LACERTA-BENCH: EVALUATING PROVABLY CORRECT AND EFFICIENT BINARY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Code-generation benchmarks typically evaluate programs using finite test suites, while formal-verification benchmarks usually prove properties of source code. We study the joint evaluation of executable efficiency and machine-checked correctness of the emitted binary. We introduce LACERTA, a Lean 4 framework for reasoning about binary programs, alongside LACERTA-BENCH, a benchmark built on LACERTA for evaluating binary generation from high-level specifications. LACERTA-BENCH comprises 400 tasks derived from competitive-programming problems, each with a formal specification; batch tasks use a target-independent byte-stream interface. Models generate a binary program together with a machine-checked proof of its correctness, while deterministic gas ranks programs on their efficiency. We evaluate six language models on 100 problems under a 30-minute budget: only one system produces any accepted proof, on 20 problems. On ten of these problems with five eight-hour attempts each, 98% of attempts pass the public samples but only 55% yield a proved, scoreable artifact. Accepted binaries can differ in gas by orders of magnitude, so proof completion and execution efficiency distinguish different aspects of generated programs.

1 INTRODUCTION

Coding benchmarks usually ask whether generated programs pass a collection of tests. HumanEval, APPS, and competition-level benchmarks evaluate model-generated programs with this protocol (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022). Passing a finite suite establishes agreement on the tested inputs, not correctness for every admissible input.

Proof benchmarks and source-verification systems evaluate different objects. miniF2F and LeanDojo accept machine-checked proofs of formal statements (Zheng et al., 2022; Yang et al., 2023), while program verifiers establish functional properties of source programs (Leino, 2010; Lattuada et al., 2023). We evaluate the emitted executable against a formal contract and compare accepted executables under a fixed execution-cost measure.

The evaluation object is a **proved binary**: an executable artifact and a proof about that same artifact. Binary-level reasoning must connect instructions, memory, and host calls to the problem-domain objects. LACERTA-BENCH makes this connection reviewable by separating three components. A *semantic specification* states valid inputs and correct outputs over integers, lists, graphs, or paths. A *codec* defines their byte representation. The binary implements this interface. For every valid input, its proof must establish termination with bytes that parse to a correct output. Reviewers can therefore examine the mathematical contract without reading instruction-level proof invariants.

This factorization allows one specification–codec pair to remain fixed across different binary implementations. The contract is agnostic to how the binary is generated. Each implementation requires its own proof, but changing the implementation does not require changing or re-reviewing the mathematical statement.

The same separation supports optimization with correctness preserved as an acceptance condition. A recent DEFLATE case study optimizes a Lean implementation while retaining a round-trip theorem (Morrison, 2026). LACERTA-BENCH places that condition at the binary boundary: an agent may change the implementation or how it generates the binary, but each candidate needs a proof against the fixed contract before it contributes to the efficiency comparison.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We present LACERTA, a Lean 4 framework (de Moura & Ullrich, 2021) with a small-step execution relation and a proved-equivalent executable step function. LACERTA-BENCH builds on it with 400 competitive-programming specifications, proof-isolated workspaces, and an evaluation service. Kernel-checked correctness covers all valid inputs; deterministic gas compares execution cost on a fixed test corpus and schedule.

We choose WebAssembly as a practical initial target for its complete formal specification, maintained using SpecTec (Youn et al., 2024; Rossberg, 2025). LACERTA independently implements its execution semantics in Lean, with the interpreter tested against the official conformance suite. The benchmark contracts are independent of the instruction set; extending the verification stack to ARM and x86 remains future work.

The contributions are:

- **A binary-level evaluation object.** We connect a contract stated over problem-domain values to the execution of a concrete WASM module, so a model must produce both an artifact and its proof (Sections 2.1 and 3.1).
- **A modular benchmark.** A reviewed specification and codec can be reused across binary implementations. The corpus instantiates this design on 400 competitive-programming problems (Sections 2.1 and 2.2).
- **A proof-qualified efficiency protocol.** A kernel-checked proof establishes the universal contract; deterministic gas over fixed inputs then compares submissions accepted by both the proof and operational checks (Section 3.2).
- **An evaluation of six language models.** Across 100 problems under a 30-minute budget, one system produces accepted proofs on 20 problems and the other five on none. On ten of these problems with five eight-hour attempts each, public samples pass in 98% of attempts, while 55% yield a proved, scoreable artifact; rankings by scoreable rate and recorded gas differ (Section 4).

Section 2 describes the benchmark and its construction, Section 3 presents the evaluation protocol, and Section 4 reports the experiments. Section 5 discusses related work, limitations, and conclusions.

2 LACERTA-BENCH

LACERTA-BENCH contains 400 competitive-programming tasks with formal specifications and byte-level interfaces. Each batch task requires a binary implementation and a Lean proof that it satisfies the specified contract. We describe the task format, benchmark construction, and quality-assurance process below.

2.1 TASK SPECIFICATION AND SUBMISSION FORMAT

For each batch task, the model receives a formal specification and byte codec, public samples, and a workspace containing a placeholder binary and a theorem template. It submits a replacement `program.wasm` and a self-contained Lean proof in `Proof.lean` that the submitted binary satisfies the contract. Batch tasks use entry `main` and evaluate one contract–program pair.

Let $\mathbb{B}^*$ denote the set of finite byte strings. A benchmark problem P pairs a semantic specification S with a byte codec C:

$$S = (I, O, V, R), \qquad C = (E, D), \qquad P = (S, C), \tag{1}$$

whose components have distinct roles:

- I and O are the semantic input and output types. They contain mathematical objects such as natural numbers, lists, graphs, or paths.
- $V : I \to \mathsf{Prop}$ is the input-validity predicate. $V(i)$ records the constraints under which the contract requires an answer.
- $R : I \to O \to \mathsf{Prop}$ is the correctness relation. $R(i, o)$ means that o is an acceptable answer for i. A relation, rather than a reference function, permits several correct outputs.

- $E : I \to \mathbb{B}^*$ encodes a semantic input as the bytes presented to the program.
- $D : I \to \mathbb{B}^* \to \mathsf{Option}(O)$ parses output bytes. A result $\mathsf{some}(o)$ is a successful parse, while none rejects the bytes. The input i is available to D because output shape may depend on an input length, graph size, or other problem parameter.

This split keeps choices such as decimal text and delimiters in C, while S states the mathematical property that every implementation must establish.

To connect a candidate implementation to this contract, we describe its observable byte-stream behavior by a semantic, not necessarily computable denotation

$$p : \mathbb{B}^* \to \mathsf{Option}(\mathbb{B}^*). \tag{2}$$

For input bytes x, $p(x) = \mathsf{some}(b)$ represents normal termination after writing b; none means no such execution exists. This denotation describes program behavior independently of any execution timeout.

