

APCA in the Loop: An Empirical Study of In-Loop Patch Correctness Assessment for APR

Sahand Moslemi Yengejeh 

Computer Engineering Department, Bilkent University, Ankara, Turkey

Anil Koyuncu 

Computer Engineering Department, Bilkent University, Ankara, Turkey

Abstract

Automated program repair (APR) tools generate candidate patches and accept those that pass a given test suite. A test-passing patch, called a plausible patch, is not necessarily correct: it may overfit the test suite without fixing the underlying bug. Automated patch correctness assessment (APCA) tools were developed to detect such overfitting patches, but they have been evaluated only as post-hoc classifiers on static datasets, never inside the repair loop they are intended to support. We investigate whether placing an APCA tool inside the generate-and-validate loop of an APR system, as a second validation gate, affects the APCA's classification effectiveness, the APR tool's repair capability, and the search cost it incurs. In this design, a candidate that passes all tests but is classified as overfitting by the APCA tool is discarded, and the repair tool continues generating candidates. We study it with three APR tools (TBar, ARJA, ChatRepair) crossed with three APCA tools (ODS, Quatrain, LLM4PatchCorrect), evaluated on 549 single-method bugs of Defects4J v3 with manual correctness assessment of 1,328 patches drawn from the in-loop candidate streams, and paired non-parametric tests (McNemar, Wilcoxon) per (APR, APCA) cell. Average in-loop F_1 on the correct class sits between 0.43 and 0.66 across the nine cells, with classification harder in-loop than post-hoc on every tool. The in-loop gate raises the share of correct outputs in 14 of 18 cells, but cuts the absolute correct-fix count whenever the APR baseline's first plausible patch is already correct at a high rate. The in-loop introduces a search-cost overhead that is statistically significant in 14 of 18 cells (Wilcoxon $p < .05$), with a geometric-mean per-bug multiplier of up to $17.46\times$. Placing an APCA inside the repair loop raises the output correctness rate in most (APR, APCA) cells, but at a search-cost overhead and, where the APR baseline already produces correct first plausibles at a high rate, a reduction in absolute correct fixes. The findings indicate that APCA tools should be trained and evaluated on the in-loop patch distribution they are deployed against, rather than on the curated, terminal patches of static post-hoc datasets.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Automatic programming

Keywords and phrases automated program repair, patch correctness assessment, patch overfitting

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.77

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material

Software (Replication Package): <https://doi.org/10.5281/zenodo.20456574>

Funding This study was supported by Scientific and Technological Research Council of Turkey (TUBITAK) under the Grant Number 124E112.

Acknowledgements The authors thank TUBITAK for their support.



© Sahand Moslemi Yengejeh and Anil Koyuncu;
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. 77; pp. 77:1–77:15



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Automated program repair (APR) generates patches without direct human involvement under a generate-and-validate paradigm, where a candidate that passes all tests is called a plausible patch [11, 14, 24, 4, 12]. Plausible patches are frequently not correct [17, 9, 23], a problem known as patch overfitting [18, 17] that automated patch correctness assessment (APCA) tools address by predicting whether each plausible patch is correct or overfitting [30, 21, 27, 3].

Almost every published APCA tool operates after the generate-and-validate loop has terminated [22, 29, 2]: the repair tool runs to completion, and the APCA tool inspects its output. We refer to this arrangement as a *post-hoc* design: the APCA acts only after the repair tool has committed its output. Under such a design, APCA can reject an overfitting patch at the end, but it cannot extend the search. If the repair tool returned an overfitting patch, the opportunity to explore further candidates in the same APR run has already passed. Beyond this, we argue that evaluating APCA tools under the post-hoc design, as in most existing studies, does not reflect a real-world evaluation of these tools. APCA tools produce false labels, and human expert assessment is inevitable for both positive and negative cases. Whether the labels produced by an APCA tool enhance the repair process itself, rather than merely flagging an output the APR has already returned, has yet to be addressed.

To the best of our knowledge, no prior work has placed a general-purpose, standalone APCA classifier inside the generate-and-validate loop and studied the consequences. The closest, AdverIntent-Agent [7] and PRISM [19], either couple their filter to a single LLM repair agent or still apply after the APR has terminated.

To close this gap, we use APCA tools as *active filters* inside the generate-and-validate loop, placing them as a second validation gate. A candidate that passes all tests but is classified as overfitting by the APCA tool is treated as failed, and the repair tool continues to the next candidate. We call a patch that passes both gates an *approved patch*, a stricter notion of plausibility that requires both test-suite satisfaction and APCA approval. We investigate three research questions on this design:

- RQ1.** How effectively do APCA tools operate within the in-loop architecture?
- RQ2.** What is the impact of the in-loop architecture on the repair capability of APR tools?
- RQ3.** What is the impact of the in-loop architecture on the search cost of APR tools?

We instantiate the design with three APR tools (TBar [14], ARJA [32], ChatRepair [25]) and three APCA tools (ODS [30], Quatrain [21], LLM4PatchCorrect [35]) on Defects4J v3 [6], labelling 1,328 patches. The in-loop gate raises the share of correct outputs in 14 of 18 APR and APCA tool combinations but reduces the correct fix count where the baseline first plausible is already mostly correct, and adds a statistically significant search-cost overhead (Wilcoxon $p < .05$, geometric-mean per-bug multiplier up to 17.46 $\times$).

Contributions. We release an empirical study that evaluates APCA tools on the patch distribution the loop actually produces, a method that integrates any standalone APCA tool inside an APR loop as a second validation gate, and a labeled corpus of 1,328 in-loop plausible patches.

Sections 2–5 present background, setup, and results. Section 6 discusses implications, Section 7 addresses threats, and Section 8 concludes.

