Leveraging Search-Based and Pre-Trained Code Language Models for Automated Program Repair

Oebele Lijzenga
o.r.lijzenga@student.utwente.nl
University of Twente
Enschede, The Netherlands

Iman Hemati Moghadam
iman.hematimoghadam@utwente.nl
University of Twente
Enschede, The Netherlands
Eindhoven University of Technology
Eindhoven, The Netherlands

Vadim Zaytsev
vadim@grammarware.net
University of Twente
Enschede, The Netherlands

Abstract

Background. Automated Program Repair (APR) techniques often face challenges in navigating vast search space of possible patches and often rely on *redundancy-based assumptions*, which can restrict the diversity of generated patches. Recently, Code Language Models (CLMs) have emerged as a method for dynamically generating patch ingredients, potentially enhancing patch quality.

Aim. This study aims to enhance APR by integrating search-based methods with CLMs to improve both the quality of generated patch ingredients and the efficiency of the search process.

Method. We propose ARJACLM, a novel APR technique that uses a *genetic algorithm* for search space navigation and dynamically generates patch ingredients with the CodeLLaMA-13B model, combining redundancy-based and CLM-derived patch ingredients.

Results. Testing on 176 bugs across 9 Java projects from Defect4J shows that CLM-generated patch ingredients significantly boost ARJACLM's performance, though at the cost of increased computation time. ARJACLM outperforms ARJA and GenProg, and CLM-generated patch ingredients are of higher quality than their redundancy-based counterparts. Additionally, ARJACLM performs best when redundancy-based patch ingredients are ignored.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Program Repair, Search-Based Algorithm, Code Language Model.

ACM Reference Format:
Oebele Lijzenga, Iman Hemati Moghadam, and Vadim Zaytsev. 2025. Leveraging Search-Based and Pre-Trained Code Language Models for Automated Program Repair. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3672608.3707774

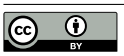
1 Introduction

Software bugs are common and their occurrence and complexity have increased with the growing size and complexity of modern software systems. As a result, software engineers must spend significant time fixing bugs to ensure proper functionality. However, localising and fixing bugs is a costly activity, estimated to be worth billions of dollars annually worldwide [1]. This makes Automated Program Repair (APR) highly desirable, leading to the development of various technologies and methodologies in recent years [2–7].

Among the proposed approaches, *generate-and-validate* techniques are the most widely used [2]. These methods work by generating numerous candidate patches to fix a specific bug and validating each one by compiling and testing to determine its effectiveness in fixing the bug [2]. The candidate patches are typically chosen based on specific criteria from a large *search space* of potential solutions. However, a significant challenge is how to effectively *define* and *navigate* this search space [8]. A small search space allows for faster exploration but may miss potential solutions if they are not included. Conversely, a large search space may contain the needed solutions but can be costly and time-consuming to explore. Thus, both how the search space is *defined* and how it is *navigated* are critical factors. We review *evolutionary algorithms* [8–10] and *learning-based algorithms* [11–13] used to tackle these challenges.

GenProg [9], a well-known search-based APR tool, defines its search space based on the assumption that the code needed to generate a repair (referred to as patch ingredients) already exists elsewhere in the buggy program. This concept is known as the *redundancy assumption* or *plastic surgery hypothesis* in the research literature [14, 15]. Although the redundancy assumption narrows the patch ingredients to those available within the existing code and makes the search space manageable, it also limits the bug-fixing capability if the necessary patch ingredient is not within the available search space [16]. This issue can be more pronounced in smaller projects. GenProg [9] also faces challenges in navigating the search space effectively, as its reliance on random mutations—such as adding, replacing, or removing statements—can lead to nonsensical patches not accepted by developers [17]. Indeed, Qi et al. [18] demonstrated that replacing GenProg's search algorithm with random search improves both the success and speed of repairs.

One notable work that aimed to improve GenProg is ARJA [8]. ARJA advances GenProg by employing a more effective search algorithm and redefining the search space. Specifically, ARJA employs a fine-granularity patch representation that enhances search space organization, enabling more precise navigation. Additionally, ARJA uses a novel filtering technique to reduce the search space by eliminating irrelevant patch ingredients, such as removing method calls not accessible at the buggy location. ARJA also employs a type-matching approach to extend the search space by generating



SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707774

new code based on syntactic patterns observed in the program, like substituting inaccessible method calls with suitable alternatives available at the buggy line. This combination of filtering and type matching enhances search space quality, reduces execution time, and improves repair effectiveness. However, Yuan and Banzhaf [8] observed that some bugs remained unfixed because the necessary patch ingredients could not be found in the program, even with the proposed type-matching approach. They also noted instances where, despite the patch ingredients being present, the search algorithm was not robust enough to locate them [8].

To address these issues, Yuan and Banzhaf [10] introduced an enhanced version of their tool called ARJA-e. This tool improves the source of potential patch ingredients by incorporating both ingredients derived from the redundancy assumption and *repair templates* proposed by Kim et al. [17]. Additionally, they categorize patch ingredients into two groups—replacement and insertion—based on their suitability for specific operations. This categorization helps reduce the search space for each operation type. However, while ARJA-e [10] successfully outperforms other approaches, including ARJA [8], its reliance on template repair may not cover all possible bug types and can overfit specific patterns [3].

In a recent work, Li et al. [11] proposed ARJANMT, a hybrid APR technique that enhances ARJA by incorporating additional patch ingredients derived from the learning-based tool SequenceR [19]. ARJANMT demonstrates that patch ingredients generated by learning-based techniques can improve the performance of traditional search-based APR techniques. However, the improvement in bug-fixing capabilities of ARJANMT over AJRA was modest (50 vs. 47) as it was significantly constrained in two ways. First, ARJANMT generates all patch ingredients beforehand based on the initial buggy program, so patch ingredients produced by SequenceR are not adapted to the subsequent changes made to the program during the patching process. As a result, the learning-based patch ingredients do not account for modifications that occur as the program evolves. In addition, SequenceR relies on limited training data, which restricts its ability to generate effective patches, a common limitation of traditional machine learning techniques [12].

In a recent study, Xia and Zhang [12] demonstrate that pre-trained code large language models can outperform existing learning-based APR techniques, even in a zero-shot learning setting, where the model is not explicitly trained for program repair. In the proposed approach, the likely buggy statement is *partially* or *completely* replaced with a mask, and then patch ingredients are generated by asking a pre-trained language model to fill in the masked statement. This method allows for the creation of patch ingredients directly from the provided code and addresses the limitations of the redundancy assumption. However, despite the impressive capabilities of pre-trained code language models in APR [12, 13, 20–25], they face several limitations as highlighted by Zhang et al. [6]. First, these models can only process a limited amount of code at once, which restricts their ability to analyze entire programs and affects the quality of the generated patch ingredients. Second, they frequently suffer from hallucinations, resulting in syntactically or semantically incorrect code [26]. Additionally, while these models excel in general code generation, they lack explicit APR capabilities. Therefore, to be effective, they need to be integrated into APR techniques that leverage them to generate code in specific contexts and create meaningful patches [13, 27]. Furthermore, pre-trained code language models are resource-intensive and time-consuming, limiting the generation of patch ingredients per buggy statement.

