# Benchmarking and Categorizing the Performance of Neural Program Repair Systems for Java

WENKANG ZHONG and CHUANYI LI*, State Key Laboratory for Novel Software and Technology, Nanjing University, China
KUI LIU, Huawei Software Engineering Application Technology Lab, China
JIDONG GE* and BIN LUO, State Key Laboratory for Novel Software and Technology, Nanjing University, China
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg
VINCENT NG, University of Texas at Dallas, USA

Recent years have seen a rise in neural program repair systems in the software engineering community, which adopt advanced deep learning techniques to automatically fix bugs. Having a comprehensive understanding of existing systems can facilitate new improvements in this area and provide practical instructions for users. However, we observe two potential weaknesses in the current evaluation of NPR systems: ① published systems are trained with varying data, and ② NPR systems are roughly evaluated through the number of totally fixed bugs. Questions such as *"what types of bugs are repairable for current systems"* cannot be answered yet. Consequently, researchers can not make target improvements in this area and users have no idea of the real affair of existing systems. In this paper, we perform a systematic evaluation of the existing nine state-of-the-art NPR systems. To perform a fair and detailed comparison, we (1) build a new benchmark and framework that supports training and validating the nine systems with unified data, and (2) evaluate retrained systems with detailed performance analysis, especially on the effectiveness and the efficiency. We believe our benchmark tool and evaluation results could offer practitioners the real affairs of current NPR systems and the implications of further facilitating the improvements of NPR.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: datasets, program repair, benchmark, empirical study

## 1 INTRODUCTION

Over the last decade, Automated Program Repair (APR) [57] has emerged as a significant research area in the software engineering community. Researchers have explored various types of repair systems, such as *heuristic-based*, *template-based*, and *constraint-based* ones, to address the challenges of APR. With the advent of deep learning techniques, the latest trend in automated program repair is Neural Program Repair (NPR) [92, 95], which

*Corresponding author.

Authors' addresses: Wenkang Zhong, zhongwenkang97@foxmail.com; Chuanyi Li, lcy@nju.edu.cn, State Key Laboratory for Novel Software and Technology, Nanjing University, 22 Hankou Road, Nanjing, Jiangsu, China, 210046; Kui Liu, brucekuiliu@gmail.com, Huawei Software Engineering Application Technology Lab, , Hangzhou, Zhejiang, China, 310000; Jidong Ge, gjd@nju.edu.cn; Bin Luo, luobin@nju.edu.cn, State Key Laboratory for Novel Software and Technology, Nanjing University, 22 Hankou Road, Nanjing, Jiangsu, China, 210046; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, , Luxembourg, Luxembourg; Vincent Ng, vince@hlt.utdallas.edu, University of Texas at Dallas, , Richardson, Texas, USA.

ACM 1557-7392/2024/8-ART
https://doi.org/10.1145/3688834

utilizes deep neural networks to translate buggy code into fixed ones. In recent years, numerous NPR systems have been proposed, which could be categorized into three types: Neural Machine Translation-based (NMT-based) [2, 3, 8, 37, 48, 73, 74, 97], Open-Source-LLM-based [4, 21, 28, 54, 71, 88] and Closed-Source-LLM-based [26, 78]. In comparison to previous APR techniques, the state-of-the-art NPR systems have shown more promising bug-fixing results, as evidenced by the number of bugs correctly fixed within well-established benchmarks such as Defects4J [32] and QuixBugs [39].

To facilitate the development of NPR, it's important to understand the current affairs of existing NPR systems, which requires a fair and systematic empirical comparison of them. Such empirical studies are crucial and considered important in the field of traditional generate-and-validate APR [9, 44, 46, 65]. However, performing such a comparison on NPR systems cannot be based on published systems and results because ① current systems are trained with varying data, and ② use different settings for evaluation. As is known, for learning-based systems, the two factors impact the measured performance largely. For example, a larger beam size, which defines the number of candidate patches that an NPR system generates, surely has a higher probability of containing a correct patch. However, it also decreases efficiency since more patches need to be evaluated. We find that such potential biases are ignored in the current evaluation of NPR systems.

The pre-requirements of a fair comparison of existing NPR systems are to make those model-independent factors the same. Therefore, to facilitate the comparison and application of various NPR systems, firstly, **we build a program repair benchmark that supports the training of nine state-of-the-art NPR systems [4, 8, 21, 48, 54, 74, 88, 97], including building a <bug, fix> dataset and wrapping the source codes of the systems in an easy-to-use framework**. We focus on Java repair tools since repairing Java bugs is the most popular ecosystem in APR. Then, based on the benchmark tool, **we perform a systematic evaluation of the nine selected NPR systems**. In addition to normal evaluation that focuses merely on the number of totally fixed bugs, we further evaluate NPR systems considering their practicability. Our evaluation aims to provide researchers with the real affairs of various NPR systems and answer practical questions that users may encounter, for example, *which types of bugs can current NPR systems handle?* Concretely, we categorize our evaluation into the following two research questions:

**RQ1.** **Effectiveness.** Currently, NPRs' performance is measured by counting the total number of correctly fixed bugs in some datasets. However, such metrics have little guidance for users. Questions such as *"What types of bugs are repairable?"* are not answered yet. When a bug is given, users still have no idea which system can fix it and which cannot. For researchers, it's hard to understand the real insufficiency of current NPR systems and then make target improvements. In this question, we aim to investigate the performance of current systems in detail, that is, categorizing and measuring their repairability on different types of bugs.

**RQ2.** **Efficiency.** When utilizing an NPR system, users should first set the candidate number, and then the system produces as many patches as is set. A larger candidate number means a higher probability of fixing the bug, but meanwhile brings more costs on tool execution and patch validation. For efficient usage of NPR systems, it's important to know how NPRs' performance changes when the candidate number changes. In addition, researchers only validate the first test-adequate patch for evaluation, but in practice, users have the choice to validate more patches for better performance. Such investigations are also lacking in the current evaluation. Thus, in this question, we evaluate NPR systems' repairability under different numbers of candidate patches and plausible patches with manual validation.

To conclude, in this paper, we make the following contributions:

(1) We build a new framework tool, *NPR4J*, which supports training and evaluating seven state-of-the-art NPR systems. It is extensive to new datasets and new NPR tools.

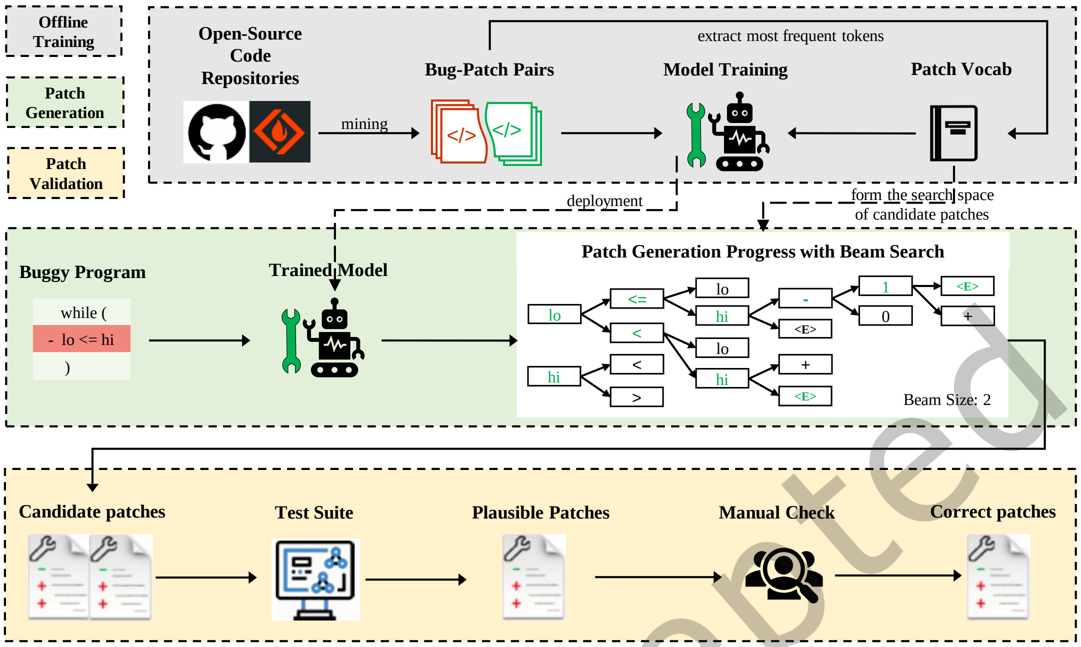

Fig. 1. The Offline Training, Patch Generation, Patch Validation progress of general NPR systems. The patch-generation phase shows how translation-based NPR systems generate patch tokens with beam search.

(2) We conduct a large-scale experiment, including retraining seven NPR systems and evaluating them on three datasets. Our experiments produce 9,070 plausible patches. All those patches are manually evaluated and 1,849 correct patches are identified.
(3) We perform a novel analysis on the effectiveness and the efficiency of NPRs, where the goal is to investigate NPRs' performance on different types of bugs, as well as the performance change when using different inference and validation settings.

The remainder of this paper is organized as follows. In section 2, we briefly introduce the NPR systems. Then we explain our motivations for building a new benchmark and evaluate the efficiency and effectiveness of NPRs. In section 3, we introduce how we build the new benchmark, *NPR4J*, including data collection and framework implementation. In section 4, we present the detailed research plan for the evaluation, including descriptions of research questions, objectives, metrics, and other experimental settings. In section 5, we analyze the experiment results and discuss the implications for practitioners. Section 8 illustrates the related works and section 9 presents the threats to validity. Section 10 concludes the paper and presents future work.

## 2 BACKGROUND AND MOTIVATIONS

This section first introduces the general three phases for training and evaluating an NPR system. Then, we discuss the motivations for why our evaluation needs to retrain existing NPR systems and why we focus on the effectiveness and efficiency of the NPR system.

## 2.1 An Introduction on Neural Program Repair Systems

As shown in Fig. 1, an NPR system needs to be trained on many bug-patch pairs. During the training phase, the goal of an NPR system is to adjust the network parameters to maximize the conditional probability of possible patches given the buggy codes:

$$P(Patch|BuggyProgram) = \prod_{i=0}^{n} P(token_i|BuggyProgram, token_{j<i}) \tag{1}$$

This is feasible since a neural network has been proven to be able to approximate any function to any accuracy [20]. *BuggyProgram* represents the input of NPR systems and *token* represents the output at each time step. Different NPR systems have different designs for these elements. For example, *BuggyProgram* can be modeled at the class level ([4, 88]), the method level ([27, 48, 74, 97]) or the line level ([8, 54]). A wider context can give an NPR system an advantage when encountering bug fixes that involve the use of variables or methods from the context but can at the same time reduce its efficiency since it needs to select more tokens. For representing the outputs, current systems use two different forms of *token*: ① modeling as a textual code token [4, 37, 48, 74] or subtoken [28, 54, 88], or ② modeling as an AST node [27, 97, 98].

The training process of NPR systems is similar to neural models applied in other fields. As shown in Figure 1, NPR systems are trained on large-scale bug-fix pairs mined from open software repositories such as GitHub. During the training progress, the NPR system is optimized by updating its parameters to fit the data distribution of the train set.

When using NPR systems for patch generation, the candidate number should be set first, then the system generates as many candidates as the set number. As shown in the patch generation phase presented in Figure 1, the patch generation process of an NPR system is a multi-step generation process where one code token is generated in each time step. The search space is defined by a vocabulary consisting of the most frequent tokens mined from the training data. At each step, the NPR system calculates the probability distribution over the patch token vocab. This process can form a huge search space. Imaging the vocab size is 1,000 and the total step is 5, the system can generate $1,000^5$ candidates. It is practically impossible to validate such a huge number of patches. Thus, NPR systems rely on a simple and effective strategy called beam search [74] to reduce the candidate space. Beam search is an adaption of greedy search in which only the $k$-best hypothesis is retained at each step, where the hypotheses are ranked by probability and $k$ is known as the beam size. The middle part of Figure 1 illustrates how an NPR system generates candidate patches with a beam search strategy when the beam size is 2. Given the buggy program as input (left), the patch generation process starts by generating the top-2 candidates for the first term (i.e., *lo*, *hi*). In the next time step, the beam search algorithm expands each current hypothesis and determines that the Top-2 most likely are those following the node *lo*. Therefore, the other two branches (i.e., <,>) are pruned. The search continues until each hypothesis reaches $< E >$, the end symbol for the sequence.

## 2.2 Motivation of Retraining existing NPR systems

In addition to the various design of NPR modules, the quantity, and quality of the training data also has a vital impact on the performance of NPR systems. As shown in Figure 1, NPR systems are trained on bug-fix pairs mined from open-source code repositories such as GitHubDuring the training phase, the NPR system updates its parameters to make the input-output of the model close to the distribution of the training data. However, we find two potential biases related to the training data:

(1) **NPR systems are trained on very different data, which may make the comparison between them unfair.** As studied in our previous investigation [94], current NPR systems are trained on very different data samples that are collected and filtered by different strategies. When a new NPR system is compared with existing systems, it generally uses their published performance on certain datasets such as

Defects4J [32], instead of re-training them with the same data. As is known, a DNN model can have very different performance when trained with different training data. Thus, it is still unclear to what extent the improvement of a new NPR system comes from the new model architecture.

(2) **There are some overlapped bugs between NPR systems' training data and the generally used evaluation data set (e.g., Defects4J [32]), which may cause the unreal measured performance.** Since the training data for NPR and evaluation datasets for program repair are generally collected from open-source repositories, there may exist some overlapped data samples. A recent investigation [22] finds that among the eight datasets that are used to train NPR systems, six of them overlapped with Defects4J [32]. Thus, the current measured performance of existing NPR systems may not exactly reflect the real repairability since they fix some bugs only because they have seen the same bugs in the training data.

Therefore, our evaluation requires retraining existing NPR systems. To further facilitate the application and evaluation of those systems, we build a framework tool that supports training and evaluating seven state-of-the-art NPR systems and build a new pure dataset for both training and evaluating them. The concrete contents of how we collect the dataset and build the framework tool are introduced in section 3.

## 2.3 Motivation of Evaluating the Effectiveness

Monperrus et al. [56] presents a critical review of the evaluation of Automated Program Repair in 2014. The authors argue that: *a contribution on automatic software repair should answer the following questions: For which defect class does it work? This names the enemy and enables the community to answer the related questions: what are the "repairable" defect classes, why is a defect class easy/hard to repair?* We strongly agree with their opinions. Unfortunately, the following research on developing new APR systems barely analyzes the performance of different defect classes. Currently, the majority of them are evaluated by merely counting the number of correct fixed bugs on well-established benchmarks such as Defects4J [32]. Such stuffless evaluation brings little usefulness to users since they still don't know when a bug is given, or whether an APR system could fix it.

