PATCH: **Empowering Large Language Model with Programmer-Intent Guidance and Collaborative-Behavior Simulation for Automatic Bug Fixing**

YUWEI ZHANG, Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China
ZHI JIN*, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University; School of Computer Science, Wuhan University, China
YING XING, School of Intelligent Engineering and Automation, Beijing University of Posts and Telecommunications, China
GE LI*, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China
FANG LIU, State Key Laboratory of Complex & Critical Software Environment; School of Computer Science and Engineering, Beihang University, China
JIAXIN ZHU†, WENSHENG DOU†, and JUN WEI*†, Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

Bug fixing holds significant importance in software development and maintenance. Recent research has made substantial strides in exploring the potential of large language models (LLMs) for automatically resolving software bugs. However, a noticeable gap in existing approaches lies in the oversight of collaborative facets intrinsic to bug resolution, treating the process as a single-stage endeavor. Moreover, most approaches solely take the buggy code snippet as input for LLMs during the patch generation stage. To mitigate the aforementioned limitations, we introduce a novel stage-wise framework named PATCH. Specifically, we first augment the buggy code snippet with corresponding dependence context and intent information to better guide LLMs in generating the correct candidate patches. Additionally, by taking inspiration from bug

*Corresponding authors
†Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing, China.

Authors' Contact Information: Yuwei Zhang, zhangyuwei@iscas.ac.cn, Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China; Zhi Jin, zhijin@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University;, Beijing and School of Computer Science, Wuhan University, Wuhan, China; Ying Xing, xingying@bupt.edu.cn, School of Intelligent Engineering and Automation, Beijing University of Posts and Telecommunications, Beijing, China; Ge Li, lige@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Fang Liu, fangliu@buaa.edu.cn, State Key Laboratory of Complex & Critical Software Environment; School of Computer Science and Engineering, Beihang University, Beijing, China; Jiaxin Zhu, zhujiaxin@otcaix.iscas.ac.cn; Wensheng Dou, wsdou@otcaix.iscas.ac.cn; Jun Wei, wj@otcaix.iscas.ac.cn, Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China.

ACM 1557-7392/2025/2-ART
https://doi.org/10.1145/3718739

management practices, we decompose the bug-fixing task into four distinct stages: bug reporting, bug diagnosis, patch generation, and patch verification. These stages are performed interactively by LLMs, aiming to simulate the collaborative behavior of programmers during the resolution of software bugs. By harnessing these collective contributions, PATCH effectively enhances the bug-fixing capability of LLMs. We implement PATCH by employing the powerful dialogue-based LLM ChatGPT. Our evaluation on the widely used bug-fixing benchmark BFP demonstrates that PATCH has achieved better performance than state-of-the-art LLMs.

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

Additional Key Words and Phrases: Bug Fixing, Large Language Model, Bug Management, Multi-Agent Collaboration

1 INTRODUCTION

Software systems, by virtue of inherent complexity and inadequate testing, inevitably contain bugs that can lead to substantial losses [88, 101]. To expedite the resolution of software bugs, automatic bug fixing [105] has been proposed as a means to mitigate the costs associated with software debugging. Traditional approaches generally entail mutating the buggy code through predefined search strategies [19, 20]. Nevertheless, these approaches face challenges primarily due to the time-intensive nature of validation strategies, such as verifying the correctness of generated patches using exhaustive test suites. With the rapid advancements in deep learning (DL), there has been a surge of interest in neural-based bug-fixing approaches [104, 112], exploiting the powerful representation capabilities of DL models to autonomously learn bug-fixing patterns. However, existing neural-based approaches [30, 47, 98, 107, 114, 115] collect historical bug-fixing datasets sourced from open-source code repositories for supervised training, which may restrict their generalizability to unseen bug types [90]. More recently, researchers have commenced leveraging large language models (LLMs) for bug fixing without the necessity of fine-tuning. The application of LLMs to bug fixing [25, 28, 89] involves devising prompts that can consist of either the buggy code alone or a combination of the buggy code and a few task-specific bug-fixing pairs, with the goal for LLMs to learn from the provided prompts and generate patches for the given buggy code. While current LLM-based approaches have demonstrated promising results compared to previous neural-based techniques, they still possess certain limitations as illustrated in Figure 1.

Limitation 1: Missing additional information associated with the buggy code as guidance for LLMs. The left part of Figure 1 presents a motivating example derived from the real-world bug-fixing benchmark BFP [81], which comprises the buggy method, highlighting its fault location (i.e., the buggy hunk), alongside the ground-truth patch written by the corresponding programmer. When provided with insufficient input information, the LLM outputs an incorrect candidate patch, predicting a different token (i.e., the variable **`md5hex`**) based on the buggy method content to replace the input parameter (i.e., the variable **`md5`**) of the function **`setMd5sum`** invoked via the variable **`meta`** within the buggy hunk. Notably, even experienced programmers find it challenging to identify the appropriate patch for fixing the buggy hunk by examining the code snippet of the buggy method alone. In this case, the object **`DefaultMetadata`** initialized by **`meta`** represents a cross-file data dependency within the corresponding code repository, which is evident from the import information (framed by the blue rectangle) provided in the buggy class file. Intuitively, when the repository-level dependencies (i.e., the contextual information of **`setMd5sum`**) are utilized as prompt inputs, the LLM can deduce that the data type of **`md5`** (i.e., **`byte[]`**) does not conform to the required input parameter type of **`setMd5sum`** (i.e., **String**). Furthermore, when human programmers encounter a bug during the real-world software development environment, they begin by debugging the buggy hunk, considering its surrounding context, and analyzing compiler-generated error messages to identify the root cause of the bug. Subsequently, they document their intent by summarizing the key information required to fix the corresponding bug via natural language. By considering the intent description provided by the human programmer (i.e., *fixing different data type errors*), we hypothesize that the LLM can reason more effectively about the necessary modifications for patching the buggy hunk.

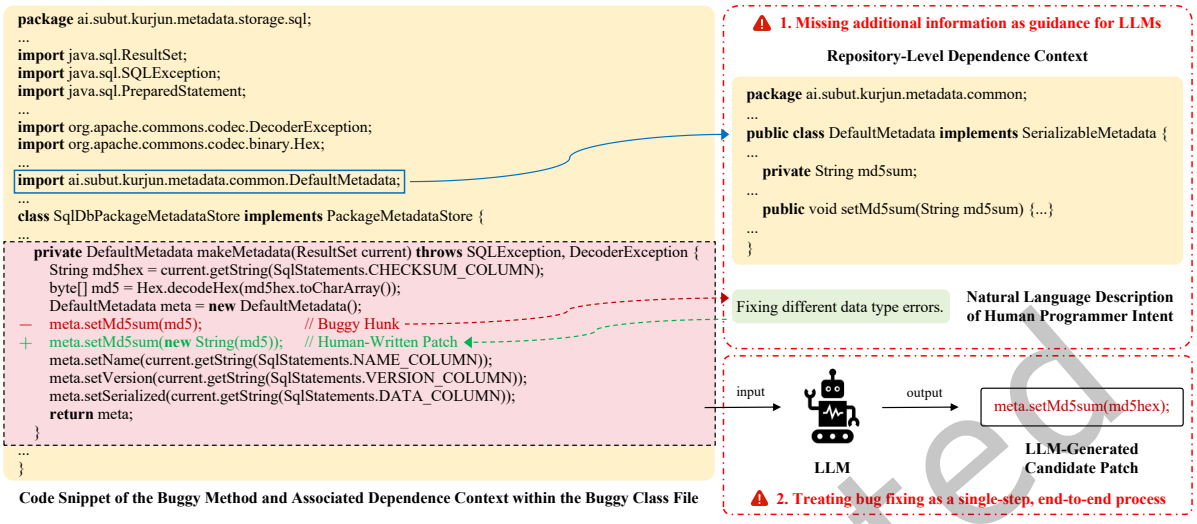

Fig. 1. Limitations of Existing LLM-Based Bug-Fixing Approaches.

Limitation 2: Treating the task of bug fixing as a single-step, end-to-end procedure. In practice, bug fixing is a multifaceted task wherein every discovered bug undergoes a specific and intricate process before being effectively resolved [14]. Although LLMs have demonstrated capabilities akin to human logical understanding [86], they continue to struggle with resolving bugs that involve nested program structures and cross-file dependence relationships. Software debugging inherently necessitates multi-step reasoning, a process that poses a considerable obstacle for LLMs that predominantly rely on pattern recognition rather than authentic cognitive processes. Consequently, this reliance limits the effectiveness of LLMs in resolving complex bugs. Human programmers, by contrast, tend to seek teamwork as a means of tackling intricate debugging-related tasks in software engineering (SE) practices [43, 51]. However, current LLM-based bug-fixing approaches typically focus on directly utilizing LLMs to generate candidate patches under the setting of zero-shot or few-shot prompting paradigms, neglecting the interactive and collaborative behaviors exhibited by human programmers (e.g., one reviewer will be responsible for verifying the correctness of the candidate patch generated by the corresponding developer) during the resolution of complex software bugs. Zeng et al. [102] conducted a comprehensive evaluation of bug-fixing performance using eight open-access state-of-the-art LLMs on the BFP benchmark. Their empirical findings reveal that all the evaluated LLMs exhibit low accuracy, with performance rates below 15% in the task of bug fixing. Given that the effectiveness of LLMs is highly contingent on the surface structure of the prompts used [109], it is imperative to explore more effective prompting techniques to enhance the ability of LLMs in generating correct patches.

To bridge the gap between the capabilities of LLMs and human programmers in bug fixing, this paper presents a stage-wise framework, referred to as PATCH. This framework introduces two novel mechanisms that empower the LLM with **P**rogr**A**mmer-in**T**ent guidance and **C**ollaborative-be**H**avior simulation. These mechanisms effectively address the two aforementioned limitations, resulting in a significant improvement in the bug-fixing performance of LLMs. The specific details of PATCH are outlined as follows.

Novelty 1: Augmenting the buggy code snippet with additional dependence context and guided programmer's intent as input to the LLM. This paper employs the widely-used benchmark BFP [81] for evaluation, which includes an extensive collection of paired bug-fixing instances across a diverse range of

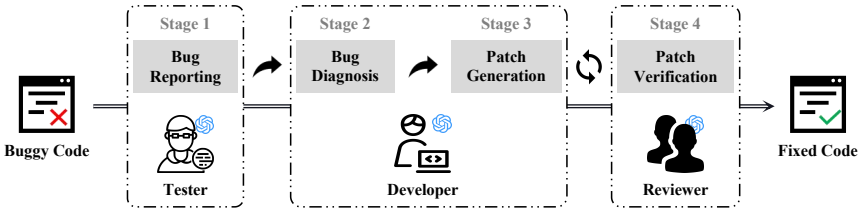

Fig. 2. The Brief Structure of PATCH.

real-world bugs, rather than limiting the scope to specific bug types [49]. We commence by enhancing the buggy code snippet through the incorporation of code dependency contexts extracted from both the class and repository levels. Additionally, we integrate the programmer's intent, articulated in natural language, as supplementary guidance. **Effective bug fixing necessitates a comprehensive grasp of both code semantics and the intent of human programmers. This augmentation aims to guide LLMs in generating effective and accurate candidate patches.** It is vital to leverage sufficient contextual information as fixing ingredients for patch generation. Moreover, understanding the context of a buggy code snippet is essential for human programmers in identifying the root cause of the discovered bug and proposing potential bug-fixing suggestions. Therefore, collecting adequate contextual information from the corresponding GitHub repository can be more effective in generating the correct patches. Furthermore, given the lack of test suites for the BFP benchmark, we leverage commit messages, which are written in natural language, as proxies for the programmer's intent. These messages serve as crucial artifacts within the continuous software development process [92]. As illustrated in Figure 1, it is important to note that the information provided solely describes the type of bug (e.g., *fixing different data type errors*) without detailing the specific process required for its resolution. Consequently, this information can be utilized to accurately simulate the actual development process, minimizing the risk of information leakage issue that could lead to unintended advantages for LLMs. Recent empirical investigations [7] have shown that leveraging the commit messages authored by corresponding programmers as supplementary guidance enhances the performance of LLMs by narrowing down the search space. Thus, we collect the bug-fixing commits associated with the buggy code snippets from GitHub to automatically simulate the programmer's intent. Such information facilitates LLMs in better comprehending the desired fixing goals, thereby improving the generation of candidate patches.

Novelty 2: Empowering the bug-fixing performance of LLMs by simulating the collaborative behaviors of programmers via effective bug management practices. While LLMs trained on code have achieved commendable efficacy in aiding human programmers, their proficiency diminishes notably when confronted with complex debugging-related SE tasks that require multi-step logical reasoning within programs. More recently, researchers have demonstrated the impressive capabilities of LLMs in generating helpful outcomes when tasks are disassembled into a set of modular units with precise queries [53, 86]. Recognizing the significance of an efficient bug management process for successful bug fixing [58], we decompose the task of bug fixing into four distinct stages: bug reporting, bug diagnosis, patch generation, and patch verification. **Drawing inspiration from bug management practices, we closely examine the programmers involved at various stages of the bug's life cycle and analyze the impact of their interactions on improving the efficiency of bug fixing.** As depicted in Figure 2, PATCH aims to imitate the collaborative problem-solving abilities exhibited by programmers (i.e., the tester, the developer, and the reviewer) throughout the entire life cycle of a bug. To be specific, effective bug resolution initially relies on the tester's comprehensive understanding of the bug, leading to the filing of a detailed bug report. This report provides essential information to the developer for successfully resolving the bug. Within PATCH, the developer has a two-fold responsibility. Firstly, the developer engages in the diagnosis

process by consulting historical bug corpus and conducting self-debugging. Secondly, the developer generates the candidate patch, guided by the information obtained in the previous stages. Since the correctness of the candidate patch generated on the first attempt cannot be guaranteed, the reviewer's involvement in PATCH becomes crucial. The reviewer provides the feedback and collaborates with the developer throughout the workflow, playing a vital role in ensuring the correctness of the generated patch.

