Prove It, Don't Read It: Contract-First Verification of AI-Generated Code with Sealed Contexts and Evidence-Graded Verdicts

holdtrue-dev

ORCID 0009-0003-3552-8155 · holdtrue-dev@proton.me

July 3, 2026

Abstract. Large language models now write code faster than people can meaningfully read it, yet the dominant acceptance interface for AI-generated code is still line-by-line human review. We argue that this interface neither scales nor reliably catches plausible-but-wrong programs, and that the artifact a human signs off on should change: from the implementation to a short, machine-checkable contract. We present `holdtrue`, an open-source verification harness built on this principle. Two AI contexts that never share memory split the work: an *author* turns a plain-language intent into a formal contract, which the human reads and approves; behind an information-flow *curtain*, an *implementer* writes code from the contract alone, never seeing the intent, the private reference oracle, or the held-out tests. A layered harness—strict static typing, symbolic-execution proof, property-based testing, held-out differential testing, a negative-probe that rejects vacuous contracts, and mutation analysis—runs in an unprivileged sandbox and returns one of four evidence-graded verdicts: `GUARANTEED`, `ENFORCED`, `UNGUARANTEED`, or `FAILED`. We describe the protocol and its threat model, the verdict semantics, a worked example, and a reference implementation with six language backends, and we argue that graded, evidence-backed verdicts are the honest interface for delegated programming.

Keywords: AI code generation; design by contract; formal verification; property-based testing; mutation testing; symbolic execution; human oversight.

1 Introduction

Large language models can produce working programs from natural-language descriptions (Chen et al., 2021; Austin et al., 2021), and in many workflows they now produce most of the code. The acceptance interface, however, has not moved: a human reads the generated diff and decides whether to trust it. Reviewer attention is the scarcest resource in this loop, and it is spent on the artifact least suited to human inspection—a wall of plausible code.

Reading is not only slow; for this class of code it is unreliable. Generated code is, by construction, optimized to look right. Benchmark audits show that the test suites used to declare LLM code "correct" substantially overstate correctness once strengthened (Liu et al., 2023). Security studies find that a large fraction of assistant-generated completions in sensitive scenarios are vulnerable (Pearce et al., 2022), and that developers with an AI assistant write *less* secure code while believing they wrote *more* secure code (Perry et al., 2023). Plausibility bias is precisely the failure mode human review is worst at catching.

Our position is that the object under review should change. A reviewer should approve a *contract*—a short, declarative, machine-checkable statement of what the code must do—and delegate to machines the question of whether a given implementation satisfies it. The reviewer's cost then scales with the size of the specification rather than the size of the implementation, and the answer to "does the code meet the spec?" comes back as checkable evidence rather than as an impression formed while reading. In short: review the guarantee, not the code.

This paper describes `holdtrue`, an open-source harness that operationalizes this position. Its contributions are: **(i)** a protocol that splits code generation across two sealed AI contexts—a contract author and an implementer—with the human approval gate placed at the contract, and an information-flow *curtain* that keeps the intent, the reference oracle, and the held-out tests away from the implementer (§3); **(ii)** a layered verification harness that combines strict typing, symbolic-execution proof, property-based testing, held-out differential testing, a non-vacuity probe, and mutation analysis, executed in an unprivileged sandbox (§3.5); **(iii)** a four-tier, evidence-graded verdict semantics that refuses to overstate what was established (§3.6); and **(iv)** a reference implementation with six language backends and a reproducible example suite (§4).

2 Related Work

Specifications as the unit of trust. The idea that programs should be judged against explicit assertions goes back to axiomatic semantics (Hoare, 1969) and was made an engineering discipline by design by contract (Meyer, 1992). `holdtrue`'s contracts are executable pre- and postconditions in this tradition, written with the `deal` library for Python (life4 contributors, 2024). Dijkstra's (1972) observation that testing shows the presence of bugs, never their absence, motivates the harness's separation between a proof tier and sampled tiers.

