LEGO-REVIEW: SCALING CODE REVIEW DATA WITH VERIFICATION FROM PULL REQUEST HISTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

AI coding agents produce a growing share of software changes, making automated code review increasingly important. A promising way to train capable reviewers is to scale the tasks they practice on and the verified trajectories used for training. However, code review yields open-ended, natural-language findings, for which there is no general test-based verifier comparable to the tests used for feature implementation or issue resolution. We observe that pull request (PR) histories offer an implicit verification signal: discussions, code revisions, and other actions can show which findings were accepted or fixed and which were withdrawn or rejected. We introduce Lego-Review, a pipeline to turn this implicit evidence into verifiable review tasks and filtered trajectories. We collect merged GitHub PRs, reconstruct the code snapshots reviewers examined, and use a curation agent to consolidate findings across each PR's discussions and revisions. The agent labels each finding as supported, opposed, or uncertain and binds it to the reviewed snapshot. A finding-level verifier then filters teacher trajectories by crediting recovery of supported high-priority findings and penalizing repetition of opposed ones. From 357k PRs, we build 112k review tasks in eight languages and 24k filtered trajectories. On our proposed Lego-Review-Bench, F1 rises from 5.7 for the base model to 43.1 with 1k filtered trajectories and to 52.9 with 16k, showing effective scaling with verification from PR histories. We release the data, the benchmark, and the trained reviewer to support open research on scalable code review.

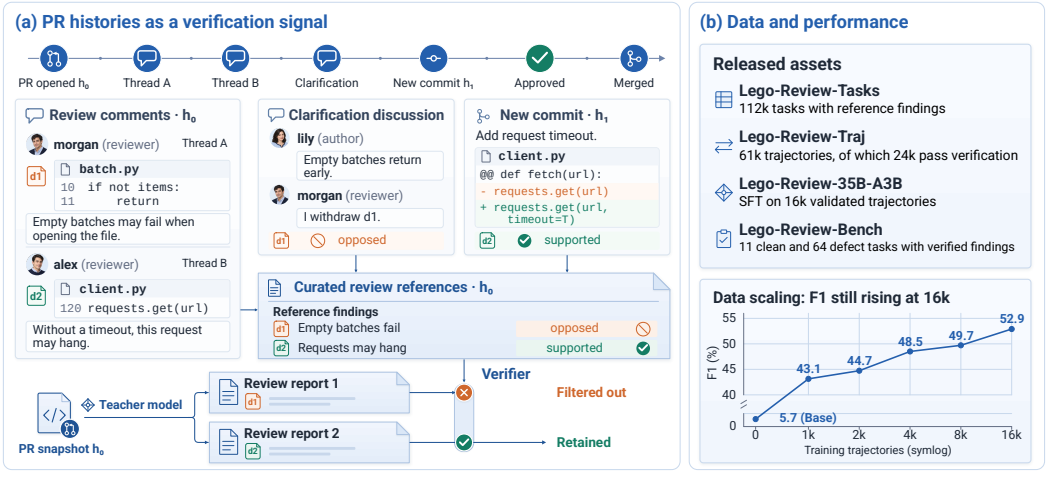


Figure 1: PR histories supply a finding-level verification signal for review agents. (a) At snapshot h_0, the PR history marks finding d_1 opposed after its reviewer withdraws it and d_2 supported after a later commit fixes it. (b) We release 112k review tasks and 61k trajectories, of which 24k pass verification, together with Lego-Review-Bench and Lego-Review-35B-A3B, whose performance scales with the number of training trajectories.

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

1 INTRODUCTION

AI coding agents are increasingly capable of implementing features, resolving issues, and constructing software repositories from scratch (Wang et al., 2025; OpenAI, 2026; Du et al., 2025; Zhao et al., 2026). As they take on more development work, the bottleneck shifts from writing changes to reviewing them. A reviewer must inspect a change in its repository context, find actionable problems, and decide when no blocking problem remains. Can review agents acquire these abilities by training on review tasks and trajectories at scale? The answer depends on whether those data can be verified: unfiltered review may contain missed defects and unsupported findings.

Coding agents benefit from a relatively direct verification mechanism. For many implementation and issue-resolution tasks, executable tests check the resulting patch and determine which trajectories enter training (Yang et al., 2025; Badertdinov et al., 2026; Tao et al., 2026). Review has no corresponding executable outcome for each reported problem. Findings are open-ended descriptions, and accepting or rejecting a change gives little information about whether the report identified the right defects. A scalable review pipeline therefore needs to construct tasks and verify reports at the finding level.

We observe that pull request (PR) histories offer a way to do both. Review discussions identify candidate problems; replies, subsequent commits, and review decisions provide evidence about how those problems were handled. A finding that leads to a fix or explicit agreement is supported, whereas one that is withdrawn or explicitly rejected is opposed (Figure 1). This evidence must be tied to the code the agent actually reviews. A concern raised after a later revision cannot become a reference for an earlier snapshot unless the problem was already present there. The absence of a historical comment also cannot establish that an unmentioned finding is wrong. These constraints make the history a useful but partial finding-level verification signal.

We build Lego-Review around this signal (Figure 2). The pipeline reconstructs PR snapshots as review tasks and uses a curation agent to consolidate distinct concerns, label their support, and bind them to the relevant snapshot. It also creates clean tasks from approved final snapshots after checking prior concerns and post-merge feedback. A teacher agent reviews the tasks without seeing the history, producing trajectories that we filter according to their findings and review decisions. On defect tasks, the verifier rewards recovery of supported high-priority findings and penalizes repetition of opposed ones; on clean tasks, the verifier selects reports that accept the change without high-priority findings.

This process turns 357k merged PRs into 112k review tasks with reference findings, and retains 24k of 61k teacher trajectories. In an audit of 100 PRs, 94.9% of supported high-priority reference findings describe technically valid problems. We train Lego-Review-35B-A3B on the verified data and evaluate it on our proposed Lego-Review-Bench. The results suggest that:

- **Verified trajectories support continued scaling.** Training on 1k to 16k validated trajectories raises Lego-Review-Bench F1 from 43.1 to 52.9, with recall rising from 35.8 to 47.6 (Section 5.3).
- **Verifying findings improves training beyond checking decisions.** Of the trajectories that correctly reject a change with supported high-priority findings, 48.8% do not recall all of them and fail finding-level verification. In an ablation that retains trajectories by decision alone, F1 rises from 44.3 to 46.5 between 2k and 8k trajectories, whereas finding-level verification rises from 44.7 to 49.7 (Section 5.3).
- **Clean examples help reduce false alarms.** Replacing 20% of defect trajectories with clean ones raises Clean Rate, the share of runs on clean changes that return a valid report with no high-priority findings, from 75.8% to 90.9% and F1 from 41.3 to 43.1 (Table 5).

Together, these results suggest that PR histories can play for review agents the role that tests play for coding agents, providing a scalable verification signal at the level of individual findings.

2 RELATED WORK

Scalable Data Construction for Software Agents. Data pipelines for coding agents construct tasks at scale from real repositories, using executable tests to check the resulting code. In issue res-

olution, the tests added by the fixing PR verify a patch (Badertdinov et al., 2025; 2026). SWE-smith and SWE-Lego fine-tune on passing trajectories, with performance rising as the data grows (Yang et al., 2025; Tao et al., 2026). In feature implementation, the tests developers wrote for the new feature verify it (Li et al., 2025). SWE-Dev builds 14k such tasks and trains on them (Du et al., 2025). In whole-repository construction, an agent builds an entire project from its documentation, and executable tests over the finished project verify the result (Ding et al., 2026; Yang et al., 2026). DeNovoSWE builds 4.8k of them from real repositories and fine-tunes on trajectories filtered by test pass rate (Zhao et al., 2026). Code review, however, has no general test-based verifier for its open-ended findings. Lego-Review uses PR histories to supply a finding-level verification signal for review tasks and teacher trajectories.

Code Review Models, Data, and Evaluation. CodeReviewer pre-trains on code changes and human reviews to generate review comments (Li et al., 2022). Later work uses LLM-based semantic filtering to remove noisy review comments (Liu et al., 2025). Other work examines misaligned diff–comment pairs, including comments that require context beyond the diff (Centellas-Claros et al., 2026). CoRAL trains a review-comment generator with reinforcement learning, using rewards based on semantic similarity to reference comments and usefulness for code refinement (Sghaier et al., 2025). SWE-Review closes the loop on issue resolution with agentic code review, training reviewers on trajectories whose accept-or-reject decisions agree with test outcomes (Wang et al., 2026). Benchmarks score reviewers against human review comments (Kumar, 2026; Zverianskii et al., 2026), change requests extracted from PR timelines (Zeng et al., 2026), expert-verified issues (Zhang et al., 2026a), or executable tests (Zhang et al., 2026b). Lego-Review constructs agentic review tasks from human PRs and filters teacher trajectories at the finding level.

Finding-Level Verification Signals from PR Histories. PR histories record how participants handled review comments. MelcotCR selects comments that led to subsequent code changes as training targets (Yu et al., 2025), while empirical studies use subsequent edits as an indicator of comment usefulness (Bosu et al., 2015; Goldman et al., 2025). Comments can raise multiple problems, and a problem can recur across threads or code revisions. Lego-Review consolidates these concerns into distinct reference findings, binds them to the reviewed snapshot, and labels their support using the PR history.

3 Data Construction

Our pipeline turns merged PRs into review tasks that carry the verification signal from PR histories and validated review trajectories (Figure 2). A review task presents a code change in a repository environment and is either a defect task, whose PR history supports at least one finding in the reviewed change, or a clean task, whose history records approval and no unresolved findings. We first select candidate PRs of both kinds and reconstruct the snapshot that reviewers examined. A reference curation agent then reads each candidate's history and diffs, consolidates the problems raised into reference findings with priorities and support labels, and binds them to the review snapshot. Finally, a teacher reviews each task. A verifier checks findings on defect tasks against the references, while clean trajectories are retained when the report accepts the change without high-priority findings.

3.1 Review Task Construction

A review task must present the snapshot to which the PR history's evidence refers. For a defect task, we use the first reviewed snapshot, which contains the change as reviewers first encountered it. For a clean task, we use the approved final PR head. We prepare candidates of both kinds from 356.6k merged PRs across eight languages.

A defect candidate needs at least one substantive comment from a non-author reviewer, that is, an inline comment or a non-approving review body beyond a bare approval such as "LGTM", together with further review activity or code revisions that show how the comment was handled. A clean candidate needs approval of the final PR head by at least two distinct human reviewers other than the author, with no unresolved change request left on that head. These rules admit 108.3k defect and 41.9k clean candidates.

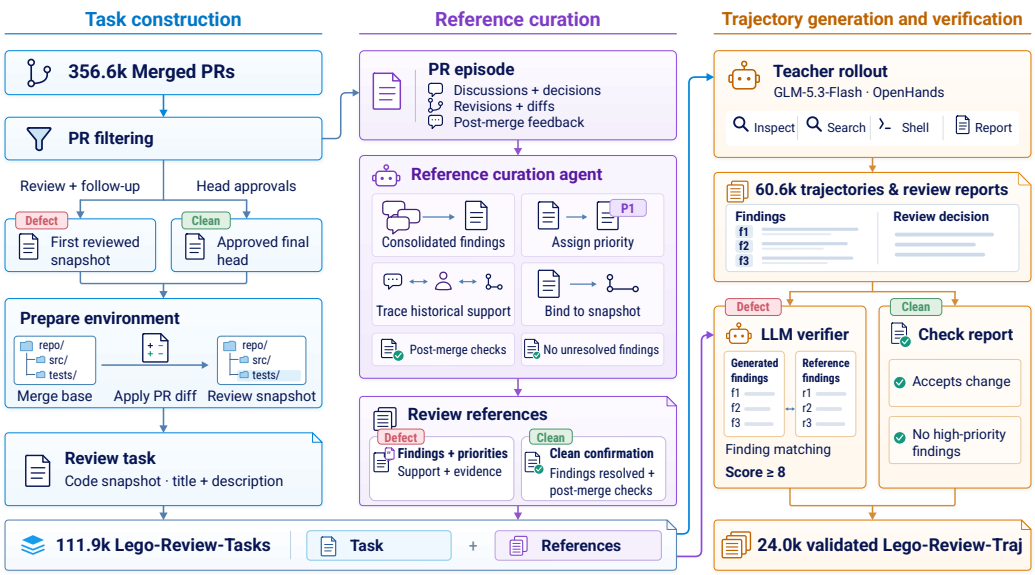


Figure 2: Lego-Review constructs review tasks and validated trajectories from PR histories. Task construction reconstructs the review snapshots and repository environments. Reference curation consolidates findings, assigns priorities and support labels, and binds them to the reviewed snapshots. For clean tasks, it checks prior concerns and post-merge feedback. The teacher generates review trajectories, which are verified against references for defect tasks and checked for acceptance without high-priority findings for clean tasks.

