MATHLIBPR: Pull Request Merge-Readiness Benchmark for Formal Mathematical Libraries

Zixuan Xie* [1] **Xinyu Liu*** [1] **Shangtong Zhang** [1]

Abstract

The ecosystem of Lean and Mathlib has become the de facto standard for large language model (LLM) assisted formal reasoning with remarkable successes in recent years. Those successes, however, only consume Mathlib as an essential dependency but do not directly contribute to it. In the meantime, the growth of Mathlib has recently been bottlenecked by the review process, which requires human reviewers to judge whether proposed pull requests (PRs) follow the Mathlib's conventions and are worth integrating as part of a shared mathematical infrastructure. This leads to our central question: can LLMs help review Mathlib PRs? To this end, we introduce MATHLIBPR, a benchmark built from real Mathlib4 PR histories. We further propose a staged evaluation protocol and use it to evaluate both LLM models (e.g., DeepSeek, Qwen, Goedel, and Kimina) and LLM agents (e.g., Codex and Claude Code). Surprisingly, both LLM models and LLM agents struggle to distinguish merge-ready PRs from build-passing PRs that were revised or never merged. By turning Mathlib PR histories into a supervised signal, MATHLIBPR provides a step toward reviewer assistants and reward models that could help evaluate PRs and steer LLMs toward producing merge-ready Mathlib contributions.

1. Introduction

Mathlib (Mathlib-Community, 2020) has become the de facto standard for large language model (LLM) assisted formal reasoning. Built on Lean4 (Moura and Ullrich, 2021) , Mathlib is both a large formal library and an active open-source project, with reusable definitions, theorems, tactics, and APIs spanning diverse areas of mathematics.

Recent successes around this ecosystem of Mathlib and Lean4 span several levels. MiniF2F (Zheng et al., 2022) makes formal Olympiad-level problems a standard benchmark for LLM reasoning. AlphaProof (Hubert et al., 2026) demonstrates the power of LLM-based formal reasoning in competition mathematics. The Gauss agent (Math Inc., 2025a) and its strong Prime Number Theorem formalization (Math Inc., 2025b) show the scale at which LLM formalization projects are now being attempted. Various areas of machine learning theory have also been formalized in Lean, including optimization (Li et al., 2024), reinforcement learning (Zhang, 2025), and statistical machine learning (Sonoda et al., 2025; Zhang et al., 2026b)

These successes, however, only consume Mathlib as an essential dependency but do not directly and systematically contribute to it. This gap is precisely what surfaced in a recent Lean Zulip discussion around the LLM-assisted De Giorgi–Nash–Moser formalization (Lean Zulip, 2026). The discussion contrasts code that compiles and certifies a target theorem with code that is reusable, maintainable, stated at the right level of generality, and suitable for integration into Mathlib. Although the correctness of LLM-generated code can be checked by the Lean kernel, whether such code is merge-ready for Mathlib is a more complex judgment that requires human review. To our knowledge, the only LLM-based project that is specifically targeting the growth of Mathlib is MathlibLemma (Liu et al., 2026), which studies the automatic discovery and formalization of missing folklore lemmas and reports that a subset of its verified outputs has already been merged into Mathlib. However, Liu et al. (2026) also acknowledge that verified proofs require further human review before becoming mergeable Mathlib contributions.

Unfortunately, human review is difficult to scale. The Mathlib Initiative roadmap reports that Mathlib now contains over 1.9 million lines of formally verified mathematics written by more than 500 contributors (Initiative, 2026). It also identifies the review queue as the primary constraint on Mathlib's growth, with roughly 300 pull requests (PRs) in backlog and median wait times around two weeks (Initiative, 2026). Each PR requires human reviewers to judge whether the proposed code fits the surrounding library, follows Mathlib conventions, and is worth integrating as part

[1]Department of Computer Science, University of Virginia. Correspondence to: Shangtong Zhang <shangtong@virginia.edu>.

The 3^{rd} *AI for Math Workshop* at the 43^{rd} International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.

of a shared mathematical infrastructure. This leads to our central question: can LLMs help review Mathlib PRs?

To study this question, we introduce MATHLIBPR, a benchmark for evaluating whether LLMs can judge the merge-readiness of Mathlib PR snapshots. Each example is drawn from a real Mathlib4 PR and has already passed build checks. [1] `Merge-ready` examples correspond to snapshots accepted into Mathlib, while `not-merge-ready` examples correspond to build-passing snapshots that were later revised or never merged. Since every example already builds, the benchmark rules out trivial strategies based on syntax errors or compilation failure and focuses on reviewer-like judgment. Our contributions are threefold.

1. We construct MATHLIBPR from real Mathlib4 PR histories, yielding a benchmark for evaluating merge-readiness judgments on build-passing Mathlib contributions.
2. We propose a staged evaluation protocol that progressively supplies richer review-relevant context to evaluate LLMs on merge-readiness tasks.
3. We conduct an empirical study of both LLM models (e.g., DeepSeek (Guo et al., 2025), Qwen (Yang et al., 2025), Goedel (Lin et al., 2025), and Kimina (Wang et al., 2025)) and LLM agents (e.g., Codex (OpenAI, 2026) and Claude Code (Anthropic, 2026)) , showing that reviewer-like merge-readiness judgment remains difficult even when every snapshot already passes build checks.

More broadly, MATHLIBPR is intended as a step toward reviewer assistants and reward models for formal mathematical libraries. Such models could help evaluate human-written PRs, triage LLM-generated contributions, and steer future LLM models and agents toward generating Lean code that is not only correct, but also maintainable, integrated, and merge-ready for Mathlib.

2. Related Work

LLM-assisted formal theorem proving and benchmarks. Recent work has produced strong results in LLM-assisted formal theorem proving on Olympiad-level, undergraduate, and library-scale problems. Benchmarks such as MiniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), FIMO (Liu et al., 2023a), PutnamBench (Tsoukalas et al., 2024), and FormalMATH (Yu et al., 2025) measure whether systems can generate proofs that are accepted by a proof assistant. LeanDojo (Yang et al., 2023) extracts theorems and premises from Mathlib to support retrieval-augmented proving over a large formal library. Recent provers and agents, including DeepSeek-Prover (Xin et al., 2024), Goedel-Prover-V2 (Lin et al., 2025), Kimina-Prover (Wang et al., 2025), Seed-Prover (Chen et al., 2025), and AlphaProof (Hubert et al., 2026), demonstrate that proof assistants provide a reliable correctness signal for evaluating and improving formal proof generation. MATHLIBPR differs in what it asks of a system. Once a Mathlib PR snapshot already builds, the question is no longer whether the kernel accepts the proof, but whether the contribution is suitable for integration into a maintained formal library.

Automated code review and pull-request evaluation. Automated code review has been studied in mainstream software engineering. CodeReviewer (Li et al., 2022) formulates review tasks such as code-change quality estimation, review comment generation, and code refinement, trained on large-scale open-source review data. More recent benchmarks such as SWRBench (Zeng et al., 2025) and Sphinx (Zhang et al., 2026a) evaluate LLM-based PR review with project-level context and structured review criteria. Beyond review, SWE-bench (Jimenez et al., 2024) uses real GitHub issues and PRs to evaluate whether agents can edit a repository to resolve an issue. MATHLIBPR is closest to this line in spirit but differs in domain and signal. Its examples come from a formal mathematical library where every snapshot already passes build checks, and its task is to judge merge-readiness from code and immediate artifacts rather than from post-hoc review outcomes.

Library growth and contribution quality in Mathlib. Mathlib is both a corpus of verified theorems and a maintained open-source library whose contributions must remain reusable, documented, and coherent with surrounding APIs (Mathlib-Community, 2020). Prior work on maintaining Mathlib emphasizes that kernel checking is only one part of library quality: linters, documentation, review practices, and project-level conventions are needed to reduce maintainer burden and preserve library coherence (van Doorn et al., 2020; Baanen et al., 2025). Several recent systems contribute to Mathlib growth from the generation side. MathlibLemma (Liu et al., 2026) studies the automatic discovery and formalization of missing folklore lemmas, and reports that some generated lemmas have been upstreamed into Mathlib. Related systems synthesize useful lemmas or conjectures from proof traces, library contexts, or neuro-symbolic templates (Sivaraman et al., 2022; Onda et al., 2025; Alhessi et al., 2025), while other feedback-driven provers generate intermediate lemmas or subgoals as scaffolding for solving hard theorems (Dong et al., 2024; Zhou et al., 2025; Ospanov et al., 2025; Varambally et al., 2025). MATHLIBPR addresses the other side of library growth. Rather than generating new statements or proofs, it turns historical Mathlib review outcomes into a benchmark for

[1] We use the public Mathlib4 repository, `leanprover-community/mathlib4`, as the source of PR histories.