In summary, PATCH breaks down the bug-fixing task into smaller, more manageable subtasks with the aim of improving the accuracy of automatic bug fixing by incorporating additional information and implementing efficient bug management practices. Moreover, by involving multiple programmers, PATCH can enable the inclusion of diverse perspectives and feedback to facilitate the bug-fixing process, thereby mitigating misunderstandings and ensuring the quality of the generated candidate patches. Given the remarkable advancements in generative artificial intelligence (AI), LLMs (e.g., ChatGPT [59]) have exhibited commendable performance across various SE tasks [3, 56, 63], opening avenues for inter-model interaction and collaboration. Specifically, PATCH employs three ChatGPT agents, each playing a distinct programmer role as showcased in Figure 2, to emulate collaborative efforts in real-world bug management practices. The main contributions of this paper can be summarized as follows:

- We present the first attempt at enhancing the capabilities of LLMs for automatic bug fixing by leveraging effective bug management practices. Our alignment approach simulates the interactive behavior of programmers engaged in bug management, which enables LLMs to collaborate and generate correct patches.
- We introduce a stage-wise framework called PATCH, consisting of three ChatGPT agents, each responsible for specific stages within the bug management process via system instructions and prompts. Our proposed framework shifts the focus from end-to-end bug fixing to a conversation-driven text-generation task, enhancing the understanding and utilization of LLMs in bug fixing.
- We construct a meta-rich bug-fixing benchmark that incorporates additional dependence context and the programmer's intent associated with the buggy code snippet. This augmentation aims to provide better guidance to LLMs in generating the correct patches for complex bugs.
- We conduct extensive experiments on publicly available bug-fixing benchmarks and thoroughly evaluate each component of the proposed framework. The experimental results demonstrate that PATCH surpasses state-of-the-art LLMs, highlighting its superior performance.
- We publicly release the replicate package [106] of PATCH on Zenodo. The open-sourced artifacts can better support the researchers in the SE community in reproducing PATCH.

Article Organization. The remainder of this paper is organized as follows: Section 2 introduces in detail the proposed framework. Section 3 and Section 4 provide the experimental setups and results of our research. Section 5 presents case studies and discloses the threats to the validity of our approach. Section 6 describes the related work. Section 7 draws conclusions and indicates directions for future work.

2 THE PROPOSED FRAMEWORK PATCH

In order to mitigate the limitations discussed in Section 1 concerning existing approaches, we present a novel stage-wise framework dubbed PATCH. This framework embodies a programmer-like behavior simulation aimed at augmenting LLMs in the task of bug fixing, leveraging beneficial SE practices (i.e., bug management). Within the process of bug management practice, human programmers engage in collaborative efforts to discover, report, and resolve software bugs, constituting a pivotal facet of the software development life cycle. Such standardized practices not only enhance collaboration efficacy within development teams but also serve to uphold software quality. With this inspiration, our main objective in this paper is to design system components that emulate the cognitive processes of different programmers involved in the bug management process by utilizing the powerful

conversation-based ChatGPT model. As illustrated in Figure 3, PATCH involves three ChatGPT agents (i.e., $\textbf{ChatGPT}_{\text{Tester}}$, $\textbf{ChatGPT}_{\text{Developer}}$, and $\textbf{ChatGPT}_{\text{Reviewer}}$), each assigned to specific stages (i.e., **Bug Reporting**, **Bug Diagnosis**, **Patch Generation**, and **Patch Verification**) within the bug management process.

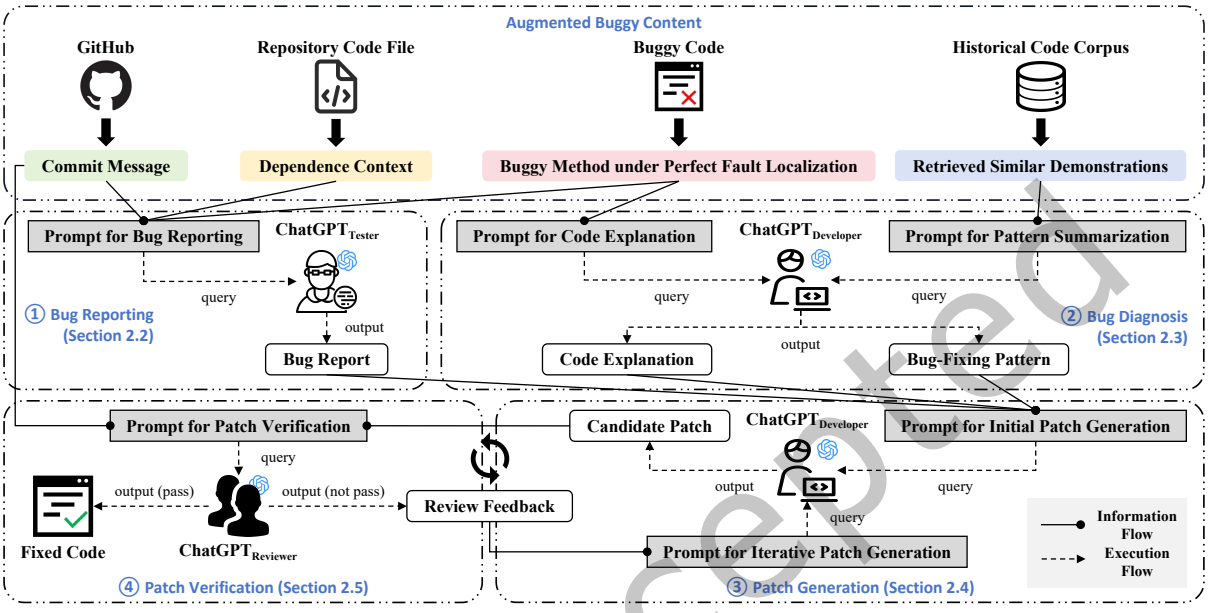

Fig. 3. Overview of PATCH.

2.1 Task Formulation

Before delving into the details of PATCH, this subsection provides a formal description of the bug-fixing task. Following recent studies that utilize LLMs for bug fixing [66, 75], this paper focuses on fixing single-hunk bugs written in the Java programming language. In this scenario, the generation of candidate patches necessitates program modifications, such as deletions, insertions, or replacements, either at a single line or within a consecutive chunk of code. This process operates under the assumption of perfect fault localization, meaning that LLMs are aware of the location information pertaining to the buggy code that requires to be fixed.

Specifically, this paper approaches the bug-fixing problem as a conversation-driven text-generation task leveraging the advancements in LLMs. Unlike existing LLM-based bug-fixing techniques, which generate candidate patches directly from the given buggy code, PATCH enhances the patch generation process with augmented buggy content (as shown in the upper part of Figure 3), consisting of the **buggy code snippet** with additional **dependence context** (if exists) at the class and repository levels, the programmer's intent (i.e., **commit message**) conveyed in the corresponding GitHub commit, and **similar bug-fixing demonstrations** retrieved from the collected historical code corpus. Regarding the extraction of the **dependence context**, we initially employ the static analysis tool Spoon [65] to parse the associated code files (i.e., the buggy class file and other dependent class files pertinent to cross-file dependencies) into abstract syntax trees (ASTs). Subsequently, we extract the necessary contextual dependencies related to the buggy method via data-flow analysis (e.g., definition-use chains). While traversing the AST of the buggy class file, PATCH gathers three types of information

as class-level dependencies: the imports of project-specific library depended upon by the buggy method, the global variables (defined within the buggy class scope) utilized in the buggy method, and the signatures of methods invoked within the buggy method. For repository-level dependencies, we extract the global variables utilized within the buggy method, along with the signatures of methods invoked by the buggy method, from the dependent classes that are imported in the buggy class. Due to the input token length limitation of LLMs, we restrict the extraction to only one layer of cross-file dependencies. Formally, given a new single-hunk bug, we propose leveraging the LLMs within PATCH to generate a candidate patch p for fixing the corresponding bug. The bug-fixing task is defined as $p = \texttt{LLM}(\textbf{PROMPT})$. Let $\textbf{PROMPT} = \mathbb{I} \oplus \mathbb{C} \oplus \mathbb{F}$ be the input prompt, where $\mathbb{I}$ denotes the system instruction for prompting LLM to align the collaborative behavior of human programmers, $\mathbb{C}$ denotes the augmented buggy content of the given single-hunk bug provided by PATCH, $\mathbb{F}$ denotes different dimensions of feedback information from earlier bug management stages, and $\oplus$ denotes the concatenation operation. Given **PROMPT**, the goal of LLM is to learn the conditional probability $\mathcal{P}(p|\textbf{PROMPT})$.

When engaged in a single-turn conversation, the system instruction and user-defined prompt are used as input to generate an assistant message with ChatGPT. The system instruction plays a crucial role in defining the behavior of the assistant, allowing the agents to simulate the corresponding programmer behaviors. The user-defined prompt serves as a means to convey requests or comments for the assistant to respond to. By utilizing specific prompts, PATCH effectively aligns the collaborative abilities of the programmers. This enables an interactive bug-fixing process, where the outputs from earlier stages are used to construct the input prompts for subsequent stages. As a result, PATCH optimizes the ability of LLMs to generate correct patches that fix the given bugs. While PATCH is a general framework capable of being applied to various LLMs, this paper employs the state-of-the-art ChatGPT model, which is tailored for dialogue-based interactions.

2.2 Bug Reporting

During the initial phase of bug management, the tester identifies a bug within the source code and proceeds to file a detailed report elucidating the nature of the bug. In practice, bug reports play a crucial role in bug fixing as they provide the developer with essential information regarding the discovered bug. These specific details significantly assist the developer in resolving the bug [117, 118]. To simulate the tester's behavior, PATCH is designed to generate an initial bug report that outlines the underlying cause of the given buggy code.

Figure 4 illustrates the prompt generated to query **ChatGPT$_{\text{Tester}}$** during the bug reporting stage, along with the corresponding output. The **System Instruction** specifies the persona adopted by **ChatGPT$_{\text{Tester}}$** in its responses. The primary objective of **ChatGPT$_{\text{Tester}}$** is to report the root cause of the given buggy method based on its fault location (i.e., buggy hunk). To ensure a highly relevant response, the **User-Defined Prompt** provides crucial details and context to **ChatGPT$_{\text{Tester}}$**, i.e., the definition of the bug reporting subtask and the augmented buggy content required for the bug reporting stage. Additionally, we use special delimiters (e.g., **[Task Definition]**) to explicitly indicate distinct parts of the **User-Defined Prompt**, facilitating **ChatGPT$_{\text{Tester}}$** to better comprehend the relationships between various contexts. Following the provided prompt, **ChatGPT$_{\text{Tester}}$** produces a bug report that describes the bug's nature and its impact on the given buggy method according to the guidance information from the dependence context and the commit message. As shown in the generated bug report, **ChatGPT$_{\text{Tester}}$** correctly identifies the root cause of the buggy hunk (i.e., *It's possible that the setMd5sum method expects a different data type than the md5 byte array, leading to a type error.*). This information will then be assigned to the developer to assist in the resolution of the discovered bug during the subsequent stage.

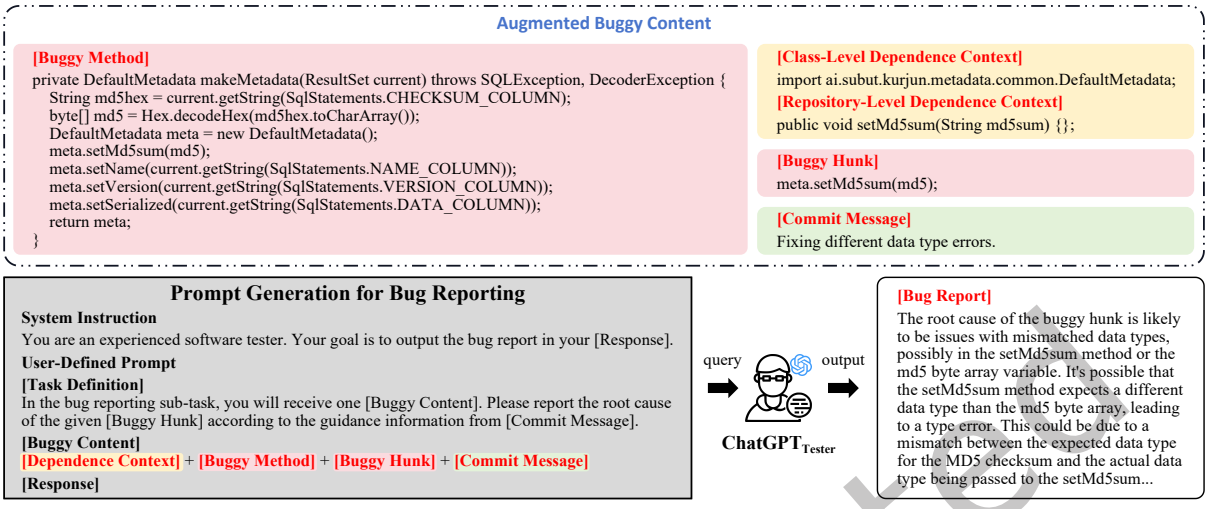

Fig. 4. A Prompting Example of the Tester's Behavior during the Bug Reporting Stage.