Verification machinery. Each layer of the harness is an established technique. Symbolic execution (King, 1976; Cadar & Sen, 2013) underlies the proof tier, implemented with CrossHair, an SMT-backed checker for Python contracts (Schanely, 2024). Property-based testing originates with QuickCheck (Claessen & Hughes, 2000) and reaches Python as Hypothesis (MacIver et al., 2019). Mutation analysis measures whether a test suite would notice injected defects (DeMillo et al., 1978; Jia & Harman, 2011). Differential testing against an independent implementation (McKeeman, 1998) is one classical answer to the oracle problem (Barr et al., 2015). `holdtrue` contributes no new checker; its contribution is the protocol that arranges these checkers so their combined outcome is trustworthy to a non-reader.

Assessing and constraining LLM code. HumanEval and MBPP established functional-correctness benchmarks (Chen et al., 2021; Austin et al., 2021); EvalPlus showed that weak test suites inflate them (Liu et al., 2023). Closer to our setting, recent work formalizes user intent before or alongside generation: nl2postcond studies LLM translation of natural language into formal postconditions (Endres et al., 2024), TiCoder clarifies intent through user-ratified tests (Fakhoury et al., 2024), Clover checks generated code, docstrings, and annotations for mutual consistency (Sun et al., 2024), and Misu et al. (2024) synthesize verified Dafny methods. `holdtrue` differs in two ways: the specification is the *only* artifact the human approves, and generation is adversarially partitioned—the implementer cannot see the evaluation materials, a discipline analogous to train/test hygiene and a direct mitigation of specification gaming and Goodhart-style optimization against the visible check (Amodei et al., 2016; Manheim & Garrabrant, 2018).

3 The `holdtrue` Protocol

3.1 The trust object

`holdtrue` relocates human judgment to a single artifact: the contract. A usable trust object must be *short* (readable in one sitting), *declarative* (states what must hold, not how), *total over its stated domain* (explicit preconditions bound the promise), and *machine-checkable* (every clause can be executed, sampled, or proven). Code satisfies none of these for a non-author; a contract can satisfy all four. Approving one is the entire residual review burden the protocol asks of the human.

3.2 Threat model

We assume both AI contexts are fallible and possibly optimizing against the checks they can see. The failure modes, and the mechanism each is mapped to, are: **(a)** the implementer is simply wrong → layered checks (§3.5); **(b)** the implementer overfits to visible tests → held-out differential tests and a private reference oracle that never cross the curtain; **(c)** the author writes a vacuous or weak contract → a negative-probe that requires the contract to reject known-broken implementations, plus human approval; **(d)** the test suite itself is inadequate → mutation analysis; **(e)** generated code misbehaves at check time → an unprivileged sandbox with no network and a read-only system; **(f)** the contract diverges from the human's intent → the approval gate itself, which is the one failure mode no machine in the loop can close (§5).

3.3 Protocol walkthrough

Figure 1 shows the flow. **(1)** The human states an intent in plain language. **(2)** The *author* context drafts a machine-checkable contract from it, together with visible property tests, a private reference oracle, and held-out differential tests. **(3)** The human reads and approves the contract—this, not the code, is the sign-off. **(4)** The approved contract crosses the curtain; nothing else does. The curtain is enforced by the filesystem rather than by instruction: the implementer's workspace contains the contract and nothing else. **(5)** The *implementer* context writes code from the contract alone. **(6)** The harness checks the code against the contract in a sandbox and returns a verdict with its evidence.

Both contexts run on whatever assistant the user selects—coding-agent CLIs (claude, aider, gemini, codex), chat APIs (Anthropic, OpenAI, Ollama), or an arbitrary command via `HOLDTRUE_AGENT_CMD`. The protocol is deliberately indifferent to this choice: the verdict derives from the checks, not from trust in any model.

