
Diffusion Models for Code Verification

Robert Joseph George

Carson Eisenach
Anima Anandkumar

Samuel Tenka
Dean Foster

Udaya Ghai

Abstract

Large language models have made candidate programs cheap to produce, but not cheap to trust. A plausible completion may compile, pass a finite test suite, and still violate its intended behavior. Program verification closes this gap by pairing an implementation with a formal specification and a machine-checked proof that the implementation satisfies it. The proof kernel provides the trust boundary, but finding a proof remains an expensive search over decompositions, completions, and repairs. Decoder speed therefore matters: within the same wall-clock budget, faster generation permits more candidates to reach the exact verifier. We study whether diffusion language models can turn parallel token generation and bidirectional repair into useful verified search. We train one 8B hybrid model on 600,000 verifier-filtered construction, local-repair, and global-repair transitions. The same weights run autoregressively, as native masked diffusion, or with self-speculative decoding. To our knowledge, this is the first diffusion language model trained for end-to-end program verification in LEAN, where success requires a kernel-checked program, specification, and proof. We further introduce global refinement, a verifier-gated action that revises an interacting proof region after repeated local failure. Our autoregressive global system verifies 56% of VERINA tasks and 66/161 (41.0%) CLEVER tasks; the Goedel-Code-Prover baseline reaches 52.4% and 34.8%, respectively. Self-speculation with global refinement reaches 59.3% and 49.1%. In a controlled 8B audit, self-speculation preserves verified accuracy while reducing mean generation latency by $4.8\times$ and neural evaluations by $5.09\times$. Native diffusion exposes a stronger speed-accuracy trade-off. These results show that diffusion is most useful for verification when its saved model calls are coupled to verifier-checked search rather than treated as raw decoding throughput.

1 Introduction

Large language models can now produce programs, patches, and tests at a scale that makes generation itself increasingly cheap. Correctness has not become cheap with it. A generated program can look idiomatic, compile successfully, and pass every supplied test while remaining wrong on an untested input or violating an unstated invariant. This gap matters most when generated code is used in infrastructure, scientific computing, security-sensitive systems, or as a component that will itself generate further software.

Program verification replaces sampled evidence with a mathematical contract. The programmer states the intended behavior as a formal specification; a proof then establishes that the implementation satisfies that specification for every input covered by the contract. A small trusted kernel checks the proof independently of the model that produced it. In our setting, LEAN elaborates source-level terms and tactics into a proof term, and its kernel checks that term against the theorem statement [de Moura et al., 2015, The mathlib Community, 2020]. Verification does not guarantee that a specification captures every real-world requirement, but it sharply separates two questions that ordinary generation conflates: what the program should do, and whether this implementation actually does it.

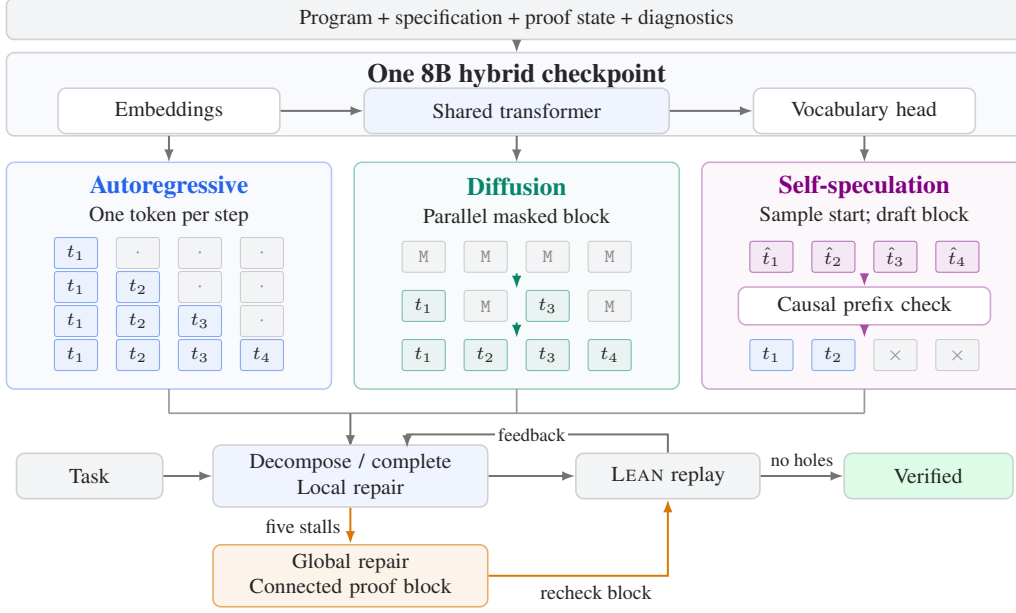


Figure 1: **One model, three decoders, two repair scopes.** AR emits tokens sequentially; diffusion fills masked positions (M); self-speculation drafts a block and checks its prefix causally. Five local stalls enable a global rewrite. Statements stay fixed, and success requires a complete LEAN proof.

Verifiable code generation asks a model to construct the implementation and the machine-checkable proof together. A nearly correct program, a proof with one remaining hole, or an artifact that fails to elaborate receives no credit. VERINA and CLEVER evaluate this setting over 189 and 161 programming tasks, respectively [Ye et al., 2026, Thakur et al., 2025]. The exact verifier is what makes generated code trustworthy, but reaching an accepted artifact requires an iterative search rather than a single completion.

Compiler feedback identifies malformed terms, unsolved goals, and type mismatches. Search uses this feedback to repair failed proof regions and try new decompositions. Under fixed compute and time limits, both generation and checking consume the budget. Faster inference is useful when it produces more diverse candidates that the verifier can check within those limits.