2.3 Bug Diagnosis

Upon receiving a bug report from the tester, the developer commences the diagnosis process by utilizing the available information provided in the report. Practically, when assigned a newly discovered bug, the developer first engages in debugging practices. This involves carefully analyzing the source code line by line and documenting their findings in natural language. This self-guided approach enhances the efficiency of bug fixing without the need for external expert guidance [10, 64]. Furthermore, the developer consults historical bug corpora to extract bug-fixing patterns that shed light on the causes and resolutions of similar issues. This mining process aids in acquiring valuable knowledge pertaining to the reasons behind bug occurrences and the corresponding fixes [61, 77]. To mimic the developer's diagnostic behavior, PATCH initiates by employing rubber duck debugging techniques to provide a detailed, line-by-line explanation for the given buggy code snippet. Then, PATCH retrieves relevant bug-fixing demonstrations for pattern summarization by analyzing the paired buggy and fixed methods. These two forms of guidance serve to aid the developer in generating the correct patches.

2.3.1 Code Explanation. Figure 5 illustrates an example prompt alongside the corresponding output obtained during the code explanation phase. The primary objective of **ChatGPT$_{\text{Developer}}$** is to elucidate the given buggy method using the rubber duck debugging technique. This debugging process mimics a common practice employed by human programmers, which involves articulating their code line-by-line using natural language, as if conversing with a rubber duck [76]. Inspired by the principle of rubber ducking, PATCH specializes the persona of **ChatGPT$_{\text{Developer}}$** and its responses in the **System Instruction**. Unlike traditional line-by-line explanations, PATCH prompts **ChatGPT$_{\text{Developer}}$** to explain the buggy method with additional program context, i.e., functional description. According to the **[Task Definition]**, **ChatGPT$_{\text{Developer}}$** initially summarizes the intent of the buggy method, providing a functional description to clarify the program's expected behavior. Subsequently, **ChatGPT$_{\text{Developer}}$** explains the code implementation of the buggy method with the guidance of method summary generated by **ChatGPT$_{\text{Developer}}$**. This operation ensures that **ChatGPT$_{\text{Developer}}$** handles the buggy method within a sufficiently concise context. Consequently, **ChatGPT$_{\text{Developer}}$** is able to provide a detailed code explanation of the

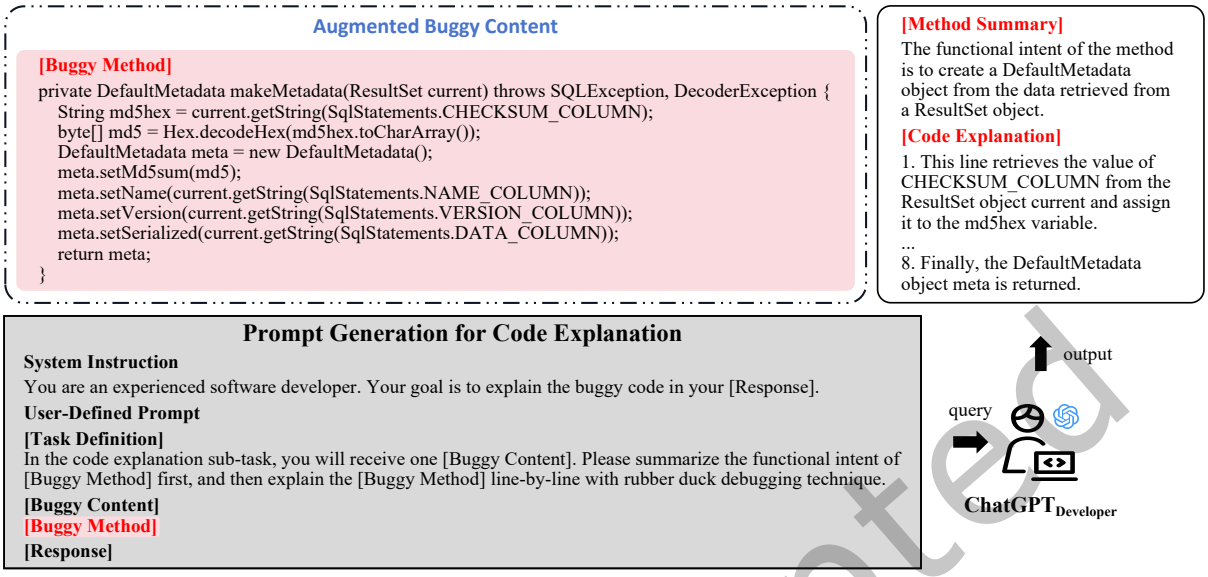

Fig. 5. A Prompting Example of the Developer's Behavior during the Code Explanation Stage.

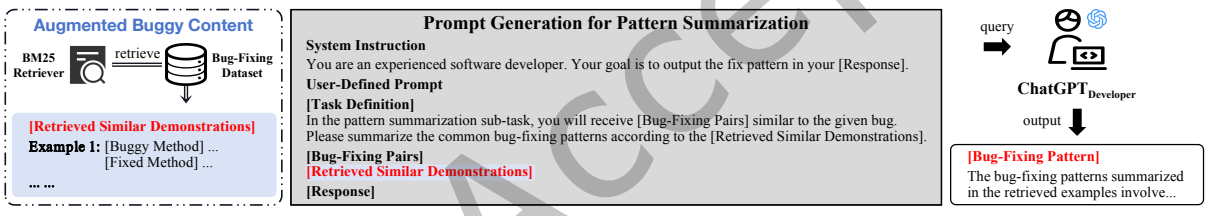

Fig. 6. A Prompting Example of the Developer's Behavior during the Pattern Summarization Stage.

buggy content, thereby enhancing the efficiency of debugging without relying on additional artifacts such as unit tests.

2.3.2 Pattern Summarization. As depicted in Figure 6, the main goal of **ChatGPT$_{\text{Developer}}$** at this phase is to summarize bug-fixing patterns by analyzing similar demonstrations retrieved from the historical code corpus. The initial step involves retrieving similar programs from a collected bug-fixing dataset based on the given buggy content. To achieve this, PATCH employs the sparse keyword-based BM25 score [68] as the retrieval metric. The BM25 score, a probabilistic model widely used in previous studies [38, 85], operates as a bag-of-words retriever, estimating the lexical-level similarity between two sentences. Higher BM25 scores indicate greater similarity between the sentences. Let $\mathcal{D} = \{(B_i, F_i, L_i, C_i)\}_{i=1}^{|\mathcal{D}|}$ represent a bug-fixing dataset containing $|\mathcal{D}|$ 4-tuple bug-fixing pairs, where B_i denotes the i-th buggy method, F_i denotes its paired fixed version, L_i denotes the location of the corresponding bug (i.e., the buggy hunk), and C_i denotes the mined GitHub commit message related to the bug. Specifically, PATCH respectively retrieves the most relevant bug-fixing pair from $\mathcal{D}$ based on a multi-faceted buggy context $C \leftarrow \{b, l, c\}$, where b denotes the given buggy method, l denotes its buggy hunk, and c denotes the commit message. For instance, PATCH selects the top-1 retrieved output from the BM25 retriever based on the computed similarity score $f(B_i, b)$ when considering the context of buggy method,

where f denotes the relevance scoring function. To ensure retrieval performance, we further establish a dynamic threshold criteria: each retrieved bug-fixing pair must have a BM25 score greater than the length of the input query (i.e., the given buggy method, buggy hunk, or commit message) with an added term corresponding to the average length of C across all bug-fixing pairs in $\mathcal{D}$. In other words, the top-1 output is retained only if $f(B_i, b) > len(b) + \frac{\sum_{i=1}^{|\mathcal{D}|} len(B_i)}{|\mathcal{D}|}$. Specifically, PATCH selects up to three demonstrations (including paired buggy and fixed methods) as the retrieved results from $\mathcal{D}$, taking into account the potential for duplicates when evaluating different buggy contexts, or excluding outputs that do not meet the threshold criteria. **ChatGPT$_{\text{Developer}}$** is then able to summarize the common bug-fixing patterns based on retrieved similar demonstrations, providing insights into the root causes and resolutions of the given bug.

2.4 Patch Generation

Once the root cause of a bug has been identified, the developer embarks on the process of creating a patch to resolve the discovered bug. Previous studies have employed LLMs to directly generate candidate patches based on the provided buggy code. However, bug fixing is an intricate task that poses significant challenges in generating the correct patches from scratch. In the context of PATCH, the responsibilities of the developer encompass two main aspects. Firstly, the developer generates an initial candidate patch according to the bug report filed by **ChatGPT$_{\text{Tester}}$** and the guidance obtained during the bug diagnosis stage. Secondly, the developer refines the candidate patch by incorporating review feedback when the candidate patch does not meet the desired fixing goal. This subsection exemplifies the process of initial patch generation.

As illustrated in Figure 7, the objective of **ChatGPT$_{\text{Developer}}$**, as stated in the **System Instruction**, during the initial patch generation stage is to patch the buggy hunk using the feedback information outlined in the **User-Defined Prompt**. Existing LLM-based approaches have encountered challenges regarding the accuracy of patch generation. In light of this, PATCH provides **ChatGPT$_{\text{Developer}}$** with a guided process for bug reporting and diagnosis. This enables **ChatGPT$_{\text{Developer}}$** to generate an initial candidate patch by incorporating information from the bug report, the code explanation, and the bug-fixing patterns as prompt. Subsequently, the generated candidate patch undergoes further verification by the reviewer.

2.5 Patch Verification

Generating the correct patches in a single attempt presents a significant challenge for complex bugs. Therefore, the review process for the generated candidate patches assumes considerable significance as a pivotal activity within software peer review [67, 83]. When presented with a candidate patch generated by the developer, the reviewer needs to assess its effectiveness in resolving the given bug. Algorithm 1 details the interaction process between the developer and the reviewer. The inputs include two ChatGPT agents (i.e., **ChatGPT$_{\text{Developer}}$** and **ChatGPT$_{\text{Reviewer}}$**), the initial candidate patch p_{init} generated by **ChatGPT$_{\text{Developer}}$**, the patch verification prompt **PROMPT$_{\text{PV}}$**, the iteration patch generation prompt **PROMPT$_{\text{PG}_{\text{iter}}}$**, and the hyper-parameter for the maximum iteration number **maxIterNum**. The algorithm produces as output the final candidate patch p, which has been verified by **ChatGPT$_{\text{Reviewer}}$**. The algorithm begins by initializing the current iteration turn **currentIterNum** (Line 1). Subsequently, **ChatGPT$_{\text{Reviewer}}$** generates the review feedback $\mathbf{C}_{\text{review}}$ for p_{init} using **PROMPT$_{\text{PV}}$** (Line 2). If **ChatGPT$_{\text{Reviewer}}$** passes p_{init} (Line 3), the algorithm considers the bug resolved, and p_{init} is regarded as the final output (Line 4). In cases where **ChatGPT$_{\text{Reviewer}}$** does not approve p_{init}, indicating the bug remains unfixed, an interactive process is initiated between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$**. The hyper-parameter **maxIterNum** serves as a termination criterion, capping the maximum number of iterations allowed to generate a candidate patch for fixing the bug (Line 6). During each iteration, **currentIterNum** is incremented (Line 7).

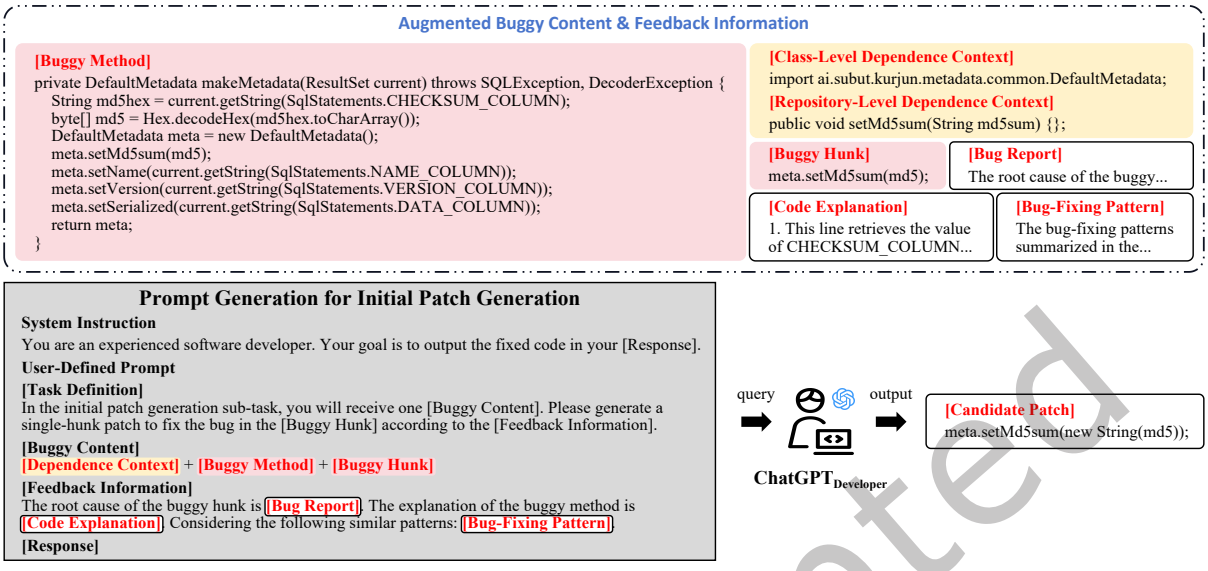

Fig. 7. A Prompting Example of the Developer's Behavior during the Initial Patch Generation Stage.