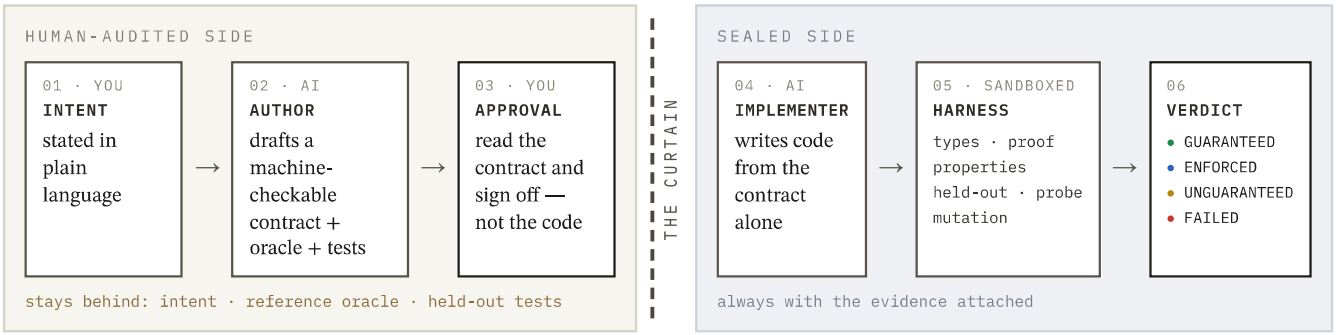

Figure 1. The `holdtrue` protocol. The author and the implementer share no memory; the curtain is enforced by the filesystem—the implementer's workspace contains the approved contract and nothing else. Only the contract crosses; the intent, the private reference oracle, and the held-out differential tests never do.

3.4 Contracts

A contract pins one or more functions with four kinds of clause: a *precondition* bounding the domain of the promise, *postconditions* relating inputs to the result, an *exception clause* stating what may be raised, and a *typed signature* the implementer must match. Listing 1 shows the complete contract for the running example, generated from the one-sentence intent *"Clamp a number into a range: given x, lo, hi, return x if it sits inside [lo, hi], otherwise the nearest bound."*

```
@deal.pre(lambda x, lo, hi: lo <= hi)
@deal.ensure(lambda x, lo, hi, result:
             lo <= result <= hi)
@deal.ensure(lambda x, lo, hi, result:
             result == min(max(x, lo), hi))
@deal.raises()
def clamp(x: int, lo: int, hi: int) -> int: ...
```

Listing 1. The approved contract for `clamp`: precondition (valid range), a bounding and an exact-value postcondition, a no-exceptions clause, and the typed signature.

Note what the second postcondition adds: without it, any in-range constant would satisfy the contract. Contract strength is not an aesthetic preference; it is what the verdict tiers measure (§3.6). Alongside the contract, the author emits Hypothesis property tests that the implementer *may* see, and two artifacts that it may not: a private reference implementation acting as an oracle, and held-out differential tests that compare candidate outputs against that oracle on generated inputs (McKeeman, 1998).

3.5 The verification harness

Verification is a fixed cascade of independent gates. **L1 Types:** the implementation must pass `mypy --strict` before anything else runs. **L2 Proof:** CrossHair symbolically executes the function against every contract clause; if it exhausts the input space, the contract is proven for *all* inputs, not a sample. **L3 Properties:** Hypothesis runs the visible property tests. **L4 Held-out:** the differential tests compare the implementation against the private oracle on inputs the implementer never saw. **L5 Negative-probe:** the harness runs the contract against known-broken stand-ins (buggy controls); a contract that fails to reject them is vacuous, and no green result built on it is allowed to mean anything. **L6 Mutation:** cosmic-ray injects defects into the accepted implementation to confirm the test suite would notice, guarding against false comfort from an inadequate suite (Jia & Harman, 2011).

Every gate executes inside a bubblewrap sandbox: no network, a read-only system, writes confined to a scratch directory. The harness fails closed—without the sandbox it refuses to run AI-written code unless explicitly overridden. Each run emits a Markdown and a JSON evidence report recording the verdict, the deciding check, and what was and was not tested.

3.6 Verdict semantics and revision

The harness never answers without evidence, and never upgrades evidence rhetorically. Table 1 gives the four tiers. `GUARANTEED` requires both a completed proof (L2) and a non-vacuous contract (L5, L6): proven, and the proof is *about something*. `ENFORCED` is the honest tier for shapes a prover cannot exhaust—strings, lists, floats, loops, rich validated types—where the contract is checked at runtime on every call (via deal, with pydantic validating rich-type bounds) and every sampled and held-out input is clean; a violating input raises rather than slipping through. `UNGUARANTEED` means only sampled evidence exists—typically because the contract is too weak to support more—and human review is still required. `FAILED` returns a concrete counterexample. A contract may pin several functions; each gets its own verdict, and the overall result is only as strong as its weakest part.