To capitalize on the strengths of both search-based techniques and pre-trained Code Language Models (CLMs) while addressing their respective limitations in automated program repair, this study proposes ARJACLM. ARJACLM is an innovative search-based APR technique built upon ARJA, which uses CLMs to dynamically generate patch ingredients. ARJACLM leverages a *genetic algorithm* for search-based navigation of the patch space and utilizes CLMs in a *zero-shot setting* to enhance the quality of patch ingredients. The search space in ARJACLM incorporates both patch ingredients derived from the redundancy assumption and those generated on demand by the CLM. Further detail is provided in Section 2.

ARJACLM is evaluated on a subset of 176 bugs from 9 Java projects in the Defect4J dataset [28] and compared with a version of ARJACLM that does not utilize CLMs, as well as with ARJA [8] and a version of GenProg [29] capable of repairing bugs in Java projects. The results demonstrated that ARJACLM outperforms these approaches but at the cost of increased computation time. We also observed that CLM-generated patch ingredients are of higher quality than their redundancy-based counterparts, and ARJACLM performs best when redundancy-based patch ingredients are ignored. In summary, the contributions of our study are as follows:

(1) A novel APR technique that uses both search-based methods and pre-trained language models to improve the navigation of the search space and enhance the quality of generated patches.
(2) An evaluation of the proposed approach on a subset of bugs from the Defect4J dataset, and comparing it with other existing search-based techniques, namely ARJA and GenProg.
(3) Public release of ARJACLM [30], allowing other researchers to replicate experiments and contribute to further enhancements.

The remainder of this paper is organized as follows. We proceed by discussing the foundation of our proposed approach. Next, we present and discuss the results from our experiments in Section 3, followed by an analysis of limitations and threats to validity in Section 4. A survey of related work is presented in Section 5, and finally, in Section 6, we conclude and discuss future work.

2 Proposed Approach

The goal of this research study is to evaluate the efficacy of integrating search-based techniques with code language models for automated program repair. This involves extracting modification points, and iteratively mutating the code using a search-based algorithm, while consulting with a language model to enhance modification guidance. The proposed approach, depicted in Fig. 1, consists of *three* steps executed consecutively, and detailed as follows.

The process begins with two inputs: a *Buggy Program* and a *Test Suite* which is used for testing the functionality of the input program. In the test suite, there has to be at least one case that causes it to fail, indicating the presence of the bug. The first phase, named *Preparation and Analysis*, begins with the extraction of relevant data—referred to as ingredient statements—from the buggy program using the *Source Data Extraction* component. This data is *later* filtered and categorized based on the lines it covers, and serves as the basis for replacing and inserting code during patch generation.

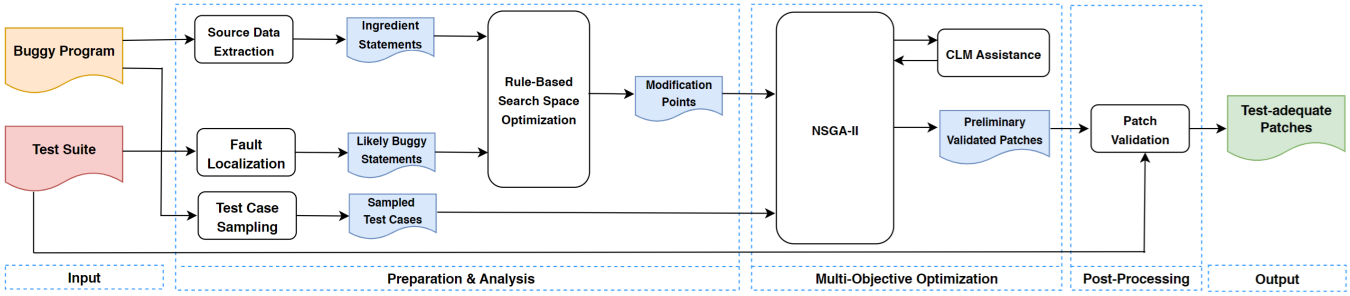


Figure 1: Integrating Search-Based Methods with Pre-Trained Language Models for More Effective Bug Fixing.

In the preparation and analysis phase, we also employ a *Fault Localization* technique to identify likely buggy statements (LBSs). As illustrated in Fig. 1, it takes as inputs a buggy program and a test suite and finds LBSs by analyzing how the test suite behaves on the input program. Each LBS is assigned a suspiciousness score between 0 (low) and 1 (high), indicating the likelihood of a defect.

Another activity done during the preparation and analysis phase includes determining what *kind of changes* should be made to a buggy statement i.e., delete it, replace it, or insert something before or after it. In this step, we also decide about the *ingredient statements*—statements that can be included with the buggy statements. For instance, one type of change may include replacing a method call in the buggy statement with another call serving as the ingredient statement. These activities are done in *Rule-Based Search Space Optimization* step. It involves identifying which operations and ingredient statements are relevant, based on their visibility and scope relative to the buggy statements. This step is essential to reduce the search space and also ensure the changes can be effectively applied. In this study, we follow the rules defined by the ARJA study [8] to filter the type of changes and ingredient statements. However, as will be discussed in Section 2.3, we also extend some rules to enhance the effectiveness of the search space.

The outcome of the *Rule-Based Search Space Optimization* is a set of modification points. These points represent statements in the buggy program that can be modified by a patch and specify how the buggy statements can be altered. Fig. 2 illustrates an example of a modification point. As shown, each modification point has a unique index, a suspiciousness score, the likely buggy statement that can be modified, and the patch operations and patch ingredients that can be used to modify the buggy statement. For example, in the provided example, a potential solution is to *replace* the provided buggy line: `result = value;` with `result += value;`.

Index	9
Suspiciousness	0.733
Buggy Statement	result = value;
Operation Types	Insert , Replace , Delete

Ingredient Statements	result += value;
	if (value < 0) return -1;
	....
	applyResult(value);

Figure 2: A Modification point in ARJACLM.

In the second phase of the proposed approach, a search-based algorithm evolves patches to modify the identified modification points to generate a set of *Preliminary Validated Patches* that could potentially fix the bugs. The employed search-based algorithm refines the resulting patches using the input test suite. ARJACLM uses a subset of tests from the input test suite to speed up the search process. This subset is determined through *Test Case Sampling*, performed in the first phase, discussed later in Section 2.2.

