

How Reliable Is LLM-as-Judge for Patch Correctness Assessment? An Empirical Study

Shanggui Zhan  

School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Xingqi Wang¹  

School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Dan Wei 

School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Xin Xiang 

School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China

Abstract

Background. LLM-as-Judge is increasingly adopted in automated program repair (APR) to assess patch correctness, yet its validity as a measurement instrument remains unaudited: potential benchmark-identifier leakage, evidence-presentation bias, and explanation unreliability have not been systematically examined.

Aims. We investigate whether LLM Judge produces unbiased, stable, and well-grounded patch correctness judgments across three validity dimensions: metadata leakage, evidence presentation, and explanation reliability.

Method. On a balanced, paired dataset of 326 patches from 163 Defects4J bugs, we evaluate GPT-4o and DeepSeek-V3 across a 12-setting prompt matrix covering benchmark-identifying metadata exposure and evidence presentation variants. We further conduct a manual analysis of 60 GPT-4o explanations using a five-category failure taxonomy and three independent annotators.

Results. Under sanitized conditions, GPT-4o achieves an accuracy of 77.3% (MCC = 0.575, FPR = 17.2%), while DeepSeek-V3 achieves 80.4% accuracy (MCC = 0.625, FPR = 22.1%), with no statistically significant difference between the two models. Exposing combined benchmark-identifying metadata yields a suggestive accuracy increase of 4.3 percentage points for GPT-4o and 1.8 percentage points for DeepSeek-V3. Declaring that all tests have passed does not significantly bias either model; in contrast, providing a developer reference patch leads to substantial and statistically significant improvements for both models. Explanation failures are concentrated in misclassified cases, with the false-positive quadrant showing a 93.3% failure rate, primarily driven by hallucinated evidence and missed edge cases.

Conclusions. LLM Judge is useful as a triage aid but is insufficient as a standalone correctness oracle. Reliable deployment requires prompt sanitization, explicit false positive rate reporting, clear distinction between reference-assisted and standalone assessment, and treating LLM explanations as investigative hypotheses rather than self-validating justifications.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Software and its engineering → Empirical software validation

Keywords and phrases automated program repair, patch correctness assessment, LLM-as-Judge, empirical study

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.21

¹ Corresponding author.



© Shanggui Zhan, Xingqi Wang, Dan Wei, and Xin Xiang;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 21; pp. 21:1–21:20



Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Category Technical Track Paper

Supplementary Material *Software (Source Code and dataset)*: https://github.com/Polaris-sg/llm_judge_Study; archived at `swb:1:dir:c2d9f88b0987fbddfd1b0a14ba11089186dba88c`

1 Introduction

Automated Program Repair (APR) aims to automatically generate patches that fix software bugs, but assessing the correctness of these patches is a critical challenge for deploying APR tools in real-world scenarios [26, 9, 39, 28]. While existing test suites can verify whether a patch passes all tests, they cannot guarantee that the patch truly fixes the defect rather than merely passing the tests [3]. This phenomenon, known as overfitting, occurs when a patch may pass the tests but does not align with the developer’s intent. To address this issue, Automated Patch Correctness Assessment (APCA) techniques have been proposed to tackle the overfitting problem in APR [14].

In recent years, the rapid advancement of large language models (LLMs) has catalyzed a paradigm shift within the APCA landscape, moving from the exploration of learning-based classifiers toward the adoption of LLM-as-Judge (LLM Judge) approaches [36]. LLM4PatchCorrect [40] demonstrated that in-context learning with LLMs outperforms specialised classifiers on Defects4J, achieving over 84.4% accuracy. RePaCA [5] subsequently reported 83.1% accuracy by combining reasoning LLMs with reinforcement learning for static patch assessment. Beyond standalone APCA methods, LLM Judge is increasingly used as a de facto assessment layer in agentic APR pipelines [23], in which a separate model validates each candidate patch before it is accepted. The appeal is clear: LLM Judge requires no task-specific training data, generalises across projects and languages, and produces human-readable explanations that researchers can inspect.

Despite this widespread adoption, the reliability of LLM Judge as a measurement instrument has received virtually no rigorous scrutiny. Existing studies typically treat patch-assessment accuracy as sufficient evidence of trustworthiness. However, accuracy alone cannot rule out three fundamental threats to measurement validity. First, LLMs are trained on large-scale corpora that may contain Defects4J bug reports, patches, and correctness labels. Exposing a bug identifier or project name in the prompt may enable the model to rely on metadata-induced shortcuts, including possible memorization signals, instead of reasoning from the code alone. Such shortcuts can inflate apparent accuracy through benchmark leakage [15]. Second, the order and content of presented evidence can affect LLM judgments in general SE tasks [38]. In APR, a declarative statement that all tests passed, or the inclusion of a developer reference patch, may systematically bias the LLM toward labeling overfitting patches as correct. This bias can increase false positive rates in ways that remain hidden when only aggregate accuracy is reported. Third, even when the predicted label is correct, the accompanying natural-language explanation may include hallucinated code references or unsupported reasoning steps. Such explanations can undermine the validity of human-in-the-loop review processes that rely on them [8]. Wang et al. recently showed that test-based patch assessment in SWE-bench has a 7.8% false-correct rate and 29.6% behavioral divergence from human evaluation [27]. If test-based assessment is unreliable, it remains unclear whether LLM Judge can serve as a trustworthy substitute or instead introduces different forms of unreliability.

In this paper, we seek to clarify these questions by systematically examining the reliability of LLM-as-Judge for patch correctness assessment. Specifically, we formulate four rigorously controlled research questions to evaluate the stability, impartiality, and groundedness of

LLM Judge from the perspective of empirical software engineering measurement. To address these questions, we construct a balanced, paired dataset comprising 326 patches generated by APR tools for Defects4J [10] bugs. We evaluate GPT-4o and DeepSeek-V3 across varied prompting configurations spanning two dimensions: benchmark-identifier leakage and evidence-presentation bias. In addition, we conduct a manual analysis of 60 GPT-4o-generated explanations, applying a five-category failure taxonomy to assess the reliability of the produced rationales. Our study newly raises and answers the following four research questions.

RQ1: What is the agreement between LLM Judge and human ground-truth labels under sanitized conditions? (3.1) This establishes the baseline reliability of LLM Judge when no confounding information is present.

RQ2: Does exposing benchmark-identifying information (bug ID, project name, repair tool name, or developer patch) to LLM Judge systematically shift its judgments? (3.2) If judgments shift toward ground truth upon receiving identifiers the model may have encountered during pre-training, the effective accuracy may be influenced by metadata-induced shortcuts rather than code reasoning alone.

RQ3: Does the type and content of evidence included in the prompt systematically bias LLM Judge toward labelling overfitting patches as correct? (3.3) We distinguish two sub-forms: test-passing bias (inflated FPR when the prompt states that all tests pass) and reference-similarity bias (inflated correct-label rate for patches similar to the developer reference).

