

JARE: A Differential-Testing Workbench for Auditing AI-Generated Specifications

Jerry Chen
McGill University

Elizabeth Kourbatski
McGill University

Allenna Tang
McGill University

Abstract

AI-assisted programming increasingly allows language models to generate not only code, but also the specifications and tests used to judge that code. This creates a subtle failure mode: a generated implementation may pass its generated tests while still satisfying the wrong specification. We present JARE, a differential-testing workbench for auditing AI-generated specifications. JARE represents candidate specifications as executable Python predicates, compares their behavior against a trusted reference predicate or human-confirmed labels, and reports concrete counterexamples when the candidate accepts inputs it should reject or rejects inputs it should accept. It classifies specifications as GOOD, UNDERCONSTRAINED, OVERCONSTRAINED, or MIXED. On a 35-candidate benchmark of security-adjacent specification tasks, JARE correctly classified all candidates, while a naive LLM baseline correctly classified 32 of 35. JARE does not prove correctness, but it turns vague specification review into concrete, human-checkable counterexample review.

1 Introduction

Specification-driven development separates what a program should do from how it does it. With the rise of "vibecoding," a single AI system can now generate the natural-language requirement, the specification, the implementation, and the tests in one pass. This creates a circularity problem. If the implementation and the tests are both derived from the generated spec, they agree with that spec by construction. A passing test suite then shows only that the code matches the spec — not that the spec matches what the human actually wanted.

The result is a failure mode that is easy to miss. The generated specification may have no syntax error, no type error, and no obvious bug. Instead, it has an intent error: it accepts inputs that should be rejected, or rejects inputs that should be accepted. A path sanitizer might block `../` but miss `..\` - it can look plausible, pass generated tests, and still permit security-relevant mistakes. A human reviewer rarely has time to enumerate every edge case, so "does this spec match the requirement?" is hard to answer in practice. As more developers use LLMs to draft specs, this kind of failure gets more common and harder to catch.

We present JARE, a differential-testing workbench for auditing AI-generated specifications. Rather than asking a developer to decide whether a spec is correct in the abstract, JARE produces a short list of concrete inputs where the candidate spec disagrees with the intended behavior. Confirming a handful of counterexamples is much faster than auditing a spec from scratch. JARE also reports what was tested, which requirements were exercised, and where the candidate looks too permissive or too strict.

JARE represents specifications as ordinary Python predicates. For benchmarked tasks, a trusted gold predicate defines the intended behavior. JARE generates authored and mutation-based probes,

runs both the candidate and the gold predicate on those probes, and treats disagreements as counterexamples. The direction of disagreement then determines whether the candidate is classified as GOOD, UNDERCONSTRAINED, OVERCONSTRAINED, or MIXED. In no-gold deployment mode, JARE compares the candidate against human-confirmed labels on concrete probes instead.

A useful analogy is a version-control merge tool. A merge tool does not decide for you whether two branches should be combined; it surfaces the specific lines where they conflict and asks the developer to resolve them. JARE plays the same role for specifications. It does not pronounce a spec correct or incorrect on its own. It surfaces the concrete inputs where the candidate spec and the intended behavior disagree, and leaves the final judgment to the developer.

Our contributions are:

1. **A human-in-the-loop spec audit.** JARE shows the developer the specific test cases where a candidate spec disagrees with the intended behavior. The developer can then confirm or reject those cases, rather than blindly trusting a black-box. This keeps the human in control of the final decision while JARE reduces the search burden.
2. **Directional diagnosis.** JARE classifies each spec as good, underconstrained, overconstrained, or mixed. This distinction matters because accepting forbidden inputs and rejecting allowed inputs imply different kinds of specification failures. It attributes each violation to the atomic requirement responsible, and flags requirements that cannot be behaviorally tested as un-auditable. The specs are executable Python predicates, so the audit does not require a proof assistant.
3. **A controlled benchmark with planted defects.** We evaluate JARE on 35 candidate specifications across security-adjacent tasks. Each task includes known-good, known-underconstrained, and known-overconstrained variants. This lets us measure both detection and localization against ground truth. We also compare JARE against a naive baseline where an LLM is simply asked whether the spec is correct.

2 Related Work

The techniques JARE builds on — differential testing, property-based testing, mutation testing — are individually well-established. The contribution is their assembly into an audit workflow targeting AI-generated specifications, producing human-verifiable counterexamples attributed to specific requirements. The novelty is in the problem framing and the output artifact, not the underlying mechanisms.