ARJACLM, similar to ARJA [8], employs a *multi-objective genetic algorithm* based on NSGA-II [31] to balance two key objectives: *maximum correctness* and *minimal modification*. Indeed, between two valid solutions, the one with fewer changes is preferred over the other. In this way, the fitness function promotes patches that reduce the number of changes that can be applied to the program. Both ARJA and ARJACLM utilise the *redundancy assumption* to generate patch ingredients. It means that the code necessary for generating a repair already exists elsewhere in the buggy program. However, ARJACLM extends this approach by consulting pre-trained code language models to generate additional patch ingredients directly from the code surrounding the buggy line, with the likely buggy line itself masked. This allows the exploration of a vast search space of potential patch ingredients. The genetic algorithm and its integration with code language models are described in Section 2.4.

In the final phase of the proposed approach, serving as a *post-processing* step, the preliminary validated patches generated in the search-based phase undergo rigorous validation against the *entire* input test suite. This is necessary as preliminary validated patches are only validated based on a subset of test cases as discussed before. Patches that successfully pass all the test cases are considered *Test-Adequate Patches* and are deemed valid fixes for the identified bugs.

In the following subsections, a more detailed description of each step is provided.

2.1 Fault Localization

To identify likely buggy statements, we used the GZoltar toolset version 1.7.3 [32, 33] and applied the Ochiai metric [34] to measure suspiciousness of likely buggy statements, similar to ARJA [8]. We then use GZoltar's information to (1) filter unrelated test cases (see Section 2.2) and (2) determine program entities (e.g., classes, methods, fields, variables) accessible at the buggy statement's location (see Section 2.3). As in ARJA [8], we narrow the search space by applying a suspiciousness threshold (0.1) and a modification points limit (40), selecting statements that meet these criteria.

2.2 Test Case Sampling

Most of the execution time of search-based APR techniques consists of the evaluation of patches. Therefore, it is necessary to filter out unrelated test cases to speed up the process. ARJA [8] filters positive tests based on their code coverage. Indeed, positive test cases that do not run any of the lines associated with the likely buggy statement are excluded [8]. ARJACLM, however, leverages a straightforward test sampling technique. Indeed, test cases located in the same package as a negative test case are always used. Moreover, the rest positive test cases are randomly selected until a fraction of all positive tests are sampled. The sampling ratio parameter is a configurable parameter of ARJACLM and determines how many positive test cases are sampled. This test sampling technique prioritizes test cases which are likely related to the buggy code based on their location and avoids the use of a complex coverage-based sampling technique used by ARJA. As discussed in Section 2, the preliminary validated patches generated in the search-based phase undergo rigorous validation against the *entire* input test suite at the end. Therefore, patches that *successfully pass all test cases* in the project are deemed valid fixes for the identified bugs.

2.3 Rule-Based Search Space Optimization

During the preparation and analysis phase, we determine the possible changes for a buggy statement—such as deletion, replacement, or insertion—and also decide on the ingredient statements that can accompany the buggy statements. For example, a change might involve replacing a method call with another *visible* and *compatible* method that serves as an ingredient statement.

The *Rule-Based Search Space Optimization*, shown in Fig. 1, identifies relevant *operations* and *ingredient statements* based on their appropriateness and visibility, thereby reducing the search space and ensuring effective changes. In this study, we follow ARJA's rules [8] for filtering operations and ingredients, while extending some rules to enhance search space effectiveness. We define three types of rules: (i) disabling specific operations, (ii) disabling specific ingredients, and (iii) disabling certain operations on particular ingredients. These rules are applied to each modification point.

2.3.1 ***Operation Screening.*** Operation screening determines which operations are suitable for modifying a likely buggy statement and should be applied at the relevant modification point. We follow two rules from ARJA [8]. First, we avoid deleting a `return` statement if it is the last statement in a non-void method as it results in a compiler error [8]. Second, we do not delete a variable declaration because if it is used in the program, this would lead to a compiler error. However, if the variable is unused, it is considered redundant and does not affect program correctness [8].

2.3.2 ***Ingredient Screening.*** Ingredient screening filters patch ingredients based on their *visibility* and *compatibility* with the code surrounding a likely buggy statement. To determine ingredient statements for each buggy line, we first extract the scope of all classes, fields, variables, and methods (including their parameters). These are then filtered based on their visibility and compatibility for each modification point. For instance, if a method is `private` and the buggy line is outside its class, it is filtered out as calling this method would result in a compiler error due to its inaccessibility.

Table 1: A comparison of types of patch ingredients supported by ingredient symbol screening of ARJA and ARJACLM. A checkmark is provided only if the respective technique can handle all symbols in the input.

	Symbol Visibility		Type Compatibility	
Statement	ARJA	ARJACLM	ARJA	ARJACLM
`x = y;`	✓	✓	✓	✓
`x = y + 1;`	✓	✓	✗	✗
`f(x, y);`	✓	✓	✓	✓
`z = f(x, y);`	✗	✓	✗	✓
`z = f(x, 1);`	✗	✓	✗	✗
`f(x).y();`	✗	✓	✗	✓
`f(x).y(z);`	✗	✓	✗	✓

In comparison to ARJA, the ingredient screening procedure in ARJACLM validates more code symbols, and is more strict as a result. Table 1 shows a comparison of types of patch ingredients supported by the ingredient symbol screening procedures of ARJA and ARJACLM. Symbol visibility checking assesses whether a symbol is within scope at a particular location in the code, while type compatibility checking determines whether code symbols do not violate type constraints. ARJACLM is capable of resolving the visibility of all code symbols (e.g., fields, methods) within a patch ingredient, whereas ARJA struggles with method invocations which do not make up the entire statement. For instance, ARJA can screen `f(x, y);`, but fails to handle `z = f(x, y);` as the method invocation is nested in an assignment statement. Neither ARJA nor ARJACLM can evaluate the type compatibility of results of unary operators, binary operators and literals. Finally, both ARJA and ARJACLM ignore code elements that they cannot screen, rather than rejecting the entire ingredient. Thus, false positives can arise if code symbols which violate visibility or type constraints are presented within code elements not supported by the screening procedure.

We also adopt six rules from ARJA to restrict certain programming instructions. For example, `continue` and `break` statements can only be used as ingredients for a buggy statement within a loop. Additionally, we apply six rules from ARJA to control operations on specific ingredients. For instance, an assignment statement cannot be inserted before another assignment statement that modifies the same left-hand side as the inserted assignment has no effect on the program's behaviour. Additional information about the ingredient screening rules is available on the paper's webpage [30].

