiong Liu, Ziyi Shou, Jing Xiong, Yudong Zhou, Claudia Beth Ong, Austen Jeremy Sugiarto, Yaoxi Zhang, Wai Ming Tai, Huan Cao, Dongcai Lu, Jiacheng Sun, Qiang Xu, Shen Xin, and Zhenguo Li. Mathesis: Towards formal theorem proving from natural languages. *arXiv preprint arXiv:2506.07047*, 2025.

Ke Zhang, Patricio Gallardo Candela, Sudhir Murthy, Yi Xie, Zhi Wang, and Maziar Raissi. Beyond compilation: Evaluating faithful natural-language-to-lean statement formalization. *arXiv preprint arXiv:2606.31002*, 2026.

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 2023), Datasets and Benchmarks Track*, 2023. arXiv:2306.05685.

A MODEL SWEEP TABLE

The values plotted in Figure 3, with the `gemini-3.7-flash` pairing in both roles as an extra row.

Table 4: Model sweep: other judges, with informalizer `claude-haiku-4.5` unless stated. Macro recall (mean of the `confirmed` and `disputed` recalls) ± its 95% interval. The `claimcheck` rows are the variant whose judge also sees the artifact.

	Lean	Isabelle	Rocq	Dafny
`gemini`: baseline	94.6 ±1.2	94.8 ±1.2	91.4 ±1.9	91.4 ±2.9
`gemini`: baseline + list	95.6 ±1.1	95.2 ±1.2	92.2 ±1.8	95.6 ±2.1
`gemini`: claimcheck	95.2 ±1.1	96.1 ±1.1	92.8 ±1.8	95.3 ±2.2
`luna`: baseline	63.4 ±2.1	60.7 ±2.1	61.1 ±2.8	69.0 ±4.6
`luna`: baseline + list	72.8 ±2.2	75.2 ±2.2	69.7 ±2.6	84.0 ±3.8
`luna`: claimcheck	82.0 ±2.1	85.9 ±2.0	82.8 ±2.5	87.6 ±3.3
`deepseek`: baseline	54.7 ±1.2	62.5 ±1.8	55.4 ±1.7	75.1 ±4.2
`deepseek`: baseline + list	55.2 ±1.2	68.7 ±2.0	54.3 ±1.3	80.3 ±3.9
`deepseek`: claimcheck	82.2 ±1.9	83.8 ±2.6	77.9 ±3.4	81.3 ±3.5
`gemini` in both roles	94.7 ±1.2	95.0 ±1.3	91.7 ±1.9	95.3 ±2.2

B VARIANT EXAMPLES

Table 5: Every Dafny variant on one Clover program, `swap`. `C1` is the only class where the requirement is what is wrong: the contract is still a correct swap, but the requirement now asks for something else. `C2` narrows the contract to inputs where `X==0`. `C6` returns the inputs unchanged. The control merges two `ensures` clauses into one, which changes the text but not the meaning.

variant	requirement	contract	label
original	Swap two integers.	`ensures x==Y` `ensures y==X`	`confirmed`
docstring mutated (`C1`)	Assign the second integer to the first integer.	unchanged	`disputed`

contract mutated (`C2`)	Swap two integers.	`requires X==0`
		`ensures x==Y`
		`ensures y==X`
		`ensures y==0`
			`disputed`

contract + body (`C6`)	Swap two integers.	`ensures x==X` `ensures y==Y`	`disputed`
control, meaning-preserving (`merge_ensures`)	Swap two integers.	`ensures (x==Y) && (y==X)`	`confirmed`

Why `C1` is kept and `C3` is not. `C1` has a polarity the rest of the benchmark lacks: the docstring is the thing that is wrong, so the pair disagrees because of its requirement. The fourth released class, `C3`, weakens docstring and contract in tandem. The pair then often still agrees with itself and would be `confirmed` for our question even though Clover rightly calls it incorrect for its own, and that agreement is not reliable enough to label the class by rule in either direction. Its 60 rows are therefore left out of the benchmark, and labelling them by hand is future work. Clover's own annotation of *how* each variant diverges is retained on the rows we keep, so a directional question such as whether a contract guaranteeing more than its docstring claims should count as unfaithful can be asked as a filter rather than baked into the labels.

C EARLIER RESULTS (BEFORE THE ROW-NAME FIX)

Row identifiers in our data record how each row was built: `putnam_1962_a1__control__true_and` on the PutnamBench slices, `abs__C1` on Dafny. An earlier version of the harness rendered that identifier as the artifact's name inside both `claimcheck` prompts, while the baseline prompt never received it, so one arm could read the label off a string the other never saw. The direction of such a leak is not obvious — a judge that trusts a `control` tag accepts rows it should accept, and a `semantic` tag does the reverse — so it does not simply inflate a score, it makes it uninterpretable.

Every figure in this paper was produced after that identifier was cut back to its base form, and no row's name, artifact or context now records how the row was made. The earlier runs are superseded and we do not reproduce them here. They differed in three further ways: the judge was handed the formal artifact alongside the informalization and told to be strict, which is the variant Section 5 now reports second; rows were batched rather than sent one per call; and the evaluation covered a SQL slice derived from Spider, which the benchmark of Section 2 no longer contains. The released code and data include that slice and those runs.