PrevaRank: Ranking Plausible Patches by Historic Feature Frequencies

Shifat Sahariar Bhuiyan[a], Abhishek Tiwari[b,*], Yu Pei[c], Carlo A. Furia[a]

[a] *USI Università della Svizzera italiana, Lugano, Switzerland*
[b] *University of Southern Denmark, Odense, Denmark*
[c] *The Hong Kong Polytechnic University, Hong Kong, China*

Abstract

Automated program repair (APR) techniques have achieved conspicuous progress, and are now capable of producing genuinely *correct* fixes in scenarios that were well beyond their capabilities only a few years ago. Nevertheless, even when an APR technique can find a correct fix for a bug, it still runs the risk of *ranking* the fix lower than other patches that are plausible (they pass all available tests) but incorrect. This can seriously hurt the technique's practical effectiveness, as the user will have to peruse a larger number of patches before finding the correct one.

This paper presents PrevaRank, a technique that ranks plausible patches produced by any APR technique according to their feature similarity with historic programmer-written fixes for similar bugs. PrevaRank implements simple heuristics, which help make it scalable and applicable to any APR tool that produces plausible patches. In our experimental evaluation, after training PrevaRank on the fix history of 81 open-source Java projects, we used it to rank patches produced by 8 Java APR tools on 168 Defects4J bugs. PrevaRank consistently improved the ranking of correct fixes: for example, it ranked a correct fix within the top-3 positions in 27% more cases than the original tools did. Other experimental results indicate that PrevaRank works robustly with a variety of APR tools and bugs, with negligible overhead.

1. Introduction

Despite their significant technical improvements, automated program repair (APR) techniques remain mostly the *best effort*: the patches they produce come with no absolute guarantees of *correctness*. In practice, an APR tool will output several *plausible* (valid) patches for the faulty program given as input. Each plausible patch has only been validated against the test suite (also given as input), but it may or, more commonly, may not fully implement the expected, correct behavior for all possible inputs. The needle-in-a-haystack problem is

*Corresponding Author

thus *ranking* the plausible patches in a way that those that are correct—if any exist—come before those that are not. This way, when the developer goes through the list of patches in ranking order, they will not have to inspect many plausible fixes until they find one that they can accept as correct.

This paper presents PrevaRank, a technique that ranks plausible patches, trying to put any correct ones higher up in the ranking. It is common for an APR technique to include patch ranking heuristics as an integral part of its repair process—for example, by means of genetic algorithms [1], abstract state enumeration [2, 3], or similarity as evaluated by a classifier trained on historical fix data [4]. In contrast, PrevaRank works independent of any patch generation process, and hence can be used to rank the patches produced by any APR tool.

The key idea of PrevaRank is to rank a patch higher, the more syntactically similar it is to historic fixes for the same category of bugs. The historical data comes from mining software repositories for programmer-written fixes: for each bug category, PrevaRank records the syntactic features most frequently used by programmers to fix those bugs. For example, it may observe that null-pointer dereferencing errors are frequently fixed by adding a conditional guard that checks whether a call's target is **`null`**. Then, PrevaRank uses this frequency information to estimate whether a new patch is likely correct. For example, it will rank patches of null-pointer dereferencing errors higher (more likely to be correct) if they add a conditional guard to the program, and lower if they don't.

We designed PrevaRank's training and ranking processes to be as lightweight as possible: training simply records the relative frequencies of bug category/patch feature combinations; and ranking directly uses these frequencies to order patches for the same bug. While the same general idea could be implemented using more sophisticated machine learning algorithms, we want to demonstrate that PrevaRank's heuristics are effective even when implemented straightforwardly. Furthermore, PrevaRank's simplicity ensures that it easily scales to ranking a large number of patches; hence, it could be used as a post-processing step of any APR pipeline with negligible overhead.

We implemented PrevaRank and used it to rank 23 032 patches produced by 8 state-of-the-art APR tools for Java on 168 bugs in the popular Defects4J curated collection [5]. While Defects4J includes many more bugs, our evaluation could only consider bugs for which APR tools can produce at least four plausible patches—and hence *ranking* them is a meaningful challenge. Our experiments indicate that PrevaRank often successfully *improves* the ranking of correct patches over the one produced by the original tool: it improved the rank of 29% of the correct patches from outside to inside the top-3 ranks; and strictly improved the rank of 43% of the correct patches. At the same time, PrevaRank rarely worsens the ranking of correct fixes (and, when it does so, it tends to affect fixes that were already ranked very far from the top positions): in our experiments, it worsened the rank of only 2% of the correct patches from inside to outside the top-3 ranks. PrevaRank's effectiveness is largely independent of the nature of the patches—which APR tool produced them, and which bugs they repair. Our comparison with state-of-the-art patch ranking technique Shibboleth [6] indicates that PrevaRank's heuristics remain competitive against tools that use more

sophisticated machine learning techniques and include dynamic information (e.g., execution traces). Overall, these results suggest that PrevaRank is a widely applicable, scalable, effective technique to improve the ranking of correct fixes produced by APR systems—and thus it can help alleviate the aggravating patch overfitting problem.

1.1. LLMs for APR: Capabilities and Limitations

As the generative AI juggernaut continues to automate more and more software engineering tasks, can PrevaRank's approach remain useful in certain applications?

As noticed in recent surveys [7, 8, 9], the landscape of APR techniques has broadened greatly. It is clear that learning-based techniques are increasingly popular—especially those based on LLMs (Large Language Models). At the same time, different contributions target differ usage (and thus, evaluation) scenarios, which complicates comparing their performance and capabilities across the board.[1] This highlights a first strength of PrevaRank: since it is applicable to rank any set of (plausible) patches—independent of how the patches were generated in the first place—and it is very lightweight in terms of computational resources (as we discuss in quantitative terms below), the entry barrier for using PrevaRank is generally quite modest, which contributes to its flexibility.

On the other hand, if we extrapolate from the pace of progress of LLMs for automating software engineering tasks, we may be tempted to conclude that there won't be much room for traditional techniques such as PrevaRank. There is clear evidence, however, that there remain practical scenarios where the applicability of LLMs is limited by constraints of various kinds:

- *Scalability:* as highlighted in a recent survey [9], even state-of-the-art APR techniques based on LLMs are often only demonstrated on fixing bugs with so-called perfect fault localization. This means that the location where to apply a patch is given as input to the APR system. In contrast, applying APR "in the wild" would require end-to-end automation—from detecting a bug to devising a fix for it. There has been plenty of progress to support larger context windows or to summarize a project's codebase in a way that it amenable to end-to-end analysis; however, the state of the art indicates that there still is a gap between research prototypes and industrial-strength tools when it comes to APR technology [9].

- *Cost & performance:* despite significant improvements in open-source and "small" models, there remains a major performance gap between the LLMs that one can run on local hardware (i.e., your laptop) and frontier models, which are only accessible in the (commercial) cloud. Consider, for example, the latest generation of Qwen models [10]: on the CodeForces coding

[1] For example, even when focusing on techniques evaluated on the customary Defects4J, "many systems evaluate on different subsets of bugs and mix *pass@k* with accuracy-style metrics" [9].

benchmark [11], the score difference between the best performing *Qwen3-235B-A22B/thinking* and the simpler *Qwen3-4B/non-thinking* (which can be run on a well-equipped laptop) is 64.5 percentage points (98.2% vs. 33.7%). More specifically for APR, the RepairBench benchmark [12] indicates that the gap between the best model (`gpt-o4-mini`) and the most affordable one (`mistral-small-2503`)[2] is 30 percentage points (50.3% vs. 20.4%).[3] Thus, state-of-the-art performance requires access to frontier models, which incur significant costs for regular usage [13].

	TECHNIQUE	AVG. $ PER FIXED
LLM-based	ChatRepair [14]	0.42
	RepairAgent [15]	0.14
	RepairBench [12]	0.20–0.50
traditional	Jaid [16]	0.07
	Restore [17]	0.04

Table 1: A comparison of the costs to fix a bug with different APR technologies.

To better gauge the difference in actual cost between different kinds of techniques, Table 1 summarizes the cost per correctly fixed bug reported in the experimental evaluations of different tools. ChatRepair [14] and RepairAgent [15] are state-of-the-art LLM-based APR approaches, whereas RepairBench [12] is a leaderboard that compares frontier LLM models applied to APR. While each publication may compute the costs in a slightly different way (and LLM costs keep on changing), it is clear that the cost per fixed bug hovers in the $0.1–$0.5 range. Jaid [16] and Restore [17] are two traditional APR techniques we developed in previous work; their cost per fixed bug is an order of magnitude less than the LLM-based tools: $0.04–$0.07. Finally, let's consider PrevaRank. We call it a "lightweight" tool because its running costs are negligible compared to the actual costs to produce plausible fixes. As we discuss in Section 3, ranking n fixes takes on average $n \times 0.03$ seconds; using the same cloud-computing costs applied to Jaid's and Restore's experiments, this translates to a cost of $\$10^{-7}$ per ranked fix. Concretely, this means that PrevaRank's overhead is simply negligible.

- *Security & privacy:* even when the costs of accessing frontier models are not a problem, security and privacy concerns may prevent using LLMs hosted by a third-party organization. In addition to the security issues inherent in using cloud computing services, LLMs present specific risks for data privacy [18, 19, 20]. Given that there are limits to the models that

[2]Note that this "small" model has 24 billion parameters, which means that it still is too big to be run on standard hardware with good performance.

[3]These numbers are from the 2025-06-11 snapshot of RepairBench's leaderboard at `https://repairbench.github.io/` (the latest update at the time of writing).

can be run on private hardware (see previous point), the tension between security and performance can complicate the adoption of APR techniques based on LLMs.

- *Customization:* frontier models are commonly showcased on benchmarks using so-called high-resource languages: widely popular languages like Python, JavaScript, and Java, which feature prominently in the LLMs' vast training data. In contrast, researchers have shown that the capabilities of even the best models degrade significantly on low-resource languages [21]; for example, the *pass*@1 score [22] on code generation tasks decreases, on average, by 38.6 percentage points when going from Python (a high-resource language) to Racket (a low-resource language, which still has a considerable user base).[4] These data imply that APR techniques based on LLMs may be difficult or impractical to customize to work reliably on languages that are not as popular as Python or Java. This, in turn, highlights other scenarios where traditional APR techniques may be preferable because they are language-agnostic and offer more flexibility.

In summary, while researchers and practitioners are busy at work trying to overcome some of the aforementioned limitations, traditional APR techniques still have something to offer in certain application domains.

1.2. Positioning

PrevaRank is technique-agnostic, in that it can be applied to rank the plausible patches produced by any APR technique, regardless of how the technique works. The experimental evaluation that we present in Section 3 mainly applies PrevaRank to patches generated by traditional APR tools that are not based on LLMs. This is simply because most recent LLM-based APR techniques follow an iterative generation process, which rarely produces more than one plausible fix per bug; under these conditions, ranking is immaterial. This is not an intrinsic limitation of PrevaRank, as it remains applicable to rank any kind of patches regardless of how they were generated; in fact, Section 3.3 discusses a small experiment where PrevaRank ranked the patches produced by a zero-shot prompted LLM.

It is clear that LLM-based techniques are lionized in recent APR research, since they allow researchers to develop novel approaches that leverage the capabilities of frontier generative AI models. At the same time, traditional APR methods—based on structured search and template-based techniques—still work well for certain bugs that can be fixed with small, local syntactic changes (e.g., adding null checks or adjusting a conditional) [23]. Therefore, such traditional methods remain useful as baselines and in highly constrained systems [8]; in other words, they work well as specialized tools. As for LLM-based APR methods, while they are often general purpose, it has been recently shown

[4] `https://www.tiobe.com/tiobe-index/`

that the capabilities of different agentic systems for program repair are often complementary, as "no single agent consistently outperforms others" [24]. A similar complementarity has been observed in various test-driven LLM-based APR approaches [25]. Finally, Section 1.1 presented several scenarios where techniques based on state-of-the-art LLMs are impractical because they are too expensive or fail to satisfy safety and privacy constraints.

These considerations suggest that the landscape of APR techniques remains varied and populated by complementary solutions that are applicable in different conditions. As a result, PrevaRank's approach remains useful too: *i*) it provides a practical way to improve the accuracy of APR systems with a lightweight, inexpensive technique that is applicable across the board; and *ii*) it supports a simple way of integrating multiple APR approaches to leverage their complementary fixing capabilities.

1.3. Contributions

In summary, this paper makes the following contributions:

i) PrevaRank: a novel technique for ranking plausible APR patches, and its implementation.

ii) An experimental evaluation of PrevaRank ranking plausible patches produced by state-of-the-art APR tools for Java, suggesting that PrevaRank is often effective, and compares favorably to other APR patch ranking techniques.

iii) The implementation of PrevaRank and all details of the experimental evaluation, available as a replication package (see Section 6).

1.4. Discussion of Implications

Let's wrap up the introduction with a brief discussion of the implications of the research results obtained described in the rest of the paper.

The main selling point of PrevaRank for *practitioners* is that it is a very affordable technique, which runs on basic hardware and is applicable to rank plausible patches regardless of the technique used to generate them. Whether PrevaRank's effectiveness—how much it successfully improves the ranking of correct fixes—is sufficient in practice depends, of course, on the specific requirements of the concrete application scenarios. It remains that APR is not a solved problem [9], and the state of the art in this field often advances in dribs and drabs; hence, PrevaRank's capabilities are a valuable, if incremental, result.

For *researchers*, PrevaRank's work indicates that exploring APR techniques that are not based (primarily or exclusively) on LLMs remains an interesting research problem, which may lead to complementing the state of the art, and cater to the needs of users with different requirements (see the constraints discussed in Section 1.1). Another suggestion is that allowing an APR tool to explicitly output multiple plausible patches for the same bug may suggest novel opportunities of combining repair with ranking techniques, which may open up new paths to research progress in automated program repair.

2. How PrevaRank Works

Background: How APR Works. Automated program repair (APR) encompasses a wide variety of techniques aimed at producing, completely automatically, source-code fixes for programming errors. The input of an APR tool usually includes the source code of a buggy program, as well as a test suite T that is used as a (partial) specification of the expected correct program behavior; thus, T should include at least one failing test, which exposes the bug to be repaired.