Correctness then asks whether this byte-level behavior realizes the mathematical requirement for every valid input:

$$\begin{aligned}\mathsf{Solves}(P, p) \;\equiv\; \forall i : I,\; V(i) \to \exists b : \mathbb{B}^*, o : O.\; & p(E(i)) = \mathsf{some}(b) \\ & \wedge\; D(i, b) = \mathsf{some}(o) \;\wedge\; R(i, o).\end{aligned} \tag{3}$$

For each valid input i, the theorem follows the data through encoding, execution, and decoding: p must terminate on $E(i)$, its output bytes b must parse as a value o, and that value must satisfy $R(i, o)$. Correctness therefore covers both the computed answer and its byte interface. The guarantee applies to canonical encodings $E(i)$; agreement with an external checker's presentation rules is part of contract validation (Section A.5).

The contract remains fixed across candidate implementations. Figure 1 illustrates the Missing Number specification, codec signatures, and required proof statement.

(a) Specification and codec

`Spec.lean`

```
def n (xs : List Nat) : Nat := xs.length + 1
def Valid (xs : List Nat) : Prop :=
  2 ≤ n xs ∧ n xs ≤ 2 * 10 ^ 5 ∧ xs.Nodup
    ∧ ∀ x ∈ xs, 1 ≤ x ∧ x ≤ n xs
def Correct (xs : List Nat) (m : Nat)
    : Prop := 1 ≤ m ∧ m ≤ n xs ∧ m ∉ xs

def encodeInput (xs : List Nat)
    : ByteArray := ...
def parseOutput (_ : List Nat)
    (bs : ByteArray) : Option Nat := ...
```

(b) Proof obligation

`Proof.lean`, fixed by the judge

```
theorem program_solves :
  LacertaEval.Solves problem
    (Wasm.programOf «module») :=
  by
    sorry -- proof to supply
```

Figure 1: Missing Number: (a) mathematical specification and codec signatures; (b) required theorem. `problem` packages (a), and «`module`» represents the submitted binary. Codec bodies are omitted; `sorry` marks the proof body to supply.

2.2 Benchmark construction

Lacerta-Bench formalizes all 400 tasks in the CSES Problem Set (Laaksonen et al., 2026), covering 18 categories such as arithmetic, sorting, dynamic programming, graphs, strings, and geometry.

Specifications reuse Mathlib definitions (The mathlib Community, 2020) and shared vocabulary for paths, matchings, and query traces. Each specification defines what makes an output correct, without prescribing an algorithm. For example, a shortest-path problem accepts any valid path of minimum length, even when several paths meet that requirement. Table 2 gives the corpus coverage and specification sizes.

2.3 QUALITY ASSURANCE

Every specification was audited by a human against the original problem statement. Each task has a reference Lean implementation and a machine-checked proof that it satisfies the specification, establishing that the specification admits a solution. All reference implementations pass their respective problem tests. These reference solutions support benchmark validation and are withheld from evaluated models. Corpus CI additionally compiles the specifications, builds each batch checker, and verifies acceptance of the official sample answers (Section A.1).

CSES problem statements and source-level solutions are publicly available and may have appeared in model training data. Our benchmark-specific WASM implementations and Lean correctness proofs were not publicly available during evaluation. Knowing a source-level solution does not satisfy the benchmark's central requirement: constructing a machine-checked proof that the submitted binary meets its formal specification. Prior familiarity may nevertheless assist implementation and proof construction.

3 EVALUATION PROTOCOL

Evaluation checks correctness and measures execution cost for the same submitted binary. The judge first runs the problem tests and meters gas. Submissions that pass within the execution limits proceed to proof checking; those with accepted proofs receive an efficiency score. Figure 2 summarizes the protocol.

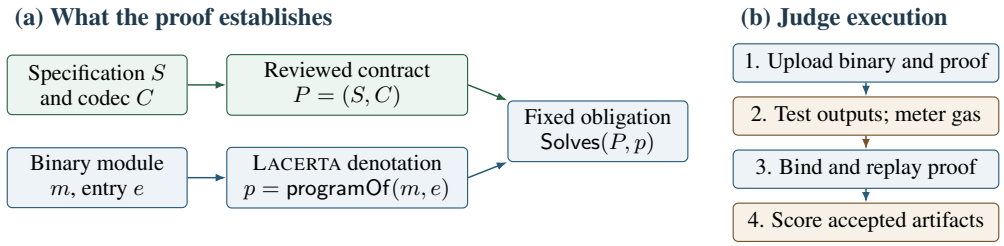

Figure 2: Evaluation path. (a) The contract P and module denotation p determine the proof obligation. (b) The judge tests and meters the binary before proof replay; both stages must accept for the submission to contribute to the gas score.

3.1 CORRECTNESS VERIFICATION WITH LACERTA

The judge reconstructs the proof obligation from the submitted binary and the fixed task contract. It prints the binary as WASM text and elaborates that text into a Lean syntax tree m, the *represented module*. An export name e identifies the entry point whose behavior is to be proved.

LACERTA assigns the module deterministic, instruction-level execution semantics. For input bytes $x \in \mathbb{B}^*$ and output bytes $y \in \mathbb{B}^*$, $\mathsf{Runs}(m, e, x, y)$ states that execution reaches normal termination after writing exactly y. Determinism gives each fixed (m, e, x) at most one normally terminating output. The benchmark packages this relation as the partial function $\mathsf{programOf}(m, e)$ and proves

$$\mathsf{programOf}(m, e)(x) = \mathsf{some}(y) \quad \Longleftrightarrow \quad \mathsf{Runs}(m, e, x, y). \tag{4}$$

Substituting this denotation into Eq. (3) gives the submitted theorem:

$$\mathsf{Solves}\big(P, \mathsf{programOf}(m, e)\big). \tag{5}$$

The contract stays fixed, and each submitted module must establish its own instance of this theorem. Comparator (Böving, 2025) checks the theorem's name and type against the reconstructed obligation, rejects unapproved axioms, and replays the proof through a Lean kernel. A submission is *proof-accepted* when Comparator accepts this theorem.

LACERTA's executable interpreter is proved to agree with its transition semantics. An Iris-Lean program logic (de Medeiros et al., 2026) supports proofs about loops, memory, and function calls;

Section A.2 gives the interpreter and proof-infrastructure details. The interpreter passes 99.4% of records in the official WASM conformance-suite run (Section A.3).

The guarantee concerns the represented module under the formal execution model, modeled host, and audited contract. Applying it to the uploaded bytes relies on faithful conversion into the Lean representation. Sections A.4 and A.5 detail the artifact correspondence and trusted components.

