APE-Bench I: Towards File-level Automated Proof Engineering of Formal Math Libraries

Huajian Xin Luming Li Xiaoran Jin
Jacques Fleuriot Wenda Li

ByteDance Seed, University of Edinburgh

Presenter: Haotian Liu

October 13, 2025

Outline

Outline

Background

- Large Language Models (LLMs) have achieved strong results in formal theorem proving.
- Current benchmarks test isolated, goal-driven tasks. Real-world formal mathematics is dynamic, interconnected, and evolving.
- Inspired by code generation tasks: from HumanEval to SWE-bench. The formal math community lacks a realistic benchmark paradigm.

Goal: Move from static proofs to workflow-level reasoning and editing.

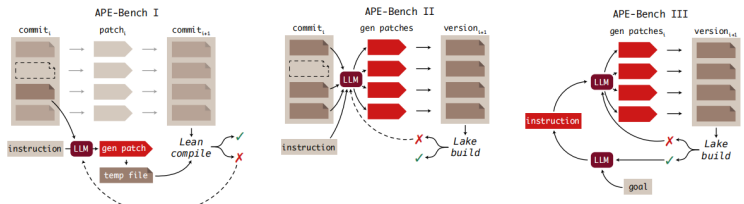
Proposed Paradigm: Automated Proof Engineering

Task format

- **Input:** Natural language instruction + pre-edit Lean file.
- **Output:** Model-generated code patch.

A staged benchmark suite: APE-Bench I → III

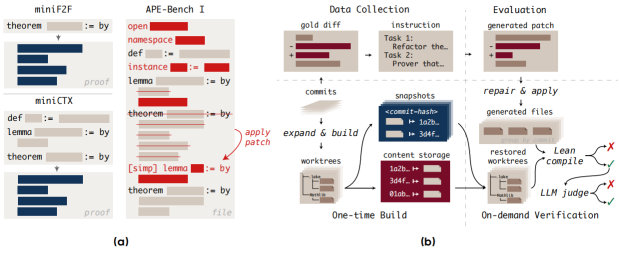
- **APE-Bench I:** Single-file, instruction-guided edits.
- **APE-Bench II:** Multi-file coordination & project-level verification.
- **APE-Bench III:** Autonomous agents with iterative repair.



APE-Bench I: Design Goals

APE-Bench I built from 10K+ real Mathlib4 commits, targeting file-level edits. It has four core goals:

- **Realism:** Preserve real file structure and developer behavior.
- **Stratification:** Multi-stage filtering to remove trivial edits.
- **Scalable Evaluation:** Syntax check via Lean compiler and Semantic check via "LLM-as-a-judge".
- **Continual Evolvability:** Auto-update with Mathlib4 evolution.



Outline

Task Format and Objective

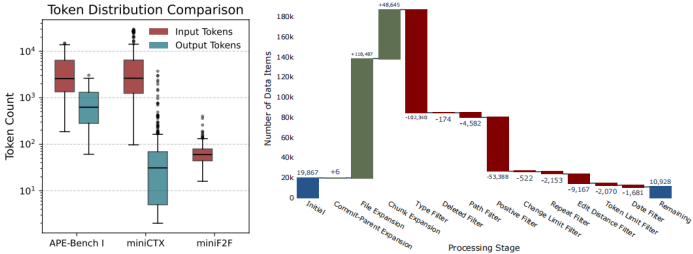
Each task in APE-Bench I defines a localised file-level proof engineering problem, grounded in an actual edit to a Lean source file. It can be specified as a triplet.

- **Instruction:** is a concise imperative-style natural language command describing the intended modification.
- **PreFile:** contains the complete Lean source code of the target file before the edit.
- **GoldDiff:** is a unified diff extracted from human-written commits that serves as the task's ground truth, specifying changes relative to PreFile.

Task Extraction from Mathlib4 Commits

- **Step 1: Commit Sourcing and Diff Decomposition**: Start from 19,867 commits (Aug 2023–Mar 2025) and each commit is decomposed into file-level diffs, isolating modified files and their corresponding changes. Then segment into semantically scoped diffs.
- **Step 2: Granular Chunk Expansion**: Break large edits into smaller fragments for model interpretability and yield$>$187,000 edit candidates.
- **Step 3: Multi-Stage Filtering**: Filter data by Semantic relevance, Content quality, and Structural integrity($\leq$100 changed lines, $\leq$16k tokens).
- **Step 4: Compilation Validation**: Compile post-edit files in Lean, keep only syntactically valid patches.