The output of an APR tool consists of source-code patches that "repair" the buggy program and pass all tests in T. Such patches are called *plausible* because they have been validated (at least) against the test suite T. However, they may or may not be actually *correct*: since a test suite only checks a finite number of possible program behaviors, it is possible that a plausible patch is actually incorrect because the patched program still behaves incorrectly in scenarios not checked by any of the available tests in T.

Since complete and formal specifications of software are rarely available, the ultimate judge of correctness is the developer, who can use their intuition and knowledge of the codebase to determine whether a certain program behavior is acceptable or wrong. Correspondingly, APR tools are usually evaluated on historic bugs, using the programmer-written fix for a certain bug as the yardstick to evaluate correctness.

Another consequence of the under-specification provided by tests is that an APR tool may produce different plausible patches for the same bug. This motivates the usefulness of *ranking* plausible patches according to their likelihood of being correct.

In the following, we use the term *fix* only to denote a programmer-written repair that is considered correct. In contrast, the term *patch* denotes any program modification that is produced by an APR system; thus, a patch may or may not be correct.

2.1. An Overview of PrevaRank

In a nutshell, PrevaRank is a technique to rank plausible patches produced by automated program repair tools based on how they are similar to programmer-written fixes to historic bugs with comparable characteristics. Thus, the PrevaRank approach consists of two phases: a training phase that learns from historic bugs and fixes, and a ranking phase that uses the learned frequencies as heuristics to rank newly produced plausible patches.

Training. Fig. 1 overviews PrevaRank's training phase, whereas Algorithm 1 illustrates it with pseudo-code. In this phase, PrevaRank mines software repositories for historical data about bugs and their corresponding programmer-written fixes, in order to learn which features of fixes are more commonly associated with certain bug categories. PrevaRank classifies each bug b in a *category* $\mathsf{c}(b)$ based on how it is described by the developers that identified and fixed it in the related commit messages; examples of bug categories include

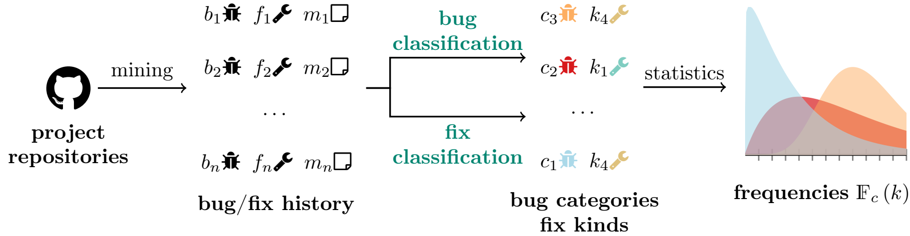

Figure 1: An overview of PrevaRank's *training* phase: by mining historic data in software repositories, PrevaRank builds distributions $\mathbb{F}_c\,(k)$ that summarize how frequently a certain patch kind k was used by developers to fix a certain bug category c.

"overflow" and "null-pointer" bugs. PrevaRank also classifies each programmer-written fix f in a *kind* $\mathsf{k}(f)$ based on its syntactic features, that is how it modifies the buggy source code. Examples of features combined to define fix kinds include "adding a conditional", "changing the return type", and "initializing a local variable". We curate bug categories and fix kinds so that they are mutually exclusive: any bug belongs to exactly one category, and any fix belongs to exactly one kind (a combination of several syntactic features). PrevaRank summarizes this information by reporting *frequencies* $\mathbb{F}_c\,(k)$: the fraction of all bugs of category c in the training set whose programmer-written fix is of kind k.

Ranking. Fig. 2 overviews PrevaRank's ranking phase. In this phase, PrevaRank uses the frequency information collected in the training phase as heuristics to rank a set P of plausible patches produced by an automated program repair tool for some bug b. First, PrevaRank classifies each patch $p \in P$ in a *kind* $\mathsf{k}(p)$; the classification of plausible patches is exactly as the classification of programmer-written fixes, since it is based entirely on the patch's syntactic features. Then, PrevaRank *estimates* b's bug category as the category $\widehat{\mathsf{c}}(b)$ for which patches of the same kinds as those in P are more frequent. Since the classification of bugs into categories uses information that is generally only available *after* a bug has been fixed, Eq. (2) is an educated guess; however, our experiments will show that this heuristics is often reliable. Finally, PrevaRank ranks each patch $p \in P$ by decreasing values of $\mathbb{F}_{\widehat{\mathsf{c}}(b)}\,(\mathsf{k}(p))$. Intuitively, the more frequent a fix of that kind was in the historical data for the estimated bug category, the higher it is ranked over the other patches.

The rest of this section describes the main steps of PrevaRank's approach in detail, and discusses other aspects of its current implementation.

2.2. Training Data

PrevaRank's training phase mines historical data about bugs and their programmer-written fixes in GitHub repositories. For each pair c^-, c^+ of consecutive commits in a project repository's main branch, PrevaRank considers the triple $\langle b, f, m\rangle$ where: *i)* b is the source code in commit c^-; *ii)* f is the "diff" of c^+ over c^-; *iii)* m is c^+'s commit message.

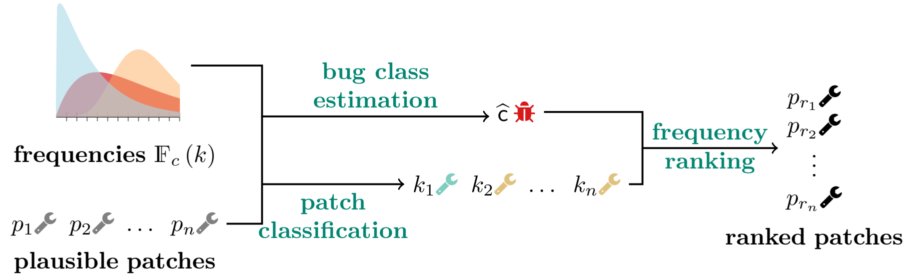

Figure 2: An overview of PrevaRank's *ranking* phase: PrevaRank determines the *kind* of each plausible patch (generated by a program repair tool), using the same heuristics used in the training phase; based on the frequency statistics collected in the training phase, it also *estimates* the *category* $\overline{c}$ of the bug these patches are fixing; finally, it ranks each plausible patch of kind k as per the frequency distribution $\mathbb{F}_{\overline{c}}(k)$.

PrevaRank determines whether to further consider a triple $\langle b, f, m\rangle$ according to some heuristics. First, it discards the triple if f involves (adds, removes, or modifies) more than five lines of code: since generating complex multi-line repairs is generally outside the current capabilities of APR, including complex patches would make our training data less representative of the bugs and fixes that we want to be able to rank. Second, PrevaRank only retains the triple if its commit message m includes keywords that suggest that it is a bug-fixing commit (as opposed to a commit that adds or changes functionality, or that affects only tests or documentation).

2.3. Bug Classification

Each bug $b = \langle b, f, m\rangle$ in the training data is labeled with a *category* $\mathsf{c}(b)$. To this end, we introduced a classification based on a small number of well-defined bug categories: too many fuzzy bug categories would dilute the "signal" in the training data, and would be much harder to heuristically match in the ranking phase (see Section 2.6). Based on some exploratory analysis—as well as on common bug categories used in curated collections such as Defects4J [26, 5]—we decided to focus on three widely applicable, well-known, and distinct bug categories: *overflow* bugs, *null-pointer* dereference bugs, and *logic* bugs (i.e., bugs that do not crash the program but involve the violation of a logic condition, such as an assertion failure). Overall, these three categories cover a diverse range of different bugs that are within the purview of APR's state of the art; this helps make PrevaRank applicable in combination with several different APR tools—which we'll demonstrate in the experiments.

For the whole ranking process to work, the training data has to be categorized accurately (i.e., no misclassifications); otherwise, the training phase would end up with spurious associations between bugs and patches. To this end, we categorized each bug in a semi-automated way: first, we used a keyword-based search to assign a *tentative* category to each bug automatically; then, we manually reviewed and validated the keyword-based classification. The first, automated

```
LOGIC_BUGS_PATTERN = ".*\b(logic|logical)\b.*"
    + ".*\bfix|bug|issue|wrong|error|fault|assert|correct|condition|unexpected|\b(
     incorrect|wrong)\s+(function|output|result)\b.*|"
NULL_BUGS_PATTERN = "(?i)((null\s*pointer|NPE|NullPointerException))"
OVERFLOW_BUGS_PATTERN = "(?i)(.*\b(buffer|array)\b.*\b(overflow)\b.*|"
    + ".*\bout\sof\s(bounds|range|limit)\s.*|"
    + ".*\b(heap|stack|integer)\s(overflow)\b.*|"
    + "(Array|String|Index)OutOfBoundsException.*|"
    + "\b(BufferOverflowException\b.*)\n)"
```

Listing 1: Regex patterns used to identify bug *categories*

step speeds up the second, manual step and makes the classification effort reasonable. Furthermore, note that this classification need only be done once to prepare the training data for PrevaRank; using PrevaRank to rank the patches produced by an APR tool remains a fully automated process. The rest of the section details the classification process.

Keyword-based search. For each category, we defined a number of keywords that may indicate a bug of that category; For example "buffer" and "overflow" may indicate overflow bugs, whereas "NPE", "null", and "pointer" may indicate null-dereference bugs; Listing 1 details the regexes we used in our experiments. If the commit message m of a bug b matches the regex associated with category c, then we tentatively assign category c to bug b. If more than one regex matches, the manual validation step will determine if any of the corresponding categories is correct, or if it is preferable to omit the bug from the training set to avoid ambiguity. In our experiments, we found 167 cases of bug commit messages matching more than one regex, which we excluded from the training set.

Manual validation. The keyword-based search applies simple heuristics that may misclassify a bug; we only used as a starting point for a systematic manual validation of the classification. To this end, two authors independently inspected each bug b, together with its patch f, commit message m and any other information related to each data point with a keyword-based match, in order to ascertain whether the automatic classification was correct. If both authors accept the automatic classification, the data point $\langle b, f, m\rangle$ is retained and the bug b is classified of category $\mathsf{c}(b)$; otherwise, the data point is removed from the training data. Henceforth, "training data" refers to the culled training data where every bug has validated category.

2.4. Patch Classification

PrevaRank assigns a kind $\mathsf{k}(f)$ to the fix f in each triple $\langle b, f, m\rangle$ in the training data. Informally, a patch kind represents one class of syntactic modifications that are present in f.

To come up with a general selection of patch kinds, we first considered common *syntactic features* that may be involved in Java patches. Table 2's leftmost

FEATURE	ADD	REMOVE	MODIFY
assignment	assignment	assignment	target, expression
conditional	conditional, else	conditional, else	condition (strengthen, weaken, other)
loop	loop, **break**, **continue**	loop, **break**, **continue**	condition (strengthen, weaken, other)
method call	call	call	callee, call arguments
return	**return**	**return**	returned value
try/catch	**catch**, **finally**	**catch**, **finally**	exception type
synchronized block	block	block	lock object, block
field declaration	field	field	signature, initialization
method declaration	method	method	signature
assert	assertion	assertion	predicate (strengthen, weaken, other)

Table 2: The main feature/modification combinations that PrevaRank uses to classify patches. A FEATURE is a syntactic feature of Java code; a modification ADDS a certain feature, REMOVES it, or MODIFIES some of its components.

```
if (buffered.getBuffer() != null && buffered.getBuffer().length > 65536)
```

Listing 2: An example of programmer-written fix classified as "*conditional: modify condition (strengthen)*". The diff is shown in blue and underlined.

column lists the main ones we considered for this work, including assignments, conditionals (**if**), loops (**for**, **while**), as well as class-level declarations. For each syntactic feature, we considered its main components, and how a patch that adds, removes, or modifies code may affect them. Table 2's other columns list several such *combinations*, such as "ADD assignment" (the patch introduces a new assignment), "REMOVE **catch**" (the patch removes a **catch** block from an existing **try**/**catch**), and "MODIFY method signature" (the patch changes the types or the visibility of an existing method). This process identified 67 such combinations of feature/modification; for brevity, Table 2 lists only a representative subset. The list we compiled is consistent with empirical studies of the most common bug-fix patterns that are found in open-source Java software [27, 28, 29, 30].[5] Listing 2 shows an example of patch from project Apache Tomcat[6] that targets a *conditional* by *modifying* (precisely, *strengthening*) its *condition*.

A patch *kind* is any element of the power set $K = 2^M$, where M is the aforementioned set of all feature/modification combinations. Since a given patch may involve more than one modification, this definition of K ensures that each fix f belongs to a *unique* kind $\mathsf{k}(f)$. Combining features into kinds is key to having a sufficiently fine-grained, and thus discriminating, classification of patches.

[5] For example, the five most common categories of bug-fix patterns in Pan et al.'s manual analysis [30] correspond to "modify condition in conditional" (IF-CC in [30]), "modify call arguments in method call" (MC-DAP and MC-DNP), "modify expression in assignment" (AS-CE), and "add conditional" (IF-APC) in Table 2's classification.

[6] https://github.com/apache/tomcat/commit/e886ee20183b5f5d2a902843b6d324b58d991ea8

Algorithm 1: PrevaRank's training phase.

```
   Input: An ordered sequence of consecutive commits c_1, c_2, ..., c_m
   Output: A collection of frequencies 𝔽: C, K → [0, 1]
   // # commit pairs of bug category c ∈ C and fix kind k ∈ K
 1 N: C, K → ℕ ← 0        // initialized to all zeros
 2 foreach x ∈ {1, ..., m − 1} do
 3 |   c⁻, c⁺ ← c_x, c_{x+1}        // pair of consecutive commits
 4 |   b, f, m ← code(c⁻), diff(c⁻, c⁺), message(c⁺)
 5 |   if size(f) ≤ 5 ∧ is_bug_fix(m) then
 6 |   |   c ← bug_category(m)       // bug classification
   |   |   // human validation of bug category
 7 |   |   if valid(b, f, m, c) then
 8 |   |   |   k ← patch_kind(f)       // patch classification
 9 |   |   |   N(c, k) ← N(c, k) + 1   // increment count
10 |   |   end
11 |   end
12 end
   // Compute frequencies for each c ∈ C and k ∈ K
13 foreach c ∈ C do
14 |   foreach k ∈ K do
15 |   |   𝔽(c, k) ← N(c, k) / Σ_{k∈K} N(c, k)
16 |   end
17 end
18 return 𝔽
```

2.5. Historic Frequencies

As described in the previous sections, PrevaRank assigns a bug category $\mathsf{c}(b)$ and a patch kind $\mathsf{k}(f)$ to the bug b and fix f in each triple $\langle b, f, m\rangle$ in the training data T. Based on this classification, PrevaRank summarizes the associations between bugs and fixes by means of a historic *frequency* $\mathbb{F}_c \colon K \to [0, 1]$ for each bug category $c \in C$. Given a unique fix kind $k \in K$ and bug category $c \in C$, $\mathbb{F}_c(k)$ is the fraction of all triples in the training data where a bug of category c was fixed with a patch of kind k:

$$\mathbb{F}_c(k) \;=\; \frac{|\{\langle b, f, m\rangle \in T \mid \mathsf{c}(b) = c, \mathsf{k}(f) = k\}|}{|\{\langle b, f, m\rangle \in T \mid \mathsf{c}(b) = c\}|} \tag{1}$$

You can think of $\mathbb{F}_c(k)$ as a normalized frequency distribution that captures how frequently a fix of kind $k \in K$ was used historically to repair a bug of category $c \in C$. These frequencies are the overall output of PrevaRank's training phase.

