

Beyond Type-checking: Towards Holistic Evaluation of Formal Specification Generation

Srijith Nair*

Department of Electrical & Computer Engineering
The Ohio State University
nair.203@osu.edu

Aditya Vempaty

Emergence AI
aditya@emergence.ai

Jia Liu

Department of Electrical & Computer Engineering
The Ohio State University
liu@ece.osu.edu

Ashish Jagmohan

Emergence AI
ashish@emergence.ai

Abstract

When generating verifiable code, natural language requirements are mapped to machine checked code using LLMs and agentic workflows. A crucial component of this pipeline is specification generation (SpecGen), which produces a formal contract against which an agent can prove implementation correctness. Proof generation can obtain deterministic feedback from a theorem prover, but SpecGen lacks a definitive check that a generated specification captures the user’s intent. A checked proof can therefore establish correctness against a specification that misrepresents the intended behaviour. We take a step towards holistic SpecGen evaluation with a unified dataset assembled from 350 existing Lean tasks, including 189 from VERINA and 161 from CLEVER, and a framework covering formal validity, reference similarity and equivalence, and behavioural adequacy. We distinguish acceptance of required inputs from acceptance of valid outputs and rejection of invalid outputs, while making each metric’s evidence scope explicit. Across four SpecGen configurations, restricting the *generalized tree edit distance* (GTED) comparison, a reference similarity measure, to 32 jointly measurable VERINA tasks changes the VERINA configuration’s position from second to fourth in mean similarity, showing the importance of measurement coverage. In an authored control, a specification achieves 100% positive test recall and negative test rejection while accepting 0% of required inputs. This demonstrates that perfect postcondition scores can miss an unusable input contract, motivating separate feedback on input coverage and output constraints.

1 Introduction

With increasing capabilities of LLMs and agentic workflows, automatic generation of code based on user defined specifications is rapidly becoming a reality. State-of-the-art agentic coding systems like Codex (Chen et al., 2021) and Claude Code (Anthropic, 2026), can generate code in a specified programming language due to training and harness engineering that ground the LLM’s output via planning, breaking the implementation into several manageable pieces and designing test-cases. While impressive, these systems are still limited fundamentally due to their inherently probabilistic nature, and the lack of formal correctness guarantees. Thanks to decades of research in formal methods, *verifiable code generation* (VCG) has emerged as a promising approach to generating code

*Work done as a summer AI intern at Emergence AI.

jointly with formal proofs of correctness. Verifying code correctness against a formally specified contract is a critical component of the development pipeline in domains where mistakes in code can have severe consequences, such as safety critical systems in airplanes (Cousot et al., 2005; Delmas and Souyris, 2007), e-commerce and cloud computing applications (Newcombe et al., 2015), kernel (Konnov et al., 2019) and compiler (Leroy, 2009) development, blockchain (Cassez et al., 2022), defense systems and space applications (Verhulst et al., 2011), among many others.

In the pre-LLM era, formal verification was a laborious and expensive process, requiring highly skilled engineers to manually write formal specifications and proofs. Due to the advent of LLMs, formal verification is now being automated, with LLMs generating formal specifications. This often involves a multi-step process of generating: 1) a formal specification contract from a user’s requirement, specified in natural language called *SpecGen*, and 2) a code implementation that satisfies the specification called *CodeGen*, and 3) obtaining a formal proof of correctness for the implementation against the specification called *ProofGen*. A checked proof establishes correctness relative to the formal contract and its assumptions. Assurance about user intent also requires the contract to capture the requirement correctly.

Formal correctness claims can be checked by tools such as Lean 4 (Moura and Ullrich, 2021), Dafny (Leino, 2010), Verus (Lattuada et al., 2023), Rocq (Barras et al., 1997), etc. These tools check formal correctness claims using proof checking or automated verification. Generated specifications are mathematically expressed using Hoare logic (Hoare, 1969; Floyd, 1993) as follows: Given a pure function C which maps inputs x to outputs y such that $y = C(x)$, if x satisfies the *preconditions* $P(x)$, then y must satisfy the *postcondition* $Q(x, y)$, i.e.,

$$\forall x, y \text{ s.t. } P(x), y = C(x) \implies Q(x, y). \quad (1)$$

Motivation. Proof and code generation can be evaluated deterministically by using feedback from a theorem prover’s compilation process once the user’s intent is captured correctly by the formal specification, and agentic systems like Agarwal et al. (2026) are rapidly becoming better at proving specifications. While ProofGen in itself can still be a complicated task, it is nevertheless, more well-defined than SpecGen, since the latter lacks a definitive feedback mechanism in spite of being a critical component of the VCG pipeline. In SpecGen, the specification must capture a user’s requirements before a proof of implementation correctness can provide the intended assurance.

Evaluating this *intent-specification gap* is a challenging problem, primarily because existing metrics for SpecGen evaluation are limited. For instance, compiler-assisted type checking detects syntax and type errors in a specification, but does not establish agreement with user intent. A well-typed or syntactically correct predicate can still express the wrong relation. Thus, an agent can cheat the evaluation using type-checking. Nevertheless, specification-generation pipelines commonly expose compact signals such as compilation, reference similarity, test success, or equivalence-search success. These signals answer different questions, and their failure modes differ as well. We take a step toward comprehensive SpecGen evaluation by unifying existing evaluation signals and asking: *which dimensions of specification quality do different evaluation signals reveal, and what feedback can they provide to a SpecGen agent?* This enables the development of a high-quality agentic SpecGen system that can leverage this feedback in the right manner.

Contributions. Existing benchmarks already recognize specification generation as a distinct task in the VCG pipeline. VERINA (Ye et al., 2025) supports modular code, specification, and proof evaluation; CLEVER (Thakur et al., 2026) evaluates specification certification relative to a hidden reference. Our contributions add to these benchmarks richer evaluation capabilities and are summarized in the following points:

- Our primary contribution is a SpecGen evaluation framework that unifies multiple evaluation signals and organizes them by the questions they answer and the evaluation modality they require.
- We assemble a unified dataset for SpecGen evaluation, comprising 350 existing Lean SpecGen tasks from the VERINA (Ye et al., 2025) and CLEVER (Thakur et al., 2026) datasets, preserving source provenance information. We also include the VeriCoding (Bursuc et al., 2025) and AlgoVeri (Zhao et al., 2026) datasets, although these aren’t used for the evaluations in this paper. Importantly, we do not claim to introduce newly authored datasets of SpecGen problems.
- Several of our behavioural metrics rely on our hand-engineered extraction tools which extract preconditions, postconditions, Lean signatures and declarations from the generated outputs. Also,

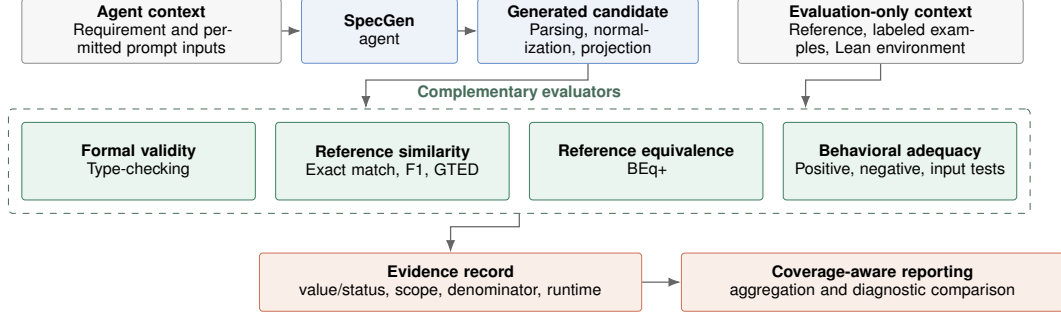


Figure 1: Evaluation-system overview. The SpecGen agent receives only its permitted generation context. The resulting candidate is combined with evaluation-only artifacts by four complementary evaluator families. Each evaluator emits a scoped result before coverage-aware aggregation, so unavailable evidence remains distinct from metric failure.

we provide compatible adapters for the VERINA and CLEVER evaluation harnesses to enable their use on the unified dataset.

