

Towards a Human-in-the-Loop Framework for Reliable Patch Evaluation Using an LLM-as-a-Judge

Sherry Shi, Renyao Wei, Michele Tufano, José Cambronero, Runxiang Cheng,

Franjo Ivančić, Pat Rondon

Google, USA

{sherryshi,renyaow,tufanomichele,jcambronero,chengsam,ivancic,rondon}@google.com

Abstract

Reliable evaluation is crucial for advancing Automated Program Repair (APR), but prevailing benchmarks rely on execution-based evaluation methods (unit test pass@k), which fail to capture true patch validity. Determining validity can require costly manual annotation. To reduce this cost, we introduce a human-in-the-loop approach to LLM-based patch validity judgment. Inspired by the observation that human judgment is better aligned when using a shared rubric, we first employ an LLM to generate a per-bug rubric, followed by a one-time human review and optional refinement to this rubric, and then employ an LLM to judge patches using the refined rubric. We apply this approach to assign binary validity labels to patches for issues found by Google sanitizer tools. Our results show that this approach yields substantial agreement with human consensus (Cohen’s kappa 0.75), high recall (0.94) and high precision (0.80), when considering patches that have unanimous agreement from 3 human raters on the validity labels. On the full dataset including patches where human raters disagree, we find this approach can still be further improved (Cohen’s kappa 0.57, recall 0.93, precision 0.65) and identify possible future directions.

1 Introduction

Automated Program Repair (APR) stands as a promising frontier in software engineering, with the potential to significantly enhance developer productivity by automatically fixing bugs. As LLM-based APR systems become more capable, their deployment in industrial settings has begun in earnest [19, 26, 29]. The success of these systems hinges on a reliable evaluation procedure to ensure that offline results accurately reflect online performance, so that development and deployment decisions can be made with high confidence.

For years, APR research has employed execution-based evaluation as the de facto standard, with popular benchmarks (such as MBPP [4], HumanEval [6], and SWE-Bench [13]) measuring functional correctness through test pass rates like pass@k. While instrumental, this evaluation paradigm has a critical limitation: a patch can contain additional behaviors that are not assessed by the tests, potentially invalidating it. Indeed, Wang et al. [32], Yu et al. [38], Zhu et al. [43] uncovered a significant number of cases where patches that passed tests in benchmarks like SWE-Bench were nonetheless functionally incorrect. These invalid patches highlight a gap between test performance and true validity. While these patches can be caught by human review, manual patch assessment is time-consuming and expensive to scale. More importantly, we find that manual patch assessment suffers from its own critical flaw: inconsistency. In our study of the human assessment process, we found low inter-rater agreement (Fleiss’s kappa 0.31) among developers assessing patch validity independently.

However, such inconsistency can be largely resolved by allowing developers to perform their assessments with a shared, high-quality rubric as reference. As we will show in §2, inter-rater agreement substantially increases if all raters assess generated patches of a bug following the same set of fix requirements for that bug. More importantly, such rubric-based evaluation paradigm allows us to consider the potential of employing LLM-as-a-Judge for offline APR patch assessment.

LLM-as-a-Judge has emerged as an increasingly attractive approach to overcome the scalability limitation of manual evaluation, where an LLM provides an assessment of the code generated by another model. Although misjudgements by the LLM call its practicality into question, Crupi et al. [9], Tong and Zhang [31], Zheng et al. [40] have shown promise in this direction. In particular, Zhuge et al. [44] achieved 90% alignment rate between humans and their agentic LLM judge on AI development tasks. While their work demonstrates a high alignment rate with humans using a set of manually-crafted set of requirements, the practical challenge of crafting these requirements remains. The significant manual effort required to create high-quality evaluation criteria for diverse and evolving sets of bugs remains a critical bottleneck for scalable evaluation in APR, highlighting an opportunity for LLMs to assist.

To overcome the reliability and scalability limitations of human evaluation, especially for bugs with no pre-existing fix requirements, we propose a human-in-the-loop framework to evaluate APR patch validity using LLM-as-a-Judge. Building on our insight that a shared, high-quality patch validation rubric is the cornerstone of reliable evaluation, our two-stage framework operationalizes rubric-guided evaluation at scale. First, an LLM generates a draft rubric for a given bug, which two human experts then review and refine into a “golden” evaluation standard. Second, an LLM judge uses this golden rubric to assess the validity of candidate patches. Our evaluation, conducted on 115 patches for 48 bugs reported by Google’s sanitizer tools, demonstrates that this approach yields substantial agreement with human consensus (Cohen’s kappa 0.75), high recall (0.94) and high precision (0.80), when considering patches that have unanimous human agreement from 3 raters on the validity labels (70.4% of the dataset). On the full dataset including patches where human raters disagree, we find this approach can still be further improved (Cohen’s kappa 0.57, recall 0.93, precision 0.65) and we identify possible directions.

This paper makes the following contributions:

- An empirical study of human evaluation for code patches that demonstrates low inter-rater agreement when raters act independently, but improves when raters refer to a common rubric.

- A novel human-in-the-loop framework that leverages an LLM to generate task-specific rubrics which are then refined by a human developer.
- A large-scale evaluation demonstrating that LLM-as-a-Judge, when guided by human-refined rubrics, achieves moderate to substantial levels of agreement with humans.

