VeriSoftBench: Repository-Scale Formal Verification Benchmarks for Lean

Yutong Xin[♠*] **Jocelyn Qiaochu Chen**[♢♣♡*] **Greg Durrett**[♢] **Işil Dillig**[♠]

[♠]The University of Texas at Austin [♢]New York University
[♣]University of Alberta [♡]Alberta Machine Intelligence Institute (Amii)
maxryeery@utexas.edu jocelyn.chen@ualberta.ca

arXiv:2602.18307v2 [cs.SE] 22 Sep 2026

Abstract

Large language models have achieved striking results in interactive theorem proving, particularly in Lean. However, most benchmarks for LLM-based proof automation are drawn from mathematics in the Mathlib ecosystem, whereas proofs in software verification are developed inside definition-rich codebases with substantial project-specific libraries. We introduce VeriSoftBench, a benchmark of 500 Lean 4 proof obligations drawn from open-source formal-methods developments and packaged to preserve realistic repository context and cross-file dependencies. Our evaluation of frontier LLMs and specialized provers yields three observations. First, provers tuned for Mathlib-style mathematics transfer poorly to this repository-centric setting. Second, success is strongly correlated with transitive repository dependence: tasks whose proofs draw on large, multi-hop dependency closures are less likely to be solved. Third, providing curated context restricted to a proof's dependency closure improves performance relative to exposing the full repository, but nevertheless leaves substantial room for improvement. Our benchmark and evaluation suite are released at https://github.com/utopia-group/VeriSoftBench.

1 Introduction

Interactive theorem provers (ITPs) such as Lean (Moura & Ullrich, 2021), Rocq/Coq (The Coq Development Team, 2008), and Isabelle (Wenzel et al., 2008) provide machine-checked guarantees, but this rigor comes with a well-known cost: even routine mathematical arguments require low-level, often tedious formalization, making mechanized proof development extremely labor-intensive. To reduce this burden, decades of work have pursued automation within ITP workflows, for example, using domain-specific tactic libraries (Gladshtein et al., 2024; Rossel, 2024) or integration with external solvers (Mohamed et al., 2025; Paulson & Blanchette, 2015; Jiang et al., 2022). More recently, large language models (LLMs) have enabled a new class of *neural* proof automation methods (Lin et al., 2025; Ren et al., 2025; Varambally et al., 2025), and high-quality benchmarks have become crucial for measuring progress in the proof settings where automation is ultimately needed.

Most widely used benchmarks for LLM-based proof automation are drawn from *mathematics* and operate in a shared ecosystem centered around Mathlib. Suites such as MiniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), LeanDojo (Yang et al., 2023), and PutnamBench (Tsoukalas et al., 2024) have been instrumental in driving rapid improvements. In these settings, proof goals are typically stated over well-known mathematical structures that are already formalized in shared libraries, so the prover can rely heavily on pre-existing definitions and lemmas. Formal verification, by contrast, presents a structurally different landscape: projects that verify compilers, protocol implementations, or language runtimes define bespoke syntax, semantics, and domain abstractions with no counterpart in shared libraries. Consequently, proof obligations depend on deep chains of project-specific defi-

[*]Equal contribution.

nitions distributed across multiple files (Figure 1). This raises a key question: *How well do current provers generalize when proofs are grounded in large, project-specific libraries rather than a shared ecosystem?*

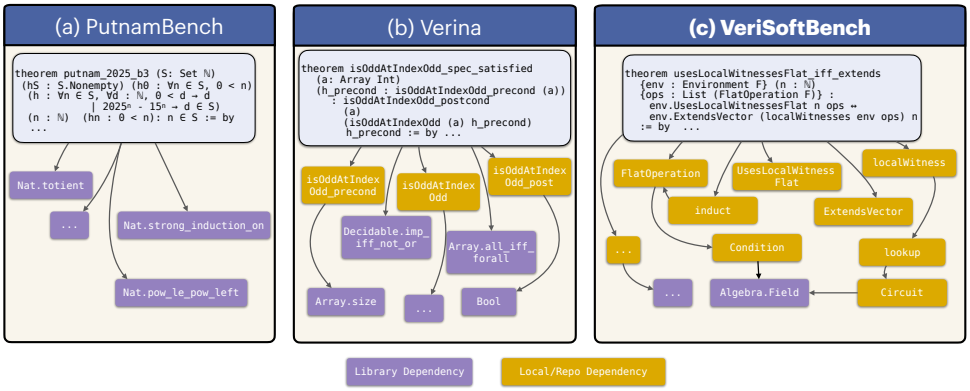


Figure 1: Contextual dependencies comparison between (a) mathematical benchmark proofs, (b) lightweight verification tasks, and (c) repository-scale verification. PutnamBench proofs rely almost entirely on library (Mathlib) dependencies (purple). Verina introduces a small number of project-specific definitions (yellow) but remains largely library-driven. In contrast, proofs from a real-world ZK circuit DSL repository (Verified-zkEVM, 2025) depend heavily on interconnected local abstractions, forming a dense repository-level dependency graph. Purple denotes library dependencies; yellow denotes local/repository dependencies.

This question is particularly pressing given how neural provers have evolved. Earlier tactic-prediction systems (Yang & Deng, 2019; First et al., 2020; Lample et al., 2022; Polu & Sutskever, 2020) interact with the proof assistant step by step, with the kernel managing the environment visible at each goal. Recent LLM-based provers (Lin et al., 2025; Ren et al., 2025) instead consume a prompt containing the proof obligation together with repository context and generate a complete proof. In this paradigm, *what context to provide*—and how the prover reasons over it—becomes a central challenge. However, Lean 4, where much of recent prover development is concentrated, lacks a benchmark that targets this challenge in the verification setting.

This paper introduces VeriSoftBench, a benchmark of 500 Lean 4 proof obligations drawn from open-source formal-methods developments and packaged to preserve cross-file dependency structure. By preserving the original project environments, VeriSoftBench enables controlled evaluation of proof automation over repository-scale software-verification developments. To make the role of repository context explicit and measurable, we evaluate provers under two complementary regimes. The *curated* regime is an intentionally oracle-like diagnostic condition, constructed using ground-truth proof dependencies to isolate context utilization from premise selection. The *full-repository* regime is the more realistic deployment condition, in which the prover must identify useful declarations within the codebase.

Our experiments yield three main findings. First, provers that perform well on Mathlib-centric mathematics tasks transfer poorly to this repository-grounded setting. Second, performance degrades as *transitive* dependency structure increases, such as when discharging a goal requires reasoning over a large, multi-hop closure of project-local definitions and lemmas. Third, curated context improves over full-repository context, but substantial room for improvement remains even when the relevant context is provided. This highlights that context retrieval is necessary but not sufficient for successful proof automation.