2.6. Bug Category Estimation

In the ranking phase, PrevaRank inputs the frequencies $\mathbb{F}_c$ produced by the training phase, and a collection P of patches produced by an APR system for a certain bug b. PrevaRank assigns a kind $\mathsf{k}(p)$ to each patch $p \in P$ according to the same algorithm used in the training phase to classify fixes

(Section 2.4). However, it cannot classify the *bug* under repair in the same way it classified bugs in the training phase: bug classification (Section 2.3) relies on the commit message that accompanies a programmer-written fix—as well as, possibly, on other information. In contrast, the bug category should be *estimated automatically* in the ranking phase, which works on automatically generated patches whose correctness and intent are unknown in general.

PrevaRank estimates b's bug category $\widehat{\mathsf{c}}(b)$ as the category that was most frequently associated with the kinds of patches P in the training data:

$$\widehat{\mathsf{c}}(b) \;=\; \operatorname*{argmax}_{c \in C} \sum_{p \in P} \mathbb{F}_c\left(\mathsf{k}(p)\right) \tag{2}$$

Since we only have to distinguish between a small number of bug categories, this simple heuristics is generally robust and leads to effective results—as demonstrated in Section 3.

Tie-breaking. If two or more categories obtain the same score in Eq. (2), we pick the category with the larger number of training instances $|T_c|$; if still tied, we choose the one with larger $\max_{p \in P} F_c(k(p))$; remaining ties are resolved by a fixed (lexicographic) category order to keep the procedure deterministic.

2.7. Patch Ranking

In the final step of the ranking phase, PrevaRank ranks each patch $p \in P$ among those produced by an APR system for a bug b according to the value $\mathbb{F}_{\widehat{\mathsf{c}}(b)}\left(\mathsf{k}(p)\right)$: the higher this value is for p relative to the other patches in $P \setminus \{p\}$, the higher p is ranked by PrevaRank.

An important detail is how to handle *ties* in the ranking. When several patches all have the same value of $\mathbb{F}_{\widehat{\mathsf{c}}(b)}\left(\mathsf{k}(p)\right)$, then PrevaRank sorts them in the same relative order as the APR system's original ranking. For example, if four patches have score $p_3 = 0.7, p_1 = p_4 = 0.5, p_2 = 0.3$, and the original APR system ranks them in the higher-to-lower order p_4, p_3, p_2, p_1, PrevaRank ranks them as p_3, p_4, p_1, p_2. Note that ties are not very frequent, thanks to the fine-grained classification into patch kinds.

3. Experimental Evaluation

PrevaRank's experimental evaluation addresses the following research questions:

RQ1 How *effective* is PrevaRank's ranking of patches?
This RQ investigates how frequently PrevaRank ranks *correct* patches higher than plausible but incorrect ones.

RQ2 How does PrevaRank *compare* to other state-of-the-art patch ranking techniques?
This RQ compares PrevaRank to other approaches to patch ranking, focusing on effectiveness and complementarity.

	SIZE (kLOC)	COMMITS	STARS	FIXES
mean	2 303	20 567	9 725	81
max	22 500	103 949	68 900	578
min	34	2 076	46	2
total	186 533	1 665 958	787 737	6 583

(a) Characteristics of the 81 Java open-source projects used as training data. The table reports the average (mean), maximum, minimum, and total value of the projects' SIZE (in thousands of lines of code), number of COMMITS (on the main branch), number of GitHub STARS, and number of bug FIXES that we included in our training data.

PROJECT	VERSION	MLOC	COMMITS	KSTARS	% JAVA
spring-boot	v3.1.2	0.80	44 563	69	98%
elasticsearch	8.9.1	4.60	71 418	65	100%
spring	v6.0.11	1.50	27 652	53	98%
guava	32.1.2	1.00	6 155	48	100%
RxJava	3.1.6	0.48	6 042	47	100%
retrofit	2.9.0	0.04	2 076	42	96%
dubbo	3.2.5	0.45	7 008	39	99%
dbeaver	23.1.4	0.89	24 589	33	100%
afa	3.5.1	1.20	11 544	26	77%
flink	1.17.1	3.70	33 859	22	86%

(b) The ten largest GitHub projects used by PrevaRank's training phase.The table reports the analyzed project VERSION, the project size in millions of lines of code (MLOC), the number of COMMITS, the thousands of GitHub stars (KSTARS), percentage of source code that is Java (% JAVA), and the number of bug FIXES that we included in our training data.

Table 3: Subjects used by PrevaRank's training phase.

RQ3 How *robust* is PrevaRank's classification of bugs and patches?
This RQ investigates whether PrevaRank's approach is *robust*, that is how the amount and variety of historical data that is available for training affects PrevaRank's ranking performance.

RQ4 Can PrevaRank's manual training steps be fully *automated*?
This RQ investigates whether it is feasible to use LLMs (Large Language Models) to automate the bug identification and classification steps of PrevaRank's training.

The rest of this section describes the experimental setup and the answers to these research questions.

Implementation and performance. We implemented PrevaRank in Java, comprising 5.3k LOC. PrevaRank uses GumTree [31] to analyze patch features, JavaParser [32] to parse Java code, and JGit[7] to mine GitHub repositories.

Since it is a lightweight, scalable tool, we did not need to analyze in detail the running-time performance of PrevaRank. In particular, PrevaRank's running

[7] `https://github.com/eclipse-jgit/jgit`

time is roughly proportional to the number of plausible patches it has to rank: on average, PrevaRank takes 30 milliseconds per patch (median: 28 ms, max: 41 ms, standard deviation: 7.5 ms). As we mentioned in Section 1, PrevaRank's scalability and modest computational cost are key in making it a complementary tool to approaches based on generative AI, which are usually heavyweight and only accessible on specialized hardware/cloud platforms.

3.1. Training Phase

Projects. We extracted the training data by mining the commit histories of 81 popular open-source projects hosted by GitHub. We randomly selected these projects among those with at least 40 stars, 30 thousand lines of code, and 2 thousand commits. Table 3a summarizes the main characteristics of these 81 projects; Table 3b lists the ten largest projects in our selection.

History mining. We mined the commit histories of the selected 81 projects, in order to extract suitable training data to be used as described in Section 3.1. We started from all 1 665 958 commits in our projects. We discarded all commits that add, remove, or modify more than five lines of code; 224 965 commits satisfy this criterion. Three considerations guide the decision to restrict patches to five lines of code. First, empirical studies show that real-world bug fixes are typically small: the median patch size in DefectsJS is four lines [33], and large-scale analyses of production Java projects report that corrective patches rarely exceed 20 lines [34, 35]. Second, existing APR tools, including recent LLM-guided tools, generally produce patches of only a few lines. Third, our patch-kind classifier infers syntactic intent (e.g., null-check addition, condition strengthening). For larger multi-line patches that introduce complex logic or method bodies, such inference becomes unreliable without deeper program context. For these reasons, we focus on small patches, which are both representative of real bug fixes and compatible with the abstraction capabilities of our classifier.

Next, we discarded all 217 181 commits whose commit message does *not* match any of Listing 1's regular expressions. As described in Section 2.3, we used the regexes as *necessary* conditions, followed by a manual inspection that all regular expression matches are indeed bug-fixing commits; precisely, two authors carried out the manual inspection independently, and we selected a commit, out of the remaining 7 784 ($= 224965 - 217181$), only if both agree that it is a genuine bug-fixing commit, resolving a bug of the category given by the regex (i.e., overflow, null pointer, or logical). This conservative process, which took around two full days of work, gives us a good confidence that the selected 6 583 commits do indeed represent bug fixes of a certain category (1 093 logical, 5 161 null pointer, 329 overflow) in a project's history.

Patch classification. Based on a systematic analysis of the selected 6 583 bug-fixing commits, we identified 67 individual constructs; Table 2 lists the most common ones. Since a patch may combine any number of these constructs, there is an astronomically large number of $2^{67} \simeq 10^{20}$ possible unique patch kinds; in practice, though, we only found a much smaller number of 229 patch

Table 4: Ten frequently occurring patch kinds in our training data. For each patch kind, the table reports the percentage of bugs of each category that were fixed by a patch of that kind.

PATCH KIND	% BUG OF CATEGORY logical	null pointer	overflow
add conditional	5	13	6
modify method arguments	7	4	7
add **null** check	1	16	3
modify condition (strengthen)	7	3	14
add field initialization + *modify* method signature	6	2	2
modify condition (other)	3	1	2
add field initialization + *modify* method call arguments	2	1	2
add method call + *add* method signature	8	5	8
modify assignment expression	2	2	3
add conditional + *modify* block scope	1	7	1

	TOOLS	AV?	> 1?	NO PFL?	≥ 10?
analysis	SimFix [38], kPAR [39], (jGenProg, jKali, jMutRepair) [40], DeepRepair [41]	✔	✔	✔	✖
analysis	ARJA [1], Cardumen [42], Jaid [3], RSRepair [43], TBar [44]	✔	✔	✔	✔
learning	Gamma [45], Circle [46], Cure [47], SEQUENCER [48]	✔	✔	✖	
learning	Dlfix [49], KNOD [50], Repilot [51]	✔	✖		
learning	AlphaRepair [52], CoCoNuT [53], TRANSFER [54]	✖			
learning	Recoder [55], Tenure [56]	✔	✖		
learning	ITER [57], RapidCapr [58], RewardRepair [59]	✔	✔	✔	✔

Table 5: Automated program repair tool selection. For every group of APR TOOLS, whether their experimental artifacts (patches) are available for Defects4J bugs (column AV?), whether they include more than one plausible patch per bug (> 1?), whether they did *not* use perfect fault localization (NO PFL?), and whether they include at least 5 plausible patches for at least ten fixed Defects4J bugs (≥ 10?).

kinds. This indicates that "real" patches usually only involve a small number of constructs, and combine them by predictable patterns—which is consistent with the generally observed "naturalness" of software [36, 37]. Table 4 shows the ten most frequently observed patch kinds, and the corresponding bugs they fix.

Historic frequencies. The last step of PrevaRank's training phase is the computation of historic frequencies $\mathbb{F}_c(k)$ for each bug category c and fix kind k. This step is computationally trivial, as it only requires to compute frequencies in a dataset. In our experiments, PrevaRank took less than 1 minute to compute the frequencies for all selected 6 583 bug-fixing commits.

3.2. Subjects: APR Tools and Patches

The overall goal of this evaluation is assessing whether PrevaRank can *improve* the ranking of correct patches (over merely plausible ones) compared to that produced by the APR tools that produced the patches in the first place. In order to also demonstrate flexibility, the evaluation should include as many different APR tools—and as many different patches—as possible.

	Chart			*Closure*			*Lang*			*Math*			*Mockito*			*Time*			*all bugs*		
TOOL	F	P	B	F	P	B	F	P	B	F	P	B	F	P	B	F	P	B	F	P	B
ARJA	4	23	6				7	38	5	13	90	7				3	24	8	27	175	6
Cardumen	6	30	5	3	15	5	5	25	5	24	120	5				3	15	5	41	205	5
ITER	2	62	31	10	130	13	3	30	10	14	137	10				1	24	24	30	383	13
Jaid	7	1918	274	19	2167	114	16	1347	84	20	2803	140	2	23	12	2	66	33	66	8324	126
RSRepair	4	29	7				5	37	7	12	199	17				2	30	15	23	295	13
RapidCapr	7	348	50	24	625	26	13	383	29	23	721	31	5	130	26	3	64	21	75	2271	30
RewardRepair	2	377	188	23	4588	199	11	2200	200	10	1999	200	3	600	200	2	400	200	51	10164	199
T-BAR	4	112	28	8	338	42	7	150	21	12	580	48				2	35	18	33	1215	37
all tools	15	2899	193	50	7863	157	32	4210	132	54	6649	123	9	753	84	8	658	82	168	23032	137

Table 6: Summary of the patches used in PrevaRank's experiments. For each APR TOOL, for each Defects4J project identifier (*Chart*, *Closure*, ...), the table reports: the number F of faults of that project correctly fixed by the tool; the total number P of plausible patches produced by the tool for those faults; the average number B of patches produced by the tool for each of those faults (i.e., B = P/F). The bottom row combines data from *all tools*; the rightmost column combines data from all Defects4J projects.

APR tool selection. To this end, we initially considered the 26 APR tools listed in Table 5: this list includes implementations of traditional approaches (e.g., jGenProg, kPAR), more advanced approaches combining different dynamic analysis techniques (e.g., Jaid, TBar), as well as numerous recent approaches that use some form of machine learning (e.g., AlphaRepair, ITER)—which have grown in popularity in recent years.

