RESEARCH-ARTICLE

PReMM: LLM-Based Program Repair for Multi-method Bugs via Divide and Conquer

LINNA XIE, Nanjing University, Nanjing, Jiangsu, China

ZHONG LI, Nanjing University, Nanjing, Jiangsu, China

YU PEI, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

ZHONGZHEN WEN, Nanjing University, Nanjing, Jiangsu, China

KUI LIU, Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China

TIAN ZHANG, Nanjing University, Nanjing, Jiangsu, China

View all

Open Access Support provided by:

Nanjing University

The Hong Kong Polytechnic University

Huawei Technologies Co., Ltd.

Published: 09 October 2025
Accepted: 12 August 2025
Received: 26 March 2025

PReMM: LLM-Based Program Repair for Multi-method Bugs via Divide and Conquer

LINNA XIE, Nanjing University, China
ZHONG LI*, Nanjing University, China
YU PEI*, The Hong Kong Polytechnic University, China
ZHONGZHEN WEN, Nanjing University, China
KUI LIU, Huawei, China
TIAN ZHANG* and XUANDONG LI, Nanjing University, China

Large-language models (LLMs) have been leveraged to enhance the capability of automated program repair techniques in recent research. While existing LLM-based program repair techniques compared favorably to other techniques based on heuristics, constraint-solving, and learning in producing high-quality patches, they mainly target bugs that can be corrected by changing a single faulty method, which greatly limits the effectiveness of such techniques in repairing bugs that demand patches spanning across multiple methods. In this work, we propose the PReMM technique to effectively propose patches changing multiple methods. PReMM builds on three core component techniques: the faulty method clustering technique to partition the faulty methods into clusters based on the dependence relationship among them, enabling a divide-and-conquer strategy for the repairing task; the fault context extraction technique to gather extra information about the fault context which can be utilized to better guide the diagnosis of the fault and the generation of correct patches; the dual-agent-based patch generation technique that employs two LLM-based agents with different roles to analyze the fault more precisely and generate patches of higher-quality. We have implemented the PReMM technique into a tool with the same name and applied the tool to repair real-world bugs from datasets Defects4J V1.2 and V2.0. PReMM produced correct patches for 307 bugs in total. Compared with ThinkRepair, the state-of-the-art LLM-based program repair technique, PReMM correctly repaired 102 more bugs, achieving an improvement of 49.8%.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Correctness**.

Additional Key Words and Phrases: Automated Program Repair, Multi-method Bugs, Large Language Models, Divide and Conquer, Context-Aware Repair

ACM Reference Format:
Linna Xie, Zhong Li, Yu Pei, Zhongzhen Wen, Kui Liu, Tian Zhang, and Xuandong Li. 2025. PReMM: LLM-Based Program Repair for Multi-method Bugs via Divide and Conquer. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 319 (October 2025), 29 pages. https://doi.org/10.1145/3763097

*Corresponding authors.

Authors' Contact Information: Linna Xie, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, xieln@smail.nju.edu.cn; Zhong Li, State Key Laboratory for Novel Software Technology, Software Institute, Nanjing University, Nanjing, China, lizhong@nju.edu.cn; Yu Pei, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China, yupei@polyu.edu.hk; Zhongzhen Wen, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, wenzhongzhen@smail.nju.edu.cn; Kui Liu, Huawei Software Engineering Application Technology Lab, Huawei, China, brucekuiliu@gmail.com; Tian Zhang, ztluck@nju.edu.cn; Xuandong Li, lxd@nju.edu.cn, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China.

ACM 2475-1421/2025/10-ART319
https://doi.org/10.1145/3763097

1 Introduction

Automated Program Repair (APR) aims to generate patches for bugs[1] in software programs automatically, and it is a promising way to alleviate the arduous burden of manual debugging from software developers and to enhance software reliability. Researchers and practitioners have developed various APR techniques over the past decades based on techniques like heuristic search [21, 22, 52], constraint solving [10, 20, 35, 39], and machine learning [11, 16, 17, 40, 57, 58, 61].

Recently, with the rapid development of large language models (LLMs) and their swift adoption in software engineering practices, researchers have also explored leveraging these models to enhance the capabilities of automated program repair techniques [3, 7, 12, 19, 25, 43, 50, 53, 54]. More concretely, existing LLM-based APR approaches take a faulty method and the other relevant information as input, generate a group of candidate patches that change the faulty method in various ways, and check whether each candidate patch is desirable through validation. This line of research has achieved remarkable results and represents the state of the art in APR [55, 59].

While LLMs have significantly improved the success rates of APR techniques, existing LLM-based APR approaches have a fundamental limitation, i.e., they assume the necessary changes to correct a defect are confined to a single method. However, a considerable number of real-world defects are more complex and demand patches spanning across multiple methods. For instance, over 20% of the bugs in the Defects4J datasets require multi-method patches. The limitation essentially renders these approaches ineffective in repairing multi-method faults because each single-method patch they generate is just a partial patch and will always fail to make all tests pass in validation.

In this work, we propose a novel technique named PReMM (Program Repair for Multi-method Bugs) to effectively generate high-quality multi-method patches for program defects via *divide-and-conquer*. Effective program repair with PReMM has three cornerstone techniques, namely *faulty method clustering*, *comprehensive fault context analysis*, and *dual-agent-based program repair*.

First, in view of the fact that, although patch validation in automated program repair is typically performed through testing, not every faulty method would influence the execution of every test case, PReMM performs *faulty method clustering* (FMC) to partition the faulty methods into mutually exclusive clusters with two faulty methods falling into the same cluster if and only if their patches may interfere with each other's validation. Based on the clustering result, PReMM *divides* the task of generating an appropriate multi-method patch for the whole buggy program (or a program patch) into a group of sub-tasks, each focusing on generating a desirable (single- or multi-method) patch for one cluster of faulty methods (or a cluster patch). Since clusters are disjoint, their patches can easily be *merged* to form a final program patch without introducing conflicts.

Second, even though each cluster typically contains a subset of all the faulty methods, and the task of generating cluster patches is most likely less formidable than that of generating program patches, it is still beyond the capability of existing LLM-based APR tools that aim to generate single-method patches if the cluster contains more than one faulty method. To achieve a better understanding of the fault and the expected patch, PReMM conducts *fault context extraction* (FCE) to extract rich contextual information about the fault in question. Here, the rich contextual information about a fault includes not only the usual suspects like fault location and failed tests, which existing LLM-based APR techniques utilize already, but also similar code segments from the current project, the invocation relation among the failed tests and the faulty methods, and the key tokens.

Third, to make better use of the gathered contextual information to guide fault diagnosis and patch generation, PReMM employs two LLM-based agents to generate desirable patches in iterations. In particular, the *fault analysis agent* is responsible for establishing accurate fault diagnoses, and the *repair generation agent* concentrates on deriving high-quality cluster patches from the diagnoses. If

[1]We use terms bug, defect, and fault interchangeably in this work.

the cluster patches can merge into a program patch that makes the buggy program pass all the tests, the program patch is valid, and PReMM outputs it as the repair result. Otherwise, if a cluster patch fails to compile or validate, the corresponding errors are fed back to the fault analysis agent to drive another round of dual-agent-based program repair.

We have implemented the PReMM technique into a tool with the same name to automatically repair multi-method bugs in Java programs. While PReMM primarily targets repairing buggy Java programs, the proposed approach is highly generic and can be easily migrated to support the automated repair of buggy programs written in other programming languages. Taking a buggy program, the location information about the bug, and a group of passing and failing test cases as the input, PReMM first partitions the faulty methods containing the buggy code into mutually exclusive clusters based on whether the patches to the methods can interfere with each other's validation and whether there is any invocation relation among them, and then gathers rich contextual information about the bug in question. Afterward, PReMM repeatedly employs two LLM-based agents to generate candidate patches for the faulty method clusters under the guidance of the extracted bug context information and validates the patches by compiling and running them against the tests, until a valid patch that can make the program pass all tests is found or the repair effort fails because the allocated time is used up or the maximum number of attempts have been exhausted.

To evaluate the effectiveness of PReMM, we conducted comprehensive experiments using real-world bugs from the well-known Defects4J benchmark dataset. Additionally, we compared the patches generated by our approach against those produced by 8 state-of-the-art automated program repair techniques, including, e.g., ThinkRepair [59], ChatRepair [55], and Mulpo [29]. Overall, PReMM produced *correct* patches, i.e., patches manually confirmed to be semantically equivalent to those written by developers, for 307 bugs, 86 of which (including 63 single-method bugs and 23 multi-method bugs) were not correctly repaired by any of the 8 baseline techniques before, representing unique contributions of our approach. When compared specifically with ThinkRepair, which offers the best overall effectiveness among existing program repair techniques on Defects4J, PReMM correctly repaired 102 more bugs, achieving a substantial improvement of 49.8% (=102/205).

Contributions. This paper makes the following contributions:

(1) We propose the PReMM approach to repairing multi-method bugs automatically and effectively. To the best of our knowledge, PReMM is the first generic approach to repairing bugs that span across multiple methods;
(2) We implement the PReMM approach into a tool with the same name and make it publicly downloadable to facilitate its easy application. The download package includes comprehensive instructions for adaptation and deployment on new datasets;
(3) We experimentally evaluate the repairing power of PReMM on the Defects4J benchmark and empirically demonstrate PReMM's effectiveness in repairing real-world bugs and its superiority over existing NMT- and LLM-based program repair techniques. We also evaluate PReMM and a variant of ThinkRepair on two additional benchmarks, Defects4J-Trans [24] and GitBug-Java [48], that are less likely to be affected by the issue to mitigate the threat posed by the potential issue of data contamination with Defects4J.

Structure. The rest of this paper is organized as follows. Section 2 explains with examples how the key components of PReMM help it produce high-quality patches in practice. Section 3 details how PReMM proposes patches to buggy programs step by step. Section 4 describes the experiments we conduct to evaluate PReMM and reports on the experimental results. Section 5 reviews recent work that is closely related to ours. Section 6 concludes the paper.

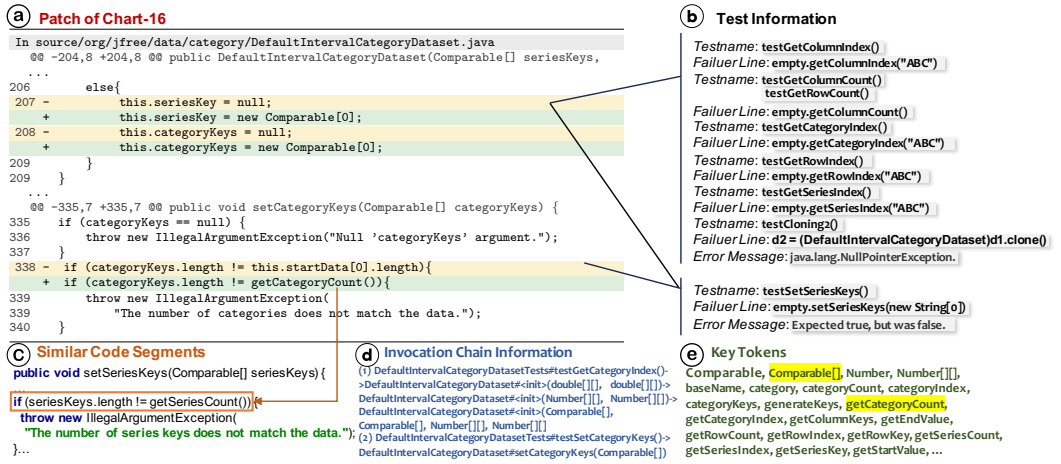


Fig. 1. Bug Chart-16 from Defects4J V1.2, Its Human-Written Patch, and the Contextual Information.