Diffusion language models offer two mechanisms that may help. First, masked denoising can update several positions in parallel and can condition on both sides of a region [Austin et al., 2021, Li et al., 2022, Nie et al., 2025]. Second, a hybrid checkpoint can use its non-causal pathway to draft several tokens and its causal pathway to accept the longest matching prefix. This self-speculative mode preserves greedy causal decoding while amortizing multiple token decisions over fewer model calls [Zhang et al., 2024, Fu et al., 2026]. Both mechanisms matter only if the saved model time becomes useful verifier-checked work.

Decoder choice does not address every failure. Local repair can repeatedly edit one declaration even when several proof bodies need to change together. We therefore separate the *decoder*, which determines how tokens are generated, from the *repair scope*, which determines how much of the proof plan may change. Local repair edits one hole or one diagnosed region. Global refinement is trained as a distinct action: after repeated local stalls, it may revise an interacting block of proof bodies while preserving theorem statements and surrounding code. Every proposal is still accepted only after LEAN replay (Figure 1).

Our experiments ask three questions. Can one hybrid model support AR, diffusion, and self-speculative proof generation without changing the trust boundary? Does training on local and global repair transitions improve end-to-end verification? Finally, do measured decoder savings translate into better fixed-budget search, rather than merely faster isolated completions?

Prior diffusion work studies formal theorem proving or constrained code generation [Wang et al., 2026, Zeng et al., 2025, Shao et al., 2026]. To our knowledge, ours is the first diffusion language model trained specifically for end-to-end program verification in LEAN.

We make three contributions:

- We train one 8B hybrid checkpoint on 600,000 verified decomposition, completion, and repair transitions, then evaluate the same weights with autoregressive, native-diffusion, and self-speculative decoding. This is, to our knowledge, the first diffusion model trained for end-to-end program verification in LEAN.
- We evaluate local and global refinement under the same exact verifier. On CLEVER, the 600K autoregressive global system verifies 66/161 tasks at pass@32, compared with 56/161 for Goedel-Code-Prover. On VERINA, local AR reaches 53.4% and global AR reaches 56%, compared with 52.4% for Goedel-Code-Prover.
- We measure decoder cost rather than infer it from block size. Self-speculation matches autoregressive accuracy in the 8B audit at $4.8\times$ lower mean latency, and the 3B block sweeps expose the accuracy–cost curves of native diffusion and self-speculation.

2 Hybrid Verified Search

2.1 A verification task

Consider an illustrative task: cap a natural number at a supplied limit. The program `clip` chooses the smaller value. Its specification requires the result to be at most the limit and equal the mathematical minimum; the theorem states that every input satisfies both requirements.

Program

```
def clip (n limit : Nat) : Nat :=
  if n <= limit then n else limit
```

Specification

```
def clipSpec (n limit result : Nat) : Prop :=
  result <= limit ∧ result = min n limit
```

Theorem

```
theorem clip_correct (n limit : Nat) :
  clipSpec n limit (clip n limit) := by
  exact ⟨clip_le n limit, clip_eq_min n limit⟩
```

Editable proof bodies

Local	Global
clip_le	clip_le
clip_eq_min	clip_eq_min
clip_correct	clip_correct

Shaded bodies may change

The proof uses two helpers: `clip_le` proves the bound, and `clip_eq_min` identifies the result. If `clip_le` contains the invalid tactic `exact Nat.le_refl limit`, local repair replaces that proof body with `unfold clip; split <;> omega`. After repeated stalls, global repair can rewrite both helper bodies and the final proof together. It rechecks the whole block while preserving the program, specification, and theorem statements. The shaded regions above show the difference in edit scope.

2.2 Verifier-checked transitions

Let x be the current LEAN file, d its compiler diagnostics, and U the set of unresolved declarations. The prover state is $s = (x, d, U)$. An action chooses a source region r and proposes replacement text $\hat{r} \sim p_\theta(\cdot \mid \text{ctx}(s, r))$. We accept the transition only after replay:

$$s \rightarrow s' \quad \text{iff} \quad V_a(s, x[r \leftarrow \hat{r}]) = \text{accept}. \quad (1)$$

Here V_a is the runtime admission check for the selected action. Decomposition may introduce helper holes: it requires constructive acceptance and a lower structural difficulty score. Completion can also introduce a temporary hole while removing an invalid region. Final success requires the entire artifact to pass LEAN with no open holes. Changing the decoder does not change these admission rules.

2.3 One checkpoint, three decoders

We initialize from Nemotron-Labs-Diffusion-8B, whose shared weights support a causal path and a masked non-causal path [Fu et al., 2026]. We fine-tune both paths jointly:

$$\mathcal{L} = \mathcal{L}_{\text{AR}} + 0.3 \mathcal{L}_{\text{diff}}, \quad (2)$$

where $\mathcal{L}_{\text{AR}}$ is next-token cross-entropy and $\mathcal{L}_{\text{diff}}$ predicts randomly masked tokens from bidirectional context. All decoder comparisons below use these same weights.

Autoregressive generation The causal path emits one token at a time with a KV cache. It is the reference decoder for decomposition, proof construction, and repair.

Native diffusion The non-causal path denoises an active block of B masked positions. At each step it commits selected tokens and leaves the remaining positions masked. The commitment schedule uses the model’s confidence. Larger blocks reduce serial neural evaluations but can commit mutually inconsistent positions before their left context is settled.