We restore each review snapshot by fetching the repository's merge-base commit and applying the PR diff at that snapshot as uncommitted changes. The general-purpose execution environment provides a shell and basic command-line tools, leaving repository-specific dependency and test setup to the agent as needed.

3.2 Review Reference Curation from PR Histories

PR histories interleave threads, responses, and revisions across several snapshots; turning them into references requires deciding which problems were raised, how each was handled, and where each applies. The curation agent, built on GLM-5.3-Flash, processes each PR in one session. Its input is a PR episode containing review discussions and decisions, revision and merge events, and the associated participants, organized by thread and time. The agent also receives full PR diffs at reviewed snapshots and available commit diffs. Clean episodes additionally include collected post-merge Issues and reverts.

We construct one reference finding for each distinct problem raised during review. The agent separates multiple problems within a comment and merges repeated descriptions across threads. It then inspects the relevant code to assign a priority from P0 to P3. High-priority (P0/P1) findings describe problems that should be fixed before merge, while low-priority (P2/P3) findings concern improvements and nits.

The agent next traces how each finding was handled in the PR history. It reads subsequent replies and checks later diffs for edits that address the reported problem. The agent cites the relevant review decision, quotation, or code edit as evidence. After the session, a deterministic checker checks these citations against the PR records. We map explicit acceptance by the author or a maintainer, a change request in the source review, or a demonstrated fix to `supported`. We assign `opposed` when the reviewer withdraws the finding, or when the author or a maintainer explicitly rejects it. Missing or conflicting evidence yields `uncertain`. Appendix B gives the detailed priority definitions and evidence rules.

We combine findings from reviews of different PR snapshots into one reference set for each defect task. Findings raised on the first reviewed snapshot enter this set directly. For findings raised later, the agent checks the first reviewed snapshot's diff and retains only those already present there. The agent also marks problems that predate the PR as pre-existing. We retain the task when this set contains at least one `supported` finding.

For clean candidates, the curation agent confirms that every finding raised during review was fixed, withdrawn, or explicitly rejected. We additionally inspect post-merge feedback for defects reported after approval. The agent reads collected Issues that reference the PR and any revert PRs or commits. A candidate is excluded when an Issue attributes a concrete defect to the PR or when a revert has landed.

3.3 Review Trajectory Generation and Verification

A teacher, GLM-5.3-Flash in the OpenHands scaffold (Wang et al., 2025), reviews the reconstructed repository snapshot with the PR title and description. Historical discussions and reference findings are not included in its input. The teacher inspects files, searches the repository, and runs shell commands to identify actionable problems introduced by the change, and submits a structured report containing its findings and a review decision. We record its reasoning, tool calls and results, and final report as a review trajectory. The same procedure applies to defect and clean tasks.

An LLM verifier checks the findings in each defect trajectory against the references at the review snapshot. Given the PR context, reference findings, and generated findings with code locations, the verifier matches descriptions of the same underlying problem, allowing differences in wording or proposed repairs. It assigns a report-level score from 0 to 10, crediting recovery of supported high-priority findings and penalizing repetition of `opposed` findings. Unmatched findings are neither credited nor penalized because PR histories do not establish their validity (Appendix F.1).

Both routes require a trajectory from the intended review snapshot with a valid report and message structure. We retain a defect trajectory when its task contains at least one supported high-priority finding, its report rejects the change, and its verifier score is at least 8. Clean tasks have no reference findings and do not require the LLM verifier. We retain their trajectories when the report accepts the change without high-priority findings, with low-priority feedback allowed (Appendix C.5).

4 Data Analysis

Applied to the 356.6k merged PRs, the pipeline yields Lego-Review-Instances, 111.9k review tasks from 11.5k repositories in eight languages, and Lego-Review-Traj, 60.6k review trajectories on these tasks, of which 24.0k pass verification (Table 1. Each task presents a code change at its review snapshot in a repository environment, together with the PR title and description. The 85.0k defect tasks additionally carry 300.5k reference findings, of which 80.2% are supported and 6.4% opposed. Verification passes 32.0% of the trajectories on defect tasks and 62.3% of those on clean tasks.

Section 4.1 audits whether the references reflect what the PR histories record and whether the problems they describe hold in the code. Section 4.2 checks whether the verifier applies them reliably and whether the trajectories it selects train better reviewers.

Table 1: From 356.6k merged PRs, the pipeline yields 111.9k review tasks and 24.0k validated review trajectories. Construction yield from merged PRs to validated review trajectories by route.

Statistic	**Defect**	**Clean**	**All**
Merged PRs	–	–	356,606
Candidate PRs	108,323	41,920	150,243
Review tasks	84,958	26,979	111,937
Review trajectories	45,501	15,054	60,555
Validated review trajectories	14,573	9,382	23,955
Pass rate (%)	32.0	62.3	39.6

4.1 Reference Faithfulness and Validity

Curated references are faithful to the PR histories (Table 2). We audit 100 randomly sampled defect PRs in two passes. GPT-5.6-Sol audits each reference against the code and the PR history, and a human reviews every judgment and adjudicates the invalid and unresolved ones against the source discussions and revision diffs. Of the 122 supported high-priority findings in the sample, 99.2% preserve the original claim, 98.4% carry the correct support label, and 98.3% apply to the review snapshot. Verification therefore credits the problems that reviewers raised at the snapshot they reviewed.

The support label predicts technical validity. The same audit judges whether each described problem holds in the code. Of the 367 reference findings in the sample, 84.9% hold. The rate rises to 91.2% for supported findings and falls to 35.0% for opposed ones, and supported high-priority findings reach 94.9%. The gap justifies the verifier's asymmetry, which credits recovery of supported high-priority findings and penalizes repetition of opposed ones (Section 3.3).

Table 2: Curated references are faithful to the PR histories, and the support label predicts technical validity. Audit of 100 defect PRs. Left: agreement of supported high-priority findings with the PR history. Right: share of findings whose described problem holds in the code, by label and priority. Rates are over decided judgments.

Faithfulness to PR history	%
Claim preserved	99.2
Support label correct	98.4
Applies to the review snapshot	98.3
Claim preserved and applicable	97.5

Technical validity of findings	%
All reference findings	84.9
Opposed	35.0
Supported	91.2
Supported high-priority	94.9

4.2 Verifier Reliability

The verifier's matches agree with the audit (Table 3). We audit 100 randomly sampled reports on defect tasks in two passes. GPT-5.6-Sol audits each match and the validity of each reference, a human adjudicates the disputed cases, and we recompute coverage of the valid supported high-priority findings. Adjudication confirms coverage of 93 of the 96 references that the verifier's matches cover. Audited coverage is 69.1% overall against the verifier's 66.5%, and the two differ by at most 2.3 points within any score group: 5.9% against 3.6% for Low (scores 0–3), 61.3% against 60.4% for Medium (4–7), and 100.0% for both at High (8–10). Verification therefore retains reports scoring at least 8, and in this sample every such report recovers all valid supported high-priority findings.

Higher verifier scores select better training data. After supervised fine-tuning (SFT) on 1k defect trajectories without clean data, using the recipe and benchmark of Section 5.1, Lego-Review-Bench F1 rises from 38.6 for Low to 40.1 for Medium and 41.3 for High, with recall rising from 31.1 to 34.0. Random, a 1k sample from the same pool without verification, reaches 39.3, between Low and High. The score therefore ranks trajectories by their training value, with the High group gaining 2.0 points over an unselected sample and the Low group losing 0.7.

Table 3: The verifier's matches agree with the audit, and higher verifier scores select better training data. Left: coverage of supported high-priority findings (%) by the verifier and after audit. Right: Lego-Review-Bench results (%) after SFT on 1k defect trajectories from each score group or from a random sample. Low, Medium, and High denote verifier scores 0–3, 4–7, and 8–10, and results pool three repeats. Bold marks the best per column.

Review reports	**Verifier**	**Audited**
Overall	66.5	69.1
Low	3.6	5.9
Medium	60.4	61.3
High	100.0	100.0

Training data	**Precision**	**Recall**	**F1**
Random	50.7	32.0	39.3
Low	50.8	31.1	38.6
Medium	51.8	32.6	40.1
High	**52.8**	**34.0**	**41.3**

5 TRAINING EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks. Evaluating a reviewer requires references that hold in the code, extend beyond the review comments, and include clean changes. The Martian Code Review Bench (Zverianskii et al., 2026) offers an independently annotated external reference of 173 human review comments on 50 PRs from five repositories, but its references contain only the comments that human reviewers left on the original PRs, and it has no clean PRs. We therefore build Lego-Review-Bench: 75 PR snapshots from 46 repositories in eight languages, drawn from the repositories of SWE-bench Verified, Multilingual, and Pro and of DeepSWE, with 430 reference findings, 87 of them high-priority, and 11 clean tasks. Candidate findings are collected through two channels, real GitHub review comments and an independent review of the snapshot conducted without access to those comments. Each candidate is then verified in the code, with its trigger, consequence, and evidence recorded. Retained findings pass multi-model re-review with reproduction attempts, and an expert reviewed all 75 tasks and confirmed the clean ones. Benchmark repositories are excluded from training, and Martian serves as an external comparison.

Training. All student models are fine-tuned from Qwen3.5-35B-A3B-Base, a mixture-of-experts model with 35B total and 3B activated parameters, with the same full-parameter SFT recipe, which trains for three epochs at a learning rate of 5×10^{-5} with cosine decay, a warmup ratio of 0.1, and a global batch size of 64 (Appendix E). Repositories used by Martian, SWE-bench Verified, Multilingual, and Pro, and DeepSWE are removed from the training candidate pool before sampling.

Evaluation. Every reviewer runs in the same OpenHands scaffold as the teacher (Wang et al., 2025), with the same prompt and tools, a 512k-token inference context, and a budget of 500 steps and 180 minutes per run. We run three repeats per model and benchmark. GPT-5.6-Sol matches each reviewer's findings to the benchmark references following the Martian procedure (Zverianskii et al., 2026). Each generated finding is compared with each reference in a separate call that returns whether the two describe the same underlying problem, and a finding that matches no reference counts as a false positive (Appendix F.5).

Metrics. Precision, recall, and F1 are computed over finding matches pooled across PRs and the three repeats. Clean Rate is the fraction of runs on clean tasks that return a valid report with no high-priority finding. It counts only high-priority findings because clean tasks contain no problem that must be fixed before merge, while lower-priority suggestions on them are matters of judgment.

5.2 MAIN RESULTS

The frontier models separate cleanly on Lego-Review-Bench (Table 4). Their F1 ranges from 60.4 to 68.1, with Fable 5.1 and GLM-5.3 at the top and each Flash model below its full-size counterpart, and all of them accept clean changes, with Clean Rate between 84.8% and 93.9%. The comparable-size open models score far lower, 24.4 for the instruction-tuned Qwen3.5-35B-A3B and 17.7 for Gemma 4-31B-it. The benchmark thus ranks reviewers in line with their known capability.