2 Background and Related Work

Automated Program Repair. Generate-and-validate APR produces candidate patches and halts at the first one that passes the test suite, called a plausible patch [4, 12]. Plausible patches are frequently not correct [17, 9, 23]. They satisfy the tests by exploiting coverage gaps rather than implementing the intended fix, a problem known as patch overfitting [18, 16]. Generation strategies range from template- and search-based tools (TBar [14], ARJA [32]) to learning- and LLM-based ones (ChatRepair [25]). In all of these families, the acceptance criterion is the test suite, leaving overfitting open.

The most recent line of APR work moves beyond this strict framework. Agent-based systems let an LLM autonomously invoke tools, inspect code, and reason about whether a candidate is correct before returning it. SWE-agent [28], AutoCodeRover [34], and RepairAgent [1] therefore embed a notion of in-loop correctness assessment, but that assessment is intertwined with the agent’s own generation step and cannot be detached as a standalone classifier applied to patches from other APR systems. Whether placing an external, standalone APCA classifier inside the loop of an arbitrary APR tool changes its repair behaviour is a separate question and has not been studied.

Automated Patch Correctness Assessment. APCA tools take a plausible patch and predict whether it is correct or overfitting using signals such as execution traces [27], generated tests [26, 31], static features [30, 13, 5], learned code representations [33, 10], natural-language correlations [21], and pre-trained language models [35, 3]. Across all of them, the evaluation is the same: classify each patch in a labelled dataset collected after the APR has terminated [23, 29, 2]. Under this setting, the APCA acts as a passive annotator that can flag the patch the APR has already returned, but is never measured on the longer and noisier candidate stream it would face inside the loop.

Two recent works move closer to our setting. AdverIntent-Agent [7] embeds adversarial test generation inside a multi-agent LLM repair system to filter overfitting patches as they are generated. Its filter, however, is tightly coupled to artifacts produced by its own reasoning agent and cannot be detached as a standalone APCA classifier for other APR tools, so the design conflates the LLM repair agent and the APCA filter. PRISM [19] is a standalone APCA classifier that scores each APR tool’s ranked plausible sequence and reports top- k and review-effort metrics. It operates on the order in which plausibles arrive but still applies after the APR run, so it does not extend the search and study the effects of it. To the best of our knowledge, no prior work places a general-purpose, standalone APCA classifier inside the generate-and-validate loop of an arbitrary APR tool as a second validation gate.

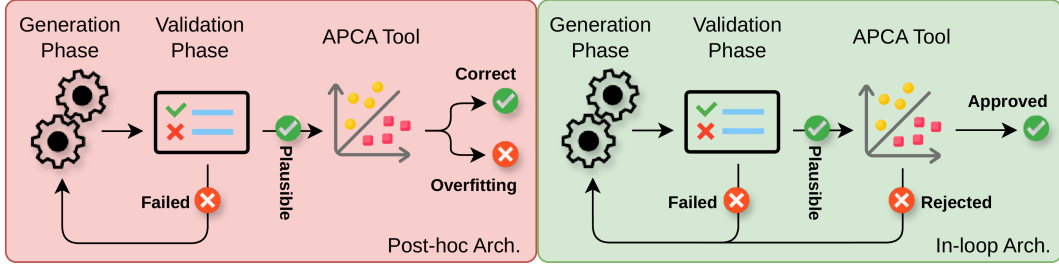
3 Methodology

The fundamental difference between the prevailing post-hoc architecture and our proposed in-loop architecture lies in the feedback loop of the repair process (Figure 1).

Let $C = \{c_1, c_2, \dots\}$ be the ordered stream of candidate patches generated by an APR tool for a given bug. We define a validation test suite as a predicate $T : C \rightarrow \{0, 1\}$, where $T(c_i) = 1$ if c_i passes all tests (making it a *plausible patch*), and 0 otherwise. We define an APCA classifier as a predicate $A : C \rightarrow \{0, 1\}$, where $A(c_i) = 1$ if the classifier predicts the patch is correct (making it an *approved patch*), and 0 if it predicts overfitting.

In the post-hoc automated assessment architecture, the automated assessment phase is decoupled from the search. The generation and validation phases run until they find the first candidate $c_{baseline}$ that satisfies T :

$$c_{baseline} = \arg \min_{c_i \in C} \{i \mid T(c_i) = 1\} \quad (1)$$



■ **Figure 1** The prevailing post-hoc architecture (left) versus the in-loop architecture we propose.

The search terminates immediately, returning $c_{baseline}$ as the final repair output. Then the APCA tool A , if applied to $c_{baseline}$, yields $A(c_{baseline})$. If $A(c_{baseline}) = 0$, the patch is flagged as overfitting, but the opportunity to explore alternative candidates has already passed.

In contrast, in-loop architecture integrates the APCA tool as a second validation gate. The search terminates only when it identifies an *approved patch* $c_{approved}$ that satisfies both the test suite and the APCA gate, defined as follows:

$$c_{approved} = \arg \min_{c_i \in C} \{i \mid T(c_i) = 1 \wedge A(c_i) = 1\} \quad (2)$$

If $T(c_i) = 1$ but $A(c_i) = 0$, the candidate is rejected, and the generator is triggered to produce the next candidate. To prevent infinite execution, we impose a search budget limit. If the APR tool exhausts this budget without achieving an approved patch, the repair is marked as a failure. This design forces the APR tool to explore deeper into the patch space. The APCA works as an in-loop filter, not as a generator: it does not change how the APR tool produces candidates or their order in the candidate stream. It only changes the stopping rule, from the first plausible patch to the first approved patch. Because the APR tools do not repeat a candidate in the same candidate stream, no duplicate patch enters the loop.

4 Experimental Setup

APR Tools. We use TBar (template-based), ARJA (genetic programming), and ChatRepair (LLM-based) with their published default configurations (ChatRepair: multi-turn conversational). All three use perfect fault localization to isolate the assessment gate from localization errors.