We culled this initial list by dropping any APR tool that does not satisfy all these criteria: *i*) The tool has been evaluated on the Defects4J benchmark (v. 1.2.0 or later [26, 5]), and its experimental artifacts (in particular, the patches generated in the experiments) are publicly available. This criterion is obviously needed to ensure that we can feed the tool's patches to PrevaRank. *ii*) The tool's patches include multiple patches for the same bug. If a tool produces at most one patch per bug, there is nothing to rank, and hence PrevaRank's capabilities are irrelevant. Importantly, even if an APR tool could, in principle, produce multiple patches per bug, we do not consider it if it has not been evaluated in this scenario in the tool's paper's evaluation; in other words, we only consider the tool's performance in the experimental conditions set by its authors.[8]As we discuss in Section 4, this criterion also excludes most recent LLM-based APR approaches, which typically employ an iterative patch generation process, producing only a very small number of candidates per iteration. *iii*) The tool does *not* use perfect fault localization (i.e., it only produces patches that modify the location of the programmer-written fix, which is given as input). Again, even if a tool *could* take a different fault localization input, we stick to the way it has been evaluated by its authors. The reason for avoiding patches generated with

[8]For example: AlphaRepair does not include patches but its implementation is available. We ran it (settings: Ochai fault localization, top-40 most suspicious locations, beam setting 5) on Defects4J bugs, but it generated more than 4 plausible patches for only 3 bugs. This does not mean that it is not capable of producing more patches with different settings; however, we stick to the experimental artifacts provided by its authors, which are more likely to demonstrate an optimal usage of the tool.

perfect fault localization is that, even if there are multiple patches per bug, they all tend to be very similarly syntactically, since they all originate from the same key information about the bug location; hence, they have little discriminatory power for a feature-based approach like PrevaRank's (which, in fact, would tend to merely replicate the same ranking as the original tool). *iv*) Finally, the APR tool's available patches should consist of at least 5 patches for each of at least 10 Defects4J bugs that it can fix correctly. This criterion is simply to ensure that we have a nontrivial collection of subjects to experiment with PrevaRank used on a certain APR tool's output. Table 5 details which tools satisfy each of these criteria. We ended up with a selection of 8 APR tools (ARJA, Cardumen, ITER, Jaid, RapidCapr, RewardRepair, RSRepair, and TBar) that satisfy all criteria.

Patch selection. From the experimental artifacts of the 8 selected tools, we collected all available plausible patches for Defects4J bugs such that the tool produces at least one correct fix and at least four plausible patches. If a tool only produces plausible, incorrect patches for a bug, their ranking is immaterial. Also, if a tool produces fewer than four plausible patches for a bug, even the worst ranking will put a correct fix in the top-3; hence, we exclude these bugs from our evaluation as they would not be relevant to assess effectiveness. We use the widely accepted definition of correctness: a patch is a correct fix if it is semantically equivalent to the programmer-written fix for the same bug (which is available in Defects4J). Table 6 gives some details about the selected bugs and patches. Overall, the 8 APR tools included correct fixes (and at least four plausible patches) for 168 bugs (123 logic, 34 null pointer, 11 overflow), and 23 032 plausible patches (15 601 logic, 5 272 null pointer, 2 159 overflow). This selection of patches is quite varied, as it includes patches of bugs from 6 different Defects4J projects, and tools that produce a range of different plausible patches (from Cardumen's 5 plausible patches per bug, up to RewardRepair's whopping 199 per bug).

3.3. RQ1: Effectiveness

We conducted two series of experiments to evaluate PrevaRank's effectiveness. In the first series, each run of PrevaRank ranks the plausible patches $P(t, b)$ produced by APR tool t for bug b. In the second series, each run of PrevaRank ranks the plausible patches $P(b)$ produced by any APR tools that could fix bug b. In other words, in the first series each tool is considered *individually*, whereas in the second series patches from different tools are ranked together.

Individual tool patch ranking. Each point in Figure 3's scatterplot represents one of the 168 bugs used in the experiments; a point at coordinates (x, y) denotes that the APR tool ranked at position x the (first) correct fix for the corresponding bug, whereas PrevaRank ranked it at position y. Visually, it's clear that the majority of points are below the diagonal $x = y$ line (i.e., $y < x$), which means that PrevaRank usually improves (lower is better) the ranking of correct fixes. On the other hand, these points do not seem to follow any distinct pattern per

			TOP 1								TOP 3								TOP 5								TOP 10							
			#				%				#				%				#				%				#				%			
TOOL	K	BUGS	N	B	O	P	N	B	O	P	N	B	O	P	N	B	O	P	N	B	O	P	N	B	O	P	N	B	O	P	N	B	O	P
ARJA	*	27	16	6	1	4	59%	22%	4%	15%	5	16	0	6	19%	59%	0%	22%	1	24	0	2	4%	89%	0%	7%	0	27	0	0	0%	100%	0%	0%
Cardumen	*	41	18	9	4	10	44%	22%	10%	24%	1	28	0	12	2%	68%	0%	29%	0	41	0	0	0%	100%	0%	0%	0	41	0	0	0%	100%	0%	0%
ITER	*	30	17	6	2	5	57%	20%	7%	17%	3	15	1	11	10%	50%	3%	37%	0	23	1	6	0%	77%	3%	20%	0	26	0	4	0%	87%	0%	13%
Jaid	*	66	38	16	5	7	58%	24%	8%	11%	22	30	0	14	33%	45%	0%	21%	12	35	2	17	18%	53%	3%	26%	6	45	1	14	9%	68%	2%	21%
RapidCapr	*	75	52	14	4	5	69%	19%	5%	7%	25	31	3	16	33%	41%	4%	21%	18	39	3	15	24%	52%	4%	20%	9	58	0	8	12%	77%	0%	11%
RewardRepair	*	51	30	8	4	9	59%	16%	8%	18%	5	18	1	27	10%	35%	2%	53%	0	28	0	23	0%	55%	0%	45%	0	39	0	12	0%	76%	0%	24%
RSRepair	*	23	16	1	1	5	70%	4%	4%	22%	5	10	0	8	22%	43%	0%	35%	2	15	0	6	9%	65%	0%	26%	2	19	0	2	9%	83%	0%	9%
T-BAR	*	33	15	14	1	3	45%	42%	3%	9%	4	21	2	6	12%	64%	6%	18%	2	25	2	4	6%	76%	6%	12%	1	29	1	2	3%	88%	3%	6%
all tools	L	236	135	56	13	32	57%	24%	6%	14%	46	111	6	73	19%	47%	3%	31%	22	153	8	53	9%	65%	3%	22%	11	194	1	30	5%	82%	0%	13%
	N	83	49	17	6	11	59%	20%	7%	13%	17	46	0	20	20%	55%	0%	24%	7	60	0	16	8%	72%	0%	19%	6	68	0	9	7%	82%	0%	11%
	O	27	18	1	3	5	67%	4%	11%	19%	7	12	1	7	26%	44%	4%	26%	6	17	0	4	22%	63%	0%	15%	1	22	1	3	4%	81%	4%	11%
	*	346	202	74	22	48	58%	21%	6%	14%	70	169	7	100	20%	49%	2%	29%	35	230	8	73	10%	66%	2%	21%	18	284	2	42	5%	82%	1%	12%

Table 7: Comparison between PrevaRank and other APR tools. Each row reports the number of BUGS of category K for which the given APR TOOL produces at least one correct fix; * denotes bugs of any category, whereas L are logical, N are null-pointer, and O are overflow bugs. The internal columns indicate the number # and percentage % of these bugs BUGS that fall into one of four scenarios: N) neither the other tool nor PrevaRank, rank a correct fix in the TOP-n; B) both the other tool and PrevaRank rank a correct fix in the TOP-n; O) only the other tool ranks a correct fix in the TOP-n (PrevaRank does not); P) only PrevaRank ranks a correct fix in the TOP-n (the other tool does not). The bottom part of the table considers all tools together, with a breakdown by bug category (Logical, Null, and Overflow). All scenarios where PrevaRank strictly improves over the other tool are highlighted.

APR tool (denoted by their color), which suggests that PrevaRank tends to work well regardless of which APR tool produced the patches it is ranking.

Table 7 details the effectiveness of PrevaRank quantitatively: for each APR tool, it partitions the bugs fixed by that tool in 4 groups: group N includes those for which a correct fix appears within the top-k neither in the tool's original ranking nor in PrevaRank's ranking; group B includes those for which a correct fix appears within the top-k in both the tool's and PrevaRank's rankings; group O includes those for which a correct fix appears within the top-k only in the tool's original ranking; and group P includes those for which a correct fix appears within the top-k only in PrevaRank's ranking. Equivalently, group P consists of the bugs where PrevaRank *improves* the ranking of correct fixes from outside to inside the top-k; and group O consists of the bugs where PrevaRank *worsens* the ranking of correct fixes from inside to outside the top-k. As it is customary, we consider the top-1, top-3, top-5, and top-10 positions in the ranking, which reflect realistic scenarios [60, 61, 62] where a developer would be willing to inspect at most ten fixes to identify a correct one. Figure 4 displays similar data as Table 7 with an additional break-down by bug category.

Within this data, it is interesting to compare the size of groups P and O, which gives an idea PrevaRank's ranking quality compared to the original APR tool's. The first thing we notice is that the size of group P is practically always greater than the size of group O; the only exceptions are when both the tool's original ranking and PrevaRank's ranking include all correct fixes within the top-k. This indicates that, while PrevaRank's ranking is not infallible (as there are situations where the original APR tool's ranking is more accurate), it systematically provides a net improvement.

In fact, the size of group P is not just barely bigger than the size of group O: in 87% of cases, it is at least twice as big; in 23% of cases, it is 10 times larger

CATEGORY	TOOL	MIN O	MIN P	AVERAGE O	AVERAGE P	MEDIAN O	MEDIAN P	MAX O	MAX P
any bug	ARJA	1	1	3.2	2.5	3.0	2.0	10	8
	Cardumen	1	1	2.5	1.8	2.0	2.0	5	5
	ITER	1	1	5.6	2.2	3.0	2.0	30	6
	Jaid	1	1	27.0	7.0	4.0	2.0	789	119
	RSRepair	1	1	5.5	3.4	4.0	2.0	14	18
	RapidCapr	1	1	8.1	5.4	4.0	3.0	80	34
	RewardRepair	1	1	17.7	2.1	5.0	2.0	162	5
	T-BAR	1	1	6.7	4.3	3.0	1.0	106	45
	all tools	1	1	11.5	4.1	3.0	2.0	789	119

Table 8: Statistics about ranks of correct fixes comparing PrevaRank to other APR tools. For every tool (and *all tools* together), the table reports the minimum, average (mean), median, and maximum rank of the correct fix in the tool's original ranking (columns o) and in PrevaRank's ranking (columns p).

or more. This is easy to see in Figure 4, where PrevaRank's bar segments are often several times wider than the original tool's. Thus, PrevaRank brings a quantitative advantage with its capability of improving the ranking of numerous correct fixes. We also see that PrevaRank's effectiveness is largely independent of the category of bugs, APR tools, or top-k rank we consider. The few outliers usually correspond to narrow, specific scenarios. For instance, Jaid ranks in position 1 the correct fix for an overflow bug, which PrevaRank ranks in position 2; still, PrevaRank ranks in position 1 the correct fix for another overflow bug, which Jaid ranks in position 7. Another example are the correct fixes for three overflow bugs that Cardumen ranks at positions 1, 2, and 4, whereas PrevaRank ranks at positions 3, 3, and 1. In all, we are talking about small differences in rank in a few corner cases.

Another trend visible in Table 7 is that the difference between groups p and o does not grow monotonically with the rank cutoff k but peaks for top-3. This behavior is simply a saturation effect: there aren't many bugs that admit many different plausible patches.

Table 8 gives a final view on the effectiveness of PrevaRank in ranking the patches produced by each APR tool. The average (mean and median) rank of a correct fix in PrevaRank's ranking is nearly always better (i.e., lower position) than the rank in the APR tool's original ranking; PrevaRank's overall mean rank is nearly 1/3 of the original rank (4.1 vs. 11.5). It is also interesting that PrevaRank often improves the ranking in worst-case scenarios where there are a very large number of plausible, incorrect patches; the most extreme case is an overflow bug for which Jaid produces 810 plausible patches: Jaid ranks the correct fix at position 789, whereas PrevaRank massively improves its rank to position 6.

Multiple tool patch ranking. Let us now consider only the 93 Defects4J bugs (63 logical, 24 null pointer, 6 overflow) that at least two APR tools can correctly fix. Table 9 compares effectiveness in three scenarios: *i*) The best (highest) rank of

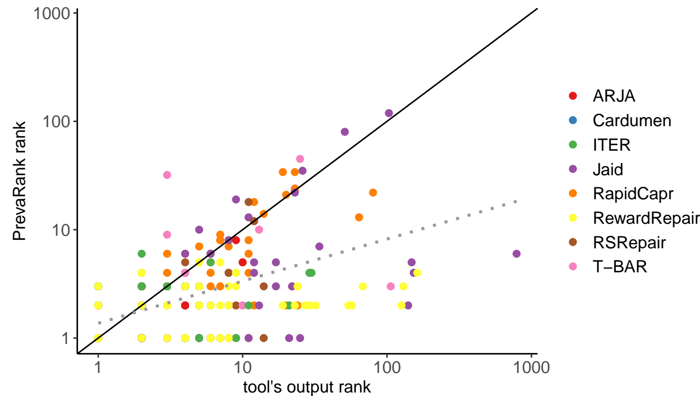

Figure 3: Each point represents a bug fixed by an APR tool. The point's x coordinate is the rank of the (first) correct fix assigned by the APR tool; the point's y coordinate is the rank assigned by PrevaRank. Thus, points below the diagonal line correspond to bugs where PrevaRank improved the APR tool's ranking of correct fixes. The dotted line is the linear regression line of the points, which highlights the data trend. Axis scales are logarithmic.

	RANK		TOP 1		TOP 3		TOP 5		TOP 10	
	A	M	#	%	#	%	#	%	#	%
APR tools	2.1	1.0	57	61%	79	85%	87	94%	92	99%
PrevaRank *individual*	1.4	1.0	63	68%	91	98%	93	100%	93	100%
PrevaRank *cumulative*	1.6	1.0	48	52%	93	100%	93	100%	93	100%

Table 9: Effectiveness of PrevaRank when ranking multiple tool patches. The table compares the best (highest) rank of all *APR tools*, to the best PrevaRank rank working on each tool's patches *individually*, to PrevaRank rank's on all patches *cumulatively*. It reports the Average (mean) and Median rank, and the number # and percentage % (over all 93 bugs that more than one tool fixed) of bugs in the TOP k.

the (first) correct fix among all APR tools' output patches (for the tools that can fix the bug): this corresponds to running all APR tools, and taking the most effective one for each bug. *ii*) The best (highest) rank of the (first) correct fix among PrevaRank rankings of each APR tools' output patches *individually*: this corresponds to running PrevaRank on the patches produced by one APR tool at a time, and taking the best result for each bug. *iii*) The rank of the (first) correct fix in PrevaRank's ranking of *all* APR tools' output patches *cumulatively*: this corresponds to taking the union of all APR tool patches, and asking PrevaRank to rank it.