- We present initial results covering 4 SpecGen configurations on the 189 VERINA and 161 CLEVER tasks. Our authored controls separately demonstrate why accepting valid outputs, rejecting invalid outputs, and retaining required inputs deserve distinct feedback.

The strongest controlled finding is that perfect postcondition-test scores can coexist with zero required-input recall. This demonstrates a blind spot of postcondition-only feedback, not the prevalence of that error in model outputs. We use compatibility checks and evidence-status reporting to distinguish observed behaviour from unavailable measurements. These engineering checks support the interpretation of the metrics rather than constituting the central research claim.

2 Benchmark Design

2.1 Evaluation-system Overview

Our SpecGen evaluation system is summarized in Figure 1. It treats each generated specification as an artifact to be examined through several complementary forms of evidence. For every task, we combine the candidate with: (i) the available evaluation context, i.e., the Lean environment; (ii) the reference specification available in the dataset; (iii) the natural-language statement (user’s intent); and, (iv) optionally the labeled examples. After representation-specific parsing and projection, independent evaluators measure formal validity, reference similarity, reference equivalence, and behavioral adequacy as detailed in Section 2.3. Each result is stored together with its scope, denominator, runtime, and evidence status. Consequently, unsupported representations, missing dependencies, timeouts, and evaluator failures are made distinguishable from observed metric failures, and downstream reports can compare systems without treating unavailable evidence as a score of zero.

2.2 Datasets and SpecGen agents

We evaluate four productive SpecGen agent configurations on two datasets: 189 VERINA tasks (Ye et al., 2025) and 161 CLEVER tasks (Thakur et al., 2026). The evaluated configurations are:

- a direct-prompted LLM baseline, with no retries or post-processing;
- the VERINA prompting and rendering pipeline (Ye et al., 2025);
- the CLEVER few-shot pipeline (Thakur et al., 2026); and
- a LeanAide adapter (Gadgil et al., 2022).

All configurations use the same base agentic LLM, GPT-5.6 (Sol), in our experiments. They mainly differ in their prompts, output representations, retry budgets, and post-processing, thus representing a progression toward increasingly structured SpecGen pipelines drawn from the literature.

2.3 Supported SpecGen Evaluation Metrics

In this section we describe the evaluation metrics used in our study. Let $\hat{Q}$ denote a generated postcondition candidate, Q its reference specification from the dataset, and $\mathcal{E}$ the recorded Lean environment. We group the evaluation signals into four complementary metric families. Binary and graded metrics are defined only when their required artifacts and evaluator dependencies are available.

- **Formal validity:** This family asks whether the generated artifact is accepted as a Lean declaration in its intended environment. Following standard proof-assistant evaluation (Moura and Ullrich, 2021), the type-checking metric is a boolean signal indicating whether the compiler accepts the specification. This can catch missing dependencies and unsupported representations, contingent on timeouts, and evaluator failures by the Lean environment $\mathcal{E}$. It does not establish agreement with either the reference or the natural-language requirement.
- **Reference similarity:** This family compares $\hat{Q}$ with Q at textual, symbolic, and structural levels. Its metrics quantify resemblance to a supplied reference, but cannot establish logical equivalence rigorously.
 - *Normalized exact match.* Let a normalization function $\mathcal{N}(\cdot)$ remove Lean4 comments, potential Markdown in LLM generated outputs, and the first declaration name, and collapse whitespace. Then the normalized exact match (NEM) metric is a boolean signal indicating:

$$\text{NEM}(Q, \hat{Q}) = \mathbf{1}\{\mathcal{N}(\hat{Q}) = \mathcal{N}(Q)\}, \quad (2)$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. This is a project-specific normalization, so its definition, rather than an external source, determines the metric.

- *Signature F1.* Let $S_{\hat{Q}}$ and S_Q be the multisets of normalized features extracted from $\hat{Q}$ and Q . For multisets A and B , define their overlap by

$$I(A, B) = \sum_u \min\{m_A(u), m_B(u)\}, \quad (3)$$

where $m_A(u)$ is the multiplicity of u in A . Less formally, $I(A, B)$ counts the number of features that appear in both multisets. We define the signature precision, recall, and F1 as

$$P_{\text{sig}} = \frac{I(S_{\hat{Q}}, S_Q)}{|S_{\hat{Q}}|}, \quad R_{\text{sig}} = \frac{I(S_{\hat{Q}}, S_Q)}{|S_Q|}, \quad F_{1,\text{sig}} = \frac{2P_{\text{sig}}R_{\text{sig}}}{P_{\text{sig}} + R_{\text{sig}}}. \quad (4)$$

Thus repeated binders and types contribute according to their multiplicities. The evaluator assigns F1 of one when both feature multisets are empty, and zero when only one is empty or their overlap is zero. Empty extraction can reflect unsupported declaration forms, so a score of one need not indicate matching signatures.

- *Logical-atom F1.* Let $A_{\hat{Q}}$ and A_Q be the multisets of normalized identifiers and supported logical and arithmetic operators extracted from the two statements. The current tokenizer omits numeric literals and the Unicode operators $\leq$, $\geq$, and $\neq$, limiting sensitivity to changes in these features. Using the same multiset overlap in (3), we define the logical-atom precision, recall, and F1 as:

$$P_{\text{atom}} = \frac{I(A_{\hat{Q}}, A_Q)}{|A_{\hat{Q}}|}, \quad R_{\text{atom}} = \frac{I(A_{\hat{Q}}, A_Q)}{|A_Q|}, \quad F_{1,\text{atom}} = \frac{2P_{\text{atom}}R_{\text{atom}}}{P_{\text{atom}} + R_{\text{atom}}}. \quad (5)$$

Signature and logical-atom F1 instantiate the standard F-score with different feature extractors.

- *Generalized tree-edit similarity (or distance) (GTED).* GTED (Liu et al., 2025) standardizes each statement using Lean elaboration and its *language server protocol* (LSP) hover information, constructs operator trees $\mathcal{T}_{\hat{Q}}$ and $\mathcal{T}_Q$, and computes their tree-edit distance $d_{\text{TED}}(\mathcal{T}_{\hat{Q}}, \mathcal{T}_Q)$. Matching the released GTED implementation in Liu et al. (2025), we report the normalized similarity

$$\text{GTED}(\hat{Q}, Q) = 1 - \frac{d_{\text{TED}}(\mathcal{T}_{\hat{Q}}, \mathcal{T}_Q)}{\max\{|\mathcal{T}_{\hat{Q}}|, |\mathcal{T}_Q|\}}, \quad (6)$$

where $|T|$ is the number of nodes in T . Insertions, deletions, and unequal-label substitutions have unit cost; equal-label substitutions have zero cost. Consequently, $\text{GTED}(\hat{Q}, Q) \in [0, 1]$, with 1 indicating identical operator trees under the adopted construction.

For reference similarity metrics, a high score does not necessarily indicate that the candidate and reference are logically equivalent.

