Ethereum Verification Benchmark: Evaluating AI Agents on Lean 4 Proofs for Smart Contracts

Thomas Marchand
Independent Researcher
Lisbon, Portugal
research@thomas.md

***Abstract*—Formal verification helps find serious errors in smart contracts, but writing machine-checked proofs still takes substantial expert work. AI agents can now use compiler feedback to revise proof scripts. This makes it useful to test whether agents can produce complete proofs, not just informal arguments or small code fragments. Existing benchmarks rarely test this ability for smart contract obligations. Ethereum Verification Benchmark is an open evaluation suite for measuring whether AI agents can generate Lean 4 proofs of Ethereum smart contract correctness.**

The benchmark uses Verity's shallow embedding of contract semantics in Lean 4. It contains 135 proof tasks from 25 deployed or ecosystem-relevant Ethereum cases. Each task has a fixed public proof interface and a fixed verifier pipeline. Agents do not see the hidden reference proofs. The harness rejects `sorry`, `admit`, new axioms, theorem-statement edits, and unavailable imports. Version manifests record fingerprints for each task and execution context, so scores can be reproduced. We describe the benchmark design and the v0.1 release results. In complete runs, GPT-5.5 verifies 66 of 135 tasks, GLM 5.2 verifies 53, and Opus 4.8 verifies 45. Other complete runs verify as few as 31 tasks. Even the best complete run solves less than half of the suite. A budget-scaling sidecar suggests that some failures depend on search effort as well as model choice.

***Index Terms*—smart contracts, formal verification, Lean 4, theorem proving, benchmarks, AI agents, Ethereum**

I. Introduction

Formal verification is now used in smart contract assurance. Tools such as the Certora Prover and K-based Ethereum analyses such as KEVM are used on production protocols. Foundry has also made property tests common in Ethereum engineering practice [1]–[4]. The hard part is often not running the prover. It is turning protocol intent into exact invariants and then writing the proof. For large decentralized finance protocols, this work can take weeks or months and requires knowledge of protocol design, EVM semantics, and theorem proving over fixed-width arithmetic.

AI agents make this problem easier to measure. Modern agents can revise proof scripts inside a repository by using compiler feedback. Benchmarks in mathematics and programming, including miniF2F, LeanDojo, HumanEval, and SWE-bench, have made progress easier to compare [5]–[8]. Those benchmarks do not test the specific ability needed by Ethereum verification teams: generating complete Lean 4 proofs for smart contract state-transition properties under a machine-checked verifier.

Ethereum Verification Benchmark targets this setting. It evaluates agents on proof-only tasks from Ethereum protocols and ecosystem contracts. Each task gives the agent a fixed contract model, a fixed formal specification, and one editable Lean 4 proof file. The agent succeeds only when an independent verifier accepts the submitted proof. The verifier checks that the theorem statement has not changed, rejects proof placeholders, enforces a trusted axiom ledger, and rebuilds the proof in an isolated workspace. Reference proofs are not present in that workspace.

The benchmark is not a replacement for contract verification tools. Certora, KEVM, K, Foundry, and related tools help engineers state and check contract properties. Ethereum Verification Benchmark instead evaluates the proof-producing agent. It asks whether an agent can complete a formal proof after the contract semantics and property have already been fixed. This differs from general software engineering and theorem-proving benchmarks because the tasks combine uint256 arithmetic with protocol invariants from Ethereum systems.

This paper contributes a benchmark protocol for AI-generated smart contract proofs. The protocol isolates tasks, checks theorem integrity, rejects proof placeholders, and accounts for axioms in CI. It also describes the v0.1 suite: 135 tasks across 25 cases and 25 protocol families. The suite mainly covers accounting, solvency, storage effects, and linked-list ownership structures. We report the v0.1 result artifacts. The best complete run solves 49% of tasks. Per-task outcomes show stronger results on local arithmetic and storage obligations than on longer invariant chains and cross-state reasoning.

II. Related Work

Table I summarizes the closest lines of work. Smart contract verification tools check properties of programs. Coding benchmarks check whether agents can write or edit code. Theorem-proving benchmarks check proof search, mostly on mathematical tasks. Ethereum Verification Benchmark sits at their intersection: the input is a smart contract proof obligation, and the output must be a Lean 4 proof accepted by an independent verifier.