Verdict	Evidence standard
• GUARANTEED	Proven over all inputs by symbolic execution; contract shown non-vacuous (catches injected bugs, rejects broken stand-ins).
• ENFORCED	Contract checked at runtime on every call; clean over every sampled and held-out input; not proven for all inputs.
• UNGUARANTEED	Sampled evidence only, usually because the contract is too weak to earn more; still needs human review.
• FAILED	A concrete counterexample, reported with the exact input that breaks the contract.

Table 1. Evidence-graded verdicts. Tiers are defined by the strength of the evidence, never by confidence in the model that wrote the code.

Contracts themselves may need revision—for instance when the self-check finds a clause unsatisfiable. Revision is *never silent*: the harness proposes a fix, refuses any change that would weaken a check, waits for human approval, and records every change in an append-only changelog. Monotonicity matters; a loop that could quietly weaken the specification until the code passes would reintroduce exactly the Goodhart dynamic the curtain exists to prevent (Manheim & Garrabrant, 2018).

4 Implementation and Worked Example

4.1 *Reference implementation*

`holdtrue` is implemented in Python 3.12 with a uv-pinned toolchain and is released under Apache-2.0. One command, `holdtrue make "<intent>"`, drives the full pipeline from a blank directory to a verdict; `verify`, `run`, `tui`, and `studio` expose the same loop for scripting and interactive use. A project directory persists the intent, the contract, the visible tests, the private oracle and held-out tests, the evidence reports, and the revision log. One artifact is deliberately *not* persisted: the generated implementation itself lives only in a temporary workspace during the run. What survives is the contract and the evidence—a direct expression of the thesis that these, not the code, are the objects worth keeping.

4.2 *The clamp example, end to end*

The repository ships controls that exercise every verdict from one contract (Listing 1). Verifying the correct control yields `GUARANTEED`: CrossHair exhausts the integer input space, and the probe and mutation gates confirm the contract and tests reject broken variants. Substituting the buggy control yields `FAILED` with the specific counterexample input. Most instructive is the third run: the *same correct code* checked against a deliberately weakened manifest returns `UNGUARANTEED`—the code is fine, but the contract is too weak to earn a guarantee. The verdict grades the promise, not the programmer.

4.3 *Example suite and language backends*

Fourteen shipped examples span the tiers: single proven functions (`clamp`, `abs`, `square`, `repeat`); multi-function proven contracts (`dnd`, `chess`, `clock`); rich-typed domains that cap at `ENFORCED` (`checkout`, `nights`, `pagination`, `scheduler`, `poker`, `semver`); and `billing`, whose report mixes proven money helpers with enforced document functions in a single contract.

Six language plugins share the harness through a three-method interface (`available`, `author_instructions`, `run_checks`); Table 2 lists each toolchain and its verdict ceiling. Python is the only backend where the proof tier is always available, because CrossHair ships as an ordinary package; Rust reaches it when the Kani model checker is installed; the rest cap at `ENFORCED`—and are reported as such, not dressed up.

Language	Properties	Mutation	Proof	Ceiling
Python	Hypothesis	cosmic-ray	CrossHair	• GUARANTEED
Rust	proptest	cargo-mutants	Kani (opt.)	• GUARANTEED*
TypeScript	fast-check	Stryker	—	• ENFORCED
Go	rapid	gremlins	—	• ENFORCED
Java	jqwik	PIT	—	• ENFORCED
C♯	FsCheck	Stryker.NET	—	• ENFORCED

Table 2. Language backends and verdict ceilings. Type gates (mypy --strict, tsc, cargo, go vet, javac, dotnet) precede all other checks. *Rust reaches the proof tier only with Kani installed; otherwise it caps at ENFORCED.

5 Discussion and Limitations

The trusted computing base. A verdict is only as sound as the checkers beneath it: the harness, deal, CrossHair, Hypothesis, the mutation engine, and the sandbox are all trusted. CrossHair is incomplete by nature; the design response is that incompleteness only ever *downgrades* a verdict (proof unavailable → `ENFORCED`), never upgrades one.