We believe a detailed evaluation that identifies the defect classes that can be repaired or not by APR systems could benefit the community. Moreover, we argue that such evaluation is essentially important for NPR systems. A typical characteristic of NPR systems is that they only require the source codes of the bug as the input, which means that they can accept and produce outputs for almost any kind of bug. *However, would NPR systems generate outputs effectively equally for all those bugs?* In the deep learning field, We noticed the latest hot term "Model Hallucination" [23], which means the generated content of large language models that is nonsensical or unfaithful to the provided source content. In the NPR field, we also have the same worry. They may produce patches for any given bug but do not promise that their outputs are effective. In some worse cases, the generated patches may introduce more bugs [72]. Therefore, a comprehensive evaluation of NPR systems should answer what types of bugs they are good or poor at. In this paper, we name such evaluation as the **Effectiveness** of the NPR model and perform a seminal evaluation of the effectiveness of existing NPR models by categorizing their performance into different defect classes.

## 2.4 Motivation of Evaluating the Efficiency

Evaluating the efficiency of APR systems could provide users with practical instructions on how to make a cost-performance balance when utilizing them in practice. Generally, a new APR tool is evaluated by counting the number of correct fixes within well-established benchmarks (e.g., Defects4J [32]). For non-learning APR approaches, researchers execute them on each bug, until reaching the terminate condition (a test-adequate patch generated or reaching the set execution timeout). Such evaluation could provide a comparison of efficiency between different APR systems. If system A can achieve comparable performance with less time than B, we can say that it has higher efficiency. However, how the single system's performance changes under different

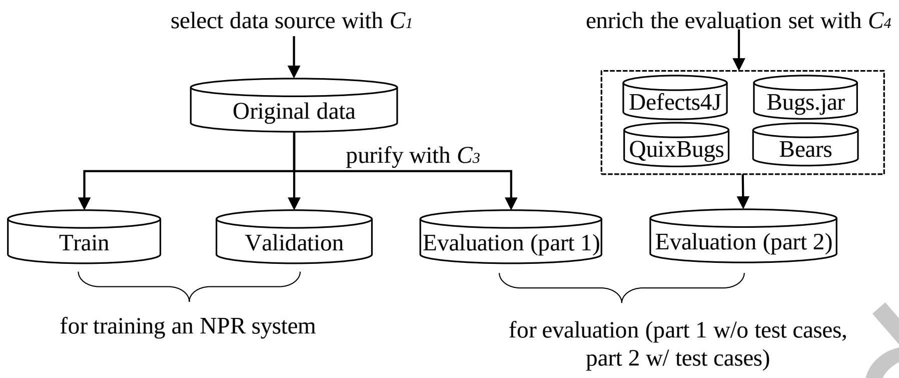
select data source with $C_1$
enrich the evaluation set with $C_4$
Original data
Defects4J
Bugs.jar
QuixBugs
Bears
purify with $C_3$
Train
Validation
Evaluation (part 1)
Evaluation (part 2)
for training an NPR system
for evaluation (part 1 w/o test cases, part 2 w/ test cases)

Fig. 2. Construction process of our benchmark data.

execution costs is still unknown (e.g., using different times out). Users still have no idea how to set such external parameters when utilizing existing systems.

In this paper, we name "the performance change of APR systems under different costs" as the **Efficiency**. We focus on evaluating NPR systems since they have become popular in recent years. Our evaluation concentrates on two aspects:

(1) **NPRs' performance change when using different candidate numbers.** As illustrated before, when utilizing an NPR system, the candidate number should be set first, then the system generates as many candidates as the set number. The costs of executing NPR systems and executing test cases for validation largely depend on the candidate number. A large candidate number requires more machine resources for patch generation and more candidates to validate. In a nutshell, users could control the costs by setting the candidate number. Thus, in this paper, we aim to investigate how NPRs' performance changes when using different candidate numbers.

(2) **NPRs' performance change when using different validation strategies.** Generally, when evaluating new APR systems, researchers only examine the first test-adequate patch. However, it doesn't mean that the next or later patches can't fix the bug. We understand the rationale of researchers since a new APR system would be evaluated on benchmarks with hundreds of bugs. Large-scale manual validation is a huge cost then. But in practice, users have the choice to check more than one plausible patch for a higher probability of fixing the bug. Thus, in this paper, our aim is to provide users with instructions on how the repair performance changes when the manual validation strategy changes (validating top-k plausible patches).

## 3 NPR4J: BENCHMARK FOR TRAINING AND EVALUATING NPR SYSTEMS ON JAVA

### 3.1 Dataset Construction

To run our empirical experiments, we construct a new benchmark dataset instead of reusing any of the existing datasets. The reasons are two-fold. First, to avoid data leakage, samples related to bugs in the evaluation set should be excluded from the training dataset. However, existing training data that the previous NPR systems used may not provide enough meta information to do the exclusion. Second, some existing datasets may pose potential threats to experimental performance due to some issues with data. For example, although CoCoNut [48] reports that the training set they use (Java 2006) contains over three million samples, we find that nearly one-third of the samples are duplicated pairs and non-character changes, through checking their raw data by string matching.

We argue that for mitigating threats brought by data issues, the experiment training and evaluation data should satisfy the following criteria:

- *Criterion #1: Each data sample from the data source should provide enough meta information.*This criterion is designed for selecting a data source. The data source we used in this experiment must provide enough meta information to ensure data traceability, including commit sha, commit message, and class-level context. For NPR, such traceability is important. For example, meta information such as the repository of bugs should be provided to exclude bugs from the training data that belong to the same repository as bugs for evaluation. And commit messages are required to filter non-bug-fixing code changes.
- *Criterion #2: Evaluation-related Bugs should not appear in the training set.* Considering that codes within the same projects may contain some cheating information that is inaccessible for the NPR systems under empirical scenarios, we perform a strict strategy, dividing data by projects.
- *Criterion #3: Each bug in the evaluation set should be attached to a corresponding issue or bug report.* This criterion is to ensure that the bugs we use for evaluation are highly reliable samples from the real world.
- *Criterion #4: The benchmark should be peer-reviewed and contain human-written patches for each bug.* For perfect fault localization, we need human-written patches to locate the buggy position of the source program.

With the guidance of the above four criteria, we construct a new benchmark dataset called *NPR4J-Benchmark*, following the process shown in Figure 2. The overall process can be summarized into the following three steps:

**Step 1: Selecting the data source.** It is to collect enough bug-fix pairs. A common way is to crawl large amounts of data from the code repositories such as GitHub. However, it can be laborious and time-consuming. Fortunately, previous studies have done this work [8, 38, 48, 58, 61, 74, 97]. To select proper data sources from them, we obey $C_1$. Among the four data sources, we select the source of BFP [74] as our original data source, which contains 787,178 bug-fixing commits for Java. Each data sample is attached with meta information including repository, commit message, and commit URL.

**Step 2: Splitting sub-datasets.** This is to construct three sub-datasets: *training*, *validation*, and *evaluation*. The training set contains the samples that the NPR systems rely on to learn to fix bugs. During the offline training process, the NPR systems can get an early evaluation of the validation set to modify hyper-parameters of repair models. Eventually, the evaluation set is used to measure the performance of trained the NPR systems. One well-known basic rule is that there should be no overlap between data in the training, validation, and evaluation sets. We obey $C_2$ for the splitting phase. Concretely, we follow the practice by Recoder [97], excluding data samples that belong to a clone project of projects that evaluate data uses or a program repair project that uses these projects from the training set.

**Step 3: Purifying and enriching evaluation resources.** To construct a more diverse benchmark for evaluation, we collect bugs from multiple sources. The first source is the data source selected in the first step. According to previous research [29], bug-fix commits without peer-reviewing may contain bug-irrelevant changes and some of the commits are of low quality. Thus, to ensure the reliability and quality of bugs in the evaluation set, we follow $C_3$ to purify bugs. Firstly, two authors randomly select 200 bug-fix commit messages and perform a manual analysis to see what common textual patterns the bugs satisfying $C_3$ have in their commit messages. We find that such bugs usually indicate the specific information of the Issue in the commit message with a text pattern of "Issue/Bug" + ":/#" +NUMBER. Then, We perform a regular match on the commit message of each bug-fix commit and only keep the samples whose commit message matches the pattern. Next, we select the existing benchmarks in APR as additional data sources, following $C_4$. With the guidance of $C_4$, Defects4j [32], Bugs.jar [68], QuixBugs [39] and Bears [49] are added to our evaluation set. Finally, to avoid the fairness issue caused by data leakage, we use string comparison to compare the data in the evaluation and the training set and remove the overlapping samples from the training set.

Table 1. Statistics of *NPR4J-Benchmark*.

| **Source** | **Training** | **Validation** | **Evaluation** | |
|---|---|---|---|---|
| | | | **Diversity** | **Empirical** |
| BFP [74] | 144,641 | 13,739 | 12,815 | — |
| Bears [49] | — | — | — | 36 |
| Bugs.jar [68] | — | — | 480 | — |
| Defects4J [32] | — | — | — | 331 |
| QuixBugs [39] | — | — | — | 34 |
| Total | 144,641 | 13,739 | 13,295 | 391 |

The statistics for the benchmark dataset are listed in Table 1. In total, we collected 144,641 bug-fix samples in the training set and 13,739 in the validation set. Each sample consists of six types of information: (1) the buggy line, (2) the fixed lines, (3) method-level context, (4) class-level context, (5) fault location, and (6) meta information. For evaluation, we collected 13,706 bugs in total. Specifically, our evaluation set consists of two parts: Diversity and Empirical. Diversity provides 13,295 bugs (12,815 bugs from BFP [74] and 480 bugs from [68]). For empirical validation, we collect 411 bugs (119 bugs from Bears [49], 260 bugs from Defects4J [32] and 32 bugs from QuixBugs [39]) with corresponding test suites in the empirical part.

## 3.2 Framework Implementation

Although most NPR systems open source their codes, we notice that there exist some usability issues proposed in their GitHub repositories (For example, there are open issues concerning the evaluating procedure, data availability, and hyper-parameter settings, etc., of CoCoNut[1]). One major reason is that codes of the NPR system are often very complex. For example, their codes are often made up of multiple programming languages (i.e., Java for code preprocessing and Python for deep learning). The Python part is implemented with various deep learning frameworks (e.g., PyTorch, Tensorflow , OpenNMT, and Fairseq, etc.) [2]. Thus, modifying their codes to satisfy new requirements (i.e., training on a new dataset) requires users' expertise in both program repair and deep learning fields. To ease our experiments and benefit future research on NPR field, we wrap original codes of nine NPR systems (Tufano [74], SequenceR [4], CoCoNut [48], Codit [3], Edits [8], Recoder [97], CodeBERT-ft [54], UniXCoder-ft [21], CodeT5-ft [21] ) into a new framework, called *NPR4j-Framework*. The overall workflow of *NPR4j-Framework* is shown in Figure 3. *NPR4j-Framework* supports training NPR systems from scratch and using trained systems to fix bugs empirically. All codes are organized into three main components: DataManager, Preprocessor, and Trainer. DataManager is responsible for processing the original data into a standard data form which we named, diff. The core part of the DataManager is the DiffParser. The DiffParser parses each pair of buggy and fixed Java class files into diffs, each of which represents the exact line-level difference between the buggy and the fixed program, and is associated with the position of the buggy line. The preprocessor implements a variety of processing methods that are designed by the six NPR systems to represent the buggy code. It consists of a Tokenizer and an Abstractor. The tokenizer is responsible for dividing textual codes into a list of tokens. The Abstractor supports common code abstraction operations that most NPR systems use to simplify the codes, such as replacing the string elements with constant IDs. Once the preparation of data ends, the Trainer is pressed into service to train the NPR model. During this phase, all parameters such as the hyperparameters of the model should be written on a configuration file. Then, the configuration file will be delivered to the Vocab Builder and

[1]https://github.com/lin-tan/CoCoNut-Artifact/issues
[2]OpenNMT: https://opennmt.net/, Fairseq: https://github.com/pytorch/fairseq

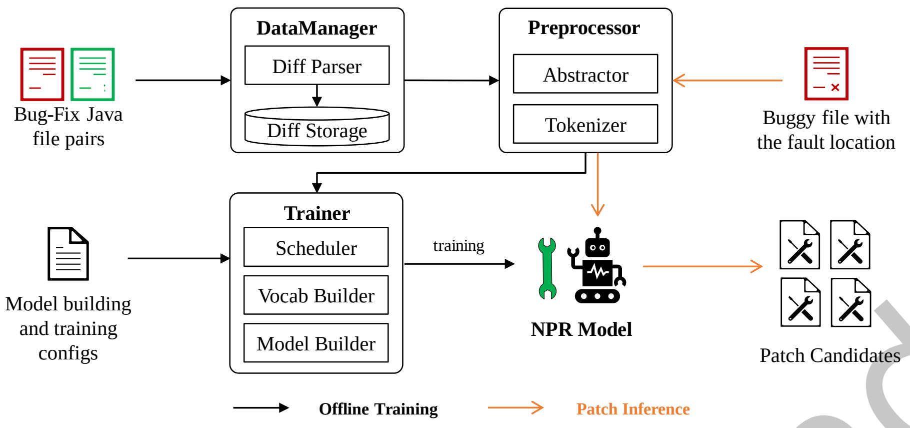

Fig. 3. The workflow of *NPR4J-Framework*

the Model Builder to configure the vocab and initialize the model for training. The Scheduler determines when to stop the training.

Besides training and evaluating an NPR system with the data provided in *NPR4J-Benchmark*, the framework also allows users to train the NPR systems on their datasets. Users just need to provide pairs of buggy-fix Java class files and use the DataManager and the DiffParser to process the data. What's more, our framework is extensive. The well-defined interface makes it easy to add a new NPR system.

## 4 RESEARCH DESIGN ON INVESTIGATING EFFECTIVENESS AND EFFICIENCY

This section introduces detailed experiment design and setup for research questions on efficiency and effectiveness.