2 Motivation

The ultimate goal of APR is to generate correct patches. However, the field has long grappled with a significant “evaluation gap” between Fail-to-Pass (F2P) patches, those that make a failing test suite pass (measured by `pass@k`), and correct patches, those that actually address the underlying bug and would pass human code review (measured by `valid@k`). This gap is not a new phenomenon; it has persisted across generations of APR techniques [24]. For instance, on Defects4J v1.2 [15], Recoder [42] found that only 54.3% of its plausible patches were correct, ContrastRepair [16] showed 62.5% correctness (75 out of 120 F2P), and ChatRepair [35] showed 51.5% correctness (52 out of 101 F2P).

This discrepancy highlights that `pass@k` is an insufficient signal and it is limited by the quality of the underlying tests. Test suites may have insufficient coverage, weak assertions, or missing edge-case handling, making them susceptible to “hacks” where a patch overfits to the tests without addressing the bug’s root cause. While one can potentially improve the execution-based signal by enhancing test quality, it is not a complete solution; in large-scale, complex software systems, it is often intractable to create a test suite that perfectly captures all functional and non-functional requirements.

Consequently, to bridge this evaluation gap and determine the final verdict on patch validity, the field has already turned to manual assessment from expert developers as the de facto gold standard. However, the reliability of this manual process is not universally established and may vary by context. For instance, Oliva et al. [21] found low inter-rater agreement when applying discrete labels for issue clarity and test coverage on complex, SWE-Bench-like software tasks.

As the first step towards building a more automated system to evaluate patch validity, we conduct a preliminary study to answer the following question: *To what extent do human raters agree on the validity of APR patches, and how does the evaluation rubric affect their agreement?*

Our study involves three of the authors acting as raters. We used a preliminary dataset of 52 F2P patches generated for five distinct bugs reported by Google’s Sanitizer tools. For each bug, the three raters referred to the bug report and ground-truth patch to independently author their own evaluation rubric, which outlines the requirements for a valid fix. Each rater then produced a binary assessment (`VALID` or `INVALID`) for all 52 patches under two settings: (1) using self-authored rubric, and (2) using a rubric authored by another rater.

First, to quantify the reliability of manual patch validity assessment, we calculated Fleiss’s kappa on the raters’ assessments based on their self-authored rubrics. Our analysis revealed a low level of agreement among the three raters (e.g., 0.31 as shown in Table 1). The main reasons for disagreements were different interpretations

Table 1: On 52 F2P patches of 5 bugs, three raters show substantial disagreement when using self-authored rubrics (Cohen’s kappa ranges from 0.19 to 0.39).

Raters	Rubric	Fleiss’s κ or Cohen’s κ
Rater 1, 2, 3	Self-authored rubric	0.31
Rater 1 and 2	Self-authored rubric	0.37
Rater 2 and 3	Self-authored rubric	0.19
Rater 3 and 1	Self-authored rubric	0.39

Table 2: Rater’s self-agreement is moderate (0.35 to 0.49) when using two different rubrics.

Rater	Rubric	Cohen’s kappa
Rater 1	Rater 1 rubric & Rater 3 rubric	0.35
Rater 2	Rater 2 rubric & Rater 1 rubric	0.36
Rater 3	Rater 3 rubric & Rater 2 rubric	0.49

Table 3: When new raters use the same rubric, pairwise agreement improves to moderate/high agreement (Cohen’s kappa).

Rater	Rubric	Cohen’s kappa
Rater 1 and 2	Rater 1’s rubric	0.84
Rater 2 and 3	Rater 2’s rubric	0.67
Rater 3 and 1	Rater 3’s rubric	0.53

of the root cause, disagreements on the acceptability of unnecessary changes, disagreements on non-functional requirements, and misunderstandings of the code. This suggests that human raters’ patch validity assessment in the current manual evaluation process can be unreliable, in part because human raters can easily develop and follow their own subjective criteria.

Further analysis reveals that the rubric is a primary contributor to discrepancies among human raters. Rater self-agreement, which compares assessments from the same rater using two distinct rubrics (their own and another’s), was found to be moderate. Its Cohen’s kappa values are between 0.35 and 0.49 (Table 2). Conversely, when pairs of raters utilized the same rubric for evaluation, their agreement significantly improved, yielding Cohen’s kappa values ranging from 0.53 to 0.84 (Table 3). This preliminary study supports the hypothesis that basing patch validity assessment on a common rubric facilitates reliable and reproducible assessments of APR patch validity.

This finding raises a natural next question: *What constitutes a good rubric?* Intuitively, a rubric should be precise and informative if it is to be shared among human raters to assess patch validity. To investigate what constitutes such a “golden rubric,” the three initial human raters collectively analyzed their disagreements and distilled a set of best practices into a standardized template. Listing 1 shows an example rubric following this template. This template specifies key sections including the root cause of the bug, a checklist of requirements for a valid fix, and concrete examples of acceptable and unacceptable solutions.

```
## Root Cause
The bug is a `use_of_uninitialized_value` error. A new
→ member, `redacted_property_name`, was added to
→ `RedactedClass` in `some/redacted/path.h` but was not
→ initialized in any constructor. Subsequent code reads
→ this uninitialized member, triggering the error.
```

Requirements

- A correct patch must satisfy the following requirements:
1. **Initialize the Member:** `redacted_property_name` must be initialized.
 2. **Correct Default Value:** Must be initialized to `redactedValue` (which indicates an 'unset' state).
 3. **Robust Implementation:** Use a C++11 in-class member initializer directly in the struct definition.
 4. **Correct Location:** The change must be in `some/redacted/path.h`, not a workaround in the test.
-

Listing 1: Illustrative rubric example.