3.2 EFFICIENCY MEASUREMENT AND SCORING

Gas measures execution work through fixed integer charges for instructions, memory operations, and host I/O. For a fixed binary, meter, schedule, and input, gas is deterministic. The judge executes a metering copy of the binary under Wasmtime (Bytecode Alliance, 2026); Section A.6 specifies the schedule.

A submission is *scoreable* when its proof is accepted and it passes every full-test execution below the gas cap. A locally accepted proof can accompany a binary that the official judge rejects at the gas stage. Only scoreable artifacts receive an efficiency score.

For each test, gas is compared with the least gas used by a scoreable submission on that test in the leaderboard snapshot. Matching this reference scores one, and using at least 100 times its gas scores zero. Intermediate ratios follow the logarithmic rule in Eq. (8); the problem score averages the per-test scores.

4 EXPERIMENTAL EVALUATION

The proof obligation fixes what a binary must compute but leaves its execution cost unconstrained. Evaluating language models against it raises two questions. Across the corpus, how often does an attempt end with a proved binary at all? And where proofs are obtained, what separates a sample-passing binary from a proved one, a proved binary from a scoreable one, and a scoreable binary from an efficient one? The first question concerns performance across the corpus and calls for a representative sample of problems; the second concerns repeatability and efficiency and calls for repeated attempts with enough time to complete proofs. We therefore evaluate in two stages: 100 problems under a 30-minute budget, then five eight-hour attempts at ten of these problems.

4.1 STUDY DESIGN

Six language models are evaluated under one protocol, using the same OpenCode harness and 64-vCPU machine configuration. Each attempt starts from an isolated workspace containing the problem's specification, codec, public samples, a placeholder binary, and the proof scaffold, without any solution binary or completed proof. The agent receives the same prompt for every problem and system up to the problem name and CSES identifier, may build or hand-write the WASM binary, and can run the local judge on the public samples. An attempt is *proved* when the local comparator accepts a proof within the generation budget, and *scoreable* when the official judge additionally accepts that binary on every hidden test within the fixed gas cap of 7.5×10^{11} per test. Attempts that stop without an accepted proof count as unsuccessful, and rates and mean charges include them. Sections B.1 and B.3 give the shared configuration and complete prompt.

Stage 1: 100 problems, 30 minutes. The first evaluation measures performance across the corpus. Each system makes one attempt at each of 100 problems under a 30-minute budget, for 600 attempts. The 100 problems are sampled uniformly at random from the 400-problem corpus and span 16 of the 18 CSES sections (Section B.2). Because the sample is random, the resulting pass@1 estimates each system's success rate on the benchmark as a whole rather than on a chosen subset, and the distribution of outcomes across sections shows where the benchmark has headroom.

Stage 2: ten problems, eight hours. The first evaluation provides limited information about proof completion: most attempts end with a binary but an unfinished proof, and only one system completes any proof within 30 minutes (Section 4.2). One short attempt per problem does not show whether the other systems can complete proofs at all or how reliably any system does so, and it yields too few scoreable artifacts to compare efficiency. The second evaluation therefore gives each system five independently initialized eight-hour attempts at a subset of ten problems sampled uniformly at

random from the 100, for a total of 300 attempts. The ten problems span eight CSES sections and a range of recorded solver counts (Table 8), covering range queries, maximum flow, geometry, and trees. This subset provides variation in both algorithmic approaches and the structures required for proofs. The five attempts per problem give pass@5 for each system. The eight-hour budget produces 165 scoreable artifacts, enough to compare execution efficiency: an artifact's relative gas is its total hidden-test gas divided by the lowest total among the scoreable artifacts for the same problem. The reference is an observed minimum, not a known optimum. Section B.6 reports attempt-level rates, generation cost, and completion time.

4.2 Proof completion under a 30-minute budget

Under the 30-minute budget, 20 of 600 attempts end with an accepted proof, and every one belongs to GPT-6 Astra (Fig. 3a). The other five systems produce none. Most attempts do produce a program: 456 replace the placeholder binary, but only 20 yield an accepted proof (Table 6). All 20 accepted proofs are also scoreable.

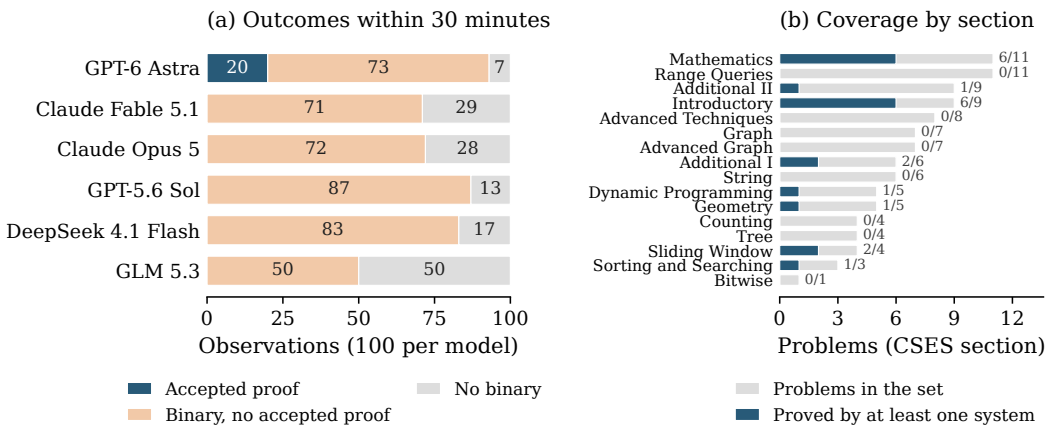

Figure 3: The 30-minute evaluation: 100 problems, one attempt per system. (a) Outcome of every attempt; a binary counts once it replaces the placeholder. (b) Problems in the set by CSES section, with those proved by at least one system.

The 20 proved problems are concentrated in the Mathematics and Introductory sections, which contribute 12 of them, whereas none of the 11 range-query, eight advanced-technique, 14 graph, or six string problems is proved (Fig. 3b). Proved problems have a median of about 9,300 recorded CSES solvers, against roughly 3,500 for the set as a whole, although three of the 20 rank among the 50 least-solved problems of the corpus. The accepted proofs are completed late in the budget: their median completion time is 26 minutes, and seven are completed in the final minute. Under this budget the evaluation therefore separates systems mainly by whether they complete any proof.

4.3 Proof acceptance and execution limits under an eight-hour budget

With eight hours and five attempts per problem, sample success is nearly saturated: 294 of 300 attempts pass the public tests, but 190 produce accepted proofs and 165 are scoreable (Fig. 4). The proof and execution requirements therefore distinguish outcomes that the samples largely leave unresolved.