LLM work on smart contracts has mostly studied code generation, vulnerability detection, or repair [11]. That is useful, but it is not the same target. This paper measures

TABLE I
COMPARISON WITH NEARBY VERIFICATION AND AGENT-EVALUATION WORK.

Work	Target	Input	Output	Agent eval.	Contracts
Certora Prover [1]	Contract properties	Contract plus specifications	Verification result	No	Yes
KEVM and K [2], [3]	EVM semantics and analyses	Program plus formal semantics	Semantics-based proof or analysis	No	Yes
Ethereum model checking [9]	Solidity control flow	Contract model	Model-checking result	No	Yes
Move model checking [10]	Move contracts	Move model	Model-checking result	No	Yes
HumanEval and SWE-bench [7], [8]	Code generation and repair	Prompt or repo issue	Passing code change	Yes	No
miniF2F and LeanDojo [5], [6]	Theorem proving	Formal theorem	Machine-checked proof	Yes	No
Ethereum Verification Benchmark	Ethereum proof tasks	Fixed contract model and theorem	Accepted Lean 4 proof	Yes	Yes

whether an agent can produce the proof artifact itself after the property and formal model are fixed.

III. Background: Verity and Lean Proof Tasks

Verity is a Lean 4 framework for writing specifications and proofs for smart contracts. It uses a shallow embedding of contract semantics. In a shallow embedding, contract operations are represented using host-language definitions and propositions rather than as uninterpreted syntax alone. This gives proof authors direct access to Lean 4's theorem prover while preserving a semantic model of contract storage and state transitions.

A benchmark task fixes the implementation side and the theorem to prove. The contract model defines the operation being verified. The specification states the intended postcondition or invariant. The target theorem connects the implementation to the specification. The editable proof file initially contains the theorem statement and an empty proof body; hidden reference proofs exist only for benchmark validation and are not present in the agent workspace.

Listing 1. Schematic benchmark task shape.

```
theorem deposit_sets_total_assets
  (s : VaultState) (assets : UInt256)
  (h : assets > 0) :
  spec_total_assets (deposit s assets h) =
    s.totalAssets + assets := by
  -- agent must complete this proof
```

The benchmark does not claim that every Ethereum behavior is fully modeled. Instead, each case declares the formal surface it exercises, and the suite makes coverage limitations explicit.

IV. Benchmark Design

A. Task Contract

Every task is a proof-only problem. Agents may edit only designated proof files. They may inspect the public task files. They also receive helper libraries and a generated summary. They may not see the hidden reference proof, modify the theorem statement, add axioms, or alter the verifier. This keeps scores focused on proof synthesis rather than benchmark adaptation.

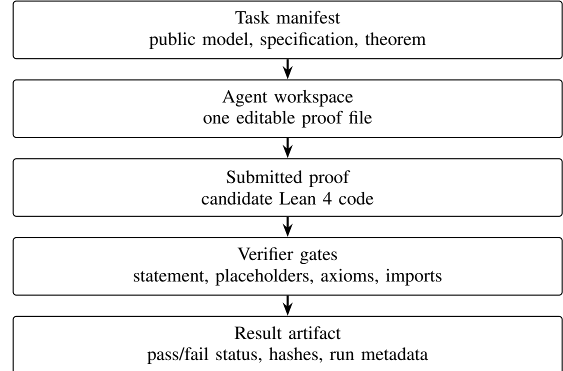

Fig. 1. Benchmark workflow. Agents see only the public task files. Hidden reference proofs are used for suite validation, not during agent runs.

B. Axiom Ledger and Placeholder Rules

The harness rejects `sorry`, `admit`, and new `axiom` declarations in submitted proof files. Existing trusted assumptions are tracked through a repository-level axiom ledger. CI enforces that the ledger remains explicit and that proof files cannot silently expand the trusted computing base. This is essential for a benchmark where otherwise an agent could "solve" a task by assuming the target theorem.

C. Theorem-Statement Integrity

A proof attempt is not successful merely because Lean compiles. The verifier checks that the theorem statement remains byte-for-byte consistent with the task interface, modulo permitted proof-body edits. This prevents agents from weakening preconditions, changing postconditions, or proving a related but easier theorem.

D. Isolated Workspaces

Each run occurs in a fresh workspace containing public task files and generic libraries, but not hidden proofs, private build artifacts, credentials, or non-public imports. The final verifier then rebuilds the submitted proof in a separate private copy. This two-step design reduces leakage from cached state and catches imports that were not available to the agent.