Following this template, the three raters developed golden rubrics and provided them to three new human raters who had no prior involvement in the study. These new raters were asked to assess the validity of the same 52 F2P patches. The new raters achieved substantial inter-rater agreement (Fleiss's kappa 0.66 as shown in Table 3), demonstrating that this template can help generate shared rubrics that effectively improve assessment consistency.

Overall, this preliminary study shows that using a templated rubric can effectively standardize the task and provide a reliable and consistent assessment of patch validity. These results motivate us to develop a framework to efficiently generate high-quality rubrics and apply rubric-guided assessment of patch validity at scale.

3 Framework Overview

Based on the insights from our preliminary study, we develop a framework with the goal to provide a scalable and reliable offline evaluation for patch validity. Figure 1 provides an overview of the two main stages of our framework:

- (1) **Rubric Generation:** A one-time, human-in-the-loop process with LLM-based rubric generator to produce high-quality “golden rubrics” for patch evaluation.
- (2) **Patch Evaluation:** A repeatable, automated process where an LLM judge applies the golden rubrics to assess patch validity.

3.1 Rubric Generation

The rubric generation stage consists of two steps:

LLM-based draft rubric generation. We use an LLM to generate an example-specific rubric for each bug. Given the bug context (e.g., bug description) and its ground-truth fix as input, the LLM is instructed to generate the rubric draft that contains all the required sections listed in our predefined template (Listing 1), such as the root cause of the bug and a list of requirements for a valid fix.

Manual rubric refinement. Two human developers review and refine a draft rubric into a golden rubric. One developer reviews

and edits the draft rubric if needed, e.g., correcting inaccurate requirements, removing requirements overfitting to the ground-truth fix, and ensuring all elements required to fix the bug are captured. The developer also documents justifications for the edits. Another developer reviews and confirms all edits and the justifications. The final rubric will serve as the golden rubric.

3.2 Patch Evaluation

Given the bug context (e.g., bug description), the patch in the unified diff format [10], and the golden rubric as input, the LLM judge is instructed to output its validity assessment. The assessment consists of a “thought” section that captures its thinking steps, a binary label (“VALID” or “INVALID”), and a justification that captures its reasoning more concisely. We store the LLM judge’s output label and justification for further analyses, such as qualitatively understanding why certain patches are accepted or rejected.

3.3 Framework Formalization and Notations

We now provide a simple formalization of our framework, with notation that will be used in the subsequent sections of this paper. We refer to the bug context (e.g., bug description) as b , the ground-truth fix (patch) as f , and a F2P patch as p . The two main stages of our framework can then be formalized into:

- (1) An draft rubric r is generated by the rubric generator R :

$$r = R(b, f)$$

Human developers, H , review and refine r into the golden rubric r_{gold} :

$$r_{\text{gold}} = H(r)$$

- (2) The LLM judge J_{lm} outputs a tuple (v, t) on the patch p of bug b with golden rubric r_{gold} , where v is the validity label and t is justification for the label:

$$(v, t) = J_{\text{lm}}(b, p, r_{\text{gold}})$$

4 Empirical Study Design

We evaluate both the rubric generator and the LLM-as-a-Judge components of our framework to study its effectiveness. In this section, we describe the evaluation dataset, experimental setup and metrics used in our research questions. We use Gemini 2.5 Pro as the LLM throughout our evaluation [8].

4.1 Dataset

In this paper, we focus on sanitizer bugs, because (1) they can identify issues that can lead to security vulnerabilities, denial-of-service attacks, and other serious problems; and (2) they have structured bug reports that provide a consistent form of information (i.e., bug type, failing test, reproduction command) compared to human-reported bugs which may vary in detail. These reasons make sanitizer bugs a promising area for high-impact and effective automated APR patch validation. In our case, sanitizer bugs are automatically reported by a suite of sanitizer tools [28] that are run on a regular basis in Google’s monorepo.

We first collect 50 sanitizer bugs following prior bug curation process [26]. We generate 20 patches for each bug using our APR agent [26] and verify the F2P behavior of each patch by checking if

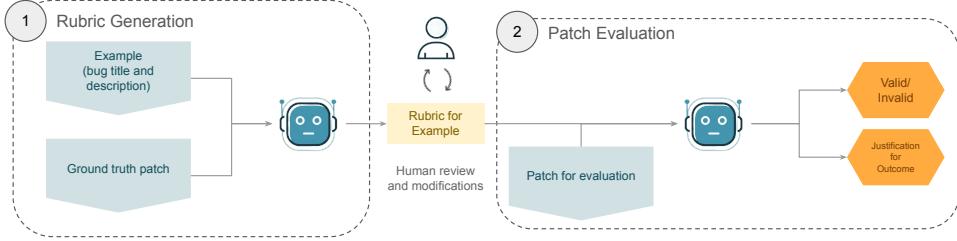


Figure 1: Overview of our two-stage framework. ① **Rubric Generation:** An LLM first generates a per-bug rubric based on the bug’s description and the ground truth patch. This rubric is reviewed and refined once by two human experts. ② **Patch Evaluation:** The refined rubric is repeatedly used by an LLM judge to evaluate patches of the same bug. The LLM judge outputs a binary validity label and a natural language justification to the label, enabling both quantitative and qualitative analyses.

Table 4: Bugs and sampled patches by bug types in our dataset.