APCA Tools. We use ODS (202 AST features via the Coming framework), Quatrain (BERT encoding of bug-report and patch pairs), and LLM4PatchCorrect (GPT-based correctness reasoning, `gpt-4.1` via the OpenAI API). ODS and Quatrain use the authors' pre-trained models (on Defects4J 1 tool-generated patches [23, 21]) at their default threshold.

Benchmark and Selection Criteria. We evaluate on Defects4J v3 [6] (854 active bugs, 17 projects). Following ChatRepair [25], we scope to single-method developer fixes (the granularity at which ChatRepair's infill and TBar's fix patterns operate), giving 549 eligible bugs across the single-line, single-hunk, and single-function categories. Because ODS and Quatrain were trained on Defects4J 1 patches [30, 21], we report every result split into Defects4J 1 (282 bugs, 6 original projects) and Defects4J 2 and 3 (267 bugs, 11 added projects).

■ **Table 1** Has-plausible and has-approved bug counts per APR tool.

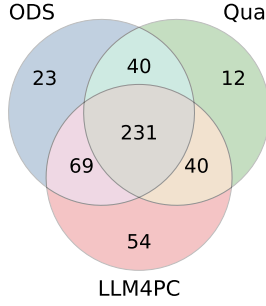
APR tool	Has-plausible	Has-approved		
		ODS	Quatrain	LLM4PC
Defects4J 1				
TBar	62	52	47	52
ARJA	11	4	6	4
ChatRepair	90	56	51	87
Defects4J 2 and 3				
TBar	49	31	23	27
ARJA	22	18	11	15
ChatRepair	176	133	123	174

Execution Procedure. We compare the baseline and in-loop configurations of Section 3. For each subject bug, the APR generates the candidate stream C and every plausible candidate is forwarded in parallel to the three APCAs, so they inspect the same sequence in the same order (important for non-deterministic APR tools like ChatRepair). When the APCA rejects a plausible patch, we do not restart the APR tool. It continues along the same candidate stream and moves to the next candidate. We use each APR tool’s own continue mode where it exists, and otherwise modify the tool’s implementation to behave the same way. The APR tools drop candidates that fail to compile or fail the tests. For a bug, the search halts when all three APCAs have approved a patch or the 30-minute budget is exhausted.

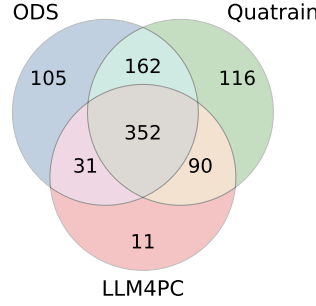
An (APR, bug) attempt is *has-plausible* when the APR produced at least one plausible patch and *has-approved* for an APCA when that APCA approved one. The APR tools together produced a plausible patch for 125 of 282 Defects4J 1 bugs and 189 of 267 Defects4J 2 and 3 bugs. Table 1 shows the per-APR tool has-plausible and has-approved attempt counts.

Metrics. We quantify the impact of the architecture on the classification performance of each APCA, the repair capability of each APR tool, and the search cost of the repair process, each reported per APR and APCA combination within each Defects4J group. Every metric carries subscript B for the baseline (post-hoc) and L for the in-loop. The repair rate RR is the share of returned patches that are correct, the correct-fix count CF is the total number of correct patches returned, and the search cost is the number of candidates the APR tool examines before it halts. HP and HA are the number of has-plausible and has-approved attempts.

- **APCA Performance (RQ1):** accuracy, precision, recall, and F_1 on the *correct* class for each APCA, evaluated on the in-loop patch distribution rather than on pre-collected datasets.
- **Repair Capability (RQ2):** per has-plausible attempt, the baseline correct fixes CF_B (has-plausible attempts whose $c_{baseline}$ is correct) and the in-loop correct fixes CF_L (has-approved attempts whose $c_{approved}$ is correct). The repair rates $RR_B = CF_B/HP$ and $RR_L = CF_L/HA$ give the share of correct outputs under each architecture (HP , HA from Table 1). The filtering effects are $\Delta CF = CF_L - CF_B$ and $\Delta RR = RR_L - RR_B$, positive when the in-loop wins. We test the matched per-bug binary outcomes with **McNemar’s test** (exact binomial for discordant counts < 25 , Edwards’ continuity correction otherwise) and report the continuity-corrected **odds ratio** OR , the number of bugs the baseline gets right but the in-loop misses divided by the number the in-loop gets right but the baseline misses. An OR above 1 means the in-loop drops more correct fixes than it recovers, and an OR below 1 means it recovers more.



■ **Figure 2** Split of the 1,328 manually labelled patches across the three APCA tools by which tool inspected, for Defects4J 1 (left, 626 patches) and Defects4J 2 and 3 (right, 702 patches).



■ **Figure 3** Number of the 8,984 generated plausible patches each APCA tool inspected before approving, and how those subsets overlap.

- **Search Cost (RQ3):** per has-approved bug: the baseline candidate position $B_i = (k \mid c_k = c_{baseline})$ and the in-loop position $L_i = (k \mid c_k = c_{approved})$. Per cell we report n (has-approved count), the medians $\tilde{B}$ and $\tilde{L}$, the median per-bug ratio $\tilde{L}/\tilde{B}$, and the geometric mean $GM(L/B)$. We test the paired values with the two-sided **Wilcoxon signed-rank test** (zeros dropped) and report the matched-pair **Cliff's δ** (negligible $|\delta| < 0.147$, small < 0.33 , medium < 0.474 , large otherwise).

Every paired statistical test uses $\alpha = 0.05$. We choose these paired tests to match the data. In RQ2 each bug gives two yes/no outcomes, correct under the baseline and correct under the in-loop, so we use McNemar's test, which is built for paired binary outcomes. In RQ3 each bug gives two candidate positions, B_i and L_i , whose per-bug ratios are skewed and far from normal, so we use the two-sided Wilcoxon signed-rank test, a paired test that does not assume a normal distribution.