- **Reference equivalence:** This family seeks formal evidence that the candidate and reference express equivalent propositions. Let $\pi(\hat{Q})$ and $\pi(Q)$ denote *projected* candidate and reference statements. BEq^+ (Poiroux et al., 2025) attempts to prove the bidirectional implication:

$$\mathcal{E}, \pi(\hat{Q}) \vdash \pi(Q) \quad \text{and} \quad \mathcal{E}, \pi(Q) \vdash \pi(\hat{Q}). \quad (7)$$

Writing $q_{\rightarrow}$ and $q_{\leftarrow}$ for the corresponding Lean-checked proof outcomes, its success criterion is

$$\text{BEq}^+(\hat{Q}, Q) = q_{\rightarrow} \wedge q_{\leftarrow}. \quad (8)$$

A successful check involves constructing a Lean kernel checked evidence of equivalence for the compared projections. While BEq is one of the strongest signals of semantic agreement, constructing equivalence proofs is a challenging task in itself, and a proof-search such as this often fails. Failure to find either proof is classified as inconclusive, since search failure does not establish non-equivalence.

- **Behavioral adequacy:** For VERINA tasks with compatible candidate representations, this family evaluates the candidate on finite labeled examples. Let the candidate contract consist of a precondition $\hat{P}(x)$ and postcondition $\hat{Q}(x, y)$. Additionally, let T^+ contain intended input-output pairs that should pass the specification, T^- the unintended pairs, and D_{req} the supplied required inputs.

- *Positive-test recall* is given by:

$$R_+(\hat{Q}) = \frac{1}{|T^+|} \sum_{(x,y) \in T^+} \mathbf{1}\{\hat{Q}(x, y)\}. \quad (9)$$

It measures the fraction of supplied intended behaviors admitted by the candidate postcondition.

- *Negative-test rejection* is given by:

$$R_-(\hat{Q}) = \frac{1}{|T^-|} \sum_{(x,y) \in T^-} \mathbf{1}\{\neg \hat{Q}(x, y)\}. \quad (10)$$

It measures the fraction of supplied unintended behaviors excluded by the postcondition.

- *Behavioral balanced accuracy* gives the two test classes equal weight, i.e.:

$$\text{BAcc}(\hat{Q}) = \frac{R_+(\hat{Q}) + R_-(\hat{Q})}{2}. \quad (11)$$

This is the usual balanced-accuracy construction (Brodersen et al., 2010), applied to behavioral acceptance and rejection.

- *VERINA pass@I* is the binary finite-suite criterion given by:

$$\text{Pass@1}_{\text{VERINA}}(\hat{Q}) = \mathbf{1}\{R_+(\hat{Q}) = 1 \wedge R_-(\hat{Q}) = 1\}. \quad (12)$$

At the system level, the reported rate over tasks $i = 1, \dots, n$ is

$$\text{Pass@1} = \frac{1}{n} \sum_{i=1}^n \text{Pass@1}_{\text{VERINA}}(\hat{Q}_i). \quad (13)$$

This is the $k = 1$ specialization of $\text{pass}@k$ reporting (Chen et al., 2021), with success defined here as passing both finite VERINA test classes.

- *Required-input recall* evaluates the precondition independently:

$$R_{\text{req}}(\hat{P}) = \frac{1}{|D_{\text{req}}|} \sum_{x \in D_{\text{req}}} \mathbf{1}\{\hat{P}(x)\}. \quad (14)$$

This prevents input domain coverage from being conflated with postcondition behavior. In particular, a false precondition can coexist with $R_+ = R_- = 1$ while yielding $R_{\text{req}} = 0$.

These scores establish behavior only on the supplied finite examples. They neither prove universal correctness nor show that the test oracle fully captures the natural-language requirement.

Soundness and completeness status: Categorical summaries derived from the negative- and positive-example outcomes used above. They are not independent measurements, but rather a convenient way to report the two test-class results. We retain the benchmark’s terminology while reporting R_+ and R_- explicitly to make the operational meaning unambiguous.

The benchmark additionally provides *LeanScorer* (Xuejun et al., 2025), a graded, requirement-conditioned LLM assessment of intent alignment. The integrated implementation aggregates its judgments using a *Sugeno fuzzy integral* (Sugeno, 1972). We treat it as an auxiliary learned assessment outside the four primary metric families.

Evaluation time is recorded separately as telemetry. Every conditional metric is reported with its applicable denominator and evidence status, and unavailable evidence is represented by $\perp$, rather than by a numeric zero or a semantic-failure label.

3 Experimental Results

3.1 Implementation details

We analyze the performance of our four agents on VERINA and CLEVER datasets, using the configurations introduced in Section 2.2. Each configuration attempts every task in the corresponding dataset, giving 756 VERINA and 644 CLEVER task–configuration records. Generations are checkpointed, and their manifests record the model, prompt and adapter fingerprints, and task identities. Both runs configure medium reasoning effort for GPT-5.6 (Sol), an initial output-token limit of 4,096, and a ceiling of 16,384; however, due to varying prompts and harnesses, this cannot ensure equal realized token use. Because candidate budgets and repair procedures differ, these experiments compare complete agent configurations, and they do not completely isolate the effect of prompting or repair.

We report candidate production and recorded type-checking success over all attempted tasks, including generation failures in the denominator. Logical-atom F1 averages include the saved zero scores for generation failures. GTED averages use available measurements, with their counts reported explicitly. The shared reference projection uses Lean’s Mathlib 4.20 library. Our reference audit (Appendix B) also checks the native VERINA and CLEVER environments (Lean 4.18 and 4.24) to ensure consistency. Behavioral results are available only for VERINA in these runs. CLEVER contains test cases, but these runs lack the compatible VERINA-format test metadata required by the behavioral evaluator.

3.2 VERINA and CLEVER results

Formal validity and candidate coverage. Figure 2 shows that most configurations obtain high recorded type-checking rates, but candidate availability also matters. On VERINA, the VERINA and CLEVER configurations each return 189 candidates, of which 173 (91.5%) typecheck; the prompted baseline returns 189, with 161 (85.2%) typechecking. LeanAide returns 181 candidates, all recorded as typechecking, for an overall rate of 95.8%. On CLEVER, the CLEVER configuration and prompted baseline return all 161 candidates, with 145 (90.1%) and 138 (85.7%) typechecking. LeanAide’s conditional rate remains 100%, but it returns only 135/161 candidates (83.9%). The VERINA configuration returns 145/161 candidates and records 111/161 type-checking successes (68.9%). Thus, a perfect conditional type-checking rate can conceal failures to produce a candidate. These observations do not establish that repair causes better specifications, since the configurations differ in more than their repair steps.

Reference resemblance depends on both dataset and measurement coverage. The prompted baseline has the largest conditional mean GTED on VERINA, 0.349, but only 46/189 measurements (24.3%). The VERINA configuration obtains 0.335 on 115 tasks, while CLEVER and LeanAide obtain 0.232 and 0.243 on 150 and 143 tasks. On CLEVER, the CLEVER configuration has mean GTED 0.321 on 124/161 tasks, compared with 0.233 on 107 tasks for LeanAide and 0.125 on 156 tasks for the prompted baseline. No GTED score is available for the VERINA configuration on this dataset. The apparent ordering therefore changes across datasets, and the averages describe different measurable subsets. They cannot be interpreted as a controlled semantic-quality ranking.