Self-speculation The non-causal path drafts a block, and the causal path accepts the longest prefix that agrees with its greedy token decisions. Rejected suffix tokens are discarded. At temperature zero, this uses diffusion for lookahead while retaining greedy causal decoding as the acceptance rule.

Stochastic search Repeated greedy calls on an unchanged proof state provide no new candidates. The default NVIDIA diffusion and LinearSpec serving paths are greedy; we add stochastic samplers so that temperature, nucleus sampling, and request seeds control diversity in both paths. NVIDIA’s native Python generation interface also exposes temperature; our extension concerns the serving algorithms.

For diffusion, we separate *where to commit* from *what to commit*. At masked position i , untempered logits z_i give confidence $c_i = \max_v \text{softmax}(z_i)_v$. Positions above threshold τ are committed; if none qualifies, the most confident position advances. At each selected position we sample a token from $q_i(v) \propto \exp(z_{iv}/T)$, restricted to the top 64 non-mask tokens and then to the requested nucleus. Thus temperature changes token choices while the confidence rule controls which positions become visible. Later denoising steps condition on the sampled commitments.

For self-speculation, we sample the causal token that starts each block using temperature and nucleus sampling. The diffusion draft and causal prefix check within the block remain greedy. After accepting the matching prefix, we sample the next block’s starting token from the causal logits at the boundary. This creates diverse block continuations with the same shared weights. At $T = 0$ both extensions recover their greedy paths. At $T > 0$, the self-speculative sampler defines a hybrid search policy; it does not preserve the distribution of token-by-token stochastic AR decoding.

2.4 Local and global refinement

The local loop follows Goedel-Code-Prover: decompose the task, fill proof holes, compile, and repair the diagnosed region [Li et al., 2026]. A step is a stall when it closes no hole, shrinks no unresolved set, and reaches no better compilable checkpoint. After five consecutive stalls, the controller may generate up to eight replacements for one connected declaration block. Each proposal may receive two compiler-guided repair calls. At most two global rewrites are accepted in one trajectory, and an accepted rewrite resets the stall counter. Runtime rewrites preserve declaration headers, theorem statements, declaration order, and surrounding code, and cannot add placeholders.

This action is smaller than a separate planning agent. Goedel-Architect, for example, generates an explicit dependency blueprint, assigns its lemmas to provers, and revises the blueprint from their failures [Chung et al., 2026]. Our controller rewrites the source region directly with the same trained checkpoint used for local completion.

2.5 Training data

The original corpus contains 432,000 locally generated decomposition and completion transitions. GPT-5.6 Sol produces decompositions; completion mixes GPT-5.6 Luna and Goedel-Prover-V2 [Lin

Table 1: **Training mixture.** Repair rows enter the corpus only after scope-specific LEAN replay succeeds.

Component	Rows	Share	Admission rule
Decomposition	281,000	46.8%	constructive transition accepted
Completion	151,000	25.2%	accepted step in successful trajectory
Local repair	104,000	17.3%	restored proof passes LEAN
Global repair	64,000	10.7%	restored sketch passes QuickCheck
Total	600,000	100%	parent-disjoint split

et al., 2025]. Completion rows are accepted edits from trajectories that eventually verify without holes. The 600K mixture adds 104,000 local and 64,000 global repair transitions (Table 1). Sol constructs corruptions of accepted proofs and decomposition sketches, with repair-scope reasoning; the supervised target restores the accepted artifact. Local corruptions must fail LEAN while their restorations pass without placeholders. Global corruptions must fail official QuickCheck with a counterexample while their restored sketches pass. These sketches can still contain helper holes; their admission is distinct from a completed proof.

The final split has 589,160 training and 10,840 validation rows, partitioned by parent problem. We train for three epochs on 64 A100 GPUs with sequence length 10,240, packed global batch 128, peak learning rate 10^{-5} , cosine decay, and 0.1 warmup.

2.6 Verifier-rewarded post-training

We also explored RLVR from the 600K code-verification checkpoint, combining AR GRPO with diffusion TraceRL. AR updates score sampled token sequences; diffusion updates replay the sampled denoising actions, including the positions and tokens committed at each step. Rewards come from the code-verification runtime’s decomposition checks, and advantages are computed within prompt-decoder groups. Completion replay regularizes both paths. The final ten-update pilot used

$$\mathcal{L}_{\text{RLVR}} = \mathcal{L}_{\text{AR-GRPO}} + 0.5 \mathcal{L}_{\text{diff-TraceRL}} + 0.08(\mathcal{L}_{\text{AR}} + 0.3 \mathcal{L}_{\text{diff}}) + \mathcal{R}_{\text{KL}}. \quad (3)$$

Here $\mathcal{R}_{\text{KL}}$ regularizes against both the rollout policy and a frozen SFT checkpoint, with coefficient 0.01 for each term. Self-speculation remains a decoding composition and has no separate RL loss.

3 Evaluation

Benchmarks VERINA contains 189 manually curated LEAN programming tasks with implementations, specifications, proofs, and tests; its original evaluation found proof generation substantially harder than code or specification generation [Ye et al., 2026]. CLEVER contains 161 curated end-to-end problems and checks generated artifacts with the LEAN type checker [Thakur et al., 2025]. A task counts only when the final file elaborates with no open holes.

Search budget A trajectory includes decomposition, completion, compiler feedback, and repair. Each receives up to 128 decomposition iterations, 128 proof epochs, five local fixes per epoch, a 10,240-token context, and three hours of wall time. `pass@32` is the union of 32 independent complete trajectories, each with its own decomposition; it is not one decomposition followed by 32 completion samples. The Goedel-Code-Prover rows use its released checkpoint and upstream search program with the same number of independent attempts and the same time limits.