E. Verifier Pipeline

The verifier checks the workspace policy and theorem integrity. It then checks placeholder rejection, axiom compli-

ance, import validity, and Lean 4 compilation. A task result is counted as verified only if all checks pass. Other outcomes are recorded as failures or invalid artifacts. This keeps provider failures and zero-token runs visible without ranking them as complete results.

V. Suite Composition

Version 0.1 contains 135 active task manifests over 25 cases and 25 protocol families. The tasks are drawn from Ethereum protocols and ecosystem contracts, including security challenge cases. The suite intentionally mixes simple local storage updates with harder arithmetic and invariant-preservation tasks.

The active difficulty distribution is 40 easy, 70 medium, 24 hard, and 1 other. Stronger areas include accounting conservation, local state preservation, guarded solvency, storage writes, access-control identity, and linked-list ownership structures. Thinner areas in v0.1 include deep reentrancy reasoning, oracle manipulation, governance, timelock properties, liveness, cross-contract composition, cryptographic assumptions, and adversarial EVM behavior.

VI. Methodology

A. Versioned Manifests

Scores are comparable only relative to a fixed benchmark version. Version 0.1 records the ordered task set, source commit, run mode, budget, and compatibility fingerprints. The task set identifier is `7328d15b6379`. The harness identifier is `28a43f1654ee`. The environment identifier is `a4e6af95c296`. The source commit recorded by the manifest is `8c3cc8bc958c`.

Each task also has a task fingerprint over execution-relevant files and fields, and a task-interface fingerprint over the public files and theorem interface visible to models. This separation allows future releases to distinguish changes that require rerunning all tasks from changes that only affect a subset.

B. Rerun Planning and Leaderboards

The rerun planner compares old and new manifests. Harness, mode, budget, or environment changes force broad reruns. Task additions, removals, or changed fingerprints affect only the relevant task subset when compatible. Published leaderboards separate complete rows from partial rows, so incomplete model runs remain visible without being ranked against complete-suite runs.

C. Metrics

The primary metric is verified tasks over total active tasks. Token counts are reported when providers expose them. They are secondary because provider settings and agent wrappers differ. The v0.1 release separates complete-suite runs from partial artifacts, so the headline comparison ranks only complete runs while retaining partial runs as diagnostic evidence.

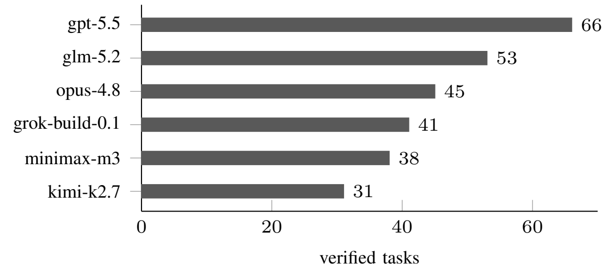

Fig. 2. Complete-suite v0.1 scores. The best complete run verifies 66 of 135 tasks; no run reaches half of the suite.

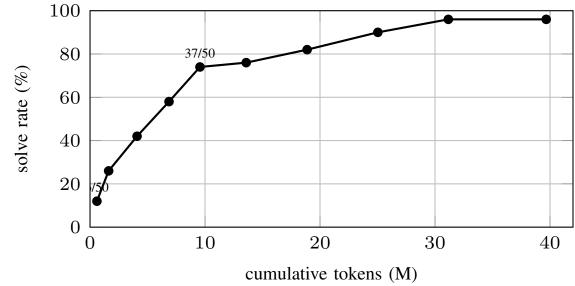

Fig. 3. MiniMax M3 budget-scaling cascade on a selected 50-task panel. Solved tasks are skipped at later profiles. The run reaches 48/50 before saturating.

VII. Preliminary Results

Table V reports every complete-suite v0.1 run published in the release artifacts. GPT-5.5 verifies 66 of 135 tasks, the strongest complete score in the release. GLM 5.2 verifies 53 tasks, Opus 4.8 verifies 45, grok-build-0.1 verifies 41, MiniMax M3 verifies 38, and Kimi K2.7 verifies 31. No complete run crosses the halfway point. The spread matters: current agents do not fail uniformly. Still, the best result leaves 69 tasks unverified. The suite therefore measures partial progress without treating proof generation as solved.