2 PReMM in Action

Figure 1a shows in the GNU diff format the changes programmers made to correct bug Chart-16 from the Defects4J benchmark [18]. The bug affected two methods `DefaultIntervalCategoryDataset`[2] and `setCategoryKeys` of class `DefaultIntervalCategoryDataset`. Method `DefaultIntervalCategoryDataset` is responsible for initializing an interval dataset with series and category keys, and it sets two fields `seriesKeys` and `categoryKeys` to `null` when given an empty data array, which, however, will cause a `NullPointerException` later when the fields are used as receivers to invoke their member methods. Method `setCategoryKeys` sets the category keys for a `DefaultIntervalCategoryDataset` object, and during the process, it compares the length of its argument `categoryKeys` to that of `this.startData[0]` without making sure that `this.startData` has at least one element, which will cause an unexpected exception when `this.startData` is `null` or empty. Accordingly, an appropriate and complete patch for this defect must include changes to both methods. Changes to method `setCategoryKeys` are needed to correctly count the number of categories of `this` object, and adjustments to method `DefaultIntervalCategoryDataset` are necessary to make sure that the two fields `seriesKeys` and `categoryKeys` are always initialized with non-null arrays.

When applied to the fault, PReMM was able to successfully generate the same repair as crafted by the human developers in under 7 minutes. How did PReMM generate the repair? Three techniques distinguish PReMM from previous LLM-based approaches to program repair, namely faulty method clustering, fault context extraction, and dual-agent-based patch generation. PReMM first clustered the two faulty methods into two different invocation-wise clusters since there was no invocation relation between them (Figure 1b). This step enables PReMM to repair each faulty method independently, greatly reducing the complexity of, and increasing the chances of success for, the repairing task. Then, it constructed possible invocation chains from the failing test methods to the two faulty methods (Figure 1d), identified method `setSeriesKeys` of the same class as the code segment that is the most similar to the faulty method `setCategoryKeys` (Figure 1c), and gathered tokens like `Comparable` and `getCategoryCount` as the key ingredients for correcting the faulty methods (Figure 1e). Such rich contextual information about the fault provides extra guidance to the generation of high-quality patches. Finally, PReMM employed two agents, one for fault analysis and the other for

[2]Without loss of generality, we regard constructors as member methods of their enclosing classes in this work.

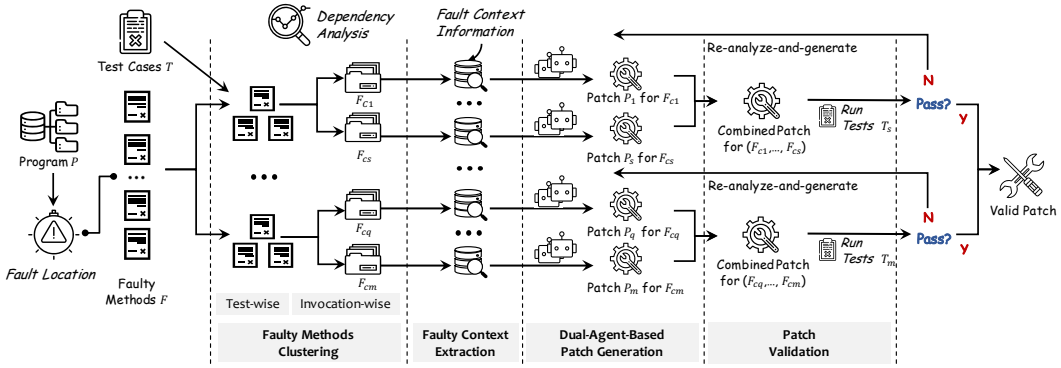


Fig. 2. Overview of PReMM.

patch generation, to analyze the fault and produce patches for the methods in a collaborative and iterative way, allowing the agents to improve their analysis and generation results based on the feedback received. After 4 and 4 iterations, PReMM was able to generate the valid sub-repairs for methods `DefaultIntervalCategoryDataset` and `setCategoryKeys`, respectively, which merged later into a final repair that not only passed all the tests but also was manually confirmed to be correct.

3 The PReMM Approach

We now explain in detail how PReMM effectively generates multi-method repairs to program faults step by step. Since fault localization is usually developed as an independent field, and most existing APR techniques employ off-the-shelf fault localization tools in their repair pipeline, we do not discuss the fault localization step below. Instead, we assume that the location information about the buggy lines is available already before applying PReMM to repair a buggy program. Figure 2 illustrates the overall process of automated program repair with PReMM. Roughly speaking, taking a buggy program P, the lines of code in the program that are considered buggy, and a group of passing and failing test cases T as the input, PReMM goes through a two-stage process to produce a repair that can make the program pass all the test cases. In the first stage, PReMM partitions the faulty methods F into mutually exclusive clusters based on whether the patches to the methods can interfere with each other's validation and whether there is any invocation relation among them (Section 3.1). In the meanwhile, PReMM also gathers rich contextual information about the bug in question (Section 3.2). In the second stage, PReMM employs two LLM-based agents to repeatedly generate and validate candidate patches for the faulty method clusters until a valid program patch is found or the stop criterion is met (Section 3.3). Effectively, PReMM implements a *divide-and-conquer* repairing strategy. It *divides* the more challenging task of program patch generation into a group of hopefully less demanding subtasks of cluster patch generation, and it *conquers* the subtasks and *combines* the subtasks' outcomes to form the program repair to be validated or reported.

In the rest of this section, we will use the following notations. Let P denote the buggy program to be repaired, $T_\checkmark$ and $T_\times$ represent the two sets of passing and failing test cases, respectively, and $T = T_\checkmark \cup T_\times$ denote the complete set of test cases. Let M represent all methods in P, and $F = \{f_1, \ldots, f_n\} (1 \leq n)$[3] denote the set of identified faulty methods, where $F \subseteq M$. PReMM aims to generate a repair R such that the repaired program $R(P)$ passes all tests in T, denoted as $R(P) \models T$.

[3]While PReMM can be applied to repair single-method bugs (i.e., $n = 1$), we designed it mainly for correcting multi-method bugs (i.e., when $n > 1$), and that is when PReMM's repairing power is expected to shine.

For a method $m \in M$, we use I_m to denote the set of methods (directly or indirectly) invoked by m. Since a test $t \in T$ is typically defined as a method, we also use I_t to denote the set of methods (directly or indirectly) invoked by t. Besides, we overload the notation and use $I_{M'}$ and $I_{T'}$ to denote the set of methods (directly or indirectly) invoked by a set M' of methods and a set T' of tests, respectively. In other words, we have $I_{M'} = \cup_{m \in M'} I_m$ and $I_{T'} = \cup_{t \in T'} I_t$. Additionally, for a method m, we use T_m to denote the set of tests in T that (directly or indirectly) invokes m and $T_{M'} = \cup_{m \in M'} T_m$ to denote the set of tests that (directly or indirectly) invoke any method from M'.

3.1 Faulty Method Clustering

Effectively proposing high-quality multi-method patches is highly challenging. On the one hand, because of the vast patch space to explore when repairing each faulty method and the combinatorial explosion of possible method patches, it is highly unlikely that a program repair tool can generate a program patch that simultaneously contains the appropriate changes to all the faulty methods. On the other hand, correcting a multi-method fault by separately repairing each faulty method will not work well either when those methods depend on each other, and therefore, their patches influence each other's generation and validation.

Since neither of these two extreme approaches is generally viable in practice, PReMM adopts a middle path based on the divide-and-conquer strategy. As explained earlier, PReMM partitions the set F of faulty methods into disjoint clusters, dedicates repair sub-tasks to generate patches for each cluster of faulty methods, and combines the generated cluster patches to form candidate program patches. Here, as all the faulty methods within the same cluster are repaired together, for this strategy to work well, it is critical to partition the faulty methods appropriately to strike a good balance between the benefits and the drawbacks brought about by the other faulty methods in the same cluster when generating cluster patches. That is, faulty methods within a cluster should provide valuable information for guiding the generation of patches to each other, and the amount of such information should not become overwhelming for the program repair tool to consume.

PReMM partitions the faulty methods F based on two types of dependence, namely *test-dependence* and *invocation-dependence*, among them. The rationale behind faulty method clustering based on the test-dependent relation is that, since changes to one faulty method may affect the test execution results produced in validating other faulty methods from the same test-wise cluster, patches to faulty methods in a test-wise cluster should be *validated* together. The rationale behind faulty method clustering based on the invocation-dependent relation is that, faulty methods with invocation dependence relation typically implement related functionalities and may share similar context, and therefore patches to such faulty methods should be *generated* together.

PReMM constructs the Class Hierarchy Analysis (CHA) call graph [14] for both faulty method clustering and invocation chain extraction for two reasons. First, CHA call graphs strike a good balance between precision and construction costs [14]. Second, PReMM only cares about the invocation relations among the failing tests and input faulty methods, which seldom come in large amounts, so we don't need to worry about producing extremely large clusters. CHA call graphs turned out to be sufficient to enable PReMM to produce high-quality repairs, and the average number of methods per cluster was below 2 in our experimental evaluation on Defects4J (Section 4).

3.1.1 Test-Wise Clustering. Test-wise clustering partitions faulty methods $F = \{f_1, \ldots, f_n\}$ into test-wise clusters based on their test-dependence relations, which capture both direct and indirect dependencies induced by shared test executions. Two faulty methods f_1 and f_2 ($f_1, f_2 \in F$) are *test-dependent* on each other, denoted as $f_1 \leftrightarrow_t f_2$, if and only if 1) they are both invoked by at least one test, i.e., $\exists t \in T : f_1 \in I_t \wedge f_2 \in I_t$ (e.g., Figure 3a shows an example of this case), or 2) there exists

another faulty method f_3 that is test-dependent on both f_1 and f_2, i.e., $\exists f_3 \in F : f_1 \leftrightarrow_t f_3 \wedge f_2 \leftrightarrow_t f_3$ (e.g., Figure 3b shows an example of this).

To compute the test-dependence relation $\leftrightarrow_t$ among faulty methods, PReMM performs the following steps: (1) Static Analysis: PReMM first statically analyzes the program P and the test suite T to construct a call graph G and determine, for each faulty method f, the set $T_f \subseteq T$ of tests that invoke f. (2) Construct Direct Test-Dependency Set U: We define and construct the set $U \subseteq F \times F$ of faulty method pairs with direct test dependency such that each pair (f_1, f_2) is in U if and only if f_1 and f_2 are invoked by at least one common test: $U = \{(f_1, f_2) : f_1 \in F \wedge f_2 \in F \wedge T_{f_1} \cap T_{f_2} \neq \emptyset\}$. (3) Transitive Closure: Finally, the test-dependence relation $\leftrightarrow_t$ between faulty methods is computed as the transitive closure of U: $\leftrightarrow_t = U \cup U \circ U \cup U \circ U \circ U \cup \ldots$. This ensures that indirect dependencies (e.g., $f_1 \leftrightarrow_t f_3 \wedge f_3 \leftrightarrow f_2 \implies f_1 \leftrightarrow f_2$) are included. PReMM can easily partition the faulty methods F based on the test-dependence relation so that any two faulty methods fall into the same cluster if and only if they are test-dependent on each other, and we call each resultant cluster from this partition operation a *test-wise cluster*.

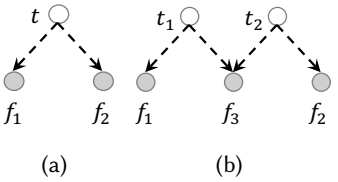


Fig. 3. Test-Dependency Relation.