Training on validated trajectories turns the base model into a reviewer that approaches the teacher. On Lego-Review-Bench, F1 rises from 5.7 to 52.9, more than twice the instruction-tuned model and 81.6% of the teacher's 64.9. The reviewer also accepts clean changes: Clean Rate reaches 90.9%, against the teacher's 93.9% and within the frontier range.

The remaining gap to the teacher is in recall. The trained reviewer reaches 59.6 precision on Lego-Review-Bench against the teacher's 63.9, but 47.6 recall against 65.8. Section 5.3 shows that recall keeps rising with more training data.

Martian's references are limited to the comments that human reviewers left on the original PRs (Section 5.1), so any finding outside them is scored as a false positive regardless of its validity. Recall and precision therefore trade off more sharply than on Lego-Review-Bench: the two frontier models with the highest recall, Opus 5 and Fable 5.1, have the lowest precision among frontier models, 28.4 and 33.4, and the lowest F1 among them. Under this reference, the trained reviewer's

F1 rises from 3.2 for the base model to 40.2, matching Opus 5 (40.5), with a recall of 46.1 against 27.2 for the instruction-tuned model and 63.0 for the teacher.

Table 4: Training on our trajectories raises Lego-Review-Bench F1 from 5.7 to 52.9 while reaching a frontier-level Clean Rate. Lego-Review-35B-A3B is trained on 16k validated trajectories; results (%) pool three repeats. Bold marks the best among the comparable-size open models and ours.

	Lego-Review-Bench				Martian		
Reviewer	**Precision**	**Recall**	**F1**	**Clean Rate**	**Precision**	**Recall**	**F1**
Frontier models and teacher							
Fable 5.1	61.7	75.9	68.1	90.9	33.4	70.1	45.2
Opus 5	51.4	81.2	63.0	84.8	28.4	71.1	40.5
GLM-5.3	64.1	72.6	68.1	90.9	37.3	65.1	47.4
GLM-5.3-Flash (teacher)	63.9	65.8	64.9	93.9	39.7	63.0	48.7
DeepSeek-V4-Pro	68.1	56.1	61.5	84.8	43.8	63.0	51.7
DeepSeek-V4-Flash	78.2	49.1	60.4	90.9	51.2	53.8	52.4
Comparable-size open models							
Qwen3.5-35B-A3B	41.4	17.3	24.4	42.4	32.0	27.2	29.4
Gemma 4-31B-it	**62.4**	10.3	17.7	78.8	**45.0**	17.2	24.8
Base model and ours							
Qwen3.5-35B-A3B-Base	23.5	3.3	5.7	3.0	9.4	1.9	3.2
Lego-Review-35B-A3B	59.6	**47.6**	**52.9**	**90.9**	35.7	**46.1**	**40.2**

5.3 Scaling with Validated Data

We train reviewers on nested subsets of 1k, 2k, 4k, 8k, and 16k validated trajectories, each with 20% clean data (Section 5.4) and the same recipe (Figure 3).

Review capability keeps rising with validated data through 16k trajectories. Lego-Review-Bench F1 improves at every doubling, from 43.1 at 1k to 52.9 at 16k, with no sign of saturating. Recall, the metric that separates the reviewer from the teacher (Section 5.2), rises from 35.8 to 47.6 and narrows that gap from 30.0 to 18.2 points.

Scaling improves precision alongside recall. Precision rises from 54.2 at 1k trajectories to 59.6 at 16k, showing that the higher recall is accompanied by stronger agreement between the reported findings and the references. Clean Rate remains between 87.9% and 90.9% across the five scales, with 90.9% at both 1k and 16k.

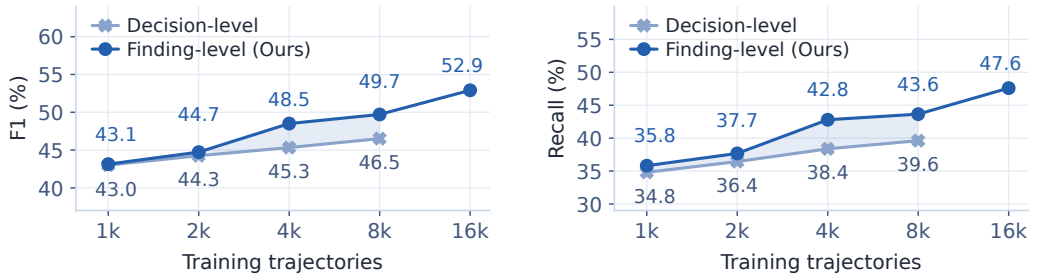
The performance gap between finding-level and decision-level verification widens as the training data scales. To isolate the difference, we retain defect trajectories by decision alone, keeping every report that rejects the change without checking its findings (Appendix A), and train on the same clean subset at each size up to 8k. The two criteria reach the same F1 at 1k (43.1 against 43.0) and 2k (44.7 against 44.3), then separate: finding-level verification leads by 3.2 points at both 4k (48.5 against 45.3) and 8k (49.7 against 46.5), mainly through recall. Within the range we test, finding-level verification therefore accounts for the continued gains in recall as the training data grows.

5.4 Effect of Clean Training Data

Clean tasks supply the training signal for accepting a correct change. To measure their effect, we train on 1k trajectories with 0%, 10%, 20%, and 40% clean data, drawing the defect trajectories from the validated set (Table 5). Trained on defect trajectories alone, the reviewer reports a high-priority finding on one in four clean changes, with a Clean Rate of 75.8%.

Adding clean trajectories raises Clean Rate while keeping recall in a similar range (Table 5). Clean Rate reaches 87.9% with 10% clean data, within the range of the frontier models, while recall varies between 33.8 and 35.8 across the four mixtures.

With the total fixed at 1k trajectories, increasing the clean proportion replaces defect trajectories. Raising the proportion from 20% to 40% increases Clean Rate from 90.9% to 93.9%, while F1



(a) F1 by training size

(b) Recall by training size

Figure 3: Review capability continues to improve with validated data through 16k trajectories. Lego-Review-Bench (a) F1 and (b) recall across training sizes. A pronounced gap in decision-level verification emerges at 4k trajectories and persists thereafter.

changes from 43.1 to 42.7 and recall from 35.8 to 35.6. We adopt 20% clean data for all other experiments, combining the highest F1 among the tested mixtures with a Clean Rate of 90.9%.

Table 5: Clean trajectories reduce false alarms without reducing detection. Lego-Review-Bench results (%) at 1k trajectories by clean proportion. Bold marks the best per column.

Clean (%)	Precision	Recall	F1	Clean Rate
0	52.8	34.0	41.3	75.8
10	51.4	33.8	40.8	87.9
20	**54.2**	**35.8**	**43.1**	90.9
40	53.3	35.6	42.7	**93.9**

6 Limitations

Our references cover only the problems that reviewers raised in the PR histories, so verification credits recovery of those problems and gives no credit to valid findings outside them. Verification leaves such findings unpenalized, and in the audited reports that pass verification 95.7% of them hold in the code (Appendix C.4). Support in the PR history is also not technical validity. In our audit, 5.1% of the supported high-priority findings do not hold in the code, and these errors remain in the references (Section 4.1), where each affects only the scoring of its own task. Finally, our experiments use one teacher and scaffold, one base model, and supervised fine-tuning on up to 16k trajectories. Future work should extend the training to other base models and larger validated sets, since the pipeline draws on PR histories that keep accumulating.

7 Conclusion

We introduced Lego-Review, a pipeline for scaling review-agent training with verification from PR histories. The pipeline reconstructs review snapshots and uses a curation agent to consolidate findings across discussions and revisions, label each as supported, opposed, or uncertain, and bind it to the relevant snapshot. A finding-level verifier then filters teacher trajectories by crediting recovery of supported high-priority findings and penalizing repetition of opposed ones. From 357k merged PRs, we built Lego-Review-Instances, a collection of 112k review tasks in eight languages, and Lego-Review-Traj, a corpus of 61k teacher trajectories, of which 24k pass filtering. We trained Lego-Review-35B-A3B on 16k filtered trajectories, reaching 52.9 F1 on our proposed Lego-Review-Bench, compared with 43.1 when training on 1k. Finding-level filtering yields greater gains than checking decisions alone as the training data grows, while clean examples reduce false alarms. Together, these results suggest that PR histories can play for review agents the role that tests play for coding agents, providing a scalable verification signal at the level of individual findings.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic data sets, to clean and reformat data sets, and to support qualitative analysis of review findings. As part of our method, LLMs curate review references from PR histories and generate the teacher trajectories used for training (Section 3). LLMs also score trajectories and match findings during evaluation. They also assist in constructing the Lego-Review-Bench references and in the audits.(Sections 4 and 5, Appendix C). We have not used generative AI tools to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design or provide feedback on research methodology or experiments, to implement methods, to interpret results, or to assist with translation, and formulating mathematical claims and writing proofs are not applicable to this work. We have reviewed all AI-assisted work: AI-generated code was read, verified, and tested by the authors before use, every audit judgment and every candidate finding in Lego-Review-Bench was read and adjudicated by an author before it was counted, and every reported number was traced by the authors to its experiment artifacts. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The data construction pipeline, audit protocols, training configuration, and evaluation protocol are documented in the main text and the appendices, and all prompts are reproduced in Appendix F. We will release Lego-Review-Instances, Lego-Review-Traj, Lego-Review-Bench, and Lego-Review-35B-A3B, together with the complete codebase and evaluation pipeline, upon acceptance.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.20411.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2602.23866.

Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at Microsoft. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pp. 146–156, 2015. doi: 10.1109/MSR.2015.21.

Leonardo Centellas-Claros, Estefania Pakarati-Cofre, Juan Pablo Sandoval Alcocer, and Diego Elias Costa. Rethinking training data for generating code review comments. *arXiv preprint arXiv:2607.25851*, 2026. URL https://arxiv.org/abs/2607.25851.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941v2.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang.

NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2512.12730.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. SWE-Dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025. URL https://arxiv.org/abs/2505.16975.

Saul Goldman, Hong Yi Lin, Jirat Pasuksmit, Patanamon Thongtanunam, Kla Tantithamthavorn, Zhe Wang, Ray Zhang, Ali Behnaz, Fan Jiang, Michael Siers, Ryan Jiang, Mike Buller, Minwoo Jeong, and Ming Wu. What types of code review comments do developers most frequently resolve? In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3760–3765, 2025. doi: 10.1109/ASE63991.2025.00312.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026. URL https://arxiv.org/abs/2607.07946v1.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Deepak Kumar. SWE-PRBench: Benchmarking AI code review quality against pull request feedback. *arXiv preprint arXiv:2603.26130*, 2026. URL https://arxiv.org/abs/2603.26130.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. FEA-Bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025. URL https://arxiv.org/abs/2503.06680.

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 (ESEC/FSE)*, pp. 1035–1047, 2022. doi: 10.1145/3540250.3549081.

Chunhua Liu, Hong Yi Lin, and Patanamon Thongtanunam. Too noisy to learn: Enhancing data quality for code review comment generation. In *Proceedings of the IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 236–248, 2025. doi: 10.1109/MSR66628.2025.00043.

OpenAI. Introducing codex. https://openai.com/zh-Hans-CN/index/introducing-codex/, 2026.

Oussama Ben Sghaier, Rosalia Tufano, Gabriele Bavota, and Houari Sahraoui. Leveraging reward models for guiding code review comment generation. *arXiv preprint arXiv:2506.04464*, 2025. URL https://arxiv.org/abs/2506.04464.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026. doi: 10.48550/arXiv.2601.01426. URL https://arxiv.org/abs/2601.01426v2.