Overall, the effectiveness of PrevaRank hardly changes when used *cumulatively* on all patches produced by multiple tools. Furthermore, PrevaRank's ranking remains generally better than even the best one of all APR tools. The only exception is for the top-1 rank: the best APR tool in each case ranks the correct fix in the top position in 61% of the bugs; PrevaRank used individually improves this to 68% of the bugs, whereas PrevaRank used cumulatively only does it for 52% of the bugs. In this scenario, PrevaRank has to rank with perfect

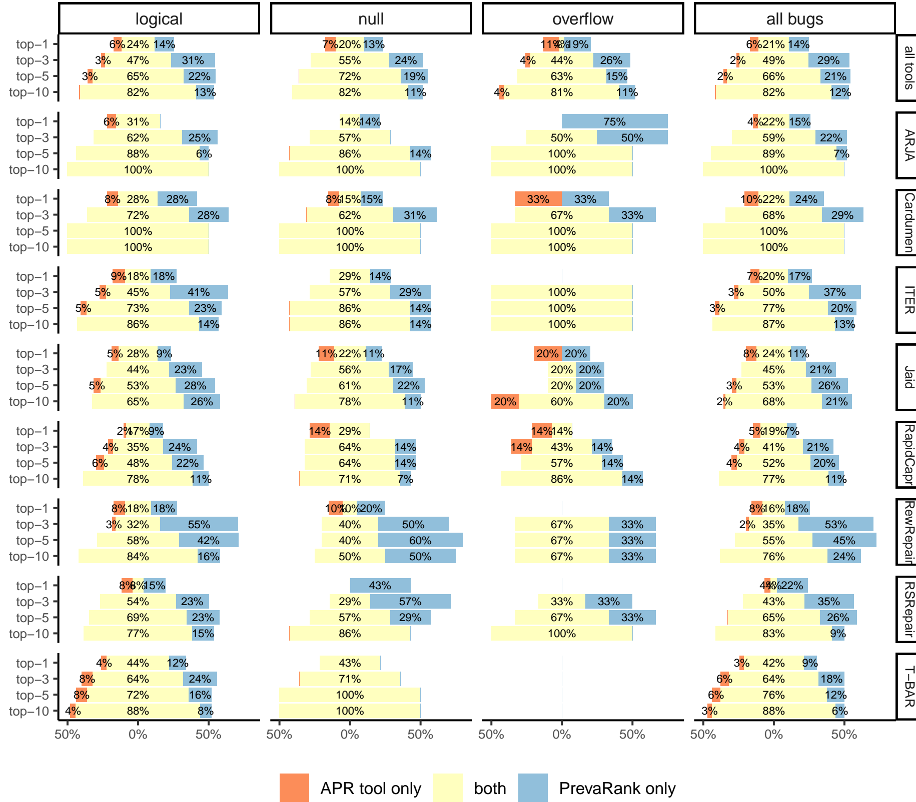

Figure 4: Percentage of bugs of each category, among those for which each APR tool can generate at least one correct fix, that are ranked in the top-1, top-3, top-5, and top-10 positions only in the APR tool's original ranking, only in PrevaRank's ranking, or in both rankings. (These groupings correspond to Table 7's three scenarios o, p, and b, respectively.)

precision a large number of patches, many of which are very similar syntactically to the correct fix—several are correct, but many aren't—and hence tend to have similar scores. PrevaRank still outperforms the best tools for the top-2 ranks (88% of bugs for PrevaRank cumulative vs. 76% for the best APR tool), but it misses a few correct fixes in the top-1 rank.

In most cases, PrevaRank improves the ranking of correct fixes compared to the original ranking produced by the APR tools: it ranks 8% more correct fixes in the top-1, 27% more in the top-3, and 11% more in the top-10. PrevaRank's effectiveness is largely independent of the APR tool that produced the patches, and of the category of bugs that are fixed.

Ranking LLM-produced patches. You may have noticed that our selection of APR tools in Section 3.2 does not include any recent LLM-based technique. As we explain in Section 4, LLM-based APR approaches often operate a closed, iterative patch generation/validation loop, and produce at most one plausible patch per

bug; therefore, a ranking technique is not directly applicable. Nevertheless, we would like to determine whether PrevaRank can still be useful on the kinds of fixes that are produced by state-of-the-art LLMs.

To this end, we conducted a small experiment with Defects4J bug *Lang22*. We prompted `GPT-4o mini` (temperature: 0.9) 20 times, with a zero-shot prompt showing the buggy code, a failing test input, and a request to provide a patch that corrects the bug. Out of the 20 patches produced by the LLM, 4 were correct. We then submitted the 20 patches to PrevaRank, which gave rank 2 to the first correct patch.[9] For comparison, the *pass*@2 score [22] of the LLM is 37%, which indicates that one should expect to find a correct patch in the first two attempts only about one third of the times. While we cannot directly compare the inherently probabilistic output of an LLM to PrevaRank's deterministic output, this small experiment suggests that an approach like PrevaRank is not limited to traditional APR techniques but remains applicable regardless of how the ranked patches were generated.

3.4. RQ2: Comparison with Other Ranking Approaches

In recent years, there has been a growing interest in approaches to classify, assess, validate, and rank patches produced by APR systems [63, 6, 64, 65, 66, 67, 68, 69]. We can roughly classify these approaches in two groups:

Sel. The first, larger group comprises approaches [63, 64, 67, 70, 66, 68, 64] that perform an additional validation step—following the APR algorithm's validation—with the goal of *selecting* a smaller collection of patches that are suggested to the user of the APR tool for their inspection. Most of these approaches are dynamic (i.e., test-based).

Rnk. The second, smaller group comprises approaches [65, 69, 68, 6] that expressly perform post-patch *ranking*.

A fair, comprehensive comparison of PrevaRank against several of these ranking tools would be difficult to execute, since each tool's experiments are not always comparable, as they may target different sets of patches or the respective publications may not include all necessary details. Instead, we identify one ranking tool in each group that: *i*) is recent, which improves the chances that its experiments details are available and are comparable to PrevaRank's; *ii*) has been experimentally compared against other tools in the same group, and the experiments showed that it (generally) outperformed them. Based on these criteria, we selected xTestCluster [64] as the benchmark for tools in group *Sel.*, and Shibboleth [6] as the benchmark for tools in group *Rnk.*.

Comparison with xTestCluster. xTestCluster [64] is a technique that groups APR patches in clusters based on how similar their runtime behavior is on the

[9] Interestingly, all 4 correct patches correspond to patch kind *add conditional*, whereas the incorrect patches belong to other patch kinds.

# CLUSTERS	# BUGS	% BUGS
1	38	72%
2	12	23%
3	3	5%

(a) Number # BUGS and percentage % BUGS of 53 Defects4J bugs (used in xTestCluster's evaluation) whose ranking of patches produced by PrevaRank spans 1, 2, or 3 of xTestCluster's clusters.

TOOL	TOP 1	TOP 2
Shibboleth	43%	66%
PrevaRank	53%	92%

(b) Percentage of 66 Defects4J bugs (used in Shibboleth's evaluation) ranked within the top-1 or top-2 positions by Shibboleth and by PrevaRank.

Figure 5: PrevaRank comparison with the State-of-the-art

available tests. Since xTestCluster (like other techniques in group *Sel.*) does not actually rank patches, we cannot directly compare its effectiveness against PrevaRank's. Instead, we use xTestCluster's *semantic* clustering capabilities to assess whether PrevaRank's ranking—which uses mainly syntactic features—is also consistent with the patches' semantic behavior. To this end, we consider all 413 patches for 53 Defects4J bugs used in xTestCluster's experiments. For each bug, xTestCluster partitions its available plausible patches into clusters according to semantic similarity: xTestCluster produced 158 clusters in total, 3 clusters per bug on average, from a minimum of 1 cluster to up to 15 clusters per bug. We first feed the patches for each of these bugs to PrevaRank as usual; this produces a ranking of the patches among each bug's plausible patches. For each bug b, we identify the rank r_b of the first correct patch among those for b; finally, we count how many clusters c_b the patches in all ranks $1..r_b$ span. The smaller c_b is, the more PrevaRank's ranking is consistent with semantic similarity, because all patches that are likely to be correct according to PrevaRank's output are also behaviorally similar. Indeed, Figure 5a shows that c_b is never larger than 3; in 72% of the cases, c_b is just 1. This suggests that PrevaRank's ranking is usually consistent with semantic similarity of patches.[10]

Comparison with Shibboleth. Since Shibboleth [6] is applicable to ranking several different patches produced by various APR techniques for the same bug, it is directly comparable to PrevaRank. Consistently with the rest of our evaluation, we only consider the 66 Defects4J bugs used in Shibboleth's evaluation that include at least four plausible patches each (for a total of 827 patches). Figure 5b shows the percentages of these bugs where Shibboleth or PrevaRank ranked a correct patch in the top-1 or top-2 position. Since Shibboleth was evaluated based on the top-1 and top-2 ranks, we do not consider lower ranks in this comparison. PrevaRank generally ranks more bugs in high positions, which suggests that PrevaRank is an effective technique that improves over the state

[10]Another way to look at these results is that PrevaRank's ranking suggests an order in which to inspect xTestCluster's clusters that is likely to give better ranking to clusters with correct patches. This observation could be useful to choose the order in which to inspect xTestCluster's clusters.

LOC OF PATCH	COMMITS	SELECTED
< 6	224 965	6 583
< 20	517 416	8 202

Table 10: How the number of SELECTED bug-fixing COMMITS changes if we allow all patches of up to 19 lines of code (bottom row) instead of 5 lines of code (top row) as done in the rest of the paper.

of the art of patch ranking.

Even though it is based on syntactic features, PrevaRank's ranking is usually consistent with clustering of patches by semantic similarity. PrevaRank's ranking accuracy also compares favorably to the capabilities of the state-of-the-art ranking tool Shibboleth.

3.5. RQ3: Robustness

We assessed PrevaRank's robustness to changes in the quantity and variety of training data with two experiments:

- *Patch variety:* As explained in Section 2.4, PrevaRank crucially relies on a syntactic *patch classification*, which assigns any given patch p to a *kind* $\mathsf{k}(p)$ based on its syntactic features. The set of possible kinds is determined by the data used for training, which only includes patches modifying at most 5 lines of code. Below, we investigate whether the restriction to patches of this size affects the variety and distribution of patch kinds that are available for classification.
- *Training data:* Like every data-driven technique, PrevaRank's capabilities depend on the size of its training data. Below, we investigate how PrevaRank's effectiveness varies as it is trained on fewer data.

Patch Variety. Table 10 shows how many historical patches we found in the same 81 open-source projects described in Section 3.1 if we consider all patches that modify up to 19 lines of code—increasing the bound of 5 lines of code used in the rest of the paper's experiments. We still did not include *all* available commits for two reasons: first, in order to keep the manual effort feasible; second, because large-scale empirical studies of program repair suggest that it is exceedingly uncommon that an APR system can generate longer patches [33, 34, 35]. Even with a limit of 20 modified lines, Table 10 clearly shows that the new selection includes substantially more commits (2.3 times more); out of the 517 416 available commits, 8 202 (1.2 times more than in the rest of the paper) were confirmed as genuine fixes targeting an overflow, null pointer, or logical bug.

The 8 202 patches were classified following the same procedure of Section 3.1. Despite including a substantially larger number of patches, the obtained distribution of the kinds of patches was very similar to the one induced by the smaller selection used in the rest of the experiments. In particular, the ten most

SAMPLE	RANK A	RANK M	TOP 1 #	TOP 1 %	TOP 3 #	TOP 3 %	TOP 5 #	TOP 5 %	TOP 10 #	TOP 10 %	WORSE #	WORSE %	SAME #	SAME %	BETTER #	BETTER %
500	6.4	3	81	23.7 %	209	61.1 %	267	78.1 %	303	88.6 %	131	38.3 %	198	57.9 %	13	3.8 %
2000	5.7	2	102	30.1 %	230	67.8 %	283	83.5 %	310	91.4 %	90	26.5 %	230	67.8 %	19	5.6 %
4000	5.1	2	105	30.4 %	250	72.5 %	295	85.5 %	318	92.2 %	76	22.0 %	248	71.9 %	21	6.1 %
all	4.1	2	121	35.4 %	266	77.8 %	300	87.7 %	322	94.2 %						

Table 11: Robustness of PrevaRank with training data of different size. The table shows how the AVERAGE (mean) and MEDIAN rank, and the number # and percentage % of bugs whose correct patch (produced by any tools) is ranked in the TOP *k* vary as PrevaRank is *trained* with a different SAMPLE of the training data: 500 commits, 2000 commits, 4000 commits, and *all* 6583 commits. In the right-hand side, it shows the number # and percentage % of bugs whose correct patch is ranked WORSE, the SAME, and BETTER when PrevaRank uses the sampled commits as training data compared to when it uses all commits.

frequently occurring patch kinds among the 8 202 patches are the same as those in Table 4; and the frequencies of logical, null pointer, and overflow errors for each kind are very similar (precisely, the difference in the most frequent bug category frequency is within 2.2%). These results indicate that considering much larger patches does not introduce significant changes in the variety and frequency of path kinds; correspondingly, the behavior of PrevaRank would also remain similar. Notice that our a priori restriction to only three broad categories of bugs (logical, null pointer, overflow) helps robustness even with a much larger selection of patches.

Training Data. In order to assess to what extent PrevaRank's ranking output changes as we *train* it with increasingly smaller training data, we retrained PrevaRank with three different random *samples* of the complete training data (described in Section 3.1, which we used in the rest of the paper's experiments): a sample size of 4000 (60%), 2000 (30%), and 500 (8%, or an order of magnitude less than the original dataset) out of all 6583 bug commits. Precisely, we performed *stratified sampling* [71] using bug kinds (logical, overflow, null pointer) as strata to ensure that bugs of all kinds are adequately represented in the sample. This resulted in the following selection of commits for each sample size:[11]

SAMPLE	*logical*	*overflow*	*null*
500	87	28	385
2000	600	150	1250
4000	900	250	2850

After retraining PrevaRank using each sample, we compared its ranking results (on the same APR patches used in the rest of the evaluation) to those obtained in RQ1 (i.e., with PrevaRank trained on all 6583 bug commits). Figure 6 and Table 11 show the results of these experiments, comparing them to

[11]We repeated 3 times the sampling for each size; we report averages of all measures.