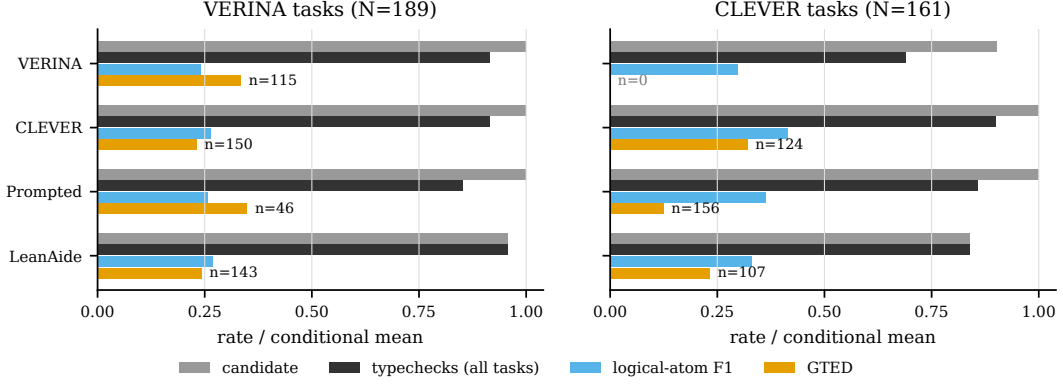


Figure 2: Saved-run profiles on VERINA and CLEVER. Candidate production and type-checking rates use all attempted tasks. Logical-atom F1 means include saved generation-failure zeros. GTED means use available scores; annotations give GTED sample sizes. The VERINA configuration has no GTED measurements on CLEVER ($n = 0$), not a measured similarity of zero. Type-checking records can concern different candidate projections or an adapter’s internal check.

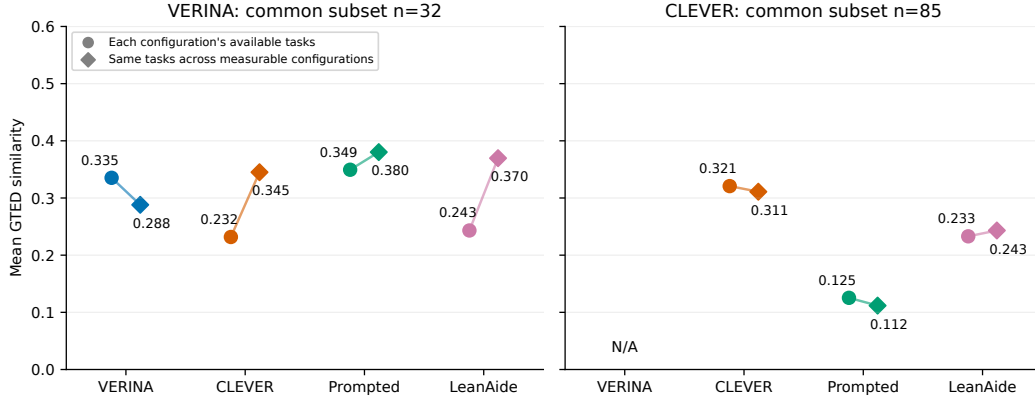


Figure 3: GTED means before and after restricting to jointly measurable tasks. Circles use each configuration’s available measurements; diamonds use the intersection of task IDs: 32 VERINA tasks across four configurations and 85 CLEVER tasks across three. The VERINA configuration has no GTED measurements on CLEVER. Both comparisons retain the historical projections; matching tasks does not establish semantic quality.

The comparison in Figure 3 uses each configuration’s available measurements for, say GTED. This comparison is not controlled, since the support of each agent is different. In order to normalize the comparison, we restrict the analysis to the intersection of task IDs which have GTED measurements for all configurations. This yields 32 tasks on VERINA and 85 tasks (excluding the VERINA agent) on CLEVER. The VERINA agent’s mean decreases from 0.335 to 0.288, while the CLEVER and LeanAide means increase from 0.232 and 0.243 to 0.345 and 0.370. The prompted baseline increases from 0.349 to 0.380. The VERINA configuration therefore moves from second to fourth in observed mean GTED. On CLEVER, the 85-task intersection among the three measurable configurations preserves their ordering, with means of 0.311, 0.243, and 0.112 for CLEVER, LeanAide, and the prompted baseline. Thus, unequal measurement coverage can change the comparison, although it does not do so uniformly across datasets. Note that the intersections cover only 16.9% and 52.8% of the respective datasets and are selected by metric availability; thus they are not representative samples.

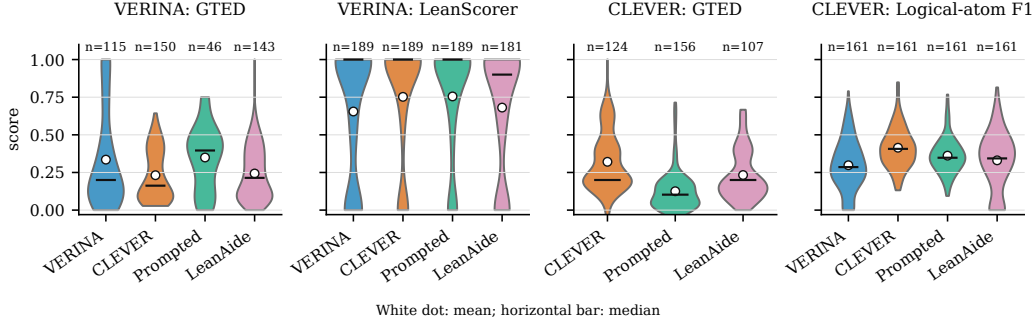


Figure 4: Distributions of salient conditional metrics. White circles denote means, horizontal bars denote medians, and annotations report available measurements. Missing values are omitted rather than scored as zero.

Logical-atom F1 adds a different view: on VERINA its means lie between 0.241 and 0.269 across configurations, whereas on CLEVER they range from 0.298 to 0.414. In particular, the prompted baseline’s relatively high VERINA GTED does not coincide with the highest logical-atom F1. Neither signal supplies a correctness threshold. Moreover, inspecting the selected declaration is necessary even when a structural score is available: the CLEVER examples in Appendix A show a reference helper selected in place of the intended contract. These measurements characterize the recorded representations, with that limitation, rather than establishing intent fidelity. Appendix B.1 examines the score distributions and concrete normalization failures, including recorded GTED scores of one that compare fragments rather than complete contracts.

Natural-language alignment. On the VERINA inventory, mean LeanScorer values are 0.655, 0.752, 0.755, and 0.681 for VERINA, CLEVER, Prompted, and LeanAide, respectively, over 189, 189, 189, and 181 available generations. Figure 4 shows the corresponding LeanScorer distributions alongside selected structural metrics. Note that LeanScorer is an LLM-based judge and not a formal guarantee. A complete scalar-metric table that tests the four agentic methods on VERINA and CLEVER, is reported in Appendix B.3.

Behavioral outcomes require an evidence audit. Across the four VERINA configurations, the historical evaluator reports 185 finite-test passes, 469 nonpasses, and 102 unavailable records. Of the 469 nonpasses, 357 (76.1%) carry explicit harness errors; 112 lack sufficient evidence to identify a witnessed semantic failure. For example, the VERINA configuration records 173 type-checking successes, yet all 165 of its behavioral nonpasses have harness errors. Its attributed precondition and postcondition declarations are not consistently preserved by the behavioral projection. This disagreement primarily exposes incompatible evaluation paths that prevent behavioral evaluation.

The CLEVER configuration, prompted baseline, and LeanAide have 95, 29, and 61 reported finite passes, respectively. These remain historical finite-test outcomes, not independently revalidated correctness claims. Six records have perfect positive-test recall but imperfect negative-test rejection; unresolved proof or execution failures prevent counting them as six confirmed intent errors. Appendix B gives the complete status breakdown. The CLEVER dataset has no behavioral measurements in these runs, and neither dataset provides real-output required-input recall. Consequently, the experiments do not estimate the prevalence of restrictive preconditions in model-generated specifications.