Ruoyu Wang, Jierun Chen, Shaowei Wang, Chaofan Tao, Sidi Yang, Yuxin Jiang, Kim-Hui Yap, Lifeng Shang, Xiaohui Li, and Haoli Bai. SWE-Review: Closing the loop on issue resolution with agentic code review. *arXiv preprint arXiv:2607.06065*, 2026. URL https://arxiv.org/abs/2607.06065v1.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741v3.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-Bench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

Yongda Yu, Guohao Shi, Xianwei Wu, Haochuan He, XueMing Gu, Qianqian Zhao, Kui Liu, Qiushi Wang, Zhao Tian, Haifeng Shen, and Guoping Rong. Fine-tuning LLMs to analyze multiple dimensions of code review: A Maximum Entropy regulated long Chain-of-Thought approach. *arXiv preprint arXiv:2509.21170*, 2025. URL https://arxiv.org/abs/2509.21170v1.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. SWR-Bench: Assessing LLM performance in real-world code review comment generation. *Proceedings of the ACM on Software Engineering*, 3(FSE):3093–3115, 2026. doi: 10.1145/3808144.

Lei Zhang, Yongda Yu, Minghui Yu, Xinxin Guo, Zhengqi Zhuang, Guoping Rong, Dong Shao, Haifeng Shen, Hongyu Kuang, Zhengfeng Li, Boge Wang, Guoan Zhang, Bangyu Xiang, and Xiaobin Xu. AACR-Bench: Evaluating automatic code review with holistic repository-level context. *arXiv preprint arXiv:2601.19494*, 2026a. URL https://arxiv.org/abs/2601.19494.

Yuntong Zhang, Zhiyuan Pan, Imam Nur Bani Yusuf, Haifeng Ruan, Ridwan Shariffdeen, and Abhik Roychoudhury. Code review agent benchmark. *arXiv preprint arXiv:2603.23448*, 2026b. URL https://arxiv.org/abs/2603.23448.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. DeNovoSWE: Scaling long-horizon environments for generating entire repositories from scratch. *arXiv preprint arXiv:2606.10728*, 2026. URL https://arxiv.org/abs/2606.10728.

Aleksandr Zverianskii, Ashley Zhang, Jacob Clyne, Antía Garcia, Fazl Barez, and Shriyash Upadhyay. Code Review Bench, 2026. URL https://github.com/withmartian/code-review-benchmark.

A An Incorrect Finding Retained by Decision-Level Verification

SWE-Review selects its default training trajectories by comparing the review decision with the candidate patch's executable outcome, approving a resolving patch or requesting changes on an unresolved one (Wang et al., 2026). Decision-level verification retains the full trajectory, exploration and findings included, whenever the decision matches, so individual findings receive no check.

Figure 4 shows an incorrect finding in a selected `treq` trajectory. The task is to close an HTTP connection when its response collector raises an exception. The reviewer requests changes, arguing that the patch's bare `except:` catches `KeyboardInterrupt` and prevents users from interrupting the program. The handler instead closes the connection and immediately re-raises the exception, as confirmed by a local check with `KeyboardInterrupt`. Decision-level verification retains this trajectory because its `request_changes` decision matches the unresolved outcome released with the trajectory, carrying the incorrect finding into the training data.

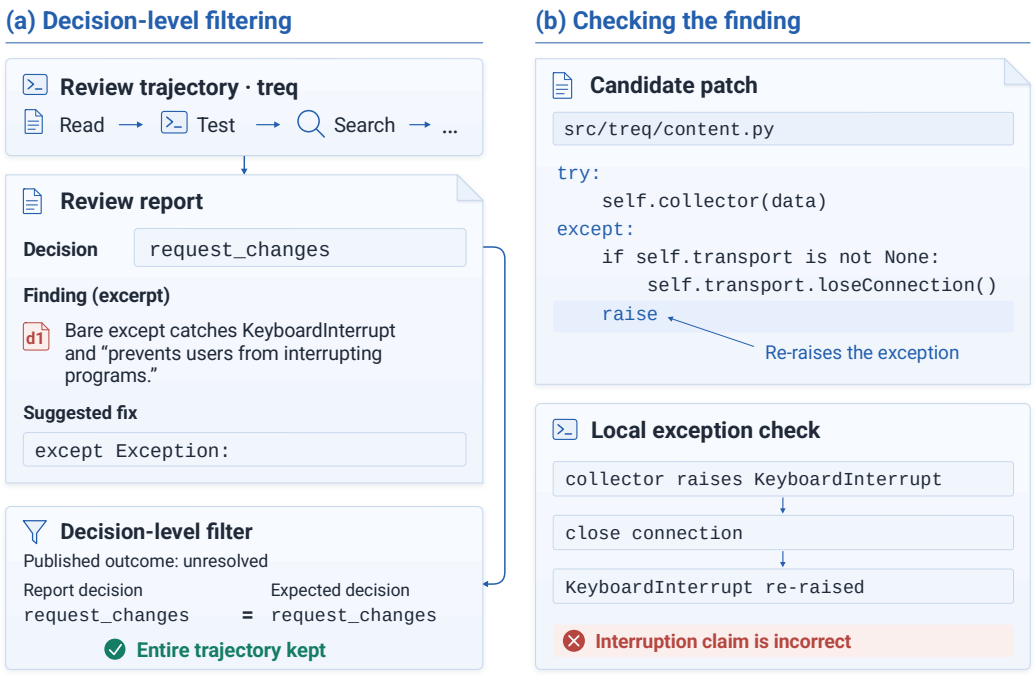


Figure 4: Decision-level verification can retain an incorrect finding. In a `treq` example from SWE-Review-Traj, (a) the decision agrees with the executable reference, while (b) the handler re-raises the exception that the reviewer claims it suppresses. Tool interactions are schematic and the finding is excerpted.

The same gap appears in our teacher trajectories. Among the 28,475 reports that reject a change on a defect task with supported high-priority findings, 13,902 (48.8%) fail finding-level verification, so decision agreement alone would admit them.

B Review Reference Curation Details

The curation agent processes each PR in one session, reading its episode, snapshot diffs, and available commit diffs. Each snapshot diff shows the full change from the merge base. For clean candidates, the agent also reads collected post-merge Issues and reverts.

B.1 Finding Priorities

The agent assigns priority from the finding's claim and code context using Table 6. How a request is phrased and how it is answered do not determine priority.

Table 6: Priority definitions used during review reference curation.

Priority	Definition
P0	A problem whose merge would cause a failure, such as a crash, wrong result, data loss or corruption, security vulnerability, build failure, or failing test.
P1	A substantive problem to fix before merge, including errors in the change's logic, missing edge-case or error handling, performance regressions, untested behavior changes, API or design mistakes, and documentation that would mislead users.
P2	An improvement, such as clearer naming or documentation, improved maintainability or test readability, reduced duplication, or an additional test for behavior already covered.
P3	A nit, style or personal preference, or a claim that identifies no concrete problem.

B.2 Evidence Categories and Checks

The agent assigns one evidence category to each finding (Table 7). A later resolution supersedes earlier responses. When several supporting categories apply, a fix takes precedence over a change request, which takes precedence over verbal acceptance.

Table 7: Evidence categories and their fixed mapping to support labels.

Category	Required evidence	Label
Acceptance	The PR author or a maintainer other than the person who raised the finding explicitly accepts it or states that it has been handled. A commitment to handle it in a follow-up also qualifies.	`supported`
Change request	The review containing the original comment requests changes.	`supported`
Fix	A later commit or snapshot diff contains a code change addressing the finding.	`supported`
Withdrawal	The participant who raised the finding explicitly retracts it.	`opposed`
Rejection	The PR author or a maintainer other than the person who raised the finding explicitly rejects it or its premise, and the PR proceeds without the requested change.	`opposed`
Unclear	Evidence is absent or conflicting, or the code needed to determine the outcome is unavailable.	`uncertain`

For acceptance, withdrawal, or rejection, the agent quotes the required participant's response. Answers and explanations qualify only when they explicitly accept or reject the finding. A change request cites the source review's state. A fix cites a later commit or snapshot, a file path, and an added or removed line that addresses the finding. A rebased copy of an existing addition does not count as a fix. Thread markers and commit messages guide inspection of the changed code.

A deterministic checker verifies that the agent classifies every supplied message and that its citations, quotations, participant roles, and event order agree with the episode. Fix evidence must quote a changed line from the cited file. Version assignments must name a supplied snapshot or commit, or mark the finding's origin as pre-existing or unknown. We exclude outputs that fail these checks and map the remaining evidence categories to their fixed support labels.

C Dataset Statistics and Audit Protocols

C.1 Construction Yield and Task Composition

Table 1 in Section 4 gives the yield from merged PRs to review tasks, and the paragraph below accounts for each stage. Each trajectory corresponds to a distinct review task. Within the generation budget, the teacher reviews 60,555 of the 111,937 tasks, and trajectories for the remaining tasks will be added in a future release. The 60,555 review trajectories span 9,824 repositories, and the 23,955 validated trajectories span 6,882.

We collect merged PRs from open-source repositories selected by GitHub stars. The PRs were merged between August 2011 and May 2026, and 70.0% of the released tasks come from PRs merged in 2023 or later. Of the 356,606 merged PRs, the rule-based filter rejects 201,943 with no

qualifying review activity and 4,420 whose only review discussion is a single unanswered thread. Among the 150,243 candidates, 5,572 return no result and 8,597 produce labels that fail the evidence check. Task construction further excludes 19,662 PRs whose history yields no reference at the review snapshot, 1,747 with empty, binary, or oversized patches, 1,182 without a patch and base commit, and 1,546 in held-out benchmark repositories. Verification removes 22,995 defect trajectories that fail finding-level verification, two without a valid report, 7,931 from tasks without a supported high-priority finding, and 5,672 that fail clean verification. The tasks cover eight programming languages, assigned by each repository's primary GitHub language. Go, Python, and Java account for 57.1% of them (Table 8).

Table 8: Tasks by programming language. A few repositories appear under two languages.

Language	Defect	Clean	All (%)	Repositories
Go	19,914	8,152	28,066 (25.1)	2,539
Python	16,211	3,000	19,211 (17.2)	1,846
Java	11,810	4,789	16,599 (14.8)	1,439
Rust	10,842	2,792	13,634 (12.2)	1,885
TypeScript	10,375	2,714	13,089 (11.7)	1,533
C++	7,295	2,137	9,432 (8.4)	908
JavaScript	4,622	1,462	6,084 (5.4)	782
C	3,889	1,933	5,822 (5.2)	608
All	84,958	26,979	111,937 (100.0)	11,537

Table 9 gives the full statistics of the release.

Table 9: Full statistics of the review tasks, reference findings, and validated review trajectories. Trajectories are generated by GLM-5.3-Flash with OpenHands; trajectory rows other than the pass rate describe validated trajectories. High-priority rows count P0/P1 findings, as assigned during curation, that are not marked pre-existing.

Review tasks	**Defect**	**Clean**	**All**
Tasks	84,958	26,979	111,937
Repositories	11,178	4,397	11,537
Avg. files / lines changed	7.0 / 262.2	7.0 / 193.6	7.0 / 245.7
Reference findings (defect)	**Supported**	**Opposed**	**Uncertain**
Findings	241,105	19,193	40,210
Tasks with ≥ 1	84,958	14,660	23,264
Avg. per task	2.8	0.2	0.5
High-priority findings	94,518	3,501	8,812
Tasks with ≥ 1 high-priority	58,641	3,218	6,957
Review trajectories	**Defect**	**Clean**	**All**
Validated trajectories	14,573	9,382	23,955
Pass rate (%)	32.0	62.3	39.6
Repositories	6,033	2,757	6,882
Avg. findings per report	4.4	1.4	3.3
Median assistant turns	52	49	51
Training tokens	–	–	1.9B

Released data. Each task in Lego-Review-Instances records the repository, PR number, route, base and review commits, PR title and description, the patch under review, and its reference findings with their claims, categories, priorities, support labels, and snapshot scope. Clean tasks also record their number of approving reviewers. Each trajectory in Lego-Review-Traj records the task it reviews, the teacher, the full sequence of messages and tool calls, the final report, and the verifier's score, justification, and verification outcome.

The released tasks have a median of 4 per repository, and the ten largest repositories account for 1.2% of them. Among the defect tasks, 22,153 contain more than one supported high-priority finding.