RQ4: Do the natural language explanations produced by LLM Judge provide trustworthy reasoning, or do they contain hallucinated or unsupported claims? (3.4) This question assesses whether explanations can serve as a valid basis for human-in-the-loop review.

In summary, we make the following contributions:

- We empirically demonstrate that LLM Judge judgments for patch correctness assessment are sensitive to benchmark-identifying information exposed in the prompt, including bug identifiers, project names, repair tool names, and developer patches.
- We show that evidence presentation can systematically distort LLM Judge decisions by separating test-passing bias from reference-similarity bias.
- We introduce an explanation reliability analysis based on 60 manually annotated GPT-4o explanations, three independent annotators, structured adjudication, and a five-category failure taxonomy. The analysis shows that explanation failures concentrate in misclassified cases and differ substantially across confusion-matrix quadrants.
- We provide concrete protocol recommendations for APR researchers who use LLM Judge as a patch evaluation instrument.

2 Study Design

2.1 Dataset Construction

Our study dataset builds upon the dataset introduced by Lin et al. [16], which comprises 1,179 patches spanning 356 Defects4J bugs produced by 23 APR tools and human developers, with ground-truth correctness labels established through rigorous manual inspection. Using this raw source directly would introduce two confounds that are particularly harmful for a reliability study: (1) Class imbalance (533 correct vs. 646 overfitting patches, with substantial per-bug skew) would make precision, recall, and F1 difficult to interpret without controlling

for prior probability. (2) Within-bug correlation would inflate agreement statistics: multiple patches from the same bug share identical buggy code, test suites, and contextual information, so LLM judgments on them are not independent observations.

To eliminate both confounds, we randomly retain only bugs that contain at least one correct and at least one overfitting patch. When multiple candidate patches exist within a given class, we prefer patches from APR tools over developer patches, as the former carry an independent developer-written reference patch that is used in RQ3. The resulting main dataset comprises 326 patches drawn from 163 Defects4J bugs (Chart: 18, Closure: 55, Lang: 29, Math: 53, Time: 8), with a 1:1 correct-to-overfitting ratio. Of the 163 correct patches, 54 originate from APR tools and are accompanied by an independent developer reference patch, while the remaining 109 are developer patches. The 1:1 balance ensures that accuracy, precision, recall, F1, and MCC are directly interpretable without class-prior adjustment, and the paired structure enables within-bug McNemar’s tests to control for bug context.

2.2 Prompt Design

Sanitized Baseline Prompt (S0/E0.). All experiments share a common sanitized base prompt that contains only information directly derivable from the patch itself: the buggy function source code and the unified-diff patch. The LLM is instructed to output a strict JSON object with three fields: `label` $\in \{\text{correct}, \text{overfitting}, \text{uncertain}\}$, `confidence` $\in [1, 5]$, and a 2–4 sentence explanation. The `uncertain` label allows the model to abstain when the provided evidence is insufficient to classify a patch as either `correct` or `overfitting`; it is not a ground-truth class in our dataset.

Leakage Settings for RQ2. The inputs of APCA tools may combine patch code with auxiliary artifacts, which may directly or indirectly expose benchmark-identifying information [40]. To measure benchmark-identifying information leakage, we define six main prompt variants that progressively expose meta-information about the patch origin, plus two counterfactual controls, as shown in Table 1. Each variant appends one or more information blocks to S0. S5 is treated as an upper-bound reference because the developer patch constitutes a near-complete answer. The S4R control randomly reassigns the complete bug-ID/project/tool metadata triple to a different patch using a fixed seed, while S4F replaces the real bug ID with a non-existent benchmark-style identifier and keeps the real project and tool names. These controls test whether any S4 shift depends on the correct metadata linkage or merely on the presence of benchmark-looking text.

■ **Table 1** Leakage prompt settings (RQ2). Each variant appends the listed information to the sanitized baseline S0.

Setting	Additional Information	Purpose
S0	(none – sanitized baseline)	Baseline
S1	Bug identifier (e.g., <code>Math-1</code>)	Bug-ID leakage
S2	Project name (e.g., Apache Commons Math)	Project leakage
S3	Repair tool name (e.g., <code>TBar</code>)	Tool leakage
S4	Bug ID + project + tool	Full ID leakage
S4R	Shuffled bug ID + project + tool	Counterfactual metadata control
S4F	Fake bug ID + real project + tool	Fake-ID format control
S5	Developer/reference patch diff	Reference leakage

Evidence Settings for RQ3. To measure evidence-presentation bias, we define five prompt variants that vary the type of evidence provided alongside the patch, as shown in Table 2. **E2** is the critical test-passing bias probe: it exposes only the fact that all tests passed, without any test code, so the LLM cannot reason about test semantics. **E4** provides the developer patch, enabling a reference-similarity bias analysis separate from the leakage experiment.

■ **Table 2** Evidence prompt settings (RQ3). Each variant appends the listed evidence to the sanitized baseline **E0**.

Setting	Additional Evidence	Bias Probe
E0	(none – patch only)	Baseline
E1	Failing test source code	Test-semantic reasoning
E2	Declaration that all tests passed	Test-passing bias
E3	Passed/failed test list (execution summary)	Execution evidence
E4	Developer/reference patch diff	Reference-similarity bias

2.3 LLM Configuration

In the main experiments for RQ1–RQ3, we evaluate two judge models, GPT-4o [7] and DeepSeek-V3 [2], both configured with a temperature of 0 to reduce sampling variability [21]. GPT-4o represents a widely used closed-source model, whereas DeepSeek-V3 represents a widely used open-source model. Both models receive identical prompts and are evaluated on the same 326-patch dataset across the complete prompt matrix (**S0–S5**, **S4R/S4F**, and **E0–E4**); because **E0** is identical to **S0**, this yields 12 executed settings per model.

2.4 Explanation Failure Taxonomy

For RQ4, we define a five-category failure taxonomy, summarized in Table 3. Three annotators (three authors of this paper) independently apply this taxonomy to a stratified sample of 60 explanations drawn from the **S0** results, with 15 explanations sampled from each confusion-matrix quadrant (TP, TN, FP, and FN). For each explanation, annotators record whether any failure is present, assign zero or more taxonomy-defined failure types, and rate the severity on an ordinal scale (none, minor, major, critical).

2.5 Experimental Metrics

In RQ1, we report Accuracy, macro-F1 [22], Matthews Correlation Coefficient (MCC) [18], Cohen’s Kappa [1] (against human labels), False Positive Rate (FPR: proportion of overfitting patches mislabelled as correct), and uncertain rate (proportion of patches labelled **uncertain**). Because the dataset contains only **correct** and **overfitting** ground-truth labels, **uncertain** predictions are counted as errors for accuracy, macro-F1, MCC, and Cohen’s Kappa. Note that FPR is the primary safety-critical metric in this setting: each false positive accepts an overfitting patch as correct, thereby inflating reported repair effectiveness.