In summary, this paper makes the following contributions:

Proof domain	Benchmark	Lang.	Proj. defs	Multi-module	Repo retr.	GT proofs
Mathematics	MiniF2F	Lean	✗	✗	✗	✗
	ProofNet	Lean	✗	✗	✗	✗
	LeanDojo	Lean	✓	✓	✓	✓
	PutnamBench	Multi.	✗	✗	✗	✗
Veri-coding	FVAPPS	Lean	✗	✗	✗	✗
	CLEVER	Lean	✗	✗	✗	✗
	Verina	Lean	✓	✗	✗	◐
Verification	MiniCodeProps	Lean	✓	✗	✗	✗
PL & Verif.	**VeriSoftBench (Ours)**	**Lean**	✓	✓	✓	✓

Table 1: Comparison of theorem proving benchmarks. LeanDojo's scope is confined to Mathlib; VeriSoftBench features 23 diverse repositories.

- We introduce VeriSoftBench, a benchmark of 500 Lean 4 proof obligations drawn from open-source formal-methods repositories and packaged to preserve repository context and cross-file dependencies, providing a testbed for research on repository-scale premise retrieval, noisy long-context proof generation, and reasoning over deep chains of project-specific definitions and lemmas.
- We provide an evaluation of frontier LLMs and specialized provers under curated and full-repository context regimes, analyzing the relationship between prover success and structural properties of repository dependence.

2 VeriSoftBench

VeriSoftBench is a benchmark of 500 Lean 4 proof obligations drawn from real, open-source formal-methods developments. We target Lean 4 as it is the primary language of recent neural theorem provers. As Table 1 shows, existing Lean benchmarks either operate within the shared Mathlib ecosystem or introduce limited project-specific context without multi-module structure. VeriSoftBench fills this gap: each task is extracted from a fixed Git commit and packaged to preserve the original proof environment, including project-specific abstractions and cross-file dependencies. Theorems are phrased in terms of project-defined datatypes and invariants, and completing the proof requires reasoning within the surrounding codebase.

2.1 Dataset Construction

Benchmark Collection. The benchmark is drawn from a diverse set of 23 Lean repositories, covering compiler correctness, programming language semantics and type systems, verification frameworks, and applied verification projects (e.g., zero-knowledge circuits, protocols, and optimization problems). Figure 2 shows the distribution of these repositories across topics. These categories reflect the current landscape of Lean 4 verification developments: unlike the Lean mathematics ecosystem, which is largely consolidated within Mathlib, software verification efforts are distributed across independent, domain-specific projects. Table 5 in the appendix provides per-repository descriptions, task counts, and dependency statistics.

Benchmark Selection. When selecting benchmarks, we require each task to have a *valid* proof. This means that the ground-truth proof script should contain no proof holes (e.g., `sorry` or `admit`), and any lemmas or theorems it invokes are themselves recursively proven by hole-free scripts. In addition, the Lean file containing the target theorem must compile after replacing the ground-truth proof with `by sorry`, ensuring that the statement, imports, and required definitions are well-formed independently of the original proof term. Finally, we exclude trivial goals by requiring that the theorem is not discharged solely by standard

automation (e.g., `simp`, `omega`, `grind`) and that it references at least one repository-local definition.

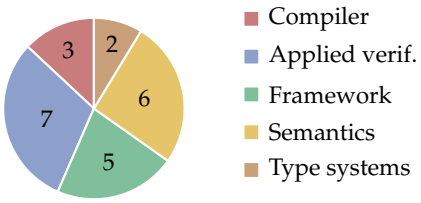


Figure 2: Repository topics.

Beyond enforcing validity, we select tasks that span a range of difficulty metrics. To this end, we score each benchmark along two axes: *proof complexity*, approximated by the length of the repository's ground-truth tactic script, and *contextual dependency*, measured by how many distinct definitions and lemma statements the proof draws from repository-local files. In summary, VeriSoftBench is constructed by first filtering candidate theorems for validity and non-triviality, and then sampling 500 tasks from the remaining pool to balance repository category and the two difficulty axes described above. Across repositories, we select an average of 10.4% and a median of 8.6% of eligible proof obligations; selection rates range from 0.2% for lean-mlir to 26% for LeroyCompilerVerificationCourse. Table 2 summarizes the resulting dataset. Two repositories account for 119 of the 500 tasks; these are substantially larger verification developments whose full-repository context exceeds 200k tokens. We retain them deliberately, as they represent the regime where repository-scale reasoning is most challenging. Thus the non-uniform distribution reflects both the scale and maturity of the current Lean software-verification ecosystem and our goal of retaining substantial repository-scale developments. We balance diversity, realism, and evaluation cost rather than sampling uniformly by repository or exhaustively retaining every theorem; Appendix B.1 verifies that the main trends hold within repositories.

System Specific Subset. Evaluation on Harmonic's Aristotle system requires strict adherence to specific library versions (Mathlib v4.24.0) and a pre-compiled project environment rather than in-prompt context presentation. To ensure compatibility, we filtered our full task set for version alignment and performed targeted manual refactoring to resolve versioning discrepancies. As the complexity of this engineering effort precluded scaling to the entire 500-task benchmark set, we collected a finalized subset of 100 fully-compatible tasks (VeriSoftBench-Aristotle), enabling evaluation on the state-of-the-art specialized prover.

Cat.	Repos	Tasks	Curated	Full Repo	Tr.	Total
Regular	21	381	5,407	44,934	21	813
Large	2	119	6,339	2,928,342	30	29,112

Table 2: Dataset statistics. "Large" = context >200k tokens. Values are per-repo averages.

Two aspects of this configuration merit clarification. First, because Aristotle operates within a compiled project, the evaluation context *additionally* includes same-file lemma statements preceding the target theorem. Removing these lemmas while retaining a compilable environment would require restructuring the source files and selectively re-compiling, which was infeasible at scale. As a result, VeriSoftBench-Aristotle represents a simpler setting in which all same-file lemma statements—including those used by the ground-truth proof—are available to the prover. Second, one might ask whether the curated context (Section 2.2) could be packaged as a standalone compilable project for Aristotle. This is not straightforward: the curated context tracks definitions transitively but does not transitively track the lemma dependencies of included lemmas, so it does not in general form a self-contained, compilable unit.

Task Composition. Each task consists of the target theorem, the local file context up to the theorem, all required local-repository definitions and helpful repository lemma declarations for stating and proving the target theorem, and the necessary library imports, as illustrated in Figure 3. Tasks are extracted without rewriting definitions or replacing local abstractions with equivalent standard-library formulations, preserving the original dependency structure of each proof.