### 4.1 Research Questions on Effectiveness

**RQ1.1:** *What are the repairability of NPRs across different bug types?*

**Objective:** Current evaluation protocol mainly compares various NPR systems through the fixed bugs within well-established datasets such as Defects4J [32]. However, users may have more detailed requirements when utilizing NPR systems in practice. For example, they may concentrate on certain types of bugs (e.g., incorrect condition statements). To fulfill their requirements, an NPR system that performs well on fixing condition statements should be selected. However, from the mere number of totally fixed bugs, users can not have a comprehensive understanding of how each NPR system performs on different types of bugs. Questions like *"To what extent could current NPR systems handle simple or harder bugs?"* or *"To what extent could current systems handle bugs that involve project-specific knowledge?"* can not be answered yet. In response, We aim to explore the effectiveness of NPR systems by investigating the kind of bugs that can be well-handled by existing NPR systems through this research question.

**Method:** First of all, we need a bug taxonomy that reflects the complexity of generating patches. Although there have been some widely used bug classification formalisms such as Orthogonal Defect Classification (ODC) [5] and Common Weakness Enumeration (CWE) [50], they are not suitable for our purpose. Their definitions of bug categories are usually more complete, include more bug attributes, and require experts to classify them accurately. In our experiment, we only need to classify bug-fix pairs at the granularity level of code changes to examine NPR's performance in detail. And it would be better if the bug classification costs could be lower.

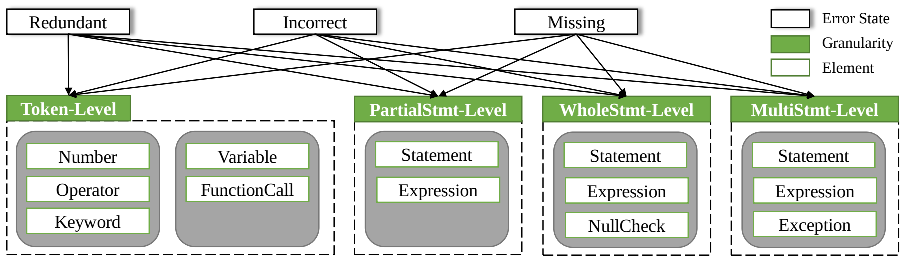

Fig. 4. The bug taxonomy used in our study.

Therefore, we adapt to a lighter bug classification proposed by Ni et al. [62]. Based on it, we design a new bug taxonomy from the perspective of *ErrorState* and *ErrorElement*, as shown in Figure 4:

(1) *ErrorState*: Errorstate refers to the reason why the error code is incorrect compared with the patch, The reasons include *Redundant*, *Incorrect* and *Missing*. Correspondingly, the repair actions each error state needs are respectively *Delete*, *Replace,* and *Insert*.
(2) *ErrorElement*: ErrorElement is the concrete buggy part of the source code. We classify all elements into four grains: ① *Token-level*, which includes singleton tokens such as the operator and the keyword, ② *PartialStmt-level*, which includes part of a statement or expression, such the composition of an operator and a variable, ③ *WholeStmt-level*, which involves changes to a whole expression or statement and ④ *MultiStmt-level*, which involves changes to multiple expressions or statements.

In our taxonomy, a bug can be regarded as a combination of *ErrorState* and *ErrorElement*. Based on the proposed taxonomy, we classify every bug in the evaluation set. Then, we count the number of correctly fixed bugs that belong to each concrete type of bug for each NPR system. Finally, we analyze the repairability of each NPR system considering a coarse grain of *ErrorState* and *ErrorElement*.

**Evaluation Metrics:** We use four metrics in *RQ1.1*: (1) the number of correct fixed bugs and the fix rate of each NPR system for different types of bugs, (2) the number of correctly fixed bugs and the fix rate of each NPR system on bugs considering different error elements, (3) the number of correctly fixed bugs and the fix rate of each NPR system on bugs considering different error states.

**RQ1.2:** *To what extent can NPR systems be replaced by each other or previous non-learning approaches?*

**Objective:** When a new APR system is published, whether it can fix a new bug that can not be fixed before is also an important point to measure its significance [85]. As is known, neural networks are more expansive than non-learning approaches in terms of computational costs. Therefore, in practice, choosing whether an NPR system should be used to replace the previous non-learning tools may also require the knowledge of how much new improvement can be brought. This question aims to investigate the extent to which NPR Systems can be replaced by each other, or cost-effective approaches such as simple templates and G&V APR approaches. If most fixes of an NPR system can be covered by simpler approaches, we need to re-examine whether it is necessary to use an NPR system. Also, we want to see what kind of fixes could each NPR system uniquely generate.

**Method:** To answer *RQ1.2*, we compare NPR systems' overlapping fixes with two kinds of APR approaches: **① different NPR systems**, **② previous non-learning APR systems**.

Firstly, we compare the selected nine NPR systems by counting their overlapped and unique fixed rate. We aim to see how NPRs' performance differs when fed the same training data. Secondly, we try to cover as many APR tools as possible, however, the costs of executing them are unaffordable for us. Thus, we adopt their published repair results on Defects4J for comparison, since Defects4J is a part of our evaluation set. For fairness, NPR and APR tools should maintain the same FL settings (in our experiments we use the perfect fault localization for all NPR tools). The selection criterion for non-learning-based APR tools is: *the tool should have published repair results on Defects4J under the perfect fault localization setting*. We collect 23 candidate APR tools [11, 14, 24, 25, 34, 35, 40, 42, 43, 52, 69, 77, 82–84, 90, 91] according to the APR survey [57] and some empirical studies [9, 44, 46] and filter 7 of them [14, 24, 35, 69, 77, 83, 91]. We compare NPR's fix results with 16 APR tools in total: ACS [82], SimFix [25], TBar [43], Fixminer [34], kPAR [40], AVATAR [42], Nopol [84], DynaMoth [11], ARJA [90], Cardumen [53], GenProg-A [52], jGenProg [52], jKali [52], jMutRepair [52], Kali-A [90], RSRepair-A [90]. We collect their repair results and compare them with NPRs' fixes on Defects4J v1.

**Evaluation Metrics:** In *RQ1.2*, we use four metrics to measure the difference between APR systems: (1) the overlapping and unique fix rate of each NPR system, (2) the number of NPR systems' fixes that are covered by non-learning G&V approaches.

## 4.2 Research Questions on Efficiency

**RQ2.1:** *How does the repairability of each NPR system vary with the number of candidates?*

**Objective:** *RQ1.1* focuses on the efficiency of NPR systems with respect to the machine evaluation efforts for the patch validation phase. As mentioned in Figure 1, an NPR system generates a set of candidates according to the set beam size. Then, all candidates need to execute the test suite. If the NPR system has high efficiency, it can rank the correct patches higher, which in turn implies that it can be deployed with a smaller number of candidates. As a result, the effort cost of executing the test suite can be reduced.

**Method:** To answer *RQ1.1*, we first investigate the repairability of the selected NPR systems by computing the number of correctly fixed bugs and the average ranks of the first correct patches. Then, we analyze the distribution of correct patches generated by each NPR system. Finally, we determine how performance changes as we alter the number of candidates.

**Evaluation Metrics:** We use three metrics to answer *RQ2.1*: (1) the number of correctly fixed bugs, (2) the average rank of the first correct patches, and (3) the distribution of correct patches over all bugs.

**RQ2.2:** *How does the repairability of each NPR system vary with the number of plausible patches?*

**Objective:** *RQ1.2* focuses on measuring the efficiency of NPR systems with respect to human evaluation efforts for the patch validation phase. As mentioned in Figure 1, all plausible (test-adequate) patches need manual validation to ensure their correctness. In the normal patch validation protocol, APR systems stop when encountering the first plausible patch or reaching the set timeout. It is reasonable for some G&V approaches since they are designed to continuously generate patches if no terminate signal is given. However, for NPR systems, the mode is "generate all candidates as set, then validate them". Once the candidate size is set, the system will generate candidates as the set number. So there is less need to give a terminate signal like "stop when encountering the first test-adequate" patch. It means that in practice, it's easier to set NPR systems for generating more than one plausible patch. If an NPR system can have significant performance when considering more than one plausible patch, it may also be a considerable option to validate a bit more patches manually. Our experiments, to support the aforementioned options, aim to show how the performance of selected NPR systems changes when considering different numbers of plausible patches.

**Method:** To answer this question, we measure the repairability of NPR systems under different manual evaluation strategies. Concretely, we count the number of correct/test-adequate patches when evaluating the top-1, top-2, top-5, top-10, and all plausible patches. In addition, we count the number of manual validations required to fix a bug under different strategies.

**Evaluation Metrics:** We use two metrics to answer *RQ2.2*: (1) the number of correct/plausible patches and (2) the precision of NPR systems within the top-k plausible patches.

Table 2. Selected NPR Systems

| Selected | Reason | NPR Systems |
|---|---|---|
| No | Not Public | Tang |
| | Failed to Train | CURE, DEAR, DLFix, CODIT |
| | Not Supervised Training | SelfAPR, Incoder, CodeX, GPT-series |
| Yes | Public and Trainable | Edits, Tufano, CoCoNut, Recoder SequenceR, CodeBERT-ft Codet5-ft, UniXCoder-ft |

Table 3. Basic information of selected NPR systems

| Style | NPR system | Input Scope | Model Base |
|---|---|---|---|
| NMT-based | Edits [8] | line | Transformer [75] |
| | Tufano [74] | method | GRU [6] |
| | CoCoNut [48] | method | CoCoNut |
| | SequenceR [4] | class | LSTM [19] |
| | Recoder [97] | class | Recoder |
| LLM-based | CodeBERT-ft [54] | line | CodeBERT [12] |
| | RewardRepair [88] | class | T5 [67] |
| | CodeT5-ft [21] | method | CodeT5 [76] |
| | UniXCoder-ft [21] | method | UnixCoder [18] |

## 4.3 Subject NPR Systems

Our study focuses on NPR systems targeting Java programs since Java is the most popular programming language and has been widely researched by the program repair community. To determine which systems should be included in our study, we first search the popular databases (e.g., IEEE Xplorer, ACM digital library, and SpringerLink, etc.) with keywords "neural/translation/learning program/bug repair/fix". Then, we use an APR-related website [3] and a literature review of APR [57] to supplement the search results. Additionally, the NPR systems considered in our study are selected based on the following criteria:

(1) *Availability*: Our study involves the training and inference of NPR systems, thus NPR approaches without publicly available tools are excluded.
(2) *Trainability*: Some NPR tools provide trained models in the code repository, but lack the necessary code to retrain the model. We exclude such approaches from our study since we need to retrain NPRs on the same training data for a fair comparison.
(3) *Supervised Training Requirement*: Our study focuses on NPR models that require supervised training, including NMT-based models and LLM-based models that need fine-tuning. Models requiring self-supervised training or can be directly used with zero-shot or few-shot samples are not included.

Guided by the above criteria, we select nine NPR systems[4] ([4, 8, 21, 48, 54, 74, 88, 97]), as shown in Table 2. Despite our best efforts to include all eligible systems, we excluded some NPR systems due to code reproduction costs. As shown in Table 2, six NPR systems [2, 3, 27, 28, 37, 73, 98] are excluded since we have problems on re-training them. Tang's model [73] is not open-sourced. CURE [28] and Dear [37] lose some crucial code files to support training. KNOD [27] and Tare [98] do not provide the codes for pre-processing training data so we can't handle our own dataset to train them. For each model, we have tried our best to reproduce these models and

[3]http://program-repair.org
[4]Due to time and cost limitations, we only consider NPR published before September 2023.

send e-mails to the authors for help. The other four systems [26, 78, 79, 86] are excluded since they don't require a supervised training progress.

We only list the basic information of the selected NPR systems in Table 3, including their inputs, outputs, and model bases. In the following part, we will briefly describe the selected models.

**Tufano** [74] trains an RNN-based Encoder-Decoder model which is able to translate the entire buggy method into a fixed version. To reduce the difficulty of learning, they abstract identifiers and literals in the buggy code to simplify the input and output. During the abstraction process, the source code is first fed to a Java parser, which recognizes the identifiers and literals in the stream. Then the parser generates and substitutes a unique ID for each identifier/literal within the buggy context. Only the most frequent words are kept. Such abstraction reduces the model's choices when generating patches, therefore increasing the patching rate.

**SequenceR** [4] fixes bugs based on sequence-to-sequence learning on source code. Compared with Tufano's model [74], it uses an additional copy mechanism to overcome the unlimited vocabulary problem that occurs in handling big code. The model takes the abstract context of the buggy line as the input and predicts the fixed result of that line. The abstract buggy context consists of line-, method-, and class-level information of the buggy line. Specifically, at the line level, special tokens are inserted before and after the buggy line to indicate the location of the bug. information. Then, the remainder of the buggy method is kept in the representation. Finally, all the instance variables and initializers, along with the signature of the constructor and non-buggy methods from the buggy class are added to the input unless reaching a truncation limit.

**CoCoNut** [48] uses ensemble learning on the combination of Convolutional Neural Networks (CNNs) and a context-aware neural machine translation architecture to automatically fix bugs for multiple programming languages. To better represent the context of a bug, it introduces a new context-aware architecture that represents the buggy source code and its surrounding context separately. Such architecture enables the model to distinguish the buggy line and the context better. To reduce the size of vocabulary, it leverages world-level tokenization by considering underscores, camel letters, and numbers as separators.

**Edits** [8] models the patch generation process as performing token-level insertion and deletion operations on the buggy code. Edits adds the two additional pointers to Transformer[75], implementing the editing operation of "insertion" and "deletion". Edits takes the buggy code line as the input. When generating patches, it outputs edit operations on the buggy code, each indicating the insertion start position, the removal end position, and any missing tokens.

**Recoder** [97] designs a syntax-guided decoder to generate edits on the AST of the buggy method. It takes (1) AST traversal sequence, (2) Tag Embedding, and (3) AST-based Graph as inputs. During inference, Recoder uses a novel provider/decider architecture to output edit operations on the current AST. It uses three providers to represent insertion, modifying, and copying operations. Providers are responsible for providing choices for expanding a non-terminal of the current AST. The decider estimates the likelihood of using each provider.

**CodeBERT-ft** [54] is an early LLM-based APR work. It leverages CodeBERT [12], a bimodal pre-trained language model for both natural and programming languages as the model base, and fine-tune it with single-statement-level code changes from the ManySStuBs4J dataset [33].

**RewardRepair** [88] uses a sequence-to-sequence style model T5 [67] as the backbone model. Besides, it improves NMT-based program repair with a loss function based on program compilation and test execution information, rewarding the network to produce patches that compile and that do not overfit. For code representation, RewardRepair follows CoCoNut [48] to represent the buggy code and context code as two separate token sequences and enriches the context as SequenceR [4].