To analyze the impact of benchmark-identifying information leakage and evidence presentation bias on LLM Judge judgments, we introduce Label Flip Rate, Flip Direction, Odds Ratio (OR), ΔAcc and ΔFPR metrics in RQ2 and RQ3. Here, Label Flip Rate denotes the proportion of patches whose predicted labels change relative to the baseline setting (**S0** or **E0**); Flip Direction indicates the number of flips toward the ground-truth label (Better) or

■ **Table 3** Explanation failure taxonomy.

Failure Type	Definition
Hallucinated Evidence	Explanation references variables, methods, logic, or behavioral claims absent from or unsupported by the provided code.
Test Overtrust	Explanation uses “tests pass” as the primary or sole justification for correctness (circular argument).
Behavioral Misunderstanding	Explanation mischaracterises what the patch actually does semantically.
Missed Edge Cases	Explanation fails to identify boundary-input behavior where the patch diverges from the correct fix.
Contradictory Explanation	Explanation is internally inconsistent, or contradicts the final label.

away from it (Worse) relative to the baseline; Odds Ratio is the ratio of toward-ground-truth flips to away-from-ground-truth flips; and ΔAcc and ΔFPR represent the changes in accuracy and FPR relative to the baseline setting, respectively.

We further apply McNemar’s test [19] (paired nominal, two-sided) to each baseline–variant label pair, comparing S_0 with each S_i for RQ2 and E_0 with each E_i for RQ3. We report Holm-corrected p -values (p_{Holm}) to assess whether the observed label shifts relative to the baseline are statistically significant. For RQ3, we additionally compute a Test-passing Bias Index = $(\text{FPR}_{E2} - \text{FPR}_{E0}) / \text{FPR}_{E0}$, and for E4 we stratify patches by token-level similarity to the developer patch and examine whether flip rate correlates with similarity quartile. In RQ4, we assess inter-annotator agreement using Fleiss’ Kappa [4] for the binary failure label, Krippendorff’s Alpha [12] for nominal failure types and ordinal severity ratings.

3 Results

3.1 RQ1: Baseline Agreement

We begin by examining how closely the two LLM judges (i.e., GPT-4o and DeepSeek-V3) align with human ground-truth labels when all benchmark-identifying information has been removed from the prompt. Table 4 reports the agreement metrics for GPT-4o and DeepSeek-V3 under the sanitized S_0 setting on the 326-patch balanced dataset. As shown in Table 4, DeepSeek-V3 obtains slightly higher accuracy, Macro-F1, MCC, and Cohen’s Kappa than GPT-4o; however, this advantage comes at the cost of a higher false-positive rate, indicating that DeepSeek-V3 is more prone to accepting overfitting patches as correct.

To better understand where these aggregate differences originate, Table 5 presents the full confusion matrices. GPT-4o correctly identifies 125 of 163 correct patches (recall = 76.7%) and 127 of 163 overfitting patches (recall = 77.9%), yielding a relatively symmetric error profile. DeepSeek-V3, by contrast, exhibits an asymmetric pattern: it achieves higher recall on correct patches ($140/163 = 85.9\%$) but lower recall on overfitting patches ($122/163 = 74.8\%$). This asymmetry is directly reflected in its elevated FPR (22.1% vs. 17.2%): DeepSeek-V3 is more inclined to label patches as correct, which recovers more true positives at the expense of also admitting more overfitting patches.

■ **Table 4** RQ1 baseline agreement metrics (S0).

Metric	GPT-4o	DeepSeek-V3
Accuracy	77.3%	80.4%
Macro-F1	0.796	0.817
MCC	0.575	0.625
Cohen’s Kappa	0.571	0.620
FPR (overfitting→correct)	17.2%	22.1%
Uncertain rate	5.8%	3.4%

■ **Table 5** RQ1 confusion matrices (S0). Rows are ground truth; columns are LLM predictions.

Model		Corr.	Overfit.	Unc.	Total
GPT-4o	Correct	125	27	11	163
	Overfitting	28	127	8	163
DeepSeek-V3	Correct	140	17	6	163
	Overfitting	36	122	5	163

The **uncertain** label is used sparingly by both models, though GPT-4o invokes it more frequently than DeepSeek-V3 (5.8%, 19 patches vs. 3.4%, 11 patches). Despite these distributional differences, a paired cross-model McNemar comparison on the same 326 patches reveals no statistically significant difference in overall accuracy ($p = 0.164$). DeepSeek-V3 is correct where GPT-4o is wrong on 26 patches, whereas GPT-4o is correct where DeepSeek-V3 is wrong on 16 patches (OR = 1.63); the two models produce different labels on 48 patches in total. In other words, the two models exhibit comparable baseline reliability, albeit through markedly different error profiles.

■ **Table 6** Confidence-stratified RQ1 results under S0. FP/OF denotes false-positive acceptances over all overfitting patches in the confidence band.

Model	Conf.	N	Acc.	FP/OF	FPR
GPT-4o	1-3	19	0.0%	0/8	0.0%
	4-5	307	82.1%	28/155	18.1%
DeepSeek-V3	1-3	16	31.3%	0/10	0.0%
	4-5	310	82.9%	36/153	23.5%

As shown in Table 6, confidence provides only a limited triage signal: lower-confidence predictions should be prioritized for human review, but false-positive acceptances from both models still occur among high-confidence predictions. Therefore, confidence cannot validate a **correct** judgment or exempt high-confidence accepted patches from verification.

Answer to RQ1

LLM Judge provides a useful but incomplete signal under sanitized conditions. GPT-4o achieves 77.3% accuracy (MCC = 0.575) with FPR = 17.2%, while DeepSeek-V3 achieves 80.4% accuracy (MCC = 0.625) with FPR = 22.1%; the difference in overall accuracy is not statistically significant ($p = 0.164$). The two models fail in different ways: GPT-4o is more conservative, whereas DeepSeek-V3 accepts more correct patches but also more overfitting patches. Thus, neither model is reliable enough to serve as an unsupervised patch-correctness oracle, although both can provide a triage-level assessment signal.

3.2 RQ2: Benchmark-Identifying Information Leakage

In RQ2, we investigate whether exposing benchmark-identifying metadata alters model judgments beyond what is already inferable from the sanitized patch context alone. Table 7 summarizes the effect of progressively reintroducing such metadata into the prompt. Specifically, each row reports the change in accuracy, the change in FPR, the number of label flips relative to S0, and the Holm-corrected McNemar p -value. **GPT-4o**. For GPT-4o, the

■ **Table 7** RQ2 leakage results. “Better” / “Worse” count patches that flip to / from the ground-truth label vs. S0. “OR” is the odds ratio of flipping toward vs. away from ground truth. p_{Holm} applies Holm correction within each model’s test family.