Clean tasks have no reference findings because every finding raised in their reviews was fixed, withdrawn, or rejected before merge.

Of the supported findings, 95.3% are supported by a fix in a later diff, 3.8% by explicit acceptance, and 0.9% by a change request, and the fix share is 96.3% among supported high-priority findings. When more than one kind of evidence applies, the agent records the one backed by code, so the fix share also absorbs findings that were accepted and then fixed.

C.2 Reference Audit

The reference audit checks extraction fidelity, support labels, and technical validity against the PR's code and discussion. We sample 100 defect PRs from tasks with completed references, ordering PRs by a seeded hash (seed 42). Of these PRs, 79 contain 122 supported high-priority findings. The sample retains its production labels throughout the audit.

GPT-5.6-Sol audits each PR in one session. It first judges technical validity from the code, then reads the support labels and episode evidence. GPT-6-Astra re-examines the adverse judgments, and a human expert reviews every judgment and adjudicates the invalid and unresolved ones using the initial decisions, source discussions, relevant functions and call paths, and revision diffs. Each judgment records a valid, invalid, or unresolved outcome and the supporting evidence.

Table 10 includes all 367 reference findings in the sample. Among decided judgments, 91.2% of supported findings are technically valid, compared with 35.0% of opposed and 47.6% of uncertain findings. Another 11.8% of supported findings remain unresolved. The final row covers the 122 supported high-priority findings (Section 3.3) used in the main analysis.

Table 10: Supported findings have higher technical validity than opposed or uncertain findings. Technical validity of the 367 audited reference findings by label. The last row restricts to supported high-priority findings that apply to the review snapshot and are not pre-existing (Section 3.3). Rates exclude unresolved judgments.

Findings	Valid	Invalid	Unresolved	Valid (%)
All labels	276	49	42	84.9
Supported	259	25	38	91.2
Opposed	7	13	1	35.0
Uncertain	10	11	3	47.6
Supported high-priority	111	6	5	94.9

For the 122 supported high-priority findings, content fidelity is 99.2%, snapshot applicability is 98.3%, and technical validity is 94.9%. These measures check whether extraction preserves the original claim, whether the problem applies to the review snapshot, and whether the claim holds in the code, respectively. Support-label accuracy is 98.4%, with no unresolved judgments, when compared with the audited PR history.

Table 11: Errors found among the 122 audited supported high-priority findings. The two snapshot errors are also counted under technical validity.

Check	Errors	Observed failure
Content fidelity	1	The reference adds an incorrect execution path to an original review comment about a missing null guard.
Snapshot applicability	2	One problem is already fixed at the review snapshot; the other is introduced in a later revision.
Technical validity	6	Two snapshot errors and four findings contradicted by the code's control flow or existing implementation.

One technically invalid finding (Table 11) warns that a pointer to a loop variable will change across iterations, but the loop breaks immediately after taking the address. Unresolved findings require platform-specific build or runtime evidence, or an explicit design contract.

C.3 Report Audit

The report audit samples 100 reports from the defect trajectories that have a valid report. We stratify by score group (Low, Medium, or High) and review decision, giving six strata, and sample proportionally within each stratum (seed 42). The sample contains 23 Low, 29 Medium, and 48 High reports.

GPT-5.6-Sol audits each report, checking reference claims against the reviewed code before reading the matching materials. GPT-6-Astra re-examines the adverse judgments, and a human expert adjudicates disputed references and matches using the code and claim text, including checks for missed matches. Findings match when they describe the same underlying problem, even if their wording or proposed repairs differ.

The sampled tasks contain 149 supported high-priority findings: 137 are technically valid, nine invalid, and three unresolved. Four reports have no valid references and are excluded from coverage means: three Low and one Medium. We compute coverage of valid, applicable references within each report, then weight reports by their stratum's population-to-sample ratio. Production matches cover 96 of the 149 reference findings, and adjudication confirms coverage of 93 of them, with one more unresolved. Restricting the denominator to valid references gives 91 covered findings out of 137 and a weighted mean per-report coverage of 69.1% (Table 12). The unresolved match falls outside this valid-reference denominator.

Table 12: Audited reference coverage increases across the score groups. Weighted mean per-report coverage (%) under production matches, adjudicated matches, and valid references only. Ranges reflect unresolved matches.

Group	Reports	Production matches	Adjudicated matches	Valid reference coverage
Overall	100	66.5	66.0–66.3	69.1
Low	23	3.6	3.6	5.9
Medium	29	60.4	59.0–59.8	61.3
High	48	100.0	100.0	100.0

C.4 Technical Validity of Unmatched Findings

Verification neither credits nor penalizes a generated finding that matches no reference (Section 3.3). This audit checks whether such findings hold in the code, using the same 100 reports as the report audit and every finding they contain. Of the 39 reports that pass verification, 37 have recoverable code, with 94 unmatched findings.

GPT-5.6-Sol audits each finding against the reviewed revision and its base, reading only the code and the claim, while scores, decisions, and production matches are withheld. GPT-6-Astra re-examines every invalid, unresolved, or preference-only judgment together with a 20% random sample of the valid ones, and a human expert reviews every judgment. A finding is valid when its premise, consequence, and applicability hold in the code, including grounded improvements to tests, documentation, or design, and a material false premise, consequence, or repair makes it invalid as written. Pure preferences are counted separately, priority is not reassessed, and a finding counts as matched if the production verifier matched it to any reference.

Unmatched findings in these reports rarely turn out invalid (Table 13). Of the 94 unmatched findings, 4 (4.3%) are invalid. The remaining 90 comprise 51 defects and 39 grounded improvements. Penalizing unmatched findings instead would remove 36 of the 39 reports. Leaving them unscored therefore admits few invalid claims into the training data, and the trained reviewer bears this out. Its precision on Lego-Review-Bench lies within the range of the frontier models, while its gap to them lies in recall (Section 5.2).

C.5 Clean Task and Report Audits

The clean route supplies the accept decisions in the training data, and it can fail in two ways. A clean task may still contain a problem that should have blocked the merge, so that accepting it teaches a

Table 13: Unmatched findings in reports that pass verification are mostly valid defects or grounded improvements. Counts over the 37 audited reports with recoverable code; the report rows use the 39 sampled reports that pass verification where matching alone decides.

Unmatched findings	**Count**
Valid, defect	51
Valid, grounded improvement	39
Invalid	4
All	94
Reports	
With any unmatched finding	36 of 39

false acceptance, and a report that passes clean verification may carry low-priority findings that are wrong. We audit both on the same 50 clean tasks at their final PR heads (Table 14).

For the task audit, GPT-5.6-Sol reviews each final head against the merge base without seeing the approvals, the PR history, or the teacher's report, and lists candidate problems. GPT-6-Astra adjudicates each candidate for technical validity, whether the PR introduced it, and its priority, and a human expert reviews every adjudication. The history is consulted only afterwards, to explain how a remaining problem escaped curation. Adjudication finds nothing worth fixing before merge in 47 of the 50 tasks (94.0%). Each of the other three contains a high-priority problem introduced by the PR. The teacher rejects two of these changes, so their reports fail clean verification, and of the 28 reports on the cohort that pass, one (3.6%) accepts a change with a problem worth fixing.

For the report audit, we take the 44 teacher reports on these tasks. Of the 28 that pass clean verification, eight contain no findings and 20 carry 42 low-priority findings. GPT-6-Astra audits each finding against the reviewed code, the relevant call paths, and the tests before reading the PR history, and a human expert adjudicates its judgments. Of the 42 findings, 34 hold in the code, five are invalid, one is unresolved, and two express preferences without a technical claim. Of the 34 valid findings, 33 concern code introduced or exposed by the PR, and 41 of the 42 have no counterpart in the PR history. Examples include an unused import left by the PR and a URL equality test that takes an object-identity shortcut without exercising the intended comparison. Clean verification therefore retains reports that accept mergeable changes and add mostly valid low-priority feedback, which is the data behind the Clean Rate gains in Section 5.4.

Table 14: Clean tasks contain nothing worth fixing before merge in 47 of 50 cases, and the low-priority findings in reports that pass clean verification are mostly valid. Audits of 50 clean tasks and the teacher reports on them.

Clean tasks	**Count**
Audited	50
Nothing worth fixing before merge	47
High-priority problem introduced by the PR	3
Teacher reports on these tasks	
All	44
Passing clean verification	28
Accepting a change with a problem worth fixing	1
Low-priority findings in passing reports	
All	42
Valid	34
Invalid	5
Unresolved or preference	3

D Lego-Review-Bench Composition and Martian Limitations

D.1 Composition and Sources

Lego-Review-Bench contains 75 PR snapshots from 46 repositories represented in SWE-bench Verified and Multilingual (Jimenez et al., 2024), SWE-bench Pro (Deng et al., 2025), and DeepSWE (Huang et al., 2026). These comprise 29 Python, 16 Go, eight TypeScript, six JavaScript, and four each in C, Java, Ruby, and Rust. There are 36 bug fixes and 39 feature implementations or enhancements (Table 15).

The benchmark contains 430 reference findings: 87 high-priority and 343 low-priority findings. Of the 75 tasks, 45 contain a high-priority finding, 19 contain only low-priority findings, and 11 contain none. This mix evaluates both defect detection and false alarms. Each task specifies a base revision, review snapshot, and review cutoff. Benchmark repositories are excluded from the training candidate pool.

Table 15: Composition of Lego-Review-Bench. The 11 clean tasks have no reference findings after code review and adjudication.

Statistic	Value	Statistic	Value
PR snapshots	75	Bug fixes	36
Repositories	46	Features / enhancements	39
Languages	8	Reference findings	430
Defect / clean tasks	64 / 11	High / low priority	87 / 343

D.2 Limitations of Martian

Martian contains 50 PRs with 173 reference findings and no clean tasks, so it cannot measure false alarms on changes that should be accepted. Its references also appear to miss many valid problems. Across the frontier models in Table 4, recall on Martian stays within about 10 points of recall on Lego-Review-Bench, while precision is 23 to 28 points lower for every model. Because findings outside the references count as false positives, Martian F1 favors reviewers that report less. It ranks DeepSeek-V4-Flash, which has the lowest recall, first (52.4 F1), and the two models with the highest recall, Opus 5 and Fable 5.1, last (40.5 and 45.2). Within both the DeepSeek and GLM families, the larger model recalls more on both benchmarks and leads by F1 on Lego-Review-Bench, yet trails on Martian (DeepSeek-V4-Pro 51.7 vs. 52.4, GLM-5.3 47.4 vs. 48.7). We therefore evaluate training choices on Lego-Review-Bench.

E Training Configuration

E.1 Trajectory Formats and Validity Checks

Each generated finding records a description, code location, P0–P3 priority, and confidence. Clean verification retains reports that contain no high-priority findings and whose review decision is `patch is correct`.

Before verifying either route, we check that the review snapshot matches the task, the final report parses into findings and a review decision, and the trajectory contains valid messages and tool interactions. For defect trajectories, we retain the verifier's matches and score justification. The verifier assigns an integer from 0 to 10 using the matching and scoring instructions in Appendix F.1. The prompt describes 0 as finding none of what mattered and 10 as finding everything that mattered. Verification uses this score directly, retaining trajectories that score at least 8, reject the change, and review a task with a supported high-priority finding.

E.2 Training Configuration

Students start from Qwen3.5-35B-A3B-Base, a mixture-of-experts model with 3B activated parameters. All controlled student runs use full-parameter SFT with the configuration in Table 16.

Before sampling training trajectories, we exclude repositories used by Martian (Zverianskii et al., 2026), SWE-bench Verified and Multilingual (Jimenez et al., 2024; Yang et al., 2025), SWE-bench Pro (Deng et al., 2025), and DeepSWE (Huang et al., 2026) from the training candidate pool.

Table 16: Shared SFT configuration for all student runs.