Decoder audits We measure model latency and neural function evaluations (NFE) separately from compiler time. The 8B audit aligns AR, native diffusion at block size 1, and self-speculation on all 244 MINIF2F test theorems. The block-size study uses the same 3B checkpoint, prompt format, 244 tasks, 4,096-token cap, and temperature zero across both decoder families. Native diffusion uses confidence threshold 0.9 at $B \in \{8, 16, 32\}$; self-speculation uses greedy causal acceptance at the same block sizes.

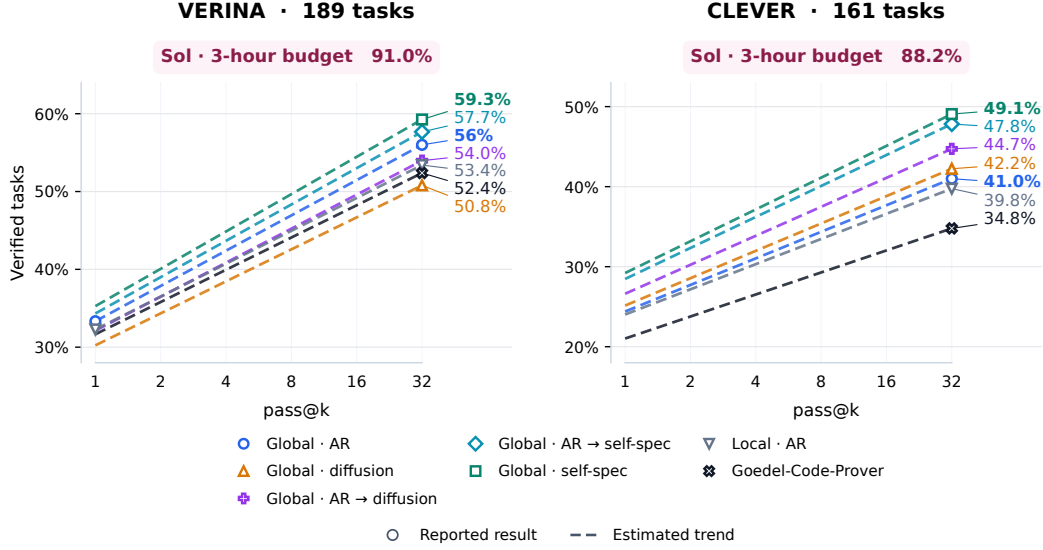


Figure 2: **Code verification from pass@1 to pass@32.** Markers show reported results; dashed curves estimate unreported budgets. We interpolate in $\log_2 k$ and transfer the VERINA AR pass@1/pass@32 ratios when a pass@1 result is unavailable, including all CLEVER curves (Appendix A). The panels use separate, tightened vertical ranges. GPT-5.6 Sol appears above each panel as a three-hour reference.

Table 2: **Stage-specific decoding: reported scores and estimates.** Decomposition and completion modes are listed separately. Scores are percentages over 189 VERINA and 161 CLEVER tasks.

System	Refinement	Generation mode		Score type	Verified (%)	
		Decomposition	Completion		VERINA	CLEVER
8B systems: pass@32						
Goedel-Code-Prover	Local	AR	AR	Reported	52.4	34.8
Ours (432K)	Local	AR	AR	Verina estimated	47.2	33.5
Ours (600K)	Local	AR	AR	Reported	53.4	39.8
Ours (600K)	Global	AR	AR	Reported	56	41.0
		AR	Diffusion	Reported	54.0	44.7
		AR	Self-spec	Reported	57.7	47.8
		Diffusion	AR	Estimated	52.8	38.5
		Diffusion	Diffusion	Reported	50.8	42.2
		Diffusion	Self-spec	Estimated	54.5	45.3
		Self-spec	AR	Estimated	57.6	42.2
		Self-spec	Diffusion	Estimated	55.6	46.0
		Self-spec	Self-spec	Reported	59.3	49.1
GPT-5.6 Sol	Tool use	3-hour budget		Reported	91.0	88.2

4 Results

4.1 VERINA and CLEVER search

Table 2 separates the decoder used to construct a decomposition from the decoder used to complete its proof obligations. Thus AR → diffusion in Figure 2 uses AR for decomposition and diffusion for completion; AR → self-spec changes the completion stage to self-speculation.

Table 3: **Controlled 8B direct-proof audit on 244 MINIF2F theorems.** Latency and NFE measure model generation; compiler time is excluded. The SGLang throughput column is a separate serving control.

Decoder	Verified	Mean sec.	Mean NFE	Output tok/s
AR	114/244	48.62	1412.50	560
Native diffusion, $B=1$	114/244	50.03	1412.50	–
Self-speculation	115/244	10.18	277.43	3904

On VERINA, our 600K local AR system reaches 53.4%, compared with 52.4% for Goedel-Code-Prover; global AR reaches 56% (Figure 2). On CLEVER, the 432K local AR checkpoint verifies 54/161 (33.5%); the 600K local AR system reaches 39.8% and global AR reaches 66/161 (41.0%). The latter verifies ten more tasks than Goedel-Code-Prover, a gain of 6.2 percentage points.

The separate pass@1 runs do not show a gain from adding repair data alone: the 432K local checkpoint and 600K global system both solve 63 VERINA tasks, while the 600K local system solves 61. The global controller triggered 852 times and accepted 68 rewrites (8.0%). Seven tasks were solved only by the global-enabled pass@1 run, but only one of those traces contains an accepted global rewrite. We therefore report the two-task end-to-end difference without treating it as a clean causal effect.