2.4 Genetic Repair Algorithm Details

This section outlines how we adapted a multi-objective genetic algorithm to evolve patches that modify specific points in the code to fix bugs. ARJACLM employs the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [31], which is designed to optimize multiple objectives simultaneously.

2.4.1 ***Solution Representation.*** We adapted the solution representation introduced by ARJA [8] to encode patches. We begin by arranging the modification points in a *random order*. Each modification point is represented by three components: the enabled/disabled status, the patch operations, and the patch ingredients. For instance, a solution with three modification points is illustrated in Fig. 3.

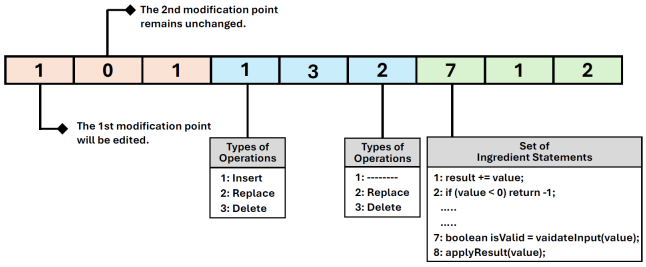


Figure 3: Patch representation in ARJACLM.

The enabled/disabled status (highlighted in orange in Fig. 3) determines whether a modification will be applied to the buggy program. The patch operation (highlighted in blue) specifies which operators (insert, replace, and delete) will be applied, and the patch ingredient (highlighted in green) determines the ingredient used. For instance, in Fig. 3, the first modification is applied since its edit status is true, and the statement `boolean isValid = validateInput(value);` (marked as number 7 in the patch ingredient) is inserted (as indicated by number 1 in the patch operation) before the buggy line.

While we follow the ARJA representation, our approach introduces two key differences. First, unlike the public implementation of ARJA[1], ARJACLM allows new patch ingredients to be introduced dynamically. This flexibility means patch edits are not restricted to pre-defined, redundancy-based ingredients; instead, novel ingredients generated ad hoc by a CLM can be directly incorporated. Second, our representation permits patch ingredients to consist of an arbitrary number of statements, rather than being limited to a single statement, thus providing CLMs with greater freedom in generating code. Further details are provided in Section 2.4.4.

2.4.2 ***Population Initialization.*** ARJACLM constructs the initial population randomly in the same manner as ARJA [8]. Each modification point is initialized with a randomly selected patch operation and redundancy-based ingredient determined for that modification point, using a uniform distribution. However, edits are initialized as enabled with a probability of $suspciousness \times \mu$, where *suspiciousness* is the suspiciousness score of the respective modification point, and μ is a configurable parameter of ARJACLM. As a result, more suspicious statements are more likely to be modified by patches in the initial population, which guides genetic search to explore the modification of more suspicious code.

2.4.3 ***Fitness Evaluation.*** The multi-objective genetic algorithm used in ARJACLM aims to minimize the same objectives as those employed by the ARJA: the weighted failure rate, and patch size. Specifically, the weighted failure rate is defined by Equation 1, where T_n represents the set of negative tests, and T_p denotes the set of sampled positive tests, and w is a configurable parameter with a range of $0 \leq w \leq 1$.

$$f_1(x) = \frac{|\{t \in T_n \mid x \text{ fails } t\}|}{|T_n|} \times (1-w) + \min(1, \frac{|\{t \in T_p \mid x \text{ fails } t\}|}{5}) \times w \quad (1)$$

[1]https://github.com/yyxhdy/arja

The weighted failure rate f_1 differs from ARJA in how it computes the failure rate for positive tests. Specifically, ARJACLM divides the number of failed positive tests by 5, instead of the number of positive tests adapted by ARJA. This adjustment is made because T_p (the set of positive tests) is usually much larger than T_n (the set of negative tests) and can be vary significantly between different buggy programs. As a result, this approach makes f_1 a more sensitive and consistent measure of the failure ratio for positive tests. Patches where $f_1(x) = 0$ are considered test-adequate.

Our second objective, the patch size as defined in Equation 2, is the count of enabled edits within the patch.

$$f_2(x) = |\{e \in x \mid e \text{ is enabled}\}| \quad (2)$$

The genetic algorithm in ARJACLM simultaneously minimizes f_1 and f_2, aiming to optimize for test-adequate patches that perform minimal modifications to the program. Patches that either fail to compile, exceed the test execution timeout or have zero enabled edits, are assigned a fitness value of $+\infty$ for both objectives.

2.4.4 ***Genetic Operators.*** *Crossover* and *mutation* were used as genetic operators to evolve the current solutions. Crossover combines parent patches to create offspring that inherit the good traits from their parents. Conversely, mutation introduces new patch elements into the population by modifying the resulting offspring.

The crossover is applied N/2 times per generation, producing N offspring patches, where N is equal to the population size. In this study, *Binary tournament* selection is used to determine parents for crossover. In this process, two individuals are randomly selected, and the one with the superior fitness is chosen as a parent. In order of occurrence, a patch wins the tournament if: 1. The patch compiles successfully while the other does not. 2. The patch is test-adequate while the other is not. 3. The patch is dominated by fewer individuals than the other [31]. 4. The patch has a greater crowding distance compared to the other [31]. A tournament patch is selected randomly if these criteria fail to determine a winner. After selecting the parents and performing crossover, the mutation is applied to each offspring to introduce genetic diversity.

Crossover: Fig. 4 illustrates the crossover operator employed by ARJACLM. Crossover is performed independently to edit enabled/disabled status, patch operations and patch ingredients.

Edit statuses are exchanged between parents with a 0.5 probability if they differ. As shown in Fig. 4, the first edit statuses are exchanged between the two parent individuals. For combining patch operations, a *single-point* crossover is employed. A cut point is randomly selected, and all operations beyond that point are exchanged between the two parents to produce two offspring. Single-point crossover is also used to combine patch ingredients, but with a separate, randomly chosen cut point as shown in Fig. 4.

Mutation: ARJACLM uses two mutation operators: *redundancy mutation*, adapted from ARJA [8], which operates under the redundancy assumption that the code needed to generate a repair (referred to as fix ingredients) is already present elsewhere in the buggy program [8, 9, 15], and *CLM mutation*, which utilizes a CLM to generate patch ingredients. The CLM mutation is selected with a configurable probability p_{clm} (e.g., 0.4). If not, the redundancy mutation is selected.

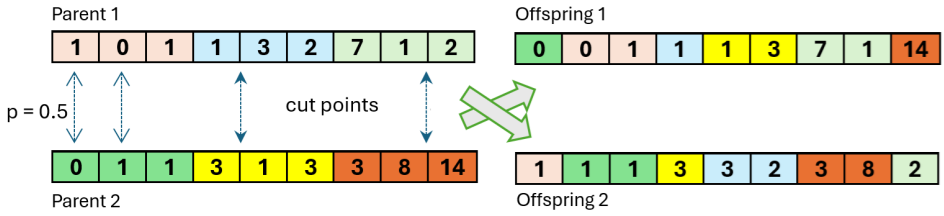


Figure 4: Illustration of the ARJACLM Crossover Operation.