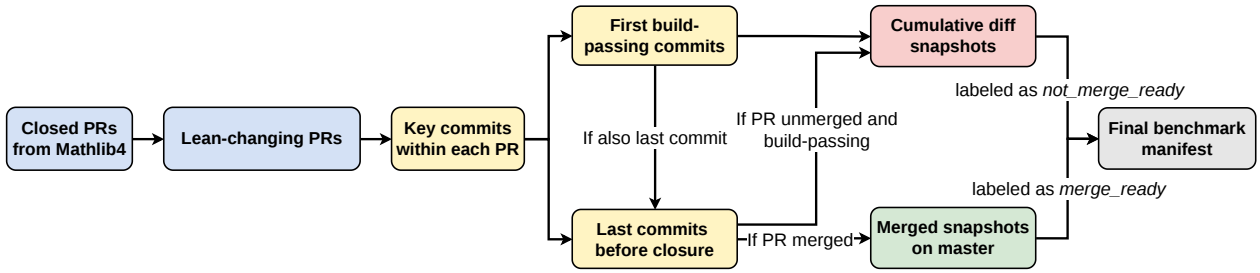

Figure 1. Overview of the MATHLIBPR dataset construction pipeline. Starting from closed PRs in the Mathlib4 repository, we retain PRs with Lean file changes. Historical build-check records identify build-passing commits on PR branches, including the first build-passing commit and, for unmerged PRs, the last build-passing commit before closure. These states provide `not_merge_ready` examples when they are later revised or never merged. For merged PRs, the `merge_ready` example is the version accepted into Mathlib4 on the master branch. We also apply a temporal cutoff so that positive and negative examples come from a comparable period of Mathlib4 history. Cumulative diffs are reconstructed for the selected snapshots, and the resulting labeled examples form the final MATHLIBPR manifest.

evaluating whether build-passing contributions are merge-ready.

3. Task Definition and Dataset Construction

3.1. Merge-Readiness of Build-Passing PR Snapshots

MATHLIBPR evaluates merge-readiness judgments for build-passing Mathlib4 PR snapshots. A snapshot represents the state of a PR at a selected commit, together with the reconstructed code changes for that state. Given this snapshot-level context, the task is to decide whether the selected state is ready to be accepted into Mathlib4. All selected snapshots pass the build checks by construction, and the benchmark deliberately withholds reviewer comments, discussion threads, and other post-hoc social signals that would otherwise leak the outcome. MATHLIBPR therefore probes the gap between build-passing code and merge-ready code, asking whether a system can make a reviewer-like judgment from the code and its immediate artifacts alone.

The dataset labels are binary. A snapshot is labeled `merge_ready` if it is the version accepted into Mathlib4, and `not_merge_ready` if it was either later revised before acceptance or belongs to a PR that was never accepted. The `not_merge_ready` label is snapshot-level: it indicates that the selected state was not merge-ready at that point in the PR history, not that the underlying PR could never be merged.

3.2. Dataset Construction Pipeline

We construct MATHLIBPR from closed PRs in the Mathlib4 repository; Figure 1 summarizes the pipeline. We start from closed PRs, retain those with Lean file changes, and use historical build-check records to recover build-passing states on PR branches. Because such records are not uniformly available throughout Mathlib4 history, they determine the period from which `not_merge_ready` examples can be reliably constructed.

For `not_merge_ready` examples, we draw two kinds of snapshots from PRs with recovered build-check records: (i) the first build-passing commit, when this state is later revised before acceptance; and (ii) the last commit before closure of an unmerged PR, when that commit is build-passing. In both cases, we reconstruct the snapshot's cumulative diff against the merge-base between the PR branch and its target Mathlib4 branch.

For `merge_ready` examples, we take the version accepted into Mathlib4 on the master branch. To keep the two classes drawn from a comparable period, we retain accepted snapshots only from the period beginning with the earliest retained `not_merge_ready` snapshot. When selected snapshots from the same PR coincide, we keep only one and assign its label by the rules above. The resulting labeled snapshots form the final benchmark manifest, whose fields we describe in Appendix A.

3.3. Dataset Statistics

We construct the current version of MATHLIBPR from closed PRs available in the Mathlib4 repository as of April 5, 2026. Table 1 summarizes the construction coverage and final benchmark composition. The final dataset contains 15,895 labeled build-passing snapshots from 12,063 PRs: 11,409 `merge_ready` snapshots and 4,486 `not_merge_ready` snapshots. Among merged PRs, the dataset includes 3,687 within-PR pairs, each consisting of an earlier build-passing snapshot and the version ultimately accepted into Mathlib4. These pairs enable a controlled within-PR analysis (Section 6.2) that tests whether models can recognize revision progress on the same underlying contribution. The benchmark is publicly available at `https://huggingface.co/datasets/MathlibPR/MathlibPR`, with a Croissant metadata file conforming to NeurIPS 2026 E&D requirements.

Table 1. Construction coverage and final benchmark composition for MATHLIBPR.

Component	Unit	Count
PR-level coverage		
Closed Mathlib4 PRs as of April 5, 2026	PRs	33,444
PRs with at least one Lean-file change	PRs	24,172
PRs represented in the final benchmark	PRs	12,063
Final benchmark composition		
Labeled build-passing snapshots	Snapshots	15,895
`merge_ready` snapshots	Snapshots	11,409
`not_merge_ready` snapshots	Snapshots	4,486
Within-PR pairs	Pairs	3,687

4. Experimental Setup

4.1. Evaluated Systems

We evaluate two classes of systems on MATHLIBPR: LLM models, which receive only the supplied prompt, and LLM agents, which can additionally inspect a read-only local repository checkout.

LLM models. We first evaluate LLM models on the full benchmark. The evaluated models are the open-weight reasoning models DeepSeek-R1-Distill-Qwen ("DeepSeek") and Qwen3-8B ("Qwen"); the library specialist Goedel-Prover-V2-32B ("Goedel"), a model pre-fine-tuned on Mathlib; and Kimina-Prover-Distill-8B ("Kimina"), a model trained via RL on formal competitions such as MiniF2F (Zheng et al., 2022).

LLM agents. We additionally evaluate coding agents. Beyond the prompt, these agents can inspect a read-only local checkout at the snapshot commit and use coding tools to gather evidence before making their judgment. Because this setting is substantially more expensive, we run it on a balanced subset of 500 `merge_ready` and 500 `not_merge_ready` examples rather than on the full benchmark.

We evaluate Codex via `codex exec` with GPT-5.4, and Claude Code via the local Claude CLI backed by Claude Sonnet 4.6. Each sample is processed in a fresh per-sample CLI session, so no state is carried across examples. Agents are restricted to read-only repository inspection: they cannot edit files, install dependencies, or access web resources, GitHub or PR pages, remote APIs, or git history. These restrictions are essential, because otherwise the task can be short-circuited by retrieving the known PR outcome instead of judging merge-readiness from the provided artifacts.

4.2. Three-Stage Evaluation Protocol

We evaluate LLM models and agents on MATHLIBPR under a three-stage protocol that supplies progressively richer evidence for each snapshot. All stages share the same review prompt contract and structured output schema; only the supplied evidence changes from Stage 1 to Stage 3. This design lets us ask not only whether current systems can make reviewer-like judgments, but also whether those judgments improve as richer review context is supplied.

Stage 1 provides code-local evidence for the snapshot: the diff itself, the changed-file context, and compact digests of relevant Mathlib review guidance on naming, style, and documentation. [2] Stage 2 adds structured diagnostics covering linting, imports, declaration placement, documentation, and API fit. These diagnostics largely come from the same commit-level GitHub check records used to identify build-passing states in Section 3.2, repurposed as review signals. For accepted snapshots merged into master, comparable historical diagnostics are not always recoverable in the same form. In those cases, we backfill linting diagnostics locally. Diagnostics are treated as evidence rather than as oracle labels: they may surface potentially review-relevant issues, but the system must still determine whether those issues actually block merge-readiness. Stage 3 further adds the PR title and description as stated intent.

The same staged restriction applies to LLM agents. Although they may inspect the local repository, our packaged diagnostics and PR intent are supplied only at Stages 2 and 3 respectively. Full prompt templates and input field definitions are deferred to Appendix B.

[2] See `https://leanprover-community.github.io/contribute/naming.html`, `https://leanprover-community.github.io/contribute/style.html`, and `https://leanprover-community.github.io/contribute/doc.html`.

4.3. Structured Output Schema

All systems are required to return a structured JSON review rather than free-form commentary. This follows prior LLM-evaluation work that uses explicit criteria, form-filling, or customized score rubrics to make judgments more structured, comparable, and aligned with human assessment (Liu et al., 2023b; Kim et al., 2024; Hashemi et al., 2024). The output includes an overall verdict, a scalar merge-readiness score, confidence fields, axis-level assessments along eight rubric axes, and brief supporting fields such as major strengths, blockers, and minimal required changes. The eight axes operationalize the concerns emphasized in Mathlib's reviewer and contributor documentation[3] into a structured code-centric rubric. Outputs are validated against a fixed schema. The eight axes and full schema are detailed in Appendix B.