**CodeT5-ft** [21] leverages CodeT5 [76], an identifier-aware unified pre-trained encoder-decoder model for code understanding and generation as the backbone model. The authors empirically compare the impact of four kinds of code representation methods, abbreviated as CR1, CR2, CR3, and CR4, considering different factors such as the input-output context and the abstract. In this paper, since our goal is to compare the state-of-the-art NPR systems, we only choose the best-performed model (fine-tuned CodeT5 with CR4).

**UniXcoder-ft** [21] leverages UniXcoder [18], a unified cross-modal pre-trained model for programming language as the backbone model. The model utilizes mask attention matrices with prefix adapters to control the model behavior and leverages cross-modal contents like AST and code comment to enhance code representation. Same as to CodeT5-ft [21], we choose the best-performed model (fine-tuned UniXcoder with CR4).

## 4.4 Experimental Setup

This section introduces the experimental settings of all our experiments:

**Dataset:** Since the performance of neural networks is sensitive to the amount of data they are trained on [36], we retrain them on the same training set to ensure the fairness of the comparison. Our training data is derived from the source of BFP [74], which consists of 144,641 samples. We empirically evaluate the trained systems on three popular benchmarks: Defects4J [32], Bears [49], and QuixBugs [32]. To avoid data leakage, We exclude samples from the training set that belong to a project that is the same as the samples in the evaluation set.

For evaluation, we focus on the **one-hunk bug-fixing scheme**. In the StandUp4NPR paper[94], we focus on the one-line bug-fixing scheme since it's the most popular research scheme in the APR field. However, recent LLM-based tools have shown their repairability on one-hunk bugs. Thus, we extend our evaluation set to one-hunk bugs. Finally, we obtained 391 bugs for empirical evaluation, including 331 bugs of Defects4J [32], 34 bugs of QuixBugs [39], 36 bugs of Bears[5] [49].

**Fault Localization:** Following the setting used in previous NPR research, we use a line-level restricted fault localization [41] setting in our experiment, which means the precise fault line of the buggy file is given. We do this by comparing buggy codes with developer patches.

**Training and Parameter Tuning:** We use the codes released by original papers to train the selected NPR systems, and apply the early stop strategy [63] to decide when to stop training. Besides, for parameter tuning, we perform a random search on adjustable parameters such as the learning rate, batch size, and the number of hidden layers. Since CodeBERT-ft [54] and RewardRepair [88] are pre-training-based models, we fine-tuned them with our data. The other NPR systems are trained from scratch.

**Number of Candidates:** We set the beam size and the number of candidates to 300 for each NPR system. Note that CoCoNut [48] ensembles 10 models. Since our goal is to make all models generate an equal number of candidate patches, we take the first 30 candidates from each model for CoCoNut.

**Patch Validation:** A common patch validation strategy of automated program repair is to stop validation when encountering the first plausible patch. In our experiment, since we aim to explore the efficiency of NPR systems, we validate all plausible patches.

During the patch validation progress, all test-adequate patches require a manual check to determine correctness. According to previous research ([51, 87]), this manual process can introduce some biases. In this paper, to minimize such biases, we resort to the patch assessment process proposed by Liu et al. [47], where the authors summarize 15 rules that define the semantic equivalence of code snippets. To improve coverage, we define five additional rules (shown in Table 4). Patches are considered to be correct if and only if they obey one of these 18 rules.

In total, our experiment produced 12053 plausible patches. To handle such a large number of patches, we employ six graduate students in computer science who are familiar with Java to help us check the correctness of the plausible patches. We set a time limit of 20 minutes per check. For each candidate patch, students need to judge whether it confirms one of the rules and reject it otherwise. Each candidate is checked by at least two

[5]The original Bears benchmark contains 251 bugs. However, many bugs have rotten with dependencies having disappeared (in particular snapshots) and version problems. The benchmark as of Sep 2023 contains 118 bugs.

Table 4. Rules used to confirm the semantic similarity between tool-generated and developer-provided patches.

| Rule | Rule-Name | Description | Example (developer_patch) | Example (APR-generated patch) |
|---|---|---|---|---|
| **R0** | Identical Patch | The patch generated by an APR tool is identical to the developer's patch except for the diffs in format and comments. | while (lo < hi) { | while ( ( lo ) < hi ) { |
| **R1** | Different fields with the same value (or alias) | Different fields with the same value (or alias) | for (Integer count : counts) { | for ( int count : counts ) { |
| **R2** | Same exception but different messages | The APR-generated patch has the same exception as the developer's patch but does not specify the identical and concrete message. | throw new NumberFormatException(str +" is not a valid number."); | throw new NumberFormatException(); Variable |
| **R3** | Variable initialization with new rather than a default value | The patch is generated by using a new initialized variable but not a default variable in the program, the two variables return the same value. | if (str == null) str = ""; | if (str == null) str = new String(); |
| **R4** | if statement instead of a ternary operator, or the contrary | The APR patch is implemented by inserting a new if statement but not a ternary operator to fix the bug. | classes[i] = array[i] == null ? null : array[i].getClass(); | if (array[i] == null) continue; classes[i] = array[i].getClass(); |
| **R5** | Unrolling a method | All donor code of fixing a bug is rolled in a method, the patch unrolls the method but not invokes the method. | abs(a); | if (a < 0) a=-1; |
| **R6** | Replacing a value without a side effect | The buggy variable is not replaced, its value is replaced with the correct one without other side effects. | int g = (int) ((v - this.lowerBound) / (this.upperBound | value = Math.min(v, this.upperBound); - |
| **R7** | Condition Area Equal | The patch enumerates and negates the condition that is positively considered in the buggy code. | while (lo < hi) { | while( lo <= hi +1&& lo < hi ) { |
| **R8** | Unnecessary code uncleaned | The unnecessary code is not removed in the patch, such code will not be executed at the end or impact the execution of the patched program. | if (i < maxCode) { | ObjectCodec codec; if (i < maxCode) { |
| **R9** | Return earlier instead of a packaged return | The patch checks the value of a boolean variable before other related boolean variables and returns the related value if the condition is (not) satisfied. Developers however prefer to package the boolean variable with others and have a single return. | return foundDigit && !hasExp && !hasDecPoint; | if (hasDecPoint==true){return false;} |
| **R10** | More null checks | The APR-generated patch consists of more null checks than the corresponding developer's patch. | if (searchList[i] == null \|\| replacementList[i] == null) | if(noMoreMatchesForReplIndex[i] \|\| searchList[i]==null \|\| searchList[i].length()==0 \|\| replacementList[i]==null) |
| **R11** | Additional unneeded check | The APR-generated patch considers unneeded check but such check is already considered in the context of non-buggy code. | while (lo < hi) { | while( lo < hi && lo >=0) { |
| **R12** | Partial code is not included | The patch fixes a partial code but passes all tests, and the patch code is identical to the partial code of the developer's patch. | if (str.trim().startsWith("−")) throw new NumberFormatException(str + "is not a valid number."); | if (str.startsWith("−")==true)throw new NumberFormatException(); return null; |
| **R13** | Less accurate comparison | The patch code is not compared with the same accurate value as the code of the developer's patch. | return FastMath.pow(2 * FastMath.PI, -0.5 * dim) * | return FastMath.pow(2 * FastMath.PI, -dim / 2d) * |
| **R14** | the field but not its getter | The patch uses a field but not a getter method that returns the field. | return cAvailableLocaleSet.contains(locale); | return cAvailableLocaleList.contains(locale); |
| **R15** | Un-actionable code but not removing them | The APR-generated patch does not remove the buggy code but makes it un-actionable (it will never be executed). | buggy_line_need_remove | if (false) buggy_line_need_remove |
| **R16** | Equal value of operation expression | Expressions involving operations return the same value | return levenshtein(source.substring(1), target.substring(1)); | return (1 * levenshtein(source.substring(1), target.substring(1))); |
| **R17** | Add Cast Type | Cast is added, but the code is not affected | return levenshtein(source.substring(1), target.substring(1)); | return (int)levenshtein(source.substring(1), target.substring(1)); |
| **R18** | Rolling | Compress a long code into an equivalent short code | CtExpression<T> returnedExpression = ret.getReturnedExpression(); returnedExpression.setParent(null); return returnedExpression; | return ret == null ? null : ret.getReturnedExpression( ) ; |
| **R19** | DefaultParam | Some parameters are deleted, but they have default values, which does not affect the correctness of the function | encodeBase64(binaryData, false) | encodeBase64(binaryData) |

checkers, and if either one checker claims the candidate is not correct, it will be rejected. Finally, we got 2892 correct patches in total.

## 5 EXPERIMENT RESULTS

### 5.1 RQ1. Effectiveness

**RQ1.1:** *What are NPR systems' repairability across different types of bugs?* To answer this question, we first classify the bugs of three benchmarks with the bug taxonomy proposed in Figure 4. Then, we count the number of bugs repaired by each NPR system on different types of bugs. Besides, we analyze the repairability of NPR systems when considering edit scope and edit action types.

**Finding 1: Even the SOTA NPR system currently handles only limited types of bugs well, such as simple bugs related to keywords and operators. In addition, the repair ability of NPR systems on different types of bugs also varies significantly.**

Table 5 displays the number of different types of bugs generated by each NPR system. **Firstly**, we find they are good at fixing simple bugs but poor at fixing complex bugs that require modification with multiple expressions or statements. For example, more than 80% of bugs with type *IncorrectKeyword* and *IncorrectOperator* can be fixed

Table 5. Repairability of NPR systems on different types of bugs.

| Bug Type | # Bugs | NMT-based | | | | | LLM-based | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Recoder | CoCoNut | SequenceR | Tufano | Edits | CodeT5 | UniXCoder | CodeBERT | RewardRepair |
| MissingMultiStatement | 129 | 0 | 0 | 0 | 0 | 0 | **3 (2%)** | 1 | 1 | 0 |
| MissingWholeExpressionOrStatement | 42 | 0 | 0 | 1 | 0 | 0 | **3 (7%)** | 2 | 0 | 1 |
| IncorrectWholeExpressionOrStatement | 28 | **5 (18%)** | 0 | 1 | 0 | 0 | 4 | **5 (18%)** | 0 | 4 |
| IncorrectVariable | 27 | **13 (48%)** | 8 | 11 | 5 | 0 | 9 | 8 | 8 | 12 |
| MissingBooleanExpression | 23 | 1 | 0 | **4 (17%)** | 1 | 0 | 2 | 2 | 2 | 1 |
| IncorrectPartialExpressionOrStatement | 18 | 1 | 0 | 0 | 0 | 0 | **3 (17%)** | **3 (17%)** | 1 | 2 |
| IncorrectOperator | 17 | 12 | 13 | **17 (100%)** | 14 | 9 | 13 | 13 | 16 | 12 |
| IncorrectFunctionCall | 17 | 5 | 5 | 5 | 2 | 1 | 6 | 4 | 3 | **7 (41%)** |
| MissingPartialStatementOrExpression | 13 | 2 | 3 | 5 | 2 | 1 | **7 (54%)** | 6 | 3 | 4 |
| IncorrectNumberValue | 12 | 2 | 4 | 6 | 1 | 2 | 2 | 3 | **7 (58%)** | 4 |
| MissingNullCheck | 11 | 0 | 2 | 2 | 1 | 1 | 4 | **5 (45%)** | 4 | 1 |
| MissingFunctionCallOrConstructor | 11 | 1 | 1 | **2 (18%)** | 0 | 1 | **2 (18%)** | 1 | 0 | 0 |
| RedundantExpression | 10 | 5 | 2 | **6 (60%)** | 1 | 1 | 3 | 3 | 5 | **6 (60%)** |
| MissingVariable | 7 | **3 (43%)** | 1 | 1 | 0 | 0 | 2 | 2 | 1 | 1 |
| IncorrectMultiStatement | 7 | 0 | 0 | 0 | 0 | 0 | **1 (14%)** | **1 (14%)** | 0 | 0 |
| MissingMultipleBoolean | 6 | 0 | 0 | 0 | 0 | 0 | **1 (17%)** | **1 (17%)** | 0 | 0 |
| ExpressionSwap | 5 | 3 | 1 | 2 | 0 | 2 | **4 (80%)** | **4 (80%)** | 2 | **4 (80%)** |
| IncorrectKeyword | 5 | 3 | 3 | 2 | 2 | 3 | **4 (80%)** | **4 (80%)** | **4 (80%)** | 3 |
| MissingKeyword | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MissingException | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Sum | 391 | 56 | 43 | 65 | 29 | 21 | **73** | 68 | 57 | 62 |

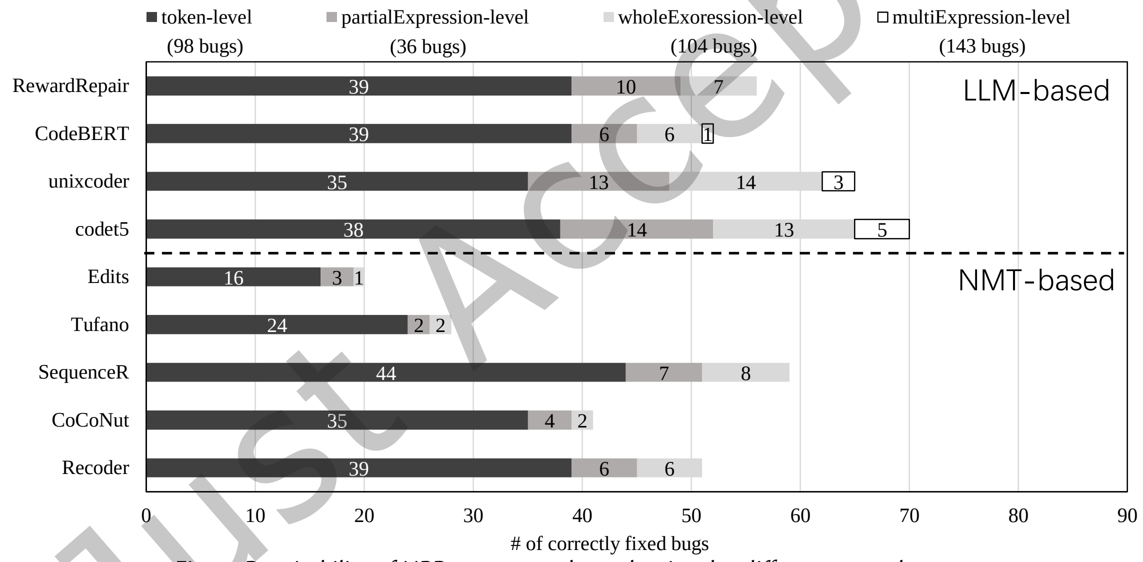

Fig. 5. Repairability of NPR systems on bugs that involve different error elements.