Among the 8B systems, self-speculation with global refinement achieves the highest accuracy: 112/189 (59.3%) on VERINA and 79/161 (49.1%) on CLEVER (Table 2). Native diffusion with global refinement verifies 96/189 (50.8%) and 68/161 (42.2%). Relative to global AR, native diffusion loses accuracy on VERINA but gains on CLEVER. Decoder choice therefore affects verified accuracy as well as generation cost; the direct latency measurements below examine the latter separately.

RLVR pilot The code-verification pilot completed ten optimizer updates and saved the resulting checkpoint. Its rollout reward fluctuated from 0.0112 to 0.6083 and ended at 0.2849, with nonzero learning signals for both decoder paths. These are training rewards on sampled decomposition states, not benchmark pass rates. The retained evaluation records do not establish a paired post-RLVR gain on VERINA or CLEVER; the main results above use the SFT checkpoint.

4.2 Comparison with GPT-5.6 Sol

GPT-5.6 Sol verifies 172/189 (91.0%) VERINA tasks and 142/161 (88.2%) CLEVER tasks with a three-hour budget (Figure 2 and Table 2). It remains substantially ahead of our strongest 8B system, whose self-speculative global search reaches 59.3% and 49.1% at pass@32. This comparison places the compact model’s results against a stronger tool-using system; the model scale and trajectory budgets differ.

4.3 Decoder cost under exact checking

Self-speculation verifies one more theorem than AR while cutting mean latency from 48.62 to 10.18 seconds and mean NFE from 1412.50 to 277.43 (Table 3). The serving control increases output throughput from 560 to 3904 tokens/s. These measurements establish that the hybrid path can accelerate the same causal policy; they do not by themselves guarantee a higher end-to-end pass@ k .

Figure 3 reports the requested block-size costs rather than nominal $1/B$ estimates. Native diffusion falls from 92 to 75 verified tasks as B grows from 8 to 32, while mean latency falls from 11.77 to 6.22 seconds. Self-speculation moves from 90 to 89 verified tasks over the same range and falls from 9.34 to 4.18 seconds. Its mean NFE at $B=32$ is 142.75, compared with 212.77 for native diffusion.

5 Related Work

Verified code and hierarchical proof search VERINA evaluates code, specification, and proof generation over manually curated LEAN tasks [Ye et al., 2026]. CLEVER removes test-case supervision and implementation-leaking specifications from its curated end-to-end tasks [Thakur et al., 2025]. Goedel-Code-Prover trains one policy for decomposition and completion and uses hierarchical proof

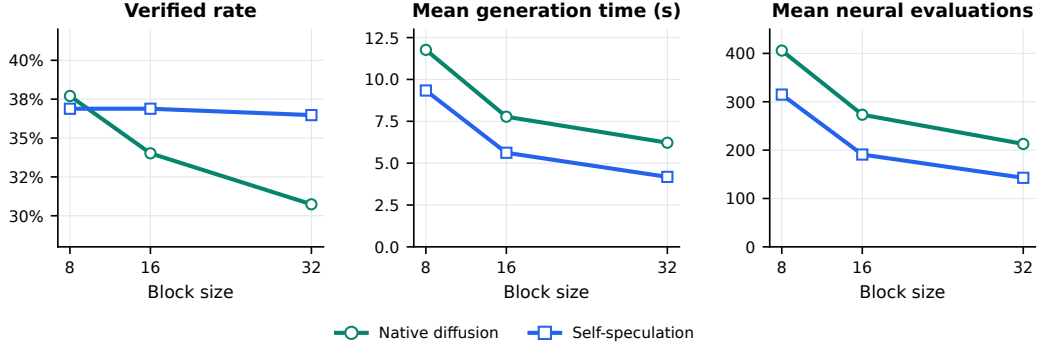


Figure 3: **Measured block-size frontier.** Both families use the same 3B checkpoint, 244 MINIF2F tasks, deterministic decoding, and the same token cap. Native diffusion uses threshold 0.9. Self-speculation retains nearly constant verified accuracy while its latency and NFE fall with block size; native diffusion is also cheaper at larger blocks but loses more accuracy.

search [Li et al., 2026]. Goedel-Architect instead builds and revises an explicit dependency blueprint around tool-equipped prover instances [Chung et al., 2026]. Our global action is a verifier-gated source rewrite performed by the same compact checkpoint used for completion.

Diffusion and self-speculative decoding Discrete diffusion replaces left-to-right factorization with iterative denoising [Austin et al., 2021, Li et al., 2022, Nie et al., 2025]. Nemotron-Labs-Diffusion trains one model for AR, diffusion, and self-speculation [Fu et al., 2026]; Draft & Verify establishes the broader lossless self-speculative pattern [Zhang et al., 2024]. Diffusion-Proof trains diffusion models for whole-proof generation and local proof correction [Wang et al., 2026]. TreeDiff uses syntax-tree structure during code denoising [Zeng et al., 2025]. CDC and Lookahead-then-Verify intervene inside a diffusion trajectory to enforce program or grammar constraints [Shao et al., 2026, Kou et al., 2026]. Our verifier acts between source-edit transitions and evaluates exact LEAN obligations.

6 Limitations

Faster inference still competes for finite GPU time, memory, and verification capacity. Our pass@32 search allows up to 96 hours of combined search time per task; parallel execution reduces elapsed time but still consumes those resources. The decoder measurements exclude compiler time, so they do not establish how many additional programs can be verified under a fixed total compute budget. A full comparison must account for generation, checking, and unsuccessful repair attempts together.