The intent–contract gap. No machine in the loop can certify that the approved contract means what the human meant. The approval gate is the irreducible human act, and reading a contract, while far cheaper than reading code, is a skill; a subtly wrong precondition silently narrows the promise. The negative-probe bounds vacuity but not misalignment.

Correlated authorship. The author context writes both the contract and the reference oracle, so a shared blind spot can survive: an oracle that encodes the same misunderstanding as the contract will not catch it differentially. Running the author and implementer on *different* model families is a cheap diversity measure the provider abstraction already permits; making oracle authorship independent is future work.

Scope of the proof tier. `GUARANTEED` is currently realistic for pure, integer-shaped functions; strings, floats, and loop-heavy code land at `ENFORCED`. The unit of verification is the function or small module with an explicit signature —not stateful services, I/O, or whole systems. Mutation and proof also cost wall-clock time, and the sandbox requires Linux (bubblewrap). We view these as the honest current boundary of the approach rather than incidental gaps: the protocol is designed so that widening any of them (a stronger prover, a new language plugin, an independent oracle author) raises ceilings without changing what a verdict means.

6 Conclusion

When machines write the code, the human bottleneck should not be reading it. `holdtrue` restructures delegation so that the human approves a short machine-checkable contract, sealed contexts prevent the implementer from optimizing against the evaluation, and a layered harness returns a verdict graded by the strength of its evidence—proof where possible, runtime enforcement where not, and an explicit refusal to call either one the other. We offer the protocol, the verdict semantics, and the open reference implementation as a template for verification-first delegated programming, and as a small argument that honesty about evidence is a design principle, not a disclaimer.

Availability

`holdtrue` is open source under Apache-2.0 at github.com/holdtrue-dev/holdtrue (project site: holdtrue-dev.github.io). Every verdict discussed in §4.2 is reproducible from the shipped examples with two commands (`uv sync`; `holdtrue verify examples/clamp --impl …`).

AI-Generation Statement

In keeping with JAIGP's open-prompting ethos: this article was drafted by Claude (Anthropic) under the direction of the human author, who designed and implemented `holdtrue` and reviewed and approved the text. Technical claims describe the public repository and project site as of July 2026. Suggested categories: Software Engineering; Formal Verification.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete problems in AI safety. *arXiv:1606.06565*.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. *arXiv:2108.07732*.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering, 41*(5), 507–525.

Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM, 56*(2), 82–90.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. *arXiv:2107.03374*.

Claessen, K., & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (pp. 268–279).

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer, 11*(4), 34–41.

Dijkstra, E. W. (1972). Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, & C. A. R. Hoare, *Structured Programming* (pp. 1–82). Academic Press.

Endres, M., Fakhoury, S., Chakraborty, S., & Lahiri, S. K. (2024). Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering, 1*(FSE), 1889–1912.

Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., & Lahiri, S. K. (2024). LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering, 50*(9), 2254–2268.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10), 576–580.

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 37*(5), 649–678.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385–394.

life4 contributors. (2024). *deal: Design by contract for Python* [Software]. https://github.com/life4/deal

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*.

MacIver, D. R., Hatfield-Dodds, Z., et al. (2019). Hypothesis: A new approach to property-based testing. *Journal of Open Source Software, 4*(43), 1891.

Manheim, D., & Garrabrant, S. (2018). Categorizing variants of Goodhart's law. *arXiv:1803.04585*.

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

Meyer, B. (1992). Applying “design by contract”. *IEEE Computer, 25*(10), 40–51.

Misu, M. R. H., Lopes, C. V., Ma, I., & Noble, J. (2024). Towards AI-assisted synthesis of verified Dafny methods. *Proceedings of the ACM on Software Engineering, 1*(FSE), 812–835.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions. In *Proc. IEEE Symposium on Security and Privacy* (pp. 754–768).

Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 2785–2799).

Schanely, P. (2024). *CrossHair: An analysis tool for Python that blurs the line between testing and type systems* [Software]. https://github.com/pschanely/CrossHair

Sun, C., Sheng, Y., Padon, O., & Barrett, C. (2024). Clover: Closed-loop verifiable code generation. *arXiv:2310.17807*.