Model	Setting	ΔAcc	ΔFPR	Better	Worse	OR	p_{Holm}
GPT-4o	S1 +bug ID	+2.1	+1.8	14	7	2.00	0.946
	S2 +project	+0.6	+2.5	10	8	1.25	1.000
	S3 +tool	+2.1	−5.5	18	11	1.64	0.946
	S4 +all IDs	+4.3	−5.5	20	6	3.33	0.056
	S4R +shuffled IDs	−1.5	−4.9	16	21	0.76	1.000
	S4F +fake ID	+2.1	−4.9	16	9	1.78	0.946
	S5 +dev patch	+10.4	−11.0	40	6	6.67	< 0.001
DeepSeek-V3	S1 +bug ID	+1.2	−0.6	12	8	1.50	1.000
	S2 +project	−0.6	+2.5	5	7	0.71	1.000
	S3 +tool	+2.1	−5.5	15	8	1.88	1.000
	S4 +all IDs	+1.8	−0.6	11	5	2.20	1.000
	S4R +shuffled IDs	+0.3	−2.5	12	11	1.09	1.000
	S4F +fake ID	+2.1	−4.9	14	7	2.00	1.000
	S5 +dev patch	+7.1	−9.8	32	9	3.56	0.003

individual identifiers in S1–S3 produce only small, non-significant changes, indicating that no single metadata field yields a measurable shift. Combining the identifiers in S4 instead produces a positive shift that is marginal after Holm correction; we therefore interpret it as suggestive rather than conclusive evidence of metadata sensitivity. The shuffled-metadata and fake-ID controls do not reproduce the full S4 pattern, so this signal is not explained solely by benchmark-looking text and does not constitute direct evidence of memorisation. By contrast, S5 produces the largest, statistically significant improvement, showing that developer-reference access substantially amplifies agreement.

DeepSeek-V3. DeepSeek-V3 shows a similar directional tendency under S3 and S5, but its metadata-only effects are more attenuated. Neither the individual identifiers nor their combination in S4, including the counterfactual controls, is significant after Holm correction. This provides little evidence of identifier-driven sensitivity for this model on our dataset. In contrast, S5 again produces a large, significant improvement, confirming that the developer-reference effect is robust across both models.

Cluster-aware robustness. Because each bug contributes one correct and one overfitting patch, we additionally resampled bug-level pairs with replacement and computed 95% bootstrap confidence intervals for the core comparisons. For GPT-4o, the S4 accuracy shift remains positive, while S4R and S4F cross zero. For DeepSeek-V3, all three metadata-only intervals cross zero. By contrast, the developer-reference conditions are stable across clusters for both models. These intervals support a conservative reading: S4 is a suggestive GPT-4o metadata shift, whereas S5 is a robust reference-patch effect.

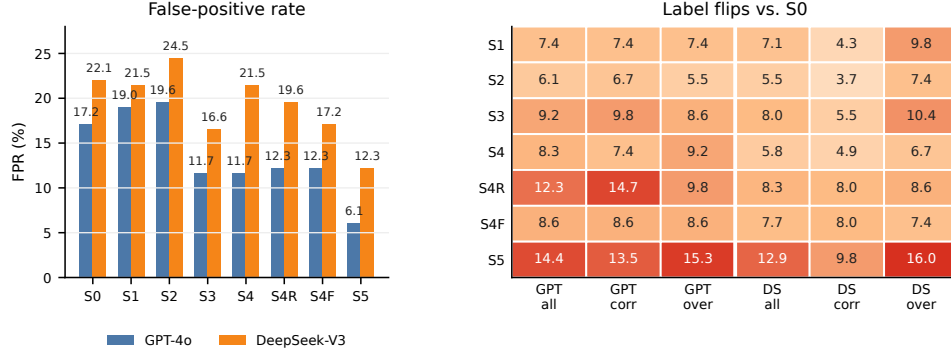


Figure 1 RQ2 metadata-leakage effects for GPT-4o and DeepSeek-V3. The left panel reports false-positive rates; the right panel reports label-flip rates relative to S0.

Fig. 1 visualises these patterns from two complementary perspectives. As shown in the left panel of Fig. 1, DeepSeek-V3 consistently maintains a higher FPR than GPT-4o across all metadata exposure conditions, even when accuracy is comparable. The right panel shows that the label-flip rate is highest for S5 (developer patch) for both models, with a pronounced skew toward flipping overfitting patches to correct, which is consistent with a strong reference-similarity effect.

Answer to RQ2

Benchmark-identifying metadata can shift LLM Judge outcomes, but the effect is uneven across metadata types and models. Individual bug IDs, project names, and tool names do not produce significant shifts; however, their combination in S4 yields a suggestive GPT-4o improvement, which becomes marginal after Holm correction ($p_{\text{Holm}} = 0.056$) and does not appear for DeepSeek-V3. The shuffled-metadata and fake-ID controls do not reproduce the full S4 pattern, suggesting a cautious metadata-induced-shift interpretation rather than a strong causal claim. By contrast, exposing the developer patch in S5 produces large and significant gains for both models ($p \leq 0.003$). Therefore, including benchmark-identifying information, especially developer patches, can overestimate LLM Judge reliability by up to 10 pp.

3.3 RQ3: Evidence-Presentation Bias

In RQ3, we investigate whether different forms of supporting evidence systematically alter LLM Judge behaviour when benchmark-identifying metadata remains sanitized. Table 8 summarizes the effect of each evidence variant relative to E0. Both models are evaluated under all four evidence conditions.

Test code (E1). As shown in Table 8, failing test source code produces model-dependent behaviour: the two models move in different accuracy directions, while both show higher FPR. Although neither within-model change is significant after Holm correction, the cross-model paired comparison is significant, indicating that the models interpret raw test code in qualitatively different ways.

Test-passing bias (E2). Contrary to our initial hypothesis, declaring that all tests passed does not significantly alter either model’s judgments. The corresponding test-passing bias is small and directionally inconsistent, indicating that a test-passing declaration alone is not strong evidence of correctness for either model.

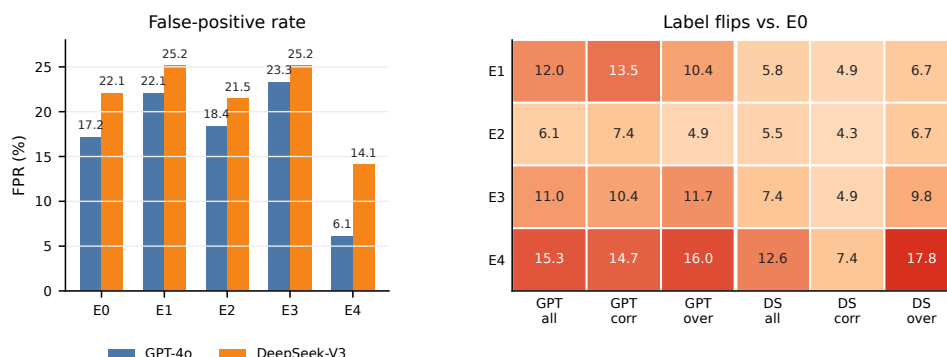
■ **Table 8** RQ3 evidence bias results. p_{Holm} applies Holm correction within each model’s test family.