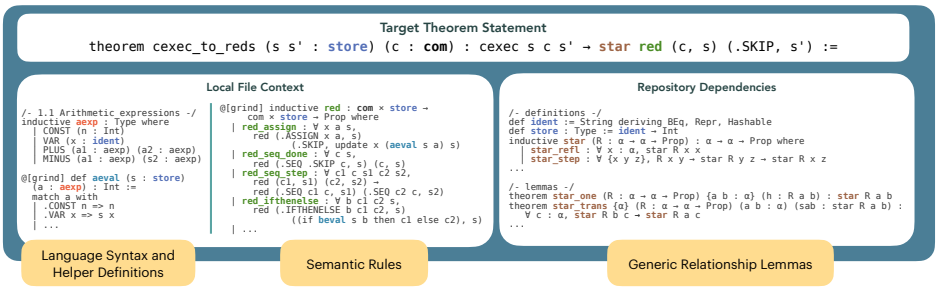


Figure 3: An example VeriSoftBench task. The target `cexec_to_reds` relates two project-defined execution relations. The left (local-file) panel supplies the language syntax and semantic rules in scope at the theorem; the right (repository) panel supplies cross-file base types and relational lemmas. Matching colors trace each project-specific symbol in the target to the declaration that gives it meaning, illustrating why the proof cannot be treated as an isolated theorem statement.

2.2 Contextual Configurations

To study how provers use repository-local context under different levels of noise, we evaluate each task under two contextual configurations. Both configurations include the same *base context*, which includes: (i) the contents of the local file up to the target theorem (i.e., the definitions in scope) excluding the lemmas, and (ii) the external library (e.g., Mathlib, Std) imports and declarations required for the theorem statement to type-check.

Curated Context. The curated context mode is an intentionally oracle-like diagnostic condition that provides the model with a focused set of dependencies designed to approximate the necessary background knowledge with minimal distraction for completing a proof. In addition to the base context, it includes only the explicit repository dependencies: the specific definitions and lemmas from other files in the same repository that are directly invoked by the ground truth target theorem, and the transitively invoked definitions by any dependency in the context. To ensure the integrity of the evaluation and prevent models from mimicking domain-specific proof scripts, we elide the proof bodies of all provided lemmas and definitions. Because these premises are identified from the ground-truth proof, this configuration is not representative of deployment. It instead represents a best-case diagnostic scenario that separates context utilization from retrieval performance without leaking the target proof.

Full Context. In this setting, the prover receives the entire repository library in addition to the base context. This includes all definitions and lemma/theorem statements across *all files* in the repository. It is worth noting that the resulting context can be extremely long: among the 500 tasks, 119 come from repositories whose full-repository context exceeds hundreds of thousands of tokens, and one reaches hundreds of millions of tokens (Table 2). Such instances are far beyond the effective context windows of current models, underscoring the need for reliable context-selection mechanisms. When the full repository context exceeds a model's context window, we first ensure that all declarations present in the curated context are included, then fill the remaining window capacity with additional repository declarations. This guarantees that the relevant dependencies are always available, even when the full repository cannot be provided in its entirety.

Proof isolation. Both configurations are designed to ensure that the target proof is never revealed in the provided context: we include only *preceding* content within the local file and never provide subsequent lemmas or proofs. Whenever we provide repository lemmas or definitions (either in the curated set or as part of the full repository), we include only their statements and elide proof bodies. Nevertheless, the Full Context configuration exposes the broader module ecosystem in which the theorem lives. As such, this ambient structural information can provide cues about recurring abstraction patterns or common proof structure that is largely absent in the curated configuration.

3 Experiments

3.1 Baselines

To evaluate performance on VeriSoftBench, we select a diverse set of models spanning frontier general-purpose LLMs and specialized formal reasoning systems.

We evaluate three primary baseline models that represent the current frontier of general-purpose reasoning and code generation: GPT-5.2, Claude-Opus-4.5, and Gemini-3-Pro. To broaden the evaluation beyond closed frontier models, we include Goedel-Prover-V2-32B (Lin et al., 2025), an open-weight specialized Lean prover. We also evaluate Harmonic's Aristotle Prover (Achim et al., 2025), a strong specialized prover system. These systems cover complementary categories—general-purpose LLMs, an open-weight prover model, and an integrated prover service—while remaining compatible with independently packaged Lean repositories and their project-specific toolchains.

3.2 Evaluation pipeline

We evaluate general-purpose LLMs using a generation-check-repair loop, as illustrated in Figure 4. For each evaluation run, the model receives a task associated with the selected contextual configuration as input, and the evaluation proceeds in three stages: (1) we sample k candidate proof scripts from the model. Each candidate is generated as a tactical proof block intended to satisfy the target goal; (2) we wrap each generated script into its original file context and attempt verification using the Lean 4 compiler. To ensure a stable and efficient evaluation, the all task environments are pre-built prior to verification; we use the specific toolchain version associated with the source repository of each task and enforce a 300-second compilation timeout per candidate[1]; (3) for any failing script, we invoke the model for up to r repair rounds using the compiler error message, corresponding code lines from each error location, and the current proof state as feedback. A task is marked as "Solved" if any of the k candidates or their subsequent repairs pass verification. For specialized theorem-proving systems with an internal feedback loop (e.g., Goedel-Prover-V2 and Aristotle), we evaluate them as end-to-end systems rather than wrapping them in an external repair loop, as they already contain their own internal feedback loop.

3.3 Data Configurations

We consider two variations of VeriSoftBench that differ in scope and in the degree to which they accommodate prover-specific engineering constraints.

VeriSoftBench-Full. Our primary evaluation setting is the full VeriSoftBench benchmark, consisting of 500 proof obligations. This setting is used for all models and provers that can consume our standard packaging of repository context.

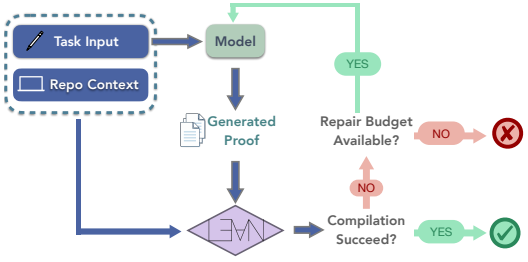


Figure 4: Evaluation pipeline.

VeriSoftBench-Aristotle. As described in Section 3.1, Aristotle requires a pre-compiled project environment rather than in-prompt context. We evaluate it on the 100-task VeriSoftBench-Aristotle subset, which additionally includes same-file lemma statements (see Section 3.1 for details on why this is necessary and why the curated context cannot serve as a substitute). To enable a controlled comparison on this subset, we also evaluate Gemini-3-Pro under the same configuration.

[1]Large dependencies such as Mathlib are compiled once and cached per repository toolchain version.

Data Setting	Model Setting	Context Conditions	
		Curated	Full Repo
VeriSoftBench-Full	Claude Opus 4.5 (Pass@8, $r=3$)	31.2%	23.2%
	GPT-5.2 (Pass@8, $r=3$)	12.6%	10.8%
	Gemini-3-Pro (Pass@8, $r=3$)	41.0%	34.8%
	Goedel-Prover-v2 (Pass@8, $r=0$)	5.6%†	5.2%†
VeriSoftBench-Aristotle	Aristotle	-	69%*
	Gemini-3-Pro (Pass@8, $r=3$)	-	65%*

Table 3: VeriSoftBench evaluation results. † Four tasks whose required context exceeds the Goedel-Prover-V2 context window are excluded from both context conditions. *Uses the separate 100-task compatibility subset with full context and preceding same-file lemma statements (Section 2.2); these rows are not directly comparable to VeriSoftBench-Full.