Algorithm 1: Interaction Process between $\textbf{ChatGPT}_{\textbf{Developer}}$ and $\textbf{ChatGPT}_{\textbf{Reviewer}}$.

Input: $\textbf{ChatGPT}_{\textbf{Developer}}$ (the developer ChatGPT agent); $\textbf{ChatGPT}_{\textbf{Reviewer}}$ (the reviewer ChatGPT agent); p_{init} (the candidate patch generated by $\textbf{ChatGPT}_{\textbf{Developer}}$ during the initial patch generation stage); $\textbf{PROMPT}_{\textbf{PV}}$ (the patch verification prompt); $\textbf{PROMPT}_{\textbf{PG}_{\textbf{iter}}}$ (the iteration patch generation prompt); **maxIterNum** (the maximum iteration number)

Output: p (the candidate patch verified by $\textbf{ChatGPT}_{\textbf{Reviewer}}$)

```
currentIterNum ← 0
C_review ← ChatGPT_Reviewer(PROMPT_PV(p_init))
if C_review is PASS then
    p ← p_init
else
    while currentIterNum < maxIterNum do
        currentIterNum ← currentIterNum + 1
        p_iter ← ChatGPT_Developer(PROMPT_PG_iter(C_review))
        C_review ← ChatGPT_Reviewer(PROMPT_PV(p_iter))
        if C_review is PASS then
            p ← p_iter
            break
    end
end
return p
```

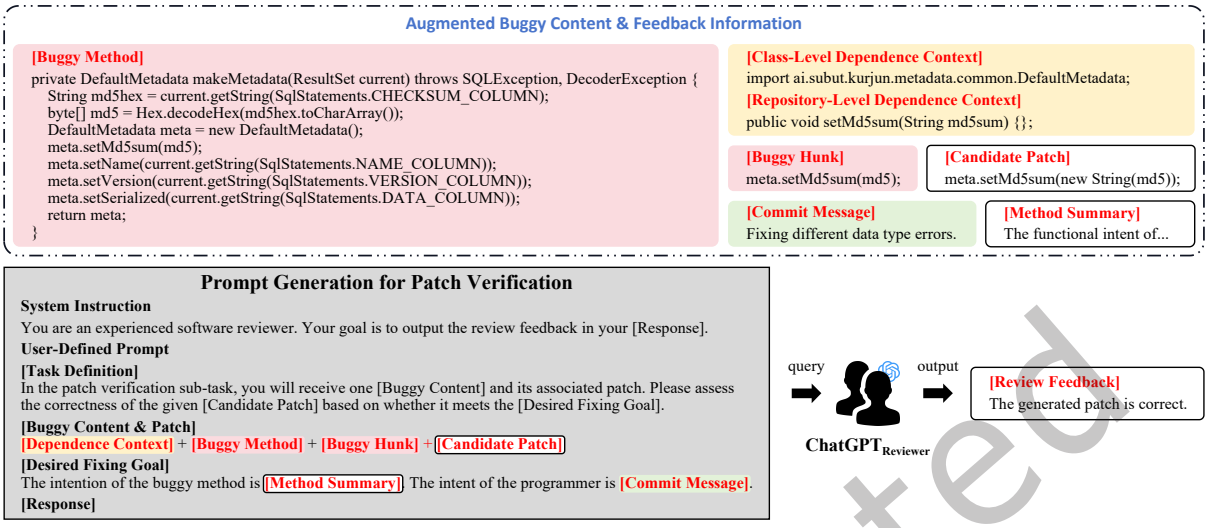

Fig. 8. A Prompting Example of the Reviewer's Behavior during the Patch Verification Stage.

To generate an updated iteration candidate patch p_{iter}, **ChatGPT**$_{\text{Developer}}$ integrates **PROMPT**$_{\text{PG}_{\text{iter}}}$ with $\mathbf{C}_{\text{review}}$ together as input (Line 8). **ChatGPT**$_{\text{Reviewer}}$ then interactively evaluates p_{iter} to assess its suitability as a correct solution to the bug. This interactive process continues until **ChatGPT**$_{\text{Reviewer}}$ determines p_{iter} to be correct (Lines 10–12) or when the maximum number of iterations **maxIterNum** is reached.

Figure 8 delineates the review process undertaken by **ChatGPT**$_{\text{Reviewer}}$. The objective of **ChatGPT**$_{\text{Reviewer}}$ during the patch verification stage is to evaluate the given **[Buggy Content & Patch]**, determining its correctness based on the provided **[Desired Fixing Goal]**. The fixing goal encompasses two primary aspects: the functionality requirement (i.e., the method summary inferred by **ChatGPT**$_{\text{Developer}}$) and the programmer's intent as conveyed in the commit message. As depicted in Figure 8, the review feedback confirms that the candidate patch satisfies both of the aforementioned goals, thereby deeming it correct.

3 EXPERIMENTAL SETUP

3.1 Research Questions

To assess the effectiveness of PATCH, we aim at answering the following three research questions (RQs):

- **RQ1: How does** PATCH **perform in bug fixing when compared to state-of-the-art LLMs?** The objective of this RQ is to evaluate the superior effectiveness of PATCH in comparison to state-of-the-art LLM baselines within the context of bug fixing. To achieve this, we conduct a comprehensive comparison of PATCH against 13 LLMs using an augmented bug-fixing benchmark.
- **RQ2: How does each component impact the performance of** PATCH**?** PATCH introduces two essential mechanisms: the augmented buggy content and the behavior-simulation framework. The proposed framework further comprises three agents: **ChatGPT**$_{\text{Tester}}$ is responsible for bug reporting, **ChatGPT**$_{\text{Developer}}$ handles bug diagnosis and patch generation, and **ChatGPT**$_{\text{Reviewer}}$ is in charge of patch verification. In this RQ, we aim to analyze the contributions of each designed component by conducting the ablation study.

Table 1. Statistics of the BFP Benchmark in Our Experiments.

	# of Instances	Buggy Method Statistics			Buggy Class Statistics			
		LoC Range	CC Range	Complexity	LoC Range	Method Count Range	CC Range	Complexity
Training Set	28226	[1, 85]	[1, 56]	17.2%	[5, 63808]	[1, 2218]	[1, 49]	11.7%
Testing Set	3112	[1, 57]	[1, 19]	14.4%	[5, 13765]	[1, 449]	[1, 27]	9.8%

- **RQ3: What is the generalizability of** PATCH **to additional benchmarks and different LLMs?** This RQ first assesses the generalizability of PATCH by applying it to four common benchmarks in the field of automated program repair (APR). Furthermore, we extend PATCH to five open-source LLMs that support interactive dialogues to enhance evaluation diversity.

3.2 Benchmark

In this paper, we utilize the widely recognized BFP benchmark [81] as our original data source, encompassing a substantial collection of paired bug-fixing instances extracted from real-world GitHub repositories. Each instance within the BFP benchmark consists of both the buggy and the fixed Java methods. To achieve a comprehensive understanding of the provided buggy method, we first extract additional contextual dependencies associated with the buggy method as described in Section 2.1. Notably, we exclude instances that cannot be successfully parsed by Spoon, as well as those exceeding 300 tokens in length (in consideration of the maximum token limits of the LLMs). Furthermore, we collect bug-fixing commits from GitHub linked to instances within the BFP benchmark and apply a three-stage filtering mechanism to ensure commit quality. First, we filter out commits shorter than 5 tokens, excluding low-informative messages such as "done" or "bug fixing". The remaining commits undergo an automated annotation process powered by the advanced LLM GPT-4 [24]. Specifically, we provide GPT-4 with examples of irrelevant or low-quality commits to facilitate few-shot in-context learning, enabling it to accurately label the commits as either *good* or *bad*, and to generate a confidence score for each annotation. Finally, we employ random sampling [74] on the *good* commits with a confidence score greater than 0.90 for manual inspection. In particular, the first author and three experienced master students manually reviewed 380 commit samples (confidence level: 95%, margin of error: 5%). The inspection results show that GPT-4's annotation accuracy exceeds 90%, with 36 commits removed due to the inconsistencies between human and GPT-4 annotations, thereby demonstrating the effectiveness and reliability of the data annotations generated by GPT-4.

Consequently, each instance within the BFP benchmark is augmented with additional information (i.e., the dependence context and the commit message). Next, we split the augmented BFP benchmark into training and testing sets by maintaining a 9:1 ratio. To prevent any data leakage, instances originating from the same GitHub repository are not allowed to appear in different sets (e.g., one in the training set and the other in the testing set). As illustrated in the second column of Table 1, we collect 28226 bug-fixing instances in the training set and 3112 in the testing set. The **LoC Range** column specifies the range of lines of code (LoC) for both the buggy method and its corresponding buggy class. The **CC Range** column represents the range of cyclomatic complexity (CC) [50], while the **Complexity** column indicates the proportion of complex methods with a CC greater than 5 [94]. The **Method Count Range** shows the range of method counts within the corresponding buggy class file. We evaluate the bug-fixing performance of PATCH and selected baselines on the testing set. Additionally, we perform the pattern summarization process, as described in Section 2.3.2, by retrieving similar instances from the training set. In real-world development scenarios, such a code corpus for retrieval can be constructed using open-source code repositories or historical bug-fixing data collected from developers' private projects to facilitate the pattern summarization process.

3.3 Baselines

This paper centers on addressing the bug-fixing task using LLMs. Therefore, we compare PATCH against 13 state-of-the-art LLM baselines as listed in Table 2. The selection criteria are as follows:

- **Popularity.** Initially, we consider the list of popular models hosted on the Hugging Face platform, which is an open-source resource for hosting and deploying large models. From this repository, we select LLMs pre-trained or fine-tuned on a large amount of code corpus and specifically engineered to address code-related tasks. Furthermore, we include closed-source LLMs (i.e., the GPT family of models) due to their demonstrated impressive performance across a wide range of tasks.
- **Diversity.** We select LLMs with varying sizes of parameters and from different organizations.
- **Accessibility.** The selected LLMs are publicly accessible either through checkpoints (e.g., InCoder) or non-free APIs (e.g., GPT-4). Thus, closed-source models such as AlphaCode [40] are excluded from our evaluation.

Table 2. Overview of the Selected LLM Baselines.

Model	# of Parameters	Organization	Pre-Training / Fine-Tuning Code Corpus
CodeGPT [46]	124M	Microsoft	CodeSearchNet [26]
DeepSeek-Coder [23]	1.3B	DeepSeek	GitHub
CodeGen2 [55]	3.7B	Salesforce	Stack [32]
CodeGeeX2 [110]	6B	THU	Pile [18] & BigQuery & GitHub
InCoder [17]	6.7B	Facebook	StackOverFlow & GitHub & GitLab
Mistral [27]	7B	Mistral_AI	Hugging Face Repository
CodeLLaMA [69]	7B & 13B	Meta	BigQuery
StarCoder [39]	15B	BigCode	Stack [32]
GPT-NeoX [4]	20B	EleutherAI	Pile [18]
Codex [9]	175B	OpenAI	-
ChatGPT [59]	-	OpenAI	-
GPT-4 [60]	-	OpenAI	-

3.4 Metrics

This paper employs the following two evaluation metrics to compare the performance of PATCH with the LLM baselines.

- **Fix@k.** This paper first utilizes the Fix@k metric [111] to evaluate the bug-fixing performance of LLMs on the testing set. This metric measures the percentage of successfully resolved bugs within the entire testing set when k candidate patches are generated for each given bug. In other words, given a Java method with a single-hunk bug, each corresponding LLM is permitted to generate k candidate patches. The bug is considered resolved if any of the LLM-generated patches exactly match the human-written ground truth. For Fix@k, higher values indicate better performance. In our experiments, we evaluate Fix@k with k set to 1, 3, and 5, since most programmers are usually willing to review as few patches as possible in real-world development scenarios [57].
- **Levenshtein Distance.** The Levenshtein distance metric calculates the absolute token-based edit distance between the LLM-generated candidate patch and the human-written ground truth, representing the minimum number of operations required to transform the candidate patch into the ground truth. A lower

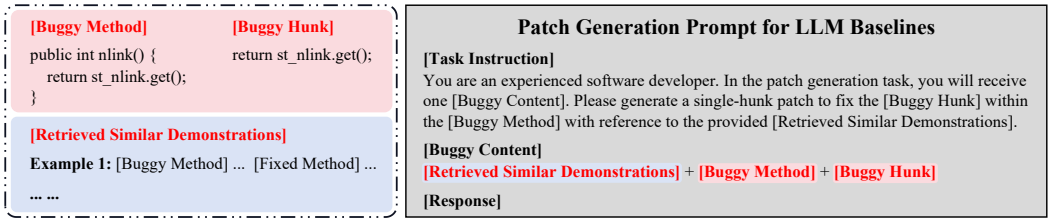

Fig. 9. A Prompting Example for the LLM Baselines.

Levenshtein distance signifies a closer correspondence between the candidate patch and the ground truth. This metric serves as a complementary measure that provides valuable insights into the usefulness of incorrect LLM-generated candidate patches for programmers.

3.5 Implementation

We implement the main logic of PATCH in Python by invoking ChatGPT through its API. Specifically, we employ the **gpt-3.5-turbo-0125** version of the ChatGPT family due to its enhanced performance and cost-efficiency. Following the best-practice guidelines from Shieh et al. [72], we design prompts and manually examine several alternative approaches with selected buggy code using the Web-version of ChatGPT. The response configurations of each ChatGPT agent are detailed as follows:

- **ChatGPT$_{\text{Tester}}$**. The maximum length of the bug report generated during the bug reporting stage is restricted to 200 tokens.
- **ChatGPT$_{\text{Developer}}$**. The maximum length for the code explanation and the bug-fixing pattern generated during the bug diagnosis stage is limited to 500 tokens. During the patch generation stage, the maximum length of the candidate patch is capped at 150 tokens.
- **ChatGPT$_{\text{Reviewer}}$**. The maximum length of the review feedback generated during the patch verification stage is constrained to 200 tokens.