The estimated budget curves and unmeasured decoder combinations illustrate possible outcomes; they cannot establish gains at those settings. Finally, a checked proof establishes that the program satisfies its formal specification. It does not establish that the specification captures every intended requirement.

7 Conclusion

Our results show the promise of diffusion and self-speculation as alternatives to autoregressive generation for code verification. Using one hybrid checkpoint, self-speculation preserves direct-proof accuracy while reducing generation time by $4.8\times$ in the controlled 8B audit. Under the same search budgets, self-speculation with global refinement reaches 59.3% on VERINA and 49.1% on CLEVER, compared with 56% and 41.0% for global AR. Faster decoding leaves more time for new candidates and compiler-guided repair. Diffusion also uses context on both sides of a masked region, making existing code and proof structure useful for local repairs and broader global refinement. As inference gets faster, the objective is to maximize fully verified outputs within a fixed compute budget. These results show the value of combining fast generation, reuse of surrounding proof context, and local and global refinement under the same exact verifier.

References

- Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, 2021.
- Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, et al. Goedel-Architect: Streamlining formal theorem proving with blueprint generation and refinement. *arXiv preprint arXiv:2606.06468*, 2026. URL <https://arxiv.org/abs/2606.06468>.
- Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.
- Yonggan Fu, Lexington Whalen, Abhinav Garg, Chengyue Wu, Maksim Khadkevich, Nicolai Oswald, Enze Xie, Daniel Egert, Sharath Turuvekere Sreenivas, Shizhe Diao, Chenhan Yu, Ye Ye, Weijia Chen, Sajad Norouzi, Shiyi Lan, Ligeng Zhu, Jin Wang, Jindong Jiang, Morteza Mardani, Mehran Maghousi, Song Han, Ante Jukić, Nima Tajbakhsh, Jan Kautz, and Pavlo Molchanov. Nemotron-Labs-Diffusion: A tri-mode language model unifying autoregressive, diffusion, and self-speculation decoding. *arXiv preprint arXiv:2607.05722*, 2026. URL <https://arxiv.org/abs/2607.05722>.
- Chengzhong Kou, Lingyue Shen, Chenyu Zhang, Mengyun Liu, Huanyu Jia, and Quanyan Zhu. Lookahead-then-verify: Reliable constrained decoding for diffusion LLMs under context-free grammars. *arXiv preprint arXiv:2602.00612*, 2026. URL <https://arxiv.org/abs/2602.00612>.
- Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM improves controllable text generation. *arXiv preprint arXiv:2205.14217*, 2022.
- Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical proof search for open state-of-the-art code verification. *arXiv preprint arXiv:2603.19329*, 2026. URL <https://arxiv.org/abs/2603.19329>.
- Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025. URL <https://arxiv.org/abs/2508.03613>.
- Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large Language Diffusion Models. In *Advances in Neural Information Processing Systems*, 2025. URL <https://openreview.net/forum?id=KnqiC0znVF>.
- Lize Shao, Michael Cardei, Zichen Xie, Ferdinando Fioretto, and Wenxi Wang. Constrained code generation with discrete diffusion. *arXiv preprint arXiv:2605.16829*, 2026. URL <https://arxiv.org/abs/2605.16829>.
- Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems*, 2025. URL <https://arxiv.org/abs/2505.13938>. arXiv:2505.13938.
- The mathlib Community. The Lean mathematical library. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 367–381, 2020.
- Ruida Wang, Rui Pan, Pengcheng Wang, Shizhe Diao, and Tong Zhang. Diffusion-Proof: Recipe for formal theorem proving beyond auto-regressive generation. *arXiv preprint arXiv:2606.19315*, 2026. URL <https://arxiv.org/abs/2606.19315>.

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://arxiv.org/abs/2505.23135>. arXiv:2505.23135.

Yiming Zeng, Jinghan Cao, Zexin Li, Yiming Chen, Tao Ren, Zhuochun Li, Dawei Xiang, Xidong Wu, Shangqian Gao, and Tingting Yu. TreeDiff: AST-guided code generation with diffusion LLMs. *arXiv preprint arXiv:2508.01473*, 2025. URL <https://arxiv.org/abs/2508.01473>.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. URL <https://aclanthology.org/2024.acl-long.607/>.

A Reproducibility Details

Training The 432K and 600K checkpoints use three epochs, sequence length 10,240, packed global batch 128, 64 A100 GPUs, learning rate 10^{-5} , cosine decay, 0.1 warmup, and Equation 2. The 600K checkpoint is exported with the trained 10,240-token context.

Serving AR and self-speculative decoding use a pinned SGLang build. Native diffusion uses the same checkpoint’s diffusion path. Runtime launches pass a greedy token-parity smoke test against the reference implementation before evaluation.

Sampler implementation Our `StochasticLowConfidence` configuration uses block size 32, maximum 32 denoising steps, confidence threshold 0.9, and a top-64 token support. The `LinearSpec` extension enables `stochastic_seed`; temperature and nucleus cutoff apply to causal block-start sampling, while drafting and prefix acceptance use `argmax`. Both implementations exclude the mask token from sampled output and derive step-specific random streams from the request seed. These are distinct from the native Python diagnostic that samples both draft and verification tokens. Sampler identity, temperature, nucleus cutoff, block size, threshold, and seed therefore belong to a run’s decoding configuration. The temperature-zero cost audits use the greedy paths; their latency results do not measure the stochastic samplers.

Global controller The controller triggers after five consecutive stalls, permits two accepted rewrites per trajectory, generates at most eight proposals per trigger, and allows two compiler-repair calls per proposal.