4 Results and Analysis

Model Performance. Table 3 summarizes the verification success rates across our two data settings (Full Set, Subset) and two contextual configurations (Curated Context, Full Context). Overall, performance is low, highlighting the inherent difficulty of the VeriSoftBench tasks.

Under the Curated Context, the best-performing model, Gemini-3-Pro, achieves a success rate of 41.0%, followed by Claude Opus 4.5 at 31.2%. Thus, even oracle-like dependency selection leaves 59% of tasks unsolved, showing that retrieval is not the only bottleneck. In the more challenging Full Repo Context, success rates decrease across all evaluated models; Gemini-3-Pro and Claude Opus 4.5 achieve 34.8% and 23.2%, respectively, while GPT-5.2 reaches 10.8%. The open-weight Goedel-Prover-V2 solves 5.6% under curated context and 5.2% under full-repository context, indicating limited transfer to repository-scale software verification.

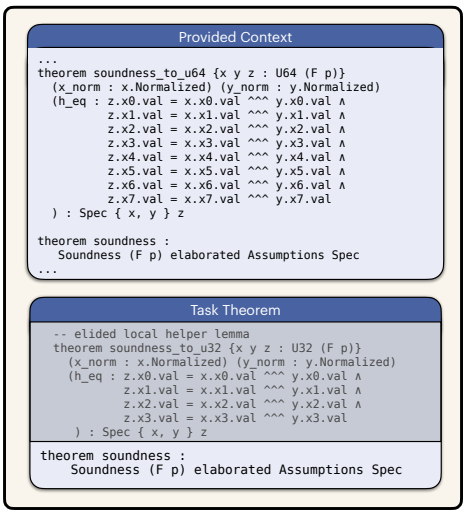


Figure 5: Full-repository context exposes the analogous `soundness_to_u64` proof as a pattern for the target `soundness_to_u32`; curated context omits it.

Aristotle Comparison. Aristotle requires a compiled project environment, so we evaluate it separately on the 100-task VeriSoftBench-Aristotle subset and compare it with Gemini-3-Pro under the same configuration (Section 2.2). With preceding same-file lemma statements available, Aristotle solves 69% of the tasks and Gemini-3-Pro solves 65%. Removing these statements and evaluating Gemini-3-Pro under the standard full-context protocol reduces its success rate to 41%, showing that the shared configuration is substantially easier than the main benchmark setting. The four-point difference in the matched comparison captures the combined effect of Aristotle's compiled environment and agentic proof search.

Performance also varies markedly by repository scale. On the 21 regular-sized repositories (381 tasks), Gemini-3-Pro achieves 47.6% (curated) and 43.7% (full). On the two large repositories whose full context exceeds 200k tokens (ArkLib and lean-mlir, 119 tasks), success drops to 17.6% and 10.9%. For these large repositories, the full context far exceeds model context windows; as described in Section 2.2, we prioritize curated dependencies and fill the remaining window with additional repository declarations.

Comparison of Context Conditions. Full-repository context consistently reduces success, although by less than its much larger search space might suggest (Table 3). For Gemini-3-Pro, for example, the gap between curated and full-repository context is 6.2 percentage points (41.0% vs. 34.8%). Curated context uses ground-truth proof dependencies to identify relevant declarations, whereas full-repository context requires the prover to find them among many additional declarations. The gap therefore measures the effect of this added repository context while keeping the relevant dependencies available in both conditions.

One possible explanation is that full context provides compensating signals despite its much larger search space. Files outside the ground-truth dependency closure can contain structurally similar proofs, auxiliary lemmas, or recurring abstraction patterns. Figure 5 gives a concrete example: `soundness_to_u64` is not required by the target proof and is therefore absent from curated context, but its proof can cue the construction of `soundness_to_u32`. This is an illustrative mechanism rather than a causal attribution for the aggregate gap.

Impact of High Contextual Dependency. The difficulty of our benchmark is closely tied to its unusually high degree of contextual dependency. Table 4 quantifies the size of the dependency footprint. Theorems in VeriSoftBench directly reference only a modest number of items (2.3 library and 11.6 project dependencies on average), but their *transitive closure* expands to 11.2 library dependencies and 37.9 project dependencies *per theorem*. In contrast, Verina tasks involve only shallow local dependencies (around 3 transitive dependencies on average), while PutnamBench theorems are entirely self-contained with zero project dependencies.

However, dependency size alone does not fully capture structural difficulty. Figure 6 shows the dependency *nesting depth* — i.e., the length of the longest chain in the transitive dependency graph induced by referenced definitions and lemmas. The distribution reveals that many theorems require reasoning across long dependency chains: roughly half of the theorems have nesting depth ≥ 5, and around 10% exceed depth 12. This indicates that relevant information is often separated from the theorem statement by multiple layers of intermediate abstractions.

These two observations highlight an important distinction. A theorem may depend on dozens of definitions (large dependency size), but the key difficulty arises when those dependencies are arranged in long chains, where each step builds on the previous one. Our correlation analysis (Table 6, Appendix) confirms this: nesting depth shows the strongest correlation with failure (Spearman $r = -0.356$, full), followed by total transitive dependency count ($r = -0.337$), while direct dependency counts are more weakly correlated. We report correlations for additional features including proof length in the Appendix.

To make this trend concrete, we bucket tasks by dependency depth and report success rates for Gemini-3-Pro (with repair) under each context regime. At depth 1–3, success is 38.0% (curated) and 32.6% (full); at depth 4–6, 25.5% and 23.5%; at depth 7+, 21.1% and 15.6%. The decline is monotonic across both regimes, and the gap between curated and full context widens at higher depths. We note that our benchmark is intentionally weighted toward tasks with nontrivial dependency structure (50% at depth ≥ 5), as this regime is underrepresented in existing benchmarks and reflects the structure of real verification codebases. Tasks with shallow dependencies are already well served by benchmarks such as Verina and MiniCodeProps.

Comparison of Proof Patterns. Beyond dependency structure, VeriSoftBench also differs from prior datasets in the proof patterns it induces. To compare proof styles across benchmarks in a controlled way, ground-truth proofs are not sufficient: many benchmarks do not provide them, and even when available, human-written proofs and prover-specific proof search can introduce stylistic confounders. We therefore perform a cross-dataset comparison (e.g., Verina vs. VeriSoftBench) using Gemini-3-Pro generated proofs that successfully verify in both datasets. This holds the prover constant and allows observed differences to be attributed primarily to the theorems themselves.