3.1.2 Invocation-Wise Clustering. Invocation-wise Clustering partitions faulty methods into invocation-wise cluster based on their invocation-dependency relations. Two faulty methods f_1 and f_2 ($f_1, f_2 \in F$) are *invocation-dependent* on each other, denoted as $f_1 \leftrightarrow_i f_2$, if and only if 1) f_1 (directly or indirectly) invokes f_2, i.e., $f_2 \in I_{f_1}$, as shown in Figure 4a, 2) f_2 (directly or indirectly) invokes f_1, i.e., $f_1 \in I_{f_2}$, as shown in Figure 4b, or 3) there exists another faulty method f_3 that is invocation-dependent on both f_1 and f_2, i.e., $\exists f_3 \in F : f_1 \leftrightarrow_i f_3 \wedge f_2 \leftrightarrow_i f_3$, as shown in Figure 4c. To compute the invocation-dependence relation $\leftrightarrow_i$ among faulty methods, PReMM calculates the set $V \subseteq F \times F$ of faulty method pairs such that each pair (f_1, f_2) is in V if and only if f_1 directly or indirectly invokes f_2 or vice versa, i.e., $V = \{(f_1, f_2) : f_1 \in F \wedge f_2 \in F \wedge (f_1 \in I_{f_2} \vee f_2 \in I_{f_1})\}$, and the invocation-dependent relation $\leftrightarrow_i$ between faulty methods is computed as the transitive closure of V, i.e., $\leftrightarrow_i = V \cup V \circ V \cup V \circ V \circ V \cup \ldots$. Based on the invocation-dependence relation, PReMM can easily partition each *test-wise cluster* so that any two faulty methods remain in the same cluster if and only if they are invocation-dependent on each other, and we call each resultant cluster from this partition operation an invocation-wise cluster. In other words, two faulty methods f_1 and f_2 belong to one invocation-wise cluster if and only if they are both test-dependent and invocation-dependent on each other, i.e., $f_1 \leftrightarrow_t f_2 \wedge f_1 \leftrightarrow_i f_2$.

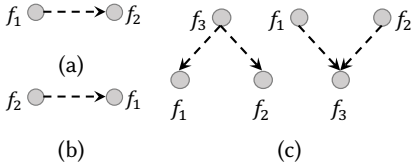


Fig. 4. Invocation-Dependence Relation.

To summarize, in faulty method clustering, PReMM first calculates the test-dependence relation $\leftrightarrow_t$ and invocation-dependence relation $\leftrightarrow_i$ among faulty methods, and then partitions the set F of faulty methods into a collection $C = \{F_{c_1}, \ldots, F_{c_m}\}$ ($1 \leq m \leq |F|$) of disjoint non-empty invocation-wise clusters satisfying that 1) $C \subset 2^F \wedge \emptyset \notin C$, 2) $\forall F_{c_1}, F_{c_2} \in C : F_{c_1} \cap F_{c_2} = \emptyset \wedge \cup_{1 \leq x \leq m} F_{c_x} = F$, and 3) $\forall F_{c_i} \in C, f_{c_1}, f_{c_2} \in F_{c_i} : f_{c_1} \leftrightarrow_t f_{c_2} \wedge f_{c_1} \leftrightarrow_i f_{c_2}$.

3.1.3 Remark. Faulty method clustering (FMC) provides multiple benefits. First, each identified invocation-wise cluster groups together the faulty methods that should be generated collectively, providing PReMM's patch generation with clearer focuses and richer contextual information (see Section 3.2). Second, each identified test-wise cluster groups together the faulty methods that

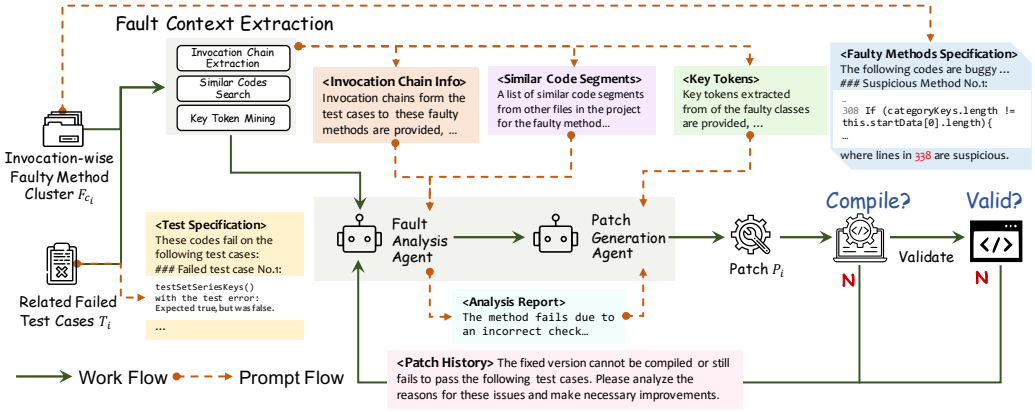


Fig. 5. How Fault Contexts Facilitate Dual-Agent-Based Patch Generation.

should be validated collectively, helping PReMM be more efficient in discovering invalid patches and more confident in the quality of the patches that validate successfully (see Section 3.3.2). Third, by allowing patch generation and validation to be performed on smaller clusters of faulty methods, faulty method clustering enables PReMM to divide the task of multi-method patch generation into several sub-tasks of similar nature but most likely on smaller scales, enhancing PReMM's repairing power and scalability. Note that faulty method clustering is always applicable. Having all the faulty methods in one cluster simply means that, according to our analysis, those faulty methods are test- or invocation-dependent on each other and should be repaired together.

3.2 Fault Context Extraction

While the generation of cluster patches is hopefully less formidable than that of program patches, it is still more demanding than the generation of single-method patches if the cluster contains more than one faulty method. Existing techniques, like ChatRepair [55] and ThinkRepair [59], rely heavily on the basic fault information, including the code snippets around the faulty code and the error messages produced by the failing tests. Richer contextual information, however, is required for repairing multi-method bugs. To better guide the remaining repairing process in understanding and repairing the fault in question, PReMM extracts the following three additional types of information about the failure caused by the fault and the fault context: 1) *Invocation chain information* that specifies how the faulty methods are invoked by the failing tests, offering extra knowledge about the invocation contexts of the faulty methods; 2) *Similar code segments* that specify code snippets from the same program that are similar to the code at the fault locations, demonstrating how similar operations were correctly performed in the same application; 3) *Key tokens* that contain relevant identifiers from the repair context, highlighting ingredients that may be crucial in constructing the desired patch. We explain in this section how the remaining three types of fault context information are extracted. The subsequent section (Section 3.3) will describe how the information is utilized to facilitate fault analysis and patch generation, as illustrated in Figure 5.

3.2.1 Invocation Chain Information. Given a invocation-wise cluster $F_c = \{f_{c_1}, \ldots, f_{c_k}\}(1 \leq k)$ of faulty methods, $T_c = \{t \in T : I_t \cap F_c \neq \emptyset\}$ is the set of tests that invoke at least one method from F_c. Let $G = \langle V, E\rangle$ be the call graph constructed for program P and tests T_c, where $V = M \cup T_c$ $E \subseteq G \times G$, and each pair of methods $\langle v_1, v_2\rangle$ is in E if and only if v_1 invokes v_2 $(v_1, v_2 \in V)$. A sequence $\rho = [v_1, \ldots, v_l]$ of methods denotes an acyclic path in G if and only if $v_i \in V(1 \leq i \leq l)$,

$\forall i : 1 \leq i < l \implies \langle v_i, v_{i+1} \rangle \in E$, and $\forall 1 \leq i, j \leq l : i \neq j \implies v_i \neq v_j$. Let $S = \{\rho_1, \ldots, \rho_r\}$ be the set of acyclic paths in G, PReMM picks for each faulty method $f \in F_c$ a path ρ_0 in S that satisfies the following conditions as the representative to encode f's invocation chain information: 1) it starts from a test method $t \in T_c$; and 2) it ends with the faulty method f. The invocation chain information captures how failing test cases T_c invoke the faulty methods F_c. Finally, to eliminate redundancy, if S contains overlapping paths—e.g., for faulty methods f_{c_s} and f_{c_t} where f_{c_s} invokes f_{c_t}, both paths $\rho_s = \{t \rightarrow \cdots \rightarrow f_{c_s}\}$ and $\rho_t = \{t \rightarrow \cdots \rightarrow f_{c_s} \rightarrow \cdots \rightarrow f_{c_t}\}$ may appear—we retain only the longer path and discard the shorter one. While the representative path ρ_0 (derived through static analysis) may not perfectly mirror every runtime execution, this design avoids the computational overhead of dynamic tracing and preserves the key structural patterns while omitting noisy runtime branches and ensuring LLM-actionable inputs.

3.2.2 Similar Code Segments. Given the buggy program P and a cluster $F_c = \{f_{c_1}, \ldots, f_{c_k}\} (1 \leq k)$ of faulty methods, PReMM employs an inheritance-hierarchy-aware code search approach from TransPlantFix [56] to identify code segments similar to the faulty methods. Here, the code similarity between two methods is calculated based on the local and global features of the two methods, with the local features reflecting their structural and syntactical similarities, and the global features assessing the inheritance relationship of the methods' containing classes.

In particular, given two methods m_1 and m_2, the local similarity between them is calculated based on their local features as follows:

$$Sim_l(m_1, m_2) = \lambda_1 * \text{jacc}(m_1.name, m_2.name) + \lambda_2 * \text{jacc}(m_1.code, m_2.code) \tag{1}$$

As shown in Equation 1, the local similarity is essentially calculated as the weighted sum of the Jaccard similarity of the two methods' names and their body. Note that, in the original TransPlantFix approach, the local similarity incorporates the similarity between the methods' names and argument type lists, and the similarity between two identifiers in CamelCase is calculated by first splitting them into sets of subtokens and then comparing the two sets. Since we are more interested in finding out the similarity between the two methods' implementations, PReMM extends the design by factoring in the similarity between the two method bodies. It splits each method body into a set of lexical tokens and then computes the Jaccard similarity between the two sets. We opted for Jaccard similarity, instead of edit distance, to keep similar code segment identification in PReMM lightweight. The parameters λ_1 and λ_2 represent the weights assigned to the similarity score of the method name and method code, respectively.

The global similarity between methods m_1 and m_2 is calculated as follows:

$$Sim_g(m_1, m_2) = \begin{cases} 1 & \text{if } m_1.cls \simeq m_2.cls, \\ \lambda_3 \cdot \text{jacc}(m_1.pkgName, m_2.pkgName) + & \\ \quad \lambda_4 \cdot \text{jacc}(m_1.clsName, m_2.clsName) & \text{otherwise} \end{cases} \tag{2}$$

As shown in Equation 2, the global similarity between two methods m_1 and m_2 captures both class hierarchy and naming relationships. Specifically, the similarity takes the value 1 if their containing classes inherit from the same superclass, denoted as $m_1.cls \simeq m_2.cls$; otherwise, it is the weighted sum of the similarities between their package names and class names. The parameters λ_3 and λ_4 represent the weights assigned to the similarity score of their package and class names, respectively.

Following the practice in TransPlantFix, PReMM computes the overall similarity score between two methods as the average of their local and global similarities. All the weights in the above two equations, namely λ_1, λ_2, λ_3, and λ_4), are set to 0.5 by default so that the Jaccard similarities contribute equally to the local and global similarity scores. Finally, for each faulty method f, we select 5 correct methods from the buggy program with the highest similarity to f and use them as similar code segments for f. These similar code segments serve as repair references by

demonstrating how analogous code is implemented elsewhere in the program, helping to guide the generation of more context-aware and consistent fixes.

3.2.3 Key Tokens. Given a cluster $F_c = \{f_{c_1}, \ldots, f_{c_k}\}(1 \leq k)$ of faulty methods, PReMM extracts key tokens from the methods' containing classes to provide tailored cues to guide repair generation for the methods. In particular, the set K of key tokens is the union of the following 4 types of tokens: 1) the set V of variable names from the method, 2) the set M of method names from the method's containing class, 3) the set PT of parameter types, and 4) the set PN of parameter names. These tokens provide crucial contextual cues for the LLM during the repair process. The effective use of these tokens helps the LLM focus on semantically relevant information, thereby avoiding the generation of irrelevant or extraneous tokens—an issue frequently discussed in previous work on LLM-based program repair, such as Repilot [51]. The inclusion of method-specific tokens ensures that the LLM has sufficient context to generate accurate repairs, as these tokens reflect the domain semantics (e.g., intent hints from method/variable names) of the code.