An output that fails this validation is treated as invalid, meaning it cannot be parsed, violates the schema, omits required fields, or exceeds the per-sample runtime budget. Invalid outputs reflect interface or execution failures rather than reviewer decisions. A schema-valid output may still abstain at the overall-verdict level via `uncertain`, which reflects a substantive decision not to commit rather than a failure to produce a usable review.

4.4. Metrics

A system's output is either invalid or carries a verdict of `merge_ready`, `not_merge_ready`, or `uncertain`. We report four primary metrics on these outputs.

MR and NMR Recall. MR Recall is recall on `merge_ready` examples, and NMR Recall is recall on `not_merge_ready` examples. We count a prediction as correct only if the verdict matches the reference label. Outputs with verdict `uncertain` and invalid outputs are both counted as errors. Reporting the two recalls separately exposes class-asymmetric behavior that a single accuracy metric would hide.

Balanced accuracy. We also report balanced accuracy as the mean of the two recalls,

$$\text{Bal. Acc.} = \frac{\text{MR Recall} + \text{NMR Recall}}{2}.$$

A high MR Recall or NMR Recall on its own can mask poor performance on the other class, so balanced accuracy summarizes both in a single number.

[3]See Mathlib's PR review guide (`https://leanprover-community.github.io/contribute/pr-review.html`) along with the naming, style, and documentation guidelines linked in Section 4.2.

Valid-output rate. Valid-output rate is the fraction of examples for which the system returns a schema-valid output within the per-sample runtime budget. This metric is informative because some systems fail by timing out or producing malformed outputs rather than by committing to an incorrect verdict.

AUROC. We also report the area under the receiver operating characteristic curve (AUROC), computed from the scalar merge-readiness scores in schema-valid outputs. AUROC measures how well the system ranks `merge_ready` examples above `not_merge_ready` examples, independent of any discrete decision threshold. Because invalid outputs do not yield usable scores, AUROC should be read alongside the valid-output rate.

5. Main Results

Table 2 reports the main results for both LLM models and LLM agents. Because LLM models are evaluated on the full benchmark while LLM agents are evaluated on a balanced subset of 500 positive and 500 negative examples, numbers should not be compared directly across the two classes.

Several patterns emerge from Table 2. First, no system reliably distinguishes `merge_ready` from `not_merge_ready` snapshots. Even with the input restricted to build-passing snapshots, the best Stage 3 LLM-model balanced accuracy is only 36.0%, achieved by DeepSeek. Qwen reaches 24.1%, and Goedel and Kimina remain much lower. This indicates that MATHLIBPR probes a substantially harder capability than detecting syntax errors or compilation failure. The model must instead reason about the review norms by which Mathlib curates an integrated mathematical library.

Second, richer context yields only limited and uneven gains. Moving from Stage 1 to Stage 3 raises balanced accuracy for DeepSeek ($18.1\% \rightarrow 36.0\%$) and Qwen ($15.8\% \rightarrow 24.1\%$), but these gains are not matched by recognition of `not_merge_ready` snapshots. DeepSeek's Stage 3 NMR Recall is only 2.3%, and Qwen's NMR Recall remains 0.0% across all three stages. Other systems do not even show monotone gains. Goedel's valid rate collapses from 68.3% at Stage 1 to 13.5% at Stage 3. Even when local code context is supplemented by diagnostics and PR intent, the reviewer-like judgment required by Mathlib is not captured well by current systems.

Third, access to repository state does not close the gap. LLM agents can inspect a read-only local checkout and use coding tools, but they still operate in the same overall difficulty regime. Codex achieves high MR Recall (92.0% at Stage 3) and is operationally reliable (Stage 3 valid rate 99.9%), but its Stage 3 NMR Recall is only 12.0%. Claude

Model	Eval. set	Stage	AUROC	MR Recall	NMR Recall	Bal. Acc.	Valid rate
LLM models							
DeepSeek	Full	S1	40.5	32.6	3.5	18.1	98.3
DeepSeek	Full	S2	51.4	61.7	3.0	32.4	97.2
DeepSeek	Full	S3	52.4	69.8	2.3	36.0	92.9
Qwen	Full	S1	49.9	31.6	0.0	15.8	96.6
Qwen	Full	S2	49.7	38.7	0.0	19.3	96.2
Qwen	Full	S3	48.4	48.3	0.0	24.1	97.0
Goedel	Full	S1	30.0	0.1	0.3	0.2	68.3
Goedel	Full	S2	51.3	2.5	0.3	1.4	18.9
Goedel	Full	S3	53.1	1.7	0.2	1.0	13.5
Kimina	Full	S1	49.5	0.1	18.6	9.3	18.8
Kimina	Full	S2	51.9	0.0	8.2	4.1	9.5
Kimina	Full	S3	51.6	0.0	9.0	4.5	11.3
LLM agents							
Codex	500/500	S1	63.2	91.6	14.6	53.1	100.0
Codex	500/500	S2	58.9	91.4	9.4	50.4	100.0
Codex	500/500	S3	61.3	92.0	12.0	52.0	99.9
Claude Code	500/500	S1	68.8	29.8	14.0	21.9	48.8
Claude Code	500/500	S2	65.7	37.2	11.2	24.2	59.1
Claude Code	500/500	S3	66.0	38.0	12.0	25.0	56.4

Table 2. Main results for all evaluated systems. All values are percentages, with AUROC multiplied by 100 and computed on schema-valid outputs with usable scores. MR Recall and NMR Recall are recall on `merge_ready` and `not_merge_ready` examples, with `uncertain` and invalid outputs counted as errors. Balanced accuracy is their mean. Valid rate is the fraction of schema-valid outputs returned within the per-sample runtime budget. LLM models are evaluated on the full benchmark, whereas LLM agents are evaluated on a balanced 500/500 subset.

Code shows the opposite reliability profile, returning a valid output on only 56.4% of Stage 3 examples. In both cases, repository access does not help systems reliably identify `not_merge_ready` snapshots.

Behind these recall metrics, systems differ substantially in what they actually output. Table 3 reports the Stage 3 prediction distribution across all evaluated systems. We focus on Stage 3 because it provides the richest context and thus the clearest view of each system's residual behavior.

The four states are distributed very unevenly across systems, often clustering on a single state. DeepSeek and Codex commit to `merge_ready` on the large majority of examples while almost never committing to `not_merge_ready`. Qwen instead splits its outputs roughly evenly between `merge_ready` and `uncertain`, again almost never committing to `not_merge_ready`. Even Codex, the most operationally reliable system, commits to `not_merge_ready` on only 8.3% of examples.

Goedel, Kimina, and Claude Code instead concentrate on invalid outputs. To understand these failures, we examined their Stage 3 invalid outputs. All of Goedel's invalid cases are prose-style analyses, and all of Kimina's are Lean code blocks. In those invalid cases, the models do not produce the required reviewer-JSON output at all. Both Goedel and Kimina are trained for Lean theorem proving, and this pattern is consistent with a mismatch between their training objective and the structured review interface required by our benchmark. Claude Code's invalid cases differ. Its finalized invalid outputs are mostly malformed near-complete JSON, while the raw logs also show a large volume of timeout-driven failures, with 35.8% of Stage 3 latest-attempt outputs exceeding the per-sample runtime budget. These failure modes therefore reflect interface or budget constraints rather than valid reviewer judgments. Representative invalid outputs are shown in Appendix C.5.

6. Analysis and Error Study

We now characterize when systems abstain via `uncertain`, and test whether they can recognize revision progress within the same PR.

6.1. Abstention Patterns

Abstention via `uncertain` is not a single uniform behavior. Table 3 shows that Stage 3 abstention rates range from 0.0% (Kimina) to 48.7% (Qwen), with nearly half of Qwen's outputs being abstentions. To understand what drives these abstentions, we report in Table 4 which rubric axes each system flags as concerning when it abstains (axes defined in Appendix B).

The dominant axes also differ across systems, in ways that track their inspection capability. DeepSeek and Qwen abstain overwhelmingly on documentation (75.8% and 83.0% of their abstentions, respectively), with proof readability a

Model	Eval. set	Pred. MR	Pred. NMR	Pred. uncertain	Invalid
LLM models, Stage 3					
DeepSeek	Full	68.9	2.1	21.9	7.1
Qwen	Full	48.3	0.0	48.7	3.0
Goedel	Full	1.5	0.2	11.8	86.5
Kimina	Full	0.0	11.3	0.0	88.7
LLM agents, Stage 3					
Codex	500/500	87.5	8.3	4.1	0.1
Claude Code	500/500	30.4	8.5	17.5	43.6

Table 3. Stage 3 prediction breakdown for all evaluated systems. Values are percentages over the relevant evaluation set and may not sum to exactly 100 due to rounding.