In our experiments, when $k = 1$, we utilize greedy decoding for each ChatGPT agent. Specifically, PATCH produces the top-1 chat completion choice for each input query with a sampling temperature of 0. For $k > 1$, a sampling temperature of 0.8 is used to generate multiple responses. We limit the maximum number of interaction turns between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$** to three, as recommended by Chen et al. [10]. Furthermore, we conduct experiments under a zero-shot setting, where task examples are not provided, aiming to demonstrate the superiority of PATCH.

4 RESULTS AND ANALYSIS

4.1 Answering RQ1

To answer this question, we conduct a comprehensive comparison of PATCH with 13 state-of-the-art baselines on the augmented BFP benchmark. For each baseline, we either reuse the official checkpoint or access the inference API for implementation. As illustrated in Figure 9, we prompt the LLM baselines by incorporating both the buggy method and several task examples, following the approach established in prior research. To ensure a fair comparison, we utilize the same bug-fixing demonstrations retrieved by PATCH during the pattern summarization phase to conduct these few-shot setting experiments. Additionally, we employ the same sampling hyper-parameters as PATCH.

Table 3. Comparison of PATCH against the LLM Baselines.

Model	# of Parameters	Evaluation Metric Result				Paired t-test Result	
		Fix@1 (%) ↑	Fix@3 (%) ↑	Fix@5 (%) ↑	Levenshtein Distance ↓	T-statistic	p-value
CodeGPT	124M	2.76	5.46	6.65	73.69	53.90	< 0.001
DeepSeek-Coder	1.3B	8.42	11.25	14.14	48.34	316.67	< 0.001
CodeGen2	3.7B	4.31	7.23	10.15	70.13	469.32	< 0.001
CodeGeeX2	6B	10.93	15.30	19.41	40.18	28.52	< 0.001
InCoder	6.7B	5.24	8.03	11.60	64.20	117.10	< 0.001
Mistral	7B	9.38	14.11	16.68	43.45	45.72	< 0.001
CodeLLaMA	7B	9.45	14.46	17.96	43.35	28.98	< 0.001
CodeLLaMA	13B	9.80	14.62	17.45	41.20	39.15	< 0.001
StarCoder	15B	13.50	17.58	20.89	34.27	43.42	< 0.001
GPT-NeoX	20B	8.93	14.56	16.16	57.37	32.57	< 0.001
Codex	175B	11.05	-	-	36.33	-	-
ChatGPT	-	14.62	18.99	22.27	33.28	43.21	< 0.001
GPT-4	-	19.96	24.42	26.00	26.07	32.08	< 0.001
PATCH	-	**33.97** (14.01 ↑)	**37.08** (12.66 ↑)	**39.81** (13.81 ↑)	**21.44** (4.63 ↓)		

4.1.1 Experimental Metric Evaluation. Table 3 presents the bug-fixing performance of different models in terms of the **Fix@k** ($k \in [1, 3, 5]$) and **Levenshtein Distance** (when $k = 1$) metrics, with the best result for each metric highlighted in bold. The numbers in red denote PATCH's improvement percentage points compared to the best baseline. Our experiments reveal the following three-fold key findings:

(1) PATCH **demonstrates superior performance compared to all the baselines on the augmented BFP benchmark.** Specifically, PATCH outperforms the best baseline GPT-4 by 14.01 percentage points in terms of **Fix@1**. These improvements underscore the effectiveness of PATCH in bug fixing, particularly considering that **Fix@1** is a stringent metric. As for the **Levenshtein Distance** metric, PATCH achieves a score of 21.44 on the augmented BFP benchmark, showcasing an improvement of 4.63 percentage points over GPT-4. We also perform a statistical comparison of performance in terms of **Fix@k** ($k \in [1, 3, 5]$) between PATCH and each baseline LLM using the paired t-test [13]. The paired t-test evaluates the null hypothesis, which posits that the difference between PATCH and each baseline is not statistically significant. If the reported p-value is less than the significance level of 0.05, the null hypothesis is rejected, indicating that the observed disparity between PATCH and each baseline is statistically significant and not due to random chance. Additionally, we compute the T-statistic to measure the effect size; a larger T-statistic indicates a more significant performance difference between PATCH and each baseline. The paired t-test results in Table 3 show that PATCH surpasses all baselines in bug-fixing performance, with the difference being statistically significant (p-value < 0.001).

(2) **Simulating programmer behavior proves to be advantageous for bug fixing.** Rather than altering the parameters of ChatGPT, PATCH explicitly prompts ChatGPT to mimic the behavior of programmers engaged in the bug management process. The substantial improvements observed over the base ChatGPT model suggest that PATCH effectively enhances ChatGPT's bug-fixing capabilities by endowing it with collaborative problem-solving skills. Furthermore, aligning ChatGPT with the interactive decision-making processes of programmers boosts its performance across various aspects, including the comprehension of intent.

(3) **Enhancing the bug-fixing performance of LLMs relies on two factors: increasing the number of parameters and designing well-crafted prompts.** Generally, an increase in parameters often leads to improved performance, as demonstrated by StarCoder-15B surpassing CodeGeeX2-6B, while CodeLLaMA-13B performs better than CodeLLaMA-7B. Notably, LLMs struggle to achieve satisfactory performance

Table 4. Comparison of the Fixed Bug Types between PATCH and the LLM Baselines.

Model	Simple Delete		Simple Insert		Simple Replace		Mixed	
	149 bugs		272 bugs		1157 bugs		1534 bugs	
	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$
CodeGPT	15 (8.66%)	22 (14.77%)	5 (1.84%)	28 (10.29%)	36 (3.11%)	90 (7.78%)	30 (1.96%)	67 (4.37%)
DeepSeek-Coder	28 (18.79%)	42 (28.19%)	14 (5.15%)	51 (18.75%)	112 (9.68%)	156 (13.48%)	108 (7.04%)	191 (12.45%)
CodeGen2	21 (14.09%)	38 (25.50%)	9 (3.31%)	38 (13.97%)	46 (3.98%)	100 (8.64%)	58 (3.78%)	140 (9.13%)
CodeGeeX2	23 (15.44%)	50 (33.56%)	51 (18.75%)	61 (22.43%)	129 (11.15%)	226 (19.53%)	137 (8.93%)	267 (17.41%)
InCoder	21 (14.09%)	44 (29.53%)	14 (5.15%)	62 (22.79%)	59 (5.10%)	97 (8.38%)	69 (4.50%)	158 (10.30%)
Mistral	29 (19.46%)	43 (28.86%)	16 (5.88%)	33 (12.13%)	124 (10.72%)	222 (19.19%)	123 (8.02%)	221 (14.41%)
CodeLLaMA-7B	29 (19.46%)	49 (32.89%)	13 (4.78%)	26 (9.56%)	136 (11.75%)	239 (20.66%)	116 (7.56%)	245 (15.97%)
CodeLLaMA-13B	28 (18.79%)	46 (30.87%)	14 (5.15%)	26 (9.56%)	137 (11.84%)	229 (19.79%)	126 (8.21%)	242 (15.78%)
StarCoder	27 (18.12%)	55 (36.91%)	52 (19.12%)	62 (22.79%)	164 (14.17%)	252 (21.78%)	177 (11.54%)	281 (18.32%)
GPT-NeoX	28 (18.79%)	59 (39.60%)	12 (4.41%)	59 (21.69%)	127 (10.98%)	146 (12.62%)	111 (7.24%)	239 (15.58%)
Codex	19 (12.75%)	-	6 (2.21%)	-	175 (15.13%)	-	144 (9.39%)	-
ChatGPT	27 (18.12%)	49 (32.89%)	15 (5.51%)	59 (21.69%)	218 (18.84%)	290 (25.06%)	195 (12.71%)	295 (19.23%)
GPT-4	38 (25.50%)	48 (32.21%)	38 (13.97%)	42 (15.44%)	277 (23.94%)	370 (31.98%)	268 (17.47%)	349 (22.75%)
PATCH	**60 (40.27%)**	**77 (51.68%)**	**96 (35.29%)**	**115 (42.28%)**	**448 (38.72%)**	**497 (42.96%)**	**453 (29.53%)**	**550 (35.85%)**

	CodeGPT	DeepSeek-Coder	CodeGen2	CodeGeeX2	InCoder	Mistral	CodeLLaMA-7B	CodeLLaMA-13B	StarCoder	GPT-NeoX	CodeX	ChatGPT	GPT-4	PATCH
CodeGPT	1	78%	59%	54%	79%	54%	70%	73%	71%	67%	46%	73%	75%	78%
DeepSeek-Coder	77%	6	38%	59%	46%	62%	68%	66%	69%	70%	42%	39%	51%	73%
CodeGen2	63%	74%	1	60%	73%	79%	78%	70%	68%	82%	31%	30%	40%	68%
CodeGeeX2	79%	45%	24%	25	28%	45%	48%	48%	74%	73%	33%	34%	46%	73%
InCoder	74%	74%	60%	58%	2	71%	77%	73%	68%	61%	34%	29%	44%	71%
Mistral	72%	56%	36%	53%	40%	5	64%	66%	63%	81%	41%	45%	57%	76%
CodeLLaMA-7B	84%	61%	35%	56%	43%	64%	3	66%	65%	67%	47%	41%	54%	73%
CodeLLaMA-13B	79%	57%	31%	54%	39%	63%	63%	4	64%	61%	46%	44%	57%	76%
StarCoder	76%	43%	22%	60%	26%	44%	45%	46%	27	41%	35%	36%	47%	72%
GPT-NeoX	41%	44%	43%	35%	47%	43%	35%	40%	37%	16	31%	45%	52%	68%
CodeX	13%	32%	12%	33%	16%	35%	40%	40%	42%	14%	26	54%	67%	76%
ChatGPT	18%	22%	9%	26%	11%	29%	27%	29%	33%	18%	41%	20	68%	84%
GPT-4	14%	22%	9%	25%	11%	27%	25%	28%	32%	16%	37%	50%	56	79%
PATCH	10%	18%	9%	24%	11%	21%	20%	22%	29%	14%	25%	36%	47%	283

Fig. 10. The Overlapping Rates and Unique Patch Numbers of the Evaluated Models.

due to their inability to comprehend how to solve complex problems. However, this limitation can be effectively addressed by incorporating essential information into the query prompts. By adopting this approach, PATCH significantly outperforms the LLM baselines. This finding validates our motivation to decompose the bug-fixing task into subtasks using well-designed prompts, as it substantially enhances the performance of LLMs in this context.

4.1.2 Bug Types Evaluation. We begin by utilizing the Gumtree algorithm [15] to compute the difference between two ASTs generated by Spoon [65]. We classify the bugs into four categories based on the necessary edit operations to transform a bug hunk into its corrected version: **Simple Delete**, **Simple Insert**, **Simple Replace**, and **Mixed**. For instance, **Simple Delete** indicates that a bug can be fixed by removing certain tokens from a specific position. Table 4 presents the comparison results (when $k = 1$ and $k = 5$), with each row representing a model and the corresponding number of correct patches generated for each bug type on the augmented BFP benchmark. The best results are highlighted in bold. The statistical findings in Table 4 reveal that LLMs excel at fixing bugs that require only deletion edit operations but encounter difficulties with more intricate ones. This observation is reasonable since insert and replace edit operations demand that the model search for additional tokens to fix the given bug. Overall, PATCH outperforms all baselines in resolving both simple and complex bugs.

Table 5. Ablation Study for PATCH.

Model	Component						Fix@1 (%) ↑	Fix@3 (%) ↑	Fix@5 (%) ↑
	Augmented Buggy Content		$\text{ChatGPT}_{\text{Tester}}$	$\text{ChatGPT}_{\text{Developer}}$		$\text{ChatGPT}_{\text{Reviewer}}$			
	Dependence Context	Commit Message	Bug Report	Code Explanation	Bug-Fixing Pattern	Review Feedback			
ChatGPT	✖	✖					14.62	18.99	22.27
	✔	✖					16.87	21.50	24.39
	✖	✔					18.86	24.33	27.28
			✔	✖	✖	✖	17.48	22.59	26.86
			✖	✔	✖	✖	17.26	22.46	26.09
			✖	✖	✔	✖	16.93	21.79	24.55
			✖	✔	✔	✖	17.89	23.01	27.37
			✔	✔	✖	✖	22.14	26.22	29.53
			✔	✖	✔	✖	20.31	25.16	28.92
			✔	✔	✔	✖	24.23	27.79	30.27
			✔	✔	✔	✔	**33.97**	**37.08**	**39.81**
GPT-4	✖	✖					19.96	24.42	26.00
	✔	✖					22.85	27.47	30.01
	✖	✔					25.96	29.37	31.62

4.1.3 Overlapping Phenomenon Evaluation. As illustrated in Figure 10, each row represents the overlapping ratio (when $k = 1$) of correct patches generated by one model with those generated by other models, while the diagonal indicates the number of unique correct patches generated by each model on BFP. The color intensity of each rectangle increases with the overlapping rate, providing a visual cue for easier interpretation. For example, PATCH generates correct patches that overlap with 29% of those generated by StarCoder (row 14, column 9). The results in Figure 10 indicate that models with superior performance tend to exhibit higher overlapping patching rates with other models. The evaluation results in Table 3 show that that PATCH, GPT-4, and ChatGPT are the top three models. The overlapping rates of other models with these three are notably higher, likely due to the adoption of similar network architectures and inference paradigms among DL-based approaches. Furthermore, PATCH uniquely fixes 283 bugs (row 14, column 14), a significantly higher number than other LLM baselines.