Although these runs cannot measure the prevalence of restrictive preconditions, authored controls let us check whether the three behavioral metrics respond separately to known postcondition and input-domain defects. Appendix B.4 reports the controls, results, and full interpretation.

4 Related Work, Limitations, and Future Work

Relationship to prior work. VERINA (Ye et al., 2025) evaluates specification generation and separates it from code and proof generation. CLEVER’s (Thakur et al., 2026) specification certification explicitly targets agreement with a reference, making the trustworthiness and preservation of that reference consequential. LeanEuclid (Murphy et al., 2024) supplies semantic evaluation in a controlled geometry domain for mathematics; semantic evaluation itself is not our novelty. VeriSpecGen (Ye et al., 2026) goes further toward requirement-level validation through traceability and targeted repair. Our contribution is complementary: a unified evaluation framework relating multiple signals to their feedback roles, with a controlled demonstration of why input-domain coverage must be assessed separately from postcondition behaviour. We explicitly identify the intent gap, which is a general phenomenon that can be observed in multiple prior works (Xie et al., 2026), and is a key challenge in verifiable code generation. We do not claim that prior literature ignored the intent gap, and our findings merely provide evidence for this observation. Importantly, we would also like to note that our evaluation framework was implemented with the aid of Codex, which is an AI coding assistant, but was rigorously tested and audited before evaluations were conducted.

Limitations. Our study covers one model and two Lean datasets, with configurations that use different generation budgets and repair procedures. Uneven metric coverage, projection failures, and historical evaluation errors limit comparisons of specification quality. Reference agreement and finite tests do not establish fidelity to the natural-language requirement. The authored controls demonstrate a possible input-coverage failure, but its prevalence in generated outputs remains unmeasured. We also do not establish that the metrics provide independent information or improve generation when used as feedback.

Potential future directions. Future work should improve preservation of complete contracts across evaluators, reassess affected scores, and measure required-input coverage in generated specifications. Independently reviewed references and checked counterexamples could strengthen evaluation against user intent. Studies across more models and domains, using matched budgets and held-out tasks, should test whether combining these signals improves specification quality over type-checking feedback alone, and at what generation cost. A more comprehensive evaluation will extend to additional data sources in our benchmarking suite, including VeriCoding (Bursuc et al., 2025) and AlgoVeri (Zhao et al., 2026). Additionally, we plan to extend the system to incorporate Dafny (Leino, 2010) and Verus (Lattuada et al., 2023) SpecGen, as well as principled evaluations of LEAN ProofGen.

5 Conclusion

We presented a framework for evaluating specification generation through multiple signals and applied it to existing VERINA and CLEVER tasks. The results show why formal validity, reference resemblance, and finite behavioral outcomes require separate interpretation: comparisons depend on measurement coverage, and evaluator failures can masquerade as specification failures. Authored controls further demonstrate that perfect postcondition scores can coexist with exclusion of every required input. Together, these findings support evaluating complete contracts and reporting the scope and availability of each signal. Establishing whether this richer feedback improves agents’ fidelity to user intent remains an important next step.

References

- Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, et al. Inductive deductive synthesis: Enabling ai to generate formally verified systems. *arXiv preprint arXiv:2605.23109*, 2026.
- Anthropic. Claude code, 2026. URL <https://claude.ai>.
- Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. URL <https://inria.hal.science/inria-00069968>. Projet COQ.
- Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. The balanced accuracy and its posterior distribution. In *2010 20th International Conference on Pattern Recognition*, pages 3121–3124, 2010. doi: 10.1109/ICPR.2010.764.
- Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, et al. A benchmark for vericoding: formally verified program synthesis. *arXiv preprint arXiv:2509.22908*, 2025.
- Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the ethereum 2.0 beacon chain. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–182. Springer, 2022.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.
- David Delmas and Jean Souyris. Astrée: From research to industry. In *International Static Analysis Symposium*, pages 437–451. Springer, 2007.
- Robert W Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993.
- Siddhartha Gadgil, Anand Rao Tadipatri, Ayush Agrawal, Ashvni Narayanan, and Navin Goyal. Towards automating formalisation of theorem statements using large language models. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on MATH-AI*, 2022.
- Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. Tla+ model checking made symbolic. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.
- Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.
- K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.
- Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

- 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. *arXiv preprint arXiv:2507.07399*, 2025.
- Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.
- Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. Auto-formalizing euclidean geometry. In *International Conference on Machine Learning*, 2024. URL <https://arxiv.org/abs/2405.17216>.
- Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.
- Auguste Poiroux, Gail Weiss, Viktor Kunčák, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17958–17980, 2025.
- Michio Sugeno. Fuzzy measure and fuzzy integral. *Transactions of the Society of Instrument and Control Engineers*, 8(2):218–226, 1972. doi: 10.9746/sicetr1965.8.218.
- Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL <https://openreview.net/forum?id=Ib0acMF5qd>.
- Eric Verhulst, Raymond T Boute, José Miguel Sampaio Faria, Bernhard HC Sputh, and Vitaliy Mezhyuev. *Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems*. Springer Science & Business Media, 2011.
- Zichen Xie, Mrigank Pawagi, Yuxin Liu, Aaditi Rai, Lize Shao, Jr. Berberian, John, Sicong Che, and Wenxi Wang. Vericontest: A competitive-programming benchmark for verifiable code generation. *arXiv preprint arXiv:2605.08553*, 2026. URL <https://arxiv.org/abs/2605.08553>.
- Yu Xuejun, Jianyuan Zhong, Zijin Feng, Pengyi Zhai, Roozbeh Yousefzadeh, Wei Chong Ng, Haoxiong Liu, Ziyi Shou, Jing Xiong, Yudong Zhou, et al. Mathesis: Towards formal theorem proving from natural languages. *arXiv preprint arXiv:2506.07047*, 2025.
- Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL <https://openreview.net/forum?id=47601CQFri>.
- Zhe Ye, Aidan Z. H. Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned formal specification synthesis via traceable refinement. *arXiv preprint arXiv:2604.10392*, 2026. URL <https://arxiv.org/abs/2604.10392>.
- Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms. *arXiv preprint arXiv:2602.09464*, 2026.

A Actual VERINA and CLEVER examples

These four rows come from saved benchmark runs, not authored controls. All use the CLEVER configuration: the September 10 run on VERINA and the September 12 run on CLEVER. Problem statements are source excerpts; listings retain all executable specification text, omitting only full-line comments. VERINA references contain a precondition followed by a postcondition; CLEVER references are complete declarations. Tables reproduce saved metrics rounded to three decimals. TC denotes type-checking, NEM denotes normalized exact match, and a dash denotes unavailable evidence. BEq 0 means no equivalence proof was obtained, not a proof of inequivalence. Required-input recall is unavailable for all four rows. Signature F1 is one in all four rows because both extracted feature multisets are empty. This reflects the evaluator’s empty-feature convention and does not establish signature agreement.

A.1 Integer square root: equivalence despite moderate similarity

Dataset row: VERINA, `verina_basic_65`.

Natural-language problem (source excerpt).

This task involves computing the integer square root of a given natural number. The goal is to determine the largest natural number r that satisfies $r * r \leq N$ and $N < (r + 1) * (r + 1)$.