Manual Patch Correctness Assessment. The APR tools generated 8,984 plausible patches in total. Figure 3 shows how many each APCA inspected. For the RQ1 and RQ2 ground truth, we label 1,328 patches (626 Defects4J 1, 702 Defects4J 2 and 3) in three stages, split across the APCAs as in Figure 2.

- **Stage 1:** every *first plausible* patch (410), establishing CF_B .
- **Stage 2:** every *first approved* patch (485, of which 163 are new beyond Stage 1), establishing CF_L .
- **Stage 3:** after excluding Stage 1 and 2 patches, we draw up to 60 Defects4J 1 and 60 Defects4J 2 and 3 patches uniformly at random from each of the seven Venn regions of Figure 3 (taking all when a region has fewer in a group). This yields 755 patches (414 Defects4J 1, 341 Defects4J 2 and 3), exceeding the Cochran sample size for a 95% confidence interval of $\pm 6\%$ at $p = 0.5$ in every per-APCA, per-group total.

We first reuse verdicts from prior studies [27, 15, 20, 30, 23] (1,469 correct, 64,939 overfitting) plus the developer patches: for each candidate we extract its target file and look for an *exact* match (identical bodies after stripping comments and whitespace) and, failing that, an *AST* match (identical `javalang` ASTs). A matched candidate inherits its reference's verdict. Exact matching resolves 103 Defects4J 1 and 111 Defects4J 2 and 3 patches, and AST matching resolves a further 62 and 13.

■ **Table 2** Performance of APCA tools under the post-hoc baseline and the in-loop architecture.

APR	Defects4J 1										Defects4J 2 and 3									
	Post-hoc					In-loop					Post-hoc					In-loop				
	n	Acc	Pr	Re	F1	n	Acc	Pr	Re	F1	n	Acc	Pr	Re	F1	n	Acc	Pr	Re	F1
ODS																				
TBar	62	.69	.92	.68	.78	89	.65	.79	.67	.73	49	.69	.47	.50	.48	90	.74	.52	.67	.58
ARJA	11	.73	.00	.00	.00	127	.89	.25	.08	.12	22	.82	1	.56	.71	90	.70	.72	.37	.49
ChatRepair	90	.59	.88	.40	.55	231	.58	.80	.35	.48	176	.55	.89	.43	.58	378	.59	.81	.45	.58
W. Average	163	.64	.84	.48	.60	447	.68	.64	.34	.43	247	.60	.81	.45	.57	558	.63	.75	.47	.57
Quatrain																				
TBar	62	.74	.97	.70	.81	101	.73	.91	.65	.76	49	.80	.70	.50	.58	113	.84	.83	.58	.68
ARJA	11	.73	.50	.67	.57	118	.96	.50	.60	.55	22	.82	1	.56	.71	110	.81	1	.34	.51
ChatRepair	90	.58	.88	.39	.54	227	.64	.88	.37	.52	176	.58	1	.41	.58	374	.61	.91	.46	.61
W. Average	163	.65	.89	.52	.64	446	.74	.79	.50	.58	247	.64	.94	.44	.60	597	.69	.91	.46	.60
LLM4PC																				
TBar	62	.77	.93	.78	.85	124	.82	.88	.74	.81	49	.80	.64	.64	.64	122	.83	.70	.59	.64
ARJA	11	.91	1	.67	.80	191	.95	.75	.27	.40	22	.86	.88	.78	.82	117	.79	.80	.35	.49
ChatRepair	90	.62	.64	.93	.76	103	.60	.62	.87	.72	176	.67	.71	.90	.80	213	.66	.70	.85	.77
W. Average	163	.70	.77	.86	.79	418	.83	.76	.56	.60	247	.71	.71	.84	.77	452	.74	.73	.65	.66

The remaining 461 Defects4J 1 and 578 Defects4J 2 and 3 patches are classified by two researchers using the semantic-equivalence rules of Liu et al. [15] (correct iff semantically equivalent to the developer fix or logically resolving the root cause without side effects). Disagreements are resolved through discussion, and labelling averaged six minutes per patch. Before the discussion step, the two researchers' independent labels agreed at a Cohen's κ of 0.79. The final corpus has 600 correct (240 Defects4J 1, 360 Defects4J 2 and 3) and 728 overfitting (386 and 342), totalling 1,328.

5 Evaluation

5.1 RQ1: APCA Performance in the Loop

Approach. RQ1 checks whether each APCA keeps its classification effectiveness on the longer, noisier in-loop patch stream it is deployed on but is never tuned or reported on. To measure performance of these tools, we evaluate each APCA against the 1,328 manually labeled patches of Section 4, using the classification metrics of Section 4. Because each APCA stops at its first approval, the three inspection subsets differ. Figure 2 shows how the corpus splits across them. As a reference point, we also score each APCA on the 410 `plausible_1` patches (Stage 1) under a post-hoc baseline. We pool the labeled in-loop patches each APCA inspected without reweighting by Venn region. The first-plausible and first-approved patches are labeled in full, and the Stage 3 sample covers the remaining regions at the Cochran size for a 95% confidence interval of $\pm 6\%$.

Results. Table 2 reports the per-cell metrics together with an Average row that gives each APCA's n -weighted mean across the three APR streams within a Defects4J group. The columns give n (labeled patches scored), accuracy (Acc), precision (Pr), recall (Re), and F_1 on the correct class, under the post-hoc baseline and the in-loop gate. Several cells show high accuracy with low F_1 on the correct class (ARJA with ODS: accuracy 0.89 at F_1 0.12 on Defects4J 1), so we read the table through the Average row's F_1 on the correct class.