3.3 LLM-Based Program Repair

As shown in Figure 2, after partitioning the faulty methods F into disjoint clusters $C = \{F_{c_1}, \ldots, F_{c_m}\}$, where each cluster $F_c = \{f_{c_1}, \ldots, f_{c_k}\}(1 \leq k)$ groups methods with shared invocation contexts, and gathering additional contextual information about the fault for each cluster F_c, PReMM, attempts to repair these invocation-wise clusters in a parallel manner. The process operates as follows: (1) *Dual-Agent-Based Patch Generation*: For each cluster F_c, two specialized LLM-based agents collaborate to generate a compilable patch $\mathcal{P}(F_c)$. And (2) *Test-Wise Patch Validation*: Generated patches $\{\mathcal{P}(F_{c_1}), \ldots, \mathcal{P}(F_{c_m})\}$ are regrouped into test-wise patches based on their test-dependency relations. Each consolidated patch is validated against its associated tests. If validation fails, detailed error traces (e.g., test assertion failures) are fed back to the corresponding dual-agent repairers. This loop continues until either: A fully valid patch is produced, or A predefined interaction limit (e.g., time, iterations) is reached. This design is particularly effective because the hierarchical validation ensures that each invocation-wise cluster patch is independently compilable during the dual-agent repair phase, preventing systemic rework during test-wise validation.

3.3.1 Dual-Agent-Based Patch Generation. For each invocation-wise cluster $F_c = \{f_{c_1}, \ldots, f_{c_k}\}(1 \leq k)$, PReMM employs two agents, namely *fault analysis agent* and *patch generation agent*, which utilize different types of contextual information extracted through the *fault context extraction* component (Section 3.2), to analyze the fault and produce compilable patches for the faulty methods in F_c in a collaborative way. As illustrated in Figure 6, this patch generation process essentially involves three tasks: 1) fault analysis, 2) patch generation, and 3) patch compilation. The resulting compilable patch is subsequently forwarded to the validation phase (Section 3.3.2).

Task 1: Fault Analysis. PReMM employs an LLM-based fault analysis agent to accomplish two crucial tasks: (1) analyze the root cause and context of the fault; and (2) refine its diagnosis by iteratively incorporating repair history (e.g., past failed patches) and validation feedback (e.g., test assertion failures). It will produce a fault analysis report to guide the program repair agent so that the latter can be better oriented in the cluster patch generation process.

In particular, the fault analysis agent is assigned at the very beginning the role of "`fault analysis expert`" and the task of "`analyzing a functional bug across multiple methods`" via a system prompt. The user prompt for the fault analysis agent in each interaction is divided into three parts, as shown in Figure 5. The first part is dedicated to describing the fault in the invocation-wise cluster of faulty methods. More concretely, this part contains i) the signature, the body, and the faulty lines of each faulty method, and ii) the signature and body of each associated failing test case, both of which can be easily derived from PReMM's input. The second part is

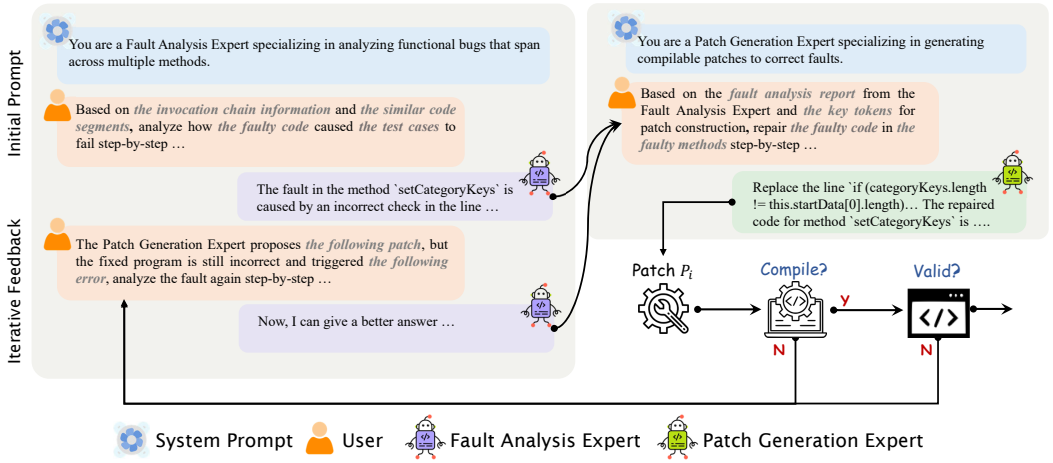


Fig. 6. The Interaction Between the Fault Analysis Agent and the Patch Generation Agent.

dedicated to presenting the information extracted from the fault context, as described in Section 3.2, including the *invocation chain information* and *similar code segments*. Considering that PReMM may need to go through multiple iterations of fault analysis and patch generation before a desirable patch is produced, and that the information about the unsuccessful attempts should be fed back to the agent, we reserve the third (optional) part of the fault analysis agent's user prompt for information about the undesirable patches generated in the latest attempt, if any, and the feedback on why the patch was undesirable. Here, the feedback can be i) the compilation errors and their locations in each faulty method if the patch fails to pass the patch compilation (Task 3) or ii) the signature and body of each failed test case if the patch fails to validate (see Section 3.3.2). The user prompt also instructs the fault analysis agent to generate a detailed report on why the specified lines of the methods were faulty. The report is then passed to the patch generation agent to guide the generation of compilable cluster patches.

Task 2: Patch Generation. PReMM employs a second LLM-based *patch generation agent* to accomplish the task: generate the patch $\mathcal{P}(F_c)$ for all the faulty methods in cluster F_c. Just like the *fault analysis agent*, the *patch generation agent* is assigned the role of "`patch generation expert`" and the task of "`generating a compilable patch to correct the given fault`" via a system prompt. The user prompt for the patch generation agent in each interaction is divided into three parts, as shown in Figure 5. Here, the first part is the same as the first part of the user prompt for the *fault analysis agent*, which describes the fault of faulty methods in the cluster; the second part contains the analysis report produced by the *fault analysis agent*; the third part contains the *key tokens* gathered during *fault context extraction* (see Section 3.2). The user prompt instructs the *patch generation agent* to generate patches for the faulty methods in question and explain the reasons for repairing the faulty methods in the given way. While the explanations do not directly affect the compilability or validity of the patches, they are valuable information to help the users better understand the patches, should they validate successfully and get reported to the users.

Task 3: Patch Compilation. This step validates the compilability of the generated patch $\mathcal{P}(F_c)$ for invocation-wise cluster F_c. Since $\forall F_{c_i}, F_{c_j} \in C(i \neq j)$, $\mathcal{P}(F_{c_i})$ and $\mathcal{P}(F_{c_j})$ are independent, PReMM first verifies its *local compilability* by integrating $\mathcal{P}(F_c)$ into the buggy program P, to produce a candidate repair $P' = P \cup \mathcal{P}(F_c)$. If P' can be compiled, $\mathcal{P}(F_c)$ proceeds to test-wise validation (Section 3.3.2). Otherwise, as described earlier, the patch and the errors that compilation

errors will be utilized as the third part of a user prompt for the *fault analysis agent* to drive the next round of compilable cluster patch generation, effectively making the process iterative. Note that this process only ensures the compilability of $\mathcal{P}(F_c)$, it cannot make all the tests pass yet.

3.3.2 Test-Wise Patch Validation. For two invocation-wise clusters F_{c_i} and F_{c_j}, if they satisfy the following condition: $\forall f_i \in F_{c_i}, \forall f_j \in F_{c_j}$, $f_i \leftrightarrow_t f_j$ (f_i and f_j, are test-dependent), we call F_{c_i} and F_{c_j} are test-dependent. Once PReMM has generated a compilable cluster patch to each invocation-wise cluster of faulty methods, denoted as $\{\mathcal{P}(F_{c_1}), \ldots, \mathcal{P}(F_{c_m})\}$, it proceeds to check the validity of these patches, i.e., whether they can make the associated tests pass. This is conducted by combining the compilable patches for all the test-dependent invocation-wise clusters in each test-wise cluster $F_C = \{F_{c_1}, \ldots, F_{c_s}\}$ into a larger patch $\mathcal{P}(F_C)$, applying them to the buggy program, and running the repaired program against the associated test cases T_C of each F_C. If the repaired program fails some test cases in T_C, the patches of the current test-wise cluster are not valid, and PReMM starts another iteration of compilable cluster patch generation on each invocation-wise cluster $F_{c_i} \in F_C$ until a valid patch for F_C is produced or the maximum number of repetitions has been exhausted. Otherwise, we have found a potentially valid patch to the faulty methods in F_C.

Finally, once all the valid test-wise cluster patches become available, PReMM proceeds to synthesize a unified program-level patch by combining the corresponding partial patches. This final step is essential because test-wise validations performed in isolation cannot ensure that the combined patch will preserve correctness. Since *faulty method clustering* is based on the call graph of the original buggy program, the invocation relations among these target methods may be different in the repaired program, potentially introducing unforeseen interactions between patched components. Therefore, to address this, PReMM applies all validated partial patches to the buggy program, forming a complete program-level patch, and performs a final validation by executing the full test suite. This step adheres to standard practice in automated program repair. If all test cases pass, the patch is deemed *valid* and reported to the user. Otherwise, if any test case fails, the combined patch is rejected and the repair attempt is considered unsuccessful.

3.4 Implementation

We have implemented the PReMM approach into a standalone tool with the same name, using Qwen2.5-72B-Instruct [49] (released in September 2024) as the base model, which ranks first on the OpenCompass Rank [41] (as of September 2024). Qwen2.5 is an open-source large language model series developed by Alibaba Cloud, and Qwen2.5-72B-Instruct is the instruction-tuned version of Qwen2.5 with 72B parameters. We utilize the model through the Alibaba Cloud API and set the temperature to 1.0 to balance creativity and accuracy, allowing the model to generate diverse yet meaningful patches, following ThinkRepair [59] and ChatRepair [55]. By default, PReMM is configured to generate at most five candidate patches in each repairing session targeting a specific bug, and it terminates once a valid patch is found, the maximum number of candidates have been generated and validated, or the allocated time is used up.

While PReMM is built on top of the Qwen2.5-72B-Instruct model in its current implementation, it is worth noting that PReMM is only loosely coupled with the underlying large language model, and we can easily replace the model with other more powerful large language models when they become available to take advantage of the latest advancements in the area.

4 Experimental Evaluation

In this section, we experimentally apply PReMM to repair bugs in real-world programs and assess its effectiveness. Our experiments are designed to address the following three research questions:

RQ1: How effective is PReMM in repairing real-world bugs? In RQ1, we assess to what extent PReMM is able to generate high-quality repairs to real-world bugs, especially bugs that demand composite repairs.

RQ2: How does PReMM compare to existing program repair tools in terms of effectiveness? In RQ2, we compare the repairing results produced by PReMM and learning- or LLM-based program repair tools that represent the state of the art. Following the previous work [55, 59], we mainly focus on the number of correct patches each technique can produce in the comparison.

RQ3: How important are its core components in helping PReMM achieve the program repair results? The PReMM technique is built on three core components, namely faulty method clustering, fault context extraction, and dual-agent-based patch generation. In RQ3, we zoom in on the core components of PReMM and investigate how important they are in helping the tool repair buggy programs.

4.1 Datasets

Following the practice in existing research [53–55, 59], we apply PReMM to repair the real-world bugs from the Defects4J datasets [18] in our experiments. Defects4J V1.2 contains 395 bugs across 6 Java projects, while Defects4J V2.0 includes 444 *different* bugs across 12 Java projects. In total, the two versions provide 839 unique bugs, of which 545 involve a single buggy method, 186 involve multiple buggy methods, and 108 involve buggy code elements outside methods (e.g., class fields). PReMM is designed to address both single-method (SM) and multi-method (MM) bugs.

Table 1 provides detailed statistics of the projects and bugs from the two datasets. In particular, for each project in the dataset, the table lists: 1) the total number of lines of code in the project (LOC); 2) the total number of bugs in the project[4] (#Bug); and 3) the number of single-method bugs (#SMB). Regarding the multi-method bugs in each project (MMB), the table also lists: 5) the total number of multi-method bugs (Tot); 6) the number of multi-method bugs where the faulty methods were/were not partitioned into different clusters (#C/#NC); 7) the total number of faulty methods involved (#FM); and 8) the total numbers of test-wise (#TC) and invocation-wise (#IC) clusters into which the faulty methods were partitioned.