by SOTA NPRs, while on more complete bugs with types *MissingMultipleExpression*, *IncorrectMultipleExpression*, *MissingMultipleBooleanExpression* and *MissingException* the fix rate decreases to merely 2%. Noting that 40% bugs in the evaluation dataset involve modification on multiple statements, we think there should be clearer discussion and evaluation of NPRs' practicality:

***Suggestion 1**: Investigating what types of bugs are more frequent and important and enable NPR to break through in these specific scenarios.*

A big advantage of these NPR systems is that they can produce an output simply by providing bug code as input, but that doesn't mean they will produce the correct output for every input. Today's NPR tends to consider the more general bug-fix scenario in the context of data collection and performance evaluation scenarios. However, as mentioned in Finding 1, the reality is that NPRs can not succeed in general bug-fix scenarios even when trained with general bug-fix data. The Bug is not a concept of nature, but a product of the real world. Therefore, considering the practicality of NPR, we believe that a more comprehensive survey is needed to investigate high-frequency bugs in real scenarios and let NPR make priority breakthroughs in these high-frequency scene bugs. **Secondly**, We find that the types of bugs that each NPR model is good at fixing are also different. For example, for bugs of types *MissingBooleanExpression* and *MissingFunctionCallOrConstructor*, the repairability of SequenceR [4] is the best system, while CodeT5[76] outperforms others on some other types of bugs. We have another two suggestions based on this:

***Suggestion 2****: The performance evaluation and comparison of NPR systems should be more detailed and categorized.*

***Suggestion 3****: A practical strategy that selects the proper tool for different bugs can improve the overall repair ability.*

Generally, when proposing a new NPR system, researchers cite the total number of bugs that can be fixed as evidence of its performance superiority. However, as shown in Table 5, if taking a deep look into all bugs, no NPR system can perform others on all types of bugs. This suggests when evaluating NPR systems with a general bug-fix evaluation dataset like Defects4J, researchers should not only consider the total number of fixes, but rather a more granular classification of bugs for a fair comparison. For Suggestion 3, our findings suggest the feasibility of a repair enhancement strategy without developing new NPR systems: choosing more appropriate existing NPR tools for different bugs can greatly improve the success rate of fixes. In this case, if we have an ideal strategy that predicts the correct NPR system for every bug, the total repair performance can have an improvement of 30% (73 to 95 fixed bugs).

**Finding 2: The performance of NPRs decreases with the length of the statement that needs to be changed to fix the bug. LLM-based systems have better performance in generating long patches than NMT-based systems.**

According to the bug taxonomy proposed in Figure 4, we divide bugs into 4 categories considering the edit length required for generating a patch to fix the bug: *token-level*, *partialExpression-level*, *wholeExpression-level* and *multiExpression-level.* We count fixed bugs of each NPR system at each level. We find that current NPR systems perform better at bug-fix that require short edit scopes than long edit scopes: the best systems can fix up to 50% bugs of type *token-level* and *partialExpression-level*, whereas the highest fix rate on bugs of type *wholeExpression-level* is only 13% and merely 3% of the bugs of type *multiExpression-level* is fixed by the SOTA system. These results should not be surprising, since a longer edit scope requires an NPR system to select more code tokens from the vocab. To improve NPRs' performance on fixing longer bugs, our suggestion for this finding is:

***Suggestion 4****: NPRs with long input and short output show better performance on fixing longer bugs.*

Among the nine NPRs, SequenceR [4], RewardRepair [88], UniXCoder [21] and CodeT5 [21] are four systems that perform best at repairing *wholeExpression-level* and *multiExpression-level* bugs. One big difference between them and other systems is that they model the input and output of a program repair task as a long-input-short-output pattern. They have a method or class-level input but output line-level or state-level patch code. While other systems, CodeBERT (line-to-line), Edits (Line-to-line), and Tufano (method-to-method) do not perform as well as previous systems. And we derive similar findings in the latest research [21]. The reason we think the long input-short output approach works is that long inputs allow the model to use more context information and short outputs make it less difficult for the model to learn and focus on generating patching tokens.

**Finding 3: Basically, NPR systems are good at generating code-removal patches, but poor when the fixing requires adding new code tokens.**

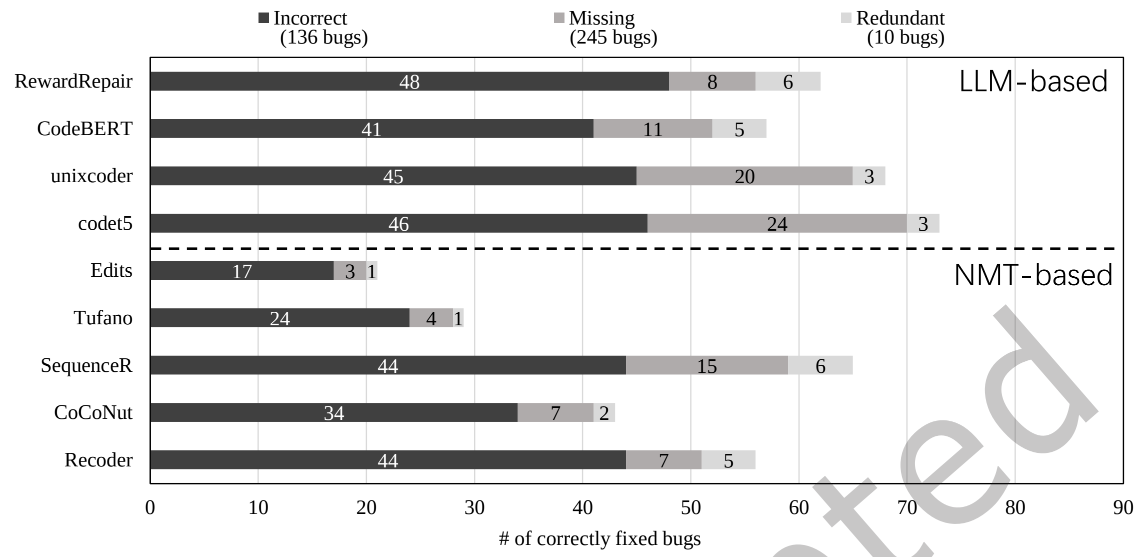

Fig. 6. Repairability of NPR systems on bugs that involve different error states.

As shown in Figure 6, we compute the reparability of each NPR system based on bugs categorized by error states. As we observed, NPR systems differ in their reparability on bugs of different error states. The highest fix rates on bugs of types *Redundant*, *Incorrect* and *Missing* are respectively 60%, 35%, and 10%. This phenomenon shows that the existing NPR system has a weak ability to introduce new code. For bugs of types *Redundant* and *Incorrect*, NPR may not need to deeply understand the code semantics and use their coding capabilities but simply delete or edit the original code tokens to repair it. However, 60% of the bugs in our validation set are of type *Missing*. Therefore, we believe that a very important direction for further exploration of NPR is:

***Suggestion 5***: *Finding ways to make the NPR model better handle bugs that require adding new code to fix.*

Having said that, we also know that this is difficult. This type of bug-fix is the most difficult to learn for NPRs because NPRs have to consider more, such as where to insert and when to end. Based on our experience replicating and training dozens of different NPR models, we suggest that researchers start by changing the training data. For example, we can classify and filter code-adding bug-fix patches in the training data, first letting the model get the correct parameter updates on simple code inserts, and then letting it try to insert longer code.

**Finding 4: LLM-based NPRs have a higher probability of fixing bugs, even if the required repair ingredients for fixing bugs are not included in the input.** To explore NPR's ability to search repair ingredients from inputs or its own search space, we categorize the bugs in the validation set by the location of the desired repair ingredients, as shown in Figure 7. We find that all NPRs have the ability to fix bugs whose repair ingredients are not included in the input context. Especially, LLM-based NPRs have better performance in handling bugs whose repair ingredients are from the source class, source projects, and even third-party projects. We hypothesize that these NPR models have been exposed to many different project codes during pre-training, and thus have better generalizability. Here we give our suggestion for LLM-based NPRs:

***Suggestion 6***: *Fine-tuning an NPR model based on a better pre-trained LLM model still works.*

**RQ1.2:** *To what extent can NPR systems be replaced by each other or non-learning APR techniques?* In *RQ1.1*, we observe that NPR systems have varying performances on different types of bugs. In this question, we further

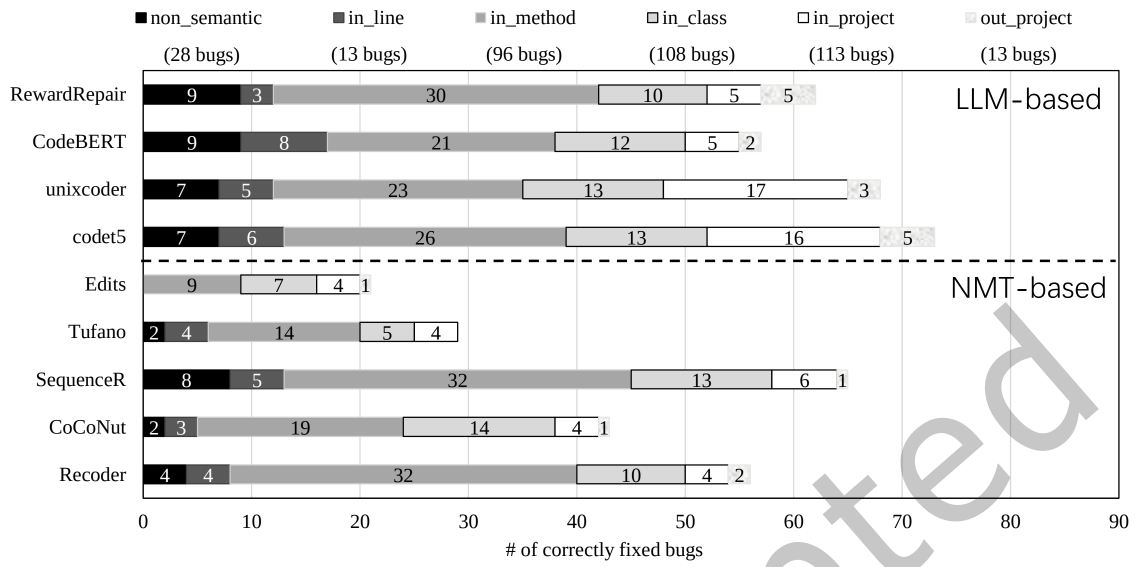

Fig. 7. The performance differences of NPRs among bugs with different .

Table 6. The overlapping and unique fixes of each NPR system.

| | NMT-based | | | | | LLM-based | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Recoder** | **CoCoNut** | **SequenceR** | **Tufano** | **Edits** | **CodeT5** | **UniXCoder** | **CodeBERT** | **RewardRepair** |
| **Recoder** | 7 | 50% | 57% | 32% | 25% | 55% | 53% | 50% | 67% |
| **CoCoNut** | 65% | 1 | 74% | 44% | 44% | 65% | 60% | 76% | 72% |
| **SequenceR** | 49% | 49% | 4 | 38% | 26% | 58% | 55% | 63% | 66% |
| **Tufano** | 62% | 65% | 86% | 0 | 31% | 75% | 65% | 86% | 68% |
| **Edits** | 66% | 90% | 80% | 42% | 0 | 76% | 66% | 90% | 76% |
| **codet5** | 42% | 38% | 52% | 30% | 21% | 5 | 80% | 46% | 54% |
| **unixcoder** | 44% | 38% | 52% | 27% | 20% | 86% | 5 | 44% | 54% |
| **CodeBERT** | 49% | 57% | 71% | 43% | 33% | 59% | 52% | 4 | 66% |
| **RewardRepair** | 61% | 50% | 69% | 32% | 25% | 64% | 59% | 61% | 4 |

investigate such performance differences between different NPR systems, as well as non-learning APR techniques.

**Finding 5: Although trained with the same data,most NPR systems can still generate unique fixes that can not be produced by other systems.**

Table 7 presents the overlapping rate and the number of unique fixes generated by the selected nine NPR systems. We find that despite using the same training data, there are differences in the bugs that NPR systems can fix correctly. Even the less capable system Edits (21 correct fixes) have 25% of its correct fixes not covered by the best system CodeT5 (73 correct fixes). And six of the nine systems (Recoder [97], SequenceR [4], CodeT5 [21], UniXCoder [21], CodeBERT [54] and RewardRepair [88]) have more than four bugs that can be uniquely fixed. This means that although the performance gap between NPRs is large, it does not mean that a poor-performance NPR system is completely useless. Some specific bugs can only be fixed by specific NPR systems. It also gives researchers new research ideas to develop better and more comprehensive NPR:

***Suggestion 7****: Utilizing the advantages of different NPRs can promote the development of more comprehensive NPR systems.*

Table 7. NPR systems' fixes that are covered by G&V APR approaches.

| | NMT-based | | | | | LLM-based | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Recoder** | **CoCoNut** | **SequenceR** | **Tufano** | **Edits** | **CodeT5** | **UniXCoder** | **CodeBERT** | **RewardRepair** |
| **ACS (16 fixed)** | 18% | 12% | 18% | 12% | 12% | 6% | 6% | 12% | 12% |
| **ARJA (11 fixed)** | 27% | 9% | 27% | 0% | 0% | 18% | 18% | 9% | 18% |
| **AVATAR (30 fixed)** | 36% | 26% | 26% | 26% | 6% | 23% | 30% | 30% | 33% |
| **Cardumem (2 fixed)** | 100% | 50% | 50% | 50% | 0% | 100% | 100% | 50% | 100% |
| **DynaMoth (3 fixed)** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| **FixMiner (34 fixed)** | 38% | 23% | 32% | 20% | 5% | 20% | 26% | 23% | 32% |
| **GenProg-A (8 fixed)** | 25% | 12% | 25% | 0% | 0% | 25% | 25% | 12% | 25% |
| **jGenProg (6 fixed)** | 33% | 0% | 16% | 0% | 0% | 16% | 16% | 0% | 16% |
| **jKali (2 fixed)** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| **jMutRepair (5 fixed)** | 80% | 60% | 80% | 80% | 40% | 40% | 60% | 80% | 80% |
| **Kali-A (5 fixed)** | 20% | 20% | 20% | 0% | 0% | 20% | 20% | 40% | 40% |
| **kPAR (33 fixed)** | 33% | 15% | 39% | 24% | 9% | 21% | 24% | 24% | 33% |
| **Nopol (2 fixed)** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| **RSRepair-A (9 fixed)** | 44% | 11% | 33% | 0% | 0% | 22% | 22% | 11% | 22% |
| **SimFix (29 fixed)** | 37% | 10% | 17% | 17% | 0% | 20% | 17% | 17% | 27% |
| **TBar (54 fixed)** | 29% | 16% | 27% | 16% | 5% | 20% | 24% | 24% | 25% |

* The data of six APR approaches is from a previous empirical study of G&V APR techniques [47]. We only count the coverage on Defects4J (v1.0) part since the previous study only evaluated APR approaches on it.