	TASK	MISCLASSIFICATIONS	FALSE POSITIVES	FALSE NEGATIVES
T_C	bug classification	5.5%		
T_+, T_-	bug-fix identification	24.7%	40.0%	9.5%

Table 12: Comparison between an LLM and the manual classification of training data.

the baseline (bottom row, corresponding to PrevaRank trained with all available data). As one should expect, the effectiveness of PrevaRank degrades when it is trained with less data: this applied to pretty much all measures, from the average rank of a correct fix to the number/percentage of bugs whose correct fix is ranked in the top-1, top-3, top-5, or top-10. Nevertheless, the worsening of ranking accuracy is gradual and not dramatic: the average rank of a correct fix, for example, worsens by only 2.3 positions (from 4.1 to 6.4) when reducing the training data to only 500 samples. Similarly, the number of correct fixes ranked in the top-10 decreases by only 4 (from 322 to 318, corresponding to 2 percentage points) when training on 4000 samples. In all cases, the ranks of the majority of bugs do not change (columns SAME in Table 11), whereas the ranks of between 1/5 and 2/5 of all bugs worsen (columns WORSE). Interestingly, for a few bugs, the ranking improves when training PrevaRank on a smaller sample (columns BETTER). Like every statistical approach that learns from data, PrevaRank's behavior is not deterministic and possibly inconsistent; importantly, our experiments show that such anomalies occur only sporadically and do not qualitatively affect PrevaRank's overall capabilities. The granularity of patch features (used to classify patches into kinds, as explained in Section 2.4) underlies PrevaRank's robust behavior: there still is a variety of features even if the training data is curtailed, which is often sufficient to still obtain a good-quality ranking.

*PrevaRank's effectiveness degrades gracefully as the size of the training data shrinks: even when less than 8% (*500/6583*) of the training data is used, the rank assigned to 57.9% of the bugs does not change.*

3.6. RQ4: Automation of Training

As described in Section 2.3 and Section 3.1, our experiments with PrevaRank involved a partially manual validation of training data, aimed at ensuring that the selection of bugs and classification of programmer-written patches used to train PrevaRank is as accurate as possible. In this section, we describe some exploratory experiments that assess whether this manual effort can be reduced.

In the time since we first developed PrevaRank, LLMs (Large Language Models) have made spectacular advances, and have been applied to automate various software engineering tasks [72, 73]. To determine whether they can replicate the manual validation we performed to select PrevaRank's training data, we set up three tasks:

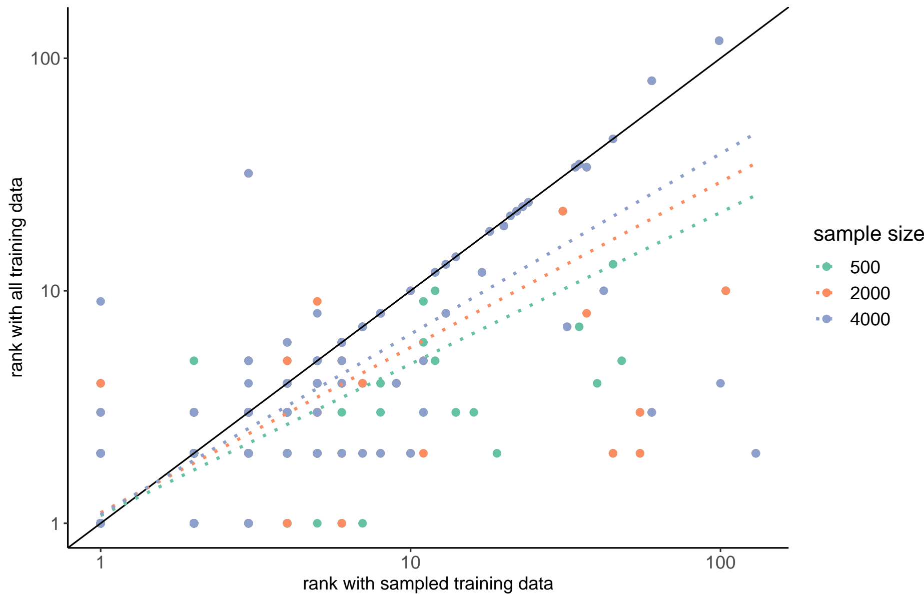

Figure 6: Each point represents a bug fixed by an APR tool. The point's x coordinate is the rank assigned by PrevaRank when it uses a randomly sampled subset of the training data; the point's y coordinate is the rank assigned by PrevaRank when it uses all training data. Thus, points below the diagonal line correspond to bugs where using all training data improves the ranking over using only a subset. The dotted lines are the linear regression lines of the points in each group. Axis scales are logarithmic.

T_- We sampled randomly 200 commits[12] out of the 6583 commits that we selected as genuine bug fixes and used for training. For each commit, we asked the LLM to determine whether it represents a genuine fix of a logical, null pointer, or overflow bug.

T_+ We sampled randomly 200 commits out of the 1201 ($= 7784 - 6583$) commits whose message matches one of Listing 1's regexes but that manual analysis discarded as *not* valid bug fixes. For each commit, we asked the LLM to determine whether it represents a genuine fix of a logical, null pointer, or overflow bug.

T_C We sampled randomly 200 commits out of the 6583 commits that we selected as genuine bug fixes and used for training. For each commit, we asked the LLM to determine whether the bug that the commit fixes is a logical, null pointer, or overflow bug.

All experiments were conducted using `GPT-4o mini`. In each query, we prompted the LLM through its API with a commit diff and message, the

[12]This sample size is sufficient to estimate the accuracy of a binary classification with up to 5.7% error and 90% probability with the most conservative (i.e., 50%) a priori assumption [74].

list of available bug categories (logical, null pointer, overflow); we asked it to answer the query in JSON format and to provide a short reason for its output. In taking stock of the results, we used our manual classification as ground truth.

Table 12 summarizes the performance of the LLM on these three tasks. In task T_C, the LLM's classification of 11/200 (5.5%) bug-fixing commits disagrees with ours; most of the misclassifications involve overflow errors, which the LLM classified as logical errors (possibly because the fixes usually involve changing a Boolean condition). In task T_-, the LLM tagged 19/200 (9.5%) commits as non-bug fixes, in contrast to our manual assessment that judged all these commits as actual bug fixes. In task T_+, the LLM tagged 80/200 (40.0%) commits as real bug fixes, in contrast to our manual assessment that judged all these commits as non bug fixes. Overall, the LLM misidentified $(80 + 19)/(200 + 200)$ (24.7%) commits. Given this substantial disagreement, we revised again our manual assessment and confirmed it as accurate. In many cases, it seems that the LLM overlooked developer-written commit messages that explicitly marked a commit as addressing a "typo" or a "copy-paste error", and focused exclusively on the patch's code, classifying the commits as fixing non-existent logic or null pointer bugs.

These results suggest that LLMs still lag behind human expertise when it comes to precisely identifying bug-fixing commits. Given that the manual effort involved in curating the training data for PrevaRank (which we estimated to be around two person-days) can be amortized over many usages, it is arguably worth the additional accuracy that it achieves—at least until LLM technology catches up. If PrevaRank were used on a continuous basis, it could also be updated with additional training data by simply supplying any new bug fix that is validated during standard code reviews of a project, without need for a dedicated batch manual analysis.

In our experiments, state-of-the-art LLMs incorrectly identified around 25% of the bug-fixing commits used for PrevaRank's training phase.

3.7. Threats to Validity

This section addresses the main potential threats to our experiments' internal and external validity and mitigation.

Internal validity. The choice of training data and of experimental subjects may affect our experiments' internal validity. The training data (Section 3.1) consisted of the commit history of 81 GitHub projects. We applied standard selection criteria (size, commits, stars) to ensure that we only used realistic projects of good quality; to avoid data leakage, we excluded any projects that feature in the Defects4J benchmark collection (which we used to evaluate PrevaRank's ranking effectiveness). In order to minimize the chances that the training data is noisy, two authors performed a manual validation to confirm that we only retained bug-fixing commits with a precise classification of their bugs. Despite these precautions, we cannot exclude that a few misclassified bug-fixing commits

remained in the training data. Even if this is the case, our experiments remain valid to empirically assess PrevaRank's ranking effectiveness.

Limiting the patches used for training to 5 lines of code is another possible threat to internal validity. In our robustness experiments, we relaxed this constraint and found that including much larger patches (up to 20 lines of code) does not seem to bring major changes in the frequency distributions that are learned by PrevaRank; this helps mitigate this threat by providing an assessment of its impact.

The robustness experiments in Section 3.5 also assess how PrevaRank's performance worsens as the training data shrink. While the performance degradation we observed in these experiments was gradual, we cannot rule out that using a very different sampling strategy (in particular, one that produces a very biased dataset) may lead to different results. Like all data-based approaches, PrevaRank's capabilities ultimately depend on the quality of the training data; but our experiments exhibited a reasonable degree of robustness.

The selection of APR patches (the experimental subjects) included all the patches available from a wide selection of APR tools that have been evaluated on the Defects4J curated collection; we only excluded subjects where ranking would make little sense (because there are only a few patches, or they all are nearly identical): this is a conservative choice, as including these scenarios (where any ranking is acceptable) could have overestimated PrevaRank's capabilities. We also directly used patches as they were made available by the authors of the APR techniques that produced them; this reduces the risk that we misuse an APR tool, leading it to produce patches that are not representative of its intended usage. Finally, our analysis of experimental results used widely accepted metrics (e.g., top-k, average rank) and standard visualizations and statistics—which mitigates further potential threats to internal validity.

External validity. concerns the generalizability of results. We tried to evaluate PrevaRank's effectiveness with as varied a selection of APR tools as possible, and we noticed that its performance does not change dramatically on different tools; this is consistent with the fact that PrevaRank was designed to be tool-agnostic. At the same time, PrevaRank is practically useful only when used in combination with APR tools capable of producing several different plausible patches for the same bug. On the one hand, several cutting-edge LLM-based APR systems do not meet this requirement, which limits PrevaRank's general applicability. On the other hand, as discussed in Section 4, there remain scenarios where more traditional APR approaches are preferable, which is precisely where PrevaRank can be proficiently applied.

While we did not collect evidence to claim further generalizability (to other categories bugs, or to other programming languages), it is possible that—thanks to their simplicity—some of PrevaRank's basic ideas are applicable also in different scenarios. Properly evaluating whether this is the case requires additional experiments and belongs to future work.

4. Related Work

Automated program repair (APR) aspires to improve software quality and reduce software development and maintenance costs by automatically suggesting fixes to program bugs [75, 76]. The most common APR techniques that target general classes of software bugs are *test-driven*: passing tests define expected (correct) program behavior that a fix should preserve, whereas failing tests define incorrect behavior that a fix should prevent or rectify. Patches that pass all tests are referred to as *valid* or *plausible*; since any collection of tests cannot completely specify program correctness, a plausible patch passing all the given tests can still violate other, unspecified constraints (or, more generally, fail to comply with the programmer's expectations) and, therefore, be incorrect. Plausible but incorrect patches are said to overfit the available tests. The overfitting problem has been widely observed in test-driven APR [2, 77]; several studies have further investigated its causes and characteristics [78, 79]. Despite these efforts, overfitting remains a major challenge for current repair systems [80, 81].

Test-driven APR techniques can be loosely classified into constraint-based, heuristic, and learning-based, depending on how they synthesize candidate patches. *Constraint-based* techniques replace suspicious expressions in the faulty program with symbolic variables ("holes"), build constraints on those variables w.r.t. all the available tests, and then solve the constraints to find valid fixes that are plausible by construction [82, 83, 84, 85]. Since a constraint must encode program behavior in purely logical form, practical challenges that constraint-based techniques face include handling expressions with side effects and generating patches that introduce complete statements.

At a high level, all constraint-based techniques search over a space of candidate patches for plausible ones. The search space is defined by the program locations that may be faulty and by the modification patterns applicable to those faulty locations (the so-called "fix ingredients" [86]), and validation is done by applying each candidate patch to the program and checking whether it passes all tests. The majority of test-driven APR techniques use heuristics to guide the search. Early examples include GenProg [87] and PAR [88]. Other approaches, such as SPR [89] and history-driven repair [4], employ novel search-based strategies based on exploiting different kinds of information. Techniques such as ssFix [90], Jaid [3], CapGen [91], and Hercules [92] make the search for a patch more efficient by abstracting the search space and selecting patch ingredients. Other systems explored similar ideas but along novel directions [2, 93, 94, 95].

Learning-based approaches inductively learn patterns programmer-written fixes in project histories, and use the learned information to improve (parts of) the APR process. Early work focused on learning repair patterns to guide template instantiation or transformation selection, thus prioritizing the generation of fixes that resemble human-written patches in the training set [96, 38]. Other approaches trained probabilistic models to estimate whether a plausible patch is likely to be correct, which supports patch candidate ranking or filtering [97, 41, 98].

With the rise of deep learning, multiple approaches have framed program

repair as a neural machine translation problem, leveraging sequence-to-sequence architectures to transform buggy code into repaired code [99, 100, 48]. Subsequent research incorporated syntactic structure into neural models, for example by leveraging AST representations, grammar-constrained decoding, or structured edit generation to improve syntactic validity and patch quality [101, 53, 55]. More recent systems enhance neural program repair by incorporating richer contextual modeling, structured decoding mechanisms, iterative refinement strategies, or execution-based feedback to improve patch correctness and handle more complex bugs. [49, 59, 50, 57].

A recent survey [7] systematically reviews a comprehensive selection of learning-based APR techniques. The survey presents the general idea of learning-based APR as a workflow comprising several stages. These stages include fault localization, data representation, patch generation, patch ranking, patch validation, and correctness assessment. This pipeline view suggests that each repair technique can focus on different stages of the overall process. In fact, many learning-based and LLM-based systems mainly focus on generating and validating patches; they often achieve this through repeated generate-then-validate cycles. In contrast, explicit ranking methods aim to improve the order of candidate patches that have already been made. The present paper's PrevaRank fits into this ranking stage. It works on candidate patches after they are generated and prioritizes those that are more likely to be correct. As a result, it is independent of patch-generation methods and can be used with various repair approaches.