Model	Setting	ΔAcc	ΔFPR	Better	Worse	OR	p_{Holm}
GPT-4o	E1 +test code	−2.8	+4.9	13	22	0.59	0.526
	E2 +tests pass	+1.2	+1.2	10	6	1.67	0.909
	E3 +exec info	+0.6	+6.1	17	15	1.13	0.909
	E4 +dev patch	+9.5	−11.0	40	9	4.44	<0.001
DeepSeek-V3	E1 +test code	+0.9	+3.1	9	6	1.50	1.000
	E2 +tests pass	+0.9	−0.6	9	6	1.50	1.000
	E3 +exec info	+0.6	+3.1	11	9	1.22	1.000
	E4 +dev patch	+7.1	−8.0	31	8	3.88	0.001

Execution information (E3). Including passed/failed test lists produces a directionally similar but statistically weaker pattern: FPR increases for both models, but neither increase remains significant after correction.

Reference patch (E4). Consistent with the S5 findings in RQ2, providing the developer patch produces the largest and statistically significant effect among all evidence variants: accuracy improves and FPR decreases for both models. This pattern indicates that the developer patch serves as a strong reference anchor rather than ordinary contextual evidence.

The cluster-aware robustness analysis gives the same practical interpretation for the two core RQ3 probes. For E2, accuracy intervals cross zero for both models, confirming that a test-passing declaration alone is not a robust source of accuracy shift. For E4, the developer-reference effect remains stable under bug-level resampling.



■ **Figure 2** RQ3 evidence-presentation effects for GPT-4o and DeepSeek-V3. The left panel reports false-positive rates; the right panel reports label-flip rates relative to E0.

To test whether the reference effect is simply driven by lexical similarity, we stratified the patches with an available developer reference into token-similarity quartiles. Flip rates under both E4 and S5 vary non-monotonically across these quartiles. Thus, the developer patch clearly improves judgments, but the mechanism is not reducible to a simple token-level similarity effect. Fig. 2 reinforces this contrast visually. As shown in the left panel of Fig. 2, the FPR increases for both models under E1 and E3, while remaining relatively stable under E2. The right panel of Fig. 2 shows that the label-flip rate is highest for E4 (developer patch) for both models, with a pronounced skew toward flipping overfitting patches to correct, which is consistent with a strong reference-similarity effect.

Answer to RQ3

We find little evidence that a bare test-passing statement is sufficient to bias LLM Judge decisions. Declaring that all tests passed (E2) does not significantly shift either model (GPT-4o $p = 0.909$; DeepSeek-V3 $p = 1.000$), and the Test-passing Bias Index is small and directionally inconsistent (+7.1% for GPT-4o and -2.8% for DeepSeek-V3). Raw test code and execution traces have more model-dependent effects and tend to increase FPR, but most within-model changes are not significant. The only consistently strong evidence effect comes from the developer reference patch (E4), which significantly improves accuracy for both models ($p \leq 0.001$). This effect is best interpreted as a near-oracle comparison signal rather than ordinary evidence-presentation bias, and its benefit is not reducible to token-level lexical similarity.

3.4 RQ4: Exploratory Explanation Reliability

To assess explanation reliability, we conducted an exploratory manual analysis of 60 LLM explanations sampled from the S0 results (15 per confusion-matrix quadrant) using the five-category failure taxonomy defined in Table 3. Three annotators independently labelled each explanation, and disagreements were resolved through a structured adjudication process. Because the sample is limited to GPT-4o under S0 and the annotation task exhibits low-to-fair agreement, we treat RQ4 as an exploratory audit of explanation risks rather than as a definitive estimate of explanation failure prevalence across models and prompts.

■ **Table 9** RQ4 inter-annotator agreement (3 annotators, 60 samples).

Metric	Value
Fleiss' Kappa (has_failure)	0.237
Krippendorff's α (has_failure)	0.241
Krippendorff's α (severity)	0.279
Exact 3-way severity agreement	28/60 (46.7%)
Full unanimous annotation	26/60 (43.3%)
Adjudication required	34/60 (56.7%)

Inter-Annotator Agreement. We first examine inter-annotator agreement (IAA) to assess the inherent subjectivity of the annotation task, with the results reported in Table 9. As shown in Table 9, agreement on the binary failure judgment is low to fair, with Fleiss' Kappa of 0.237 and Krippendorff's Alpha of 0.241. Agreement on severity is similarly modest, with Krippendorff's Alpha of 0.279. Full three-way unanimity is reached for only 26 of the 60 samples (43.3%), while the remaining 34 samples (56.7%) require adjudication. These results indicate that judging whether an LLM explanation is well grounded is an inherently subjective and challenging task, even for trained annotators. They also highlight a practical concern for human-in-the-loop review: the same explanation may be judged as hallucinated or acceptable depending on how strictly evidential support is interpreted. To resolve disagreements, we use a structured adjudication process. We first compute a majority-vote consensus and then ask the three annotators to jointly review all 34 disagreement cases. Final updates are accepted only when at least two annotators endorse the same revision. This process yields five accepted adjudication updates, while nine minority concerns remain at the preliminary consensus level.

Consensus Results. After adjudication, 20 of the 60 explanations (33.3%) were found to contain at least one failure. As shown in Table 10, the contrast across quadrants is pronounced but should be interpreted with caution. In this stratified sample, explanations

21:12 How Reliable Is LLM-as-Judge for Patch Correctness Assessment?

■ **Table 10** RQ4 explanation failure rate by quadrant.

Quadrant	Failures	Total	Rate
TP (correct, pred correct)	0	15	0.0%
TN (overfitting, pred overfitting)	0	15	0.0%
FP (overfitting, pred correct)	14	15	93.3%
FN (correct, pred overfitting)	6	15	40.0%
Overall	20	60	33.3%

■ **Table 11** RQ4 failure type prevalence and distribution by quadrant.

Failure Type	Count	Rate	TP	TN	FP	FN
Hallucinated Evidence	14	23.3%	0	0	9	5
Behavioral Misunderstanding	7	11.7%	0	0	5	2
Missed Edge Cases	10	16.7%	0	0	10	0
Test Overtrust	0	0.0%	0	0	0	0
Contradictory Explanation	0	0.0%	0	0	0	0

associated with correctly classified patches (TP and TN) contain no failures identified by consensus, whereas failures are concentrated among explanations for misclassified patches: 93.3% in the FP quadrant and 40.0% in the FN quadrant. The Fisher exact test indicates that the failure rate in the FP quadrant is significantly higher than that in the TP quadrant ($p < 0.001$), and that the failure rate in the FN quadrant is also significantly higher than that in the TN quadrant ($p = 0.035$). This result does not imply that all explanations with correct labels are reliable; rather, it indicates only that our consensus process did not identify formal failures among the 30 sampled TP/TN explanations.