According to the statistics, a significant percentage, 22.2% (=186/839) to be precise, of the Defects4J bugs demand multi-method patches and are beyond the repairing power of most existing LLM-based program repair techniques [55, 59], which highlights the necessity and importance of techniques like PReMM that aim to repair multi-method bugs. More concretely, for 97, or 52.2% (=97/186), multi-method bugs, the involved faulty methods are partitioned into different clusters, which suggests that divide-and-conquer in general and faulty method clustering in particular are effective strategies for reducing the complexity of multi-method bug repairing in many cases; for 89, or 47.8% (=89/186), multi-method bugs, the involved faulty methods are not partitioned into different clusters. This indicates that these methods are invocation-dependent on each other, meaning successful program repair must produce coordinated patches for all of them as a whole, which highlights the necessity of sophisticated techniques like fault context extraction and dual-agent-based patch generation to guide the LLMs in diagnosing the faults and generating the patches.

4.2 Baselines

To address RQ2, we compare PReMM with 8 state-of-the-art automated program repair techniques. In particular, the baselines include 4 neural-machine-translation-based (NMT-based) approaches, i.e., RewardRepair [58], TENURE [40], KNOD [16], and Mulpo [29], and 4 LLM-based APR methods, i.e.,

[4]The numbers of single-method and multi-method bugs do not add up to the total number of bugs in some projects because certain bugs only require changes outside methods, e.g., concerning class field declarations.

Table 1. The Defects4J Datasets of Real-World Bugs Used for Evaluating PReMM and the Evaluation Results.

Project	LOC	#Bug	#SMB	MMB						Valid			Correct		
				Tot	#C	#NC	#FM	#TC	#IC	Tot	#SM	#MM	Tot	#SM	#MM
Chart	230K	26	17	9	7	2	23	18	20	22	15	7	22	15	7
Closure	149K	133	100	30	8	22	65	31	38	43	41	2	35	33	2
Lang	61K	65	46	15	11	4	36	23	29	45	36	9	36	29	7
Math	186K	106	74	29	22	7	66	39	54	55	46	9	41	32	9
Mockito	22K	38	24	9	5	4	41	19	34	10	10	0	9	9	0
Time	68K	27	18	8	3	5	32	15	21	9	8	1	4	3	1
Defects4J V1.2	716K	395	279	100	56	44	263	145	196	184	156	28	147	121	26
Cli	4K	39	30	7	4	3	19	7	13	23	20	3	17	16	1
ClosureV2.0	149K	43	13	12	6	6	25	12	18	5	5	0	2	2	0
Codec	5K	18	12	2	1	1	5	2	4	10	10	0	8	8	0
Collections	64K	4	1	1	0	1	3	1	1	0	0	0	0	0	0
Compress	12K	47	36	5	5	0	15	6	11	27	25	2	22	20	2
Csv	1K	16	12	1	1	0	2	1	2	11	11	0	10	10	0
Gson	11K	18	12	3	3	0	14	5	6	7	7	0	7	7	0
JacksonCore	27K	26	13	11	6	5	41	14	22	14	10	4	13	10	3
JacksonDatabind	75K	112	61	22	4	18	59	23	31	49	37	12	44	32	12
JacksonXml	7K	6	4	0	0	0	0	0	0	3	3	0	1	1	0
Jsoup	17K	93	63	13	6	7	35	22	22	41	40	1	35	34	1
JxPath	28K	22	9	9	5	4	20	12	14	1	1	0	1	1	0
Defects4J V2.0	400K	444	266	86	41	45	238	105	144	191	169	22	160	141	19
Overall	1116K	839	545	186	97	89	501	250	340	375	325	50	307	262	45

Table 2. The Baseline APR Techniques and PReMM.

Approach	RewardRepair	TENURE	KNOD	Mulpo	AlphaRepair	GAMMA	ChatRepair	ThinkRepair	PReMM
Category	NMT-based	NMT-based	NMT-based	NMT-based	LLM-based	LLM-based	LLM-based	LLM-based	LLM-based
Repair-Granularity	Token	IR	AST	Statement/Expression/Token	Token (Cloze)		Line/Hunk/Method	Method	Method
Stop Criterion	200	5 Hours	5 Hours	3 Hours	5000	5 Hours	$\leq$ 1500	$\leq$ 125	$\leq$ 15
Venue	ICSE2023	ICSE2023	ICSE2023	ISSTA2024	FSE2022	ASE2023	ISSTA2024	ISSTA2024	–
Reference	[58]	[40]	[16]	[29]	[54]	[60]	[55]	[59]	–

AlphaRepair [54], Gamma [60], ChatRepair [55], and ThinkRepair [59]. One noteworthy difference between the two categories of program repair techniques is that, while NMT-based APR approaches require fine-tuning the underlying models on historical bug-fixing data to achieve good performance, LLM-based approaches often do not require such fine-tuning. More concretely, AlphaRepair and Gamma treat automated program repair as a cloze task and utilize pre-trained LLMs to complete the task, while ChatRepair and ThinkRepair repair bugs through conversational interactions with ChatGPT via the OpenAI API [5]. All these 4 approaches eliminate the need for fine-tuning the underlying models on bug-fixing data. Similarly, PReMM repairs bugs through conversational interactions with the Qwen2.5-72B-Instruct model via the Alibaba Cloud API, without the need for fine-tuning as well. Table 2 summarizes the basic information about the 8 baselines. For each baseline, the table lists its category, repair granularity, the stopping criterion adopted, the reference to the corresponding paper, and the venue at which the paper was published. We systematically review these baselines and the other representative program repair techniques in Section 5.

4.3 Experimental Setup

Each bug b from the datasets provides a buggy program P_b and a set T_b of tests such that P_b fails on at least one of the tests in T_b—thus triggering b. Following the standard practice in evaluations of NMT-based and LLM-based automated program repair tool [53–55, 59], we assume the availability of *perfect information* about the faults' locations in the experiments. In particular, given a bug b, we assume the perfect location information L_b about b is known before applying PReMM or a variant to repair the bug. Here, L_b contains not only the collection M_b of faulty methods associated with b but also the exact fault locations in those methods. In the repair session targeting a specific bug b, PReMM or one of its variants takes P_b, T_b, and L_b as the input and, when successful, eventually produces a *valid* patch v_b to b, i.e., v_b can make P_b pass all the tests in T_b.

We follow the practice of previous evaluations of APR tools [54, 61] and allocate by default an end-to-end timeout of 5 hours for fixing each bug. However, the average time required to repair Defects4J bugs is at the minute level in our experiments. This is primarily because PReMM only samples a small number of (maximum $3 * 5 = 15$, to be precise) candidate repairs for each bug.

All the experiments were performed on a desktop equipped with an Intel® Core™ i7-10700 CPU, 32GB RAM, running Ubuntu 22.04. To account for the randomness involved in the repairing process, we run the experiment of applying PReMM or a variant to repair each bug 3 times and consider the bug being repaired with a *valid* patch if that is the case in at least one of the runs.

Repair Results of Baseline Techniques. We refrained from repeating the experiments of the 8 baseline techniques on Defects4J bugs and reused the program repair results reported in the publications associated with the techniques in addressing RQ2. This is partly because the reported results should represent the typical effectiveness of the baseline techniques in repairing Defects4J bugs, and partly because no easy-to-use replication package is available for download for some of those techniques, e.g., while Gamma [60] provides patch generation scripts, it lacks automated validation components and produces an excessive average of 475.8 patches per bug in Defects4J V1.2, making comprehensive validation prohibitively resource-intensive. These technical constraints make reproducing their results highly challenging or even infeasible, consistent with established APR practices [55, 59] of comparing published results under identical benchmark conditions.

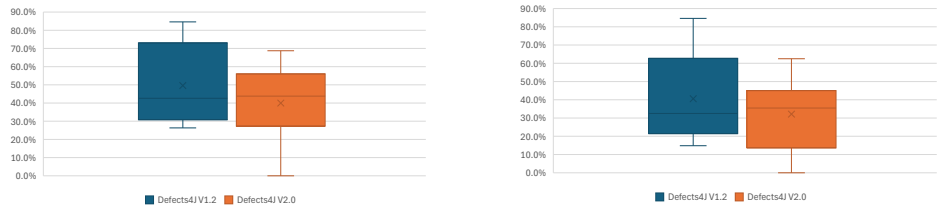
4.4 Metrics

As customary in the APR literature [16, 17, 26–28, 36, 40, 53, 54, 57, 58], we determined the *correct* patch to a bug in Defects4J by manually checking the *valid* patch v_b and comparing it to the manually-written patch for the fault under repair: v_b is *correct* if it is *semantically equivalent* to the patch manually written by the developers and included in the benchmark. Conservatively, v_b is marked incorrect if we cannot conclusively establish it as equivalent in a moderate amount of time.

For each bug, we record whether a *valid* patch is produced and whether the *valid* patch produced is *correct*. For each program repair tool x considered in the experiments and a specific collection y of bugs, we also calculate the following measurements:

- the recall of *valid* patches, or *v-recall*, achieved by x on bugs from y, calculated as the percentage of y bugs for which x produced a valid patch;
- the recall of *correct* patches, or *c-recall*, achieved by x on bugs from y, calculated as the percentage of y bugs for which x produced a correct patch;
- the *precision* achieved by x on bugs from y, calculated as the percentage of y bugs with valid patches whose patch is actually correct.

We do not record the time that PReMM spent on repairing each bug for three reasons. First, both PReMM and the LLM-based baseline techniques need to repeatedly query the underlying,



(a) v-recall (b) c-recall

Fig. 7. The Distribution of the V-Recall and C-Recall PReMM Achieved on Datasets Defects4J V1.2 and V2.0.

remote large language models for candidate patches or their components via APIs, making their repair time greatly affected by the LLMs' responsiveness. Second, when applied to a bug within their capability, both PReMM and the baseline techniques tend to produce high-quality candidate patches after a small number of attempts, leading to repair times that are magnitudes shorter than those required by earlier techniques (e.g., based on repair patterns and/or heuristics) and easily acceptable for non-interactive usage. Third, repair time was not considered an essential metric in recent work on LLM-based program repair.

4.5 Experimental Results

We report the experimental results achieved and answer the research questions in this section.

4.5.1 RQ1: Effectiveness. Table 1 also lists the program repair results PReMM has achieved. For each project in the Defects4J datasets, the table lists the total number of bugs that PReMM managed to repair with a valid patch (Valid/Tot), the breakdown of that to the numbers of single- and multi-method bugs that PReMM produced valid patches for (#SM and #MM), and the counterpart data for correct patches (Correct).

Considering all the 839 bugs in the datasets, PReMM generated valid patches for 375 of them, achieving a v-recall of 44.7% (=375/839), and correct patches for 307 of them, achieving a c-recall of 36.6% (=307/839), showing that PReMM is highly effective in proposing valid patches. The overall precision PReMM achieved is quite high, 81.9% (=307/375) to be precise. Such a high precision strongly suggests that PReMM is highly effective in guiding the program repair process to navigate the candidate patch space and discover the patches that are actually correct.

Single- vs. Multi-method Bugs. In general, PReMM is more effective in repairing single-method bugs than multi-method bugs, which is as expected. More concretely, if we consider all the 545 single-method bugs in the datasets, PReMM generated valid patches for 325 of them, achieving a v-recall of 59.6% (=325/545), and correct patches for 262 of them, achieving a c-recall of 48.1% (=262/545) and a precision of 80.6% (=262/325). In the meanwhile, if we consider all the 186 multi-method bugs in the datasets, PReMM generated valid patches for 50 of them, achieving a v-recall of 26.9% (=50/186), and correct patches for 45 of them, achieving a c-recall of 24.2% (=45/186) and a precision of 90.0% (=45/50).

We make two observations from the detailed comparison. On the one hand, multi-method bugs were beyond the repairing power of existing LLM-based program repair techniques, but PReMM managed to produce valid and correct patches for around 25% of them, which is an extraordinary achievement and highlights PReMM's effectiveness in achieving the goal we set for it. On the other hand, PReMM produced valid and correct patches for around 50% of the single-method bugs in the datasets, which underlines the remarkable effectiveness of *fault context extraction* and

Table 3. Performance of PReMM vs. Baseline Tools on Defects4J (Left to Right: Worst to Best).