The remaining failures differ across systems. Opus has 15 proved artifacts rejected at the gas stage, whereas Sol has 22 sample-passing attempts without accepted proofs (Table 7). The same separation persists at the task level: Opus proves 47 of its 50 attempts, yet it has scoreable artifacts on only eight of the ten problems, because every proved Convex Hull and Fixed-Length Paths II artifact it produces exceeds the gas cap (Table 5). Astra proves every attempt, yet its best artifact matches the lowest observed gas on only one of the ten problems. Efficiency therefore distinguishes outputs even when every attempt is proved. Failed attempts also consume generation resources; Section B.6 reports failure-inclusive charges and completion times.

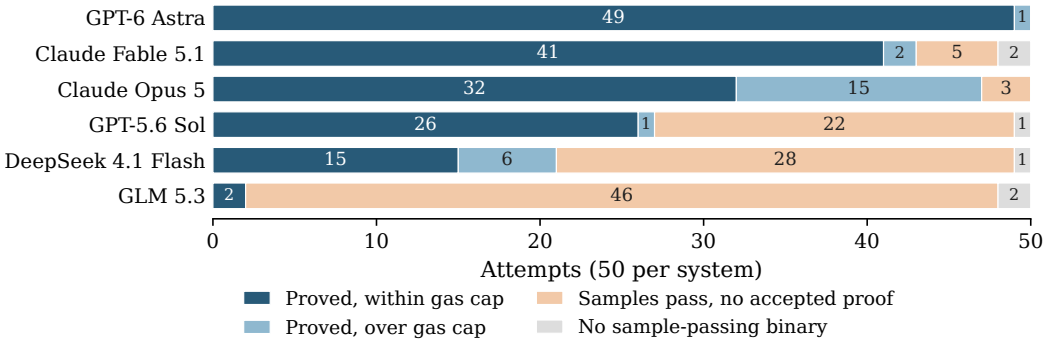

Figure 4: The eight-hour evaluation: all 300 attempts (50 per system). Labels count outcomes; sample-passing attempts without an accepted within-budget proof are shown in orange.

4.4 Efficiency requirements expose task-specific differences

Acceptance gives equal credit to every proved artifact below the gas cap. To compare execution cost, we count, for each system, the problems on which it has a scoreable artifact that uses at most τ times the lowest gas observed for that problem, where $\tau \geq 1$ is a chosen multiple. Figure 5a plots this count, as a fraction of the ten problems, against τ. At $\tau = 1$ a problem counts only if the system holds the gas record for it; at $\tau = 2$ it counts if the system's best artifact uses at most twice the record; as τ grows, every problem on which the system has any scoreable artifact eventually counts.

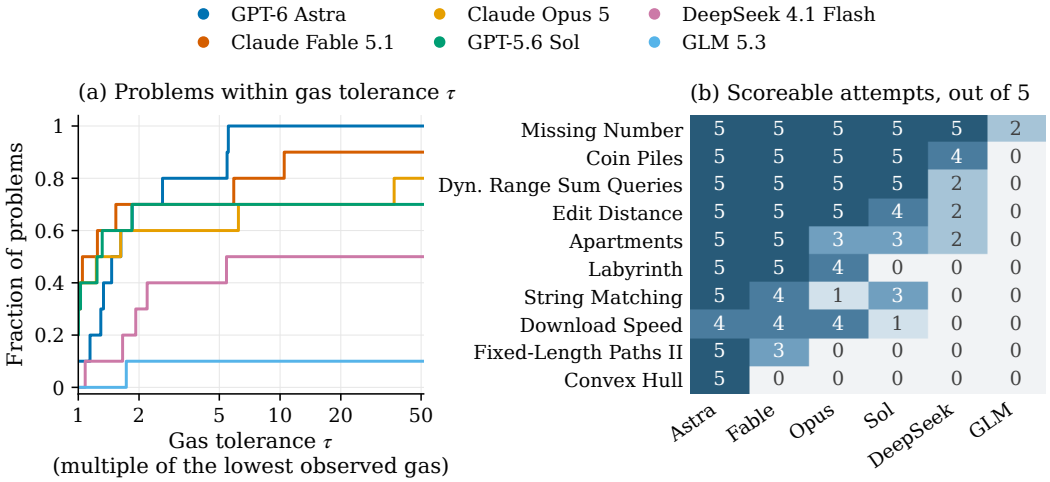

Figure 5: Execution gas among scoreable artifacts. (a) Fraction of the ten problems on which a system has a scoreable artifact using at most τ times the lowest gas observed for that problem; at $\tau = 1$ this is the fraction of problems on which the system holds the record. (b) Scoreable attempts out of five per problem and system; rows are ordered by total scoreable attempts.

At $\tau = 1$, Opus holds the gas record on four problems, Fable on three, Sol on two, and Astra on one, although Astra alone has a scoreable artifact on every problem (Fig. 5b). At $\tau = 2$, seven problems count for each of Astra, Fable, and Sol. All ten count for Astra only from $\tau = 5.5$, whereas Sol's seven all count by $\tau = 1.8$ and Sol has no scoreable artifact on the remaining three. No system leads at every τ, so a ranking by proof completion does not transfer to a ranking by efficiency.

The same count can involve different problems: of the seven problems counted for Astra and for Sol at $\tau = 2$, only four are shared. Sol alone has Dynamic Range Sum Queries and Download Speed within twice the record; Astra alone has Convex Hull. Together they account for all ten problems. Such task-specific coverage is lost in an aggregate ranking. Beyond obtaining a proof-accepted

program, the benchmark therefore poses a second task: finding lower-gas implementations of the same contract. The evidence here is variation across independent attempts on ten selected tasks, using observed references rather than known optima. It establishes distinct proved alternatives for efficiency comparison, while iterative proof-preserving optimization remains a further experiment.

5 CONTEXT, LIMITATIONS, AND CONCLUSION

LACERTA-BENCH combines a reusable formal contract, proof acceptance at the binary boundary, and empirical execution cost. We relate this combination to prior benchmarks and delimit what the evaluation establishes.

5.1 RELATED WORK

Table 1 contrasts the judged artifacts and evaluation requirements. LACERTA-BENCH's distinguishing object is the binary together with a proof of its value-and-byte contract.

Table 1: Evaluation objects and requirements. ITP denotes interactive proof with kernel replay; ATP denotes automated verification. Efficiency specifies the execution-cost measurement. Dataset sizes refer to the cited releases.