The most closely related system, which our work builds on, is VeriSpecGen, a traceable refinement framework for synthesizing intent-aligned Lean specifications from natural-language requirements [1]. VeriSpecGen decomposes natural-language descriptions into atomic requirements, generates requirement-targeted tests with traceability maps, validates generated specifications, and uses localized repair when validation fails. JARE shares the idea that requirement decomposition and traceability are useful for specification validation. However, JARE differs in its artifact and deployment target. VeriSpecGen operates in a Lean-based formal specification setting, while JARE audits executable Python predicates. This makes JARE less formally powerful, but easier to apply in lightweight AI-assisted programming workflows where the goal is to produce human-readable counterexamples quickly.

JARE also relates to recent work by Feng and Sergey on property-based testing for validating LLM-synthesized formal specifications [2]. Their work shows that property-based testing can expose

flaws in LLM-generated Lean specifications. JARE follows the same broad intuition, that property-based testing can be applied to specifications rather than code. However, JARE uses a different oracle and produces a different artifact. Their validation pipeline operates inside a Lean-based proof setting: candidate properties must be decidable for Lean’s Plausible checker to refute them, and the pipeline depends on an SMT solver and proof assistant infrastructure. JARE instead compares an executable Python predicate against human-confirmed labels, which removes the decidability requirement and eliminates the need for any verification toolchain. The downstream artifacts also differ in intent: Feng and Sergey use the counterexample to drive automated code generation. JARE uses counterexamples as the primary output — a human-readable audit report that attributes each disagreement to a specific requirement clause, intended to be read and confirmed by a developer rather than consumed by a downstream tool.

More broadly, JARE builds on the property-based testing tradition introduced by QuickCheck [3]: generate inputs, check whether a property holds, report counterexamples on failure. JARE applies this to candidate specifications rather than program implementations. The detection mechanism is differential testing, where the candidate predicate is run alongside a human-confirmed labels, and disagreements are counterexamples. The benchmark uses planted defects in the spirit of mutation testing: known flaws are introduced into otherwise-plausible specs so that detection rate can be measured against ground truth. This differs from true mutation testing, where mutations are generated systematically; here the defects are hand-authored, which limits benchmark scale.

The gap JARE addresses is practical specification auditing in AI-assisted development. Existing formal methods tools can provide stronger guarantees, but often require proof-assistant expertise and formal-language infrastructure. Direct LLM review is easier to use, but can produce confident judgments without executing the specification. JARE occupies a middle ground: it does not prove global correctness, but it provides concrete behavioral evidence that a human reviewer can inspect.

3 Methods

3.1 Problem Formulation

JARE audits a candidate specification against an intended behavior. The input is a natural-language requirement and a candidate specification represented as an executable Python predicate:

$$\text{spec}(*\text{inputs}) \rightarrow \{\text{True}, \text{False}\}.$$

The task is to classify the candidate specification into one of four categories:

- **GOOD**: no disagreement is found on the tested probes.
- **UNDERCONSTRAINED**: the candidate accepts at least one input that the intended behavior rejects.
- **OVERCONSTRAINED**: the candidate rejects at least one input that the intended behavior accepts.
- **MIXED**: the candidate makes both kinds of error.

We frame this as a behavioral alignment problem: a candidate specification may be syntactically valid and still fail to capture the intended requirement. JARE is therefore not checking syntax, types, or implementation correctness — it is checking whether the candidate predicate behaves like the intended predicate on targeted inputs.

3.2 Specification Representation

JARE represents specifications as Python predicates rather than specifications in a proof assistant such as Lean, Dafny, or Verus. This is a deliberate design choice. Python predicates are easy to execute and inspect, and they fit naturally into AI-assisted programming workflows. They also make counterexamples immediately readable: an audit can show the exact input value, the candidate’s decision, and the expected decision.

This choice has limitations. Python predicates do not provide formal guarantees, and the reference predicate may itself contain mistakes. In our benchmark, we reduce this risk by keeping trusted reference predicates simple and manually inspectable. This makes the system lightweight and accessible, but it does not provide a proof of correctness — JARE cannot guarantee that a candidate passing all probes is behaviorally equivalent to the intended predicate on all inputs.