Answer to RQ1: In conclusion, PATCH demonstrates significant superiority over the LLM baselines across the evaluation metrics, underscoring the effectiveness of PATCH in the bug-fixing task. Additionally, our findings reveal that PATCH is capable of generating a higher number of unique and correct patches compared to the LLM baselines.

4.2 Answering RQ2

To answer this question, we perform a series of ablation experiments to assess the impact of different components within the PATCH design. To ensure the fairness of comparisons, we maintained consistency in the implementation settings, adhering to the parameters detailed in Section 3.5.

4.2.1 Ablation Study. Table 5 presents the evaluation results, where each row represents one ablation model. The symbols ✔ and ✖ respectively indicate the addition and removal of the corresponding component. The best **Fix@k** ($k \in [1, 3, 5]$) results are highlighted in bold. To demonstrate the contribution of each component to bug-fixing performance, we start with evaluating the base model-ChatGPT-using only the buggy method as the input prompt to generate candidate patches. Upon integrating the `Dependence Context` and `Commit Message` into ChatGPT, we observe improvements of 2.25% and 4.24% in the **Fix@1** metric, respectively, compared to the base ChatGPT model. Further enhancements are observed when incorporating the `Bug Report`, `Code Explanation`, and `Bug-Fixing Pattern` components, which lead to additional improvements of 2.86%, 2.64%, and 2.31% in **Fix@1**, respectively, relative to the base ChatGPT model. Notably, the addition of the $\textbf{ChatGPT}_{\textbf{Tester}}$ component allows the ChatGPT agent to perform bug reporting, identify the root cause of the buggy code, and generate candidate patches based on the information available in the bug report. This improvement underscores

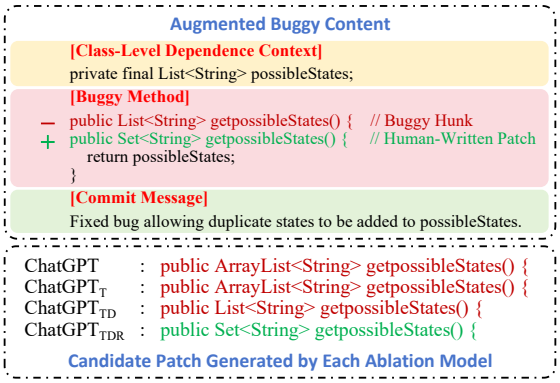

Fig. 11. An Example from the Augmented BFP Benchmark only Fixed by PATCH.

the significance of providing essential bug-related information in the bug-fixing process. The inclusion of the **ChatGPT$_{\text{Developer}}$** component further enables the ChatGPT agent to generate candidate patches informed by the bug diagnosis stage, demonstrating the efficacy of the self-guided diagnosis process in enhancing bug-fixing efficiency. Furthermore, the integration of the **ChatGPT$_{\text{Reviewer}}$** component equips ChatGPT with interactive abilities to refine candidate patches based on `Review Feedback`, leading to continuous improvements in bug-fixing performance, with an additional enhancement of 9.74% in **Fix@1**. This result emphasizes the importance of interaction and collaboration during the resolution of complex software bugs. In conclusion, all components are crucial for optimizing the bug-fixing performance of PATCH. In the subsequent set of experiments, we further evaluate the impact of augmenting GPT-4, the most advanced LLM, with additional buggy content. Our results indicate that incorporating the `Dependence Context` and `Commit Message` components enhances GPT-4's bug-fixing performance by 2.89% and 6.00%, respectively, in terms of the **Fix@1** metric. This outcome indicates the significant positive effect of incorporating `Augmented Buggy Content`, which is on par with the performance observed in ChatGPT when utilizing the **ChatGPT$_{\text{Tester}}$** and **ChatGPT$_{\text{Developer}}$** components. Additionally, this result also highlights the critical role of the proposed stage-wise framework in enhancing bug-fixing performance. Figure 11 depicts an example of bug-fixing from the augmented BFP benchmark. Among the different ablation models (where T is short for **ChatGPT$_{\text{Tester}}$**, D is short for **ChatGPT$_{\text{Developer}}$** and R is short for **ChatGPT$_{\text{Reviewer}}$**), only PATCH (i.e., **ChatGPT$_{\text{TDR}}$**) manages to successfully patch the bug. The provided commit message suggests that the root cause of the bug lies in a lack of control over the internal list, thus necessitating a patch that addresses the issue of duplicate states being added. As shown in Figure 11, it is evident that the ablation models without the **ChatGPT$_{\text{Reviewer}}$** component produce incorrect patches that either do not include any changes to the buggy hunk or fail to resolve the duplicate issue. However, with the guidance provided by the commit message as the fixing goal during the patch verification stage, PATCH is able to generate a correct candidate patch that matches the human-written ground truth.

4.2.2 The Impact of Different Commit Message Types. We further investigate the influence of commit message quality on bug-fixing performance. Following the approach outlined in existing research [79], we categorize commit messages into four types based on the presence of ***Why*** (i.e., justification for the commit change) and ***What*** (i.e., summary of the commit change) information. Table 6 provides a comparative analysis of two treatments: with and without the commit message (abbreviated as `CM`), across the four commit message types (*Why and What*, *No Why*, *No What*, and *Neither Why nor What*) for ChatGPT and GPT-4. The comparison examines the number of correct patches generated for each commit message type at $k = 1$, $k = 3$, and $k = 5$. Our

Table 6. The Effect of Different Commit Message Types on Bug Fixing.

Model	Why and What			No Why			No What			Neither Why nor What		
	1043 bugs			1541 bugs			314 bugs			214 bugs		
	$k=1$	$k=3$	$k=5$	$k=1$	$k=3$	$k=5$	$k=1$	$k=3$	$k=5$	$k=1$	$k=3$	$k=5$
ChatGPT	119	134	188	262	324	388	20	33	40	54	64	77
ChatGPT w/ CM	189 (+70)	246 (+112)	277 (+89)	316 (+54)	405 (+81)	449 (+61)	28 (+8)	41 (+8)	45 (+5)	54 (+0)	65 (+1)	78 (+1)
GPT-4	177	228	243	350	418	445	32	38	39	62	76	82
GPT-4 w/ CM	265 (+88)	292 (+64)	315 (+72)	428 (+78)	483 (+65)	521 (+76)	45 (+13)	55 (+17)	58 (+19)	70 (+8)	84 (+8)	90 (+8)

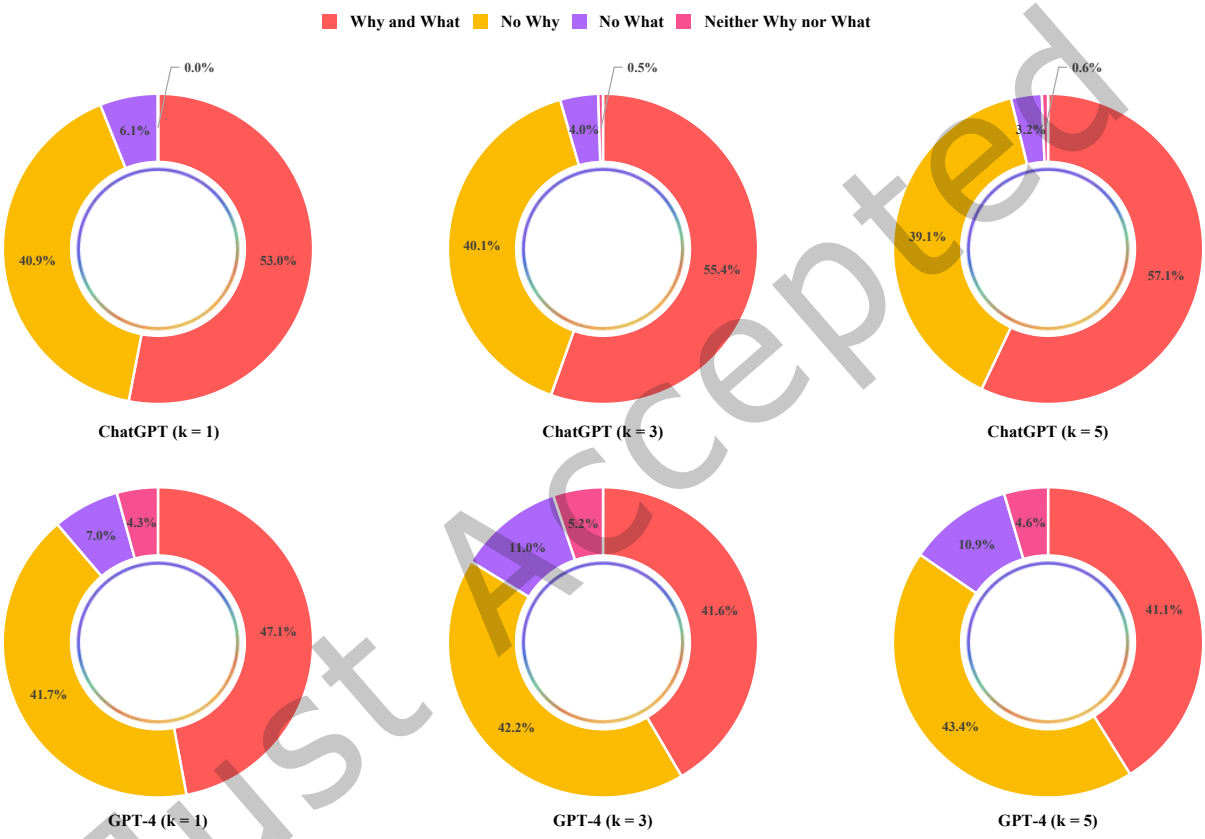

Fig. 12. The Contribution of Different Commit Message Types to Improvements in Bug-Fixing Performance.

findings indicate that all four types have a non-negative effect on performance improvements, as they contribute to generating more correct patches. Furthermore, Figure 12 illustrates the contributions of different commit message types to bug-fixing performance improvements. We observe that informative commit message types (*Why and What*, *No Why*, and *No What*) contribute significantly more to performance gains compared to the non-informative type (*Neither Why nor What*). This result is expected, as *Neither Why nor What* type commit messages lack the necessary context, preventing the LLMs from reasoning accurately to fix the bug. On average, the three informative types account for over 95% of the total contributions. Notably, for commit messages that

contain only high-level information (i.e., *No What*), the more advanced LLM GPT-4 achieves greater performance improvements than ChatGPT, highlighting its superior reasoning capabilities. Since the *No What* type commit messages lack specific details about the steps taken to fix the bug, it forces the LLMs to rely on the justification for the commit change to infer how to resolve the bug, increasing the difficulty required for solving the bug-fixing task.

4.2.3 The Impact of Interaction Turns. In order to assess the impact of the interaction between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$**, we control the number of interaction turns and illustrate the **Fix@1** results in Figure 13. When the number of interaction turns is set to zero, it signifies a complete absence of interaction between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$**. Consequently, the candidate patch generated by **ChatGPT$_{\text{Developer}}$** receives no feedback message from **ChatGPT$_{\text{Reviewer}}$**, leading to an outcome identical to that of the ablation model **ChatGPT$_{\text{TD}}$**. It is noteworthy that the most significant improvement arises from the first interaction turn. Specifically, a single interaction turn with the **ChatGPT$_{\text{Reviewer}}$** component leads to an approximately 4.4% enhancement in terms of **Fix@1** over the **ChatGPT$_{\text{TD}}$** model. As the number of interaction turns continues to increase beyond the initial round, the magnitude of improvements tends to diminish. Nonetheless, our observations indicate a consistent enhancement, suggesting that additional interactions still contribute to the capability to resolve more complex bugs. Figure 14 illustrates an exemplary interactive process between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$** during the stages of patch generation and patch verification. In this scenario, **ChatGPT$_{\text{Reviewer}}$** determines that the candidate patch generated by **ChatGPT$_{\text{Developer}}$** is incorrect based on the programmer's intent as summarized in the commit message. Consequently, **ChatGPT$_{\text{Developer}}$** is required to generate a new patch while taking into account the review feedback. As shown in Figure 14, PATCH successfully generates the correct patch through this collaborative effort.

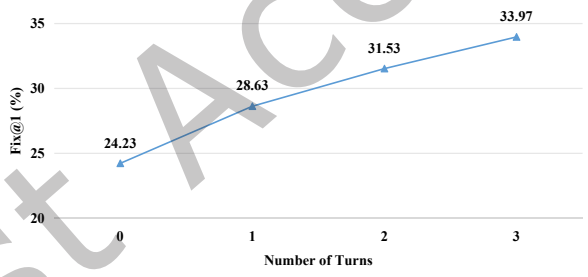

Fig. 13. The Effect of Interaction Turns on Bug Fixing.

Answer to RQ2: To sum up, all components of PATCH significantly improve the fixing performance. Regarding **Fix@1**, the addition of **ChatGPT$_{\text{Tester}}$** enhances ChatGPT by 2.86%. Furthermore, the introduction of **ChatGPT$_{\text{Developer}}$** leads to additional improvements of 6.75%, while the incorporation of **ChatGPT$_{\text{Reviewer}}$** results in continuous enhancements of 9.74%.

4.3 Answering RQ3

To answer this question, we conduct an extensive evaluation of the generalizability of PATCH across four widely used APR benchmarks and five popular open-source LLM baselines. This rigorous analysis offers a comprehensive assessment of the effectiveness and robustness of PATCH.