- **Finding 1 (LLM4PC narrowly leads).** The average in-loop F_1 on the correct class ranges from 0.43 (ODS, Defects4J 1) to 0.66 (LLM4PC, Defects4J 2 and 3). LLM4PC leads on this average in both Defects4J groups, but the three APCAs sit close together rather than separating sharply.
- **Finding 2 (No training-distribution advantage).** ODS and Quatrain were both trained on patches generated for Defects4J 1 bugs [23, 21], so a higher in-loop F_1 on Defects4J 1 than on Defects4J 2 and 3 would be expected. Instead, the average in-loop F_1 on Defects4J 2 and 3 matches or exceeds the Defects4J 1 value for all three APCAs (ODS 0.43 $\rightarrow$ 0.57, Quatrain 0.58 $\rightarrow$ 0.60, LLM4PC 0.60 $\rightarrow$ 0.66). Because the tools do no better on their own training benchmark, their Defects4J 1 results are not inflated by training-data overlap, so we report them next to Defects4J 2 and 3 rather than dropping them.
- **Finding 3 (In-loop is harder than post-hoc).** Average F_1 drops on every tool when moving from post-hoc to in-loop on Defects4J 1 (LLM4PC 0.79 $\rightarrow$ 0.60, ODS 0.60 $\rightarrow$ 0.43, Quatrain 0.64 $\rightarrow$ 0.58). On Defects4J 2 and 3, LLM4PC drops 0.77 $\rightarrow$ 0.66 and the others stay flat. The longer in-loop stream contains overfitting near-duplicates the APCA must reject, lowering recall.

RQ1 summary. In-loop classification is harder than post-hoc for every APCA. The average in-loop F_1 on the correct class stays between 0.43 and 0.66, and LLM4PC leads by a small margin.

5.2 RQ2: Impact on Repair Capability

Approach. RQ2 checks whether the in-loop gate changes what an APR tool returns, both the rate of correct outputs and the absolute number of correct fixes, since a higher rate is only useful if it does not discard correct fixes the baseline would have returned. To answer it, we measure how the in-loop gate changes what each APR *returns*, using the repair-capability metrics of Section 4 on the paired baseline and in-loop outcomes per (APR, APCA) cell within each Defects4J group. Stage 3 patches are excluded from the ground truth: neither architecture returns them, since they are not first-plausible (the baseline output) and not first-approved by any APCA (the in-loop output).

Results. Table 1 shows LLM4PC the most permissive, approving at least one patch in 359 of 410 has-plausible attempts, and Quatrain the most restrictive at 261. Table 3 reports the per-cell metrics. Each row lists the baseline triple HP/CF_B/RR_B, the in-loop triple HA/CF_L/RR_L, the change in correct-fix count ΔCF with the change in rate ΔRR in brackets, and McNemar’s p with the odds ratio OR in brackets. The sign of ΔCF largely tracks the baseline repair rate RR_B. Where RR_B is high (TBar and ChatRepair on Defects4J 1, and ChatRepair on Defects4J 2 and 3), the baseline’s first plausible patch is already correct most of the time, so a conservative APCA that rejects the attempt or approves a later overfitting patch keeps ΔCF non-positive. On Defects4J 2 and 3, where RR_B is low (TBar and ARJA, RR_B = 0.29 and 0.41), the longer search reaches correct patches the first-plausible halt would have missed, so ΔCF turns positive. ARJA on Defects4J 1 also stays non-positive, from a has-plausible set of 11 too small to gain from despite its low rate.

- **Finding 4 (In-loop raises output correctness rate in most cells).** $\Delta RR > 0$ in 14 of 18 cells, with the largest gains on the conservative APCAs over the low-RR_B APR streams of Defects4J 2 and 3 (ARJA with Quatrain +0.59, TBar with Quatrain +0.54). The four cells with $\Delta RR \leq 0$ are all within 0.02 of zero, and ChatRepair with LLM4PC keeps RR_L $\approx$ RR_B in both groups by approving nearly every plausible patch.

■ **Table 3** Repair capability of different APR & APCA configurations.