Model	Doc.	Naming	Local	File	Imports	Proof	API fit	Overlap
LLM models, Stage 3								
DeepSeek	75.8	3.6	1.1	0.5	1.4	28.3	2.4	2.6
Qwen	83.0	3.6	1.1	0.2	1.4	16.8	0.9	0.8
Goedel	69.1	17.1	23.1	5.1	17.7	50.2	22.7	17.1
LLM agents, Stage 3								
Codex	75.6	58.5	34.1	0.0	31.7	58.5	0.0	29.3
Claude Code	85.7	44.0	46.3	5.7	8.6	28.6	24.0	5.7

Table 4. Concern categories among valid `uncertain` outputs in Stage 3. Each cell reports the percentage of a system's valid `uncertain` outputs that flag the corresponding axis as `concern` or `blocker`. Categories are not mutually exclusive. Kimina is omitted because its Stage 3 run contains only one valid `uncertain` output.

distant second. Both axes are visible from the diff alone, consistent with LLM models that judge from the prompt without further inspection. Goedel, pre-fine-tuned on Mathlib, also abstains heavily on documentation but additionally raises proof readability concerns on roughly half of its abstentions. This suggests it brings more internalized knowledge of Mathlib's proof style. The LLM agents behave differently. Codex and Claude Code spread their abstentions across more axes, including naming, local structure, imports, and overlap. These axes benefit from inspecting the surrounding code rather than the diff in isolation. This contrast suggests that part of the abstention behavior reflects what each system can observe, not just what it judges to matter.

6.2. Within-PR Comparison

We next test whether systems can recognize revision progress inside the same PR. We construct pairs by taking an earlier build-passing snapshot and the final accepted snapshot from the same merged PR. For each pair, let s_{early} and s_{final} be the system's scalar merge-readiness scores for the two snapshots, as defined in the schema (Appendix B). We score the pair as correct if $s_{\text{final}} > s_{\text{early}}$, incorrect if $s_{\text{final}} < s_{\text{early}}$, and half-correct if the two scores tie. Pairwise accuracy averages this 0/0.5/1 score over usable pairs, where a pair is usable if both snapshots received a schema-valid output. Pairwise accuracy thus measures whether the system ranks the accepted final snapshot above the earlier one, not whether it predicts the correct binary label. Table 5 reports this metric for LLM models. We restrict the analysis to LLM models because the LLM agents are evaluated on a balanced 500/500 subset, which yields fewer than 15 paired PRs per stage and is too small for stable pairwise estimates.

LLM models barely beat chance on this controlled comparison. DeepSeek improves from Stage 1 to Stage 2 but plateaus at Stage 3, remaining close to chance. Qwen stays at chance across all stages. Goedel and Kimina retain too few usable pairs for their pairwise numbers to be informative. This pairwise comparison strengthens the main result. Failing to distinguish `merge_ready` from `not_merge_ready` across the full benchmark could in principle reflect only that build-passing PRs all look broadly similar to a given system. The within-PR comparison removes that confound, since both snapshots are from the same PR, and current models still cannot reliably tell which version was accepted.

7. Discussion and Limitations

Limitations. The benchmark covers a selective slice of closed PRs. `not_merge_ready` snapshots require recoverable PR-side build-check evidence, while `merge_ready` snapshots are recovered from accepted master-side commits. Combined with our temporal cutoff, this means that MATHLIBPR does not exhaustively represent all closed PR activity in Mathlib's history. The labels themselves are

Model	Stage	Total pairs	Usable pairs	Pairwise Acc.	Mean Δ score
LLM models					
DeepSeek	S1	3687	3563	38.0	-0.091
DeepSeek	S2	3687	3467	52.5	0.019
DeepSeek	S3	3687	3101	51.1	0.006
Qwen	S1	3687	3460	50.4	0.004
Qwen	S2	3687	3442	51.3	0.009
Qwen	S3	3687	3479	49.1	-0.002
Goedel	S1	3687	610	28.4	-0.106
Goedel	S2	3687	151	50.3	0.001
Goedel	S3	3687	101	49.0	-0.006
Kimina	S1	3687	297	49.7	-0.003
Kimina	S2	3687	90	48.9	-0.009
Kimina	S3	3687	99	49.0	0.008

Table 5. Within-PR pairwise analysis for LLM models on the full benchmark, restricted to merged PRs with both an earlier build-passing snapshot and the final accepted snapshot. Pairwise Acc. counts ties as half-correct. Mean Δ score is the average difference between the final and earlier snapshots' merge-readiness scores.

also not a perfect oracle. They are derived from historical maintainer decisions, and some unmerged outcomes may reflect non-technical factors such as timing, author abandonment, or reviewer availability rather than technical defects alone. Finally, LLM-agent evaluation is currently limited to a balanced 500/500 subset because of cost, which reduces direct comparability with full-benchmark model results and limits some downstream analyses such as the within-PR comparison in Section 6.2.

Implications and future work. The goal of MATHLIBPR is not to replace Mathlib reviewers. It provides a benchmark for studying whether models can approximate part of the review signal that arises after code already passes formal checking, used as an advisory signal audited by human reviewers rather than as a basis for automatic PR decisions. We see several concrete directions for using and extending it. One is reviewer assistance and PR triage, where systems may help surface likely blockers or prioritize attention without making final decisions. Another is training reward models for merge-ready code. The within-PR pair structure provides a natural preference signal, since a capable system should prefer the final accepted snapshot over an earlier build-passing but not-yet-accepted snapshot from the same PR. The benchmark also admits a prospective extension over time. Because the current release is defined by a historical cutoff, later-closed PRs and currently open PRs can support forward-looking evaluations that reduce overfitting to absorbed repository history.

8. Conclusion

We introduced MATHLIBPR, the first benchmark for evaluating whether LLMs can review Mathlib PRs. Built from real Mathlib4 PR histories, the benchmark turns historical maintainer decisions into a supervised signal that probes the gap between build-passing code and merge-ready code. We further proposed a three-stage evaluation protocol that progressively supplies review-relevant context, and used it to evaluate both LLM models and LLM agents. Across all evaluated systems, none reliably distinguishes `merge_ready` from `not_merge_ready` snapshots, and neither richer context nor repository access closes this gap. As the LLM-assisted formal mathematics ecosystem moves from consuming Mathlib to contributing back to it, judging whether generated code is merge-ready becomes the next bottleneck. MATHLIBPR provides a step toward reviewer assistants and reward models that can help close this bottleneck.

References

Yousef Alhessi, Sólrún Halla Einarsdóttir, George Granberry, Emily First, Moa Johansson, Sorin Lerner, and Nicholas Smallbone. Lemmanaid: Neuro-symbolic lemma conjecturing. *arXiv preprint*, 2025.

Anthropic. Claude code by anthropic. https://www.anthropic.com/product/claude-code, 2026. Accessed: 2026-04-30.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint*, 2023.

Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang, and Damiano Testa. Growing mathlib: Maintenance of a large scale mathematical library. *arXiv preprint*, 2025.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint*, 2025.

Kefan Dong, Arvind Mahankali, and Tengyu Ma. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv Preprint*, 2025.

Helia Hashemi, Jason Eisner, Corby Rosset, Benjamin Van Durme, and Chris Kedzie. LLM-rubric: A multidimensional, calibrated approach to automated evaluation of natural language texts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2026.

Mathlib Initiative. The mathlib initiative year 1 roadmap. https://mathlib-initiative.org/roadmap/, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, and Minjoon Seo. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Lean Zulip. AI-authored projects: De Giorgi–Nash–Moser. https://leanprover.zulipchat.com/#narrow/channel/583339-AI-authored-projects/topic/De.20Giorgi.E2.80.93Nash.E2.80.93Moser, 2026.

Chenyi Li, Ziyu Wang, Wanyi He, Yuxuan Wu, Shengyang Xu, and Zaiwen Wen. Formalization of complexity analysis of the first-order algorithms for convex optimization. *arXiv preprint*, 2024.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *ArXiv Preprint*, 2025.

Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, Haiming Wang, Wei Ju, Chuanyang Zheng, Yichun Yin, Lin Li, et al. Fimo: A challenge formal dataset for automated theorem proving. *arXiv preprint*, 2023a.

Xinyu Liu, Zixuan Xie, Amir Moeini, Claire Chen, Shuze Daniel Liu, Yu Meng, Aidong Zhang, and Shangtong Zhang. Mathliblemma: Folklore lemma generation and benchmark for formal mathematics. In *Proceedings of the International Conference on Machine Learning*, 2026.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023b.