Generated specification

```
def generated_spec
  (impl: Nat → Nat)
  (N: Nat) : Prop :=
  let r := impl N
  r * r ≤ N ∧ N < (r + 1) * (r + 1)
```

Reference specification (precondition / postcondition)

```
True

result * result ≤ N ∧ N < (result + 1) * (result + 1)
```

TC	NEM	Sig. F1	Atom F1	GTED	BEq	R_+	R_-
yes	0	1.000	0.444	0.435	1	1	1

Interpretation. The candidate binds `r := impl N` and imposes the same two bounds as the reference. Substituting the observed result for the implementation output makes these bounds coincide. The saved BEq check succeeds, and the behavioral record reports all 5 positive and 13 negative examples passing, with no harness-error flag. These remain recorded finite-test outcomes. NEM is zero because the complete declaration and paired reference body are textually different. Logical-atom F1 is 0.444 because the feature comparison includes different declaration and binding forms, not just the arithmetic relation. GTED compares an application of the `generatedSpec` wrapper against the explicit reference formula. Its recorded edit distance is 13 and larger tree size is 23, giving $1 - 13/23 = 0.435$. Moderate structural resemblance therefore coexists with a successful equivalence check on this row.

A.2 Sorted deduplication: behavioral zeros without counterexamples

Dataset row: VERINA, verina_advanced_81.

Natural-language problem (source excerpt).

Implement a Lean 4 function that, given a list of integers, removes all duplicates and returns the resulting list in ascending order.

Generated specification

```
def generated_spec
  (impl: List Int → List Int)
  (arr: List Int) : Prop :=
  ∃ result : List Int,
    impl arr = result ∧
    result.Nodup ∧
    result.Pairwise (fun a b : Int => a < b) ∧
    ∀ x : Int, x ∈ result ↔ x ∈ arr
```

Reference specification (precondition / postcondition)

True

List.isPerm arr.eraseDups result ∧ List.Pairwise ($\cdot \leq \cdot$) result

TC	NEM	Sig. F1	Atom F1	GTED	BEq	R_+	R_-
yes	0	1.000	0.171	0.423	0	0	0

Interpretation. The reference requires a permutation of the deduplicated input and nondecreasing order. The candidate states no duplicates, strict pairwise order, and equality of input/output membership. These clauses describe the same intended sorted set of integers through different predicates; the saved BEq search does not establish their equivalence. Different predicate vocabulary also makes exact match zero and logical-atom F1 low (0.171). GTED retains a candidate wrapper application rather than expanding its clauses: distance 15 over a larger tree of size 26 gives $1 - 15/26 = 0.423$.

The historical behavioral evaluator records 0/5 positive successes and 0/15 negative rejections. Its diagnostics report failure to synthesize `Decidable` for the existential and quantified test propositions. For example, a negative test supplies input `[5, 2, 2, 5]` and output `[2, 2, 5]`: the displayed candidate excludes that output through `Nodup`, yet the attempted test fails to obtain decidability. The recorded zeros do not demonstrate acceptance of this invalid output or rejection of valid outputs. This row illustrates a successful type check with unresolved behavioral evidence.

A.3 Integer addition: an inconclusive equivalence search

Dataset row: CLEVER, add(x : Int, y : Int) (row 52).

Natural-language problem (source excerpt).

Add two numbers x and y .

Generated specification

```
def generated_spec
  (impl: Int → Int → Int)
  (x y: Int) : Prop :=
  impl x y = x + y
```

Reference specification

```
def problem_spec
  (impl: Int → Int → Int)
  (x y: Int) :=
  let spec (res: Int) :=
    res - x - y = 0
  ∃ result, impl x y = result ∧
  spec result
```

TC	NEM	Sig. F1	Atom F1	GTED	BEq	R_+	R_-
yes	0	1.000	0.538	0.200	0	–	–

Interpretation. The candidate equates the implementation output with $x + y$. The reference introduces an existential result and requires $result - x - y = 0$. Over integers, substituting the implementation output and rearranging the equality gives the candidate’s condition. This algebraic reading explains why BEq 0 must remain inconclusive: it is not evidence that these contracts disagree. The saved record provides no successful proof for this comparison.

Exact match is zero and logical-atom F1 is 0.538 because the reference uses subtraction, an auxiliary predicate, and an existential equality rather than direct addition. The GTED intermediates are theorem TBD: $:=$ by sorry and theorem spec : $\text{Int} \rightarrow \text{Prop} :=$ by sorry; distance 4 and larger tree size 5 yield $1 - 4/5 = 0.200$. These fragments do not preserve the displayed contracts, so the score cannot be read as the similarity of their arithmetic meanings. Behavioral fields are unavailable because this run lacks the VERINA-format test metadata required by that evaluator, rather than because any test failed.

A.4 Triangle area: reference agreement is a separate question

Dataset row: CLEVER, triangle_area(a: float, h: float) (row 44).

Natural-language problem (source excerpt).

Given length of a side and high return area for a triangle.

Generated specification

```
def generated_spec
  (impl: Rat → Rat → Rat)
  (a h: Rat) : Prop :=
  0 ≤ a ∧ 0 ≤ h → impl a h = (a * h) / 2
```

Reference specification

```
def problem_spec
  (implementation: Rat → Rat → Rat)
  (a h: Rat) :=
  let spec (result: Rat) :=
    a = 0 → result = 0 ∧
    (a ≠ 0 → (2 * result) / a = h);
  ∃ result, implementation a h = result ∧
  spec result
```

TC	NEM	Sig. F1	Atom F1	GTED	BEq	R_+	R_-
yes	0	1.000	0.688	0.200	0	–	–

Interpretation. For nonnegative base and height, the candidate requires the area $ah/2$. The reference’s outer implication instead guards the entire conjunction with $a = 0$. When $a \neq 0$, that implication places no constraint on the result. Reading the displayed formulas at $a = 2, h = 3$, an implementation output of 999 satisfies the reference’s local obligation but violates the candidate’s required output of 3. This is a direct logical reading of this row, not a newly executed Lean check or a claim inferred from BEq 0. The candidate also leaves behavior outside its nonnegative-input guard unspecified.

Both declarations typecheck. Logical-atom F1 of 0.688 reflects shared arithmetic and comparison vocabulary, but does not capture the consequence of moving an implication guard. NEM is zero. As in the addition row, GTED normalization reduces the candidate to TBD: and the reference to its local predicate type, giving distance 4, larger tree size 5, and score 0.200. That score does not diagnose the guard issue. No behavioral measurements are available. This example separates the natural-language requirement, the generated contract, the reference contract, and what the recorded metrics actually check.

B Audits supporting measurement validity

Source correspondence and environment compatibility. We distinguish full upstream contracts, exported contracts, and the shared evaluator’s projected statements. Native checks retain the required auxiliaries and pre/post contracts while excluding implementations and correctness proofs. All 350 upstream contracts compile in their pinned native environments. Table 1 shows why a failed common projection cannot be attributed directly to the source reference. The four CLEVER export failures use `List.indexOf` where the upstream contracts use `List.idxOf`; this is a compatibility finding in Lean 4.24. The 16 CLEVER projection failures involve missing import or source-local helper context. VERINA projection failures also include declaration-selection and pre/post representation issues. Because both context and representation change, this comparison does not isolate a Lean-version effect.

Table 1: Reference compilation counts. Native and export checks use the source environment; the common projection uses Mathlib 4.20 and a different artifact.

Dataset	Tasks	Upstream native	Export native	Common projection
VERINA	189	189	189	145
CLEVER	161	161	157	145

The source/export audit records 62 VERINA and 20 CLEVER textual differences. Textual differences and an LLM-assisted alignment screen are not adjudicated semantic defect counts. Likewise, native compilation does not certify the reference’s agreement with its natural-language requirement. The retained counterexample recheck timed out importing Mathlib; we therefore do not use that recheck as successful independent certification of reference witnesses.