■ **Table 12** RQ4 severity distribution.

Severity	Count	Rate
None	40	66.7%
Minor	3	5.0%
Major	16	26.7%
Critical	1	1.7%

Table 11 presents the prevalence of each failure type. As shown in Table 11, Hallucinated Evidence is the most prevalent category (23.3%), followed by Missed Edge Cases (16.7%) and Behavioral Misunderstanding (11.7%). Notably, Test Overtrust and Contradictory Explanation were not observed in the sample. Within the FP quadrant, Missed Edge Cases is nearly universal (10/14 failed explanations), suggesting a tendency of the model to defend overfitting patches by overlooking inputs on which the evaluated patch and the correct fix diverge. In the FN quadrant, by contrast, Hallucinated Evidence dominates (5/6 failed explanations), indicating that the model fabricates code-level criticisms to justify rejecting actually-correct patches. Finally, Table 12 reports the severity distribution. Among the 20 failed explanations, 16 are rated as *major* and 1 as *critical*, indicating that the vast majority of explanation failures are not minor annotation disagreements but substantive reasoning errors that could mislead practitioners.

Answer to RQ4

LLM explanations are reliable mainly when the underlying label is already correct. In our exploratory GPT-4o/S0 sample, 33.3% of explanations contain at least one consensus failure, but the failures are concentrated in misclassified cases: 93.3% of FP explanations and 40.0% of FN explanations contain failures, whereas no consensus failures appear in the sampled TP/TN explanations. Hallucinated Evidence is the most common failure type (23.3%), followed by Missed Edge Cases (16.7%), and most failures have major severity. The low inter-annotator agreement ($\kappa = 0.237$) further suggests that explanation reliability is difficult to judge, making unverified LLM rationales a risky basis for human-in-the-loop patch review.

4 Discussion

4.1 LLM Judge as a Triage Aid, Not a Replacement

The RQ1 results demonstrate that both GPT-4o and DeepSeek-V3 achieve moderate agreement with human ground-truth labels under sanitized conditions (77.3% and 80.4% accuracy, MCC = 0.575 and 0.625, respectively), and that neither model significantly outperforms the other ($p = 0.164$). Although this level of performance is useful for triaging large volumes of candidate patches – reducing manual inspection effort by filtering out likely-overfitting patches – it is insufficient for unsupervised deployment as a sole correctness oracle. False positive rates of 17.2% (GPT-4o) and 22.1% (DeepSeek-V3) indicate that between one in five and one in six overfitting patches would be accepted as correct, potentially propagating inflated repair accuracy claims throughout the research literature. The cross-model comparison reveals broadly similar reliability across the full prompt matrix: after Holm correction, the only setting exhibiting a statistically significant patch-level accuracy difference is E1 ($p_{\text{Holm}} = 0.046$). Accordingly, the primary concern is not which model achieves marginally higher accuracy, but the fact that both models continue to accept a substantial proportion of overfitting patches. We therefore recommend that LLM Judge be positioned as a triage aid: a first-pass filter that ranks candidates for human review, rather than a replacement for ground-truth verification.

4.2 The Metadata Leakage Threat

RQ2 reveals that combining benchmark-identifying information (bug ID, project name, and tool name in S4) produces a suggestive accuracy improvement for GPT-4o (+4.3 pp, OR = 3.33) but not for DeepSeek-V3 (+1.8 pp, OR = 2.20). This model-dependent asymmetry is notable: although both models receive identical metadata, only GPT-4o exhibits a cluster-robust metadata-induced shift, and even that shift remains marginal after the expanded Holm correction ($p_{\text{Holm}} = 0.056$). The individual metadata settings for DeepSeek-V3 are likewise non-significant, suggesting that benchmark-identifying shortcuts exert a weaker influence on this model within our dataset. One plausible explanation is that the training data of GPT-4o contains more Defects4J-related content (e.g., from papers, repositories, and tutorials) than the corpus of DeepSeek-V3, thereby enabling stronger shortcut recognition; however, this result should be interpreted as a possible memorisation signal rather than direct causal evidence. The two counterfactual controls reduce, but do not eliminate, this ambiguity: shuffled metadata removes the GPT-4o improvement, while a fabricated bug ID does not reproduce the full S4 shift. Regardless of the underlying mechanism, the implication for experimental design is clear: studies that include file paths containing project names, bug identifiers in variable names, or tool attribution in comments may inadvertently enable

certain LLMs to short-circuit code reasoning – even when the effect is model-dependent. We therefore recommend that all benchmark-identifying metadata be stripped from prompts when evaluating LLM Judge reliability, in accordance with the S0 sanitization protocol, and that future leakage claims include counterfactual metadata controls.

4.3 Test-Passing Evidence: Limited Bias

Contrary to our initial hypothesis, the explicit declaration that all tests passed (E2) does not significantly bias LLM judgments for either model (GPT-4o: $p = 0.909$; DeepSeek-V3: $p = 1.000$). This cross-model consistency suggests that both LLMs, at temperature 0, do not naïvely equate test passage with patch correctness – a more sophisticated reasoning pattern than anticipated. However, providing raw test code (E1) or execution traces (E3) trends toward an increased false positive rate for both models. The most pronounced model-dependent difference emerges under the raw test code condition: DeepSeek-V3 significantly outperforms GPT-4o in patch-level correctness under E1 after cross-model Holm correction ($p_{\text{Holm}} = 0.046$), although its false positive rate remains elevated. This finding warrants caution: supplementary test evidence may introduce model-specific noise rather than uniformly improving reasoning quality.

4.4 The Developer Patch: Oracle or Leakage?

Both S5 (RQ2) and E4 (RQ3) demonstrate that providing the developer reference patch substantially improves LLM accuracy for both models: GPT-4o gains +10.4 pp and +9.5 pp, and DeepSeek-V3 gains +7.1 pp across both settings ($p \leq 0.003$). This improvement is expected and legitimate when the developer patch is genuinely available, as similarity comparison constitutes a valid reasoning strategy. However, the magnitude of the improvement raises a critical concern: when a study reports LLM Judge accuracy with developer patches included in the prompt, the reported figure reflects pattern-matching against a known answer rather than the independent code-reasoning ability of the LLM. Notably, both models converge to nearly identical accuracy under S5/E4 ($\sim 87\%$), despite differing by 3 pp under S0 – indicating that the reference patch effectively eliminates model-specific differences. Studies should therefore clearly distinguish between reference-assisted assessment, which is legitimate but requires a developer patch, and standalone assessment, which represents the common use case in which no reference is available.

4.5 Explanation Reliability: An Exploratory Risk Signal

RQ4 yields an exploratory risk signal: explanation quality is strongly contingent on label correctness. When the LLM classifies a patch correctly (TP and TN), the corresponding explanations contain no consensus-identified failures across the 30 sampled TP/TN cases. By contrast, when the classification is incorrect, explanation failures are substantially more frequent – a failure rate of 93.3% for false positives and 40.0% for false negatives.