Bug Type	# Bugs	# Patches
data_race	14	33
use_of_uninitialized_value	11	26
misaligned_pointer_use	6	13
data_race_go	3	8
fuzztest_crash	2	4
leak_detected	2	6
signed_integer_overflow	2	5
stack_use_after_scope	2	6
use_of_uninitialized_value	2	5
hwasan_tag_mismatch	1	2
invalid_bool_load	1	2
invalid_enum_load	1	3
null_pointer_use	1	2
Total	48	115

the failing test from the bug report now passes after applying the patch. We then randomly sample at most 3 unique F2P patches per bug; we ensure each sampled patch is unique by computing their hashes. Some bugs have fewer than 3 unique F2P patches generated, and 2 bugs have no F2P patches. Eventually, we obtained 115 unique F2P patches for 48 bugs. Of these 48 bugs, we have 44 C++ bugs, 3 Go bugs, and 1 Java bug. Table 4 summarizes the number of bugs and sampled patches by bug types.

4.2 Research Questions

We investigate the following research questions in our study:

- (1) **RQ1:** How effective is an LLM at generating reusable, high-quality rubrics for evaluating APR patches?
- (2) **RQ2:** How well does an LLM-as-a-Judge align with human consensus on APR patch validity?

4.2.1 RQ1: Effectiveness of LLM in Rubric Generation. We first use LLM-based rubric generator to produce draft rubrics for all 50 bugs. A team of 6 authors of this work then perform rubric refinement (§3.1) on each draft rubric to produce a golden rubric. To investigate this RQ, we study the changes between the 50 draft rubrics and their

corresponding 50 golden rubrics with both quantitative metrics and qualitative analyses:

- **Modification Rate:** The percentage of draft rubrics that were modified by human reviewers.
- **Normalized Edit Distance:** We compute the character-level Levenshtein distance between the draft rubric (r) and the golden rubric (r_{gold}), then normalize it by the draft rubric length:

$$\text{NormalizedEditDistance} = \frac{\text{Levenshtein}(r, r_{gold})}{\text{length}(r)}$$

This metric represents the proportion of the draft rubrics that was modified by the developer, and aims to quantify the magnitude of the relative change between the draft and the golden rubric.

- **Edit Type Analysis:** We analyze the nature of the changes by classifying edits as additions, deletions, or modifications.
- **Thematic Analysis:** We analyze the categories of edits by their semantic purposes (e.g., correcting inaccuracies, improving rubric generalization). To do so, we first provide the edits and justifications of these edits as input to LLM, and prompt the LLM to generate a initial set of categories. We manually review and assign each distinct edit to one or more categories, while also iteratively refining the initial set of categories (e.g., via merging, splitting, or relabeling) to create a final, robust classification scheme that accurately reflects the edits’ semantics.

4.2.2 RQ2: Performance of LLM-as-a-Judge. In this RQ, we aim to study the efficacy of LLM-as-a-Judge in assessing patch validity when it is guided by human-refined golden rubrics. Specifically, we compare the patch validity labels produced by an LLM judge against the labels produced by human raters’ consensus, and assess the agreements between an LLM judge and human consensus.

To establish a benchmark of patch validity from human consensus, three authors are assigned as raters to each of the 115 F2P patches. Working independently and without access to the LLM-as-a-Judge’s outputs, each rater uses the golden rubrics to label each patch as “VALID” or “INVALID”. If the three raters produce different labels for a patch, they will discuss to reach consensus and establish a final, single label for the patch.

Human agreement. We measure the inter-rater reliability on the labels produced before the consensus discussions to check the agreement among human raters.

We use Krippendorff’s alpha and a weighted average Cohen’s kappa (where the weight is based on the percentage of patches reviewed by each rater) to measure the inter-rater reliability among the human raters on their initial, independent ratings. This metric validates the consistency of the benchmark itself.

LLM-human agreement. We use the LLM-as-a-Judge approach to produce validity labels for these patches, and compare its labels against those of the human consensus benchmark.

We use accuracy and Cohen’s kappa to measure the agreement between the LLM judge’s labels and the human consensus. We also report precision, recall, and F1-Score of LLM judge for the VALID class by treating the human consensus as the ground truth.

5 Empirical Study Results

In this section, we present results from studying the patch validation rubrics generated by our LLM-based rubric generator (§5.1), and results from studying the agreement between LLM-as-a-Judge and human consensus on patch validity (§5.2).

5.1 RQ1: How Effective Is an LLM at Generating Patch Evaluation Rubrics?

To study the quality of LLM-generated rubrics, we analyze the edits made by human developers during rubric refinement.

5.1.1 Quantitative Analysis of Edits. The LLM-based rubric generator produce draft rubrics with substantial details, with a median and mean length of 2,004 and 2,058 characters, respectively.

However, we found that 44 out of the 50 (88%) draft rubrics needed manual revision before confirming them as golden rubrics. The magnitude of these revisions range from minor to considerable. Specifically, the median absolute edit distance is 276 characters (average 385 characters). The median normalized edit distance was 0.14, indicating that 50% of the draft rubrics required human revisions equivalent to 14% of its original content, as shown in Figure 2. The maximum observed normalized edit distance is 0.7. Importantly, no LLM-generated draft rubric required a complete rewrite.

Overall, our results imply that LLM consistently provides a valuable scaffold, positioning the human developer’s role for reviewing and refining rubrics rather than writing rubric from scratch.