Project	#Bug			RewardRepair			AlphaRepair			KNOD			GAMMA			TENURE			Mulpo			ChatRepair			ThinkRepair			PReMM		
	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M	T	S	M
Chart	26	17	9	5	5	0	9	8	1	10	9	1	11	9	2	7	6	1	13	10	3	15	15	0	11	11	0	22	15	7
Closure	133	100	30	15	15	0	23	23	0	23	23	0	24	23	1	26	24	2	26	23	2	37	37	0	31	31	0	35	33	2
Lang	65	46	15	7	7	0	13	12	1	11	11	0	16	11	5	16	13	3	19	15	4	21	21	0	19	19	0	36	29	7
Math	106	74	29	19	18	0	21	19	2	20	18	2	25	19	6	22	16	6	23	17	6	32	32	0	27	27	0	41	32	9
Mockito	38	24	9	3	2	0	5	4	0	5	5	0	3	2	0	4	3	0	7	5	1	6	6	0	6	6	0	9	9	0
Time	27	18	8	1	1	0	3	3	0	2	2	0	3	1	2	4	3	1	4	2	2	3	3	0	4	4	0	4	3	1
Defects4J V1.2	395	279	100	50	48	0	74	68	5	71	68	3	82	64	17	79	64	14	92	72	19	114	114	0	98	98	0	147	121	26
Cli	39	30	7	8	8	0	5	5	0	6	6	0	9	9	0	4	4	0	7	7	0	5	5	0	9	9	0	17	16	1
ClosureV2.0	43	13	12	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3	0	2	2	0
Codec	18	12	2	6	4	2	6	5	0	6	5	1	3	2	1	6	5	1	3	3	0	8	8	0	10	10	0	8	8	0
Collections	4	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Compress	47	36	5	4	4	0	1	1	0	3	3	0	4	4	0	8	6	2	10	10	0	2	2	0	16	16	0	22	20	2
Csv	16	12	1	3	3	0	1	1	0	4	4	0	0	0	0	3	3	0	3	3	0	3	3	0	8	8	0	10	10	0
Gson	18	12	3	1	1	0	2	2	0	2	2	0	3	3	0	2	2	0	1	1	0	3	3	0	5	5	0	7	7	0
JacksonCore	26	13	11	2	2	0	3	3	0	5	5	0	3	2	1	3	3	0	5	3	2	3	3	0	7	7	0	13	10	3
JacksonDatabind	112	61	22	11	10	1	8	8	0	11	10	1	10	9	1	10	7	3	13	10	3	9	9	0	17	17	0	44	32	12
JacksonXml	6	4	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	2	2	0	1	1	0
Jsoup	93	63	13	10	9	1	9	9	0	8	8	0	11	10	1	12	12	0	14	13	1	14	14	0	28	28	0	35	34	1
JxPath	22	9	9	4	2	2	1	1	0	3	2	1	2	1	1	0	0	0	4	3	1	0	0	0	2	2	0	1	1	0
Defects4J V2.0	444	266	86	50	44	6	36	35	0	50	47	3	45	40	5	50	44	6	60	53	7	48	48	0	107	107	0	160	141	19
Overall	839	545	186	100	92	6	110	103	5	121	115	6	127	104	22	129	108	20	152	125	26	162	162	0	205	205	0	307	262	45

dual-agent-based program repair techniques in helping PReMM diagnose and repair bugs requiring relatively smaller patches.

Generalizability across Datasets. PReMM proposed valid and correct patches to 184 and 147 bugs, respectively, from Defects4J V1.2. The corresponding v-recall, c-recall, and precision achieved are 46.6% (=184/395), 37.2% (=147/395), and 79.9% (=147/184), respectively. In contrast to that, PReMM proposed valid and correct patches to 191 and 160 bugs, respectively, from Defects4J V2.0. The corresponding v-recall, c-recall, and precision achieved are 43.0% (=191/444), 36.0% (=160/444), and 83.8% (=160/191), respectively. The box plots in Figure 7 show the distributions of the v-recall and c-recall achieved on Defects4J V1.2 and V2.0 projects. It is clear that, while the v-recall and c-recall achieved on Defects4J V2.0 are slightly lower than those achieved on Defects4J V1.2, the differences are only marginal, which shows that the results PReMM achieved on Defects4J V1.2 are generalizable to Defects4J V2.0.

> *Answer to RQ1: PReMM was effective in proposing valid patches for both single- and multi-method bugs across different datasets, and most of the valid patches it proposed were indeed correct.*

4.5.2 RQ2: Comparison. Table 3 reports the number of correct patches that different baseline program repair techniques produced on the datasets. For each baseline program repair technique and each project, the table lists the number of all bugs (T), single-method bugs (S), and multi-method bugs (M), for which the technique was able to produce a correct patch. For easy reference, we reproduce in the table the number of all bugs, single-method bugs, and multi-method bugs in each project, as well as the corresponding results produced by PReMM. For each project and the (sub)totals, the table also highlights the largest (positive) numbers of correct patches any technique has produced for all bugs, single-method bugs, and multi-method bugs, respectively.

Overall, PReMM substantially outperformed all the baseline program repair techniques in producing correct patches for the subject bugs. In particular, it generated correct patches for 307 bugs and

(a) Defects4J V1.2. (b) Defects4J V2.0.

Fig. 8. The Numbers of Defects4J Bugs Correctly Repaired by PReMM and the Baseline Techniques.

achieved a c-recall of 36.6%, which improves the same metric achieved by the baseline techniques by 49.8% (w.r.t. ThinkRepair) to 207.0% (w.r.t. RewardRepair). The huge improvement clearly highlights PReMM's superiority over the baseline techniques. Statistically, the McNemar test [38] confirms that the differences in the repair results ("success" if a correct patch is produced for a bug, and "failure" otherwise) between PReMM and each baseline tool were significant ($52.5 \leq \chi^2 \leq 157.0$, $p < 10^{-11}$)[5]. Besides, in terms of effect size, the differences between PReMM and three baseline tools, namely ThinkRepair, Mulpo, and ChatRepair, are medium, with the odds ratios (OR)[6] being 3.13, 4.04, and 4.45, respectively, and the differences between PReMM and the remaining five baseline tools are large, with the odds ratios being greater than 5.2 [1, 6, 9].

Single- vs. Multi-method Bugs. Considering single-method bugs, among all baseline techniques, ThinkRepair performed best, correctly repairing 205 such bugs in Defects4J and achieving a c-recall of 37.6%. Compared to that, PReMM produced correct patches for 262 single-method bugs in total and achieved a c-recall of 48.1%, improving ThinkRepair's c-recall by 27.9%. This result clearly demonstrates PReMM's extraordinary repairing power on single-method bugs.

Considering multi-method bugs, baseline techniques like ChartRepair and ThinkRepair failed to generate correct repairs for any of them since the techniques were not designed for that kind of task, while the other baseline techniques were able to produce *valid* or *correct* patches for some multi-method bugs. For instance, Mulpo correctly repaired 26 such bugs and achieved a c-recall of 14.0% [29], outperforming all the other baseline techniques. Since these techniques lack proper design to correctly handle the dependence relation among faulty methods, they often end up treating each faulty method individually and generating multi-method patches by naively combining the candidate patches for those methods. In contrast to that, PReMM explicitly incorporates three core component techniques to effectively address multi-method bugs. This targeted approach enabled PReMM to correctly repair 45 multi-method bugs, i.e., 19, or 73.1%, more than Mulpo.

Generalizability across Datasets. Compared with Defects4J V1.2, the repair capability of tools such as ChatRepair, Mulpo, TENURE, Gamma, KNOD, and AlphaRepair tends to degrade when repairing bugs in Defects4J V2.0, which is consistent with trends observed in prior work [32, 61]. This drop in performance is likely due to the increased difficulty of the bugs in Defects4J V2.0. In contrast to that, ThinkRepair and PReMM remained comparably effective on both versions of Defects4J, highlighting both tools' generalizability across different datasets.

[5] $\chi^2 = (b-c)^2/(b+c)$, where b denotes the number of bugs correctly repaired by PReMM but not by the baseline, and c is the reverse. The p-value quantifies the likelihood that the differences in the repairing results were purely by chance.
[6] The odds ratio is defined as $OR = b/c$. An OR value within the range [0, 1.44), [1.44, 2.48), [2.48, 4.27), and [4.27,∞) indicates a very small, small, medium, and large effect, respectively.

Table 4. PReMM Variants and the Numbers of Bugs They Correctly Repaired.

Tool	Technique				Overall			Defects4J V1.2			Defects4J V2.0		
	F	C	D	I	Tot	#SM	#MM	Tot	#SM	#MM	Tot	#SM	#MM
PReMM-FCDI	✗	✗	✗	✗	119	105	14	54	43	11	65	62	3
PReMM-FCD	✗	✗	✗	✓	218	195	23	105	91	14	113	104	9
PReMM-CD	✓	✗	✗	✓	222	195	27	107	91	16	115	104	11
PReMM-FC	✗	✗	✓	✓	235	208	27	110	92	18	125	116	9
PReMM-FD	✗	✓	✗	✓	229	206	23	104	90	14	125	116	9
PReMM-F	✗	✓	✓	✓	298	262	36	143	121	22	155	141	14
PReMM-C	✓	✗	✓	✓	238	208	30	111	92	19	127	116	11
PReMM-D	✓	✓	✗	✓	233	206	27	105	90	15	128	116	12
PReMM	✓	✓	✓	✓	307	262	45	147	121	26	160	141	19

Overall, PReMM achieves top performance on Defects4J, delivering 49 and 53 more correct repairs than the current state-of-the-art APR tool, ThinkRepair, on Defects4J V1.2 and V2.0, respectively. Moreover, in 12 out of the 18 projects in the Defects4J benchmark, PReMM achieves the highest number of correct repairs, underscoring its robust performance across diverse codebases. These results demonstrate PReMM's ability to generalize effectively to various project types and defect categories, surpassing previous tools in both the quantity and quality of generated repairs.

Unique Bugs Repaired. To better understand the relative strengths of PReMM and the baseline techniques, we analyze and compare the unique bugs correctly repaired by each tool. Overall, PReMM correctly repaired 86 unique bugs from the Defects4J datasets, among which 23 were multi-method bugs, and the other 63 were single-method bugs. Following previous work [55, 59], we break down the repair results for Defects4J V1.2 and V2.0 separately, highlighting the unique contributions of PReMM.

For Defects4J V1.2, Figure 8a presents the Venn diagram of the unique bugs correctly fixed by all the studied baselines and PReMM. We selected the top three baselines (i.e., ChatRepair, ThinkRepair, and Mulpo) based on the number of bugs correctly fixed while grouping all other APR tools under the label "Others." As shown in the diagram, PReMM successfully generated correct patches for 35 unique bugs that no prior approach has been able to fix on Defects4J V1.2. Notably, the number of unique patches generated by PReMM is far larger than the other top 3 tools, further demonstrating its superior ability to address challenging defects.

For Defects4J V2.0, Figure 8b presents a similar Venn diagram comparing the unique bugs correctly fixed by PReMM and the top three baselines (i.e., ThinkRepair, Mulpo, and TENURE), with all other tools grouped under the label "Others." As with Defects4J V2.0, PReMM demonstrates its strength by generating 51 unique patches that no other baseline has successfully fixed. This reinforces the versatility and power of PReMM, as it continues to outperform existing tools on more challenging bugs in Defects4J V2.0.

> *Answer to RQ2: Compared with the baseline techniques, PReMM produced correct patches for substantially more single- and multi-method bugs, among which 86 were not successfully repaired by any of the baseline techniques before, and its effectiveness generalizes better across datasets.*

4.5.3 RQ3: Ablation Study. To evaluate the contribution of each core component in PReMM, namely faulty method clustering, fault context extraction, and dual-agent-based patch generation, we construct eight variants of PReMM, each with one or more of PReMM's component techniques being disabled, and compare their effectiveness in program repair with that of PReMM's.