This asymmetry creates a particularly dangerous feedback loop. A practitioner reviewing a false positive explanation encounters a plausible, well-structured argument that an overfitting patch is correct. Such an explanation may reference specific code elements, discuss behavioral semantics, and present a seemingly coherent conclusion; yet within the false positive sample, 93.3% of such explanations contain hallucinated evidence, missed edge cases, or behavioral misunderstandings. The practitioner possesses no reliable signal to distinguish a trustworthy explanation from a hallucinated one without independently verifying the underlying code

reasoning – an effort that defeats the efficiency rationale for employing an LLM judge. The low inter-annotator agreement ($\kappa = 0.237$) on explanation failures further reinforces this concern: even trained annotators struggle to reliably identify whether an LLM explanation is well-grounded or subtly fabricated. If domain experts disagree on whether an explanation contains hallucinated reasoning, practitioners with less domain expertise are unlikely to detect such failures during review.

4.6 Practical Recommendations

Based on the findings of this study, we offer the following recommendations for APR researchers employing LLM Judge:

4.6.1 APCA-specific recommendations

Sanitize prompts. Removing or masking benchmark-identifying metadata (bug IDs, project names, tool names, and file paths) should be viewed as a prudent precaution to reduce the risk of metadata-induced shortcuts, rather than as a universally validated requirement. When investigating leakage, include shuffled-metadata or fabricated-ID controls so that benchmark-formatting effects are not conflated with genuine metadata linkage.

Distinguish reference-assisted assessment from standalone assessment. When a developer patch is used as evidence, this should be clearly stated, and the results should not be generalized to settings in which no reference patch is available.

4.6.2 General principles for LLM-as-Judge evaluation

Report task-specific safety metrics alongside accuracy. Evaluation should expose the decision error with the greatest task-level cost and report the full error profile. In APCA, this safety-critical metric is the false-positive rate, together with the full confusion matrix.

Do not treat explanations as self-validating. LLM explanations are least reliable precisely when they matter most – on misclassified patches. Explanations should be treated as starting points for further investigation rather than as self-contained justifications.

Deploy LLM Judge as a triage filter. LLM Judge should be used to rank and filter candidates for human review, not as a replacement for manual correctness assessment.

5 Threats to Validity

Internal validity. The main internal threat is prompt sensitivity: different prompt wordings or output formats may induce different LLM judgments. We reduce this threat by using one prompt template across all settings, varying only the information content inserted into that template, and set the temperature to 0 to reduce sampling variability. For RQ4, the low inter-annotator agreement ($\kappa = 0.237$) indicates that explanation-failure labels partly depend on the adjudication protocol. We mitigate this risk through independent annotation, majority-vote consensus, and a conservative adjudication rule in which label changes are accepted only when at least two of three annotators agree on the change direction.

External validity. Our dataset is drawn exclusively from Defects4J, a Java benchmark covering five Apache projects. Results may not generalise to other programming languages or APR benchmarks. Defects4J is a widely adopted benchmark in the APR community, encompassing multiple real-world Java projects, and thus offers a degree of representativeness. The present study focuses primarily on Java, the dominant language in this domain; future work will explore broader language support, including Rust and Python. Furthermore, GPT-4o represents a widely used proprietary model and DeepSeek-V3 represents an open-source model with competitive performance. We deliberately selected this pair to contrast proprietary and open-source LLMs, rather than to claim that two models cover the full LLM landscape. Models with different architectures, training data, reasoning capabilities, or release dates may exhibit different reliability profiles. Accordingly, the observed model-dependent effects should not be generalised to all LLMs. Future work will repeat the protocol on additional proprietary and open-source models as releases evolve.

6 Related Work

6.1 Automated Patch Correctness Assessment

The APCA technique is designed to address the overfitting problem in test-based APR, wherein a generated patch may pass all available tests yet still fail to achieve the intended repair. Existing APCA techniques can be broadly organized into dynamic and static methods, depending on whether correctness evidence is obtained from program execution or code and change representations [37, 20].

Dynamic APCA methods assess patches by observing their behavioral effects under existing, generated, or augmented tests [31, 29, 30]. For instance, Opad [31] employs mutation testing to augment test cases for patch validation, DiffTGen [29] generates tests to expose test-suite-overfitted patches, PATCH-SIM [30] compares behavioral similarity between generated and developer patches, Ye et al. [35] scale such checks to large APR corpora, and xTestCluster [17] groups generated patches by their test-observed behavior. Dynamic methods provide executable evidence and can reveal concrete counterexamples, but their effectiveness remains constrained by test generation quality, oracle availability, and the observability of the fault-relevant behavior.

Static APCA methods infer patch correctness from code, change, and contextual features without requiring additional executions [34, 24, 16, 25, 11, 14]. For example, ODS [34] uses a classifier over statically extracted code features, Tian et al. [24] learn representations for code changes, Cache [16] develops context-aware code-change embeddings, Quatrain [25] explores correlations between bug descriptions and patch descriptions, Kang et al. [11] leverage language models for patch prioritization, Shibboleth [6] analyses production/test code impact. Reliability-oriented studies further show that APCA labels and checking procedures can be unstable, that current APCA methods still leave substantial error rates, and that existing checking approaches vary widely in assumptions and coverage [13, 26, 32]. Overall, static methods reduce dependence on generated tests and scale well to large patch corpora, but they also introduce risks from dataset bias, feature leakage, and limited cross-project generalization.

More recently, LLMs have been introduced into APCA, recasting patch assessment as a natural-language and code-understanding task [36, 40, 5, 33]. For example, APPT [36] fine-tunes pre-trained models for patch correctness prediction, LLM4PatchCorrect [40] retrieves similar patches and uses them as context to assist LLM predictions of patch correctness, Yang et al. [33] apply parameter-efficient fine-tuning with attributed patch semantic graphs to inject structured patch semantics into LLMs, RePaCA [5] employs reinforcement learning fine-

tuned LLMs to generate chain-of-thought reasoning for patch correctness assessment. These methods often report stronger predictive performance than earlier feature-based classifiers, but most studies optimize accuracy rather than validating whether LLM judgments are stable, unbiased, and grounded measurement outcomes. Our work therefore differs from prior APCA research by auditing LLM-as-Judge itself: we examine how metadata leakage, evidence availability, model choice, and explanation quality affect the reliability of LLM-based patch correctness assessment.

6.2 LLMs as Judges in Software Engineering

The use of LLMs as evaluation instruments has expanded rapidly beyond APR into code generation, code review, and software testing. CodeJudgeBench [8] evaluated LLM-as-Judge for code generation and testing tasks, finding systematic biases in LLM evaluations. Liang et al. [15] reported memorisation-related benchmark-recognition signals in SWE-bench, where LLMs may recognise benchmark instances from pre-training data rather than reasoning from first principles. Bias-in-the-Loop [38] audited LLM-as-Judge for software engineering tasks and found evidence of position bias, verbosity bias, and self-enhancement bias. Our work contributes to this growing body of evidence by providing a controlled, multi-dimensional reliability audit specific to the APCA task, with particular attention to benchmark-identifying information leakage and explanation quality.