Setting	Value
Initialization	Qwen3.5-35B-A3B-Base
Fine-tuning	Full SFT
Chat template	`qwen3_5_nothink`
Global batch size	64
Learning rate	5×10^{-5}
Learning-rate schedule	Cosine decay
Warmup ratio	0.1
Weight decay	0.01
Gradient clipping	1.0
Precision	bfloat16
Epochs / checkpoint	3 / final epoch
Context length	196,608 tokens (192k)
Sampling / training / shuffle seed	42 for each use
Evaluation repeats	3 per benchmark
Clean proportions in analysis	0%, 10%, 20%, 40% at 1,000 trajectories
Clean proportion (main)	20%

F PROMPTS

This appendix reproduces the task prompts for trajectory scoring, review generation, reference curation, and benchmark matching. Braced placeholders are filled with the corresponding PR or review content. Double braces in the scoring templates escape literal JSON braces. Line wrapping and punctuation are normalized for typesetting.

F.1 TRAJECTORY VERIFIER

The verifier receives one user message containing the PR title and description, reference findings at the reviewed snapshot, and the generated findings. Each reference includes its identifier, claim, category, priority, scope, and support label. Each generated finding includes its identifier, title, body, and code location. Its priority prefix is removed from the title. The verifier returns finding summaries, semantic matches, an integer score, and a score justification.

The scoring instructions below define the qualitative criteria and the endpoints of the 0–10 scale. The model assigns the score, and the controller checks its type and range before applying the threshold of 8. There is no separate point-allocation table or arithmetic conversion from matches to scores.

```
Judge an AI reviewer's review of a pull request: match its findings to
    the defects human reviewers raised, then score how well it found
    what mattered. Text only -- you have no code and never decide who
    is technically right.

## Pull request
Repository: {repo}   #{number} Title: {title} Description (author's
    stated intent; may be inaccurate): {body}

## Human-raised defects (on the version the AI reviewed)
{defects_block}

`priority` (fixed, do not revise): P0 merging it breaks something; P1
    must be fixed before merge; P2 an improvement worth making; P3 a
    nit or no concrete problem. `scope`: `pre_existing` means the
    problem was already there -- not this review's to find; missing it
    is not a miss. `verdict` is what the participants did with the
    defect -- evidence about the defect, not the review:
- `supported` -- fixed, accepted or blocked on. Real and worth
    raising: finding it is finding something that mattered, missing it
    is missing something real.
- `opposed` -- argued down and the pull request went ahead without it.
    Raising the same thing repeats a claim the code's owners rejected.
- `uncertain` -- nobody acted either way. Raising it independently is
    a second source pointing at the same problem; not raising it is not
    a fault.

A finding that matches no human defect is neither credit nor fault:
    the humans may not have looked there, and you cannot check it. **A
    review must not score lower for having unmatched findings, and
    their count must not appear in your reasoning.**

## AI findings
{findings_block}

## Step 1 -- match
Write one line per finding saying what problem it claims, so that
    every finding has been read. Then, for every G, list the findings
    that identify the SAME underlying issue as that defect.
- Accept semantic matches: different wording is fine, and so is a
    different proposed repair, if it is the same problem. Root cause
    versus symptom, mechanism versus consequence and question versus
    answer are all the same issue.
```

```
- Two issues that merely touch are not a match: a test that would have
    caught the other, two faults growing from one cause, or partial
    overlap with something left over on both sides.
- A G may match several F and an F may match several G. Leave a defect
    unmatched rather than forcing a match. Judge only sameness -- not
    importance, correctness, or the score.

## Step 2 -- score
One integer 0-10: as a training example of reviewing this pull
    request, how well did the review find what mattered? Weigh each
    defect by priority, scope and verdict. The score is about the P0
    and P1 defects: a missed P2 or P3 costs nothing, and a found one is
    a small plus. 10 = found everything that mattered; 0 = none of it.
    One sentence on what drove the number, in terms of defects found
    and missed.

## Output -- JSON only, ids from above
{{"finding_summaries": [{{"id": "F0", "summary": "..."}}],
 "matches": [{{"g": "G0", "f": "F1", "why": "..."}}],
 "score": 7, "score_reason": "..."}}
```

F.2 Review Generation and Evaluation

The following task template is used for P0–P3 review reports in trajectory generation and benchmark evaluation. The OpenHands agent receives the repository with the PR patch applied to the base commit, and the template supplies the PR title and description. Historical review discussions and reference findings are absent from this input.

```
You are reviewing a pull request in the repository `{repo}`.

The current working directory contains the repository checked out at
    the PR's base commit, with the PR's changes applied as uncommitted
    modifications.

# Pull request context

**Title:** {pr_title}

**Description:**
{pr_description}

Use the title and description only to understand the author's intent
    (what the change is *supposed* to do). Do NOT treat them as ground
    truth -- the change may fail to implement the stated intent, and
    the description may itself be inaccurate or incomplete.

# What to report

Your goal is to surface every real problem this PR introduces. Aim for
    complete coverage of the changed code: a reviewer who misses a real
    defect fails the author just as much as one who raises noise.

A finding must satisfy all of these to be reported at all:

1. It impacts the correctness, performance, security, maintainability,
    or readability of the code.
2. The bug is discrete and actionable (not a general issue or
    combination of multiple issues).
3. The bug was introduced in this PR's changes (do not flag
    pre-existing bugs).
4. It is grounded in the code you actually read -- not in guesses
    about code you did not inspect.
```

```
**Confidence calibration** -- do not discard a finding merely because
    you are not fully certain. Instead, let your certainty decide how
    you report it:

- Certain the code is wrong -> report it, with a high
    `confidence_score`.
- The evidence points to a real problem but a benign explanation
    remains possible (e.g. it depends on an unstated assumption, on
    author intent, or on a call path you could not fully trace) ->
    still report it, with a lower `confidence_score`, and state the
    condition under which it bites.
- Only pure speculation with no supporting evidence in the code -> do
    not report.

**Priority** -- how much the problem itself matters:

- P0: merging it would break something: crash, wrong result, data loss
    or corruption, security hole, build failure, failing tests.
- P1: a substantive problem that should be fixed before merge: missing
    edge-case or error handling, performance regression, changed
    behaviour left untested, an API or design mistake, a documentation
    error that would mislead users.
- P2: an improvement worth making: maintainability, naming, comment or
    documentation accuracy, test readability, duplication, a
    micro-optimisation.
- P3: a nit, a style or personal preference, or a problem that is out
    of this pull request's scope.

Reasoning from the code you have read to a conclusion is not
    speculation; it is the job. Trace the affected call paths where you
    can, and report what the evidence supports.

Before you finish, walk the diff hunk by hunk. For each changed hunk,
    decide explicitly one of: (a) it introduces a problem you are
    reporting, (b) you traced it and it is fine, or (c) you could not
    determine it. Do not stop at the first problem you find in a file
    -- a single file often contains more than one, and finishing a file
    early is the most common way real defects are missed.

Changed test code counts as changed code. When tests are added or
    modified, check that each new assertion can actually fail: an
    assertion that compares a value to itself, that is missing its
    expected value, or whose expected value was updated to match
    current (buggy) behaviour is a real defect, not a stylistic issue.

If no qualifying findings exist, output an empty findings array with
    overall_correctness = "patch is correct".

# How to investigate

That is up to you. You have a shell and the whole repository.

# Output

Write your review as JSON to `/workspace/review.json`:

'''json
{
  "findings": [
    {
      "title": "<<= 80 chars, imperative, prefixed with [P0], [P1],
    [P2] or [P3]>",
      "body": "<Markdown explaining *why* this is a problem; cite
    files/lines>",
      "confidence_score": <float 0.0-1.0>,
```

```
      "priority": "P0" | "P1" | "P2" | "P3",
      "code_location": {
        "absolute_file_path": "<file path>",
        "line_range": {"start": <int>, "end": <int>}
      }
    }
  ],
  "overall_correctness": "patch is correct" | "patch is incorrect",
  "overall_explanation": "<1-3 sentences justifying the verdict>",
  "overall_confidence_score": <float 0.0-1.0>
}
```

```
Do not wrap the JSON in markdown fences or extra prose. Line ranges
    should be as short as possible (<=5-10 lines) and should overlap
    with the diff. One finding per distinct issue.

Write for an author who will read the comment once: say what is wrong
    and the situation in which it goes wrong, without overstating
    severity or padding with praise.
```