Like nearly every other field in software engineering, automated program repair has been influenced by Large Language Models (LLMs). Early LLM-based APR studies explored zero-shot repair settings and prompt-based generation [52]. Subsequent work integrated LLMs into copilot-style or completion-based repair pipelines [51]. Other approaches investigated entropy-based scoring to assess patch plausibility [102]. Fine-tuning strategies tailored large language models specifically for APR tasks have also been proposed [103]. Template-guided and hybrid repair methods further extended LLM-assisted APR [45, 23]. More recent systems adopt conversational, agent-based, or self-directed paradigms to iteratively generate and refine patches [14, 104, 15, 25]. Self-directed repair strategies have also been explored [104].

Early LLM-based APR techniques often rank potential fixes by metrics like entropy [105, 52, 102]—where a lower score suggests a more "natural" [36] patch. However, most newer approaches forgo explicit ranking and instead rely primarily on the LLM's generative capabilities. Several representative systems follow this paradigm [51, 103, 45]. Other recent approaches similarly adopt generate-and-validate workflows without a separate ranking component [14, 104, 23, 15, 25]. Within this context, PrevaRank remains a relevant contribution in two ways. First, as we argued in Section 1.1, while LLMs possess impressive code capabilities, their application in APR can be limited by budget, security, or practical constraints. Second, recent studies [23, 9] have shown that traditional APR approaches remain complementary to LLM-guided repair, especially when the required fix is small and syntactically localized. Edits such as adding a null check, adjusting a conditional, or correcting a simple assignment or initialization

are still effectively handled by template-based techniques [23], which avoid many of the downsides and overhead of using LLMs. While state-of-the-art LLM-driven systems routinely demonstrate strong overall APR performance, their results often depend heavily on the quality of fault localization, the size of the underlying model, and the specific inference setup [106, 9]. By helping these traditional APR approaches better rank their candidate fixes, PrevaRank directly enhances their effectiveness and practical usability.

The rest of this section reviews various approaches to rank plausible patches so that those more likely to be correct are generated, validated, or suggested earlier. Note that deep-learning models naturally incorporate information, in the form of a rich feature collection, that can be used to predict which patches are more likely correct.

Repair context. APR's redundancy assumption is the claim that large programs contain the seeds of their own repairs; APR techniques built on this hypothesis prefer patches that share characteristics with the code's repair context. For example, GenProg [87] is based on the assumption that defects can be repaired by taking code elements from other locations in the program under repair. ssFix [90] matches contextual information at the fixing location to a database of human-written fixes, and uses the matching code elements to drive patch generation. SimFix [38] combines the information extracted from existing patches and snippets similar to the code under repair to make the search for correct fixes more efficient. CapGen [91] prioritizes fix ingredients based on a notion of programming context. Hercules [92] identifies a set of repair locations with similar code and likely demanding similar repairs and builds multi-hunk fixes that modify all those locations.

Historical fixes. Given that both program bugs and their corrections are highly repetitive (that is, new bugs are often similar to old bugs, and are fixed by similar repairs), deployed programmer-written fixes in history are considered another important source of information to help prioritize candidate patches. For instance, PAR [88], Jaid [3], Restore [17], and TBar [44] construct candidate fixes by instantiating a group of repair patterns constructed manually or summarized from the literature; Tenure [56] and Gamma [45] uses a deep neural network trained on human-written fixes to predict which predefined repair patterns should be selected for repairing a new bug; SPR [89] systematically generates candidate fixes according to a set of predefined transformation functions; Prophet [97] adds on top of SPR a probabilistic model, learned from human-written fixes, to effectively prioritize the generated candidate patches. The effectiveness of these techniques is ultimately constrained by the number and diversity of their predefined repair patterns and transformation functions. To overcome this limitation, more recent approaches directly learn characteristics from correct fixes and apply the characteristics to guide program repair. The history-driven approach [4] is based on a stochastic search process that views patches as mutants, and prefers the mutants that match the characteristics learned from fix history. Genesis [96] infers code transformations with template variables from human-

written patches. Elixir [107] trains a model that captures the characteristics of correct fixes of buggy method invocations, and utilizes the trained model to guide the repair of the same type of bugs. The BATS approach [108] is based on the hypothesis that bugs that are revealed by similarly failing tests require similar patches; it trains a deep-learning model to recognize associations between similar tests and similar historical repairs, and then uses the trained model to recognize patches that are more likely to be correct. FixMiner [109] extracts patterns of correct fixes as sequences of edit actions on a program's abstract syntax tree. TypeFix [110] automatically mines, from past fixes, a collection of templates with common edit patterns and bug context information, and invokes a code pre-trained model to instantiate the templates for repairing new bugs.

Dynamic information. Given that the tests, given as input to an APR tool, define which patches are plausible, it is natural for APR techniques to prioritize candidate patches based on their impact on test execution. In approaches based on genetic algorithms [111, 112, 87], the fitness function typically includes information that comes from validation: the more tests a candidate fix passes, the "fitter" it is. Restore [17] updates the suspiciousness values of various program locations by factoring in the validation results of candidate patches generated at those locations. Inspired by the competent programmer hypothesis [113], various APR techniques generate new tests for the program and prefer candidate patches that have less impact on the program's existing correct behavior [63, 68, 114, 115, 69, 6]. Such techniques differ in how they model program behavior, and in how they quantitatively compare different behaviors. DiffTGen [66] follows a different approach, wherein it generates new tests to uncover semantic differences between the original faulty program and the repaired program; developers can then investigate its output, which helps them discriminate between overfitting and correct patches.

Compared with the existing fix prioritization techniques, PrevaRank ranks plausible fixes according to their feature similarity with historic programmer-written fixes for similar bugs. PrevaRank implements simple heuristics, which helps make it scalable and applicable to any APR tool that produces plausible patches.

5. Conclusions and Future Work

This paper presented PrevaRank: a lightweight technique to rank the patches produced by any APR technique. PrevaRank uses the frequencies of similar patches in historical project repositories to suggest which APR patches are more likely to be genuinely correct. Experiments demonstrated that PrevaRank substantially improves the ranking of correct patches (e.g., 27% more ranked in the top-3) and is quite robust with respect to the size of the training data.

We envision two natural directions for future work. First, we will consider using more sophisticated machine learning techniques than the simple frequency-based classification currently used by PrevaRank: while simplicity was a deliberate design feature to ensure scalability and generality, trading off

some of these features for even better effectiveness may be advantageous in some cases. Second, we will investigate how PrevaRank generalizes to rank patches for other types of bugs or to programming languages other than Java; its simple design may help enable such generalizations.

6. Data Availability

Our prototype implementation of PrevaRank, as well as the detailed experimental results, are available at `https://figshare.com/s/21a42b5f72978e803768`

References

[1] Y. Yuan, W. Banzhaf, Arja: Automated repair of java programs via multi-objective genetic programming, IEEE Transactions on Software Engineering 46 (10) (oct 2020). doi:10.1109/TSE.2018.2874648.

[2] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, A. Zeller, Automated Fixing of Programs with Contracts, IEEE Transactions on Software Engineering 40 (5) (2014) 427–449. doi:10.1109/TSE.2014.2312918.

[3] L. Chen, Y. Pei, C. A. Furia, Contract-Based Program Repair without the Contracts, in: Proceedings of the 2017 32th IEEE/ACM International Conference on Automated Software Engineering, Urbana-Champaign, IL, USA, 2017, pp. 637–647. doi:10.1109/ASE.2017.8115674.

[4] X. B. D. Le, D. Lo, C. L. Goues, History Driven Program Repair, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 213–224. doi:10.1109/SANER.2016.76.

[5] R. Just, D. Jalali, M. D. Ernst, Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 international symposium on software testing and analysis, 2014. doi:10.1145/2610384.2628055.

[6] A. Ghanbari, A. Marcus, Patch correctness assessment in automated program repair based on the impact of patches on production and test code, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 654–665. doi:10.1145/3533767.3534368.

[7] Q. Zhang, C. Fang, Y. Ma, W. Sun, Z. Chen, A survey of learning-based automated program repair, ACM Trans. Softw. Eng. Methodol. 33 (2) (dec 2023). doi:10.1145/3631974.

[8] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, Y. Zhang, Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities, ACM Comput. Surv. 57 (2) (Oct. 2024). doi:10.1145/3696450.

[9] B. Yang, Z. Cai, F. Liu, B. Le, L. Zhang, T. F. Bissyandé, Y. Liu, H. Tian, A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications (2025). arXiv:2506.23749.

[10] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al., Qwen3 technical report, https://arxiv.org/abs/2505.09388 (2025). arXiv:2505.09388.

[11] S. Quan, J. Yang, B. Yu, B. Zheng, D. Liu, A. Yang, X. Ren, B. Gao, Y. Miao, Y. Feng, Z. Wang, J. Yang, Z. Cui, Y. Fan, Y. Zhang, B. Hui, J. Lin, Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings (2025). arXiv:2501.01257.

[12] A. Silva, M. Monperrus, RepairBench: Leaderboard of frontier models for program repair, in: Proceedings of the International Workshop on Large Language Models for Code (LLM4Code), 2025. doi:10.1109/LLM4Code66737.2025.00006.
URL https://repairbench.github.io/

[13] A. Kipf, T. Schmidt, P. Kuo, S. Krid, M. Rengert, L. Heller, A. Zimmerer, M. Stoian, V. Pandey, A. van Renen, Waiting to decompress: The economics of llm-based compression, in: 16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026, www.cidrdb.org, 2026.

[14] C. S. Xia, L. Zhang, 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, ISSTA 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 819–831. doi:10.1145/3650212.3680323.

[15] I. Bouzenia, P. Devanbu, M. Pradel, RepairAgent: An autonomous, LLM-based agent for program repair, in: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering, ICSE '25, IEEE Press, 2025, p. 2188–2200. doi:10.1109/ICSE55347.2025.00157.

[16] L. Chen, Y. Pei, C. A. Furia, Contract-based program repair without the contracts: An extended study, IEEE Transactions on Software Engineering 47 (12) (2021) 2841–2857.

[17] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, C. A. Furia, Restore: Retrospective fault localization enhancing automated program repair, IEEE Transactions on Software Engineering 48 (1) (2022) 309–326. doi:10.1109/TSE.2020.2987862.

[18] Y. Du, Z. Li, N. Li, B. Ding, Beyond data privacy: New privacy risks for large language models (2026). arXiv:2509.14278.

[19] Z. Ma, J. Zhang, Y. Yuan, H. Yang, S. Liu, W. Yang, D. Qian, On protecting the data privacy of large language models (LLMs) and LLM agents: A literature review, High-Confidence Computing 5 (2) (2025) 100300. doi:10.1016/j.hcc.2025.100300.

[20] I. Barberá, AI privacy risks & mitigations: Large language models (LLMs), https://www.edpb.europa.eu/our-work-tools/our-documents/support-pool-experts-projects/ai-privacy-risks-mitigations-large_en, a report by the EDPB (European Data Protection Board) (2025).

[21] A. Giagnorio, A. Martin-Lopez, G. Bavota, Enhancing code generation for low-resource languages: No silver bullet, in: 33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025, IEEE, 2025, pp. 478–488. doi:10.1109/ICPC66645.2025.00058.

[22] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code (2021). arXiv:2107.03374.
URL https://arxiv.org/abs/2107.03374

[23] K. Huang, J. Zhang, X. Meng, Y. Liu, Template-Guided Program Repair in the Era of Large Language Models, IEEE Press, 2025, p. 1895–1907. doi:10.1109/ICSE55347.2025.00030.

[24] I. Ceka, S. Pujar, S. Ramji, L. Buratti, G. Kaiser, B. Ray, Understanding software engineering agents through the lens of traceability: An empirical study (2025). arXiv:2506.08311.
URL https://arxiv.org/abs/2506.08311

[25] L. Xie, Z. Li, Y. Pei, Z. Wen, K. Liu, T. Zhang, X. Li, PReMM: Llm-based program repair for multi-method bugs via divide and conquer, Proc. ACM Program. Lang. 9 (OOPSLA2) (Oct. 2025). doi:10.1145/3763097.

[26] R. Just, Defects4j: A collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research, https://github.com/rjust/defects4j, accessed: May 2024 (2024).

[27] E. C. Campos, M. de Almeida Maia, Common bug-fix patterns: A large-scale observational study, in: A. Bener, B. Turhan, S. Biffl (Eds.), 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017, IEEE Computer Society, 2017, pp. 404–413. doi:10.1109/ESEM.2017.55.

[28] E. C. Campos, M. de Almeida Maia, Discovering common bug-fix patterns: A large-scale observational study, J. Softw. Evol. Process. 31 (7) (2019). doi:10.1002/SMR.2173.

[29] B. Lin, S. Wang, M. Wen, X. Mao, Context-aware code change embedding for better patch correctness assessment, ACM Trans. Softw. Eng. Methodol. 31 (3) (2022) 51:1–51:29. doi:10.1145/3505247.

[30] K. Pan, S. Kim, E. J. W. Jr., Toward an understanding of bug fix patterns, Empir. Softw. Eng. 14 (3) (2009) 286–315. doi:10.1007/S10664-008-9077-5.

[31] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the International Conference on Automated Software Engineering, 2014, pp. 313–324. doi:10.1145/2642937.2642982.

[32] R. Hosseini, P. Brusilovsky, Javaparser; A fine-grain concept indexing tool for java problems, in: AIED Workshops, Vol. 1009 of CEUR Workshop Proceedings, CEUR-WS.org, 2013.

[33] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, M. de Almeida Maia, Dissection of a bug dataset: Anatomy of 395 patches from defects4j, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 130–140. doi:10.1109/SANER.2018.8330203.

[34] J. M. Zhang, F. Li, D. Hao, M. Wang, H. Tang, L. Zhang, M. Harman, A study of bug resolution characteristics in popular programming languages, IEEE Transactions on Software Engineering 47 (12) (2021) 2684–2697. doi:10.1109/TSE.2019.2961897.

[35] T. Durieux, F. Madeiral, M. Martinez, R. Abreu, Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 302–313. doi:10.1145/3338906.3338911.