Math Inc. Introducing Gauss, an agent for autoformalization. https://www.math.inc/gauss, 2025a.

Math Inc. Strong PNT. https://math-inc.github.io/strongpnt/, 2025b.

The Mathlib-Community. The lean mathematical library. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, 2021.

Naoto Onda, Kazumi Kasaura, Yuta Oriike, Masaya Taniguchi, Akiyoshi Sannai, and Sho Sonoda. Leanconjecturer: Automatic generation of mathematical conjectures for theorem proving. *arXiv preprint*, 2025.

OpenAI. Codex cli. https://developers.openai.com/codex/cli, 2026. Accessed: 2026-04-30.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated llm and lean collaboration for advanced formal reasoning. In *Advances in Neural Information Processing Systems*, 2025.

Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. Data-driven lemma synthesis for interactive proofs. *Proceedings of the ACM on Programming Languages*, 2022.

Sho Sonoda, Kazumi Kasaura, Yuma Mizuno, Kei Tsukamoto, and Naoto Onda. Lean formalization of generalization error bound by rademacher complexity. *ArXiv Preprint*, 2025.

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. *arXiv preprint*, 2024.

Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics. In *Intelligent Computer Mathematics*, 2020.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. *arXiv preprint*, 2025.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *ArXiv Preprint*, 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *ArXiv Preprint*, 2025.

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. In *Advances in Neural Information Processing Systems*, 2023.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, et al. Formalmath: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint*, 2025.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. Benchmarking and studying the llm-based code review. *arXiv preprint*, 2025.

Daoan Zhang, Shuo Zhang, Zijian Jin, Jiebo Luo, Shengyu Fu, and Elsie Nallipogu. Sphinx: Benchmarking and modeling for llm-driven pull request review. *arXiv preprint*, 2026a.

Shangtong Zhang. Towards formalizing reinforcement learning theory. *arXiv preprint*, 2025.

Yuanhe Zhang, Jason D. Lee, and Fanghui Liu. Statistical learning theory in lean 4: Empirical processes from scratch. *arXiv preprint*, 2026b.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022.

Yichi Zhou et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint*, 2025.

A. Benchmark Access and Manifest

A.1. Access

MATHLIBPR is hosted on Hugging Face at https://huggingface.co/datasets/MathlibPR/MathlibPR. The benchmark is derived from the public leanprover-community/mathlib4 repository and the Mathlib contribution documentation, both distributed under Apache-2.0. The derived benchmark and its evaluation harness are released under Apache-2.0 as well. The release includes a Croissant metadata file with both core dataset metadata and Responsible AI fields, conforming to NeurIPS 2026 E&D track requirements. Full field documentation, loading code, prompt materialization utilities, scoring scripts, and reconstruction scripts are maintained in the dataset card.

A.2. Manifest Schema Overview

Each MATHLIBPR example is represented by a compact source-manifest record that stores the snapshot's identity, provenance, and reconstruction metadata. Table 6 groups the manifest fields by role.

Field group	Example fields	Scope
Identity	sample_id, pr_number, target_merged, snapshot_role	record
Snapshot reconstruction	snapshot_commit_sha, diff_base_sha, diff_source	record
Selection provenance	source_commit_seq, source_commit_sha, selection_diff_path	record
Pairing metadata	negative_commit_seq, final_commit_seq, last_commit_seq, final_snapshot_source, merge_commit_source	record
Diagnostics	linter_diagnostics, import_diagnostics, location_diagnostics, doc_diagnostics, api_diagnostics	S2–3
PR intent	pr_title, pr_description	S3

Table 6. Manifest field groups for a single MATHLIBPR record. The Scope column indicates whether a group is record-level metadata or stage-gated evidence shown to the system at the indicated stages. The dataset card lists the complete field set with per-field types and semantics.

The identity fields uniquely identify the snapshot and carry its binary reference label target_merged along with the snapshot type recorded in snapshot_role, such as first_build_success_snapshot or final_snapshot. The snapshot reconstruction fields specify which repository state is evaluated and the base against which the cumulative diff is computed. The selection provenance fields record which PR-side event or commit produced the selected snapshot, supporting reproducibility audits. The pairing metadata fields link an early build-passing snapshot to the corresponding final accepted or final unmerged commit when applicable, and are used to assemble the within-PR pair manifest. The diagnostics fields hold the recovered or backfilled lint, import, location, documentation, and API check results that Stage 2 and Stage 3 prompts incorporate as evidence. The PR intent fields hold the PR title and description, exposed only at Stage 3.

A.3. Illustrative Record

The following example shows a not_merge_ready record (target_merged = 0) for the first build-passing snapshot of an unmerged PR. Some provenance, pairing, and diagnostics fields are omitted for brevity.

```
{
  "sample_id": "prXXXXX_neg_first_build_success_snapshot_seqK_<sha12>",
  "pr_number": XXXXX,
  "target_merged": 0,
  "snapshot_role": "first_build_success_snapshot",
  "diff_source": "cumulative_pr_snapshot",
  "snapshot_commit_sha": "<snapshot_commit_sha>",
  "diff_base_sha": "<diff_base_sha>",
  "source_commit_seq": K,
  "selection_diff_path": "commits/XXXXX_K_<sha7>.diff",
  "final_snapshot_source": "last_pr_commit_unmerged",
  "pr_title": "<title or null>",
  "pr_description": "<description or null>"
}
```

For an accepted positive snapshot, `target_merged` is 1, `snapshot_role` is typically `final_snapshot` or `single_build_success_commit_snapshot`, and `diff_source` is `merged_commit_patch`. The release also includes a separate within-PR pair manifest with `pair_id`, `earlier_sample_id`, and `final_sample_id` fields linking the earlier build-passing snapshot to the final accepted snapshot of the same PR.

A.4. Agent Subset Sampling

We evaluate LLM agents on a balanced 500/500 subset rather than the full benchmark because the agent setting is substantially more expensive than prompt-only evaluation. The subset is sampled with a fixed random seed (`seed=42`) and contains 500 `merge_ready` and 500 `not_merge_ready` examples selected from the full benchmark.

We use separate released subsets for each stage, since the stage-specific benchmark inputs are materialized separately and the finalized agent-stage evaluations are tracked separately in the release. Stage 1 uses code-local context only. Stage 2 adds diagnostics. Stage 3 adds both diagnostics and PR title or description as stated intent. Subset memberships overlap substantially across stages but are not identical. The released benchmark therefore provides all six stage-by-agent subsets as separate dataset configs for direct reuse.

B. Prompt Templates and Output Schema

B.1. Prompt Design Overview

All stages share a common reviewer-oriented contract: given a build-passing Mathlib4 PR snapshot, the system must judge whether the snapshot is `merge_ready` or `not_merge_ready`, or abstain via `uncertain`. The contract instructs the system to make a code-centric judgment, ignore CI status, authorship, and merge outcome, and refrain from external retrieval. Table 7 summarizes which evidence is available at each stage; the stage-specific materializations are given in the prompt templates below.

Stage	Diff	Changed files	Guidelines	Diagnostics	PR intent
Stage 1	✓	✓	✓	–	–
Stage 2	✓	✓	✓	✓	–
Stage 3	✓	✓	✓	✓	✓

Table 7. Staged prompting protocol for MATHLIBPR. All stages use the same reviewer contract and JSON output schema. Stage 1 provides code-local evidence and compact guideline digests. Stage 2 adds automated diagnostics. Stage 3 adds PR title and description as stated intent. The exact prompts are given below.

B.2. Output Schema

All systems return a structured JSON review that is validated against a fixed schema. LLM models and LLM agents share the same core schema, with one additional required field (`repo_checks_used`) included for LLM agents to record the repository inspections used in reaching the judgment. We describe the top-level fields, the verdict semantics, the eight rubric axes, and the axis label values in turn.

B.2.1. TOP-LEVEL FIELDS

Each output JSON object contains the following shared top-level fields:

- `verdict`: the system's overall judgment of the snapshot, taking values `merge_ready`, `not_merge_ready`, or `uncertain`.
- `p_merge_ready`: a scalar merge-readiness score, interpreted on $[0, 1]$, that we use for ranking-based metrics such as AUROC and within-PR pairwise accuracy.
- `overall_confidence`: the system's stated confidence in its own verdict.
- `axes`: an object containing per-axis assessments along the eight rubric axes defined in Appendix B.2.3.

- top_strengths: a short list of strengths identified in the snapshot.
- top_blockers: a short list of issues that the system considers blockers to merge.
- minimal_required_changes: a short list of changes the system would request before merge.
- other_concerns: additional concerns that do not rise to the level of top blockers.