Under this controlled comparison, Verina and VeriSoftBench exhibit systematically different tactic profiles. In Verina, 45% of correct Gemini-generated proofs open with `unfold` and 34% close with `simp`, with automation tactics accounting for 29% of tactic usage overall; 13%

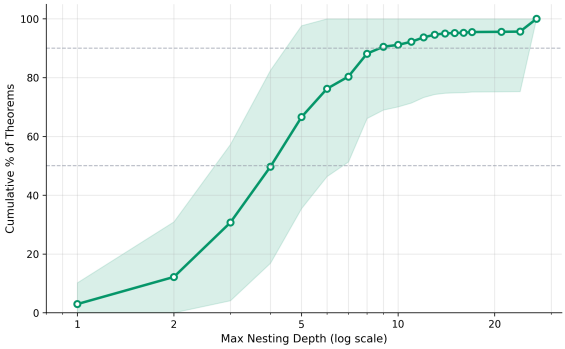


Figure 6: Cumulative distribution of dependency depth (log scale). Line: mean; shading: ± 1 std.

Bench.	N	Lib (Dir.)	Lib (Tr.)	Proj (Dir.)	Proj (Tr.)
Ours	500	2.3	11.2	11.6	37.9
Verina	189	0.0	0.0	3.0	3.0
Putnam	662	3.3	3.3	0.0	0.0

Table 4: Dependency footprint. Lib = library; Proj = project (excluding same-file); Dir. = direct; Tr. = transitive.

of Verina proofs are discharged by a single automation call. VeriSoftBench proofs follow a markedly different pattern: only 7% begin with `unfold`, 10% close with `simp`, and none are solved by one-shot automation. Automation accounts for just 10% of tactic usage in VeriSoftBench proofs, while 71% exhibit branching structure involving case analysis or induction (compared to 38% in Verina). These differences suggest that Verina proofs can often be resolved by unfolding definitions and invoking automation, whereas VeriSoftBench tasks require multi-step reasoning with explicit case splits and lemma application.

5 Related Work

Formal Verification and Benchmarks. Formal verification provides strong assurance but proof effort remains a bottleneck. Tools such as Dafny (Leino, 2010) and Verus (Lattuada et al., 2023) automate proofs via SMT solvers, while interactive theorem provers support manual proof construction, enabling verified artifacts such as CompCert (Leroy, 2009), seL4 (Klein et al., 2009), and CakeML (Kumar et al., 2014). Domain-specific frameworks like Iris (Jung et al., 2015), Verdi (Wilcox et al., 2015), and Loom (Gladshtein et al., 2026) reduce overhead but still require significant manual effort. On the benchmarking side, early suites such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) validate code by unit tests without formal guarantees. DafnyBench (Loughridge et al., 2024) targets invariant synthesis and MiniCodeProps (Lohn & Welleck, 2024) evaluates proofs for small functional programs, while FVAPPS (Dougherty & Mehta, 2025), CLEVER (Thakur et al., 2025), and Verina (Ye et al., 2025) target verified code generation. VeriSoftBench complements these by focusing on proof obligations embedded in repository-scale Lean developments with substantial project-specific context.

General ITP Benchmarks. Widely used Lean benchmarks target formalized mathematics: MiniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), LeanDojo (Yang et al., 2023), and PutnamBench (Tsoukalas et al., 2024) operate within a shared Mathlib ecosystem. The miniCTX effort (Hu et al., 2025) supplies surrounding project context but remains focused on mathematical theorems. Outside Lean, CoqGym (Yang & Deng, 2019) aggregates 71K proofs from 123 Coq projects but targets step-by-step tactic prediction, where the proof assistant manages the environment at each goal, rather than the whole-proof generation paradigm used by current LLM-based provers. VeriSoftBench complements CoqGym by targeting whole-proof generation in Lean 4 with explicitly controlled repository context. Extending similar benchmarks to other ITPs (e.g., Rocq, Isabelle) is a natural direction for future work.

LLM-based Prover Systems. Neural proof automation has progressed from early tactic prediction (Lample et al., 2022; Polu et al., 2022; Polu & Sutskever, 2020) to agentic systems using verifier feedback. COPRA (Thakur et al., 2024) and AxProver (Breen et al., 2025) run generate-check-repair loops; DSP (Jiang et al., 2023) constructs intermediate sketches. For premise selection, LeanDojo (Yang et al., 2023) and Rango (Thompson et al., 2025) apply retrieval-augmented proving. Specialized prover models include Llemma (Azerbayev et al.,

2024), Kimina-Prover (Wang et al., 2025), DeepSeek-Prover-V2 (Ren et al., 2025), Gödel-Prover-V2 (Lin et al., 2025), SeedProver (Chen et al., 2025), and Hilbert-Prover (Varambally et al., 2025). Despite significant advances, these systems are predominantly evaluated on mathematics-centric benchmarks, motivating VeriSoftBench.

6 Limitations and Future Work

Our benchmark evaluates single-theorem proving for fully specified goals, and therefore does not capture multi-stage verification workflows such as specification synthesis, proof planning, or joint proving of multiple theorems where intermediate lemmas serve several goals. Even within this setting, we focus on native long-context reasoning rather than evaluating practical retrieval-augmented or agentic context-selection baselines. The curated regime is an oracle-like diagnostic condition, not a deployable retrieval method: it uses ground-truth proof dependencies to separate premise selection from context utilization. Its best result of 41.0% shows substantial headroom even after relevant premises have been identified, but does not predict the performance of a practical retriever. VeriSoftBench is intended to support follow-up work on repository-scale retrieval and premise selection, long-context proof generation under noisy context, and robustness to multi-hop project-specific dependencies. Our evaluation also does not isolate why Goedel-Prover-V2 transfers poorly to this setting: project-specific abstractions, long dependency chains, distribution shift from mathematics benchmarks, and the Pass@8 budget may all contribute. Finally, because VeriSoftBench draws from public GitHub repositories, we cannot fully rule out contamination in the pre-training data of closed-weight models. That said, the strong correlation between dependency depth and failure suggests that memorization alone is insufficient for these tasks.

7 Conclusion

We introduced VeriSoftBench, a benchmark contribution comprising 500 Lean 4 proof obligations from open-source formal-methods developments while preserving project context and cross-file dependencies. Our experiments show that current provers achieve only modest success, with performance degrading most strongly as dependency nesting depth and transitive dependency structure grow. The intentionally oracle-like curated condition improves performance but still leaves substantial headroom, indicating that repository-scale verification requires both effective context selection and reasoning over project-specific abstractions. We hope VeriSoftBench supports future experiments on retrieval and premise selection, noisy long-context generation, and robust reasoning across deep repository dependency chains.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was conducted in a research group supported by NSF awards CCF-1918889, CNS-2120696, CCF-2210831, and CCF-2319471, CCF-2422130, CCF-2403211, CCF-2505865, CCF-2326576, and CCF-2403211 as well as a DARPA award under agreement HR00112590133 and HR0011262E020. This work is also supported in part by a Canada CIFAR AI Chair award to QC. This research computation has been supported by the Vista GPU Cluster through the Center for Generative AI (CGAI) and the Texas Advanced Computing Center (TACC) at the University of Texas at Austin, as well as a Google Compute Grant for running Gemini-3-Pro.