7 Conclusion

This paper presents an empirical reliability study of LLM-as-Judge for automated patch correctness assessment. Across a balanced Defects4J dataset, GPT-4o and DeepSeek-V3 achieve moderate accuracy but still produce substantial false-positive rates, indicating that LLM Judge is useful for triage but insufficient as a standalone correctness oracle. Benchmark-identifying metadata can induce model-dependent accuracy shifts, which reinforces the need for prompt sanitization and counterfactual metadata controls. Test-passing declarations do not significantly bias judgments in our setting, whereas raw test code and execution traces may increase false positives. Providing the developer reference patch substantially improves accuracy, but this result should be interpreted as reference-assisted assessment rather than standalone evaluation. Finally, our exploratory explanation audit shows that one-third of sampled explanations contain failures, concentrated among misclassified patches, with hallucinated evidence and missed edge cases as the dominant failure types.

Future work should extend this reliability audit to additional LLMs, programming languages, and APR benchmarks, and investigate whether prompt engineering or fine-tuning can reduce explanation failures without introducing new biases.

Data Availability Statement

To facilitate future studies, our code and data are available at: https://github.com/Polaris-sg/llm_judge_Study.

References

- 1 Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi:10.1177/001316446002000104.
- 2 DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, et al. DeepSeek-V3 technical report. arXiv preprint arXiv:2412.19437, 2024. doi:10.48550/arXiv.2412.19437.

- 3 Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–50, 2025. doi:10.1145/3702972.
- 4 Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971. doi:10.1037/h0031619.
- 5 Marcos Fuster-Pena, David de Fitero-Dominguez, Antonio Garcia-Cabot, and Eva Garcia-Lopez. RePaCA: Leveraging reasoning large language models for static automated patch correctness assessment. *Neurocomputing*, 701:134583, 2026. doi:10.1016/j.neucom.2026.134583.
- 6 Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 654–665, 2022. doi:10.1145/3533767.3534368.
- 7 Raisa Islam and Owana Marzia Moushi. GPT-4o: The cutting-edge advancement in multimodal LLM. In Kohei Arai, editor, *Intelligent Computing: Proceedings of the 2025 Computing Conference, Volume 4*, volume 1426 of *Lecture Notes in Networks and Systems*, pages 47–60. Springer Nature Switzerland, Cham, 2025. doi:10.1007/978-3-031-92611-2_4.
- 8 Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. CodeJudgeBench: Benchmarking LLM-as-a-judge for coding tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 19416–19448, 2026. doi:10.18653/v1/2026.acl-long.888.
- 9 Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pages 1161–1173, 2021. doi:10.1109/ICSE43902.2021.00107.
- 10 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014. doi:10.1145/2610384.2628055.
- 11 Sungmin Kang and Shin Yoo. Language models can prioritize patches for practical program patching. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 8–15, 2022. doi:10.1145/3524459.3527343.
- 12 K.rippendorff. Reliability in content analysis: Some common misconceptions and recommendations. *Human Communication Research*, 30(3):411–433, 2004. doi:10.1093/hcr/30.3.411.
- 13 Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 524–535, 2019. doi:10.1109/ICSE.2019.00064.
- 14 Thanh Le-Cong, Duc-Minh Luong, Xuan-Bach Dinh Le, David Lo, Nhat-Hoa Tran, Bui Quang Huy, and Quyet-Thang Huynh. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Transactions on Software Engineering*, 49(6):3411–3429, 2023. doi:10.1109/TSE.2023.3255177.
- 15 Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pages 395–405, 2026. doi:10.1145/3786583.3786882.
- 16 Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology*, 31(3):1–29, 2022. doi:10.1145/3505247.
- 17 Matias Martinez, Maria Kechagia, Anjana Perera, Justyna Petke, Federica Sarro, and Aldeida Aleti. Test-based patch clustering for automatically-generated patches assessment. *Empirical Software Engineering*, 29(5):116, 2024. doi:10.1007/s10664-024-10503-2.

- 18 B. W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) – Protein Structure*, 405(2):442–451, 1975. doi:10.1016/0005-2795(75)90109-9.
- 19 Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947. doi:10.1007/BF02295996.
- 20 Facundo Molina, Juan Manuel Copia, and Alessandra Gorla. Improving patch correctness analysis via random testing and large language models. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 317–328, 2024. doi:10.1109/ICST60714.2024.00036.
- 21 Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025. doi:10.1145/3697010.
- 22 David M. W. Powers. Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation. arXiv preprint arXiv:2010.16061, 2020. arXiv:2010.16061.
- 23 Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Advances in Neural Information Processing Systems 37*, pages 51963–51993, 2024. doi:10.52202/079017-1647.
- 24 Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 981–992, 2020. doi:10.1145/3324884.3416532.
- 25 Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022. doi:10.1145/3551349.3556914.
- 26 Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 968–980, 2020. doi:10.1145/3324884.3416590.
- 27 You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. arXiv preprint arXiv:2503.15223, 2025. doi:10.48550/arXiv.2503.15223.
- 28 Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022. doi:10.1145/3540250.3549101.
- 29 Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236, 2017. doi:10.1145/3092703.3092718.
- 30 Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 789–799, 2018. doi:10.1145/3180155.3180182.
- 31 Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841, 2017. doi:10.1145/3106237.3106274.
- 32 Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1203–1215, 2023. doi:10.1145/3611643.3616331.
- 33 Zhenyu Yang, Jingwen Wu, Zhen Yang, and Zhongxing Yu. Parameter-efficient fine-tuning with attributed patch semantic graph for automated patch correctness assessment. *IEEE Transactions on Software Engineering*, 52(7):1960–1979, 2026. doi:10.1109/TSE.2026.3684016.

- 34 He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938, 2022. doi:10.1109/TSE.2021.3071750.
- 35 He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):20, 2021. doi:10.1007/s10664-020-09920-w.
- 36 Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen. APPT: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering*, 50(3):474–494, 2024. doi:10.1109/TSE.2024.3354969.
- 37 Quanjun Zhang, Ye Shang, Haichuan Hu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. ComPass: Contrastive learning for automated patch correctness assessment in program repair. arXiv preprint arXiv:2602.07561, 2026. doi:10.48550/arXiv.2602.07561.
- 38 Zixiao Zhao, Amirreza Esmaeili, and Fatemeh Fard. Bias in the loop: Auditing LLM-as-a-judge for software engineering. arXiv preprint arXiv:2604.16790, 2026. doi:10.48550/arXiv.2604.16790.
- 39 Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639222.
- 40 Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883, 2024. doi:10.1109/TSE.2024.3452252.