Benchmark	Artifact	Specification	Proof	Efficiency	Size	Language
HumanEval, APPS	source	tests	–	–	164; 10,000	Python
EvalPlus	source	augmented tests	–	–	164	Python
Mercury	source	tests	–	runtime	1,889	Python
miniF2F	proof	formal statement	ITP	–	488	Lean, Isabelle
DafnyBench	source	formal annotations	ATP	–	782	Dafny
FVAPPS	source	formal	ITP	–	4,715	Lean
VERINA, CLEVER	source, spec, proof	formal	ITP	–	189; 161	Lean
LACERTA-BENCH	executable binary	formal + byte codec	ITP, kernel replay	deterministic gas	400	WASM (any source)

Generation and proof benchmarks. HumanEval, APPS, and AlphaCode evaluate generated programs through tests (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022); EvalPlus strengthens testing and Mercury measures runtime (Liu et al., 2023; Du et al., 2024). miniF2F and LeanDojo evaluate mathematical proofs (Zheng et al., 2022; Yang et al., 2023). DafnyBench, FVAPPS, VERINA, and CLEVER address source programs, specifications, or their proofs (Loughridge et al., 2025; Dougherty & Mehta, 2025; Ye et al., 2026; Thakur et al., 2025). Our acceptance theorem instead concerns the submitted executable and its byte interface.

Verification infrastructure. Proof-carrying code associates code with checkable evidence (Necula, 1997); verified compilers establish source-to-target preservation (Leroy, 2009; Kumar et al., 2014). Source verifiers include Dafny, Verus, and AutoVerus (Leino, 2010; Lattuada et al., 2023; Yang et al., 2025), while prior work mechanizes WASM and develops module logics (Watt, 2018; Watt et al., 2019; 2021; Rao et al., 2023). LACERTA-BENCH builds on this verification perspective to compare independently generated binaries under reusable contracts and measured gas.

5.2 LIMITATIONS AND FUTURE WORK

The evaluation covers six systems on 100 problems under a 30-minute budget and on ten of these problems under repeated eight-hour attempts, all with a common agent setup. Broader experiments across agent configurations, budgets, and problems are needed to assess how these findings generalize. Competitive-programming problems provide a controlled setting; extending the benchmark to stateful, concurrent, and externally interacting software remains future work. Further work will also investigate targets such as ARM and x86, extending the framework beyond WASM.

5.3 CONCLUSION

Proof completion and execution efficiency are distinct evaluation targets. Sample-passing binaries can lack accepted proofs, while proved binaries can still incur substantially different execution

costs. LACERTA connects readable contracts to represented binaries, enabling LACERTA-BENCH to compare implementations under a fixed correctness requirement.

AI USE STATEMENT

Generative AI tools were used to help organize, draft, and edit portions of this manuscript, to identify potentially relevant literature, and to inspect the implementation for claims that required qualification. The authors checked AI-assisted technical statements against the cited primary sources and the benchmark implementation, reviewed all generated prose, and take responsibility for the final content of the paper. Generative models are also the systems evaluated by the benchmark; Section 4 documents that use.

REFERENCES

Henrik Böving. Comparator. https://github.com/leanprover/comparator, 2025. Lean FRO. Software repository.

Bytecode Alliance. Wasmtime. https://wasmtime.dev/, 2026. WebAssembly runtime. Accessed September 2026.

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. doi: 10.48550/arXiv.2107.03374.

Markus de Medeiros, Sergei Stepanenko, Zongyuan Liu, Oliver Soeser, Fernando Leal, Alvin Tang, Max Vistrup, Ralf Jung, Mario Carneiro, Joseph Tassarotti, Michael Sammler, and Lars Birkedal. Iris in Lean. *arXiv preprint arXiv:2609.24252*, 2026. doi: 10.48550/arXiv.2609.24252.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 72–79. IEEE, 2025. doi: 10.1109/LLM4Code66737.2025.00014.

Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 16601–16622, 2024. doi: 10.52202/079017-0529.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi: 10.1017/S0956796818000151.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 179–191, 2014. doi: 10.1145/2535838.2535841.

Antti Laaksonen, Roope Salmi, and Topi Talvitie. CSES problem set. https://cses.fi/problemset/, 2026. Accessed September 2026.

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. doi: 10.1145/3586037.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009. doi: 10.1145/1538788.1538814.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572, 2023. doi: 10.52202/075280-0943.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=yBgTVWccIx. arXiv:2406.08467.

Kim Morrison. Why Lean is faster than Rust. https://kim-em.github.io/blog/2026-7-24-why-lean-is-faster-than-rust/, July 2026. Accessed September 2026.

George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119, 1997. doi: 10.1145/263699.263712.

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-Wasm: Robust and modular verification of WebAssembly programs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1096–1120, 2023. doi: 10.1145/3591265.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

Andreas Rossberg. Wasm 3.0 completed. https://webassembly.org/news/2025-09-17-wasm-3.0/, September 2025. WebAssembly news post.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 128165–128199, 2025. doi: 10.52202/085713-3857.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2020. doi: 10.1145/3372885.3373824.

Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 53–65, 2018. doi: 10.1145/3167082.

Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated WebAssembly. In *33rd European Conference on Object-Oriented Programming*, volume 134, pp. 9:1–9:30, 2019. doi: 10.4230/LIPIcs.ECOOP.2019.9.

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two mechanisations of WebAssembly 1.0. In *Formal Methods – FM 2021*, volume 13047 of *Lecture Notes in Computer Science*, pp. 61–79. Springer, 2021. doi: 10.1007/978-3-030-90870-6_4.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated proof generation for Rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3454–3482, 2025. doi: 10.1145/3763174.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21573–21612, 2023. doi: 10.52202/075280-0944.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0A4Uf88pog.

Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the WebAssembly standard up to speed with SpecTec. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1559–1584, 2024. doi: 10.1145/3656440.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A BENCHMARK CONTRACT AND EVALUATOR DETAILS

A.1 CORPUS COVERAGE AND VALIDATION

Table 2 records the corpus and the selected study tasks. Human review examines the semantic specifications alongside the source problem statements. Specification validation also combines Lean reference implementations proved correct against the specifications with execution against the problem tests. The proofs connect the implementations to the formal contracts, while testing checks agreement with the expected task behavior. Corpus CI additionally builds the specifications and batch checkers and checks agreement with official sample answers. The interpreter conformance run separately tests the execution model.

Table 2: Corpus and evaluation coverage. Only categories represented in the 100-problem evaluation are listed. The full corpus contains 400 tasks across 18 categories; specification sizes have median 31 and maximum 81 non-empty lines.

Category	Corpus	30 minutes	Eight hours
Mathematics	37	11	0
Graph	36	7	2
Sorting and Searching	35	3	1
Additional II	30	9	0
Additional I	30	6	0
Advanced Graph	28	7	0
Advanced Techniques	25	8	0
Range Queries	25	11	1
Introductory	24	9	2
Dynamic Programming	23	5	1
String	21	6	1
Counting	18	4	0
Geometry	16	5	1
Tree	16	4	1
Bitwise Operations	11	1	0
Sliding Window	11	4	0

A.2 EXECUTABLE SMALL-STEP SEMANTICS

This subsection expands the execution model used in Section 3.1. A LACERTA machine configuration is written $c = \langle q, \sigma \rangle$. The component q is the control state—a running instruction sequence, a normal return, or a structural trap—and σ is the machine store containing the instantiated module, memories, tables, globals, and modeled host state. The instruction-granular relation

$$c \xrightarrow{\ell} c' \tag{6}$$

means that configuration c takes one transition labeled ℓ to configuration c'. A label identifies the WASM instruction, host call, or administrative transition responsible. Normal return and traps are distinct terminal states.