The *redundancy mutation* is used to i) enable or disable edits, ii) modify the patch operation and iii) modify redundancy-based ingredient. Similar to ARJA, with a probability of p_{mut}, the patch operations and ingredients are replaced with a randomly selected alternative of the modification point. However, unlike ARJA, which enables and disables edits with equal probability, ARJACLM employs distinct probabilities for this action. Specifically, in ARJACLM, edits are enabled with probability p_{mut}, but disabled with probability $p_{mut} \times (|\{e \in edits \mid e \text{ is enabled}\}| + 1)$. As a result, the redundancy mutation converges towards patches with a lower number of enabled edits, which are more likely to be similar to developer-written patches. The mutation probability p_{mut} is set to 0.1/n, where n is the number of modification points, ensuring that the total mutation rate remains consistent across different numbers of modification points.

Relying only on redundancy mutation prevents ARJACLM to fix bugs if the required patch ingredients are not present elsewhere in the code. To tackle this issue, ARJACLM introduces a novel *CLM mutation* that leverages a CLM (i.e., CodeLLaMa [35]) to dynamically generate patch ingredients. ARJACLM generates patch ingredients using *mask prediction*. Mask prediction is the most straightforward method for obtaining infills for arbitrary code locations, where a buggy line of code is masked, and the model must provide accurate replacements. The model receives the prefix and suffix surrounding the masked line as input and should fill in the masked line correctly. Fig. 5 shows a scenario where a buggy line (highlighted in red) is masked, and a potential fix is generated by the employed CLM.

The CLM mutation enables and disables patch edits in the same way as the redundancy mutation. Additionally, a patch ingredient is generated by the CLM with probability p_{mut}, but only if the edit is enabled. This approach is due to the high cost of generating patch ingredients with CLMs. Finally, when a patch ingredient is generated, either an *insert* or *replace* operation is randomly selected.[2]

It is important to mention that ARJACLM leverages screening rules for combining patch ingredients and operations, as detailed in Section 2.3, to evaluate patches resulting from crossover and mutation. If a patch is incompatible with the corresponding buggy statement, it is corrected by randomly selecting an alternative redundancy-based ingredient. The patch is discarded if no such ingredient is available. This approach prevents the construction of patches containing anti-patterns without reducing the mutation and crossover rate. However, *this screening does not apply to patch ingredients generated by a CLM*. As a result, CLMs are provided with more control over what code can be generated. If a CLM is adequately powerful, this can yield unique new patches of high quality for complex code elements.

[2]Future work will adopt ARJA-e's method [10] for operator selection by ingredients.

2.4.5 ***Next Generation Formation and Stopping Criteria.*** Each generation of the genetic algorithm ends with the replacement of individuals in the population with new ones generated through crossover and mutation. However, to ensure the best individuals from the current population are preserved, a subset of elite individuals is directly carried over to the next generation. Binary tournament selection, as described in Section 2.4.4, is then used to fill the remaining spots in the new population with offspring.

This iterative process continues until the algorithm reaches the maximum number of generations, and solutions that meet the test adequacy criteria are selected as plausible patches. If no such solutions are found, the ARJACLM fails to find a patch for the bug.

3 Evaluation

To evaluate the feasibility of our approach in a practical environment, we performed experiments on real-world bugs and compared our tool with other search-based tools. This section details our experimental setup and evaluation results.

3.1 Research Questions

Our experiment aims to answer the following research questions:

- **RQ1**: What is the impact of incorporating CLM-generated patch ingredients on the overall bug-fixing performance of ARJACLM?
- **RQ2**: How does ARJACLM perform compared to other search-based automated program repair tools, i.e., ARJA and GenProg?
- **RQ3**: How does the quality of CLM-generated patch ingredients compare to redundancy-based patch ingredients?
- **RQ4**: How efficient is ARJACLM in terms of time?

3.2 Employing Defect4J for Evaluation

In our study, we utilized Defects4J version 2.0.1 [28] to address our research questions. We excluded certain systems and bugs primarily due to *flaky* tests, which produce inconsistent results because of randomness or time-based behaviour. While Defects4J offers tools to filter out flaky test failures and ensure a consistent execution environment, these tools are not compatible with ARJACLM and our fault localisation tool, GZoltar [32], both of which require direct test execution. Additionally, some buggy systems failed to compile directly or required specific settings unsupported by our instrumentation process, as noted in previous studies like ARJA [8] and ARJANMT [11]. Ultimately, 9 out of 17 Java projects and 398 out of 437 bugs in these projects met our initial criteria. However, due to the substantial computational cost involved in evaluating ARJACLM, we *randomly* selected a subset of these qualified bugs, limiting our analysis to a maximum of 20 bugs per project. As a result, ARJACLM was evaluated on 176 bugs, as shown in Table 2.

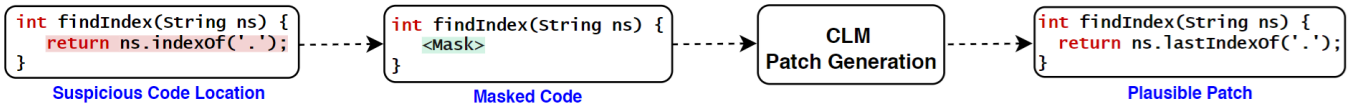


Figure 5: CLM Mutation with Mask Prediction.

Table 2: Overview of the number of totals and evaluated bugs for the 9 projects considered from Defects4J version 2.0.1.

Project Name	#Total Bugs	#Qualified Bugs	#Evaluated Bugs
JFreeChart	26	26	20
Commons CLI	39	38	20
Commons Codec	18	18	18
Commons Compress	47	45	20
Gson	18	18	18
Jsoup	93	91	20
Commons Lang	64	38	20
Commons Math	106	99	20
Joda-Time	26	25	20
Total	**437**	**398**	**176**

3.3 Selection of CLM for Patch Generation

Selecting the right CLM can significantly impact the *effectiveness*, and *efficiency* of ARJACLM. Therefore, in a separate study, we first evaluate 20 CLMs capable of generating code infills, varying in the number of parameters from 60 million to 16 billion, and compare them on their ability to generate correct bug fixes, resource consumption, compilability rate, and patch diversity. We observed that CodeLLaMA-13B [35] performs better in bug fixing and compiler error handling. Therefore, in this study, we used CodeLLaMA-13B [35] for generating patch ingredients when CLM mutation is called.