Feature extraction and missing-score conventions. Inspection of the four examples in Appendix A found empty signature features on both sides of every comparison. Their saved Signature F1 scores of one follow the empty-feature convention, without demonstrating signature agreement. Logical-atom extraction omits numeric literals and some Unicode comparison operators. We retain the historical scores and interpret them within these limits. Aggregation also distinguishes saved logical-atom zeros for generation failures from unavailable GTED measurements, which are excluded from GTED means.

B.1 Structural-score distributions and projection checks

Figure 5 shows the distributions underlying the conditional means. On VERINA, the VERINA configuration has mean GTED 0.335 but median 0.200; 14 of its 115 available scores equal one. The CLEVER, prompted, and LeanAide configurations have medians 0.162, 0.396, and 0.214. On CLEVER, the CLEVER and LeanAide configurations both have median 0.200, despite means of 0.321 and 0.233; the prompted baseline has median 0.103 and mean 0.125. These distributions expose features hidden by averages, but their upper tails require inspection of the compared artifacts.

For example, the VERINA configuration’s output for `verina_basic_39` has GTED 1 and logical-atom F1 0.182, but the normalized comparison reduces to `l : List Int` versus `result : List Int`. For `verina_basic_35`, GTED is also 1 and logical-atom F1 is 0.050, while the corresponding fragments are `arr : Array Int` and `result : Array Int`. For `verina_advanced_45`, GTED 1 coexists with logical-atom F1 0.082 and normalized theorem strings containing only `List` and `TBD:`, respectively, before the placeholder proof. These are evaluator intermediates, not the original generated specifications. They show that a perfect recorded GTED score need not compare complete contracts. We retain the historical scores and do not infer that all fourteen perfect-score records share the same defect; they have not all been individually adjudicated. Restricting comparisons to common task IDs in Figure 3 does not correct these normalization failures.

Within each dataset–configuration group with GTED measurements, Pearson correlations between logical-atom F1 and GTED range from -0.238 to 0.219 , computed on available pairs. The weak linear associations are descriptive only: projection failures and discrete score patterns prevent interpreting them as evidence of statistical independence or complementary semantic information. Likewise, an inventory of numerically available metrics is insufficient to establish valid evidence, because historical behavioral scores can coexist with execution errors.

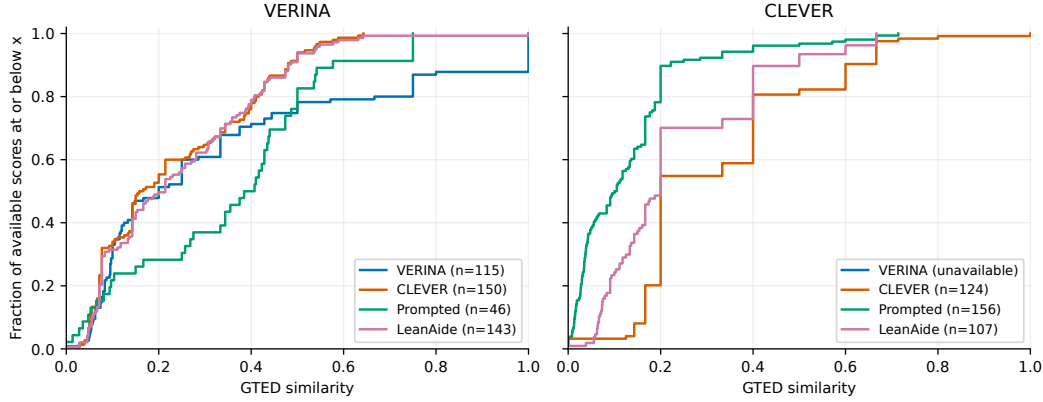


Figure 5: Empirical cumulative distributions of available historical GTED scores, with measurement counts in the legends. Each curve uses its own available subset. These are distributions of recorded structural scores, not semantic accuracy; inspected high-score examples contain incomplete normalizations.

B.2 Behavioral evidence and cohort checks

Distinguishing failed evaluation from observed behavior. We inspect availability, explicit harness-error flags, and retained diagnostics before interpreting a historical behavioral score. An explicit harness error takes precedence over its number; unsupported cases and generation failures remain separate. A failed proof attempt without a complete witness remains unresolved. A reported finite pass requires both recorded test-class scores to equal one without an inspected evaluator-error flag. Table 2 preserves these distinctions for all 756 VERINA attempts.

Table 2: Retrospective VERINA evidence status, 189 attempts per configuration. Pass denotes a reported finite pass; unresolved denotes a nonpass without sufficient evidence of a semantic failure. No historical pass is upgraded by this classification.

Configuration	Pass	Harness error	Unresolved	Unsupported	No output
VERINA	0	165	0	24	0
CLEVER	95	11	59	24	0
Prompted	29	127	9	24	0
LeanAide	61	54	44	22	8

Executable outcomes and completeness of observations. The revised behavioral path evaluates decidable propositions and records indexed Boolean outcomes, instead of inferring failed cases from failed `native_decide` proof attempts. Compilation failure, missing decidability, or incomplete output withholds the score. The behavioral path also rejects candidate commands and admissions such as `sorry` and `axiom`, protecting the interpretation of executed test outcomes. The controls in Appendix B.4 document the available runtime evidence and its remaining gap. A separate replay selected five non-demonstration task IDs by the lowest SHA256 hashes of a fixed seed followed by each ID, shared across all four configurations. However, the retained replay contains no completed task results and reports Lean-session timeouts in its acceptance checks. It does not revalidate the historical cohort or justify replacing its scores.

Cohort identity and demonstration overlap. Saved manifests associate task identities with generation fingerprints and result-file hashes, allowing the audit to track the same outputs across evaluation paths. The reference audit pins upstream revisions separately from the generation adapters. The generation interface separates requirement and signature context from evaluation-only references and tests. A secondary VERINA profile excludes the two recorded demonstrations, leaving 187 tasks per configuration. Its pass/harness-error/unresolved/unsupported/no-output counts are, respectively,

Table 3: Mean metric values for the four evaluated agents. Bracketed values are the number of available measurements; candidate production and type-checking use all attempts. Best available values within each dataset are bold (ties included). LeanScorer was added later for the exact VERINA paper inventory and was not run on CLEVER.