All variants are trimmed versions of PReMM and named by appending a different postfix to the string "PReMM-", with each letter in the postfix representing one component technique that is present in PReMM but disabled in the variant. More concretely, the letters and the component techniques they represent are as follows: F for faulty method clustering, C for fault context extraction, D for dual-agent-based patch generation, and I for iterative LLM-based repairing. Note that, we investigate in this study the contribution of iterative LLM-based repairing because it has been implemented in some, but not all, LLM-based baseline techniques, and we believe it is a major factor influencing repairing techniques' effectiveness. We do not consider iterative LLM-based repairing a novel contribution of this work or a core component technique of PReMM.

In particular, the eight variants are as follows:

- PReMM-FCDI takes a buggy program and the bug location information as the input and solely relies on the plain Qwen2.5-72B-Instruct model to generate repairs. It represents the lower boundary of all the variants in terms of effectiveness.
- PReMM-FCD expects the input as by PReMM-FCDI and the failing test cases and iterative feedback and invokes the Qwen2.5-72B-Instruct model repeatedly to repair the faulty program, but it does not incorporate any of PReMM's core component techniques.
- PReMM-CD has faulty method clustering and iterative repairing enabled and fault context extraction and dual-agent-based patch generation disabled.
- PReMM-FC has dual-agent-based patch generation and iterative repairing enabled and faulty method clustering and fault context extraction disabled.
- PReMM-FD has fault context extraction and iterative repairing enabled and faulty method clustering and dual-agent-based patch generation disabled.
- PReMM-F has faulty method clustering disabled and the other component techniques enabled. i.e., PReMM-F functions the same as PReMM except that the faulty methods are not clustered during repairing. In other words, PReMM-F always treats all faulty methods as a single cluster and generates patches for them as a whole.
- PReMM-C has fault context extraction disabled and the other component techniques enabled. i.e., PReMM-C functions the same as PReMM except that the extra fault context information (i.e., invocation chain information, similar code segments, and key tokens) is not extracted or utilized to guide the diagnosis and repair of the bug.
- PReMM-D has dual-agent-based patch generation disabled and all the other component techniques enabled. PReMM-D functions like PReMM but lacks the collaborative mechanism between the diagnostic and repair agents.

Table 4 lists the 8 variants and, for each variant, the component techniques enabled/disabled (Technique), the number of all (Tot), single-method (#SM), and multi-method (#MM) bugs it correctly repaired from both datasets (Overall), and the breakdown of that to Defects4J V1.2 and V2.0. We reproduce the corresponding information of PReMM in the table for easy reference.

PReMM-FCDI correctly repaired 119 bugs in total, and iterative repairing enabled it to correctly repair 109 more bugs. This shows that iterative repairing alone can provide a substantial performance boost, even without the core component techniques proposed in this work. Still, PReMM outperformed PReMM-FCD by repairing 89 more bugs, and the vast improvement convincingly demonstrates the effectiveness of the three core component techniques when combined together. Next, we analyze the impact of each individual technique on the results achieved in program repair.

Impact of Faulty Method Clustering. While *faulty method clustering* does not impact the repair of single-method bugs, it plays a critical role in repairing multi-method bugs. To obtain a more comprehensive understanding of the technique's impact under different circumstances, we examined the effectiveness changes caused by *faulty method clustering* when it is combined with

other component techniques. The results show that it helped PReMM-FCD, PReMM-FC, PReMM-FD, and PReMM-F correctly repair 4, 3, 4, and 9 additional multi-method bugs, respectively, leading to an increase of 17.4% (=4/23), 11.1% (=3/27), 17.4% (=4/23), and 25.0% (=9/36) in c-recall.

Faulty method clustering yielded the most significant improvement on PReMM-F, correctly repairing 9 more multi-method bugs. We manually inspected the multi-method bugs successfully repaired by PReMM and made the following observations: 1) 44.4% (50/45) of the successfully repaired multi-method bugs have faulty methods that are not partitioned into different clusters after FMC, but they are still within PReMM-F's repairing capability and can be correctly repaired by both PReMM-F and PReMM. 2) 55.6% (25/45) of the successfully repaired multi-method bugs have faulty methods that are partitioned into different clusters after FMC. Among these bugs, 9 were only repaired by PReMM after the faulty methods were partitioned into smaller clusters, while the other 16 bugs were also repaired by PReMM-F without clustering the faulty methods, which highlights the repairing power of PReMM-F.

Impact of Fault Context Extraction. Similarly, our analysis of the effectiveness changes resulting from the integration of *fault context extraction* shows that this component enabled enabled PReMM-FCD, PReMM-FC, PReMM-CD, and PReMM-C to correctly repair 10, 63, 10, and 69 additional bugs, respectively, resulting in increases of 5.0% (=11/218), 26.8% (=63/235), 5.0% (=11/222), and 29.0% (=69/238) in c-recall. It is clear from the comparison that the effectiveness of *fault context extraction* depends heavily on the presence of *dual-agent-based patch generation*. On the one hand, when dual agents were disabled (e.g., in PReMM-FCD and PReMM-CD), the extra fault context information only slightly increased the c-recall (e.g., by 5.0% and 5.0%), probably because the extra fault information becomes overwhelming or misleading when provided together with the other inputs to LLMs for program repair. This observation aligns with our earlier concerns that excessive context may mislead LLMs, mirroring findings by Liu et al [33]. On the other hand, when dual agents were present (e.g., in PReMM-FC and PReMM-C), the extra fault context information can greatly increase the c-recall (e.g., by 26.8% and 29.0%), probably because the extra fault context information can effectively help the fault analysis agent precisely diagnose the fault and guide the patch generation agent to produce patches that are more likely to be correct.

Impact of Dual-Agent-Based Patch Generation. The *dual-agent-based patch generation* mechanism also contributed substantially to the overall repair performance. Results show that it enabled PReMM-FCD, PReMM-FD, PReMM-CD, and PReMM-D correctly repair 17, 69, 16, and 74 additional bugs, respectively, resulting in increases of 7.8% (=17/218), 30.1% (=69/229), 7.2% (=16/222), and 31.8% (=74/233) in c-recall. The performance gains achieved by enabling dual-agent-based patch generation vary significantly depending on the presence of other components, especially fault context extraction, which highlights the importance of synergy between this component technique and the others. When fault context extraction was disabled (e.g., in PReMM-FCD and PReMM-CD), the improvements were moderate (7.8% and 7.2%, respectively); when fault context extraction was enabled (e.g., in PReMM-FD and PReMM-D), the improvements were much more pronounced (30.1% and 31.8%, respectively). This suggests that the dual agents alone provide incremental benefits but require extra information about the fault context to realize their full potential.

> *Answer to RQ3: All three core component techniques helped PReMM correctly repair more Defects4J bugs. Faulty method clustering was critical for repairing multi-method bugs. Fault context extraction and dual-agent-based patch generation were the most effective when applied together.*

Table 5. Bug Classification in Defects4J-Trans and GitBug-Java, and the Number of Bugs Correctly Repaired by PReMM and ThinkRepair†. Note That All Bugs in Defects4J-Trans Are Single-method Bugs.

Datasets	# Bug			ThinkRepair†			PReMM		
	Tot	#SM	#MM	Tot	#SM	#MM	Tot	#SM	#MM
Defects4J V1.2-Trans	255	255	0	51	51	-	83	83	-
Defects4J V2.0-Trans	228	228	0	56	56	-	99	99	-
Defects4J-Trans	483	483	0	107	107	-	182	182	-
GitBug-Java	199	132	47	16	16	0	41	35	6

4.6 Data Contamination

Due to possible data contamination, the repair results of LLM-based techniques on Defects4J bugs may not be representative of the techniques' actual performance in practice [45]. In view of that, we conducted extra experiments to assess PReMM's repairing power on bugs outside the Defects4J benchmark. In this section, we report on these experiments and the corresponding findings.

We adopted bugs from two benchmarks, namely Defects4J-Trans [24] and GitBug-Java [48], which are less likely to be contaminated, as subjects for our experiments. The Defects4J-Trans dataset contains 483 bugs derived by applying semantic-preserving transformation operations to the 483 *single-method* bugs in the Defects4J benchmark (versions 1.2 and 2.0), and it was first released in October 2024 (i.e., one month after Qwen2.5-72B-Instruct was released). The GitBug-Java dataset contains 199 bugs collected from the commit histories of 55 Java projects, and prior work [45] has shown that it faces a limited risk of data contamination. Among the 199 bugs, 132 involve a single buggy method, 47 involve multiple buggy methods, and 20 involve buggy code elements outside methods (e.g., class fields), which are naturally beyond the fixing power of PReMM. Table 5 lists, for each dataset, the total number of bugs and their breakdown into single- and multi-method bugs.

A good baseline technique to use in this study should both represent the state of the art in LLM-based program repair and, when compared with PReMM, suffer/benefit as little as possible from the disadvantages/advantages of its underlying LLM. In view of that, we construct the baseline technique by using ThinkRepair, the technique that delivers the best overall effectiveness among existing program repair techniques, as the basis, replacing its underlying LLM with Qwen2.5-72B-Instruct, and adopting the same settings for Qwen2.5-72B-Instruct as in PReMM. We refer to this variant of ThinkRepair as ThinkRepair†.

We then follow the same experimental protocol described in Section 4.3 and apply PReMM and ThinkRepair† to repair the bugs from the two benchmarks. Table 5 reports the same measures of the repairing results as listed in Table 3 for each benchmark and technique.

Overall, PReMM correctly repaired 182 of the (single-method) bugs from Defects4J-Trans and 41 of the bugs from GitBug-Java, achieving a c-recall of 37.7% and 20.6%, respectively. In particular, the c-recall PReMM achieved on multi-method bugs from GitBug-Java is 12.8%, which is significant, considering the challenges involved in correctly repairing multi-method bugs. Such c-recall values are at the same level of magnitude as the c-recall value achieved by PReMM on Defects4J bugs (36.0%), suggesting that PReMM can still be highly effective in repairing new bugs, i.e., bugs unlikely to be used for training its underlying LLM.

Compared with PReMM, ThinkRepair† correctly repaired 107 and 16 bugs from Defects4J-Trans and GitBug-Java, achieving a c-recall of 22.1% and 8.0%, respectively, and it did not manage to correctly repair any of the multi-method bugs in GitBug-Java. Hence, PReMM improved the c-recall achieved by ThinkRepair† on Defects4J-Trans and GitBug-Java by 70.1% and 156.3%,

respectively. The improvement is substantially greater than what's been observed on Defects4J, highlighting again PReMM's superiority over ThinkRepair†.

In summary, PReMM's overall effectiveness on datasets Defects4J-Trans and GitBug-Java was comparable to that achieved on Defects4J bugs, and the improvement PReMM brought about over ThinkRepair† in repairing Defects4J-Trans and GitBug-Java bugs was greater than that observed on Defects4J.

> *PReMM's effectiveness and its superiority over the state-of-the-art program repair techniques, as demonstrated in Section 4.5.2, is not significantly affected by the data contamination issue.*

4.7 Threats to Validity

4.7.1 Construct Validity. Construct validity concerns whether the measures used in the experiments capture salient features. We classify a patch as *correct* if it is semantically equivalent to a given programmer-written fix. Since we assess equivalence manually, others may disagree with some of our assessments. To mitigate this threat, we followed the same methodology as previous studies [54, 55, 59], and we were conservative in evaluating equivalence: each patch was subjected to a thorough examination and comprehensive discussion, with evaluations conducted by two PhD-level computer science experts; the patch evaluations were carried out independently and cross-verified to ensure consistency and reliability; if a patch does not clearly introduce the same behavior as the reference patch, we classify it as incorrect.