3.3 Atomic Requirements

For each benchmark task, we manually decompose the natural-language requirement into atomic requirements:

$$R_1, R_2, \dots, R_n.$$

For example, a password requirement might contain one clause for minimum length, another for requiring a digit, another for requiring an uppercase character, and another for forbidding spaces.

The atomic requirement descriptions are used for human readability and localization. JARE does not interpret the natural-language strings directly. Instead, each requirement is paired with executable behavior through the trusted reference predicate and with tagged probes that are designed to exercise that requirement. This allows the audit report to say not only that the candidate specification disagrees with the intended behavior, but also which requirement appears implicated.

In this prototype, decomposition is hand-authored. This gives us reliable ground truth for evaluation. In deployment, decomposition could be proposed by an LLM and reviewed by a human, but we do not evaluate that workflow at scale in this work.

3.4 Probe Construction

JARE uses two kinds of probes.

Authored probes. Authored probes are manually written examples designed to isolate individual requirements. Whenever possible, a negative probe violates exactly one atomic requirement while satisfying all others. This makes localization easier. For example, if a password satisfies every clause except the no-spaces requirement, and the candidate accepts it, the disagreement can be attributed to that clause.

Mutation-generated probes. JARE also uses mutation-generated probes. Starting from valid seed inputs, mutation operators produce variants designed to violate particular requirements. In our benchmark, the candidate predicates are simple enough that the authored probes carry most of the detection weight. Mutation-generated probes would add value against more complex candidates where positional or structural variation in inputs matters — for example, a regex-based spec that handles a forbidden character correctly in one position but not another. The current prototype does not implement mutation-generated probes in the deployment mode, where no trusted reference predicate is available; extending them to that setting is a natural next step.

3.5 Differential Testing and Audit Generation

For each probe, JARE evaluates the candidate predicate and the gold predicate. A disagreement is classified by direction:

- If $\text{spec}(x) = \text{True}$ and $\text{gold}(x) = \text{False}$, then the candidate is too permissive on x . This is evidence of underconstraint.
- If $\text{spec}(x) = \text{False}$ and $\text{gold}(x) = \text{True}$, then the candidate is too strict on x . This is evidence of overconstraint.

JARE then aggregates disagreements into a final verdict. For readability, the audit report includes representative counterexamples rather than every failing probe. The report also includes requirement-level localization when the failing probes are tagged to specific atomic requirements. If a disagreement cannot be confidently assigned to one requirement, JARE records it in an untagged or unauditable bucket.

3.6 Benchmark

We evaluate JARE on a 35-candidate benchmark of security-adjacent specification tasks. The task families include password validation, username validation, email allowlists, URL allowlists, path sanitization, role access, token expiry, and rate limiting. These tasks were chosen because silent specification errors in these domains can plausibly create security or reliability failures.

The benchmark contains 9 good candidate specifications, 10 underconstrained candidate specifications, and 16 overconstrained candidate specifications. This includes 10 adversarial candidates designed to expose common failure modes such as off-by-one boundary errors, sneaky extra constraints, and missing normalization behavior. All defects are planted, so the expected classification is known before running JARE.

We report three evaluation outcomes:

1. **Classification accuracy:** whether the candidate is correctly labeled as GOOD, UNDERCONSTRAINED, OVERCONSTRAINED, or MIXED.
2. **Localization accuracy:** whether the audit identifies the expected violated requirement set.
3. **False-flag rate:** whether correct specifications are incorrectly reported as defective.

3.7 Baseline

We compare JARE against a naive LLM review baseline. The baseline receives the natural-language requirement and the candidate Python predicate, which is the same high-level information a human reviewer would see. It is then asked whether the specification is correct and, if not, whether it is underconstrained or overconstrained.

The baseline does not receive JARE’s probe set, execution results, counterexamples, requirement decomposition, or gold predicate. This comparison is intentionally simple. It tests whether execution-grounded auditing provides useful evidence beyond asking a model to inspect the specification directly.

4 Results

4.1 Benchmark Classification Results

JARE correctly classified all 35 candidate specifications in our benchmark: 9 good specifications, 10 underconstrained specifications, and 16 overconstrained specifications. The naive LLM baseline correctly classified 32 of 35.

Table 1: Classification accuracy on the 35-candidate benchmark.