Metric	VERINA	CLEVER	Prompted	LeanAide
VERINA tasks				
Candidate production	1.000 [189]	1.000 [189]	1.000 [189]	0.958 [189]
Type-check rate (all attempts)	0.915 [189]	0.915 [189]	0.852 [189]	0.958 [189]
Normalized exact match	0.000 [189]	0.000 [189]	0.000 [189]	0.000 [189]
Signature F1	0.117 [189]	0.836 [189]	0.228 [189]	0.088 [189]
Logical-atom F1	0.241 [189]	0.265 [189]	0.258 [189]	0.269 [189]
BEq+	0.000 [189]	0.021 [189]	0.011 [94]	0.011 [181]
GTED	0.335 [115]	0.232 [150]	0.349 [46]	0.243 [143]
LeanScorer	0.655 [189]	0.752 [189]	0.755 [189]	0.681 [181]
Behavioral balanced accuracy	0.000 [165]	0.596 [165]	0.182 [165]	0.395 [159]
Positive-test recall	0.000 [165]	0.594 [165]	0.182 [165]	0.396 [159]
Negative-test rejection	0.000 [165]	0.598 [165]	0.181 [165]	0.395 [159]
VERINA pass@1	0.000 [165]	0.576 [165]	0.176 [165]	0.384 [159]
Evaluation time (s)	1.0950 [189]	0.9159 [189]	0.8256 [189]	1.4947 [189]
CLEVER tasks				
Candidate production	0.901 [161]	1.000 [161]	1.000 [161]	0.839 [161]
Type-check rate (all attempts)	0.689 [161]	0.901 [161]	0.857 [161]	0.839 [161]
Normalized exact match	0.000 [161]	0.000 [161]	0.000 [161]	0.000 [161]
Signature F1	0.012 [161]	1.000 [161]	0.043 [161]	0.000 [161]
Logical-atom F1	0.298 [161]	0.414 [161]	0.362 [161]	0.331 [161]
BEq+	0.000 [145]	0.000 [161]	0.000 [161]	0.000 [135]
GTED	–	0.321 [124]	0.125 [156]	0.233 [107]
Evaluation time (s)	0.0014 [161]	0.0013 [161]	0.0016 [161]	0.0007 [161]

0/163/0/24/0 for VERINA, 94/10/59/24/0 for CLEVER, 29/125/9/24/0 for the prompted baseline, and 61/52/44/22/8 for LeanAide. The coverage limitation persists after this exclusion. These checks support traceability and interpretation; they do not establish a budget-matched, held-out semantic leaderboard.

B.3 Complete metric comparison and benchmark dashboard

The complete comparison below reports every scalar signal used in the paper. Bracketed counts make conditional coverage explicit; in particular, LeanScorer was run on the VERINA paper inventory but not on CLEVER.

B.4 Authored calibration controls

Controlled changes to a relational specification. The authored requirement is: for any integer x , return any integer $y > x$. Required inputs are $-1, 0, 1$. For each input, the positive outputs are $x + 1, x + 2$, and the negative outputs are $x, x - 1$, giving six examples per output class and three required-input checks. Table 4 reports all eight saved controls. Figure 7 visualizes six distinct measured patterns; the equivalent rewrite duplicates the aligned scores, and the constant-true control has unavailable postcondition scores.

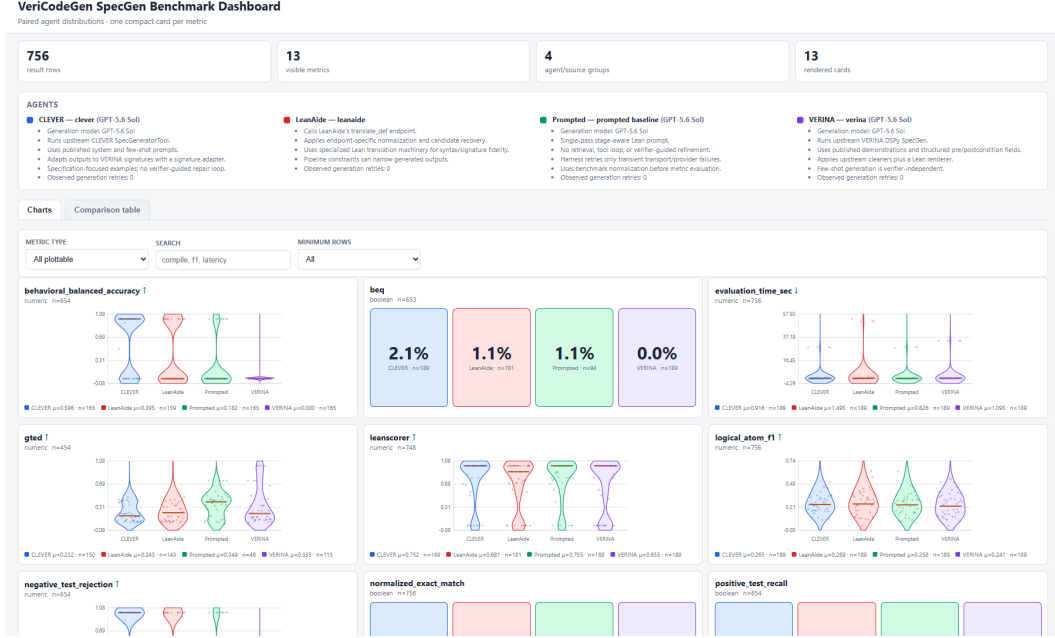


Figure 6: Static dashboard generated by the benchmark suite for the 756 VERINA task-agent records. The interface exposes metric availability, per-agent distributions, summary values, and a comparison table.

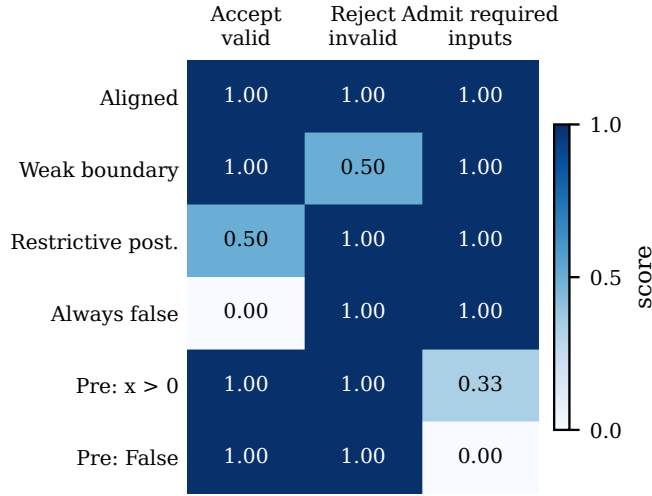


Figure 7: Selected authored controls for the requirement to return an integer $y > x$. Positive recall and negative rejection each use six labeled pairs; required-input recall uses the inputs $-1, 0, 1$. The last two rows preserve perfect postcondition scores while excluding required inputs. These controls illustrate distinct feedback dimensions, not the prevalence of model errors. This appendix gives all eight controls, including one with unavailable postcondition measurements.

The restrictive postcondition excludes $x + 2$, an allowed output under the relational target. An implementation returning $x + 1$ would nevertheless be a valid refinement of the requirement; the reduced recall measures lost allowed behaviors, not an incorrect implementation. Conversely, the false precondition excludes every required input while leaving the independently tested postcondition unchanged. The constant-true control did not produce postcondition measurements, so the retained run does not establish successful calibration of every authored control.

Table 4: Observed control scores. A dash denotes unavailable evidence. The constant-true row is retained to expose the incomplete calibration.

Precondition	Postcondition	R_+	R_-	R_{req}
True	$y > x$	1	1	1
True	$x < y$	1	1	1
True	$y \geq x$	1	1/2	1
True	$y = x + 1$	1/2	1	1
True	True	–	–	1
True	False	0	1	1
$x > 0$	$y > x$	1	1	1/3
False	$y > x$	1	1	0

Controlled separation of the behavioral dimensions. In order to study the three behavioral dimensions, we authored eight integer-return controls with known postconditions, preconditions, and required inputs. Figure 7 provides a directly measured example of what postcondition feedback omits. For the requirement to return an integer $y > x$, weakening the postcondition to $y \geq x$ yields positive-test recall 1 and negative-test rejection 1/2; restricting it to $y = x + 1$ reverses these scores. Keeping $y > x$ while changing the precondition to $x > 0$ or False preserves both postcondition scores at 1, but reduces required-input recall to 1/3 or 0. These examples support separate feedback for admitting valid outputs, rejecting invalid outputs, and retaining required inputs; they do not measure the frequency of those errors in generated outputs.