The complete core schema used for LLM model runs is given below.

```
{
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "verdict": {
      "type": "string",
      "enum": ["merge_ready", "not_merge_ready", "uncertain"]
    },
    "p_merge_ready": {"type": "number"},
    "overall_confidence": {"type": "number"},
    "axes": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "naming_style": {"$ref": "#/$defs/axis"},
        "documentation": {"$ref": "#/$defs/axis"},
        "local_structure": {"$ref": "#/$defs/axis"},
        "file_placement": {"$ref": "#/$defs/axis"},
        "imports_dependencies": {"$ref": "#/$defs/axis"},
        "proof_readability": {"$ref": "#/$defs/axis"},
        "api_library_fit": {"$ref": "#/$defs/axis"},
        "repository_overlap_generality": {"$ref": "#/$defs/axis"}
      },
      "required": [
        "naming_style",
        "documentation",
        "local_structure",
        "file_placement",
        "imports_dependencies",
        "proof_readability",
        "api_library_fit",
        "repository_overlap_generality"
      ]
    },
    "top_strengths": {
      "type": "array",
      "items": {"type": "string"}
    },
    "top_blockers": {
      "type": "array",
      "items": {"type": "string"}
    },
    "minimal_required_changes": {
      "type": "array",
      "items": {"type": "string"}
    },
    "other_concerns": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": [
    "verdict",
    "p_merge_ready",
    "overall_confidence",
```

```
      "axes",
      "top_strengths",
      "top_blockers",
      "minimal_required_changes",
      "other_concerns"
    ],
    "$defs": {
      "axis": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
          "label": {
            "type": "string",
            "enum": ["good", "concern", "blocker", "unknown"]
          },
          "confidence": {"type": "number"},
          "evidence": {
            "type": "array",
            "items": {"type": "string"},
            "maxItems": 3
          }
        },
        "required": ["label", "confidence", "evidence"]
      }
    }
  }
```

For LLM agents, the schema additionally requires `repo_checks_used`, a list of inspected files, paths, or search patterns used during read-only repository inspection:

```
"repo_checks_used": {
  "type": "array",
  "items": {"type": "string"}
}
```

with `"repo_checks_used"` appended to the top-level `required` list.

B.2.2. VERDICT SEMANTICS

The `verdict` field has three possible values:

- `merge_ready`: the system judges that the snapshot is ready to be accepted into Mathlib4.
- `not_merge_ready`: the system judges that the snapshot should not be accepted in its current form.
- `uncertain`: the system declines to issue either judgment. We treat `uncertain` as a substantive abstention rather than as an invalid output.

The reference label is binary, so `uncertain` predictions are counted as errors for class-recall and balanced-accuracy calculations.

B.2.3. AXIS DEFINITIONS

The `axes` object contains one assessment for each of the following eight rubric axes:

- `naming_style`: whether declaration names, local names, notation, and naming patterns follow Mathlib naming and style conventions.
- `documentation`: whether the snapshot provides adequate docstrings, module-level documentation, comments, and other explanatory text expected for public-facing Mathlib contributions.

- `local_structure`: whether the code is well organized within the touched files, including declaration ordering, namespace and section structure, and local maintainability.
- `file_placement`: whether the changed declarations are located in the appropriate file, module, or part of the repository rather than belonging elsewhere.
- `imports_dependencies`: whether imports and dependencies are appropriate and minimal, avoiding unnecessary coupling or misplaced dependency choices.
- `proof_readability`: whether proofs are clear, idiomatic, and maintainable, rather than opaque, brittle, or unnecessarily difficult to review.
- `api_library_fit`: whether the contribution integrates well with the surrounding Mathlib API, abstraction boundaries, and existing library design patterns.
- `repository_overlap_generality`: whether the proposed declarations are sufficiently general and not redundant with existing or more general results in the repository.

Each axis assessment consists of three fields: `label`, `confidence`, and `evidence`. The `evidence` field is a list of up to three short strings, each citing a specific concern or strength observed in the snapshot.

B.2.4. AXIS LABEL SEMANTICS

Each axis label takes one of four values:

- `good`: the system finds no material concern on this axis.
- `concern`: the system identifies an issue on this axis that may merit reviewer attention but is not by itself a blocker.
- `blocker`: the system identifies an issue on this axis that it considers a blocker to merge.
- `unknown`: the available evidence is insufficient for the system to make a confident axis-level judgment. We reserve `unknown` for axis-level evidence insufficiency, distinct from the verdict-level abstention `uncertain`.

B.3. Prompt Templates for LLM Models

The LLM model setting follows the standard chat-completion message format with two messages per evaluation. The system message fixes the reviewer contract and is identical across all stages and samples. The user message is stage-specific and instantiated per sample by filling in the diff, changed files, diagnostics, and PR intent placeholders. We give the system message once below, then show the Stage 1 user message in full, followed by the incremental additions for Stage 2 and Stage 3.

System prompt.

```
You are reviewing a Mathlib4 PR snapshot for code-centric merge-readiness.

Use only the supplied context. Ignore CI/build status, workflow or bors state,
commit order, timeline, author/reviewer identity, and eventual merge outcome.
Do not use repository access, web search, or external tools.

Rubric axes:
- naming_style
- documentation
- local_structure
- file_placement
- imports_dependencies
- proof_readability
- api_library_fit
- repository_overlap_generality
```

```
Axis labels: good, concern, blocker, unknown.
Use unknown only for axes with insufficient evidence.
Use uncertain only for the final verdict.

Return only the required JSON object. Keep evidence short and concrete.
```

Stage 1 user prompt.

```
<task>
Estimate whether this snapshot is merge-ready for Mathlib4 using only the
supplied evidence.
</task>

<context>
<diff_chunks>
{diff_chunks}
</diff_chunks>

<naming_guidelines_markdown>
{naming_md}
</naming_guidelines_markdown>

<style_guidelines_markdown>
{style_md}
</style_guidelines_markdown>

<documentation_guidelines_markdown>
{doc_md}
</documentation_guidelines_markdown>

<changed_files_complete>
{changed_files_complete}
</changed_files_complete>

<first_order_import_files>
{first_order_import_files}
</first_order_import_files>
</context>

<instructions>
1. Use the rubric axes as the checklist.
2. Use the naming/style/doc digests as authoritative written guidance.
3. Use changed files and first-order imports to judge local conventions and
   code quality.
4. Use unknown instead of guessing when evidence is insufficient.
5. Output only the required JSON object.
</instructions>
```

Stage 2 user prompt. The Stage 2 user prompt extends the Stage 1 prompt by inserting a `<diagnostics>` block inside `<context>`, and replacing the third item in `<instructions>` with one that mentions diagnostics. The new block is:

```
<diagnostics>
{linter_diagnostics}
{import_diagnostics}
{location_diagnostics}
{doc_diagnostics}
{api_diagnostics}
</diagnostics>
```

The updated instruction reads:

```
3. Treat diagnostics as evidence, not as automatically decisive.
```

All other content is identical to Stage 1.

Stage 3 user prompt. The Stage 3 user prompt extends the Stage 2 prompt by adding a <pr_intent> block at the end of <context>, and replacing the third instruction with one that mentions PR intent. The new block is:

```
<pr_intent>
title: {pr_title}
description:
{pr_description}
</pr_intent>
```

The updated instruction reads:

```
3. Treat the PR title/description only as stated intent; code and diagnostics control the
↪ judgment.
```

All other content is identical to Stage 2.

B.4. Prompt Templates for LLM Agents

The stage-specific user-message templates are shared between Codex and Claude Code. The system messages share a common reviewer contract; Claude Code additionally inlines backend-specific tool and output-format rules and the agent JSON schema. The Codex backend receives the schema out-of-band via the structured-output API rather than inline in the system message.