Category	n	JARE Classification	Baseline Classification
Good	9	9/9 (100%)	7/9 (78%)
Underconstrained	10	10/10 (100%)	10/10 (100%)
Overconstrained	16	16/16 (100%)	15/16 (94%)
Total	35	35/35 (100%)	32/35 (91%)

The most important difference is on correct specifications. The baseline falsely flagged 2 of 9 good specifications as defective, while JARE produced no false positives on good specifications. This matters because false positives make an audit tool harder to trust: developers may stop paying attention if correct specifications are repeatedly flagged as wrong.

JARE also correctly classified all 10 adversarial candidates added to the benchmark. These candidates included off-by-one boundary errors, sneaky extra constraints, and missing normalization behavior. This does not prove general adversarial robustness, but it shows that execution-grounded auditing catches plausible specification defects that direct model judgment can miss.

4.2 Localization Diagnostics

We do not include localization in the headline comparison table because the baseline does not produce requirement-level attribution. However, JARE still records localization internally.

JARE matched the expected violated requirement set in 33 of 35 cases. The two misses were not classification failures. In one URL-allowlist case, JARE found a real additional violation that our expected label omitted. In one role-access case, JARE found the overconstraint but attributed it to the untagged bucket because the exposing probe was a valid input rather than a single-fault negative probe.

These cases show that localization is useful as a diagnostic, but it depends on probe tagging and benchmark-label quality. For that reason, we treat classification accuracy as the main result.

4.3 Deployment Examples with ChatGPT-Generated Specifications

To test JARE’s no-gold deployment workflow, we asked ChatGPT 5.4 to generate security-adjacent requirements together with a candidate Python predicate containing exactly one planted defect (underconstrained or overconstrained). The full prompt is given in Appendix A.

API token validation. The most informative output concerned API token validation. The requirement stated that an API token must be exactly 32 characters long and consist only of hexadecimal characters from 0-9 and a-f. The candidate specification was:

```
lambda t: len(t) == 32 and all(c in "0123456789abcdef" for c in t)
```

ChatGPT 5.4 labeled this specification as underconstrained, claiming that it would accept uppercase hexadecimal characters. This label was incorrect. The predicate checks membership in the lowercase-only string "0123456789abcdef", so uppercase letters are rejected. JARE’s no-gold workflow caught this by testing the predicate against human-confirmed probes.

The audit proposed six probes: a valid lowercase token, a 31-character token, a 33-character token, an uppercase token, a token containing g, and an all-zero token. The human confirmed the intended labels, including REJECT for the uppercase token. The candidate agreed with all six confirmed labels, so JARE returned a GOOD verdict with a 100% agreement rate and no counterexamples.

This example shows the value of execution-grounded auditing in deployment. The model that generated the specification also gave an incorrect explanation of its defect. JARE did not rely on that explanation. Instead, it executed the predicate on concrete examples and compared the result against human-confirmed labels.

5 Discussion

JARE’s results suggest that execution-grounded auditing is a useful complement to direct LLM review. The naive baseline had access to the requirement and specification, but it still made three classification errors. JARE avoided these errors because it did not rely on a model’s abstract judgment of the predicate. It executed the predicate on targeted examples and compared the behavior against an explicit oracle.

The most practically important result is not only that JARE classified defective specifications correctly, but that it avoided falsely flagging correct specifications in the benchmark. In a real development workflow, an audit tool that repeatedly flags correct specifications may become noise. By grounding its verdicts in concrete disagreements, JARE can give developers more actionable evidence.

The deployment examples also highlight a broader lesson for AI-assisted software development. A model can generate a plausible specification and a plausible explanation of its defect, while still being wrong about the behavior of its own code. JARE is designed to interrupt this pattern. It does not ask the model to explain whether the specification is correct; it asks the predicate to run on concrete inputs.

For safety and security workflows, this distinction matters. Many serious failures arise not from code that crashes, but from validation logic that is slightly too permissive or too strict. A path sanitizer that misses one traversal pattern, an allowlist that mishandles normalization, or an access-control rule that rejects or accepts the wrong role can all create downstream risk. JARE is not a complete defense against such failures, but it offers a lightweight audit layer that makes these misalignments visible.

6 Limitations

JARE has several important limitations.

First, the benchmark is small and author-designed. Although it includes 35 candidates across security-adjacent tasks, it is not representative of all specification failures that appear in real systems. The planted defects were designed to be detectable, so perfect classification on this benchmark should not be interpreted as evidence of general robustness.