The relational semantics is also executable. The function `stepChecked?` returns the unique successor when one exists, returns no successor at a terminal configuration, and reports an internal error for malformed states outside the validation assumptions. In the equation below, ok means that this checker encountered no internal error, while the inner some contains the labeled successor. LACERTA proves that this successful result is equivalent to the relation:

$$\mathsf{stepChecked?}(c) = \mathsf{ok}(\mathsf{some}(\ell, c')) \iff c \xrightarrow{\ell} c'. \tag{7}$$

Concrete execution and symbolic proof therefore use two presentations of the same semantics rather than independently maintained interpreters. Iterating `stepChecked?` uses an interpreter step budget as a timeout; the public correctness theorem does not expose that budget.

The Lean definition of programOf uses classical choice: it selects the unique output when a normally terminating run exists and yields none otherwise. This fuel-free denotation is marked

`noncomputable`; concrete execution uses the computable, step-bounded runner `runWithFuel`, whose successful outputs are proved to agree with `programOf`. A runner timeout does not imply the absence of a terminating run.

Proving the whole-program obligation requires connecting loops, memory operations, and function calls to the mathematical result. LACERTA supports this composition with a weakest-precondition calculus and an Iris-Lean separation-logic layer (de Medeiros et al., 2026). Separation logic describes ownership of linear memory (Reynolds, 2002; Jung et al., 2018); instruction-lifting rules and representation predicates connect byte regions to words, arrays, and streams, allowing local function proofs to compose. These abstractions organize the proof while preserving the common obligation.

A.3 INTERPRETER CONFORMANCE

The archived official-suite run associated with the framework revision contains 65,142 records: 64,751 pass, 346 are skipped, 27 fail decoding, and 18 depend on an unavailable decoded module. No record reports a semantic failure, interpreter error, or exhausted step budget. The 99.4% figure uses all suite records as its denominator; it is neither instruction coverage nor validation of the benchmark's task specifications.

A.4 ARTIFACT IDENTITY

When the submission reaches the proof stage, the checker reconstructs the following chain:

1. print the uploaded `program.wasm` using the configured WASM toolchain;
2. regenerate the Lean module from that text;
3. regenerate the trusted challenge theorem from the task specification;
4. compare the candidate theorem's name, type, and axioms with the challenge; and
5. replay the accepted proof through the configured Lean kernel.

Hashes and the benchmark revision identify the artifact and contract used for a verdict. The comparator checks that the submitted theorem matches the challenge generated from that contract and represented module. The proof establishes correctness of this module; applying the guarantee to the uploaded bytes relies on faithful conversion and decoding as described in Section A.5.

A.5 GUARANTEES AND TRUST BOUNDARY

We trust LACERTA's formalization of WebAssembly execution and the task specifications to capture the intended semantics and problem requirements. We also trust the evaluation infrastructure to translate the submitted binary faithfully into its Lean representation and pair it with the correct specification. The comparator checks that the submitted theorem matches this proof obligation and uses only permitted axioms; Lean's kernel checks the proof.

A.6 GAS SCHEDULE

The current schedule gives most WASM operators unit cost, with zero-cost structural bookkeeping inherited from the meter. The explicit non-default costs are shown in Table 3. The final experimental artifact will pin the meter, Wasmtime, and WASM tooling revisions in addition to this schedule.

Table 3: Explicit components of the gas schedule. Variable byte/page charges are added to the configured operator cost; host-call charges are separate. Function entry incurs the additional one-unit charge stated below.

Operation	Configured cost component
Direct call	4
Indirect call	8
32-bit integer division	6
64-bit integer division	8
64-bit floating-point division	10
Memory growth	$100 + 50$ per page
Memory copy or fill	default $+1$ per byte
Host read or write	$10 + 1$ per byte

The meter additionally charges one unit per WASM function entry. The gas cap is part of the verdict, and a submission with a case reaching it is not scoreable.

Fix a versioned leaderboard snapshot. For a scored test case i, let g_i be the candidate's gas and let $g_i^\star$ be the least gas used by any scoreable submission in that snapshot. The case score is

$$s_i = \mathrm{clamp}\left(1 - \frac{\log(g_i/g_i^\star)}{\log 100}, 0, 1\right). \tag{8}$$

Table 4 illustrates the per-test score: matching the record gives $s_i = 1$, while using at least 100 times its gas gives zero. The problem score averages s_i over cases. Because the reference depends on the scoreable pool, the live leaderboard can change. The empirical comparisons use a frozen cohort and report total and relative gas rather than this leaderboard score.

Table 4: Per-test scores for gas multiples relative to the lowest observed scoreable gas on the same test in the snapshot. Scores are rounded to three decimal places.

Gas ratio $g_i/g_i^\star$	Case score s_i
$1\times$	1.000
$5\times$	0.651
$10\times$	0.500
$50\times$	0.151
$\geq 100\times$	0.000

B EXPERIMENTAL DETAILS

B.1 MACHINE AND AGENT HARNESS

All models use the same OpenCode harness, pinned to version 1.18.30, and the same 64-vCPU machine configuration. The environment provides Lean v4.32.2 and Mathlib v4.32.2. Each attempt runs in an isolated workspace.

Every model runs as a fixed "prover" agent with shell access, file reading and editing, search tools, a todo list, and a "lemma" subagent to which it can delegate independent lemmas. Network access and interactive questions are disabled. Reasoning effort is set to high for all models, with a 200k-token context cap and a 32k-token output cap. These settings hold the agent environment constant across models.

B.2 EVALUATION DESIGN

The evaluations use the same six models and shared problem sets, with results reported separately at 30 minutes and eight hours. An attempt succeeds when its submitted binary passes every test within the fixed gas cap of 7.5×10^{11} per test and Comparator accepts its correctness proof. An accepted local proof alone is reported as *proved*; success on both checks is reported as *scoreable*. Attempts without an accepted within-budget proof count as unsuccessful.

Stage 1 evaluates 100 problems, sampled uniformly at random from the 400-problem corpus, with one 30-minute attempt per model and problem, giving 600 attempts across the six models. Stage 2 gives each model five independently initialized eight-hour attempts on a subset of ten problems sampled uniformly at random from the 100, for a total of 300 attempts. The ten problems span eight CSES categories and a range of recorded solver counts (Table 8). Section B.4 gives the detailed outcomes.