Codex system message.

```
You are reviewing a mathlib4 PR snapshot for code-centric merge-readiness.

Use the supplied context and read-only repository inspection only.
Ignore CI/build status, workflow or bors state, commit order, timeline, author/reviewer
↪ identity, and eventual merge outcome.
Do not use web search, GitHub or mathlib4 PR pages, git history, remote APIs, or external
↪ tools beyond the allowed read-only repository checks.
Do not fetch network content with a browser, curl, wget, Python requests, or similar
↪ tools.

Rubric axes:
- naming_style
- documentation
- local_structure
- file_placement
- imports_dependencies
- proof_readability
- api_library_fit
- repository_overlap_generality

Axis labels: good, concern, blocker, unknown.
Use unknown only for axes with insufficient evidence.
Use uncertain only for the final verdict.

Return only the required JSON object. Keep evidence short and concrete.
```

Claude Code system message. The Claude Code system message begins with the same body as the Codex system message above and appends the following backend-specific section:

```
Additional rules for this Claude Code run:
- Inspect the repository snapshot using the provided read-only tools (Read, Glob, and
↪ restricted Bash).
- Do not edit files, create files, install dependencies, or access any network or web
↪ resources.
```

```
- Do not open GitHub, mathlib4 PR pages, issue pages, or any remote webpage or API.
- Do not use git history commands (git log, git blame, git show, etc.).
- Return ONLY the final JSON object matching the schema below. No markdown, no
↪ explanation, no code fences.

Required output JSON schema:
<!-- the inlined schema is identical to the agent schema given in Appendix B.2 -->
```

Stage 1 user message. The Stage 1 user message is shared between Codex and Claude Code. Two elements are agent-specific: the <task> block frames the read-only repository as additional evidence beyond the supplied local context, and the <repo_workflow> block (paste verbatim in Appendix B.5) prescribes how the system should use this access. We replace the inlined <repo_workflow> body with a placeholder here to avoid duplication.

```
<task>
Estimate whether this snapshot is merge-ready for mathlib4.
You are inside a read-only repository snapshot corresponding to this PR snapshot.
Use the repository only to gather code-centric evidence that is not visible from the
provided local context.
</task>

<context>
<diff_chunks>
{diff_chunks}
</diff_chunks>

<naming_guidelines_markdown>
{naming_md}
</naming_guidelines_markdown>

<style_guidelines_markdown>
{style_md}
</style_guidelines_markdown>

<documentation_guidelines_markdown>
{doc_md}
</documentation_guidelines_markdown>

<changed_files_complete>
{changed_files_complete}
</changed_files_complete>

<first_order_import_files>
{first_order_import_files}
</first_order_import_files>
</context>

<repo_workflow>
<!-- see Appendix B.5 -->
</repo_workflow>

<instructions>
Return only the required JSON object.
If desired, populate repo_checks_used with inspected paths or search patterns.
</instructions>
```

Stage 2 user message. Stage 2 reuses the Stage 1 user message, inserts the following <diagnostics> block inside <context>, and replaces the Stage 1 <repo_workflow> block with the Stage 2 variant in Appendix B.5. The <instructions> block is unchanged.

```
<diagnostics>
{linter_diagnostics}
{import_diagnostics}
```

```
{location_diagnostics}
{doc_diagnostics}
{api_diagnostics}
</diagnostics>
```

Stage 3 user message. Stage 3 reuses the Stage 2 user message, inserts the following <pr_intent> block inside <context>, and replaces the Stage 2 <repo_workflow> block with the Stage 3 variant in Appendix B.5. The <instructions> block is unchanged.

```
<pr_intent>
title: {pr_title}
description:
{pr_description}
</pr_intent>
```

B.5. Repository Workflow Block

Codex and Claude Code share the same <repo_workflow> templates. The block varies by stage because Stage 2 adds diagnostics as evidence and Stage 3 additionally introduces PR intent as stated scope.

Stage 1 <repo_workflow> block.

```
<repo_workflow>
1. Start from the provided diff and changed files.
2. Inspect touched declarations and nearby declarations in the same files.
3. Search the repository snapshot for:
   - possible duplicate or more general existing results
   - naming analogues in the same namespace or nearby file families
   - plausible file/module homes for the changed declarations
4. Use read-only repository tools only.
5. Do not edit files or run build, test, lint, or CI-like commands.
6. Do not use git history, workflow metadata, internet access, GitHub/mathlib4 PR pages,
↪ or any remote API.
7. Use unknown instead of guessing when repo-wide evidence remains insufficient.
</repo_workflow>
```

Stage 2 <repo_workflow> block.

```
<repo_workflow>
1. Start from the provided diff, changed files, and diagnostics.
2. Inspect touched declarations and nearby declarations in the same files.
3. Search the repository snapshot for:
   - possible duplicate or more general existing results
   - naming analogues in the same namespace or nearby file families
   - plausible file/module homes for the changed declarations
4. Treat diagnostics as evidence, but verify them against the code and repository context.
5. Use read-only repository tools only.
6. Do not edit files or run build, test, lint, or CI-like commands beyond the supplied
↪ diagnostics.
7. Do not use git history, workflow metadata, internet access, GitHub/mathlib4 PR pages,
↪ or any remote API.
8. Use unknown instead of guessing when repo-wide evidence remains insufficient.
</repo_workflow>
```

Stage 3 <repo_workflow> block.

```
<repo_workflow>
1. Start from the provided diff, changed files, diagnostics, and PR intent.
2. Use PR title/description only to interpret intended scope and expected
↪ API/documentation surface.
```

```
3. Inspect touched declarations and nearby declarations in the same files.
4. Search the repository snapshot for:
   - possible duplicate or more general existing results
   - naming analogues in the same namespace or nearby file families
   - plausible file/module homes for the changed declarations
5. Treat diagnostics as evidence, but verify them against the code and repository context.
6. Use read-only repository tools only.
7. Do not edit files or run build, test, lint, or CI-like commands beyond the supplied
↪ diagnostics.
8. Do not use git history, workflow metadata, internet access, GitHub/mathlib4 PR pages,
↪ or any remote API.
9. Use unknown instead of guessing when repo-wide evidence remains insufficient.
</repo_workflow>
```

C. Runtime, Tool Restrictions, and Validity Handling

C.1. Fresh-Session Protocol

Each evaluation example is processed in a fresh per-sample session. No state, history, or memory is carried across examples. This ensures that each judgment is grounded only in the supplied prompt, the read-only repository checkout, and the per-sample diagnostics or PR intent, with no leakage from prior samples in the same run.

Codex. Each sample is evaluated by a separate `codex exec` subprocess launched by the Python harness. The subprocess is invoked with `--ephemeral`, so no prior conversation thread is resumed and no model-side session state persists across samples. The stage-specific prompt is passed on standard input, and the working directory is a freshly materialized detached checkout for that sample. Harness-level resume affects only which sample IDs are skipped and does not reuse any model-side history.

Claude Code. Each sample is evaluated by a separate `claude -p` subprocess launched by the Python harness. The subprocess is invoked with `--no-session-persistence`, so no prior Claude session is resumed and no conversation history carries across samples. The stage-specific user prompt is passed on standard input, and the sample checkout is exposed via `--add-dir` and used as the subprocess working directory. As with Codex, harness-level resume affects only which examples are rerun and does not reuse model-side state.

C.2. Allowed Inspection Tools

For reproducibility, we record the read-only inspection tools permitted in each agent setting. All other tools, including file editing, network access, and remote API calls, are disabled.

Codex. Codex runs combine the `read-only` sandbox with a locked shell profile that sets `shell_environment_policy.inherit=none` and replaces `PATH` with a temporary directory containing the following allowlisted binaries: `awk`, `bash`, `cat`, `cut`, `dirname`, `env`, `find`, `grep`, `head`, `ls`, `pwd`, `realpath`, `rg`, `sed`, `sh`, `sort`, `stat`, `tail`, `tr`, `uname`, `wc`, and `xargs`. Write tools, `git`, `curl`, `wget`, and `python` are excluded from this set, and any remaining write attempt is additionally blocked by the sandbox.

Claude Code. Claude Code is restricted by an explicit CLI allowlist. The allowed tools are `Read`, `Glob`, and a fixed set of read-only Bash command patterns covering `awk`, `cat`, `cut`, `dirname`, `find`, `grep`, `head`, `ls`, `pwd`, `realpath`, `rg`, `sed`, `sort`, `stat`, `tail`, `tr`, `wc`, and `xargs`. Edit/write tools, git-history commands, web or network access, and MCP tools are not permitted.

C.3. Codex Runtime Configuration

We evaluate Codex via the Codex CLI (`codex exec`) backed by GPT-5.4. The runtime configuration is shared across stages, with only the dataset path, stage identifier, and selected sample set varying by run.

Configuration.

- Model name: `gpt-5.4`
- Session isolation: fresh per-sample subprocess
- Sandbox mode: `read-only`
- Structured output mode: JSON output constrained by an explicit schema
- Reasoning configuration: `model_reasoning_effort="none"`

Invocation command. Each sample is invoked through a Python harness that wraps `codex exec` with a locked shell profile (`codex_exec_locked.sh`) constraining `PATH` to an allowlisted read-only tool directory in Appendix C.2. The effective per-sample invocation is shown below.

```
codex exec \
  -c 'shell_environment_policy.inherit=none' \
  -c 'shell_environment_policy.set={PATH="<tmp_locked_tools_dir>"}' \
  -c 'model_reasoning_effort="none"' \
  --model gpt-5.4 \
  --sandbox read-only \
  --output-schema <schema_path> \
  --output-last-message <last_message_path> \
  --json \
  --color never \
  --ephemeral \
  -C <repo_root> \
  -
```

Here `<schema_path>` and `<last_message_path>` are per-sample artifact paths, `<repo_root>` is the detached checkout for the sampled snapshot, and the final − indicates that the prompt is supplied on standard input. The per-sample timeout cap was `900` seconds. In the final runs no completed Codex sample exceeded `300` seconds, so this cap was non-binding in practice.