5.1.2 Edit Type Analysis. We observe the diffs of the original and revised rubrics and classify the revisions into three types: deletion, addition, and modifications. We find that deletions were the most common (39 rubrics), followed by modifications (14 rubrics) and additions (3 rubrics). The prevalence of deletions over additions suggests that LLM tends to generate overly verbose rubric drafts that include superfluous information, rather than drafts that are lacking details and incomplete.

5.1.3 Edit Thematic Analysis. We perform thematic analysis to understand underlying developer motivations for these edits. Our analysis shows five primary categories of deficiencies of the LLM-generated drafts, as follows:

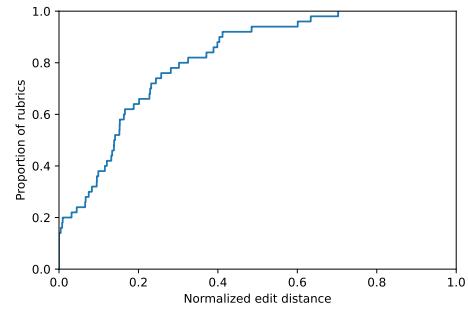


Figure 2: CDF of normalized edit distance on 50 rubrics. The median normalized edit distance is 14% of the original content, while the maximum is 70% of the original content.

- **Removing Superfluous Constraints (31 rubrics):** These edits eliminate overly general requirements not essential for a correct fix. Examples include “changes must be minimal,” or mandating changes beyond the scope of the fix, such as adding unnecessary dependencies, tests, or comments.
- **Reducing Overfitting to Ground Truth (10 rubrics):** These edits generalize the rubrics to accept more correct fix implementations, rather than accepting only implementations that mirror the ground truth patch. These edits are critical for identifying more valid (but diverse) patches.
- **Correcting LLM Errors & Hallucinations (9 rubrics):** These edits remove inaccurate diagnosis of the bug’s root cause, hallucinated explanations, or other factually incorrect information in the LLM-generated rubrics.
- **Refining Scope & Expectations (6 rubrics):** These edits refine fix scope, such as specifying which files should (or should not) be modified or ensuring that the core functional intent of a fix is preserved.
- **Standardization (5 rubrics):** These edits improve consistency across all rubrics. Examples include standardizing rubric formatting, terminology (e.g., use “primitive” over “fundamental”), and rubric structure. Standardizing rubrics could help LLM judge to have more predictable behavior.

While the LLM can generate promising draft rubrics as a strong scaffold, these deficiencies indicate that the human-in-the-loop process was essential to improve the LLM-generated draft rubrics before using them for patch evaluation.

5.2 RQ2: How Well Does an LLM Judge Align with Human Consensus on Patch Validity?

To compare LLM judge with human raters’ judgments, we establish a high-quality patch validity benchmark via human consensus, and then evaluate the performance of our LLM-as-a-Judge against it.

5.2.1 Patch Validity Benchmark via Human Consensus. Before any discussion to resolve disagreements, the raters achieved a perfect unanimous agreement on 81 out of 115 patches (70.4%), and a Krippendorff’s alpha of 0.60 on all 115 patches. After the raters meet

Table 5: Patch validity benchmarks.

Metric	B_{full}	B_{clear}
Number of bugs	48	41
Number of patches	115	81
Unanimous agreement	70.4% (81/115)	100% (81/81)
Krippendorff's alpha	0.60	1.0
Weighed average Cohen's kappa	0.76	1.0
Consensus: # VALID patches	44	35
Consensus: # INVALID patches	71	46

Table 6: LLM judge effectiveness under different ablations. Using unrevised rubrics r , free-form rubrics r_{ff} , or ground truth (GT) patches all underperform our full approach r_{gold} .

Metric	Benchmark	Ablations			
		r_{gold}	r	r_{ff}	GT patch
Cohen's kappa	B_{full}	0.57	0.38	0.29	0.39
	B_{clear}	0.75	0.52	0.33	0.44
Accuracy	B_{full}	0.78	0.70	0.62	0.69
	B_{clear}	0.87	0.76	0.65	0.72
Precision	B_{full}	0.65	0.60	0.51	0.56
	B_{clear}	0.80	0.74	0.57	0.62
Recall	B_{full}	0.93	0.66	0.82	0.84
	B_{clear}	0.94	0.71	0.83	0.89
F1-Score	B_{full}	0.77	0.63	0.63	0.67
	B_{clear}	0.87	0.72	0.67	0.73

to resolve disagreements and reach consensus, the weighted average Cohen's kappa between each human rater and the human consensus is 0.76, indicating moderate agreement.

Based on these results, we construct two benchmarks to evaluate LLM-as-a-Judge: a full benchmark B_{full} containing all 115 patches, and the clear benchmark B_{clear} containing the 81 patches that have unanimous agreement prior to rater discussions. The statistics of both benchmarks are further summarized in Table 5.

We also analyze the 34 patches where raters disagreed before reaching consensus and identify four disagreement themes:

- **Overlooked Requirements (15 patches):** raters failed to identify or apply specific criteria detailed in the rubric.
- **Unnecessary Changes (9 patches):** raters disagreed on whether to accept code changes that go beyond the core issue the patch is supposed to address.
- **Rubric Ambiguity (8 patches):** unclear, incomplete, conflicting or differently interpreted rubric guidelines.
- **Correctness Assessment (2 patches):** raters disagreed on whether the patched code is functionally correct or introduces issues.

5.2.2 LLM Judge Performance Against Human Consensus. We evaluate the LLM-as-a-Judge (J_{llm}) using golden rubrics (r_{gold}) against

both benchmarks. The bolded column in Table 6 shows the corresponding evaluation results.