F.3 Reference Curation for Defect Tasks

The curation agent receives the rendered PR conversation and snapshot and commit diffs. The prompt refers to reference findings as `defects`. Its output records each finding's source messages, priority, origin, and a cited behavior. The controller checks the citations and maps behaviors to support labels as described in Appendix B.

```
# Task

You are given one GitHub pull request as a conversation in
    `/workspace/input/pr.md`, and the code it changed as diff files
    under `/workspace/input/diffs/`. Read them and write one JSON
    object to `/workspace/result.json` describing what was raised, how
    much each thing matters, where it came from, and what happened to
    it.

## What the input contains

`pr.md` opens with a `## Contents` index giving the line number of
    every section. **Use it.** A file viewer truncates a large file
    without telling you which part it kept, so on anything long read by
    range rather than whole -- and if a read comes back marked as
    clipped, go back with ranges until you have seen all of it. A
    defect you never saw is the one mistake nothing downstream can
    catch.

- `### Snapshots` lists every version of the pull request that
    somebody reviewed, as `S1`, `S2`, ... in the order they were first
    reviewed; the final head is last. `diffs/S#.diff` is the pull
    request's **whole change** at that version, against its merge base
    -- so two snapshots can be compared file by file regardless of
    rebases. The commits in the episode are the final chain only; a
    commit `E###` may point at `diffs/E###.diff`, that commit's own
    change. Where the history was force-pushed, commits from before the
    push are not listed, and only the snapshot diffs show what changed
    between two versions. `- final chain: j/N` under a commit is its
    position on the final chain; a commit placed after the reviewed
    snapshot's commit is later than the review even when its date is
    earlier (rebases rewrite dates). A commit marked *merge commit*
    only brings upstream changes in and has no change of its own to
    cite.
```

```
- `M###` marks a message you must classify; `E###` marks an event you
    can cite. A review thread sits at the position of its first message
    with its replies nested. Under an inline comment, `- At:` gives the
    file, the line and the snapshot it was written against; `- Code:`
    says whether that line was added by the pull request, removed by
    it, or unchanged, and shows the end of the hunk the reviewer was
    looking at -- its last line is the commented line.
- `- Labels:` lists only labels that hold. `outdated` means the lines
    the comment anchors to were rewritten later -- by anything, a fix
    or a rebase or an unrelated edit -- so it tells you **where to
    look**, never what happened. `changes_requested` is the state of
    the review that carried the message.
- A diff marked *not collected* could not be fetched. That says
    nothing about what it contained; it is not evidence.

Read the diffs with `grep -n` on a file name or an identifier, then a
    short range around the hit; do not dump whole diff files. **Never
    run the same command twice.** A search that returns nothing is an
    answer -- the code is not there -- so change the query or move on;
    repeating it ends the session without a result.

Work in three steps, in order. The order is what keeps the judgements
    from contaminating each other.

## Step 1 -- classify every message, and extract what it raises

Read the chain from the top. For each `M###`, choose one:

- `defect` -- raises a concrete problem or requested change
- `other` -- everything else: a reply that only explains, accepts,
    rejects or reacts to something already raised; praise, thanks,
    acknowledgments; status, process, bot bookkeeping -- including
    requests about the pull request's mechanics: rebase, resolve merge
    conflicts, update the branch, re-run CI, change the title

Every `M###` in the document gets exactly one. A message you consider
    irrelevant is `other`, never an omission.

From each `defect` message extract what it asks for. One message can
    raise several things; several messages can raise the same thing --
    both are expressed by the messages a defect lists in `from`. Merge
    defects that describe the same underlying issue, however
    differently worded; when you cannot tell, keep them apart.

For each defect record:

- `claim` -- one sentence in your own words, preserving the original's
    modality, conditions and scope. Never turn "may fail" into "fails".
- `category` -- one short word: `correctness`, `security`,
    `performance`, `maintainability`, `documentation`, `testing`,
    `design`, or your own.
- `priority` -- how much the problem itself matters, judged from the
    claim and the code it points at (the hunk under the comment; open
    the snapshot diff if you need more), **never from what happened to
    it afterwards**:
  - `P0` -- merging it would break something: crash, wrong result,
    data loss or corruption, security hole, build failure, failing
    tests.
  - `P1` -- a substantive problem that should be fixed before merge: a
    functional error in the pull request's own logic, however small (a
    condition, guard or check that is applied when it should not be or
    not applied when it should, an incomplete fix that leaves the case
    the pull request set out to handle unhandled), missing edge-case or
    error handling, performance regression, changed behaviour left
```

```
    untested, an API or design mistake, a documentation error that
    would mislead users.
  - `P2` -- an improvement worth making: maintainability, naming,
    comment or documentation accuracy, test readability, duplication, a
    micro-optimisation, an extra test case for behaviour that is
    already tested.
  - `P3` -- a nit, a style or personal preference, or a claim that
    points at no concrete problem at all. Priority is a property of the
    problem, not of how it was said: a question, a request for
    confirmation, a hedge or a blunt demand all get the priority of the
    underlying problem.
- `introduced_in` -- where the code the claim is about came from:
  - `pre_existing` -- it was in the repository before this pull
    request. The comment sits on an unchanged line and the claim is
    about that existing code, or the claim itself says the problem is
    pre-existing, unrelated to this change, or deferred to a follow-up.
    An unchanged line is **not** pre-existing when the problem is
    caused by this pull request's change -- a call site this change
    broke, a test that stopped testing anything once this change
    altered what it calls, a default this change made wrong: the line
    is old, the problem is new, and the answer is the version that
    introduced the change.
  - `S1` -- the pull request's change already contained it at the
    first reviewed snapshot. A comment on an added line of `S1` is this
    without opening anything.
  - `S2`, `S3`, ... or `E###` -- it was introduced only by that later
    snapshot or commit. For a comment written against `S2` or later:
    open `diffs/S1.diff` and look for the same file and code; if it is
    there, the answer is `S1`; if it is not, name the snapshot (or,
    where the commit diffs show it, the commit) that first contains it.
  - `unknown` -- you cannot tell, including when the diff you would
    need is not collected. A pull-request-level comment has no anchor;
    place it by its own words and the diffs, or say `unknown`.

## Step 2 -- for each defect, choose exactly one behaviour

Take each defect **one at a time** and sweep the whole chain for what
    happened to it -- separately, so that you look for evidence about
    *this* defect instead of attaching whatever you happened to read to
    whatever you were reading it for. Choose exactly one:

| behaviour | meaning | verdict |
|---|---|---|
| `accepted` | the PR author, or a maintainer (someone who reviewed,
    approved or merged this pull request), explicitly accepts it or
    states that it has been dealt with -- including "will do in a
    follow-up" | supported |
| `changes_requested` | the review that carried the source message
    requested changes (a printed label) | supported |
| `fixed` | a later commit or snapshot changes the code so that what
    was asked is addressed -- you read the diff | supported |
| `reviewer_conceded` | whoever raised it explicitly retracts or
    withdraws it | opposed |
| `disputed` | the PR author or a maintainer explicitly rejects it, or
    a premise of it, and the pull request goes on without it | opposed |
| `unclear` | nothing bears on it either way, the evidence conflicts,
    or the code you would need is not collected | uncertain |

Rules that settle the hard cases:

1. **`fixed` is read from a diff, never inferred.** Take the defect's
    file and snapshot from its `- At:` line. Open the commits after the
    source message (`diffs/E###.diff` for a later `E###`); where the
    history was force-pushed, compare `diffs/S#.diff` for the source
    snapshot with the next snapshot's or the final head's diff for that
```

```
    file. It is `fixed` when the later change addresses what was asked
    -- an applied `suggestion` block is the plainest case. Lines being
    rewritten is not enough: a rename, a rebase or an unrelated edit
    also rewrites lines. An `outdated` label and a commit message that
    mentions the review both only tell you where to look.
2. **Later supersedes earlier.** An author who pushes back and *then*
    does the work has accepted it, or fixed it if the diff shows it.
3. **`accepted`, `reviewer_conceded` and `disputed` are things people
    said**, and need a quote from an `E###` message: `accepted` from
    the PR author or a maintainer (someone who reviewed, approved or
    merged this pull request), `reviewer_conceded` from whoever raised
    the defect, `disputed` from the PR author or a maintainer -- and
    neither `accepted` nor `disputed` ever from the person who raised
    it. A commit, push, merge or resolved thread is not someone saying
    something.
4. **Answering is not accepting.** When the defect is really a
    question -- "should this use X?", "did the test run twice?" -- a
    reply that answers it has not agreed that anything needs changing.
    `disputed` needs an explicit refusal, or a statement that the code
    is right as it is; an explanation of intent that leaves the request
    open is `unclear`, however confident. The same goes for anything
    aimed at the pull request rather than at this defect: "thanks", "I
    fixed your comments", "I'm expecting a few regressions".
5. **A defect the PR author raised about their own change** cannot be
    `accepted` on the author's own word -- agreeing with oneself is not
    evidence. It is supported through `fixed`, `changes_requested`, or
    a maintainer's `accepted`; otherwise `unclear`.
6. **When more than one applies, take the one backed by code**, then
    the printed label, then words: `fixed` over `changes_requested`
    over `accepted`.

Cite one `event` and one `quote` for every choice except `unclear`:

- for the three spoken behaviours, `event` is the `E###` printed on
    the heading of the message whose words you quote (never its
    `M###`), and `quote` is those words copied character for character;
- for `changes_requested`, `event` is the review-submitted line or the
    source message carrying the label, and `quote` is
    `changes_requested`;
- for `fixed`, `event` is the commit `E###` or the snapshot `S#` whose
    diff shows the change -- when the change is visible only in a
    snapshot diff (after a force-push), cite the `S#`, never the push
    event -- `path` is a file that diff touches, and `quote` is one `+`
    or `-` line copied exactly from **that file's section of that diff
    file**, not from another commit's diff and not a context line. A
    `+` line that already stands in the reviewed snapshot's diff for
    that file is not a fix -- after a rebase the same change reappears
    in a later commit; quote what actually changed after the review, or
    choose `unclear`.

Add `why`, one short sentence. Every cited event must be later than
    every message the defect came from; a snapshot counts as later when
    it was first reviewed after them.

## Step 3 -- read it back

Write `/workspace/result.json`, then open both it and the document
    again and check what you wrote against it: every `M###` classified,
    every `from` ref real, every quote present verbatim where you said
    it was, every cited event later than the messages the defect came
    from, every `introduced_in` either `pre_existing`, `unknown`, or a
    snapshot or commit id printed in the document. Where you chose
    `unclear`, look once more at the diffs after the source message for
    a change to the file it names -- that is the one people most often
    miss.
```

````
Fix what does not hold, then stop.

## Output

**Write your answer to `/workspace/result.json`.** That file is the
    deliverable; nothing you say in the conversation is read. The input
    directory is read-only -- write nothing else, anywhere.

One JSON object, nothing else in the file.

```json
{
  "pr": "owner/repo#123",
  "messages": {"M001": "defect", "M002": "other", "M003": "other"},
  "defects": [
    {"from": ["M001", "M009"],
     "claim": "Handle empty input before reading item 0.",
     "category": "correctness",
     "priority": "P1", "introduced_in": "S1",
     "behavior": "fixed", "event": "E011", "path": "src/parse.py",
     "quote": "+    if not items:",
     "why": "E011 adds the empty-input guard the reviewer asked for."},
    {"from": ["M004"],
     "claim": "The helper could be renamed to say what it returns.",
     "category": "maintainability",
     "priority": "P2", "introduced_in": "pre_existing",
     "behavior": "accepted", "event": "E006",
     "quote": "Good point, I'll do that in a follow-up.",
     "why": "The author agreed and deferred it."}
  ]
}
```

`path` appears only with `fixed`. `event` and `quote` are omitted only
    for `unclear`. A pull request that raises nothing has an empty
    `defects` list -- a valid and common answer.
````