As mentioned earlier, no NPR system can fully cover the repair of other systems, thus leaving room for improvement. For example, Recoder [97], which has the most unique fixes, has special modeling of output generation different from other NPRs: generating grammar rules first and then converting them to textual codes. Combining this design with other NPRs' designs, for example, pre-training and fine-tuning a grammar-guided LLM model [96], may further facilitate the performance of NPRs.

**Findings 6: Despite the performance gap in terms of the number of totally fixed bugs, most G&V APRs have more than 50% correct fixes that can not be covered by NPRs.**

Generally, APRs are evaluated on certain bug-fix datasets (e.g., Defects4J [32]) to represent and compare their performances. In terms of this metric, the state-of-the-art NPRs have significant advantages over G&V APRs. However, it does not mean that traditional G&V APRs can totally be replaced by NPRs. As shown in Table 7, most of the APRs still have more than 50% correct fixes that can not be replaced by NPRs. This is defined by the different working modes of NPRs and G&V APRs. NPR learns from bug-fix data based on neural network models, and theoretically, the performance of NPR can continuously improve as the data and model parameters increase. Most G&V APRs use heuristic rules to search and generate patches in a limited search space. The characteristics of these two methods determine that NPR tends to focus on more general bug fixes (because such bugs are more common in training data), while APR, due to the many heuristic rules designed for patch generation, tends to be more targeted towards specific bugs. For this finding, our suggestion is:

***Suggestion 8***: *Migrate mechanisms that have worked well with traditional G&V APRs to NPRs.*
This road has been partially proven to be correct. For example, some researchers adapt repair templates, which are the core methodology of template-based APRs, to NPRs and successfully improve the repair performance [55, 79, 93]. Despite the repair templates, adapting other mechanisms (constraints [11], test cases [84]) to NPRs also worth trying.

## 5.2 RQ2. Efficiency

**RQ2.1:** *How does the repairability of each NPR system vary with the number of candidates?* To answer this question, we first measure the repairability of the seven NPR systems, counting the number of correctly fixed bugs and the average ranks of the first correct patches. Secondly, we analyze the distribution of correct patches over all bugs.

Table 8. The number of fixed bugs and the average ranks of the first correct patches produced by each NPR

| | NMT-based | | | | | LLM-based | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Recoder | CoCoNut | SequenceR | Tufano | Edits | CodeT5 | UniXCoder | CodeBERT | RewardRepair |
| **# of correctly fixed bugs** | 56 | 43 | 65 | 29 | 21 | 73 | 68 | 57 | 62 |
| **first correct rank (average)** | 37.79 | 38.47 | 50.22 | 59.41 | 40.05 | 66.16 | 59.29 | 27.18 | 36.66 |

* We color each system in order of performance. The darker the color, the higher the performance ranking among the nine systems.

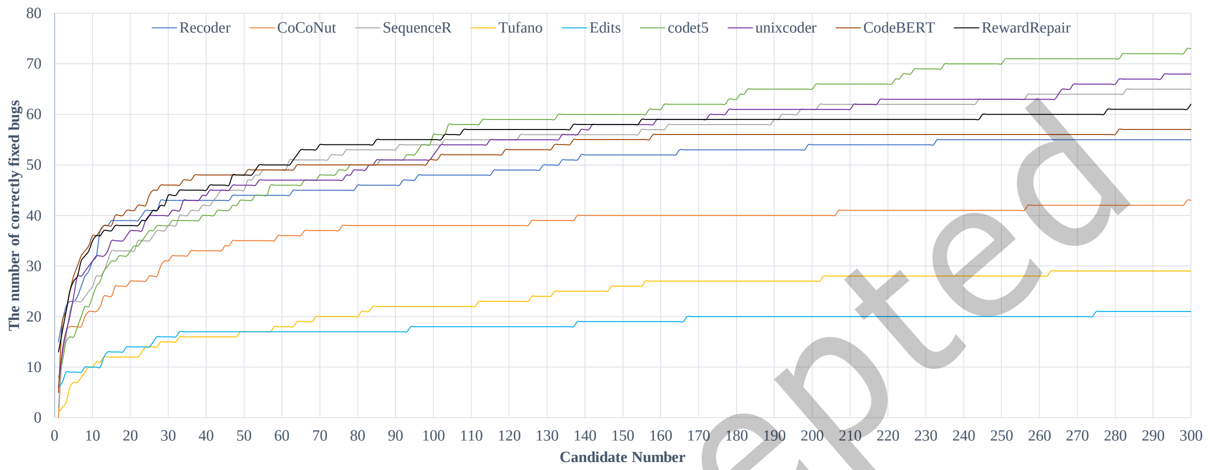

Fig. 8. The repairability and the performance change with different candidate numbers.

Third, we determine how performance changes with the number of candidates.

**Finding 7: No one system can outperform others both in repairability and efficiency. CodeBERT seems to be the most performance-efficiency balanced system.**

In Table 8, we count the number of bugs correctly fixed by the nine NPR systems when the candidate number is set to 300. In addition, we calculate the average ranks of the first correct patches produced by each NPR system. We first observe that no system is superior to other systems in both performance and efficiency. For example, in terms of the number of correctly fixed bugs, CodeT5 [21] is the best among the nine systems. However, it requires the most candidate patches (66 on average) to generate a correct patch, which means using CodeT5 needs to spend more time on validating generated patches. Considering both performance rank and efficiency rank, CodeBERT [54] is the most balanced system, which ranks 5th on performance and 1st on efficiency.

**Finding 8: The setting of the candidate number has a great impact on the final repair result of the NPR system. And the sensitivity of different systems to it is different.**

Figure 9 shows how the performance of each NPR system changes when the number of candidates is increasing. First, We find that different NPR systems exhibit different degrees of sensitivity to the number of candidates, which shows that they are not equally efficient. For example, when the candidate number is less than 100, CodeT5 is not the best system among the nine systems. Edits [8] and CoCoNut [48] have little performance engagement when the candidate number is larger than 80. Our findings related to the candidate number setting are important, since for NPRs, a larger candidate number means a higher probability of fixing the bug, but also brings more costs to the machine and human resources. For instance, when the candidate number is set to 100, CodeT5 [21] requires 20GB GPU memory for inference, which can be handled by a 1,999$ RTX 3090Ti. But when the candidate

Table 9. Performance and costs of NPRs when considering different human validation strategies (evaluating top-k test-passed patches).

| | Metric[*] | Top-1 | Top-2 | Top-3 | Top-4 | Top-5 | Top-6 | Top-7 | Top-8 | Top-9 | Top-10 | All[**] | PM_K[***] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recoder** | cn/ct | 47/88 | 52/123 | 54/149 | 54/171 | 54/190 | 55/208 | 55/223 | 55/237 | 55/251 | 55/265 | 56/380 | 18 |
| | precision | 53.41% | 42.28% | 36.24% | 31.58% | 28.42% | 26.44% | 24.66% | 23.21% | 21.91% | 20.75% | 14.74% | |
| **CoCoNut** | cn/ct | 39/63 | 42/79 | 42/92 | 42/104 | 42/116 | 42/127 | 42/136 | 42/143 | 42/149 | 42/155 | 43/225 | 26 |
| | precision | **61.90%** | **53.16%** | **45.65%** | **40.38%** | **36.21%** | **33.07%** | **30.88%** | **29.37%** | **28.19%** | **27.10%** | **19.11%** | |
| **SequenceR** | cn/ct | **51/103** | **57/149** | **61/185** | 61/213 | 62/239 | 62/258 | 62/277 | 63/293 | 64/307 | 64/320 | 65/491 | 11 |
| | precision | 49.51% | 38.26% | 32.97% | 28.64% | 25.94% | 24.03% | 22.38% | 21.50% | 20.85% | 20.00% | 13.24% | |
| **Tufano** | cn/ct | 26/55 | 28/74 | 28/91 | 29/105 | 29/116 | 29/125 | 29/132 | 29/139 | 29/145 | 29/151 | 29/231 | 4 |
| | precision | 47.27% | 37.84% | 30.77% | 27.62% | 25.00% | 23.20% | 21.97% | 20.86% | 20.00% | 19.21% | 12.55% | |
| **Edits** | cn/ct | 19/31 | 19/40 | 19/49 | 19/56 | 20/62 | 21/66 | 21/69 | 21/72 | 21/75 | 21/78 | 21/119 | 6 |
| | precision | 61.29% | 47.50% | 38.78% | 33.93% | 32.26% | 31.82% | 30.43% | 29.17% | 28.00% | 26.92% | 17.65% | |
| **codet5** | cn/ct | 50/96 | 54/137 | **61/173** | **63/199** | **63/221** | **65/243** | **67/262** | **67/278** | **68/294** | **68/309** | **73/730** | 32 |
| | precision | 52.08% | 39.42% | 35.26% | 31.66% | 28.51% | 26.75% | 25.57% | 24.10% | 23.13% | 22.01% | 10.00% | |
| **unixcoder** | cn/ct | 48/90 | 55/128 | 59/157 | 61/182 | 62/205 | 64/226 | 64/245 | 64/263 | 64/279 | 64/294 | 68/765 | 49 |
| | precision | 53.33% | 42.97% | 37.58% | 33.52% | 30.24% | 28.32% | 26.12% | 24.33% | 22.94% | 21.77% | 8.89% | |
| **CodeBERT** | cn/ct | 44/92 | 52/133 | 55/161 | 56/181 | 57/198 | 57/213 | 57/228 | 57/242 | 57/256 | 57/270 | 57/569 | 5 |
| | precision | 47.83% | 39.10% | 34.16% | 30.94% | 28.79% | 26.76% | 25.00% | 23.55% | 22.27% | 21.11% | 10.02% | |
| **RewardRepair** | cn/ct | 50/88 | 55/117 | 57/139 | 58/159 | 60/177 | 61/193 | 61/207 | 61/220 | 61/232 | 62/244 | 62/821 | 10 |
| | precision | 56.82% | 47.01% | 41.01% | 36.48% | 33.90% | 31.61% | 29.47% | 27.73% | 26.29% | 25.41% | 7.55% | |

[*] **cn** denotes the correct number of fixed bugs, **ct** denotes the human check times when top-k plausible patches are checked.
[**] **All** represents the strategy that evaluates all plausible patches until a correct patch is found.
[***] **PM_K** (PerformanceMax_K) represents that after the number of human-checked plausible patches for each bug reaches k, the total repair performance of the model will not increase.

number rises to 300, it requires 40GB GPU memory, which requires the GPU upgrade to 4,000$ A40. Hence, we give two important suggestions on this finding:

***Suggestion 9***: *For best practice, the candidate number should be set differently for each NPR system.*

***Suggestion 10***: *To make a fair comparison, the candidate number settings of the NPR system should be explained, and the performance differences between different candidate numbers should be considered whenever possible.*
For practical usage, users should first judge their hardware resources and the time consumption cost they can afford, and then choose the NPR system that meets their needs. In addition, since the NPR systems have different rankings with different candidate numbers, it is also necessary to explain the reason for setting the candidate number value and perform performance comparison under multiple candidate numbers as much as possible to make a fair comparison among NPRs.

**RQ2.2:** *How many plausible patches are needed by NPR systems to generate a correct fix?* In this question, we aim to investigate efficiency considering human evaluation time cost on checking plausible patches. We count the number of correct/plausible patches when evaluating the top-k plausible patches and list show results in Table 9.

**Finding 9: NPR systems' repairability have an improvement up to 46% when evaluating more than one plausible patch, at a cost of a significant decrease in precision.**

Generally, when evaluating NPRs, the human checker only considers the first test-passed patch (i.e., a plausible patch) on each bug. In theory, NPRs' performances may change when evaluating more than one plausible patch, at the cost of more manual checks. Table 9 represents such change on the nine NPRs in our experiment. We find that NPRs can have a huge improvement on repairability if evaluating more plausible patches. For example, CodeT5 can have an improvement of 46% when evaluating all generated plausible patches. Of course, such improvement comes with more human checks. Also the case of CodeT5, a 46% improvement in repairability requires more than 5 times human checks. So there is a performance-cost trade-off for the users:

***Suggestion 11***: *It's feasible to improve repairability if evaluating more plausible patches in practice, but users need to assess what is an acceptable labor cost.*

Table 10. Comparison of NPRs' ranks published on Defects4J V1.2 and NPR4J. We take the number of correctly fixed bugs (NPR tools stop when a first plausible patch is generated) as the ranking metric.

| **Published Year** | 2019 | 2019 | 2020 | 2020 | 2021 | 2021 | 2022 | 2023 | 2023 |
|---|---|---|---|---|---|---|---|---|---|
| **NPR System** | Tufano | SequenceR | Edits | CoCoNut | CodeBERT-ft | Recoder | RewardRepair | CodeT5-ft | UniXCoder-ft |
| **Published Ranks on Defects4J V1.2** | / | 6 | / | 5 | / | 2 | 4 | 3 | 1 |
| **Ranks on NPR4J** | 8 | 1 | 9 | 7 | 6 | 5 | 2 | 2 | 4 |

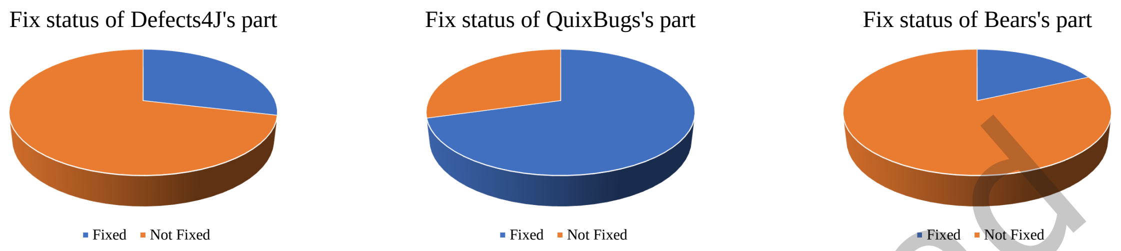

Fig. 9. The fix status of the bugs in NPR4J from each benchmark.