3.4 Algorithm Parameters and Settings

Table 3 displays the parameters and their values used in ARJACLM for the experiments conducted in this study. These values are based on ARJA [8] and preliminary experimentations conducted in this study. We identify two key parameters, the *CLM mutation probability* (p_{clm}) and the *number of context lines for mask prediction prompts* (C_{mp}), which significantly affect the integration of CLMs into ARJACLM. Further details on these two parameters and their impact on CLM's ingredient generation are provided in Section 3.7.

Table 3: Experimental Parameter Settings for ARJACLM.

Parameter	Description	Default Value
N	Population size	40
G	Maximum number of generations	20
γ_{min}	Suspiciousness threshold	0.1
n_{max}	Maximum number of modification points	40
μ	Scale for number of enabled edits in the initial population	0.06
w_{pos}	Positive test weight	0.33
e	Elite count	1
m_{mut}	Mutation probability multiplier	0.1
p_{mut}	Mutation probability	m_{mut}/n
p_{clm}	CLM mutation probability	0.4
C_{mp}	Number of lines of context for mask predict prompts	100

3.5 Experimental Hardware Setup

All experiments were performed on machines equipped with a 2.1 GHz Intel Xeon Silver 4216 processor, 32 GB of memory, and a single NVidia A40 GPU with 48 GB of VRAM. Although the CPU was shared with other users, 16 physical cores are allocated for up to 15 parallel patch evaluations to prevent starvation of CPU capacity. The GPU, however, was exclusively dedicated to the APR process and was not shared with other users. For experiments involving CodeLLaMA, 8-bit quantization was used to manage VRAM usage effectively, as providing more context without quantization would exceed the 48 GB VRAM limit for context sizes greater than 200.

3.6 Experimental Execution and Measurement

All experiments performed on ARJACLM are executed three times due to the stochastic nature of the employed search-based technique. Additional trials could provide more significant results, but this is not feasible due to the high cost involved in the evaluation of APR techniques on Defects4J. Nevertheless, three trials should provide adequate metrics for the performance of ARJACLM. For all metrics, the *average* value across all trials is reported as the overall result.

3.7 RQ1: Impact of CLM-Generated Patches

This research question explores how CLM-generated patch ingredients impact the overall bug-fixing performance of ARJACLM.

As discussed in Section 3.4, two parameters significantly influence CLM integration: the *CLM mutation probability* (p_{clm}), which determines the likelihood that CLM is used, and the *number of context lines for mask prediction prompts* (C_{mp}), which specifies the amount of code context around buggy statement provided to the CLM. To identify the optimal values for these two parameters, we run experiments varying each parameter *independently*. We tested p_{clm} values of 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0, and C_{mp} values of 100, 200, 400, and 800 as shown in Table 4, while keeping other parameters at their default settings as specified in Table 3. Note that C_{mp} is evaluated up to 800 lines of code to stay within the 16k token context limit of CodeLLaMA-13B [35].

As shown in Table 4 for 176 evaluated bugs from Defect4J, ARJACLM achieves optimal performance with higher p_{clm} values. Specifically, a **34.5%** improvement is observed at p_{clm} = 1.0 compared to p_{clm} = 0.0, where only *redundancy mutation* was used. Although this comes with a **350%** increase in computation time, it shows that CLM-based patch ingredients yield a more effective search-based APR technique compared to the redundancy assumption.

The results for C_{mp} show that providing more context is only valuable to a limited extent. While more context should theoretically improve CLM performance by providing additional code patterns and symbols, the best performance for ARJACLM is observed when roughly half of the available context size is used (i.e., 400). Moreover, a larger context size results in a longer execution time.

Table 4: Results for the evaluation of ARJACLM.

Parameter	Value	#Fixed Bugs	Total Time (in hours)
p_{clm}	0.0	29.0	19.6
	0.2	30.3	37.1
	0.4	33.0	53.3
	0.6	33.7	66.3
	0.8	36.0	78.7
	1.0	**39.0**	88.3
C_{mp}	100	31.8	27.6
	200	32.6	30.5
	400	**35.6**	33.0
	800	33.4	37.2

The results provide evidence that CodeLLaMA cannot effectively leverage its full context size in the setting of ARJACLM. It is currently unclear whether this observation is a result of the limited capability of CodeLLaMA to deal with larger contexts, or whether the setting of ARJACLM prevents them from doing so.

Summary of Results for RQ1

The integration of CLM-generated patch ingredients significantly enhances the bug-fixing performance of ARJACLM, with the best results achieved when *CLM mutation* is fully utilized. Additionally, providing a moderate amount of context (400 lines) yields the most effective results, indicating that more context is beneficial only up to a certain point.

3.8 RQ2: Comparing with Other SB Techniques

We evaluate ARJACLM against ARJA [8] and GenProg [29], both open-source tools capable of fixing bugs in Java programs using search-based techniques. ARJACLM builds on ARJA's framework, making this comparison crucial for evaluating its effectiveness and improvements. ARJA is evaluated on Defects4J 1.0 bugs from the JFreeChart, Joda-Time, Commons Lang and Commons Math projects [8]. Motwani et al. [29] evaluate GenProg on the entirety of Defects4J 1.0, which includes all bugs from the aforementioned projects. ARJACLM is evaluated on 20 bugs for each of the aforementioned projects. Therefore, we compare bug-fixing performance across these **80 repair tasks** for ARJACLM, ARJA and GenProg.

ARJA is evaluated on Defects4J with a single trial, while GenProg is tested with 30 trials. We report the best result for the GenProg to avoid underestimating its capabilities. For ARJACLM, results are averaged over three runs. Table 5 compares these tools on 80 repair tasks. ARJACLM_n represents ARJACLM without the use of CLM.

Note that the optimal parameters determined for p_{clm} and C_{mp} were not used in this research question. Instead, we used the parameter values listed in Table 3 as it facilitates a comparison of the efficiency of the employed search-based techniques, as well as makes it possible to measure the effect of redundancy and CLM mutations while keeping bug-fixing durations reasonable.

When examining Table 5, the most noticeable observation is that while ARJACLM_n builds on ARJA's framework, it yields weaker results than ARJA. As previously discussed, our implementation of ARJACLM_n differs from ARJA in several aspects, such as employing different test case sampling, search space optimisation, redundancy mutation, and the use of a different fitness function, among others.

Table 5: Comparison of GenProg, ARJA and ARJACLM on 80 repair tasks. ARJACLM_n denotes ARJACLM without CLM.