Task Extraction from Mathlib4 Commits



(a) Token distribution by benchmarks **(b)** Candidate reduction across filtering stages

Figure 3 APE-Bench I: Scale and complexity of extracted tasks. (a) APE-Bench I tasks span significantly longer input–output contexts than prior datasets. **(b)** Starting from 19k commits, the pipeline produces 185k+ candidates, filtered to 10,928 high-quality tasks.

The final benchmark consists of 10,928 structurally coherent tasks. These tasks exhibit significantly longer input and output sequences than prior benchmarks, reflecting the higher structural complexity of proof engineering tasks.

Instruction Synthesis and Task Labelling

For instruction generation, use Claude Sonnet 3.7 to extract and summarise the developer's intent from the raw diff in an imperative style and apply a reverse validation procedure to ensure semantic alignment between instruction and patch. Each task is labeled along three dimensions:

1. Task Category: Feature, Refactor, or Bug Fix.
2. Difficulty Level: Easy $\rightarrow$ Very Hard (Very Easy filtered out).
3. Task Nature: Substantial vs Superficial (only substantial kept).

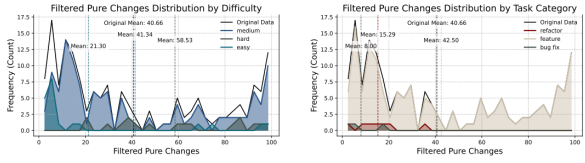


Figure 4 Distribution of code change magnitudes across semantic labels in the test set. (a) Based on non-comment line counts from gold patches, grouped by difficulty. **(b)** Patch sizes grouped by functional category.

Outline

Overview

To enable reliable assessment of model-generated edits in APE-Bench I, establish a two-stage evaluation framework that reflects the dual demands of proof engineering: syntactic correctness and semantic adequacy.
The first stage, syntactic verification, checks whether the resulting file passes Lean verification within the corresponding versioned environment.
The second stage, semantic judgement, determines whether the patch satisfies the instruction as intended.

Syntactic Verification via Eleanstic

Verification requires compiling patches within the exact version. Mathlib4 evolves rapidly, so version fidelity is essential. Introduce Eleanstic:

- Snapshot Generation: Builds each commit once, hashes all files into content-addressable storage(CAS), and records a compact binary snapshot for reproducible environments.
- Snapshot Restoration: During verification, the system restores the required snapshot into an isolated temporary worktree by fetching files from CAS. The process can reconstruct a full Lean environment in under one second while reducing storage by over 90% through global deduplication.
- Patch Verification: Applies the model patch, compiles only the modified file in isolation, and deems it valid if it builds cleanly without warnings.

Semantic Judgement via LLM-as-a-Judge

Passing Lean compilation $\neq$ fulfilling the instruction. Common semantic failure modes: Omission of required edits, Misinterpretation of the instruction, and Unrelated or excessive modifications. Introduce LLM-as-a-Judge:

- Model and Prompt: Use Claude Sonnet 3.7 (thinking mode) as evaluator and input triplet. The LLM checks whether the patch implements the instruction faithfully, maintains logical and structural consistency. The prompt is designed for multi-step reasoning and instruction-agnostic evaluation.
- Voting Strategy for Robustness: Adopt the sample@4 strategy for each task. Collect 4 independent LLM judgements and use majority voting for the final decision. Mark as semantically successful only if most votes confirm that all instruction subgoals are satisfied.

Outline

Experimental Setup

- **Test Model**: Gemini 2.5 Pro Preview, o3-mini, Claude Sonnet 3.7 (with and without thinking mode), DeepSeek R1, DeepSeek V3, GPT-4o, and Doubao 1.5 Pro. All models generate patches in unified diff format without task-specific fine-tuning.
- **Evaluation Metric**: Adopt the standard pass@k evaluation.

$$\text{pass@}k := \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

Contains **Verification pass@16** that at least one candidate compiles successfully using Lean 4, **Judgement pass@16** that at least one compiled candidate additionally satisfies the instruction according to semantic evaluation, and **Relative Decrease** that percentage decrease from Verification to Judgement success.