The results indicate a moderate-to-substantial alignment on patch validity between the golden-rubric-guided LLM judge and the consensus of human raters. Specifically, the primary alignment metric, Cohen's kappa, is 0.57 on B_{full} and 0.75 on B_{clear} . And the accuracy on B_{full} and B_{clear} are 0.78 and 0.87, respectively.

The LLM judge achieves a high recall of 0.93 on B_{full} and 0.94 on B_{clear} , suggesting that it is highly effective at correctly identifying valid patches and minimizing false negatives. We attribute this strong performance directly to the human refinement process on the rubrics. Specifically, by generalizing the rubrics and removing overfitting to the ground truth patch (a prominent edit theme), the golden rubrics (r_{gold}) explicitly permit the judge to accept a wider range of correct, alternative solutions. The high recall of this framework also allows developers to use it to filter out INVALID patches, and reduce human effort in reviewing the remaining patches deemed VALID by the LLM judge.

Conversely, the precision of 0.65 on B_{full} indicates that the judge is more prone to false positives, sometimes incorrectly labeling an invalid patch as valid. However, a deeper analysis of these errors reveals a critical insight: the LLM judge's mistakes often overlap with cases that were also difficult for human experts. Of the 22 false positives generated by the LLM judge, 14 (63.6%) occurred on patches that also generated disagreement among human raters (the 34 patches that required a consensus discussion). This suggests that many of the LLM judge's mistakes occur on patches whose validity is controversial, which the human raters also find difficult to evaluate. The higher precision (0.80) on the unanimously agreed patches (B_{clear}) further supports this observation.

We also categorize the mistakes LLM judge made on the 25 patches where it disagreed with the human consensus:

- **Overlooked Requirements (10 patches):** The LLM judge failed to identify or apply specific criteria in the rubric.
- **Unnecessary Changes (9 patches):** The LLM judge disagreed on whether to accept code changes that go beyond the core issue the patch is supposed to address.
- **Rubric Ambiguity (6 patches):** unclear, incomplete, conflicting or differently interpreted rubric guidelines.

Notably, LLM judge achieves a high negative predictive value (NPV) of 0.94 and 0.95 on B_{full} and B_{clear} respectively, indicating that its INVALID labels are highly reliable. Developers can confidently refer to issues of these invalid patches identified by our framework for improving the evaluating APR system.

6 Ablation Study

To better understand the impact of designs choices made in the rubric generation stage of our framework, we conduct several ablation studies and measure the resulting deviation from the human consensus benchmark. We investigate the impact of (1) manual rubric refinement by comparing the LLM judge's performance with the golden rubric (r_{gold}) against the draft rubric (r); (2) the rubric template by comparing against a free-form rubric with no refinement (r_{ff}); and (3) the rubric itself by comparing against the common alternative of using the ground truth patch for patch evaluation. The performance of each configuration is summarized in Table 6.

6.1 Impact of Manual Rubric Refinement

This study quantifies the value of the human-in-the-loop refinement step by comparing the performance of the LLM judge when using the final golden rubric (r_{gold}) versus the initial, draft rubric (r).

Using the initial draft rubric (r) as a baseline, the LLM judge achieves only fair alignment (Cohen’s kappa of 0.38 on B_{full}) with human raters. This is largely due to a poor recall of 0.66. This finding directly supports our analysis in RQ1: the initial, unedited rubrics are overfitted to the ground truth patch, and consequently misguide the LLM judge to reject a large number of valid alternative patches (i.e., produces many false negatives).

When human experts refine this initial draft into the golden rubric (r_{gold}), the performance improves dramatically. Recall improves from 0.66 to 0.93 on B_{full} , as the human-led generalizations (e.g., “Reducing Overfitting to Ground Truth” from RQ1) explicitly allow the judge to accept a wider range of correct solutions. Precision is improved from 0.60 to 0.65 on B_{full} , as human edits correct the “LLM Errors & Hallucinations” and “Superfluous Constraints” that misled the judge into accepting invalid patches.

Increase in precision and recall also boost Cohen’s kappa from 0.38 to 0.57 on B_{full} . These results confirm that manual rubric refinement is indispensable for elevating this patch evaluation framework from a low-agreement baseline to a reliable instrument.

6.2 Impact of Rubric Template

Our framework’s rubric generator, R , was designed to produce draft rubrics based on a manually-crafted, standardized template. To study the marginal contribution of this template, we compare performance of LLM judge on the unedited templated rubric (r) versus on the a “template-free” free-form rubric (r_{ff}), where mentions of template elements (root cause, requirements, examples of acceptable and unacceptable solutions) are removed from the rubric generation prompt.

Results in Table 6 show that the template is beneficial. Removing it causes Cohen’s kappa to drop from 0.38 to 0.29 (i.e., fair agreement) on B_{full} . This finding underscores the critical role of the rubric template: the structure the template imposes is not merely a formatting convention, but a key factor that guides the LLM-based rubric generator to reason about the root cause and encode different evaluation criteria in the rubric, which enables the downstream LLM judge to perform more reliable patch evaluation.

6.3 Impact of Rubric versus Ground Truth Patch

A common approach to use LLM-as-a-Judge for patch evaluation is to provide the ground truth patch as a reference, the task description, and the generated patch in the LLM prompt [31, 40]. Table 6 (“GT Patch”) shows that this approach achieves Cohen’s kappa of 0.39 with human consensus on B_{full} , which is better than the free-form rubric, similar to draft rubric, and worse than our full approach.