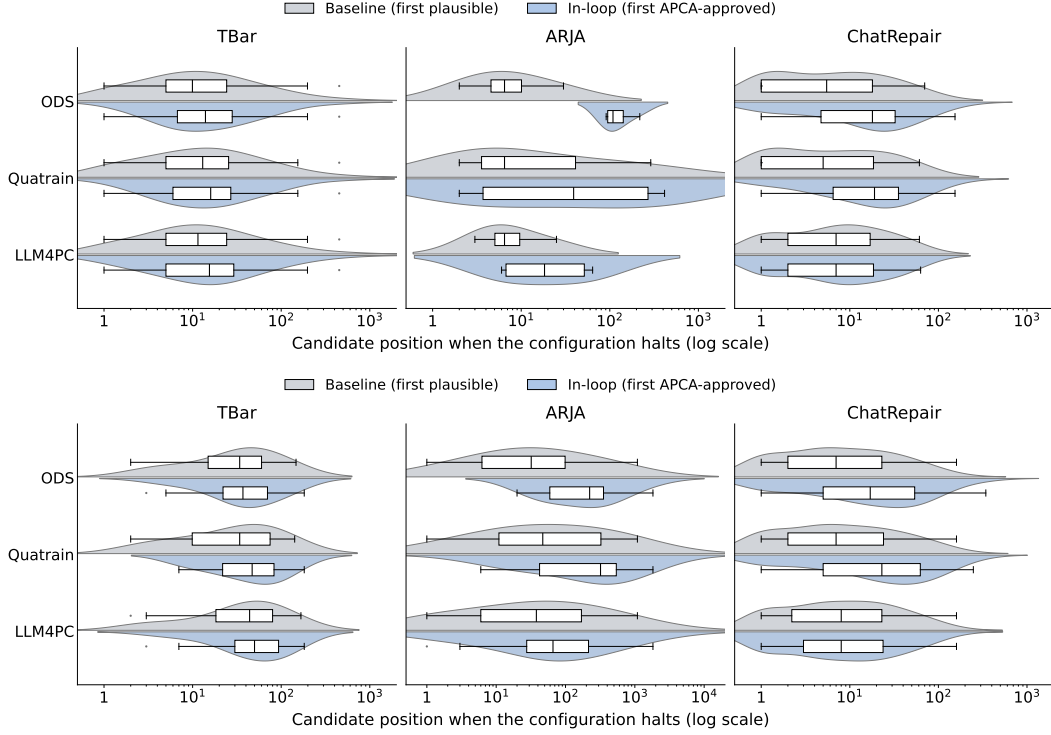
		HP/CF _B /RR _B	HA/CF _L /RR _L	ΔCF (ΔRR)	p (OR)
Defects4J 1	TBar&ODS	62/50/.81	52/41/.79	-9 (-.02)	.004 (19)
	TBar&Quatrain	62/50/.81	47/43/.91	-7 (+.11)	.039 (5.67)
	TBar&LLM4PC	62/50/.81	52/46/.88	-4 (+.08)	.219 (3.67)
	ARJA&ODS	11/3/.27	4/1/.25	-2 (-.02)	.500 (5)
	ARJA&Quatrain	11/3/.27	6/3/.50	0 (+.23)	1 (1)
	ARJA&LLM4PC	11/3/.27	4/3/.75	0 (+.48)	1 (1)
	ChatRepair&ODS	90/57/.63	56/45/.80	-12 (+.17)	.002 (9)
	ChatRepair&Quatrain	90/57/.63	51/45/.88	-12 (+.25)	.002 (9)
	ChatRepair&LLM4PC	90/57/.63	87/54/.62	-3 (-.01)	.375 (3)
Defects4J 2 and 3	TBar&ODS	49/14/.29	31/16/.52	+2 (+.23)	.727 (0.64)
	TBar&Quatrain	49/14/.29	23/19/.83	+5 (+.54)	.227 (0.41)
	TBar&LLM4PC	49/14/.29	27/19/.70	+5 (+.42)	.227 (0.41)
	ARJA&ODS	22/9/.41	18/13/.72	+4 (+.31)	.219 (0.27)
	ARJA&Quatrain	22/9/.41	11/11/1	+2 (+.59)	.625 (0.43)
	ARJA&LLM4PC	22/9/.41	15/12/.80	+3 (+.39)	.375 (0.33)
	ChatRepair&ODS	176/126/.72	133/108/.81	-18 (+.10)	.001 (4.27)
	ChatRepair&Quatrain	176/126/.72	123/112/.91	-14 (+.19)	.011 (3.15)
	ChatRepair&L	176/126/.72	174/122/.70	-4 (-.01)	.125 (9)

■ **Table 4** Patch-correctness transitions from the baseline first plausible patch to the in-loop output, summed over the nine (APR, APCA) cells in each Defects4J group.

Transition	Defects4J 1	Defects4J 2 and 3
<i>From a correct first plausible patch</i>		
to a correct approved patch (kept)	275	388
to an incorrect approved patch	17	21
to no output	38	38
<i>From an incorrect first plausible patch</i>		
to a correct approved patch (recovery)	6	44
to no correct patch	153	250
Paired outcomes	489	741

- **Finding 5 (The rate gain coexists with a count loss).** Despite the positive ΔRR , ΔCF is negative or zero in every Defects4J 1 cell and in all three ChatRepair Defects4J 2 and 3 cells: the in-loop returns fewer correct fixes in absolute count because the APCA gate rejects many has-plausible attempts. Only the TBar and ARJA cells of Defects4J 2 and 3, where RR_B is low, turn ΔCF positive.
- **Finding 6 (The count loss is statistically significant for conservative APCAs).** McNemar's test rejects equality of paired outcomes for ChatRepair with ODS and ChatRepair with Quatrain in both Defects4J groups ($p \leq .011$, OR up to 9) and for TBar with ODS ($p = .004$, OR = 19) and TBar with Quatrain ($p = .039$, OR = 5.67) on Defects4J 1. Other cells do not reach $\alpha = 0.05$ due to small discordant-pair counts.

To show where the count change in Table 3 comes from, we track each has-plausible bug from its baseline first plausible patch to the in-loop output, summed over the nine (APR, APCA) cells per group (Table 4). On Defects4J 1, where the baseline is already correct most of the time, the gate turns 55 correct first plausibles into an incorrect approved patch (17) or no output (38) and recovers only 6, which is the count loss of Finding 5. On Defects4J 2 and 3 the recoveries rise to 44, and 33 of them fall on the low- RR_B TBar and ARJA streams, where the longer search reaches correct patches the first-plausible halt would have missed.



■ **Figure 4** Distribution of candidate positions at which the baseline (first plausible) and the in-loop (first approved) halt, over has-approved bugs, for Defects4J 1 (top) and Defects4J 2 and 3 (bottom).

RQ2 summary. The in-loop gate raises the correct-output rate ($\Delta\text{RR} > 0$) in 14 of 18 cells, but it lowers the absolute correct-fix count wherever the baseline first plausible patch is already correct at a high rate.

5.3 RQ3: Impact on Search Cost

Approach. RQ3 checks how much extra search the in-loop gate costs by pushing the APR tool deeper into the candidate stream, since the rate gain from RQ2 is only worthwhile if the added cost stays acceptable. To answer it, we quantify the search-cost overhead from replacing the first-plausible halting rule of Section 3 with a first-approved halting rule, using the per-bug pairs (B_i, L_i) and the cell-level summaries of Section 4. We restrict to has-approved bugs, where both halting points exist. Summed across the nine (APR, APCA) cells (each of the 549 eligible bugs contributes up to nine matched pairs), this yields 359 matched observations on Defects4J 1 and 555 on Defects4J 2 and 3.