Pass@1 is the fraction of the 100 problems solved in the single 30-minute attempt. Pass@5 is the fraction of the ten eight-hour problems solved in at least one of the five attempts. Table 5 summarizes the two settings.

Table 5: Results by model and stage. Stage 1 uses one 30-minute attempt on each of 100 problems. Stage 2 counts are out of 50 eight-hour attempts; pass@5 counts problems solved out of ten.

	30 minutes	Eight hours		
Model	pass@1	Proved	Scoreable	pass@5
GPT-6 Astra	0.20	50	49	1.00
Claude Fable 5.1	0.00	43	41	0.90
Claude Opus 5	0.00	47	32	0.80
GPT-5.6 Sol	0.00	27	26	0.70
DeepSeek 4.1 Flash	0.00	21	15	0.50
GLM 5.3	0.00	2	2	0.10

B.3 PROMPT

All runs use the same zero-shot task instruction, with only the problem name and CSES identifier substituted. The prompt is placed in the agent's system instructions so that it persists across context compaction. Project and namespace identifiers are anonymized below.

System instruction

```
Solve lacerta-bench problem {{PROBLEM}} (CSES {{CSES_ID}}, specified
in LacertaEval/Problems/CSES/{{PROBLEM}}/Spec.lean) as the
repository's README.md describes under "Solving a problem": replace
the placeholder program.wasm with a Wasm binary that solves it,
regenerate program.wat and Program.lean, and replace the sorry in
Proof.lean with a proof of program_solves without changing its
statement.

Only program.wasm and Proof.lean are submitted, so the proof must be
self-contained in that one file. You may change the program at any
time: a simpler program is often easier to prove.

The work is accepted when the binary passes every test within the
gas limit (scripts/check_problem.py {{PROBLEM}} --samples --gas
checks the samples) and Proof.lean builds without native_decide and
without the text sorry anywhere in the file (comments included),
#print axioms program_solves listing only propext, Quot.sound and
Classical.choice.

rustc (with the wasm32-unknown-unknown target), wasm-tools and
measure-gas are installed; there is no network access.
```

Initial user message

```
Begin the task given in your system instructions, and keep working
until it is fully complete.
```

B.4 DETAILED OUTCOMES

Table 6 reports all 600 Stage 1 attempts: 100 per model. Across models, 456 attempts replace the placeholder binary; 20 produce an accepted proof and pass the full tests within the gas cap. All 20 successful attempts belong to Astra, giving pass@1 of 0.20; the other models have pass@1 of zero.

Table 6: Stage 1 outcomes on 100 problems per model, with a 30-minute budget. A binary counts once it replaces the placeholder. All accepted proofs are also scoreable.

Outcome of the attempt	GPT-6	Fable	Opus	GPT-5.6	DeepSeek	GLM
Accepted proof	20	0	0	0	0	0
Binary, no accepted proof	73	71	72	87	83	50
No binary	7	29	28	13	17	50
Attempts	100	100	100	100	100	100

Table 7 classifies all 300 eight-hour attempts. Of the 110 attempts without an accepted within-budget proof, 96 end with an unfinished proof. Rejected proofs are classified using Comparator's error messages; these categories describe the final submission state.

Table 7: Outcome of every eight-hour attempt, by system.

Outcome of the attempt	GPT-6	Fable	Opus	GPT-5.6	DeepSeek	GLM
Proved and scoreable	49	41	32	26	15	2
Proved; over gas cap on a hidden test	1	2	15	1	6	0
Proof unfinished at stopping	0	5	2	19	28	42
Proof rejected: unsolved goals	0	0	0	2	0	0
Proof rejected: unknown identifier	0	0	1	0	0	1
Proof rejected: disallowed axiom	0	0	0	1	0	0
Proof rejected: other	0	0	0	0	0	2
Proof check exceeded its time limit	0	0	0	0	0	1
Binary exceeded the sample time limit	0	0	0	0	0	2
Binary wrong on a sample	0	0	0	1	0	0
Placeholder binary never replaced	0	2	0	0	1	0
Attempts	50	50	50	50	50	50

B.5 Per-problem results

Table 8 separates proof completion from compliance with execution limits. String Matching loses 11 of 24 proved artifacts at the gas stage; Convex Hull loses six of 11. All 22 proved range-query artifacts are scoreable, yet their total gas differs by a factor of 295. Passing the gas cap therefore leaves substantial differences in execution cost. *Solvers* counts recorded CSES accounts solving each problem. *Least gas* and *Spread* give the minimum total test gas and the maximum-to-minimum ratio within each problem.

Table 8: Results on the ten eight-hour problems, ordered by recorded CSES solvers (most-solved first), with 30 attempts per problem across all six models. Gas statistics use scoreable artifacts only; G denotes 10^9 gas units.

Problem	CSES section	Solvers	Tests	Proved	Scoreable	Least gas	Spread
Missing Number	Introductory	154,579	14	27	27	0.12 G	2×
Apartments	Sorting	66,998	19	19	18	2.60 G	3×
Coin Piles	Introductory	61,902	3	24	24	0.06 G	7×
Labyrinth	Graph	46,411	17	15	14	0.56 G	3×
Edit Distance	DP	43,628	17	21	21	5.97 G	3×
Dyn. Range Sum Q.	Range Queries	32,653	2	22	22	0.31 G	295×
String Matching	String	13,814	39	24	13	0.31 G	314×
Download Speed	Graph	6,157	77	14	13	2.91 G	1502×
Convex Hull	Geometry	3,969	10	11	5	2.14 G	1×
Fixed-Length Paths II	Tree	2,129	24	13	8	2.77 G	9×

B.6 Efficiency, generation cost, and time

All efficiency comparisons use the full set of 300 eight-hour attempts. For each problem, relative gas divides a scoreable artifact's total hidden-test gas by the smallest total observed among scoreable artifacts for that problem. An attempt is *within tolerance* τ if it is scoreable and its relative gas is at most τ, that is, if it uses at most τ times the lowest gas observed for its problem. The within-tolerance rate divides this count by all 50 attempts per model, including failures. These observed references are fixed across comparisons and are not known optima.

Figure 6 compares within-tolerance rates at $\tau = 2$ with mean recorded LLM charges and completion times. Mean charges include unsuccessful attempts and exclude machine costs. The time curves count final artifacts within $\tau = 2$ completed by each recorded time, using the same denominator of 50; they do not represent separate experiments at shorter budgets. Astra, Fable, and Sol produce 31, 28, and 24 artifacts within $\tau = 2$, with mean charges of $41.58, $44.45, and $101.01 per attempt. DeepSeek averages $3.55 and produces eight. By two hours, Astra has completed 28 such artifacts, Sol 12, Fable 11, and DeepSeek one.