4.7.2 Internal Validity. Internal validity concerns whether the experiments controlled for possible confounders. An important threat to the internal validity of our experimental findings comes from possible data contamination. That is, some developer-generated patches might overlap with the training data used by the LLM. Our analysis revealed that 43.0%(132/307) of the correct patches generated by PReMM exactly matched developer fixes. However, beyond these matching patches, PReMM also generated 175 unique correct patches, which underscores its capability to produce novel fixes. Moreover, PReMM produced 188 more correct patches than the underlying LLM (Section 4.5.3). This improvement suggests that PReMM 's effectiveness is not merely a result of memorizing history patches. To mitigate the threat that bugs in Defects4J may have been contaminated and the repair results reflect memorization rather than true repair capability, we evaluated PReMM on two extra datasets of bugs, namely Defects4J-Trans and GitBug-Java, that are less likely to be affected by data contamination. PReMM's effectiveness on the two datasets was comparable to that achieved on Defects4J bugs, and the improvement it brought about over the state-of-the-art program repair techniques was greater than that observed on Defects4J, suggesting that the experimental results PReMM achieved on Defects4J were not significantly affected by the data contamination issue.

4.7.3 External Validity. External validity concerns whether the experimental findings generalize to other contexts. Evaluating PReMM on the two additional datasets of bugs, Defects4J-Trans and GitBug-Java, also helped mitigate the risk that Defects4J bugs are not good representatives of new bugs in real-world software systems, and the experimental results clearly demonstrated the generalizability of our findings on Defects4J. Considering that PReMM is not limited to using Qwen as the base LLM, and it can be easily configured to use other LLMs like ChatGPT and DeepSeek, we plan to conduct comprehensive experiments in the future to evaluate the impact of different underlying LLMs on PReMM's repairing power.

5 Related Work

Our work is closely related to existing work in the following areas.

5.1 Large Language Model

Large Language Models (LLMs) [4] have gained widespread adoption due to significant advancements in Natural Language Processing (NLP). These advancements enable LLMs to be extensively trained using billions of parameters and training samples, resulting in substantial performance improvements. LLMs can be readily adapted for downstream tasks either through fine-tuning [44] or prompting [34], as they are designed to be general-purpose models capable of capturing diverse knowledge from various domain-specific data. Fine-tuning involves updating model parameters for a specific downstream task by training the model on a targeted dataset. In contrast, prompting allows users to directly guide the model by providing natural language instructions or a few examples of the task. Compared to prompting, fine-tuning is more resource-intensive, as it requires additional training and is less practical in scenarios where sufficient labeled datasets are unavailable.

ChatGPT [47], an evolution of InstructGPT [42], is fine-tuned using the Reinforcement Learning with Human Feedback (RLHF) approach [8, 42, 62]. The RLHF process begins by fine-tuning the model using a small dataset of prompts paired with desired outputs, typically written by humans. Next, a reward model is trained on a larger set of prompts by sampling outputs generated by the fine-tuned model, which are then ranked by human evaluators. Finally, reinforcement learning [46] is employed to compute rewards for each generated output based on the reward model, subsequently updating the LLM parameters accordingly. By leveraging fine-tuning and aligning with human preferences, LLMs achieve a deeper understanding of input prompts and instructions, enabling superior performance across a wide range of downstream tasks [2, 42].

5.2 Automated Program Repair (APR)

Broadly speaking, the existing APR techniques can be classified into traditional APR approaches, learning-based APR approaches, and Large Language Model (LLM)-based APR approaches.

Traditional APR Approaches. The traditional APR approaches primarily involves three kinds: heuristic-based [21, 23, 52], constraint-based [10, 20, 35, 39], and template-based ones [13, 15, 31, 32, 37]. The heuristic-based approaches typically leverage heuristic techniques (e.g., genetic programming) to construct a search space based on previous patches and then search for valid patches within this space. The constraint-based approaches focus on a single conditional expression in the program and synthesize patches using constraint-solving or synthesis techniques. The template-based approaches typically use pre-defined repair templates that are often hand-crafted by experts based on specific software bugs to correct buggy programs. However, the generalizability of these classic APR approaches is limited because they either target specific types of bugs (e.g., bugs in conditional expressions) or have limited search spaces for the patches (e.g., spaces defined by lists of templates based on existing bugs). On the contrary, LAMAR explores the extensive knowledge within LLMs to generate patches, thereby achieving better generalizability.

Learning-Based APR Approaches. Recently, with the rapid success of deep learning technologies, a variety of learning-based APR approaches have been proposed to automatically learn bug-fixing patterns [11, 16, 17, 26–29, 36, 40, 57, 58, 61]. RewardRepair [58] trains a model with dynamic domain data. KNOD [16] uses expensive static domain data to train models. TENURE [40] constructs two large-scale datasets for training the model. Mulpor [29] adopts unsupervised pre-training tasks to acquire semantic and syntactic knowledge, then fine-tunes on a real-world bug dataset, and is ultimately used for model training. These learning-based approaches generally treat the program repair problem as a neural machine translation (NMT) task to transform buggy code into correct code by training an NMT model using a dataset of historical bug fixes. While DL models show powerful capability in learning hidden repair patterns from massive code corpora,

the performance of the DL models relies heavily on their training data. However, it is challenging to obtain high-quality training data for these models, and furthermore, they may fail to generalize to bug fix types not seen in their training data. PReMM avoids training a new DL model for repair generation but directly leverages LLMs, which are pre-trained on billions of code snippets and thus acquire a vast repository of knowledge.

LLM-Based APR Approaches. Very recently, LLMs have been successfully applied to a broad range of code-related tasks and demonstrated impressive performance. Inspired by this, researchers have started to directly leverage advanced LLMs for APR [19, 43, 53–55, 59]. AlphaRepair [54] is the first tool designed for cloze-style Automated Program Repair (APR), and its performance demonstrates that LLM-based APR surpasses the widely studied NMT-based APR techniques in real-world systems. On the other hand, ChatRepair [55] leverages the conversational capabilities of ChatGPT, iteratively incorporating test information to derive the final patch. However, its effectiveness is heavily influenced by the quality of prompts. ThinkRepair [59] introduces the first single-function repair paradigm, combining a set of knowledge-driven thought processes with interactive LLM-based repair. By exploring the extensive knowledge encoded in LLMs, the LLM-based APR approaches mitigate the reliance on historical bug fixes, enabling their application to a wider range of repair scenarios. In fact, the LLM-based APR approaches have already become the new state-of-the-art in the APR community. While promising, existing LLM-based APR approaches mainly focus on repairing a single function with straightforward fault contexts, overlooking the problem of cross-method program repair, which is often encountered in real-world scenarios. In this work, we propose a novel LLM-based approach for cross-method program repair, extending the scope of current APR techniques.

6 Conclusion

In this paper, we propose the PReMM technique that implements the divide-and-conquer strategy for effective repair of multi-method bugs. Given a faulty program to repair, location information about the fault, and a collection of passing and failing test cases, PReMM partitions the faulty methods into clusters based on the test- and invocation-dependence relation among them, extracts extra contextual information about the fault, and employs two LLM-based agents to diagnose the fault and generate desirable patches iteratively. Experimental evaluation results of PReMM on real-world bugs from datasets like Defects4J, Defects4J-Trans, and GitBug-Java clearly demonstrate that PReMM is effective in correctly repairing both single- and multi-method bugs, it considerably outperformed the state-of-the-art NMT- and LLM-based program repair techniques, and all three core component techniques play essential roles in helping PReMM achieve its effectiveness.

Data Availability

The PReMM tool and a replication package are available [30] at https://github.com/LinnaX7/PReMM.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 62402214 and 62232014, the Natural Science Foundation of Jiangsu Province under Grant No. BK20241194, and the China Postdoctoral Science Foundation under Grant Number 2025T180420. Zhong Li is supported by the Postdoctoral Fellowship Program of CPSF under Grant Number GZB20250386. Yu Pei is supported in part by the Hong Kong Polytechnic University Fund under Grants P0051205 and P0051074.

References

[1] 2025. Interpret Odds Ratio - R Method. https://easystats.github.io/effectsize/reference/interpret_oddsratio.html. Accessed: 2025-07.

[2] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Jong C. Park, Yuki Arase, Baotian Hu, Wei Lu, Derry Wijaya, Ayu Purwarianti, and Adila Alfa Krisnadhi (Eds.). Association for Computational Linguistics, Nusa Dua, Bali, 675–718. https://doi.org/10.18653/v1/2023.ijcnlp-main.45

[3] Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Usvsn Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, Angela Fan, Suzana Ilic, Thomas Wolf, and Matthias Gallé (Eds.). Association for Computational Linguistics, virtual+Dublin, 95–136. https://doi.org/10.18653/v1/2022.bigscience-1.9

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[5] chatgptendpoint. 2023. Introducing ChatGPT and Whisper APIs. https://openai.com/blog/introducing-chatgpt-and-whisper-apis.

[6] Henian Chen, Patricia Cohen, and Sophie Chen. 2010. How Big is a Big Odds Ratio? Interpreting the Magnitudes of Odds Ratios in Epidemiological Studies. *Communications in Statistics - Simulation and Computation* 39 (03 2010), 860–864. https://doi.org/10.1080/03610911003650383

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[8] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4302–4310.

[9] J. Cohen. 2013. *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis.

[10] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (Hyderabad, India) *(CSTVA 2014)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/2593735.2593740

[11] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2021. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. arXiv:2105.09352 [cs.SE] https://arxiv.org/abs/2105.09352

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE] https://arxiv.org/abs/2204.05999

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[14] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[15] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 888–891. https://doi.org/10.1145/3236024.3264600

[16] 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 *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1251–1263. https://doi.org/10.1109/ICSE48619.2023.00111

[17] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[18] 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

[19] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*. https://openreview.net/forum?id=rHlzJh_b1-5

[20] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[21] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

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

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

[24] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Evaluating the Generalizability of LLMs in Automated Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 91–95. https://doi.org/10.1109/ICSE-NIER66352.2025.00024

[25] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Randy, M-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, S. Gunasekar, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. [n.d.]. StarCoder: May the Source be With You! *Transactions on machine learning research* ([n. d.]). https://par.nsf.gov/biblio/10483982

[26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. Improving automated program repair using two-layer tree-based neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 316–317. https://doi.org/10.1145/3377812.3390896

[28] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[29] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1554–1566. https://doi.org/10.1145/3650212.3680381

[30] LinnaX7. 2025. *LinnaX7/PReMM: PReMM: LLM-Based Program Repair for Multi-Method Bugs via Divide and Conquer.* https://doi.org/10.5281/zenodo.16927561

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyande. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations . In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/SANER.2019.8667970

[32] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638

[34] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (Jan. 2023), 35 pages. https://doi.org/10.1145/3560815

[35] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[36] 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 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[37] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[38] Quinn McNemar. 1947. Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages. *Psychometrika* 12, 2 (1947), 153–157. https://doi.org/10.1007/BF02295996

[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[40] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-Based Neural Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1456–1468. https://doi.org/10.1109/ICSE48619.2023.00127

[41] OpenCompass. 2024. OpenCompass Rank. (2024). https://rank.opencompass.org.cn/home.

[42] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2011, 15 pages.

[43] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair* (Pittsburgh, Pennsylvania) *(APR '22)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3524459.3527351

[44] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[45] Daniel Ramos, Cláudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2024. Are Large Language Models Memorizing Bug Benchmarks? *CoRR* abs/2411.13323 (2024). https://doi.org/10.48550/ARXIV.2411.13323 arXiv:2411.13323

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[47] John Schulman, Barret Zoph, Jacob Hilton Christina Kim, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, and Nick Ryder.

2022. ChatGPT: Optimizing Language Models for Dialogue. (2022). https://openai.com/blog/chatgpt/.
[48] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 118–122.
[49] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/
[50] Yue Wang, Weishi Wang, Shafiq 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685
[51] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3611643.3616271
[52] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233
[53] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129
[54] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101
[55] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831. https://doi.org/10.1145/3650212.3680323
[56] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2023. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 107, 13 pages. https://doi.org/10.1145/3551349.3556893
[57] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages. https://doi.org/10.1145/3551349.3556926
[58] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222
[59] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1274–1286. https://doi.org/10.1145/3650212.3680359
[60] 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 *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 535–547. https://doi.org/10.1109/ASE56229.2023.00063
[61] 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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544
[62] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2020. Fine-Tuning Language Models from Human Preferences. arXiv:1909.08593 [cs.CL] https://arxiv.org/abs/1909.08593

Received 2025-03-26; accepted 2025-08-12