Table VI reports the non-complete artifacts from the same v0.1 release. It includes partial and invalid runs. The most notable partial artifact is GPT-5.5 Pro, which verifies 35 of 54 attempted tasks. That is a high within-run rate, but it is not a complete-suite result and therefore should not be ranked against complete 135-task runs. Invalid zero-usage artifacts are retained in the release manifest because they document failed provider or scheduling attempts rather than proof capability.

The MiniMax budget-scaling sidecar in Figure 3 answers a different question from the leaderboard. It does not rerank models on the full benchmark. Instead, it selects a 50-task panel consisting of the 38 tasks solved by MiniMax M3 in the complete v0.1 run, topped up with 12 tasks solved by GPT-5.5 but not MiniMax, and reruns only the still-unsolved tasks at increasing budgets. On this selected panel, MiniMax reaches 48 of 50 tasks at the high budget profile, or 96%. This indicates that some apparent model gaps are budget-sensitive. The experiment is reported as a sidecar because its adaptive cascade methodology is not the same as the complete-suite

TABLE II
BENCHMARK SUITE COMPOSITION IN V0.1.

Case	Tasks	Hard	Dominant property
alchemix/earmark_conservation	5	2	accounting_conservation
balancer/reclamm_swap_rounding	1	1	arithmetic_rounding
cork/pool_solvency	1	1	accounting_bound
damn_vulnerable_defi/side_entrance	5	0	balance_credit_update
ethereum/deposit_contract_minimal	5	0	monotonic_counter
forgeyields/global_solvency	7	0	guarded_solvency
ipor/plasma_vault_redeem_split	2	0	fee_payout_bound
kleros/sortition_trees	6	1	weighted_selection
lagoon/guardrails	3	0	compliance_boundary
lido/vaulthub_locked	5	1	accounting_bound
nexus_mutual/ramm_price_band	4	0	price_computation
onedelta/caller_address_integrity	10	0	access_control_identity
paladin_votes/stream_recovery_claim_usdc	26	4	authorization_state
piku/fund_conservation	4	0	accounting_conservation
polaris/bonding_curve	4	0	reserve_state_transition
polygon/agglayer_bridge	2	0	authorization_state
reserve/auction_price_band	4	1	price_computation
rootstock/flyover_quote_lifecycle	3	0	lifecycle_accounting
safe/owner_manager_reach	15	10	linked_list_invariant
term_finance/term_auction_clearing	1	0	accounting_and_rate_guard
termmax/order_v2_buy_xt_single_segment	1	0	reserve_state_transition
usual/dao_collateral	5	0	accounting_conservation
wildcat/borrow_liquidity_safety	1	0	accounting_bound
zama/erc7984_confidential_token	12	0	supply_update
zodiac/roles_decoder_faithfulness	3	3	calldata_decoder_metadata

TABLE III
MOST REPRESENTED PROPERTY CLASSES IN V0.1.

Property class	Tasks
accounting_conservation	14
accounting_bound	14
access_control_identity	10
authorization_state	9
linked_list_invariant	8
guarded_solvency	7
accounting_update	6
price_computation	5

TABLE IV
PROOF-FAMILY COVERAGE IN V0.1.

Proof family	Tasks
state_preservation_local_effects	46
functional_correctness	32
authorization_enablement	25
protocol_transition_correctness	22
refinement_equivalence	10

leaderboard.

Manual inspection suggests that agents are strongest on local state updates, direct arithmetic equalities, and proofs where the needed lemmas are close to the target theorem. Failures often involve larger invariant chains. They also involve non-obvious arithmetic normalization, stateful reasoning across several fields, or finding the right helper lemma. This aligns with the suite composition. Local accounting and storage obligations are better represented and currently easier. Compositional protocol reasoning remains a future target.

TABLE V
PRELIMINARY COMPLETE-SUITE V0.1 RESULTS. ROWS ARE RANKED ONLY FOR RUNS MARKED COMPLETE BY THE RELEASE MANIFEST.

Rank	Model	Verified	Tok./task
1	gpt-5.5	66/135 (48.9%)	0.4M
2	glm-5.2	53/135 (39.3%)	0.5M
3	opus-4.8	45/135 (33.3%)	0.7M
4	grok-build-0.1	41/135 (30.4%)	0.8M
5	minimax-m3	38/135 (28.1%)	0.8M
6	kimi-k2.7	31/135 (23.0%)	0.3M

TABLE VI
NON-COMPLETE V0.1 RELEASE ARTIFACTS. THESE ROWS DOCUMENT PARTIAL OR INVALID RUNS. THEY ARE NOT RANKED WITH COMPLETE-SUITE RUNS.