Qualitative Results

Model	Verification pass@16	Judgement pass@16	Relative Decrease (%)
Gemini 2.5 Pro Preview	18.79%	**18.04%**	3.99%
Claude Sonnet 3.7 (thinking)	12.73%	12.73%	**0.001%**
DeepSeek R1	15.30%	12.55%	17.97%
DeepSeek V3	12.44%	11.81%	5.06%
Claude Sonnet 3.7	11.33%	10.83%	4.41%
o3-mini	**20.13%**	8.60%	57.28%
GPT-4o	15.16%	6.73%	55.61%
Doubao 1.5 Pro	6.19%	2.98%	51.86%

Table 2 Overall model performance on APE-Bench I using pass@16. Relative Decrease is computed as the percentage of Lean-compilable outputs that fail LLM-based semantic judgement.

Gemini 2.5 Pro Preview achieves the best semantic success (18.04%), while o3-mini has the highest verification rate (20.13%) but a large drop in semantic accuracy (57.28%). Claude Sonnet 3.7 (thinking) maintains near-perfect consistency. Overall, even top models solve only about one-sixth of tasks, showing a large gap to real-world proof engineering.

Structural Robustness

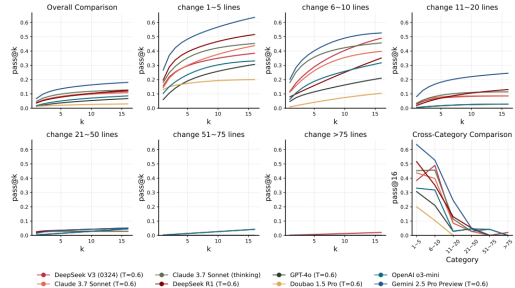


Figure 6 Model pass@16 success rate grouped by ground-truth patch size. Performance declines rapidly with increasing patch size. Gemini 2.5 Pro Preview leads in small to medium ranges, but no model demonstrates robustness beyond 50-line edits.

Model performance drops sharply as edit size increases. While small edits (1–5 lines) achieve moderate success, performance declines for medium (11–20) and large edits (>20), reaching near-zero beyond 50 lines. Gemini 2.5 Pro Preview performs best on simple tasks but fails on structurally complex ones, revealing a key scalability weakness in current LLMs.

Task-Scale Sensitivity

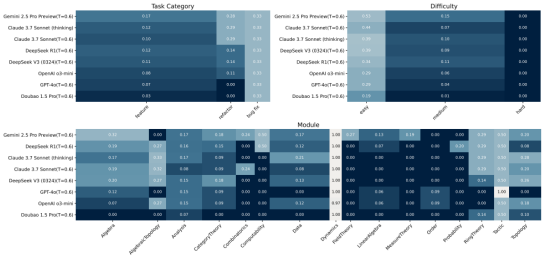


Figure 7 Heatmap of evaluation pass@16 success rates across task types, difficulty levels, and mathematical modules. Column widths represent the proportion of each class in the test set. Color intensity indicates model performance.

- Task Category: Models perform best on localized Bug Fix tasks but struggle with Feature Addition requiring global coherence.
- Difficulty Level: Performance drops steeply from Easy to Hard tasks, showing weak reasoning over complex dependencies.
- Mathematical Domain: Across domains, Dynamics and Tactic are easier, while Combinatorics, Measure Theory, and Set Theory remain challenging.

Semantic Robustness

Error Class	Description
Component Omission	Required definitions, proofs, or examples are missing
Incompleteness Issue	The edit is structurally partial, unfinished, or placeholder
Mathematical Error	The logic is invalid or inconsistent with Lean's formal semantics
Quality Defect	Redundant or unabstracted edits, or lack of documentation
Requirement Mismatch	Misalignment with the intended instruction

Table 3 Semantic error classes and their typical manifestations.

To investigate the gap between syntactic validity and semantic correctness, conduct a systematic analysis of failure cases and define five high-level semantic error classes, summarized in Table 3. It captures recurring patterns where models produce syntactically valid but semantically inadequate patches.