Project	GenProg	ARJACLM_n	ARJA	ARJACLM
JFreeChart	4	7	8	9.7
JodaTime	1	1	4	3.0
Commons Lang	2	2	3	5.0
Commons Math	4	4	5	5.3
Total	11	14	20	23

Although these modifications were intended to enhance accuracy, they may have inadvertently introduced false negatives. It is clear that these changes were not fully optimized, and ARJACLM_n, as a novel search-based APR technique, requires further refinement to match ARJA's capabilities. However, ARJACLM_n outperforms GenProg and can provide a framework for evaluating the incorporation of CLMs.

As shown in Table 5, ARJACLM achieved the best results, fixing 23 out of 80 bugs—64% better than ARJACLM_n. It is clear that CLM-generated patch ingredients can successfully contribute to improved repair performance. However, ARJACLM only slightly outperformed ARJA on the limited set of repair tasks (23 vs. 20), suggesting that a more efficient search technique, at least as effective as ARJA, could further improve results.

Summary of Results for RQ2

ARJACLM outperforms other search-based techniques, especially ARJACLM_n and shows promise with CLM-generated ingredients. Yet, a more efficient search-based technique could improve results.

3.9 RQ3: CLM vs. Redundancy-Based Patches

Table 6 compares the quality of CLM and redundancy-based patch ingredients. For ARJACLM_n, 60.6% of redundancy-based patches compile successfully. In contrast, for ARJACLM, 43.5% of CodeLLaMA-generated patch ingredients are syntactically valid Java instructions, and 68.2% of them compile successfully. However, ARJACLM evaluated fewer patches than ARJACLM_n (334.6 vs. 563.4), indicating that fewer final CLM patches are tested to verify the bug is fixed, which shows the overall accuracy of CLM-generated patch ingredients.

A manual analysis of syntactically incorrect infills was performed to determine the cause of syntax errors. In some cases, we observed that CodeLLaMA attempts to complete a function rather than provide infill for the mask token provided inside of it. This occurred in the case where there was a bug elsewhere in the code. In this case, CodeLLaMA attempts to complete the code and does so in a correct manner. However, this hallucination produces a code completion rather than an infill, which results in syntax errors when the generated code is inserted into the existing code.

Summary of Results for RQ3

The results show that only 30% of CLM-generated patches are compilable compared to 60% for redundancy-based patches, but CLM patches perform better as fewer patches were tested for their ability to fix bugs.

Table 6: Quality of CLM vs. Redundancy-Based Patches.

	#Evaluated	CLM		Redundancy
Technique	Patches	Parse Rate	Compilation Rate	Compilation Rate
$ARJACLM_n$	563.4	-	-	60.6%
ARJACLM	334.6	43.5%	68.2%	60%

3.10 RQ4: Time Efficiency of ARJACLM

Table 7 shows the computation time for two versions of ARJACLM. $ARJACLM_n$ is significantly faster than ARJACLM. This difference is expected due to the resource demands of CodeLLaMA, a 13-billion-parameter model used in this study. Specifically, $ARJACLM_n$ averages 7.9 minutes per bug, while ARJACLM averages 32.3 minutes, representing a 309% increase. Indeed, ARJACLM on average requires 26.9 minutes to generate ingredients with CodeLLaMA, leaving on average about 5.4 minutes per bug for patch validation. This is an improvement over $ARJACLM_n$ (5.4 vs. 7.9 minutes), and it is mainly result because fewer patches were evaluated in ARJACLM as discussed in the previous research question.

Table 7: Comparison of Time per Bug for ARJACLM Variants.

	Time per Bug (minutes)				CLM Time per Bug (minutes)			
Technique	Min	Median	Max	Avg	Min	Median	Max	Avg
$ARJACLM_n$	0.6	6.6	30.5	7.9	-	-	-	-
ARJACLM	0.8	31.8	64.8	32.3	0	26.8	60.6	26.9

Summary of Results for RQ4

ARJACLM requires significantly more time than $ARJACLM_n$, averaging 32.3 minutes per bug compared to 7.9 minutes, due to the resource demands of the employed CLM. However, the additional time may be justified compare to the time and financial cost of manual bug fixing.

4 Threats to Validity

We conducted experiments on a subset of Defects4J [28], which may not cover the actual distribution of real-world bugs. Our parameter tuning may not also be generalizable as it is possible that the selected parameter values are only effective for bugs included in the experiments. The stochastic nature of both our search-based approach and the employed CLM is another concern. While we evaluated ARJACLM in three trials, running the approach on *additional datasets* and conducting *more trials* for each would help reduce result variance and provide a more accurate assessment.

Another validity concern involves the training of CLMs on large public data repositories. Indeed, the possibility that any generated patch has been seen before by the model should be considered a concern. However, since the employed CLM, CodeLLaMA, is not trained on pairs of buggy and fixed code fragments, this limits the impact of the training data on experimental results.

Finally, the validity of this empirical study is limited by the novel ARJACLM technique and its implementation. $ARJACLM_n$ is intended to replicate ARJA, but fails to provide the same bug-fixing capabilities. Therefore, *this study does not provide direct evidence that CLMs can augment state-of-the-art search-based techniques*. Future research should investigate whether the relative performance gain of ARJACLM over $ARJACLM_n$ can be replicated for ARJA.

5 Related Work

APR tools can be categorized into *four* primary groups [2–7] : i) *constraint-based* APR tools, such as Nopol [36], which derive repair constraints from inferred or provided specifications and synthesizes patches that satisfy these constraints. However, these tools are less effective for software lacking specifications, especially legacy applications [7]; ii) *search-based* APR tools, such as GenProg [9], and ARJA [8], search a patch space by iteratively refining patches through mutating the suspicious statements until a satisfactory solution is found. However, these tools struggle with ineffective patch ingredient generation and semi-random changes [7]; iii) *template-based* APR tools, such as TBar [37], employ predefined repair templates, offering a structured approach that reduces randomness and variability compared to search-based methods [7], though they may be constrained by the availability and quality of templates [3, 7]. Template-based methods may also encounter challenges when the fixed pattern is correctly identified, but fixed ingredients are not locally available [16]; iv) *learning-based* APR tools, such as CURE [38], automatically learn bug-fixing patterns and also provide patch ingredients derived from a large number of labelled code examples, thereby potentially improve limitations of template-based approaches [38, 39]. However, the quality of code examples used in learning significantly affects accuracy, and obtaining high-quality code examples is often challenging and costly. Recent advancements in CLMs have significantly enhanced learning-based APR tools, such as AlphaRepair [12], by leveraging their deep understanding of code semantics and context to improve patch generation [3, 5, 6]. However, these models still generate a substantial number of syntactically or semantically incorrect patches [26], and more importantly, because CLMs are not specifically trained for APR tasks, their performance in this domain remains limited. These limitations have prompted researchers to integrate CLMs with other techniques to enhance their effectiveness in generating accurate patches [11, 13, 25, 27, 40–43] as done in this study by combining CLMs with search-based technique.