Results. Table 5 shows $\tilde{L} \geq \tilde{B}$ in every cell of both Defects4J groups. The columns give n (has-approved bugs), the median baseline position $\tilde{B}$, the median in-loop position $\tilde{L}$, the median per-bug ratio $\tilde{L}/\tilde{B}$, the geometric mean $\text{GM}(L/B)$, the Wilcoxon p , and Cliff’s δ . The split-violin shape in Figure 4 is consistent across cells: the in-loop distribution preserves the baseline shape and shifts right on the log axis by a tool-dependent factor. ARJA with ODS drives the upper end of the GM range ($\text{GM} = 17.46$ on Defects4J 1, 5.21 on Defects4J 2 and 3), with median in-loop candidate positions of 110 and 224 against baseline medians of 6 and 32.

■ **Table 5** Search cost per APR and APCA combination.

	Defects4J 1							Defects4J 2 and 3						
	n	$\widetilde{B}$	$\widetilde{L}$	$\widetilde{L/B}$	GM(L/B)	p	δ	n	$\widetilde{B}$	$\widetilde{L}$	$\widetilde{L/B}$	GM(L/B)	p	δ
TBar&ODS	52	10	14	1	1.34	<.001	+.29	31	34	37	1.06	1.31	<.001	+.52
TBar&Quatrain	47	13	16	1	1.25	.003	+.23	23	34	47	1.18	1.59	.001	+.57
TBar&LLM4PC	52	12	16	1	1.25	.005	+.19	27	44	50	1	1.42	.001	+.48
ARJA&ODS	4	6	110	26.95	17.46	.125	+1	18	32	224	3.62	5.21	.001	+.72
ARJA&Quatrain	6	6	112	1	2.43	.500	+.33	11	47	320	1.52	3.47	.031	+.55
ARJA&LLM4PC	4	6	28	1.46	2.53	.500	+.50	15	38	66	1	1.98	.018	+.47
ChatRep&ODS	56	6	18	1.13	2.29	<.001	+.54	133	7	17	1.11	2.05	<.001	+.54
ChatRep&Quatrain	51	5	19	1.03	2.61	<.001	+.51	123	7	23	1.20	2.43	<.001	+.58
ChatRep&LLM4PC	87	7	7	1	1.07	.068	+.05	174	8	8	1	1.03	<.001	+.08

- **Finding 7 (Search-cost overhead).** GM(L/B) ranges from 1.07 to 17.46 on Defects4J 1 and 1.03 to 5.21 on Defects4J 2 and 3. TBar pays the least (GM ≤ 1.59 everywhere, $\widetilde{L/B} = 1$ on every Defects4J 1 cell), ChatRepair a moderate amount (GM ≤ 2.61 , LLM4PC near 1), and ARJA the most (driven by ARJA with ODS).
- **Finding 8 (The overhead is statistically significant in most cells).** Wilcoxon’s signed-rank test rejects equality of B_i and L_i in 14 of 18 cells ($p < .05$, $p < .001$ in 7). Cliff’s δ is positive throughout, with 11 cells in the large band. The four non-significant cells are the three small- n ARJA Defects4J 1 cells ($n \leq 6$, masking δ between +.33 and +1) and ChatRepair with LLM4PC on Defects4J 1 ($\delta = +.05$, negligible, and the Defects4J 2 and 3 cell reaches $p < .001$ at $\delta = +.08$, large n but small effect).

RQ3 summary. The in-loop gate adds a search-cost overhead that is significant in 14 of 18 cells (Wilcoxon $p < .05$), with a geometric-mean per-bug multiplier up to $17.46\times$.

6 Discussion

In-loop changes the role of an APCA tool. The in-loop architecture turns each APCA classification into a search step (RQ1): a rejected plausible patch returns the APR tool to generation, and an approved patch becomes the repair output. The cost of a false negative is therefore higher than in post-hoc use. For instance, in TBar with ODS on Chart_1, ODS rejected the correct first plausible patch (a single-token `!=` to `==` flip) and later approved an overfitting `instanceof` check, replacing a correct repair with an incorrect one. Static post-hoc evaluation does not surface this failure mode.

When the gate helps and when it hurts. RQ2 shows the gate raises the share of returned patches that are correct, but ΔCF stays at most zero in every Defects4J 1 cell because the baseline already returns a correct first plausible patch for 27% to 81% of has-plausible attempts. The picture flips on TBar and ARJA Defects4J 2 and 3, where baseline correctness drops to 29% and 41% and the longer in-loop search reaches correct patches the baseline would have missed. RQ3 shows this rate gain comes at a search-cost overhead that is statistically significant in 14 of 18 cells, with a geometric-mean per-bug multiplier up to $17.46\times$, so the gate’s practical use has to weigh ΔRR against this multiplier.

Implications for APCA design. RQ1 shows that APCA classification, although evaluated until now on pre-collected terminal-patch datasets, also operates on the in-loop stream and generalises across Defects4J 1 and Defects4J 2 and 3. A natural methodological consequence

is that future APCA evaluations should report the repair-capability gain and the search-cost overhead the tool induces inside an APR loop (RQ2 and RQ3), not classification accuracy alone, so that the trade-off is visible. A triage policy that ranks plausibles by APCA score and lets the APR keep searching, instead of a binary approve-or-reject gate, would retain the ΔRR gain of RQ2 without the count loss it currently causes.

7 Threats to Validity

Internal Validity. Manual correctness labels are inherently subjective: Le et al. [8] found significant disagreement among human evaluators. We mitigate this by having two authors independently assess each patch, resolving disagreements through discussion, and cross-referencing against prior published labels where available [14, 30, 23]. TBar’s search is deterministic, so any in-loop deviation from the baseline output is attributable solely to the APCA gate. ARJA and ChatRepair are stochastic, which we mitigate by sharing the same generated candidate stream between the baseline and the in-loop configurations.