Model	Status	Verified	Coverage	Reusable
gpt-5.5-pro	partial	35/54	54/135	54/54
virtuals/deepseek-v4-flash	partial	30/135	135/135	135/135
virtuals/deepseek-v4-pro	partial	28/126	126/135	126/126
grok-4.3	partial	25/135	135/135	135/135
mimo-v2.5	partial	13/135	135/135	135/135
Step37	partial	1/135	135/135	12/135
deepseek-v4-flash	invalid	0/5	5/135	0/5
deepseek-v4-pro	invalid	0/5	5/135	0/5
mimo-v2.5	invalid	0/5	5/135	0/5

VIII. DISCUSSION AND LIMITATIONS

Ethereum Verification Benchmark measures proof synthesis under a fixed formalization. It does not measure whether an agent can discover the right specification, identify an unknown vulnerability, or decide whether the formal model matches the deployed bytecode. Those are separate assurance problems.

The benchmark is therefore best understood as one layer of an AI-assisted verification stack: once engineers choose the property and formal model, can an agent discharge the proof?

The v0.1 suite is also incomplete as a representation of Ethereum security. It covers accounting, storage, access control, and solvency properties. It has less coverage of reentrancy, oracle manipulation, governance, L2 messaging, zero-knowledge protocol integration, cryptographic primitives, and multi-contract composition. The immediate roadmap is to expand these families while preserving the same verifier discipline and manifest-based comparability.

The model comparison should be read cautiously. The release artifacts record per-task traces and hashes. They also record token totals and run caveats. Providers expose different sampling controls and execution environments. For that reason, v0.1 reports verified-task counts rather than a universal ranking of models.

IX. Conclusion

AI agents can already discharge some proof obligations in formal smart contract verification, but the capability needs to be measured with complete, reproducible tasks that reject shortcut solutions. Ethereum Verification Benchmark provides such a benchmark for Lean 4 proofs of Ethereum contract correctness. Its v0.1 suite contains 135 tasks from 25 real or ecosystem-relevant cases, enforces no-placeholder proof rules, records compatibility fingerprints, and publishes per-model artifacts. The v0.1 results give a concrete baseline: the best complete run verifies 66 of 135 tasks. Several classes of protocol reasoning remain mostly unsolved.

Acknowledgment

The public benchmark repository is available at https://github.com/lfglabs-dev/ethereum-verification-benchmark. The v0.1 release artifacts were published on June 16, 2026.

References

[1] Certora, "Certora prover documentation," https://docs.certora.com/, 2026.

[2] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the ethereum virtual machine," in *IEEE Computer Security Foundations Symposium*, 2018.

[3] G. Rosu and T. F. Serbanuta, "An overview of the k semantic framework," in *Journal of Logic and Algebraic Programming*, 2010.

[4] Foundry, "Foundry book," https://book.getfoundry.sh/, 2026.

[5] K. Zheng, J. M. Han, and S. Polu, "miniF2F: A cross-system benchmark for formal olympiad-level mathematics," in *International Conference on Learning Representations Workshop*, 2022.

[6] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "LeanDojo: Theorem proving with retrieval-augmented language models," in *Advances in Neural Information Processing Systems*, 2023.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," in *arXiv preprint arXiv:2107.03374*, 2021.

[8] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *International Conference on Learning Representations*, 2024.

[9] T. Bang, H. H. Nguyen, D. Nguyen, T. Trieu, and T. Quan, "Verification of ethereum smart contracts: A model checking approach," *International Journal of Machine Learning and Computing*, vol. 10, no. 4, pp. 588–593, 2020.

[10] E. P. Keilty, K. Nelaturu, B. Wu, and A. Veneris, "A model-checking framework for the verification of move smart contracts," in *IEEE 13th International Conference on Software Engineering and Service Science*, 2022, pp. 1–7.

[11] E. A. Napoli, F. Barbàra, V. Gatteschi, and C. Schifanella, "Leveraging large language models for automatic smart contract generation," in *IEEE 48th Annual Computers, Software, and Applications Conference*, 2024, pp. 701–710.