Building on CLM integration with other techniques, Zhang et al. [13] propose GAMMA, which uses CLMs to generate patch ingredients for template-based APR. GAMMA demonstrates that CLM-generated ingredients can be of higher quality than their redundancy-based ones, as previously demonstrated with a learning-based model [11]. As another approach, Ribeiro and Abreu [40] and Wei et al. [25] treat the APR process as a code compilation process. Ribeiro and Abreu [40] first identify the location of the bug and then provide the statements preceding the buggy location to CodeGPT for code completion, thereby replacing potentially buggy statements with newly generated ones. Wei et al. [25] use CLMs to generate patch ingredients for code completion. However, their tool, Repilot, further refines these suggestions by removing infeasible tokens and proactively completing the code based on the recommendations of an implemented completion engine. Liu et al. [27] also demonstrate that patches generated by the learning-based models can be further improved with simple edits (i.e., deleting or inserting statements) to better fit the context. Researchers such as Xia et al. [41], and Zhang et al. [42] propose enhancing CLM results through *conversational interactions*, leveraging feedback types such as test failures and iterative prompts to refine generated patches.

6 Conclusion and Future Work

In this study, we introduced ARJACLM, a tool built on ARJA [8], the leading *search-based* automated program repair tool. ARJACLM enhances ARJA by integrating CodeLLaMA to improve patch ingredient generation. ARJACLM outperforms other search-based techniques, including GenProg [9], ARJA [8] and ARJACLM$_n$ (the version without CLM). Although CLM patches are less compilable than redundance-based patches, they require fewer test validations to achieve correct results. However, ARJACLM requires significantly more time to generate patch ingredients.

Looking ahead, we plan to explore several avenues for future work that could positively impact both the effectiveness and efficiency of ARJACLM. ARJACLM masks the entire buggy statement and uses CLM to generate a replacement. However, *partial masking* as demonstrated by AlphaRepair [12] and *template-based masking* as seen in GAMMA [13], have been proven to enhance patch accuracy. In addition, the current method integrates generated ingredients into the buggy program without modifications. However, previous research [25, 27, 43] indicates that applying minor adjustments to generated ingredients helps resolve issues with incompatible instructions through simple edits. Additionally, recent studies [41, 42] reveal that engaging in conversational interactions with CLMs and providing *automated* feedback such as test failures improves the quality of the generated patches.

7 Acknowledgment

In this paper, we utilized GPT-3.5 to improve and validate the overall quality of the text, including grammar and spelling.

References

[1] H. Krasner, "The Cost of Poor Software Quality in the US: A 2022 Report," *Consortium for Information & Software Quality*, 2022.
[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, 2019.
[3] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A Survey of Learning-Based Automated Program Repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, 2023.
[4] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, 2018.
[5] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, 2024.
[6] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, and Y. Y. Z. Chen, "A Systematic Literature Review on Large Language Models for Automated Program Repair," *arXiv preprint arXiv:2405.01466*, 2024.
[7] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A Survey on Automated Program Repair Techniques," *arXiv preprint arXiv:2303.18184*, 2023.
[8] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, 2018.
[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, 2011.
[10] Y. Yuan and W. Banzhaf, "A Hybrid Evolutionary System for Automatic Software Repair," in *Proceedings of the GECCO*, 2019.
[11] D. Li, W. E. Wong, M. Jian, Y. Geng, and M. Chau, "Improving Search-Based Automatic Program Repair with Neural Machine Translation," *IEEE Access*, vol. 10, 2022.
[12] C. S. Xia and L. Zhang, "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning," in *Proceedings of the FSE*, 2022.
[13] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "GAMMA: Revisiting Template-based Automated Program Repair via Mask Prediction," in *Proceedings of the ASE*. IEEE, 2023.
[14] M. Martinez, W. Weimer, and M. Monperrus, "Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches," in *Proceedings of the ICSE Companion*, 2014.
[15] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The Plastic Surgery Hypothesis," in *Proceedings of the FSE*, 2014.
[16] D. Yang, K. Liu, D. Kim, A. Koyuncu, K. Kim, H. Tian, Y. Lei, X. Mao, J. Klein, and T. F. Bissyandé, "Where Were the Repair Ingredients for Defects4j Bugs? Exploring the Impact of Repair Ingredient Retrieval on the Performance of 24 Program Repair Systems," *Empirical Software Engineering*, vol. 26, 2021.
[17] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-Written Patches," in *Proceedings of the ICSE*. IEEE, 2013.
[18] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *Proceedings of the ICSE*, 2014.
[19] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, 2021.
[20] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proceedings of the ICSE*. IEEE, 2023.
[21] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of Code Language Models on Automated Program Repair," in *Proceedings of the ICSE*. IEEE, 2023.
[22] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair," in *Proceedings of the ASE*. IEEE, 2023.
[23] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2023.
[24] C. S. Xia, Y. Ding, and L. Zhang, "The Plastic Surgery Hypothesis in the Era of Large Language Models," in *Proceedings of the ASE*. IEEE, 2023.
[25] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair," in *Proceedings of the ESEC/FSE*, 2023.
[26] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and Evaluating Hallucinations in LLM-Powered Code Generation," *arXiv preprint arXiv:2404.00971*, 2024.
[27] C. Liu, P. Cetin, Y. Patodia, B. Ray, S. Chakraborty, and Y. Ding, "Automated Code Editing with Search-Generate-Modify," in *Proceedings of the ICSE Companion*, 2024.
[28] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the ISSTA*, 2014.
[29] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of Automated Program Repair on Real-World Defects," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, 2020.
[30] "Dataset," 2024, https://doi.org/10.5281/zenodo.14222432.
[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE transactions on Evolutionary Computation*, vol. 6, no. 2, 2002.
[32] "The GZoltar Toolset," 2024, https://gzoltar.com/index.html.
[33] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An Eclipse Plug-in for Testing and Debugging," in *Proceedings of the ASE*, 2012.
[34] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2006.
[35] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code LlaMA: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.
[36] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, 2016.
[37] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-Based Automated Program Repair," in *Proceedings of the ISSTA*, 2019.
[38] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *Proceedings of the ICSE*. IEEE, 2021.
[39] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, 2019.
[40] F. Ribeiro, R. Abreu, and J. Saraiva, "Framing Program Repair as Code Completion," in *Proceedings of the International Workshop on Automated Program Repair*, 2022.
[41] C. S. Xia and L. Zhang, "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.
[42] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "PyDex: Repairing Bugs in Introductory Python Assignments using LLMs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, 2024.
[43] A. E. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Enhancing Genetic Improvement Mutations Using Large Language Models," in *Proceedings of the SBSE*. Springer, 2023.