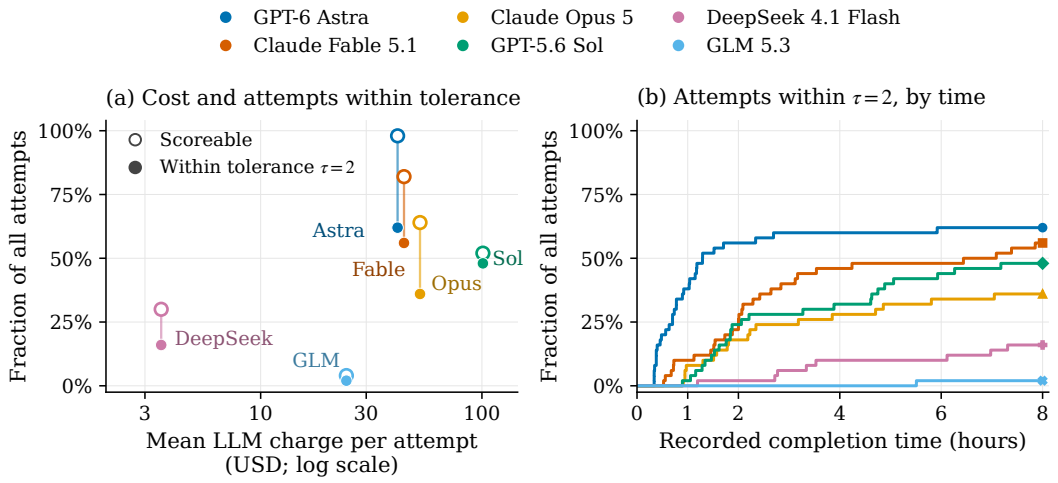

Figure 6: The eight-hour evaluation (50 attempts per system). (a) Failure-inclusive mean LLM charges versus the scoreable rate (hollow) and the rate of attempts within tolerance $\tau = 2$ (filled). Vertical segments show a change of acceptance criterion, not optimization trajectories. (b) Fraction of all attempts completing with an artifact within $\tau = 2$ by each recorded time, not predicted outcomes under shortened budgets.

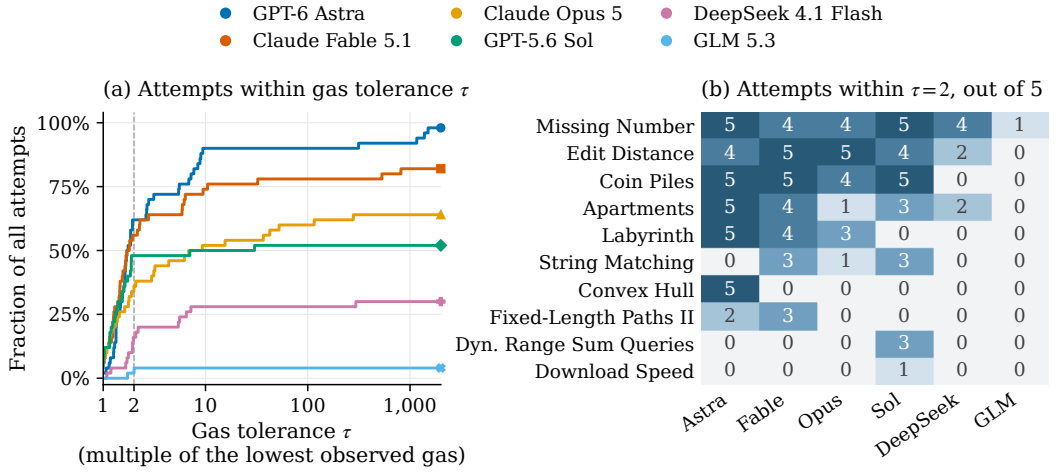

Figure 7: Attempt-level counterpart of Fig. 5 on the same 300 attempts. (a) Fraction of all attempts within gas tolerance τ; failures count as zero; the dashed line marks $\tau = 2$. (b) Attempts out of five per problem and system within $\tau = 2$; rows are ordered by their totals.

Figure 7 counts attempts rather than problems. At $\tau = 2$, Astra has 31 attempts within tolerance, Fable 28, Sol 24, Opus 18, DeepSeek eight, and GLM one; at $\tau = 1.5$ Fable has 19 to 15 each for Astra and Sol; Astra has the most at $\tau = 2$, 5, and 10.

The reference choice affects comparisons: excluding each model's own artifacts from its references changes the counts within $\tau = 2$ to 30 for Fable, 26 for Astra, and 25 for Sol. Astra's five scoreable Convex Hull artifacts then have no reference from another model. This dependence motivates reporting the gas-threshold profiles in Fig. 5, alongside proof and scoreable rates.

B.7 Effort timelines

Figure 8 shows two contrasting tasks: Missing Number and Download Speed. Every model produces a scoreable Missing Number artifact, while some obtain no accepted Download Speed proof in five attempts. Activity after the final binary change includes proof construction, checking, and other work; it is not a measure of proof authoring alone. The examples illustrate individual run histories.



Figure 8: Separate task histories, 30 attempts per panel. Within each system, the five bars are attempts 1–5 from top to bottom. Grey precedes the first binary change, orange extends to the final change, and blue follows it. Filled markers denote scoreable outcomes, hollow markers proved artifacts over the gas cap, and crosses no accepted within-budget proof. Missing milestones are placed at the clock's endpoint and time outside the budget is omitted. These are selected examples, not a representative task sample.

B.8 Case studies

A proved binary that exceeds the gas cap. DeepSeek 4.1 Flash's first String Matching attempt proved a 112-instruction binary that compares the pattern at every position. Its proof was accepted locally, but it failed the official execution check. The official judge stops at the first test over the cap, so the verdict records five tests: gas of 357, 685, 147, and 3.5×10^7 on the first four, then the cap of 7.5×10^{11} on the fifth. The scoreable Claude Opus 5 artifact for the same problem implements the prefix-function algorithm in 181 instructions and uses 6.9×10^7 gas on the highest-gas case among its 39 tests. These are separate outcomes: the first binary satisfies the formal contract for valid inputs under the modeled semantics, but its proof does not establish compliance with the gas cap.

Two proved maximum-flow binaries. Two accepted Download Speed artifacts compute a maximum flow by repeatedly searching for augmenting paths. The GPT-5.6 Sol artifact has 14 functions and 2,728 instructions and uses 2.9×10^9 gas over 77 tests; a GPT-6 Astra artifact has six functions and 411 instructions and uses 4.4×10^{12}. The contract fixes the flow value, not the execution cost of the chosen augmentation strategy.