Construct Validity. Precision does not capture all aspects of practical utility: a configuration with higher precision but fewer correct fixes overall may not be preferable. We report both repair-rate metrics (RR_B , RR_L) and absolute correct-fix counts (CF_B , CF_L) to give a complete picture. The 30-minute per-bug wall-clock budget is a resource restriction. Some APR tools fail to produce any plausible patch within it on certain bugs, which trims the has-plausible set. Extending the budget is left to future work.

External Validity. We deliberately control two parts of the setup, and these controls limit external validity. First, we evaluate only on Defects4J v3, a Java benchmark, so the findings may not carry over to other tools, programming languages, or bug benchmarks. Second, all APR tools use perfect fault localization, isolating the APCA gate as the only difference between the in-loop and baseline runs. Under imperfect fault localization the absolute numbers would differ. We chose APR tools from three paradigms (template-based, genetic-programming-based, LLM-based) to diversify within the benchmark, but the space of possible combinations is large. ODS and Quatrain are used as their authors released them, trained on patches from earlier APR tools. Retraining on tool-specific patches could change results. ODS and Quatrain were trained on Defects4J 1 patches, so their Defects4J 1 numbers could gain from training overlap. We keep these rows for completeness but do not lean on them, since Finding 2 shows their in-loop F_1 on Defects4J 2 and 3 matches or exceeds Defects4J 1.

8 Conclusion

We presented the first systematic study of an APCA tool placed inside the generate-and-validate loop of an APR tool as a second validation gate, since every prior APCA evaluation operates post-hoc on the patches an APR has already returned. We ran three APR tools (TBar, ARJA, ChatRepair) crossed with three APCA tools (ODS, Quatrain, LLM4PatchCorrect) on the 549 single-method bugs of Defects4J v3, manually labelling the 1,328 patches that the evaluation relies on. APCA classification effectiveness in the loop is at most comparable to and usually below post-hoc. The gate raises the share of returned patches that are correct in 14 of 18 cells, but cuts the absolute correct-fix count whenever the APR baseline already produces correct patches at a high rate. The in-loop also incurs a search-cost overhead, statistically significant in 14 of 18 cells, of up to a $17.46\times$ geometric-mean increase. Closing this gap therefore requires APCA tools trained or calibrated for the in-loop patch distribution rather than the post-hoc one. That is the direction we leave to future work.

Data Availability Statement

The replication package, including tool configurations, patch logs, and manual correctness labels, is publicly available on Zenodo: <https://doi.org/10.5281/zenodo.20456574>.

References

- 1 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *Proc. of ICSE*, 2025.
- 2 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):55:1–55:50, 2025. doi:10.1145/3702972.
- 3 Marcos Fuster-Pena, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraz. RePaCA: Leveraging reasoning large language models for static automated patch correctness assessment. *arXiv preprint*, 2025. arXiv:2507.22580.
- 4 Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019. doi:10.1109/TSE.2017.2755013.
- 5 Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proc. of ISSTA*, pages 654–665, 2022. doi:10.1145/3533767.3534368.
- 6 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of ISSTA*, pages 437–440, 2014. doi:10.1145/2610384.2628055.
- 7 Sungmin Kang, Juyeon Yoon, Nadia Alshahwan, Shin Yoo, and Mark Harman. AdverIntent-Agent: Adversarial reasoning for repair based on inferred program intent. In *Proc. of ISSTA*, 2025.
- 8 Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *Proc. of ICSE*, pages 524–535, 2019.
- 9 Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23:3007–3033, 2018. doi:10.1007/S10664-017-9577-2.
- 10 Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. 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.
- 11 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi:10.1109/TSE.2011.104.
- 12 Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019. doi:10.1145/3318162.
- 13 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), 2022. doi:10.1145/3505247.
- 14 Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proc. of ISSTA*, pages 31–42, 2019. doi:10.1145/3293882.3330577.
- 15 Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proc. of ICSE*, pages 615–627, 2020. doi:10.1145/3377811.3380338.

- 16 Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 702–713, 2016. doi:10.1145/2884781.2884872.
- 17 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. of ISSTA*, pages 24–36, 2015. doi:10.1145/2771783.2771791.
- 18 Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proc. of ESEC/FSE*, pages 532–543, 2015. doi:10.1145/2786805.2786825.
- 19 Dowon Song and Hakjoo Oh. Enhancing APR with PRISM: A semantic-based approach to overfitting patch detection. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):392:1–392:29, 2025.
- 20 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 (ASE)*, pages 981–992, 2020. doi:10.1145/3324884.3416532.
- 21 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 *Proc. of ASE*, pages 1–13, 2022. doi:10.1145/3551349.3556914.
- 22 Bo Wang, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Patch overfitting in program repair: A survey. *Journal of Software: Evolution and Process*, 2024.
- 23 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 *Proc. of ASE*, pages 968–980, 2020. doi:10.1145/3324884.3416590.
- 24 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint*, 2024. doi:10.48550/arXiv.2407.01489.
- 25 Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. In *Proc. of ISSTA*, 2024.
- 26 Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proc. of ISSTA*, pages 226–236, 2017. doi:10.1145/3092703.3092718.
- 27 Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proc. of ICSE*, pages 789–799, 2018. doi:10.1145/3180155.3180182.
- 28 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Proc. of NeurIPS*, 2024.
- 29 Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proc. of ESEC/FSE*, pages 1203–1215, 2023. doi:10.1145/3611643.3616331.
- 30 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.
- 31 He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2), 2021. doi:10.1007/S10664-020-09920-W.
- 32 Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2020. doi:10.1109/TSE.2018.2874648.

- 33 Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. APPT: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering*, 50(2), 2024.
- 34 Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proc. of ISSTA*, 2024.
- 35 Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, 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.