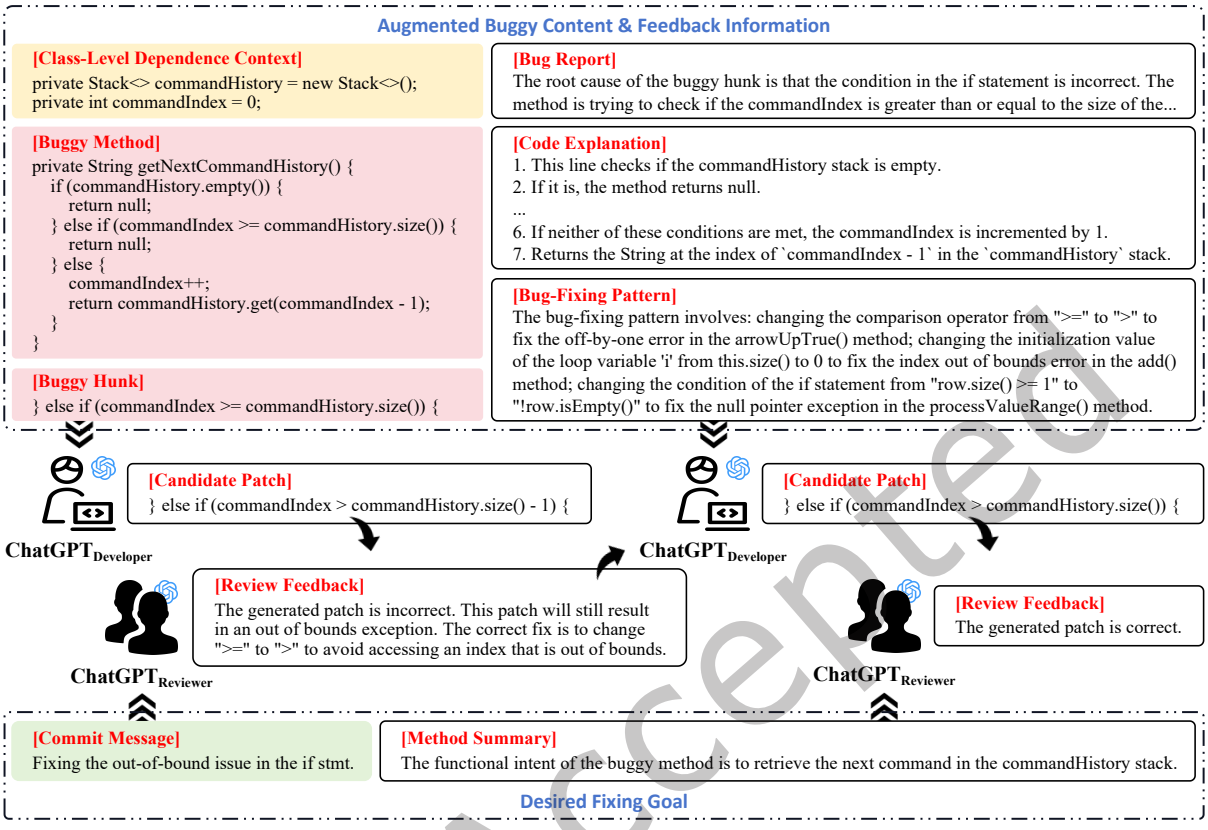

Fig. 14. An Example of the Interactive Process between **ChatGPT$_{\text{Reviewer}}$** and **ChatGPT$_{\text{Developer}}$**.

4.3.1 Generalizability Evaluation on APR Benchmarks. We select single-hunk bugs from four additional APR benchmarks for performance evaluation: Bugs.jar [70] (1000 bugs), Bears [48] (119 bugs), Defects4J [31] (313 bugs), and QuixBugs [42] (37 bugs). Specifically, we compare PATCH to 17 APR baselines, which include template-based, neural-based, pre-trained code language model-based, and LLM-based approaches. For template-based APR, we use the state-of-the-art approach TBar [44] with perfect fault localization. For neural-based APR, we select seven recently published approaches: CoCoNut [47], SequenceR [11], Recoder [114], CURE [30], RewardRepair [98], SelfAPR [96], and KNOD [29]. We further include four pre-trained code language models: CodeBERT [16], GraphCodeBERT [22], CodeT5 [84], and UniXcoder [21]. Given that the BFP benchmark lacks test suites, we evaluate PATCH by comparing it to five recent LLM-based APR approaches in this RQ: AlphaRepair [90], Repilot [87], ChatRepair [91], ThinkRepair [100], and RepairAgent [5]. These LLM-based approaches are evaluated on the Defects4J and QuixBugs benchmarks, both of which include test suites.

Experimental Settings. For Bugs.jar and Bears, we use the *exact match* metric to evaluate the correctness of each generated candidate patch. This metric is preferable for these benchmarks due to their lack of test suites, which helps avoid human bias and minimizes the manual effort required for patch validation [97]. In these experiments, PATCH generates the top-1 candidate patch for each bug, whereas the selected baselines typically generate a larger number of candidates (more than 100), as demonstrated in related studies [111, 113]. This paper reports the number of *exact match* patches within the top-10 candidates generated by each baseline

Table 7. Comparison of PATCH on Four APR Benchmarks against 17 Baselines.

Model	# of *Exact Match* Patches		# of *Correct* Patches		
	Bugs.jar	Bears	Time / # Patch	Defects4J	QuixBugs
	1000 bugs	119 bugs		313 bugs	37 bugs
TBar	-	-	3 Hour	63	-
CoCoNut	66	16	1000	47	13
SequenceR	99	14	300	41	15
Recoder	61	1	5 Hour	60	17
CURE	-	-	10000	59	25
RewardRepair	103	8	200	72	20
SelfAPR	-	-	-	86	-
KNOD	-	-	1000	102	25
CodeBERT	111	12	-	-	-
GraphCodeBERT	115	12	-	-	-
CodeT5	150	16	-	-	-
UniXcoder	164	22	-	-	-
AlphaRepair	-	-	≤ 5000	92	28
Repilot	-	-	≤ 5000	116	-
ChatRepair	-	-	≤ 200	142	**36**
ThinkRepair	-	-	≤ 125	155	**36**
RepairAgent	-	-	Avg. of 117	124	-
PATCH	**180** (9.76% ↑)	**28** (27.27% ↑)	≤ 100	**169** (9.03% ↑)	35 (2.86% ↓)

[11, 16, 21, 22, 47, 84, 98, 114], consistent with recent findings [57], which suggest that most developers are willing to review no more than 10 patches. For Defects4J and QuixBugs, we adopt the widely-used *test-passing* metric for patch assessment, following previous studies [5, 11, 29, 30, 47, 87, 90, 91, 96, 98, 100, 114]. A patch is considered *plausible* if it passes all test cases. All *plausible* patches are subsequently manually reviewed by comparing them to the developer-written ground truth. A patch is deemed *correct* if it is either 1) identical to the developer-written patch or 2) semantically equivalent to the developer-written patch. In our experiments, PATCH samples a small-scale set of candidate patches (at most 100 fixing attempts), fewer than those generated by all compared baselines, with a sampling temperature of 1 to ensure response diversity. This paper reports the number of *correct* patches generated by PATCH within these 100 attempts. In evaluating the results of corresponding baselines, we carefully compare the bug IDs correctly reported in the original papers with the set of single-hunk bugs selected from the Defects4J and QuixBugs benchmarks utilized in this paper.

Evaluation Results. Table 7 illustrates the number of bugs successfully fixed by PATCH and each baseline across different benchmarks in the context of fixing sing-hunk bugs. The **Time / # Patch** column indicates the maximum time allowed for fixing a bug or the maximum number of fixing attempts permitted for a model to generate before a *plausible* patch is obtained. A "-" indicates that no results have been reported in the corresponding papers. The best performance for each benchmark is highlighted in bold. As observed, PATCH demonstrates a substantial performance advantage over the baseline approaches across three benchmarks (except QuixBugs). Specifically, PATCH surpasses the best baseline by 9.76% in the Bugs.jar benchmark (compared to UniXcoder),

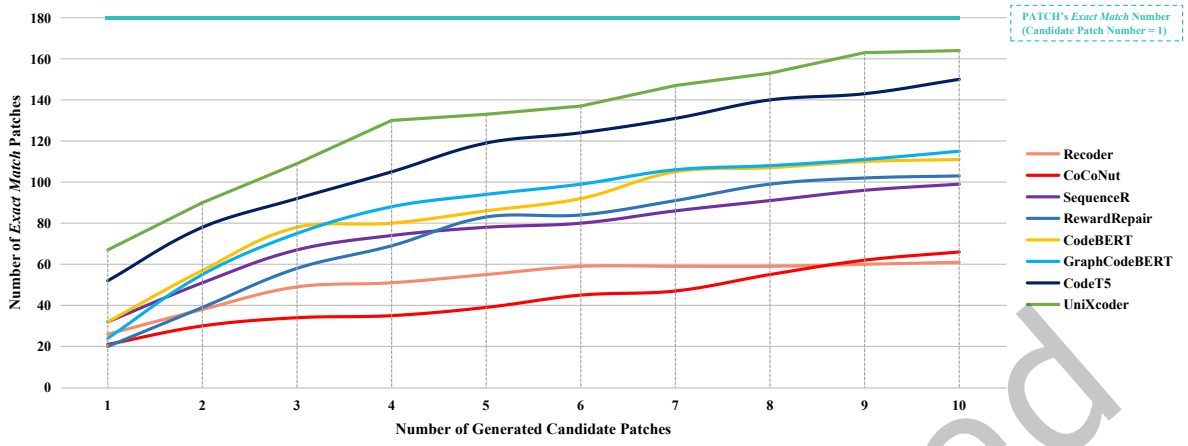

Fig. 15. The Number of Correct Patches Generated by Each Baseline on Bugs.jar under Different Candidate Numbers.

27.27% in the Bears benchmark (compared to UniXcoder), and 9.03% in the Defects4J benchmark (compared to ThinkRepair). For the QuixBugs benchmark, the two LLM-based APR approaches, namely ChatRepair and ThinkRepair, achieve superior performance, surpassing PATCH by 2.86%. This difference in performance may be attributed to two primary factors: first, these approaches leverage error message feedback obtained from the validation process using test suites; second, they sample a significantly larger number of candidate patches (over 100) for fixing a corresponding bug compared to PATCH. Consequently, the number of bugs fixed by PATCH on the QuixBugs benchmark is lower than that of ChatRepair and ThinkRepair.

Impact of the Candidate Number. We investigate the number of correct patches generated by each baseline on the Bugs.jar benchmark under different candidate numbers. As depicted in Figure 15, a large candidate set clearly increases the likelihood of containing the *exact match* patch. Notably, all baselines demonstrated improved bug-fixing performance with the increasing number of candidates. Nevertheless, it is noteworthy that PATCH consistently outperforms the baseline approaches when applied to greedy decoding (i.e., generating top-1 candidate patch for each bug).

Case Study on Defects4J. We present a Venn diagram (shown in Figure 16(a)) to further illustrate the bug-fixing performance across different approaches on Defects4J. For clarity, we highlight the top three best-performing baselines (i.e., ThinkRepair, ChatRepair, and RepairAgent) based on the number of correctly fixed bugs, and group all distinct correctly fixed bugs by the remaining baselines into an "Other" category for easier comparison. Two observations can be drawn from Figure 16(a): ① Individual approaches exhibit varying bug-fixing capabilities, with each being able to fix some specific bugs that others cannot. This suggests a degree of complementary performance among the approaches. ② Overall, PATCH generates the highest number of unique bug fixes (i.e., 36) that other baselines are unable to resolve. Figure 16(b) showcases three examples generated by PATCH that, while not exact matches, are semantically equivalent to the developer-written ground truth. For instance, in **Math-62**, the developer uses "$0.5 * (max - min)$" for the calculation, while PATCH employs the equivalent operation, i.e., "$(max - min)/2.0$". Additionally, we provide a bug (**Jsoup-83**) from Defects4J, depicted in Figure 16(c), that is uniquely fixed by PATCH. Specifically, we use the required edit operation (i.e., insert) to fix the bug as the commit message for prompting PATCH. It is reasonable to provide only the high-level description of the bug type without detailing the resolution process. As observed, PATCH benefits from the feedback information within the bug report and bug-fixing patterns. PATCH correctly introduces the

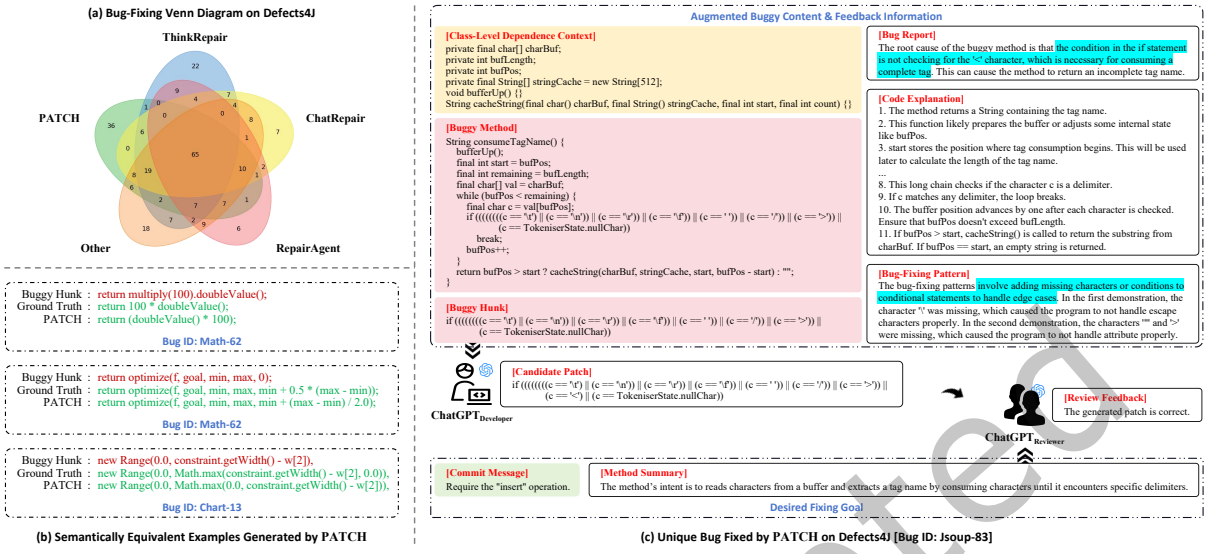

Fig. 16. Examples of the Bug-Fixing Venn Diagram, Equivalent Patches, and a Uniquely Fixed Bug by PATCH on Defects4J.