Table 2 estimates For unreported global decoder pairs, we assume additive decomposition and completion effects on the percentage-point scale. With A , D , and S denoting AR, diffusion, and self-speculation, respectively, we calculate $\hat{p}_{dc} = p_{Ac} + p_{dd} - p_{Ad}$ for $d \in \{D, S\}$, where p_{dc} is the score for decomposition mode d and completion mode c . The 432K local AR estimate on VERINA transfers the corresponding CLEVER data-size gap: $53.4 - 100(64 - 54)/161 = 47.2\%$ after rounding. These assumptions are untested. The estimates do not enter Figure 2 or the empirical gains discussed in the text.

Estimated budget curves Figure 2 uses reported `pass@32` endpoints and, where available, the separate VERINA AR `pass@1` results. For an unreported endpoint, we set $\hat{p}_1 = rp_{32}$, with $r = (100 \cdot 63/189)/56$ for global variants and $r = (100 \cdot 61/189)/53.4$ for local variants, including Goedel-Code-Prover. We then draw $\hat{p}_k = \hat{p}_1 + (p_{32} - \hat{p}_1) \log_2(k)/5$ for $1 \leq k \leq 32$, replacing $\hat{p}_1$ with a reported value where available. Transfer across benchmarks and decoders is a visualization assumption; the curves do not establish measured performance at the missing budgets. Reported markers remain at their original coordinates. The saved 32-trajectory aggregates stop remaining attempts after success, so their retained run indices do not recover complete smaller-budget evaluations.

RLVR pilot The ten-update code-verification pilot starts from the 600K SFT checkpoint at iteration 3177 and runs on eight A100 GPUs with tensor parallelism eight. Its prompt pool contains 32 decomposition states. Each update samples four AR and four diffusion rollouts, using temperatures

0.9 and 0.7, respectively. Diffusion uses block length 16, sixteen denoising steps per block, and confidence threshold 0.5. The maximum context is 10,240 tokens, the generation cap is 8,192 tokens, and the peak learning rate is 5×10^{-6} . Completion replay draws from a pool of 128 examples; checkpoints are saved every two updates. Equation 3 gives the experimental objective.

Compute Full pass@32 runs used up to 160 GPUs. Training used 64 A100 GPUs. The aligned 8B decoder audit used eight A100 GPUs.

B Prompt and Training Formats

All training rows use the chat serialization `<|system|> system <|user|> prompt <|assistant|> response`. The AR loss is applied to assistant tokens. For the diffusion objective, assistant tokens are randomly masked and reconstructed with bidirectional attention; the prompt remains visible context. The following templates are the operational prompts, with brace-delimited fields replaced per task.

B.1 Decomposition

The template below preserves the operative upstream wording and output contract; the long worked list example is omitted because it does not vary by task.

```
You are an expert in Lean 4 theorem proving. Your task is to decompose a
theorem into smaller, provable lemmas.

Think step by step. Before writing code, first analyze the theorem structure
and explain your decomposition strategy in a <reasoning> block.

Task: Decompose the theorem {theorem_name} into helper lemmas, then prove the
original theorem by combining them.

Strategies:
1. Structural decomposition: split conjunctions, disjunctions, existentials,
   and universal goals.
2. Induction decomposition: extract base and inductive cases as lemmas.
3. Rewrite chain: break A = D into A = B, B = C, and C = D.
4. Case analysis: make each branch a separate lemma.

Output requirements:
1. Put analysis in <reasoning>...</reasoning> before code.
2. Helper lemmas use 'lemma' and may have body 'by sorry'.
3. The main theorem uses 'theorem' and must contain no 'sorry'.
4. Put all code in one Lean code block.

Problem to decompose:
{formal_problem}
```

B.2 Completion and Compiler Repair

```
SYSTEM:
Lean 4 proof assistant.

Task: Complete the proof by filling in the 'sorry' placeholder with valid
tactics. The SEARCH block should match the original code, including
whitespace. First output reasoning in a 'reasoning' block, followed by exact
SEARCH/REPLACE diff blocks.

USER:
Complete the current proof goal.
Strategy: Use automation tactics to prove the goal directly.
Try, in order: grind, aesop, simp +decide, omega, decide, by_cases, induction.

{code}

Goal (Line {line}):
{goal}

Output diff blocks only.
```

```
FIX SYSTEM:
Lean 4 proof assistant.

Task: Correct the compilation errors. Avoid using 'sorry'.
First output reasoning in a 'reasoning' block, followed by exact
SEARCH/REPLACE diff blocks.
```

B.3 Repair-Scope Supervision

Every repair example begins with the following visible routing contract. This makes “local” and “global” learned action labels rather than post-hoc names assigned only by the controller.

```
Before the requested Goedel Code Prover action, emit three XML lines:
<repair_scope>none|local|global</repair_scope>,
<repair_required>true|false</repair_required>, and
<repair_assessment>brief evidence</repair_assessment>.
Then emit the originally requested decomposition or SEARCH/REPLACE action
verbatim. Use local only for one bounded declaration or connected proof
region. Use global only when the decomposition strategy or multiple
interacting helper contracts must change.
```

Local-repair rows use the following task and target format:

```
SYSTEM:
Lean 4 proof assistant.

Task: Diagnose whether the current proof state needs local or global repair,
then correct the compilation error without using placeholders.

Output the required repair-scope XML lines, followed by one reasoning block
and one exact SEARCH/REPLACE diff block.

USER:
Correct the compilation error below. Keep all working code and the target
theorem statement unchanged. Do not use ‘sorry’.

{corrupted_full_lean_file}

Lean diagnostics:
{diagnostics}

ASSISTANT PREFIX:
<repair_scope>local</repair_scope>
<repair_required>true</repair_required>
<repair_assessment>{bounded-region evidence}</repair_assessment>
```

The response restores one corrupted span. A row is admitted only when the corrupted file is rejected, the restored file is accepted without placeholders, and the SEARCH span occurs uniquely in the prompt.