References

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold

Williams, and Lawrence Wu. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/2510.01346.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. URL https://arxiv.org/abs/2302.12433.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics, 2024. URL https://arxiv.org/abs/2310.10631.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M. Taylor, Frank H. L. Koppens, and Dirk Englund. Axprover: A deep reasoning agentic framework for theorem proving in mathematics and quantum physics, 2025. URL https://arxiv.org/abs/2510.12787.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL https://arxiv.org/abs/2512.17260.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2502.05714.

Emily First, Yuriy Brun, and Arjun Guha. Tactok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428299. URL https://doi.org/10.1145/3428299.

Vladimir Gladshtein, George Pîrlea, and Ilya Sergey. Small scale reflection for the working lean user, 2024. URL https://arxiv.org/abs/2403.12733.

Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. Foundational multi-modal program verifiers. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi: 10.1145/3776719. URL https://doi.org/10.1145/3776719.

Jiewen Hu, Thomas Zhu, and Sean Welleck. minictx: Neural theorem proving with (long-)contexts, 2025. URL https://arxiv.org/abs/2408.03350.

Albert Q. Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers, 2022. URL https://arxiv.org/abs/2205.10893.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs, 2023. URL https://arxiv.org/abs/2210.12283.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *SIGPLAN Not.*, 50(1):637–650, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676980. URL https://doi.org/10.1145/2775051.2676980.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pp. 179–191. ACM Press, January 2014. doi: 10.1145/2535838.2535841. URL https://cakeml.org/popl14.pdf.

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022. URL https://arxiv.org/abs/2205.11491.

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 (extended version), 2023. URL https://arxiv.org/abs/2303.05491.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

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, 2025. URL https://arxiv.org/abs/2508.03613.

Evan Lohn and Sean Welleck. minicodeprops: a minimal benchmark for proving code properties, 2024. URL https://arxiv.org/abs/2406.11915.

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, 2024. URL https://arxiv.org/abs/2406.08467.

Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. Lean-smt: An smt tactic for discharging proof goals in lean, 2025. URL https://arxiv.org/abs/2505.15796.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

Lawrence Paulson and Jasmin Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. 02 2015. doi: 10.29007/tnfd.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022. URL https://arxiv.org/abs/2202.01344.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Marcus Rossel. lean-egg: An equality saturation tactic for lean. https://github.com/marcusrossel/lean-egg, 2024. GitHub repository, Apache-2.0 license.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024. URL https://arxiv.org/abs/2310.04353.

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, 2025. URL https://arxiv.org/abs/2505.13938.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.2 edition, 2008. https://rocq-prover.org/doc/V8.2pl3/refman/Reference-Manual001.html.

Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 347–359. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00161. URL https://doi.org/10.1109/ICSE55347.2025.00161.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition, 2024. URL https://arxiv.org/abs/2407.11214.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning, 2025. URL https://arxiv.org/abs/2509.22819.

Verified-zkEVM. clean: Lean circuit dsl for zero-knowledge circuits. https://github.com/Verified-zkEVM/clean, 2025. GitHub repository; an embedded Lean4 DSL for writing and formally verifying ZK circuits (ARB, PLONK, R1CS) as part of the Verified-zkEVM project.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. 2025. URL http://arxiv.org/abs/2504.11354.

Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (eds.), *Theorem Proving in Higher Order Logics*, pp. 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737958. URL https://doi.org/10.1145/2813885.2737958.

Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6984–6994. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/yang19a.html.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023. URL https://arxiv.org/abs/2306.15626.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation, 2025. URL https://arxiv.org/abs/2505.23135.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022. URL https://arxiv.org/abs/2109.00110.

A Prompts

Prompt for Proof Generation (Curated Context Mode)

```
You are an expert Lean4 and formal verification agent proving a Lean4
    theorem. Your output **must** strictly adhere to the specified
    format.

### Input Information
You will receive the following components, relevant to the target
    theorem:
* <used_lib_defs>...</used_lib_defs>: Imported library definitions
    directly used by the theorem.
* <used_repo_defs>...</used_repo_defs>: Definitions from the local
    project (but outside the current file), directly or transitively
    used by the theorem.
* <lib_lemmas>...</lib_lemmas>: Imported library lemmas/theorems that
    may assist the proof.
* <repo_lemmas>...</repo_lemmas>: Lemmas/theorems defined in the local
     project outside this file.
* <local_ctx>...</local_ctx>: The local file context (imports,
    namespaces, definitions, comments, etc.) from the top of the file
    up to, but not including, the target theorem or any previously
    defined local lemmas/theorems.
* <target_theorem>...</target_theorem>: The target theorem statement,
    ending with `:=`.

**Notes**
1. Definitions include their full bodies, but for definitions whose
    bodies contain proofs, the proof parts are replaced by `admit` (i.
    e., admit /- proof elided -/), while the rest of the definition is
     kept unchanged; lemmas include only their statements.
2. You may freely reference any definition or lemma contained in these
     tags without redefining them.
3. The tags for used definitions and helpful lemmas (used_lib_defs,
    used_repo_defs, lib_lemmas, repo_lemmas) may be omitted if empty.

### Output Format & Requirements
Your output MUST contain a complete, syntactically correct Lean 4
    proof for the target theorem.

**Strict Output Structure:**
<reasoning>
...
</reasoning>

<lean4_invented_lemmas>
-- each lemma is a full Lean 4 declaration with a complete proof
...
</lean4_invented_lemmas>

<lean4_proof>
-- Lean 4 proof body of the target theorem
...
</lean4_proof>

**Requirements:**
1. <reasoning> and <lean4_invented_lemmas> are OPTIONAL. Include them
    only if they aid proof construction.
2. <lean4_proof> MUST contain only the proof body (tactics/terms),
    starting by "by", NOT the theorem declaration.
```

```
3. <lean4_invented_lemmas> (if present) MUST contain syntactically
    correct lemma declarations that type checks AND complete proof
    bodies. Defining a lemma using `axiom` is NOT allowed.
4. <reasoning> contains high-level proof planning.
```

Prompt for Proof Generation (Curated Context Mode) - Cont

```
5. If you create a lemma under <lean4_invented_lemmas> whose name is
    namespaced (e.g., X.Y.Z), you MUST always refer to it using its
    fully qualified name. Do NOT rely on shortened names. For example,
     write rw [X.Y.Z], not rw [Z] for invented lemmas.
6. All Lean 4 code MUST be syntactically correct. No `sorry`, `admit`,
     or non-Lean text. Comments allowed using `--` or `/- ... -/`.
7. Follow the tags and structure EXACTLY. No extra text before,
    between, or after the sections.
8. Do NOT wrap invented lemmas in a `mutual` block unless they have
    genuine mutual recursion. Declare each lemma independently.

### Examples are provided here (example input are under "User Prompt",
     and example assistant response are under "Agent Response"):
{}

### Your task:
{}
```