Second, JARE relies on finite testing. A clean audit does not prove that a candidate specification is correct. It only means that no disagreement was found among the tested probes. A candidate

may still contain defects that the probe set does not exercise.

Third, gold-mode evaluation depends on the correctness of the gold predicate. JARE treats the gold predicate as the oracle. If the gold predicate is wrong, incomplete, or itself misaligned with the human’s intent, JARE may produce a confident but misleading audit. This is a fundamental limitation of reference-based differential testing.

Fourth, no-gold deployment has not been evaluated at scale. In that mode, JARE depends on humans to confirm labels for LLM-proposed probes. We have not yet measured how accurately non-expert developers perform this labeling task, or how often they miss subtle edge cases.

Fifth, JARE does not attempt automated repair. When the audit surfaces a counterexample, the developer is responsible for deciding how to fix the specification. A counterexample-guided repair loop is a natural extension, but is outside the scope of this prototype.

Finally, JARE does not address requirement elicitation. If the natural-language requirement is incomplete, ambiguous, or wrong, JARE cannot recover the true intent on its own. It audits a specification against an expressed requirement, not against an unstated goal.

7 Future Work

The most pressing next step is a larger and more diverse benchmark. Our 35 candidates were authored by the same people who built JARE, which limits how much the results generalize. A stronger evaluation would draw specifications from multiple domains, include more complex input structures, and use adversarial cases written by people outside the team. It would also test JARE against specs produced by different language models under different prompting strategies, since failure modes likely vary across both.

The most important deployment question is whether no-gold auditing actually works in practice. Our API-token example shows the workflow is feasible, but we have not measured how well developers confirm labels on probes they did not author themselves. A user study would clarify whether non-expert developers catch subtle edge cases when labeling, whether they trust the resulting verdicts, and whether JARE meaningfully changes how they review specs compared to reading the predicate directly. Without this, the deployment-mode results remain anecdotal.

8 Conclusion

AI-generated specifications can fail silently. They may be syntactically valid and pass their generated tests while still accepting forbidden inputs or rejecting allowed ones. This is especially concerning when specifications define security-sensitive validation logic.

JARE addresses this problem by turning specification review into an execution-grounded audit. It represents candidate specifications as Python predicates, tests them on targeted probes, compares their behavior against a gold predicate or human-confirmed labels, and reports concrete counterexamples. On our 35-candidate benchmark, JARE correctly classified all candidates and outperformed a naive LLM baseline.

JARE does not prove correctness. Its audits are only as strong as the probes and oracle behind them. However, it provides a practical and human-readable layer between blind trust in generated specifications and heavyweight formal verification. For AI-assisted development, this middle ground is valuable: it makes specification misalignment concrete, reviewable, and easier to act on.

Code and Demo

Code repository: <https://github.com/Lil-Chen05/jare> Demo video: <https://youtu.be/IYGFNyGrCic>

LLM Usage Statement

We used LLM assistance throughout the project. Claude Code helped with implementation, including scaffolding the audit pipeline, writing probe runners, and debugging. We also used LLMs to brainstorm project framing, draft early report sections, and generate candidate specification examples for the deployment evaluation. All reported results, benchmark labels, classifications, and claims were independently checked by the project team.

Dual-Use Considerations

A tool that identifies permissive edge cases in specifications could help defenders repair validation logic, but similar techniques could be used to search for weaknesses in deployed systems. JARE is designed for local developer auditing of specifications before deployment; using it to probe systems without authorization is outside its intended scope.

References

- [1] 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. <https://arxiv.org/abs/2604.10392>
- [2] Yueyang Feng and Ilya Sergey. On the Unreasonable Effectiveness of Property-Based Testing for Validating Formal Specifications. *Proofs and Intuitions*, 2026. <https://proofsandintuitions.net/2026/05/18/property-based-testing-specifications/>
- [3] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

A Deployment Prompt

The following prompt was used to elicit candidate specifications for the no-gold deployment evaluation in Section 4.3:

```
I need you to give me exactly two things: a natural-language software requirement and a candidate Python predicate spec as a single lambda expression. The requirement should be security-adjacent and easy to understand. The candidate spec should look plausible but contain exactly one subtle defect: either UNDERCONSTRAINED or OVERCONSTRAINED. Use only simple Python built-ins. The lambda must be directly usable with: jare audit-spec --requirement '<REQUIREMENT>' --spec '<PYTHON_LAMBDA>'.
```