B.4 Global-Repair Training

```
SYSTEM:
Lean 4 theorem decomposition assistant.

Task: Diagnose whether the current decomposition needs local or global repair,
then return a coherent decomposition whose helper lemmas may contain ‘sorry’
but whose main theorem is complete.

Output the required repair-scope XML lines, followed by one reasoning block
and one Lean code block.

USER:
The current decomposition for {target_theorem} failed official Goedel
QuickCheck. Diagnose the repair scope and replace the decomposition strategy.
Helper lemmas may contain ‘sorry’; the main theorem must be complete.

Original decomposition task:
{source_prompt}

Failed decomposition:
{failed_decomposition}

QuickCheck feedback:
{quickcheck_error}

ASSISTANT PREFIX:
<repair_scope>global</repair_scope>
<repair_required>true</repair_required>
<repair_assessment>{interacting-contract evidence}</repair_assessment>
```

Global rows are labeled `repair_action=replace_global_decomposition`; local rows are labeled `repair_action=repair_local`. A global row enters the mixture only when official QuickCheck rejects the corrupted plan and the replacement plan passes the corresponding replay checks.

B.5 Runtime Global Refinement

```

Lean 4 global proof-plan refiner.

Repeated bounded proof repairs have failed. Inspect the complete EVOLVE-BLOCK
and its failure history, then make the smallest compiler-checkable change that
unblocks the proof plan.

Requirements:
- Preserve all code outside EVOLVE-BLOCK.
- Reason globally about dependencies, but normally edit only one or two proof
  bodies implicated by compiler feedback.
- Do not introduce additional 'sorry' placeholders.
- Never delete, rename, or reorder an existing declaration.
- Preserve every declaration header and theorem statement exactly.
- Return one or two exact SEARCH/REPLACE patches in one diff code fence.
- Each SEARCH must be unique text wholly inside EVOLVE-BLOCK.

The runtime applies the patches transactionally and checks the result with
Lean.

USER:
Revise the interacting proof plan after repeated local failures.

Current Lean source:
{code}

Current selected goal:
{goal}

Recent failed-repair history:
{history}

Return only the smallest exact SEARCH/REPLACE patch set that compiles with any
remaining pre-existing 'sorry' placeholders.

```

The controller enables this prompt after five consecutive local stalls, samples at most eight proposals, permits two compiler-repair calls per proposal, and accepts no more than two global rewrites per trajectory. An accepted rewrite resets the stall counter; rejected proposals leave the current file unchanged.

C Use of Language Models

GPT-5.6 Sol generates decompositions and synthetic local/global corruptions. Completion uses a mixed pool including GPT-5.6 Luna and Goedel-Prover-V2; retained early-wave traces also identify Claude Opus 4.8. The packaged manifest does not record exact per-teacher proportions. The repair target restores an accepted source proof or sketch, with `LEAN` or official QuickCheck determining admission. The trained 8B checkpoint supplies proposals during the main evaluation. GPT-5.6 Sol additionally appears as the separate scale reference.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [\[Yes\]](#)

Justification: The abstract and Section 1 state the benchmark scope, verified accuracy, decoder-cost results, and the distinction between the published Goedel claim and the matched rerun.

Guidelines:

- The answer [\[N/A\]](#) means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [\[No\]](#) or [\[N/A\]](#) answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [\[Yes\]](#)

Justification: Section 6 discusses benchmark scope, the compute intensity of pass@32, the matched-baseline gap, verifier cost, and specification risk.

Guidelines:

- The answer [\[N/A\]](#) means that the paper has no limitation while the answer [\[No\]](#) means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper is empirical and does not claim a new theorem or formal proof about the learning algorithm.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Sections 2–4 and Appendix B specify the model, data mixture, training objective, decoding modes, refinement controller, verifier, budgets, and aggregation rules.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
 - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
 - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
 - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
 - (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: An anonymized artifact is not attached to this initial draft. Appendix B records the information needed to package the training and evaluation code for submission.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Sections 2–4 report the data splits, optimizer schedule, sequence length, batch size, loss weighting, inference settings, and verification rules.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: The benchmark evaluates a fixed finite task set and reports exact verified counts. Section 6 limits the claim to the matched executable baseline and does not attach unsupported confidence intervals.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Section 2 reports training hardware, Sections 3–4 report inference budgets and cost, and Appendix B records GPU counts.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: The work evaluates program-generation methods on public formal-verification benchmarks and follows the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Section 6 distinguishes formal acceptance from specification adequacy and states that benchmark success is not deployment validation.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release a new general-purpose model or a scraped human-data corpus; generated Lean terms are accepted only after kernel checking.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Sections 1–3 cite the model, prover, Lean, mathlib, Verina, and Clever sources. Their version and license files will be preserved in the anonymized artifact.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset’s creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [\[Yes\]](#)

Justification: Section 2 and Appendix B document the 600K training mixture, hybrid checkpoint, and refinement controller; release metadata will accompany the artifact.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [\[N/A\]](#)

Justification: The work does not involve crowdsourcing or human-subject research.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [\[N/A\]](#)

Justification: The work does not involve crowdsourcing or human-subject research.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [\[Yes\]](#)

Justification: Appendix C distinguishes model-generated proof proposals used in the experiments from language-model assistance used to prepare the manuscript.

Guidelines:

- The answer [\[N/A\]](#) means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.