In practice, the patch evaluation setting of the NPR system does not need to be the same as in the experimental environment for evaluating the NPR system but can have more flexibility. As an investigation, most developers are willing to check not more than 10 patches for fixing a bug. Considering our data in Table 9, evaluating top-10 plausible patches may be a safe strategy since most NPRs have little improvement when evaluating more than 10 plausible patches. Also, it's interesting to find that NPRs' performance rank may change under different human validation strategies. For example, SequenceR has the most correct fixes when the validation strategies are evaluating Top-1, Top-2, Top-3 plausible patches, while CodeT5 performs the best when evaluating more than three plausible patches. In addition, we statistics when NPRs' performances don't increase, as shown in the rightest column in Table 9. Given the above findings, we also recommend users should use different human evaluation strategies on different NPRs for optimal usage.

## 6 DISCUSSION 1: PERFORMANCE SHIFT ACROSS TIME AND BENCHMARKS

**Finding 10: NPR's ranking has changed dramatically under different training and evaluation benchmarks. Maybe data and the model scale are the keys to the repair performance.**

Table 10 represents the repair performance ranks (in terms of the number of correctly fixed bugs) of nine NPRs under Defects4J (published in their original papers) and NPR4J. The most interesting finding is that SequenceR [4], an early NPR model proposed in 2019, ranks first place among the nine NPRs when evaluated under NPR4J. All we have done is more fully train a larger-scale SequenceR. The new SequenceR has 4x more model parameters and is trained by 3x more data samples (from 40,000 to 140,000) than the original one. Such phenomenon indicates that data, model scale, and training adequacy impact the repair performance largely. The three LLM-based NPRs have relatively smaller changes in performance. We suspect this is because they have been well pre-trained and finetune has less effect on them. This suggests to take a more cautious view of NPR's development. As new NPR tools emerge, it's important to identify which parts of the innovation make sense, rather than simply judging it on performance. After all, our experiments demonstrate that it is possible to train a larger, older model with more data to achieve the latest SOTA performance.

**Finding 11: NPRs have different fix behaviors across different benchmarks.**

Figure 9 represents the fixed percentage of NPR4J's different evaluation parts. We can see that the bugs of QuixBugs part are the easiest to fix, while NPR tools have a harder time fixing Defects4J and Bears parts. Table 11 represents the difference between the correct patches generated by the NPR system and the patches written

Table 11. The difference between the correct patches generated by the NPR system and the patches written by humans on different benchmarks. We list top-3 diff patterns according to the rules listed in Table 4.

| | | **Defcts4J Part** | **QuixBugs Part** | **Bears Part** |
|---|---|---|---|---|
| **SequenceR** | **Top-1 (%)** | R0-Identical (61%) | R0-Identical (40%) | R0-Identical (77%) |
| | **Top-2 (%)** | R16-EqualExpression (20%) | R7-ConditionEqual (34%) | R16-EqualExpression (15%) |
| | **Top-3 (%)** | R7-ConditionEqual (11%) | R11-UnneedCheck (19%) | R1-SameValue (4%) |
| **Tufano** | **Top-1 (%)** | R0-Identical (81%) | R0-Identical (47%) | R0-Identical (100%) |
| | **Top-2 (%)** | R19-DefaultParam (7%) | R4-Ternary (40%) | / |
| | **Top-3 (%)** | R17-AddCast (3%) | R7-ConditionEqual (13%) | / |
| **Edits** | **Top-1 (%)** | R0-Identical (52%) | R0-Identical (71%) | R0-Identical (52%) |
| | **Top-2 (%)** | R7-ConditionEqual (32%) | R1-SameValue (14%) | R1-SameValue (14%) |
| | **Top-3 (%)** | R4-Ternary (4%) | R16-EqualExpression (14%) | R16-EqualExpression (10%) |
| **CodeBERT-ft** | **Top-1 (%)** | R0-Identical (38%) | R0-Identical (41%) | R16-EqualExpression (45%) |
| | **Top-2 (%)** | R17-AddCast (16%) | R7-ConditionEqual (37%) | R0-Identical (32%) |
| | **Top-3 (%)** | R11-UnneedCheck (16%) | R11-UnneedCheck (19%) | R1-SameValue (15%) |
| **Recoder** | **Top-1 (%)** | R0-Identical (68%) | R0-Identical (61%) | R16-EqualExpression (63%) |
| | **Top-2 (%)** | R16-EqualExpression (22%) | R7-ConditionEqual (13%) | R0-Identical (19%) |
| | **Top-3 (%)** | R7-ConditionEqual (4%) | R16-EqualExpression (13%) | R1-SameValue (19%) |
| **CodeT5-ft** | **Top-1 (%)** | R0-Identical (58%) | R0-Identical (42%) | R0-Identical (52%) |
| | **Top-2 (%)** | R8-UnneccesaryCode (11%) | R11-UnneccesaryCode (27%) | R8-UnneccesaryCode (21%) |
| | **Top-3 (%)** | R11-UnneedCheck (11%) | R7-UnneedCheck (11%) | R16-EqualExpression (13%) |
| **UniXCoder-ft** | **Top-1 (%)** | R0-Identical (52%) | R0-Identical (55%) | R0-Identical (62%) |
| | **Top-2 (%)** | R16-EqualExpression (14%) | R16-EqualExpression (25%) | R1-SameValue (15%) |
| | **Top-3 (%)** | R10-MoreNullCheck (8%) | R10-MoreNullCheck (14%) | R16-EqualExpression (15%) |
| **CoCoNut** | **Top-1 (%)** | R11-UnneedCheck (38%) | R7-ConditionEqual (25%) | R0-Identical (41%) |
| | **Top-2 (%)** | R0-Identical (13%) | R10-MoreNullCheck (14%) | R1-SameValue (24%) |
| | **Top-3 (%)** | R16-EqualExpression (13%) | R0-Identical (13%) | R16-EqualExpression (21%) |
| **RewardRepair** | **Top-1 (%)** | R0-Identical (56%) | R0-Identical (56%) | R0-Identical (80%) |
| | **Top-2 (%)** | R7-ConditionEqual (25%) | R7-ConditionEqual (32%) | R1-SameValue (16%) |
| | **Top-3 (%)** | R11-UnneedCheck (11%) | R16-EqualExpression (12%) | R16-EqualExpression (4%) |

by humans on different benchmarks. We list top-3 diff patterns according to the rules listed in Table 4. Generally, over half of NPRs' correct patches are identical to the patches written by the developer. Among the nine systems, the patches generated by CoCoNut [48] are the least similar to the developer-written patches. In addition, we observe that even for the same system, the patches generated by different benchmarks are different. For example, on Bears Part, NPRs tend to generate more correct patches that use different fields with the same value (or alias) compared with developer-written patches. This may have something to do with where the benchmark collects the bugs. For example, QuixBugs comes from programming competitions, while Bears collect bugs found during regression testing in continuous integration systems. This suggests that we should consider the specific information of each benchmark when evaluating NPR, and clarify the significance of evaluating the performance of the NPR system on a certain benchmark.

## 7 DISCUSSION 2: CATEGORIZING THE FUTURE: THE THREE FORKS OF NPR

The past few years have seen remarkable progress in applying Deep Learning to multiple domains. For Natural Language Processing, the learning paradigm is constantly updated, from supervised learning to pre-training-fine-tuning, and then to large language models. In the field of NPR, we are also seeing similar trends:

Table 12. Comparison of three types of NPRs

| | **NMT-based** | **Open-Source-LLM-based** | **Closed-Source-LLM-based** |
|---|---|---|---|
| **Performance** | good (SOTA models). | good | excellent |
| **Repair Cost** | medium | medium | high |
| **Customization Space** | Big | medium | small |

- **NMT-based Approaches.** Before 2019, non-learning approaches are the mainstream in APR, including *heuristic-based*, *template-based*, and *constraint-based*. Around 2019, researchers began to apply deep learning techniques to the APR domain. Earlier works, such as Tufano [74] and SequenceR [4], migrate the classic neural machine translation models to APR and make task-specific optimizations in their input-output representation. Gradually, researchers are not satisfied with simple adoption and begin to explore how to integrate code-specific information such as syntax information, AST structure, project information, etc., into the NPR model architecture [2, 3, 27, 73, 97, 98]. The landmark work of this type was in 2021 when the syntax-guided NPR model Recorder [97] surpasses the non-learning APR tools for the first time. Since then, more and more researchers have realized the potential of NPR and flooded into this field.
- **Open-Source-LLM-based Approaches.** BERT [7] demonstrates the importance of bidirectional pre-training for language representations. And this paradigm has also been demonstrated in programming language [12, 76]. Naturally, NPR researchers investigated new NPR approaches based on pre-trained large language models. Some of them treat APR as a cloze-style task [26, 78, 79] so LLMs can be directly used without training, but the majority of them treat APR as a down-stream task and fine-tune the pre-trained LLMs with bug-fixing data [21, 26, 54, 88, 93]. This method proves that, without changing the model structure, relying on good language representation and collecting some data for fine-tuning can also achieve similar results as the SOTA NMT-based method on APR tasks.
- **Closed-Source-LLM-based Approaches.** This type of work is also LLM-based. The difference between them and the previous category of methods lies in how researchers use these models: users cannot perform fine-tuning because these LLM models are all closed source. Generally, these models are developed by commercial companies (e.g., OpenAI). They do not publish model weights and training data and provide services through API calls. The exceptional performance of those closed-source LLMs has given rise to new branches in the field of NPR. As is known, those models allow a longer input and have a powerful ability to understand given prompts, which eases the incorporation of rich information about the buggy program (e.g., test error information [31, 64], similar bug-fixes [30]). Chat-style LLMs such as ChatGPT provide a chance to develop NPR approaches that read feedback and gradually improve generated patches (e.g., ChatRepair [80]).

The nine tools selected for our experiment include five NMT-based and four open-source-LLM-based NPR tools. Closed-source-LLM-based NPR tools are not considered due to cost and reproducibility (Closed-Source models are often updated frequently and the old versions can not be accessed). However, an obvious trend is that all three types of NPR work will continue to emerge. These three kinds of approaches are very different from either a research or practical point of view, and there is no comprehensive discussion at present. Since we have done a large empirical study of existing NPR systems, we attempt to compare these three types of systems from the perspective of research and practicality and discuss the implications for the future of NPR. As shown in Table 12, we compare the three types of NPRs from **Performance**, **Repair Cost**, and **Customization Space**.

**Performance**. In our last section, we provide a detailed comparison of NMT-based and LLM-based (Open Source) NPRs. Generally, in terms of the number of correctly fixed bugs, LLM-based NPRs exhibit higher repair performance than NMT-based ones. Parts of the NMT-based models could achieve similar performance with

Open-Source-LLM-based NPRs. Compared with these two types of methods, closed-source-LLM-based approaches have significant advantages in repair performance. For example, early work of this type, e.g., ChatRepair [80], can fix 114 bugs on Defects4J v1.2. Its performance improved 40% compared with the SOTA NMT-based and open-source-LLM-based NPR tools at that time. The latest NPR technique SRepair [81] that utilizes both open-source-LLM and closed-source-LLM could fix 176 bugs on Defects4J v1.2, which means NPR's performance has increased by another 80% in just one year (compared with ChatRepair). Having seen the magical performance of those closed-source-LLMs and knowing their performance could still be improved according to the scaling raw, it would be no surprise for us to see that closed-source-LLM-based NPRs could flush the record again and again in the future. For this type of work, we hope to have follow-up studies to investigate their performance from other perspectives:

***Suggestion 11***: *For close-source-LLM-based NPRs, the reliability and the generalizability of their performance should be studied.*

The reason we have such concerns is that closed-source-LLM do not public their source codes and training data. As is known, neural models can well remember samples that have been seen during the training process. In the field of APR, generally, popularly used datasets are collected from open-source software repositories such as GitHub, which are also parts of the training data sources of closed-source LLMs. So there may be some overlapping samples between them, causing the data leakage issue. To precisely evaluate NPRs, it's important to ensure that they know how to fix the bugs, not just remember some samples in the training set. For NMT-based and open-source-LLM-based NPRs, it's easy to solve this problem. As we do in this article, we can check the training set and evaluation set to confirm which samples are overlapping. However, close-source-LLM-based NPRs can't do so. Some works try to alleviate the data leakage problem by creating new datasets [26]. But this is still a palliative: you still can't be sure that the samples in the new dataset will overlap with the LLMs' training samples. As far as we know, there is currently no mature evaluation solution to solve this problem. We can get some inspiration from some work on the robustness of NPR tools [13]. If we make some changes to the bug program without changing the semantics (for example, changing variable names, or inserting useless code), and then NPR can still fix the bug correctly, it may indicate to some extent that NPR tools know how to fix the bug.

**Repair Cost**. Since program repair is a practical task, the repair cost is also an important attribute of the APR tool. For NPR tools, the repair cost consists of two parts: *tool training* and *tool execution*. If users want to build an NPR tool with their own data, training/fine-tuning is required for NMT-based/open-source-LLM-based NPRs. The training cost of neural networks is strongly related to the number of model parameters and training data. Among the three types of NPRs, NMT-based ones have the lowest training costs. In our experiment, the biggest NMT-based model SequenceR has 140M parameters. It requires 15 hours for training with 2 NVIDIA V100 GPUs (each GPU has 32GB memory). The size of parameters of Open-Source-LLM-based range from 60M to 220M, taking at most 2 days for training on the same GPUs. However, we should note that open-source LLMs also have the costs of pre-training. On the tool execution costs, the repair costs of NMT-based and open-source-LLM-based NPRs are similar. Since they are open-sourced, users can reduce use costs by deploying on their servers. However, for closed-LLM-based NPRs, users must pay for the API usage. Generally, the companies price the API by token lengths. For example, if you want to use GPT-4, you need to pay $30 for each 1M token input and $60 for each 1M token output [6]. This charging method is not very friendly to Closed-LLM-based NPRs. The inputs of such methods to the program repair task are the defect codes, often with additional information such as method-level or even class-level context, test codes, etc. Thus, compared with normal QA tasks, closed-LLM-based NPRs cost more per query. At this point, readers may notice that we use different dimensions (time, hardware, money) to describe the repair costs of different types of NPR. Exactly, we have to admit that the biggest difficulty in

[6]https://openai.com/api/pricing/

evaluating their repair costs right now is the inability to compare all tools horizontally and uniformly in a fair way. For fair evaluation and comparison of repair costs, we recommend that:

***Suggestion 12****: When measuring repair costs, try to avoid comparing NPR tools across categories.*

When comparing fix performance, all NPR tools can be compared together because there is a tool-independent metric (i.e., number of bugs fixed). But when it comes to the comparison of repair costs, the situation is different. For example, NMT-based NPRs are trained from scratch, and Open-Source-LLMs are pre-trained-then-fine-tuned. Whether to take the cost of the latter's pre-training into account is also a question worth discussing. For closed-source-LLM-based NPRs, you can't even calculate the cost of their pre-training. On the execution cost, if measuring them by spent dollars, the cost of running NMT-based and open-source-LLM-based methods is hard to price. Therefore, until there is a better solution, we recommend not to compare the repair costs of NPR across categories.