7 Discussion

Figure 3 compares $pass@k$ and $(pass \& LLM\text{-}valid)@k$ on the 1,000 patches (50 Sanitizer bugs \times 20 generated patches) from §4.1. It shows a notable “evaluation gap” from Fail-to-Pass behavior to LLM-judged validity: at $k = 20$, 96% of the patches passed reproduction tests, while 80% of the patches were judged valid.

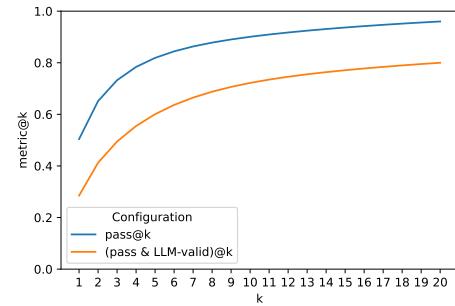


Figure 3: A 16-percentage-point drop from $pass@20$ to $(pass \& LLM\text{-}valid)@20$ due to LLM judge rejecting patches that violate requirements in rubric, fail to address root cause, are incomplete, or introduce new issues.

Out of 504 patches that passed bug reproduction tests, 219 (43.5%) were deemed invalid by the LLM judge. We perform a thematic analysis on the judge’s justifications to understand why they were rejected. We cluster the justifications with an LLM, and identify these themes below:

- **Violating Specified Requirements in Rubric (60.6%):** Patches fail to adhere to specific guidelines, best practices, or architectural patterns outlined in the rubric. These violations often relate to coding style, design principles, memory management (e.g., requiring RAII/smart pointers vs. manual new/delete), solution scope (e.g., test-only changes vs. production code changes), adherence to idioms, performance expectations, or requirements for robustness and consistency. The patch might functionally fix the bug but does so in a manner that is explicitly disallowed in the rubric.
- **Not Addressing Root Cause (26.6%):** The patch either masks the symptom, provides a superficial workaround, misidentifies the problem, or applies a fix in an incorrect or irrelevant location. For example, sanitizer errors are suppressed instead of fixing undefined behavior, or production code is modified to work around a test data issue.
- **Incomplete Implementation (7.3%):** Patches do not fully address the scope of the problem; it leaves other parts of the same problem unresolved, fails to apply the fix to all affected instances or tests, or overlooks necessary related changes (e.g., missing dependencies, header includes).
- **Introducing New Issues (5.5%):** Patches that inadvertently introduce new problems, regressions, or vulnerabilities. These new issues can manifest as compilation errors, linking errors, new logical bugs, performance regressions, or the creation of an unstable state.

We also notice some of these patches were rejected by LLM judge due to the judge’s misjudgement and inaccuracies from the rubric. Specifically, judge justifications for 5 patches mention compilation errors, but these patches passed the reproduction tests. This misjudgement is because the rubric requires updating header and adding dependencies for the updated header. The rejected patch did not use the header and thus did not need to add such dependencies.

8 Future Work

Building on our findings, we identify several promising directions for future research aimed at improving the efficiency and autonomy of our patch evaluation framework.

An immediate next step is to address the gap between the LLM judge and human consensus. We have identified the following opportunities for improving the system: (1) *Rule of Minimalism*: implement a clear, universal rule that patches containing unnecessary changes should be classified as INVALID, which should resolve inconsistencies; and (2) *Streamlined Rubric Review*: enable reviewers to assess rubrics against a set of diverse patches to confirm desired outcomes and reduce ambiguity.

We also plan to conduct a formal user study to quantify the efficiency gains from our framework. This study would measure the time and cognitive load required for an expert to refine an LLM-generated rubric compared to the effort of manually evaluating a large set of candidate patches from scratch.

Another significant avenue is to expand the judge output beyond binary correctness to non-functional aspects of code quality (e.g., maintainability, performance, security). The rubrics can also incorporate these criteria more considerably, leading to a multi-dimensional, holistic assessment of generated patches.

Our goal is to reduce reliance on human judgment and move toward a more autonomous APR evaluation ecosystem. We envision a self-improvement cycle where the LLM judge automatically flags low-confidence assessments, solicits targeted human feedback only on these ambiguous cases, and then uses that feedback to automatically refine the rubrics. For example, we can guide the rubric generator with patch rejection justifications to produce higher-quality rubrics.

9 Related Work

Test outcome has been a common metric for code correctness in program repair and translation evaluations [2, 4, 6, 13, 14, 22, 27, 34, 36]. As LLM-based APR systems are becoming more capable of generating high-quality patches [5, 8, 37, 39], developers need to further rely on manual assessment to determine patch validity (§1). Studies of other code generation tasks, such as code summarization and text-to-code [1, 18, 33], have used metrics derived from Natural Language Processing (NLP) literature to measure code structural and textual similarity [11, 23, 25, 41], as well as code naturalness [12]. However, these metrics do not capture the code validity that is critical to practical APR [6, 17, 27].

More recently, researchers started employing LLM-as-a-Judge as a substitute for human experts to perform more tailored and scalable evaluations on increasingly-complex generative tasks [30, 40]. In software engineering, recent studies explored the use of LLMs for evaluating and annotating coding task outputs such as generation [31, 44], summarization [3, 9], and execution [7, 20]. These studies have shown promising results with LLM judges, surpassing NLP metrics such as CodeBLEU [25, 31] and aligning with human experts. Our work complements existing studies by further exploring the use of LLMs to not only judge, but also to generate bug-level fix requirements (captured by our rubric) for reliable and scalable APR evaluation in an industrial context.