Prompt for Proof Generation (Full Context Mode)

```
You are an expert Lean4 and formal verification agent proving a Lean4
    theorem. Your output **must** strictly adhere to the specified
    format.

### Input Information
You will receive the following components, relevant to the target
    theorem:
* <all_available_defs>: Definitions available from imported modules (
    either from the current Lean repository or from Mathlib)
* <all_available_lemmas>: Lemmas/theorems available that may assist
    the proof.
* <local_ctx>...</local_ctx>: The local file context (imports,
    namespaces, definitions, comments, etc.) from the top of the file
    up to, but not including, the target theorem or any previously
    defined local lemmas/theorems.
* <target_theorem>...</target_theorem>: The target theorem statement,
    ending with `:=`.

**Notes**
1. Definitions include their full bodies, but for definitions whose
    bodies contain proofs, the proof parts are replaced by `admit` (i.
    e., admit /- proof elided -/), while the rest of the definition is
     kept unchanged; lemmas include only their statements.
2. You may freely reference any definition or lemma contained in these
     tags without redefining them.
3. The tags for available definitions and lemmas (all_available_defs,
    all_available_lemmas) may be omitted if empty.

### Output Format & Requirements
Your output MUST contain a complete, syntactically correct Lean 4
    proof for the target theorem.
```

```
**Strict Output Structure:**
<reasoning>
...
</reasoning>

<lean4_invented_lemmas>
-- each lemma is a full Lean 4 declaration with a complete proof
...
</lean4_invented_lemmas>

<lean4_proof>
-- Lean 4 proof body of the target theorem
...
</lean4_proof>

**Requirements:**
1. <reasoning> and <lean4_invented_lemmas> are OPTIONAL. Include them
    only if they aid proof construction.
2. <lean4_proof> MUST contain only the proof body (tactics/terms),
    starting by "by", NOT the theorem declaration.
3. <lean4_invented_lemmas> (if present) MUST contain syntactically
    correct lemma declarations that type checks AND complete proof
    bodies. Defining a lemma using `axiom` is NOT allowed.
4. <reasoning> contains high-level proof planning.
5. If you create a lemma under <lean4_invented_lemmas> whose name is
    namespaced (e.g., X.Y.Z), you MUST always refer to it using its
    fully qualified name. Do NOT rely on shortened names. For example,
     write rw [X.Y.Z], not rw [Z] for invented lemmas.
6. All Lean 4 code MUST be syntactically correct. No `sorry`, `admit`,
     or non-Lean text. Comments allowed using `--` or `/- ... -/`.
7. Follow the tags and structure EXACTLY. No extra text before,
    between, or after the sections.
8. Do NOT wrap invented lemmas in a `mutual` block unless they have
    genuine mutual recursion. Declare each lemma independently.
```

Prompt for Proof Generation (Full Context Mode) - Cont

```
### Examples are provided here (example input are under "User Prompt",
     and example assistant response are under "Agent Response"):
{}

### Your task:
{}
```

Prompt for Proof Repair

```
You are an expert Lean 4 and formal verification agent fixing broken
    Lean 4 code. Your output **must** strictly adhere to the specified
     format.

### Input Information
You will receive the following components:
* <code_before_error>...</code_before_error>: supporting Lean 4
    context, including import statements, variables, and any
    definitions that precede the broken code.
* <potential_broken_lemmas>...</potential_broken_lemmas>: one or more
    lemma declarations and their proofs that may contain errors. This
    section is optional.
```

```
* <potential_broken_theorem>...</potential_broken_theorem>: the
    theorem declaration and proof that may contain errors. Exactly one
     theorem will be provided here.
* <err_msg>...</err_msg>: one or more error messages. Each error will
    be wrapped in <err_<id>>...</err_<id>> tags. The content inside is
     structured as: <Code snippet where the error occurs>\n ErrorMsg:
    <actual message from the Lean compiler>

*Repair Requirements*
1. Identify where and why the error occurs based on the text given in
    each error message.
2. If an error is in a lemma, you may modify both the lemma statement
    (declaration) and its proof.
3. If an error is in the theorem, you MUST NOT change the theorem
    statement; you may only modify its proof.
4. The modification should be the minimal, most idiomatic Lean 4
    change necessary to resolve the error.
5. You must prioritize solving the errors instead of avoiding them.

### Output Format & Requirements
Your output MUST contain Lean 4 code that is syntactically correct.

**Strict Output Structure:**
<fixed_lean4_lemmas>
-- All candidate lemmas: each contains a full statement/declaration
    and proof.
</fixed_lean4_lemmas>

<fixed_lean4_thm_proof>
-- Lean 4 proof body (term or tactic block) for the potentially broken
     theorem,
-- WITHOUT repeating its declaration.
</fixed_lean4_thm_proof>

**Output Requirements:**
1. Omit a section entirely if you make no changes to that component. E
    .g., if all lemmas are correct, omit <fixed_lean4_lemmas>. If the
    theorem proof is correct, omit <fixed_lean4_thm_proof>
2. `<fixed_lean4_lemmas>` (if present) MUST containfull, type-checking
     Lean 4 lemma declarations and complete proofs.
   - You MUST output every candidate lemma provided in <
       potential_broken_lemmas>
3. `<fixed_lean4_thm_proof>` (if present) MUST contain only the proof
    body (term or tactics), NOT the theorem declaration, because you
    are NOT allowed to modify the theorem declaration.
4. Follow the tags and structure EXACTLY. No extra text before,
    between, or after the sections.
5. The output MUST NOT use keywords that skip the proof (e.g., admit,
    sorry).

Here is your error fixing task:
{}
```

B Repository Statistics and Dependency Analysis

Table 5 provides per-repository statistics for all 23 projects in VeriSoftBench. Repositories vary widely in both size and difficulty. The four largest repositories (ArkLib, capless-lean, clean, lean-mlir) contribute 238 of the 500 tasks and exhibit high transitive dependency counts, reflecting deep project-specific abstraction layers. Success rates range from 0% (IntroEffects, formal-snarks) to 100% (veil, lean-hoare, Lentil), though the latter come from small repositories with few tasks. Across most repositories, curated context yields equal

or higher success rates than full-repository context, consistent with the main text findings. Notable exceptions include pcf-lean, where full-repository context outperforms curated (71.4% vs. 57.1%), suggesting that ambient repository structure can occasionally provide useful proof cues.