**Customization Space**. A clear trend is that closed-source-LLM-based NPRs will be far superior to other types of methods in terms of repair performance in the future. Does that mean we do not need to develop NMT-based and open-source-LLM-based NPRs anymore? The answer is not and we try to explain this from the perspective of the customization space. By customization space, we mean how much customization can be done on NPRs to meet users' or researchers' requirements. The field of program repair often faces bugs in new programming languages and new software fields. This type of data is typical out-of-distribution data for trained NPRs. All types of NPRs will struggle if they do not see similar examples before. In this case, for NMT-based and open-source-LLM-based NPRs, the most efficient and effective approach is to collect such samples and re-train/fine-tune NPRs. However, closed-source-LLM-based NPRs have little space for adaption since you can not fine-tune them even if you have collected enough samples. The possible solution is to perform prompt engineering (e.g., providing few-shot examples). However, this is also unstable and unpredictable compared to direct training /fine-tuning NMT-based/open-source-LLM-based NPRs. We can draw a conclusion that among the three types of NPRs, closed-source-LLM-based NPRs have the minimum modification space. And we find NMT-based NPRs have the maximum room for modification. Note that compared with natural languages, programming languages have more standard syntax and richer semantic information. For Open-Source-LLMs, if you want to take advantage of their trained representations of language, you must inherit their models and vocabularies (usually BPE participles). This means that it will be difficult to change these models for feature fusion and to ensure that the output is syntactically correct. For NMT-based NPRs, it would be easier to have such modifications. Some models incorporate domain knowledge [27], being aware of syntactical grammar [97], or being type-aware [98]. This shows that the NMT-based NPR is easier to optimize in a specific direction than the other two kinds of methods.

## 8 RELATED WORK

### 8.1 Program Repair Framework

A program repair framework supports the execution of more than one program repair tool. Astor [52] is an automatic Java software repair framework for Java. It integrates six program repair tools for Java. RepairThemAll [9] is an execution framework that supports executing eleven APR tools on five benchmarks. However, it doesn't support executing APR tools beyond the integrated benchmarks. Cerberus [70] is a program repair framework that provides the interface to multiple state-of-the-art program analysis tools such as Infer and Pulse, as well as program repair tools such as Prophet, Darjeeling, Angelix, F1X, etc. In this paper, we build *NPR4J*. Different from Astor and RepairthemAll, it is the first program repair framework that supports training and evaluating NPR systems. In addition, it allows users to train and validate integrated systems with any data.

### 8.2 Bug Classification

A bug classification means a family of bugs in common. There is an open understanding of ”in common” [56]: they can share the same root cause (e.g., a programmer mistake), the same symptom (e.g., an exception) or the same kind of fix (e.g. changing an operator). For different purposes, many bug classifications have been proposed (e.g., ODC [5] and CWE [50]). In our experiment, since our goal is to distinguish the complexity of different bug-fixes, we choose to classify bugs based on their required fixes. Based on a similar previous bug taxonomy [62], we propose a bug taxonomy guided by the error states and error elements of the buggy codes.

### 8.3 Evaluation of Program Repair Tools

Empirical validations of automated program repair approaches are also related to our research. Initially, the performances of repair tools are measured by counting the number of bugs for which the repair tool can generate a patch that passes all test cases [17]. However, Qi et al. [66] introduced the concept of plausible (i.e. test-adequate) versus correct patches. They find test-adequate patches are sometimes incorrect which may cause other issues due to the incompleteness of test cases. From then on, the number of correct/plausible patches becomes a widely accepted metric for the APR community.

To boost the development of APR, various empirical studies have been conducted ([1, 10, 15, 16, 45, 47, 51, 59, 60, 89]). Among all empirical studies, the researches most related to us are two. Motwani et al. [59] investigate to what extent important and complex bugs can be fixed by 9 APR tools. And Liu et al. [47] investigates the efficiency of 16 APR tools. However, our study is very different from theirs. First, they only focus on G&V APR approaches. Compared with G&V APR approaches, NPR has many differences that require different dimensions for evaluation (e.g., the candidate number). Second, although we have the same goal that we should distinguish bugs by complexity, we use a completely different classification. They measure a bug's complexity according to the number of files containing non-comment, non-blank-line edits in the developer-written fix and the total number of non-comment, non-blank lines of code. In this paper, we propose a new bug taxonomy based on a previous work [62], implementing a straightforward way to classify bugs via error states and error code elements.

## 9 THREATS TO VALIDITY

*External validity.* Our study considers only one-line bugs of Java. All findings might thus be valid only for this configuration. Nevertheless, this threat is mitigated by the fact that we use a large number of NPR systems and bugs collected from three well-established defect benchmarks to study a performance criterion that was largely ignored in the literature.

*Internal validity.* Our manual assessment of patch correctness may threaten the validity of some of our conclusions. We mitigate this threat by using the rules for defining patch correctness from previous research [47] as our criteria. Every patch is confirmed by two checkers. A patch is regarded as correct if and only if the two checkers both accept it.

*Construct validity.* By construction, to limit resource exhaustion, we only set the number of candidates to 300. Theoretically, the performance of an NPR system may be further enhanced if there is a larger number of candidates. However, a larger number of candidates requires more GPU memory. Thus, 300 is the largest number that can be used given our machine resources (Nvidia Tesla V100 with 32 GB memory). Note that 300 are already substantially larger than that in the original settings of 6 of the NPR systems (except for CoCoNut [48]).

## 10 CONCLUSION AND FUTURE WORK

In this paper, we construct a framework tool, *NPR4J*, which supports training and reusing seven SOTA NPR systems for Java. Then, based on the framework, we re-train the seven systems and perform a comprehensive evaluation of them. Different from previous evaluations that mainly focus on repairability, we concentrate on the

**effectiveness** (NPRs' performances on different types of bugs) and the **efficiency** (NPRs' evaluated performances when using different evaluation strategies). Our study yields plenty of findings that reveal the real affairs of NPR systems, which could provide practical instructions for practitioners and facilitate the development of the NPR field. In future work, we will continue to integrate new NPR systems into NPR4J, as well as NPR systems for the other programming languages.

## ACKNOWLEDGEMENT

This research / project is supported by the National Key Research and Development Program of China (2022YFF0711404), National Natural Science Foundation of China (62172214), Natural Science Foundation of Jiangsu Province, China (BK20201250, BK20210279), CCF-Huawei Populus Grove Fund, NSF award 2034508, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s). We also thank the reviewers for their helpful comments. Chuanyi Li and Jidong Ge are the corresponding authors.

## REFERENCES

[1] Aldeida Aleti and Matias Martinez. 2021. E-APR: Mapping the effectiveness of automated program repair techniques. *Empir. Softw. Eng.* 26, 5 (2021), 99. https://doi.org/10.1007/s10664-021-09989-x

[2] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. http://proceedings.mlr.press/v139/berabi21a.html

[3] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[4] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[5] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. 1992. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Trans. Software Eng.* 18, 11 (1992), 943–956. https://doi.org/10.1109/32.177364

[6] Junyoung Chung, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated Feedback Recurrent Neural Networks. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*, Francis R. Bach and David M. Blei (Eds.). JMLR.org, 2067–2075. http://proceedings.mlr.press/v37/chung15.html

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[8] Yangruibo Ding, Baishakhi Ray, Premkumar T. Devanbu, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 275–286. https://doi.org/10.1145/3324884.3416587

[9] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[10] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[11] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, Christof J. Budnik,

Gordon Fraser, and Francesca Lonetti (Eds.). ACM, 85–91. https://doi.org/10.1145/2896921.2896931

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[13] Hongliang Ge, Wenkang Zhong, Chuanyi Li, Jidong Ge, Hao Hu, and Bin Luo. 2023. RobustNPR: Evaluating the robustness of neural program repair models. *Journal of Software: Evolution and Process* (2023), e2586.

[14] Ali Ghanbari and Lingming Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1118–1121. https://doi.org/10.1109/ASE.2019.00116

[15] Davide Ginelli, Matias Martinez, Leonardo Mariani, and Martin Monperrus. 2022. A comprehensive study of code-removal patches in automated program repair. *Empir. Softw. Eng.* 27, 4 (2022), 97. https://doi.org/10.1007/s10664-021-10100-7

[16] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. https://doi.org/10.18653/V1/2022.ACL-LONG.499

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[20] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 5 (1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[21] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181

[22] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *CoRR* abs/2303.18184 (2023). https://doi.org/10.48550/arXiv.2303.18184 arXiv:2303.18184

[23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38. https://doi.org/10.1145/3571730

[24] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 255–266. https://doi.org/10.1109/ASE.2019.00033

[25] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. 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, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[26] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[27] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1251–1263. https://doi.org/10.1109/ICSE48619.2023.00111

[28] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[29] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[30] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1646–1656. https://doi.org/10.1145/3611643.3613892

[31] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, 5131–5140. https://doi.org/10.1609/AAAI.V37I4.25642

[32] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[33] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 573–577. https://doi.org/10.1145/3379597.3387491

[34] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[35] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 314–325. https://doi.org/10.1145/3338906.3338935

[36] Suhua Lei, Huan Zhang, Ke Wang, and Zhendong Su. 2018. How training data affect the accuracy and robustness of neural networks for image classification. (2018).

[37] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 511–523. https://doi.org/10.1145/3510003.3510177

[38] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[39] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Gail C. Murphy (Ed.). ACM, 55–56. https://doi.org/10.1145/3135932.3135941

[40] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019.* IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[41] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019.* IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[43] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[44] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *J. Syst. Softw.* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[45] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *J. Syst. Softw.* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[46] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[47] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[48] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[49] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[50] Robert A Martin. 2007. Common weakness enumeration. *Mitre Corporation* 24 (2007).

[51] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empir. Softw. Eng.* 22, 4 (2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[52] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

[53] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[54] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 505–509. https://doi.org/10.1109/MSR52588.2021.00063

[55] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1456–1468. https://doi.org/10.1109/ICSE48619.2023.00127

[56] Martin Monperrus. 2014. 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 *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 234–242. https://doi.org/10.1145/2568225.2568324

[57] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[58] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. *CoRR* abs/2108.04631 (2021). arXiv:2108.04631 https://arxiv.org/abs/2108.04631

[59] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs?. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 25. https://doi.org/10.1145/3180155.3182533

[60] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2022. Quality of Automated Program Repair on Real-World Defects. *IEEE Trans. Software Eng.* 48, 2 (2022), 637–661. https://doi.org/10.1109/TSE.2020.2998785

[61] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 819–830. https://doi.org/10.1109/ICSE.2019.00089

[62] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *J. Syst. Softw.* 163 (2020), 110538. https://doi.org/10.1016/j.jss.2020.110538

[63] Lutz Prechelt. 2012. Early Stopping - But When? In *Neural Networks: Tricks of the Trade - Second Edition*, Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller (Eds.). Lecture Notes in Computer Science, Vol. 7700. Springer, 53–67. https://doi.org/10.1007/978-3-642-35289-8_5

[64] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 69–75. https://doi.org/10.1145/3524459.3527351

[65] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 24–36. https://doi.org/10.1145/2771783.2771791

[66] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 24–36. https://doi.org/10.1145/2771783.2771791

[67] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[68] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 10–13. https://doi.org/10.1145/3196398.3196473

[69] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[70] Ridwan Shariffdeen, Martin Mirchev, Yannic Noller, and Abhik Roychoudhury. 2023. Cerberus: a Program Repair Framework. In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023.* IEEE, 73–77. https://doi.org/10.1109/ICSE-Companion58688.2023.00028

[71] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *CoRR* abs/2312.15698 (2023). https://doi.org/10.48550/ARXIV.2312.15698 arXiv:2312.15698

[72] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015,* Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 532–543. https://doi.org/10.1145/2786805.2786825

[73] Yu Tang, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Muyun Yang. 2021. Grammar-Based Patches Generation for Automated Program Repair. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 1300–1305. https://doi.org/10.18653/v1/2021.findings-acl.111

[74] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[76] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[77] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 354–366. https://doi.org/10.1145/3468264.3468600

[78] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[79] Chunqiu Steven Xia and Lingming Zhang. 2022. 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, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. https://doi.org/10.1145/3540250.3549101

[80] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023). https://doi.org/10.48550/ARXIV.2304.00385 arXiv:2304.00385

[81] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Haotian Zhang, and Yuqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? *arXiv preprint arXiv:2404.12833* (2024).

[82] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[83] Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo A. Furia. 2022. Restore: Retrospective Fault Localization Enhancing Automated Program Repair. *IEEE Trans. Software Eng.* 48, 2 (2022), 309–326. https://doi.org/10.1109/TSE.2020.2987862

[84] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1

(2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[85] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 107:1–107:13. https://doi.org/10.1145/3551349.3556893

[86] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 92:1–92:13. https://doi.org/10.1145/3551349.3556926

[87] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empir. Softw. Eng.* 26, 2 (2021), 20. https://doi.org/10.1007/s10664-020-09920-w

[88] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1506–1518. https://doi.org/10.1145/3510003.3510222

[89] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empir. Softw. Eng.* 23, 5 (2018), 2948–2979. https://doi.org/10.1007/s10664-017-9552-y

[90] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering* 46, 10 (2018), 1040–1067.

[91] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 5:1–5:53. https://doi.org/10.1145/3360004

[92] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *CoRR* abs/2301.03270 (2023). https://doi.org/10.48550/arXiv.2301.03270 arXiv:2301.03270

[93] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023.* IEEE, 535–547. https://doi.org/10.1109/ASE56229.2023.00063

[94] Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. 2022. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 97:1–97:13. https://doi.org/10.1145/3551349.3556943

[95] Wenkang Zhong, Chuanyi Li, Jidong Ge, and Bin Luo. 2022. Neural Program Repair : Systems, Challenges and Solutions. In *Internetware 2022: 13th Asia-Pacific Symposium on Internetware, Hohhot, China, June 11 - 12, 2022.* ACM, 96–106. https://doi.org/10.1145/3545258.3545268

[96] Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. GrammarT5: Grammar-Integrated Pretrained Encoder-Decoder Neural Model for Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE).* IEEE Computer Society, 860–860.

[97] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. https://doi.org/10.1145/3468264.3468544

[98] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1443–1455. https://doi.org/10.1109/ICSE48619.2023.00126