Failure Characterization

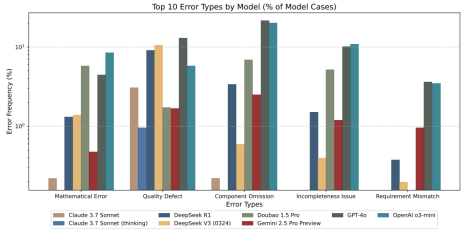


Figure 8 Distribution of semantic error types across models (log-scaled y-axis). Component omission, quality defects, and incompleteness are the most frequent error types. Claude Sonnet 3.7 (thinking) demonstrates consistently minimal error rates across all categories.

- Error Distribution: The most frequent errors are component omissions, quality defects, and incomplete edits.
- Model Differences: GPT-4o and o3-mini show the highest error rates, frequently producing ill-planned or misaligned outputs, while Claude Sonnet 3.7 (thinking) demonstrates the strongest semantic robustness.
- These failures largely stem from shallow reasoning and poor multi-step planning, revealing that type-checking alone cannot ensure true task success.

Judge Model Selection

While syntactic validity is objectively verifiable, semantic evaluation relies on subjective LLM judgment — raising concerns about potential bias when judges assess models from their own family.
Introduce Family Preference Index (FPI):

$$\text{FPI}_J = \mathbb{E}_{m \in M_{\text{same}}}[\Delta_{J,m}] - \mathbb{E}_{m \in M_{\text{other}}}[\Delta_{J,m}] \tag{2}$$

where $\Delta_{J,m}$ is the deviation between the judge's score and the majority consensus.
Positive FPI $\rightarrow$ Bias toward same-family models.
Zero or near-zero FPI $\rightarrow$ Neutral, unbiased evaluation.

Failure Characterization

Model Evaluated	Claude Thinking (%)		Gemini (%)		DeepSeek R1 (%)		DeepSeek V3 (%)		Majority Vote (%)
Gemini 2.5 Pro Preview	18.04	+0.55	16.20	-1.29	17.66	+0.17	18.05	+0.56	17.49
DeepSeek R1	12.55	-0.38	10.56	-2.37	13.80	+0.87	14.80	+1.87	12.93
Claude Sonnet 3.7 (thinking)	12.73	+0.31	11.47	-0.95	12.73	+0.31	12.73	+0.31	12.42
o3-mini	8.60	-2.22	5.94	-4.88	15.31	+4.49	13.42	+2.60	10.82
Claude Sonnet 3.7	10.83	+0.25	8.84	-1.74	11.33	+0.75	11.33	+0.75	10.58
DeepSeek V3	11.81	+0.53	9.44	-1.84	11.93	+0.65	11.92	+0.64	11.28
GPT-4o	6.73	-1.85	5.71	-2.87	13.31	+4.73	8.58	0.00	8.58
Doubao 1.5 Pro	2.98	-0.68	3.48	-0.18	4.08	+0.42	4.08	+0.42	3.66
Squared Deviation from Majority	3.31		78.02		233.61		81.69		-
Family Preference Index (FPI)	+0.95%		+2.12%		−1.16%		+1.10%		-

Table 4 Judge model comparison: Success rates across different LLM judges. Each judge column shows score and deviation from majority (colored: green for positive, red for negative). The bottom rows report total deviation from consensus (squared) and the Family Preference Index (FPI), which quantifies each judge's average relative bias toward its own model family.

Most LLM judges show slight but measurable family bias (within $\pm 2\%$). Gemini and DeepSeek V3 favor their own outputs, while DeepSeek R1 tends to rate non-family models higher. Claude Sonnet 3.7 (thinking) shows the lowest bias and highest consistency, making it the chosen primary evaluator.

Outline

Conclusion

- This paper introduces the Automated Proof Engineering (APE) paradigm and proposes APE-Bench I, the first large-scale benchmark based on real Mathlib4 proof workflows.
- Combine Lean compiler verification with LLM-based semantic judgment for rigorous evaluation.
- Reveals large performance gaps among state-of-the-art LLMs, especially in structurally complex proof-editing tasks.
- In the future, extend to APE-Bench II III for: Multi-file coordination, Project-scale verification, and Autonomous agent systems for full proof maintenance.

Thank you!