			Proj Deps		Lib Deps		Success (%)	
Repository	**Description**	**N**	**Dir.**	**Tr.**	**Dir.**	**Tr.**	**Cur.**	**Full**
ArkLib	Verified SNARKs	60	8.8	38.0	16.6	79.3	25.0	15.0
capless-lean	Capability type system	60	48.2	91.0	0.3	3.7	51.7	46.7
clean	Verified ZK circuit DSL	59	12.7	53.9	6.9	27.9	30.5	25.4
lean-mlir	SSA compiler IR verif.	59	12.5	51.3	7.6	33.7	10.2	6.8
VCV-io	Crypto protocol verif.	24	6.0	17.1	7.6	18.8	70.8	66.7
iris-lean	Separation logic framework	24	19.3	46.0	0.7	8.1	33.3	20.8
loom	Verifier generation	24	8.5	23.5	5.2	26.8	37.5	25.0
splean	Separation logic proofs	24	13.3	33.5	8.1	25.7	29.2	20.8
TTBFL	Bounded universe levels	22	15.5	34.5	1.1	11.0	63.6	59.1
FVIntmax	Blockchain protocol verif.	21	8.5	38.6	7.0	25.5	19.0	14.3
juvix-lean	Juvix compiler correct.	20	33.1	28.4	0.8	14.4	65.0	60.0
lean-formal-reas.	Formal prog. reasoning	20	7.8	14.2	0.3	1.6	90.0	85.0
wadray_verif.	Fixed-point arith. verif.	20	9.4	11.3	20.1	24.9	50.0	50.0
LeroyCompVerif	IMP compiler correct.	19	12.3	25.7	1.3	14.2	26.3	21.1
CvxLean	Convex optimization	9	5.2	11.3	9.7	30.3	55.6	55.6
pcf-lean	PCF denotational sem.	7	6.6	25.0	1.4	8.0	57.1	71.4
LeanExprEval.	Expression evaluator	6	12.3	16.7	3.3	26.0	100.0	66.7
veil	Transition system verif.	6	11.0	13.2	0.7	6.8	100.0	100.0
verified-compiler	Toy stack compiler	6	7.7	16.7	2.0	8.3	16.7	16.7
lean-hoare	Hoare logic for While	4	9.5	5.5	0.5	4.0	100.0	100.0
IntroEffects	Algebraic effects	3	4.7	19.0	0.7	15.0	0.0	0.0
formal-snarks	SNARK soundness proofs	2	17.0	27.5	78.5	99.0	0.0	0.0
Lentil	Temporal logic (TLA)	1	5.0	15.0	2.0	2.0	100.0	100.0

Table 5: Repository statistics and benchmark evaluation results. **N**: number of tasks from each repository. **Description**: brief summary of the project's verification target. Columns 4–7 detail the average direct and transitive dependencies for project and library components. The final columns report the best-model success rate under two settings: Curated Context and Full Context.

Table 6 reports Spearman rank correlations between task-level features and Gemini-3-Pro success under both context regimes. We consider dependency features at two granularities (direct references vs. transitive closure), as well as ground-truth proof length as a proxy for intrinsic proof complexity. All features are negatively correlated with success, but structural dependency features—nesting depth ($r = -0.356$) and total transitive count ($r = -0.337$)—are substantially stronger predictors than direct counts ($|r| < 0.19$). Proof length in lines ($r = -0.254$) is moderately correlated, confirming that longer proofs are harder but that dependency structure captures difficulty beyond proof length alone. Correlations are consistent across curated and full-repository settings.

B.1 Task Distribution and Robustness

The 500 tasks in VeriSoftBench are not uniformly distributed across repositories: the four largest projects (ArkLib, capless-lean, clean, lean-mlir) contribute 238 tasks (48%), while nine repositories contribute fewer than 10 tasks each. This reflects natural variation in project scale and maturity as well as deliberate retention of substantial developments that instantiate the repository-scale regime. After validity and non-triviality filtering, the selected fraction averages 10.4% across repositories (median 8.6%), ranging from 0.2% for lean-mlir to 26% for LeroyCompilerVerificationCourse. The five verification domains nevertheless each contribute between 16% and 27% of benchmark tasks. To check that our aggregate

	Curated		Full Repo	
Feature	r	p	r	p
Direct project deps	-0.128	< 0.01	-0.156	< 0.001
Direct library deps	-0.154	< 0.001	-0.179	< 0.001
Total transitive deps	-0.342	< 0.001	-0.337	< 0.001
Nesting depth	-0.338	< 0.001	-0.356	< 0.001
Proof length (lines)	-0.236	< 0.001	-0.254	< 0.001

Table 6: Spearman rank correlations between task features and Gemini-3-Pro success (with repair). *Direct* counts refer to definitions and lemmas explicitly referenced by the theorem; *total transitive* includes all project and library items reachable through the dependency graph. All correlations are negative, indicating higher values are associated with lower success rates.

findings are not driven by cross-repository differences, we verify the key trends under two controls.

First, we compare micro-averaged success rates (across all tasks) with macro-averaged rates (averaging per-repo success rates, giving equal weight to each repository). For Gemini-3-Pro (with repair), the micro-averaged rates are 40.4% (curated) and 35.8% (full), while the macro-averaged rates are 49.3% and 45.8%. The macro average is higher because several small repositories achieve near-perfect success, but the curated-vs-full gap is consistent under both aggregations.

Second, we compute the partial Spearman correlation between nesting depth and success after residualizing out repository identity. The correlation remains significant under both regimes (partial $r = -0.208$, $p < 10^{-5}$ for both curated and full context), confirming that nesting depth predicts difficulty within repositories, not just across them.

B.2 Error Analysis

	Curated		Full Repo	
Error Category	**N**	**%**	**N**	**%**
Unknown identifier	146	49.3	156	48.9
Type mismatch	45	15.2	35	11.0
Unsolved goals	32	10.8	45	14.1
Tactic failure	31	10.5	37	11.6
Other	42	14.2	46	14.4
Total failed	**296**		**319**	

Table 7: Distribution of error types for failed Gemini-3-Pro proof attempts under both context regimes.

Table 7 categorizes the Lean compiler errors produced by failed Gemini-3-Pro proof attempts (with repair) under both context regimes. Errors are classified by keyword matching on the first compiler diagnostic of each failed theorem: *unknown identifier* corresponds to Lean reporting `unknown identifier` or `unknown constant`; *type mismatch* to `type mismatch` or `application type mismatch`; *unsolved goals* to goals remaining after the proof script terminates; and *tactic failure* to any tactic-specific error (e.g., `tactic 'rewrite' failed`). Each theorem is deduplicated across runs, and only theorems where no attempt succeeded are counted.

The dominant failure mode under both regimes is *unknown identifier* ($\sim$49%), indicating that the model frequently hallucinates references to definitions or lemmas that do not exist in the project. This suggests that models struggle to accurately recall project-specific names from

the provided context. The error distribution is similar across curated and full-repository settings, with a slight increase in unsolved goals under full context (14.1% vs. 10.8%), possibly reflecting cases where the model constructs a partially correct proof but cannot close all subgoals when distracted by irrelevant repository content.