10 Threats to Validity

External Threats concern the generalizability of our results to other contexts. The study was limited to Google’s internal monorepo and sanitizer bugs (C++, Java, Go). Findings may not apply to open-source projects or other bug classes, such as logical or UI errors. We evaluated patches from only one internal APR system [26] and used only Geminin 2.5 Pro for LLM calls. Results may differ when applied to patches from other APR systems or using other LLMs.

Internal Threats concern potential confounding factors within our experimental setup. Raters were experienced engineers but not the original code authors, potentially lacking deeper context. We mitigated this threat with consensus discussions for benchmark labeling. The creation of golden rubrics involves human judgment, introducing potential variability. This was mitigated by having a second expert reviewer examine the final rubric.

Construct Threats concern whether our metrics accurately measure the concepts we claim to be studying. Though our use of a binary VALID/INVALID classification is standard, we note that patches have varying degrees of acceptability, and thus a binary judgment may, for example, overestimate the rate of patch rejection.

11 Conclusion

We introduce a human-in-the-loop framework centered on LLM-as-a-Judge to enable a more scalable and reliable offline evaluation for patches generated by APR systems. Our approach is motivated by the insight that manual patch assessment suffers from low inter-rater reliability unless guided by a shared rubric. Our framework operationalizes this insight by having human experts refine an LLM-generated rubric once per bug. The rubrics can then be used repeatedly by an LLM judge to evaluate patches for the same bug.

Our empirical study on 115 patches demonstrates the framework’s potential as a reliable judge. It achieves high recall (0.93) and proves highly reliable in its negative predictions (NPV of 0.94). When considering the 70.4% of patches where human raters unanimously agreed, our judge reached substantial alignment (Cohen’s kappa 0.75) with high precision (0.8) and recall (0.94). However, the judge’s alignment on all the patches remains moderate (Cohen’s kappa 0.57). The current results suggest that LLM-as-a-Judge can be reliable on bugs and patches where human annotators are reliable, and can be an effective automated screener that discards invalid, yet plausible patches that pass bug reproduction tests.

References

- [1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
- [2] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590* (2021).
- [3] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can llms replace manual annotation of software engineering artifacts? In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 526–538.
- [4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [7] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic bug reproduction for effective automated program repair at google. *arXiv preprint arXiv:2502.01821* (2025).
- [8] Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blstein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multi-modality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).
- [9] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaoletti, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* (2025).
- [10] Diffutils 2025. Unified Diff. https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.
- [11] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [12] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.
- [13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues?, 2024. URL <https://arxiv.org/abs/2310.06770> 7 (2023).
- [14] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radíček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.
- [15] René Just, Darioosh Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 437–440. doi:10.1145/2610384.2628055 Tool demo.
- [16] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. *arXiv:2403.01971 [cs.SE]* <https://arxiv.org/abs/2403.01971>
- [17] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
- [18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
- [19] Chandra Maddila, Adam Tait, Claire Chang, Daniel Cheng, Nauman Ahmad, Vijayaraghavan Murali, Marshall Roch, Arnaud Avondet, Aaron Meltzer, Victor Montalvao, et al. 2025. Agentic Program Repair from Test Failures at Scale: A Neuro-symbolic approach with static analysis and test execution feedback. *arXiv preprint arXiv:2507.18755* (2025).
- [20] Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. SWT-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems* 37 (2024), 81857–81887.
- [21] Gustavo A Oliva, Gopi Krishnan Rajbahadur, Aaditya Bhatia, Haoxiang Zhang, Yihao Chen, Zhihong Chen, Arthur Leung, Dayi Lin, Boyuan Chen, and Ahmed E Hassan. 2025. SPICE: An Automated SWE-Bench Labeling Pipeline for Issue Clarity, Test Coverage, and Effort Estimation. *arXiv preprint arXiv:2507.09108* (2025).
- [22] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [23] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
- [24] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 international symposium on software testing and analysis*. 24–36.
- [25] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
- [26] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating agent-based program repair at google. *arXiv preprint arXiv:2501.07531* (2025).
- [27] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lampe. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.
- [28] Sanitizers 2025. Google Sanitizers. <https://github.com/google/sanitizers>.
- [29] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-in-the-loop software development agents. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 342–352.
- [30] Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. 2024. Judgebench: A benchmark for evaluating llm-based judges. *arXiv preprint arXiv:2410.12784* (2024).
- [31] Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184* (2024).
- [32] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *arXiv preprint arXiv:2503.15223* (2025).
- [33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [34] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
- [35] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 819–831. doi:10.1145/3650212.3680323
- [36] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*. 1–10.
- [37] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [38] Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. 2025. Utboost: Rigorous evaluation of coding agents on swe-bench. *arXiv preprint arXiv:2506.09289* (2025).
- [39] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
- [40] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohai Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.
- [41] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527* (2023).
- [42] Qihao Zhu, Zeyu Sun, Yuan an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2022. A Syntax-Guided Edit Decoder for Neural Program Repair. *arXiv:2106.08253 [cs.SE]* <https://arxiv.org/abs/2106.08253>
- [43] Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayna Longpre, Kevin Meng, Rebecca Weiss, et al. 2025. Establishing Best Practices for Building Rigorous Agentic Benchmarks. *arXiv preprint arXiv:2507.02825* (2025).
- [44] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbulin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghu Raman Krishnamoorthi, Yuandong Tian, et al. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934* (2024).