C.4. Claude Code Runtime Configuration

We evaluate Claude Code via the Claude Code CLI backed by `claude-sonnet-4-6`. The runtime configuration is shared across stages, with only the dataset path, stage identifier, and selected sample set varying by run.

Configuration.

- Model name: `claude-sonnet-4-6`
- Session isolation: fresh per-sample subprocess
- Output format: `json`
- Allowed tools: `Read`, `Glob`, and a restricted Bash subset (Appendix C.2)
- Timeout: `300` seconds per sample

Invocation command. Each sample launches the following `claude -p` subprocess.

```
claude -p \
  --model claude-sonnet-4-6 \
  --output-format json \
  --system-prompt "<Appendix B.4 Claude system message>" \
  --allowed-tools \
    "Read,Glob,\
Bash(cat:*),Bash(find:*),Bash(grep:*),Bash(head:*),Bash(ls:*),\
Bash(pwd),Bash(realpath:*),Bash(sed:*),Bash(sort:*),Bash(stat:*),\
Bash(tail:*),Bash(wc:*),Bash(xargs:*),Bash(awk:*),Bash(rg:*),\
```

```
Bash(tr:*),Bash(cut:*),Bash(dirname:*)" \
  --dangerously-skip-permissions \
  --no-session-persistence \
  --add-dir <repo_root>
```

Here <repo_root> is the detached checkout for the sampled snapshot, and the stage-specific user prompt is supplied on standard input. The 300-second cap reflects a compute budget rather than a benchmark requirement.

C.5. Schema Validation and Invalid Outputs

System outputs are parsed and validated against the JSON schema in Appendix B.2. An output is treated as invalid if it cannot be parsed as JSON, fails schema validation, omits required fields, or exceeds the per-sample runtime budget. Invalid outputs are counted as errors for class-recall and balanced-accuracy calculations.

Parsing. The LLM model harness uses a permissive extraction pass before validation, since some providers emit fenced JSON, <think> traces, or near-schema variants such as rubric axes emitted at the top level rather than under axes. The harness extracts a candidate JSON payload, applies a small normalization step for these recurrent near-schema forms, and then validates against the schema. Codex and Claude Code use a stricter validator that operates on the agent's final returned message and does not strip markdown fences, remove <think> content, or issue a repair prompt after validation failure. Beyond plain JSON syntax errors, outputs are also marked invalid when they violate the schema semantically, including missing required fields, invalid enum values, wrong field types, malformed axis objects, or, for agent runs, omission of the required repo_checks_used field.

Representative invalid outputs. We provide one representative example per dominant failure mode observed in Section 5.

Goedel: prose-style analysis. This output is invalid because it never produces a schema-valid JSON review and instead returns free-form analysis.

```
### Step 1: Understand the Problem

The problem involves evaluating whether a given Lean 4 code snippet
is merge-ready for the mathlib4 repository. The evaluation is based
on several axes, including naming style, documentation, local
structure, file placement, imports and dependencies, proof
readability, API and library fit, and repository overlap and
generality.

### Step 2: Analyze the Code
[...]
```

Kimina: Lean code block. This output is invalid because it returns a fenced Lean block rather than a review JSON object.

````
```lean4
import Mathlib
import Mathlib.Algebra
import Mathlib.Data
import Mathlib.Tactic

/-!
# Mathlib4 merge-readiness analysis
[...]
-/
```
````

Claude Code: malformed near-complete JSON. This output is invalid because it is almost a complete review JSON but contains a later syntax error (Expecting ',' delimiter) and cannot be parsed. The truncation below shows only the well-formed prefix.

```
{
"verdict": "not_merge_ready",
```

```
"p_merge_ready": 0.3,
"overall_confidence": 0.75,
"axes": {
"naming_style": {"label": "concern", "confidence": 0.85, ...},
"documentation": {"label": "concern", "confidence": 0.8, ...},
"local_structure": {"label": "concern", "confidence": 0.9, ...},
[...]
},
"repo_checks_used": ["Mathlib/NumberTheory/LucasLehmer.lean ...", ...]
}
[...]
```

C.6. Timeout Policy

Each sample is given a fixed runtime budget. Outputs exceeding this budget are terminated and counted as invalid for metric purposes.

- Codex per-sample timeout: 900 s
- Claude Code per-sample timeout: 300 s
- LLM model per-sample timeout: 180 s

When a Codex or Claude Code subprocess times out, the harness records the sample as invalid immediately. Partial output is not recovered and re-validated after the timeout fires. For LLM models, the timeout applies at the provider-request level. If the request times out, the harness receives no parseable final output and records the sample as failed unless a later request retry succeeds.

The final runs used different enforced timeout caps across backends. Codex was configured at 900 s, but this cap was non-binding in practice. No completed Codex sample exceeded 300 s. Claude Code used an enforced 300 s cap for compute budget reasons rather than as a benchmark-specific restriction.

C.7. Retry and Repair Policy

The LLM model harness retries provider requests up to three times for transient failures. If a response is returned but fails JSON parsing or schema validation, the harness may additionally issue up to 2 repair retries that re-prompt the model with the validation error and the previous invalid response, instructing it to return only the required JSON object. The agent harnesses do not implement an analogous repair pass. For Codex and Claude Code, any self-correction must occur within the agent's own single run, and a timeout or invalid final output is recorded immediately.

C.8. Compute Resources

Dataset construction, prompt materialization, schema validation, and metric computation run on CPU workers and require no local accelerator. LLM inference for the open-weight models was served locally with vLLM, and the LLM agents (Codex and Claude Code) were served by their respective backends, so backend-side accelerator resources for the agents are external to this paper. Each sample is processed in a fresh subprocess with the per-sample timeout caps reported in Appendix C.6.

LLM models. The 8B-scale models, Qwen3-8B and Kimina-Prover-Distill-8B, were served on a single H100. Each stage took approximately 15 hours for Qwen and 24 hours for Kimina over the full benchmark. The 32B-scale models, DeepSeek-R1-Distill-Qwen and Goedel-Prover-V2-32B, were served on two A100 (80 GB) GPUs each. Each stage took approximately 48 to 72 hours per model over the full benchmark.

LLM agents. The Codex and Claude Code agent runs were orchestrated from a CPU host, with model inference served remotely through the respective CLI backends. For Codex, each stage took approximately 30 hours over the balanced 500/500 subset. For Claude Code, each stage took approximately 80 hours over the balanced 500/500 subset.

D. Input Packaging and Truncation

This appendix records how staged prompt inputs are materialized from the manifest record and the repository checkout. The materialization is deterministic. Given a fixed manifest record and a fixed Mathlib4 checkout at the target snapshot, the resulting Stage 1, Stage 2, and Stage 3 prompts are reproducible.

D.1. Guideline Digests

The final runs use three compact static guideline digests, one each for naming, style, and documentation, rather than the full contribution pages. These digests are shared across all samples and do not vary by sample. They are reduced markdown reference files derived from the corresponding Mathlib contribution guides[4], with the reduction performed offline before evaluation rather than dynamically during prompt construction.

The final digest sizes are modest. The naming digest is 116 lines (4,285 characters), the style digest is 104 lines (4,334 characters), and the documentation digest is 68 lines (2,414 characters). In the reported runs, these digest blocks are not further truncated.

D.2. Changed-File and Import Context

The `changed_files_complete` block contains the snapshot versions of the Lean files changed between `diff_base_sha` and the target snapshot. Small files are included in full. Large files are reduced to the file header, touched declarations, and local windows around each changed hunk, with truncation surfaced by a `[TRUNCATED ...]` marker.

The `first_order_import_files` block contains the modules named in the top-level `import` lines of the changed files. It is therefore a one-hop import context rather than a declaration-level dependency closure. Imported files are also truncated when necessary.

D.3. Diagnostics Packaging

Stage 2 and Stage 3 supply five diagnostics fields (`linter_diagnostics`, `import_diagnostics`, `location_diagnostics`, `doc_diagnostics`, `api_diagnostics`), inserted in a fixed order inside a single `<diagnostics>` block. Recovered check-run records are rendered as short textual entries; build- and workflow-oriented checks are excluded. For accepted snapshots lacking recovered historical diagnostics, the release provides a locally backfilled lint summary, marked by a header such as `source=local_lint_style` so it is distinguishable from a recovered record.

D.4. Length Limits

Input packaging uses fixed character caps rather than a sample-level token budget: 24,000 characters for `diff_chunks`, 12,000 for `changed_files_complete`, 3,000 for `first_order_import_files`, and 8,000 for each diagnostics field. When a block exceeds its cap, truncation preserves the beginning and appends a `[TRUNCATED ...]` marker.

[4] See https://leanprover-community.github.io/contribute/naming.html, https://leanprover-community.github.io/contribute/style.html, and https://leanprover-community.github.io/contribute/doc.html.