F.4 Reference Curation for Clean Candidates

The clean-candidate prompt additionally asks the agent to classify stances at the final head and collected post-merge callbacks. The controller combines these judgments with review states and the resolution of historical findings to determine clean admission.

```
# Task

You are given one GitHub pull request as a conversation in
    `/workspace/input/pr.md`, and the code it changed as diff files
    under `/workspace/input/diffs/`. It was approved at its final head
    and merged. Read them and write one JSON object to
    `/workspace/result.json` describing what was raised, how much each
    thing matters, where it came from, what happened to it, and what
    was said about the final head.

## What the input contains

`pr.md` opens with a `## Contents` index giving the line number of
    every section. **Use it.** A file viewer truncates a large file
    without telling you which part it kept, so on anything long read by
    range rather than whole -- and if a read comes back marked as
    clipped, go back with ranges until you have seen all of it.
```

```
- `### Snapshots` lists every version of the pull request that
    somebody reviewed, as `S1`, `S2`, ... in the order they were first
    reviewed; the final head is last. `diffs/S#.diff` is the pull
    request's **whole change** at that version, against its merge base
    -- so two snapshots can be compared file by file regardless of
    rebases. The commits in the episode are the final chain only; a
    commit `E###` may point at `diffs/E###.diff`, that commit's own
    change. Where the history was force-pushed, commits from before the
    push are not listed, and only the snapshot diffs show what changed
    between two versions. `- final chain: j/N` under a commit is its
    position on the final chain; a commit placed after the reviewed
    snapshot's commit is later than the review even when its date is
    earlier (rebases rewrite dates). A commit marked *merge commit*
    only brings upstream changes in and has no change of its own to
    cite.
- `M###` marks a message you must classify; `E###` marks an event you
    can cite. A review thread sits at the position of its first message
    with its replies nested. Under an inline comment, `- At:` gives the
    file, the line and the snapshot it was written against; `- Code:`
    says whether that line was added by the pull request, removed by
    it, or unchanged, and shows the end of the hunk the reviewer was
    looking at -- its last line is the commented line.
- `- Labels:` lists only labels that hold. `outdated` means the lines
    the comment anchors to were rewritten later -- by anything -- so it
    tells you **where to look**, never what happened.
    `changes_requested` is the state of the review that carried the
    message. `exact_head` on a message, or on a `review submitted`
    line, means an eligible reviewer -- not the author, not a bot --
    was looking at the final head, the code that was merged.
    `exact_head (local)` means the same on an inline or commit comment,
    which speaks about one place. A message with neither label was
    written against an earlier head, or by someone who cannot certify a
    pull request; it can still raise a defect, and it can still tell
    you what happened to one.
- The document closes with `## Post-merge callbacks`: what referenced
    this pull request after it merged, when a collection was run. A
    card is an Issue that names it, a pull request that reverts it, or
    a commit that reverts it. The section says plainly whether nothing
    was collected, nothing came back, or cards follow. The first two
    are different facts; only the second says anything.
- A diff marked *not collected* could not be fetched. That says
    nothing about what it contained; it is not evidence.

Read the diffs with `grep -n` on a file name or an identifier, then a
    short range around the hit; do not dump whole diff files. **Never
    run the same command twice.** A search that returns nothing is an
    answer -- the code is not there -- so change the query or move on;
    repeating it ends the session without a result.

Work in four steps, in order. The order is what keeps the judgements
    from contaminating each other.

## Step 1 -- classify every message, and extract what it raises

Read the chain from the top. For each `M###`, choose one:

- `defect` -- raises a concrete problem or requested change
- `other` -- everything else: a reply that only explains, accepts,
    rejects or reacts to something already raised; praise, thanks,
    acknowledgments; status, process, bot bookkeeping -- including
    requests about the pull request's mechanics: rebase, resolve merge
    conflicts, update the branch, re-run CI, change the title

Every `M###` in the document gets exactly one. A message you consider
    irrelevant is `other`, never an omission.
```

```
From each `defect` message extract what it asks for. One message can
    raise several things; several messages can raise the same thing --
    both are expressed by the messages a defect lists in `from`. Merge
    defects that describe the same underlying issue, however
    differently worded; when you cannot tell, keep them apart.

For each defect record:

- `claim` -- one sentence in your own words, preserving the original's
    modality, conditions and scope. Never turn "may fail" into "fails".
- The pull request was approved anyway. That is not a reason to soften
    a claim or lower a priority: whether a defect raised against the
    final head still stands is read off your answer, so write it as the
    reviewer meant it.
- `category` -- one short word: `correctness`, `security`,
    `performance`, `maintainability`, `documentation`, `testing`,
    `design`, or your own.
- `priority` -- how much the problem itself matters, judged from the
    claim and the code it points at (the hunk under the comment; open
    the snapshot diff if you need more), **never from what happened to
    it afterwards**:
  - `P0` -- merging it would break something: crash, wrong result,
    data loss or corruption, security hole, build failure, failing
    tests.
  - `P1` -- a substantive problem that should be fixed before merge: a
    functional error in the pull request's own logic, however small (a
    condition, guard or check that is applied when it should not be or
    not applied when it should, an incomplete fix that leaves the case
    the pull request set out to handle unhandled), missing edge-case or
    error handling, performance regression, changed behaviour left
    untested, an API or design mistake, a documentation error that
    would mislead users.
  - `P2` -- an improvement worth making: maintainability, naming,
    comment or documentation accuracy, test readability, duplication, a
    micro-optimisation, an extra test case for behaviour that is
    already tested.
  - `P3` -- a nit, a style or personal preference, or a claim that
    points at no concrete problem at all. Priority is a property of the
    problem, not of how it was said.
- `introduced_in` -- where the code the claim is about came from:
  - `pre_existing` -- it was in the repository before this pull
    request. The comment sits on an unchanged line and the claim is
    about that existing code, or the claim itself says the problem is
    pre-existing, unrelated to this change, or deferred to a follow-up.
    An unchanged line is **not** pre-existing when the problem is
    caused by this pull request's change -- a call site this change
    broke, a test that stopped testing anything once this change
    altered what it calls, a default this change made wrong: the line
    is old, the problem is new, and the answer is the version that
    introduced the change.
  - `S1` -- the pull request's change already contained it at the
    first reviewed snapshot. A comment on an added line of `S1` is this
    without opening anything.
  - `S2`, `S3`, ... or `E###` -- it was introduced only by that later
    snapshot or commit. For a comment written against `S2` or later:
    open `diffs/S1.diff` and look for the same file and code; if it is
    there, the answer is `S1`; if it is not, name the snapshot (or,
    where the commit diffs show it, the commit) that first contains it.
  - `unknown` -- you cannot tell, including when the diff you would
    need is not collected.

## Step 2 -- for each defect, choose exactly one behaviour
```

```
Take each defect **one at a time** and sweep the whole chain for what
    happened to it -- separately, so that you look for evidence about
    *this* defect. Choose exactly one:

| behaviour | meaning | verdict |
|---|---|---|
| `accepted` | the PR author, or a maintainer (someone who reviewed,
    approved or merged this pull request), explicitly accepts it or
    states that it has been dealt with -- including "will do in a
    follow-up" | supported |
| `changes_requested` | the review that carried the source message
    requested changes (a printed label) | supported |
| `fixed` | a later commit or snapshot changes the code so that what
    was asked is addressed -- you read the diff | supported |
| `reviewer_conceded` | whoever raised it explicitly retracts or
    withdraws it | opposed |
| `disputed` | the PR author or a maintainer explicitly rejects it, or
    a premise of it, and the pull request goes on without it | opposed |
| `unclear` | nothing bears on it either way, the evidence conflicts,
    or the code you would need is not collected | uncertain |

Rules that settle the hard cases:

1. **`fixed` is read from a diff, never inferred.** Take the defect's
    file and snapshot from its `- At:` line. Open the commits after the
    source message (`diffs/E###.diff` for a later `E###`); where the
    history was force-pushed, compare `diffs/S#.diff` for the source
    snapshot with the next snapshot's or the final head's diff for that
    file. It is `fixed` when the later change addresses what was asked
    -- an applied `suggestion` block is the plainest case. Lines being
    rewritten is not enough: a rename, a rebase or an unrelated edit
    also rewrites lines. Nothing can fix a defect raised against the
    final head: there is no later diff.
2. **Later supersedes earlier.** An author who pushes back and *then*
    does the work has accepted it, or fixed it if the diff shows it.
3. **`accepted`, `reviewer_conceded` and `disputed` are things people
    said**, and need a quote from an `E###` message: `accepted` from
    the PR author or a maintainer (someone who reviewed, approved or
    merged this pull request), `reviewer_conceded` from whoever raised
    the defect, `disputed` from the PR author or a maintainer -- and
    neither `accepted` nor `disputed` ever from the person who raised
    it. A commit, push, merge or resolved thread is not someone saying
    something.
4. **Answering is not accepting.** A reply that answers a question has
    not agreed that anything needs changing. `disputed` needs an
    explicit refusal, or a statement that the code is right as it is;
    an explanation of intent that leaves the request open is `unclear`,
    however confident.
5. **A defect the PR author raised about their own change** cannot be
    `accepted` on the author's own word. It is supported through
    `fixed`, `changes_requested`, or a maintainer's `accepted`;
    otherwise `unclear`.
6. **When more than one applies, take the one backed by code**, then
    the printed label, then words: `fixed` over `changes_requested`
    over `accepted`.

Cite one `event` and one `quote` for every choice except `unclear`:

- for the three spoken behaviours, `event` is the `E###` printed on
    the heading of the message whose words you quote (never its
    `M###`), and `quote` is those words copied character for character;
- for `changes_requested`, `event` is the review-submitted line or the
    source message carrying the label, and `quote` is
    `changes_requested`;
```

```
- for `fixed`, `event` is the commit `E###` or the snapshot `S#` whose
    diff shows the change -- when the change is visible only in a
    snapshot diff (after a force-push), cite the `S#`, never the push
    event -- `path` is a file that diff touches, and `quote` is one `+`
    or `-` line copied exactly from **that file's section of that diff
    file**, not from another commit's diff and not a context line. A
    `+` line that already stands in the reviewed snapshot's diff for
    that file is not a fix -- after a rebase the same change reappears
    in a later commit; quote what actually changed after the review, or
    choose `unclear`.

Add `why`, one short sentence. Every cited event must be later than
    every message the defect came from.

## Step 3 -- the final head

Two judgements, each over a closed list.

**Every `M###` message labelled `exact_head` or `exact_head (local)`**
    gets exactly one stance, about what *this message* says. Only
    messages: a `review submitted` line that carries `exact_head` but
    has no body is not a message, is counted from its state without
    you, and must not be listed. A document with no labelled message
    has an empty `head`.

| stance | meaning | needs |
|---|---|---|
| `affirmed` | the reviewer explicitly accepts the whole pull request,
    or the final head as it stands -- "LGTM", "looks good, merging", an
    approval body that says so | a quote; only on `exact_head`, never
    on `(local)` |
| `objected` | the reviewer raises a substantive problem against the
    final head, or states that it is not acceptable as it stands | a
    quote |
| `neither` | anything else: a question, a nit waved through, praise
    of one line, thanks, a note about process | nothing; a quote, if
    you give one, must still be verbatim |

Three rules.

1. **Local praise does not certify.** "Nice!" on one line says nothing
    about the rest of the change; it is `neither`. The same words in a
    review body about the whole pull request are `affirmed`.
2. **The review's state is already counted.** `APPROVED` or
    `CHANGES_REQUESTED` on a `review submitted` line is a platform
    fact, read without you. Your stance is about the words. An approval
    whose body only says "approving to unblock" or "rubber-stamp" is
    `neither`; a `COMMENTED` review whose body says "this is good to
    merge" is `affirmed`.
3. **A stance is about one message.** If the reviewer later took an
    objection back, step 2 records that on the defect as
    `reviewer_conceded`; the stance on the message that raised it stays
    `objected`.

**Every card `C###` under `## Post-merge callbacks`** gets exactly one
    verdict:

| verdict | meaning |
|---|---|
| `negative_callback` | the card's own text clearly attributes a
    concrete malfunction, regression or unfixed defect to this pull
    request, **or** it reverts this pull request's change |
| `neutral_or_positive` | this pull request fixes or resolves the
    Issue, is merely referenced, or the blame is placed elsewhere; for
    a revert card, the text says the revert is not about a defect in
```

```
    this change (an accidental merge, reverting something else,
    re-landing the same change unchanged) |
| `uncertain` | the text does not say which way |

For an `issue` card, a cross-reference, a post-merge date, or the word
    "bug" is not enough. For a `revert pull request` or `revert commit`
    card the revert is itself the blame: a bare "Revert #N" with no
    stated reason is `negative_callback`, quoting the line that reverts
    it. Whether a revert merged is a platform fact printed on the card
    and read without you; your verdict is about the words. A
    `negative_callback` needs a `quote` copied from the card; the other
    two need none. Prefer `uncertain` over guessing. When the section
    says nothing was collected, or nothing came back, `callbacks` is an
    empty list.

## Step 4 -- read it back

Write `/workspace/result.json`, then open both it and the document
    again and check what you wrote against it: every `M###` classified,
    every `from` ref real, every labelled message given a stance, every
    card given a verdict, every quote present verbatim where you said
    it was, every cited event later than the messages the defect came
    from, every `introduced_in` either `pre_existing`, `unknown`, or a
    snapshot or commit id printed in the document.

Fix what does not hold, then stop.

## Output

**Write your answer to `/workspace/result.json`.** That file is the
    deliverable; nothing you say in the conversation is read. The input
    directory is read-only -- write nothing else, anywhere.

One JSON object, nothing else in the file.

```json
{
  "pr": "owner/repo#123",
  "messages": {"M001": "defect", "M002": "other", "M003": "other"},
  "defects": [
    {"from": ["M001"],
     "claim": "Handle empty input before reading item 0.",
     "category": "correctness",
     "priority": "P1", "introduced_in": "S1",
     "behavior": "fixed", "event": "E011", "path": "src/parse.py",
     "quote": "+    if not items:",
     "why": "E011 adds the empty-input guard the reviewer asked for."}
  ],
  "head": [
    {"message": "M003", "stance": "affirmed",
     "quote": "Looks good to me, merging."}
  ],
  "callbacks": [
    {"card": "C001", "verdict": "neutral_or_positive"}
  ]
}
```

`path` appears only with `fixed`. `event` and `quote` on a defect are
    omitted only for `unclear`. `head` lists every labelled `M###` once
    and never an `E###`; `callbacks` lists every card once. Either is
    `[]` when there is nothing to list. A pull request that raises
    nothing has an empty `defects` list -- a valid and common answer on
    this route.
```

F.5 BENCHMARK FINDING MATCHING

Benchmark scoring uses the pairwise matching prompt from Martian (Zverianskii et al., 2026), run by GPT-5.6-Sol. Each call compares one generated finding with one benchmark reference and returns a Boolean match, reasoning, and confidence. This matching decision is used by the benchmark scorer separately from the verifier's 0–10 score.

The system message is:

```
You are a precise code review evaluator. Always respond with valid
    JSON.
```

The user-message template is:

```
You are evaluating AI code review tools. Determine if the candidate
    issue matches the golden (expected) comment.

Golden Comment (the issue we're looking for):
{golden_comment}

Candidate Issue (from the tool's review):
{candidate}

Instructions:
- Determine if the candidate identifies the SAME underlying issue as
    the golden comment
- Accept semantic matches - different wording is fine if it's the same
    problem
- Focus on whether they point to the same bug, concern, or code issue

Respond with ONLY a JSON object:
{{"reasoning": "brief explanation", "match": true/false, "confidence":
    0.0-1.0}}
```