[36] A. Hindle, E. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, Proceedings - International Conference on Software Engineering (2012) 837–847doi:10.1109/ICSE.2012.6227135.

[37] B. Ray, V. J. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. T. Devanbu, On the "naturalness" of buggy code, in: L. K. Dillon, W. Visser, L. A. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 428–439. doi:10.1145/2884781.2884848.

[38] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 298–309. doi:10.1145/3213846.3213871.
URL https://doi.org/10.1145/3213846.3213871

[39] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. Le Traon, You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 102–113. doi:10.1109/ICST.2019.00020.

[40] M. Martinez, M. Monperrus, ASTOR: a program repair library for Java (demo), in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 441–444. doi:10.1145/2931037.2948705.

[41] M. White, M. Tufano, M. Martínez, M. Monperrus, D. Poshyvanyk, Sorting and transforming program repair ingredients via deep learning code similarities, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 479–490. doi:10.1109/SANER.2019.8668043.

[42] A. Fonseca, M. Oliveira, Figra: Evaluating a larger search space for cardumen in automatic program repair, in: 2022 IEEE/ACM International Workshop on Automated Program Repair (APR), 2022, pp. 24–30. doi:10.1145/3524459.3527345.

[43] Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang, The strength of random search on automated program repair, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 254–265. doi:10.1145/2568225.2568254.

[44] K. Liu, A. Koyuncu, D. Kim, T. F. Bissyandé, Tbar: revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 31–42. doi:10.1145/3293882.3330577.

[45] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, Z. Chen, Gamma: Revisiting template-based automated program repair via mask prediction, in: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE '23, IEEE Press, 2024, p. 535–547. doi:10.1109/ASE56229.2023.00063.

[46] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, H. Yin, Circle: continual repair across programming languages, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 678–690. doi:10.1145/3533767.3534219.

[47] N. Jiang, T. Lutellier, L. Tan, Cure: Code-aware neural machine translation for automatic program repair, in: 2021 IEEE/ACM 43rd International

Conference on Software Engineering (ICSE), 2021, pp. 1161–1173. doi:10.1109/ICSE43902.2021.00107.

[48] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, M. Monperrus, Sequencer: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering 47 (9) (2021) 1943–1959. doi:10.1109/TSE.2019.2940179.

[49] Y. Li, S. Wang, T. N. Nguyen, Dlfix: Context-based code transformation learning for automated program repair, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 602–614. doi:10.1145/3377811.3380345.

[50] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, X. Zhang, Knod: Domain knowledge distilled tree decoder for automated program repair, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 1251–1263. doi:10.1109/ICSE48619.2023.00111.

[51] Y. Wei, C. S. Xia, L. Zhang, Copiloting the copilots: Fusing large language models with completion engines for automated program repair, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 172–184. doi:10.1145/3611643.3616271.

[52] C. S. Xia, L. Zhang, Less training, more repairing please: revisiting automated program repair via zero-shot learning, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 959–971. doi:10.1145/3540250.3549101.

[53] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, L. Tan, Coconut: combining context-aware neural translation models using ensemble for program repair, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 101–114. doi:10.1145/3395363.3397369.

[54] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, Improving fault localization and program repair with deep semantic features and transferred knowledge, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 1169–1180. doi:10.1145/3510003.3510147.

[55] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, L. Zhang, A syntax-guided edit decoder for neural program repair, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference

and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 341–353. doi:10.1145/3468264.3468544.

[56] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, C. Hu, Template-based neural program repair, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 1456–1468. doi:10.1109/ICSE48619.2023.00127.

[57] H. Ye, M. Monperrus, Iter: Iterative neural repair for multi-location patches, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. doi:10.1145/3597503.3623337.

[58] D. Hidvégi, Token budget minimisation of large language model based program repair, Master's thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Stockholm, Sweden (2023).

[59] H. Ye, M. Martinez, M. Monperrus, Neural program repair with execution-based backpropagation, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 1506–1518. doi:10.1145/3510003.3510222.

[60] Y. Noller, R. Shariffdeen, X. Gao, A. Roychoudhury, Trust enhancement issues in program repair, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 2228–2240. doi:10.1145/3510003.3510040.

[61] H. Eladawy, C. Le Goues, Y. Brun, Automated program repair, what is it good for? not absolutely nothing!, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. doi:10.1145/3597503.3639095.

[62] F. N. Meem, J. Smith, B. Johnson, Exploring experiences with automated program repair in practice, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. doi:10.1145/3597503.3639182.

[63] J. Yang, A. Zhikhartsev, Y. Liu, L. Tan, Better test cases for better automated program repair, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 831–841. doi:10.1145/3106237.3106274.

[64] M. Martinez, M. Kechagia, A. Perera, J. Petke, F. Sarro, A. Aleti, Test-based patch clustering for automatically-generated patches assessment, Empirical Softw. Engg. 29 (5) (Jul. 2024). doi:10.1007/s10664-024-10503-2.

[65] A. Ghanbari, ObjSim: lightweight automatic patch prioritization via object similarity, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 541–544. doi:10.1145/3395363.3404362.

[66] Q. Xin, S. P. Reiss, Identifying test-suite-overfitted patches through test case generation, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 226–236. doi:10.1145/3092703.3092718.

[67] F. Molina, J. M. Copia, A. Gorla, Improving patch correctness analysis via random testing and large language models, in: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST), 2024, pp. 317–328. doi:10.1109/ICST60714.2024.00036.

[68] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang, Identifying patch correctness in test-based program repair, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 789–799. doi:10.1145/3180155.3180182.

[69] Y. Yuan, W. Banzhaf, Toward better evolutionary program repair: An integrated approach, ACM Trans. Softw. Eng. Methodol. 29 (1) (jan 2020). doi:10.1145/3360004.

[70] H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, A. Habib, J. Klein, T. F. Bissyandé, Predicting patch correctness based on the similarity of failing test cases, ACM Trans. Softw. Eng. Methodol. 31 (4) (Aug. 2022). doi:10.1145/3511096.

[71] R. Arnab, Chapter 7 - stratified sampling, in: R. Arnab (Ed.), Survey Sampling Theory and Applications, Academic Press, 2017, pp. 213–256. doi:10.1016/B978-0-12-811848-1.00007-8.

[72] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large Language Models for Software Engineering: A Systematic Literature Review, ACM Trans. Softw. Eng. Methodol. 33 (8) (2024) 220:1–220:79. doi:10.1145/3695988.

[73] A. Gu, N. Jain, W.-D. Li, M. Shetty, Y. Shao, Z. Li, D. Yang, K. Ellis, K. Sen, A. Solar-Lezama, Challenges and Paths Towards AI for Software Engineering, arXiv:2503.22625 [cs] (Mar. 2025). doi:10.48550/arXiv.2503.22625.

[74] W. W. Daniel, C. L. Cross, Biostatistics: A Foundation for Analysis in the Health Sciences, 11th Edition, John Wiley & Sons, Hoboken, NJ, USA, 2018.

[75] C. Le Goues, M. Pradel, A. Roychoudhury, Automated program repair, Commun. ACM 62 (12) (2019) 56–65. doi:10.1145/3318162.

[76] L. Gazzola, D. Micucci, L. Mariani, Automatic software repair: A survey, IEEE Transactions on Software Engineering 45 (1) (2019) 34–67. doi:10.1109/TSE.2017.2755013.

[77] M. Monperrus, A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 234–242. doi:10.1145/2568225.2568324.

[78] Z. Qi, F. Long, S. Achour, M. Rinard, 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, ISSTA 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 24–36. doi:10.1145/2771783.2771791.

[79] S. Mechtaev, X. Gao, S. H. Tan, A. Roychoudhury, Test-equivalence analysis for automatic patch generation, ACM Trans. Softw. Eng. Methodol. 27 (4) (Oct. 2018). doi:10.1145/3241980.

[80] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, C. Pasareanu, On reliability of patch correctness assessment, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 524–535. doi:10.1109/ICSE.2019.00064.

[81] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, H. Jin, Automated patch correctness assessment: how far are we?, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Association for Computing Machinery, New York, NY, USA, 2021, p. 968–980. doi:10.1145/3324884.3416590.

[82] S. Mechtaev, J. Yi, A. Roychoudhury, Directfix: Looking for simple program repairs, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, 2015, pp. 448–458. doi:10.1109/ICSE.2015.63.

[83] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, M. Monperrus, Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, IEEE Transactions on Software Engineering PP (99) (2016) 1–1. doi:10.1109/TSE.2016.2560811.

[84] S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis, in: Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA, 2016, pp. 691–701. doi:10.1145/2884781.2884807.

[85] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, A. Roychoudhury, Semantic program repair using a reference implementation, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 129–139. doi:10.1145/3180155.3180247.

[86] M. Martinez, W. Weimer, M. Monperrus, Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches, in: P. Jalote, L. C. Briand, A. van der Hoek (Eds.), 36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014, ACM, 2014, pp. 492–495. doi:10.1145/2591062.2591114.

[87] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, Automatically finding patches using genetic programming, in: Proceedings of the IEEE 31st International Conference on Software Engineering, 2009, pp. 364–374. doi:10.1109/ICSE.2009.5070536.

[88] D. Kim, J. Nam, J. Song, S. Kim, Automatic Patch Generation Learned from Human-written Patches, in: Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, 2013, pp. 802–811. doi:10.1109/ICSE.2013.6606626.

[89] F. Long, M. Rinard, Staged Program Repair with Condition Synthesis, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, New York, NY, USA, 2015, pp. 166–178. doi:10.1145/2786805.2786811.

[90] Q. Xin, S. P. Reiss, Leveraging syntax-related code for automated program repair, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Piscataway, NJ, USA, 2017, pp. 660–670. doi:10.1109/ASE.2017.8115676.

[91] M. Wen, J. Chen, R. Wu, D. Hao, S. Cheung, Context-aware patch generation for better automated program repair, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 1–11. doi:10.1145/3180155.3180233.

[92] S. Saha, R. K. Saha, M. R. Prasad, Harnessing evolution for multi-hunk program repair, in: Proceedings of the 41st International Conference on Software Engineering, ICSE '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 13–24. doi:10.1109/ICSE.2019.00020.

[93] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: Proceedings of the 39th International Conference on Software Engineering, Piscataway, NJ, USA, 2017, pp. 416–426. doi:10.1109/ICSE.2017.45.

[94] J. Hua, M. Zhang, K. Wang, S. Khurshid, Towards practical program repair with on-demand candidate generation, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 12–23. doi:10.1145/3180155.3180245.

[95] A. Ghanbari, S. Benton, L. Zhang, Practical program repair via bytecode mutation, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, ACM, New York, NY, USA, 2019, pp. 19–30. doi:10.1145/3293882.3330559.

[96] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 727–739. doi:10.1145/3106237.3106253.

[97] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: R. Bodík, R. Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, ACM, 2016, pp. 298–312. doi:10.1145/2837614.2837617.

[98] L. Chen, Y. Pei, M. Pan, T. Zhang, Q. Wang, C. A. Furia, Program repair with repeated learning, IEEE Transactions on Software Engineering 49 (02) (2023) 831–848. doi:10.1109/TSE.2022.3164662.

[99] H. Hata, E. Shihab, G. Neubig, Learning to generate corrective patches using neural machine translation (2019). arXiv:1812.07170.

[100] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, D. Poshyvanyk, An empirical study on learning bug-fixing patches in the wild via neural machine translation, ACM Trans. Softw. Eng. Methodol. 28 (4) (Sep. 2019). doi:10.1145/3340544.

[101] S. Chakraborty, Y. Ding, M. Allamanis, B. Ray, Codit: Code editing with tree-based neural models, IEEE Transactions on Software Engineering (2020) 1–1doi:10.1109/TSE.2020.3020502.

[102] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 1482–1494. doi:10.1109/ICSE48619.2023.00129.

[103] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, Y. Zhang, An empirical study on fine-tuning large language models of code for automated program repair, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023, pp. 1162–1174. doi:10.1109/ASE56229.2023.00181.

[104] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, X. Yang, Thinkrepair: Self-directed automated program repair, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 1274–1286. doi:10.1145/3650212.3680359.

[105] S. D. Kolak, R. Martins, C. Le Goues, V. J. Hellendoorn, Patch generation with language models: Feasibility and scaling behavior, Deep Learning for Code Workshop (03 2022).
URL https://par.nsf.gov/biblio/10340618

[106] V. Campos, R. Shariffdeen, A. Ulges, Y. Noller, Empirical evaluation of generalizable automated program repair with large language models, https://arxiv.org/abs/2506.03283 (2025). arXiv:2506.03283.

[107] R. K. Saha, Y. Lyu, H. Yoshida, M. R. Prasad, Elixir: Effective object-oriented program repair, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 648–659. doi:10.1109/ASE.2017.8115675.

[108] H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, A. Habib, J. Klein, T. F. Bissyandé, Predicting patch correctness based on the similarity of failing test cases, ACM Trans. Softw. Eng. Methodol. 31 (4) (2022) 77:1–77:30. doi:10.1145/3511096.

[109] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, Y. Le Traon, FixMiner: Mining relevant fix patterns for automated program repair, Empirical Software Engineering 25 (3) (2020) 1980–2024. doi:10.1007/s10664-019-09780-z.

[110] Y. Peng, S. Gao, C. Gao, Y. Huo, M. Lyu, Domain knowledge matters: Improving prompts with fix templates for repairing python type errors, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. doi:10.1145/3597503.3608132.

[111] A. Arcuri, X. Yao, A novel co-evolutionary approach to automatic software bug fixing, in: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), 2008, pp. 162–168. doi:10.1109/CEC.2008.4630793.

[112] A. Arcuri, Evolutionary repair of faulty software, Applied Soft Computing 11 (4) (2011) 3494–3514. doi:https://doi.org/10.1016/j.asoc.2011.01.023.

[113] R. DeMillo, R. Lipton, F. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41. doi:10.1109/C-M.1978.218136.

[114] Z. Yu, M. Martinez, B. Danglot, T. Durieux, M. Monperrus, Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system, Empirical Softw. Engg. 24 (1) (2019) 33–67. doi:10.1007/s10664-018-9619-4.

[115] P. Cashin, C. Martinez, W. Weimer, S. Forrest, Understanding automatically-generated patches through symbolic invariant differences, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 411–414. doi:10.1109/ASE.2019.00046.