Table 8. Generalizability of PATCH on Different LLMs.

Model	Fix@1 (%) ↑	Simple Delete	Simple Insert	Simple Replace	Mixed
DeepSeek-Coder-1.3B	8.42	28	14	112	108
DeepSeek-Coder-1.3B w/ PATCH	9.19 **(9.14% ↑)**	29 **(+1)**	17 **(+3)**	120 **(+8)**	120 **(+12)**
CodeGeeX2-6B	10.93	23	51	129	137
CodeGeeX2-6B w/ PATCH	13.95 **(27.63% ↑)**	25 **(+2)**	57 **(+6)**	192 **(+63)**	160 **(+23)**
Mistral-7B	9.38	29	16	124	123
Mistral-7B w/ PATCH	15.68 **(67.16% ↑)**	39 **(+10)**	28 **(+12)**	244 **(+120)**	177 **(+54)**
CodeLLaMA-7B	9.45	29	13	136	116
CodeLLaMA-7B w/ PATCH	15.87 **(67.94% ↑)**	35 **(+6)**	25 **(+12)**	249 **(+113)**	185 **(+69)**
CodeLLaMA-13B	9.80	28	14	137	126
CodeLLaMA-13B w/ PATCH	18.83 **(92.14% ↑)**	38 **(+10)**	34 **(+20)**	287 **(+150)**	227 **(+101)**

necessary condition (i.e., "<") in the if statement to handle edge cases, ensuring the validity of consuming a complete tag. In this case, both the root cause identified by **ChatGPT$_{\text{Tester}}$** and the patterns derived from the retrieved similar demonstrations, as summarized by **ChatGPT$_{\text{Developer}}$**, contribute to the successful resolution of **Jsoup-83**, highlighting the collaborative capabilities of PATCH.

4.3.2 Generalizability Evaluation on Open-Source LLMs. We further employ PATCH to five open-source LLMs, including DeepSeek-Coder-1.3B, CodeGeeX2-6B, Mistral-7B, CodeLLaMA-7B, and CodeLLaMA-13B, all of which support interactive dialogues. Specifically, we conduct ablation experiments on each model individually to investigate the impact of interactively querying the corresponding LLM with devised prompts as specified in PATCH. To ensure the comparison fairness, the parameter configurations align with those described in Section 3.5. Table 8 presents the comparison results using the **Fix@1** metric, alongside the number of correct patches

generated for each bug type. Each model's results are depicted in two lines: the first line shows the results when the corresponding LLM directly utilizes the buggy code and a few bug-fixing demonstrations as the input prompt to resolve the given bug, and the second line shows the results when integrated with PATCH. From Table 8, we derive several key insights. ① **Integrating** PATCH **as a complementary plug-in enhances the bug-fixing performance across all LLMs.** The observed performance improvements range from 9.14% to 92.14%. ② **The performance gains exhibit a marked increase with the number of parameters exceeding 7B.** For instance, the performance gain observed with the DeepSeek-Coder-1.3B model is substantially smaller than that of the CodeLLaMA-13B model. ③ **The number of correct patches generated using** PATCH **is consistently higher across all types of bugs**, particularly for complex bug types such as **Simple Replace** and **Mixed**. In summary, PATCH effectively harnesses the latent intelligence of the LLMs in an interactive and collaborative manner, thereby improving their performance in the bug-fixing task.

Answer to RQ3: Since PATCH does not rely on fine-tuning with specific bug-fixing datasets, it demonstrates a lower susceptibility to generalizability issues. Consequently, PATCH outperforms traditional approaches across various APR benchmarks. Looking ahead, PATCH holds the potential to be seamlessly integrated with additional LLMs in a plug-and-play manner.

5 DISCUSSION

5.1 Integration with Fine-Tuned LLMs

We conduct a preliminary experiment by replacing the $\textbf{ChatGPT}_{\textbf{Developer}}$ component in the patch generation stage with the fine-tuned LLM RepairLLaMA [73] for APR, using the Defects4J benchmark. Figure 17 illustrates the intersection of correct bug fixes between PATCH and the variant model PATCH $_{\textbf{RepairLLaMA}}$. The experimental results indicates that PATCH, using vanilla ChatGPT, outperforms the variant using the fine-tuned LLaMA model. The performance decline can be attributed to two possible factors: 1) LLaMA has fewer parameters compared to ChatGPT, and 2) the input length for RepairLLaMA is constrained to 1024 tokens, which may cause certain bug examples in Defects4J, specifically those with longer buggy contexts and feedback information, to exceed this limit. Consequently, the absence of contextual information may result from the utilization of truncation strategies. Nevertheless, PATCH $_{\textbf{RepairLLaMA}}$ is still able to fix 30 bugs that PATCH could not address. Exploring more effective ways to integrate fine-tuned LLMs into PATCH will also remain as our future work.

Fig. 17. Bug-fixing Venn Diagram between PATCH and PATCH $_{\textbf{RepairLLaMA}}$ on Defects4J.

5.2 Multi-Hunk Bug-Fixing Scenario

Figure 18 presents an example of fixing the multi-hunk bug **Closure-128** from Defects4J using PATCH. Figure 18(a)-(e) illustrate the adapted prompts for the single-function multi-hunk fixing scenario at different bug-fixing stages. In the adapted framework, we modify the code representation of the buggy method by enclosing each buggy hunk with specific identifiers (i.e., **"[Buggy_Hunk_n_Begin]"** and **"[Buggy_Hunk_n_End]"**). Accordingly, PATCH generates the complete, fixed method as a candidate patch. As shown in Figure 18(d), PATCH successfully fixes **Closure-128** in the initial patch generation stage, producing a patch is semantically equivalent to the developer-written one. This is achieved through leveraging feedback information from earlier stages. Specifically, **ChatGPT$_{\text{Tester}}$** identifies the root causes of the two buggy hunks : **Buggy_Hunk_1** is caused by *missing validation before string operations*, and **Buggy_Hunk_2** results from *incorrect logic handling leading zeros for non-zero strings*. Subsequently, **ChatGPT$_{\text{Developer}}$** retrieves a similar bug-fixing example involving the patten of *changing conditions or logic for correct behavior*. In the patch generation step, PATCH addresses the first buggy hunk by adding validation for null or empty string, and fixes the second buggy hunk by adjusting the logic for handling leading zeros. Overall, PATCH can be adapted to multi-hunk bug-fixing scenarios by adjusting the content of corresponding prompts to accommodate the specific structure of these bugs.

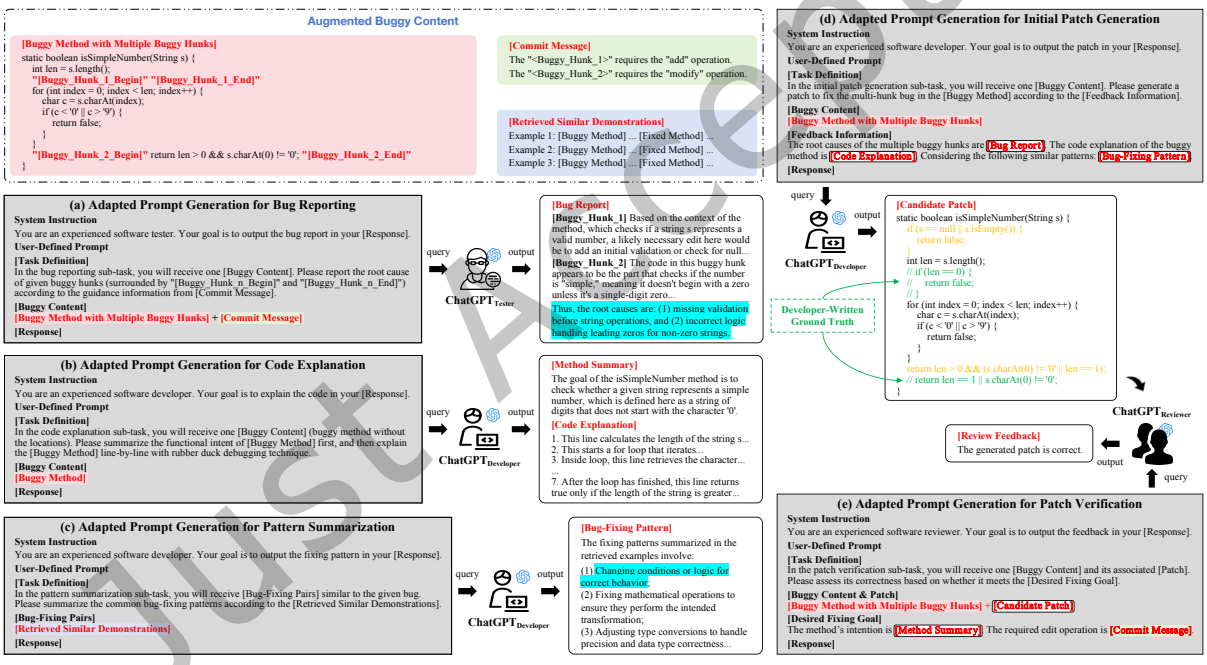

Fig. 18. An Example of Adaptation Prompts Used by PATCH for Fixing the Multi-Hunk Bug **Closure-128** from Defects4J.

5.3 Threats to Validity

In this subsection, we outline the main threats to the validity of PATCH, as detailed below:

- **External threat**: The primary threats to external validity in this paper pertain to the quality of the selected experimental subjects and the use of the commit message. The extent to which the improvements

achieved by PATCH are applicable to other bug-fixing benchmarks remains uncertain. To mitigate this concern, we have utilized the mainstream benchmark BFP, consistent with prior studies [78, 81, 84, 111], and supplemented the evaluation with four additional APR benchmarks to enhance the diversity of the evaluation. Furthermore, programmers typically write commit messages after fixing buggy code. In this paper, we assumed that programmers write these messages before bug fixing, potentially limiting PATCH's real-world applicability. Nevertheless, we view PATCH as a proof-of-concept. The empirical results indicate that utilizing commit messages as input can aid LLMs in comprehending programmers' reasoning processes. These messages offer valuable insights into the programmer's intent during the fix, thereby empowering LLMs with guided context. In addition, the retrieved bug-fixing pairs during the pattern summarization stage are crucial components of PATCH. Intuitively, when the retrieved code is less similar to the given bug, the performance of PATCH may degrade. We apply a dynamic threshold (described in Section 2.3.2) to ensure the quality of the retrieval. Empirical evidence also suggests that code reuse can reach up to 80% in real-world projects [54]. Therefore, we believe that retrieving relevant examples for summarizing bug-fixing patterns is highly feasible in practical development scenarios.

- **Internal threat**: LLMs exhibit sensitivity to prompts and hyper-parameters, especially concerning the number of task examples and natural language instructions, which can significantly affect their performance. To minimize this variability, we employ consistent prompts and hyper-parameters for PATCH and the LLM baselines. We refrain from experimental tuning of the prompt design and hyper-parameters, opting instead for empirical settings. As a result, we recognize that there is potential for further performance improvements through additional tuning. Another potential threat involves the possibility that the BFP benchmark, comprising Java projects hosted on GitHub between 2011 and 2017, may have been included in the training data of ChatGPT, raising concerns about the data leakage issue. Since ChatGPT is a closed-source model, the exact composition of its training data remains unknown. Despite this limitation, PATCH exhibits a significant improvement in bug-fixing performance compared to the base ChatGPT model, which employs the same underlying architecture. This enhancement suggests that the superior results achieved by PATCH are not merely attributable to the model's memorization of its training data. Additionally, the equivalent patch examples shown in Figure 16(b) and Figure 18(d) further supports our argument, as the Defects4J benchmark may be included in the training data of the GPT-series models [36]. Furthermore, while occasional inaccuracies arise, LLMs effectively capture the nature of bugs and provide clear, coherent code explanations. To reduce the likelihood of generating incorrect patches, we initially input commit messages at the bug reporting stage to assist ChatGPT in understanding the root cause of the given bug. Subsequently, we introduce the patch verification stage for further quality refinement.
- **Construct threat**: This paper uses **Fix@1** to assess the correctness of the generated candidate patches. Although this metric does not fully capture human judgment, it offers a strict and objective measure, facilitating rapid and quantitative evaluation of the model's performance. For the evaluation of APR benchmarks with test suites in Section 4.3.1, we adopt the widely-used *test-passing* metric to ensure a fair comparison. Future work will include additional human evaluations to further validate the models. In addition, PATCH specifically targets the resolution of single-hunk Java bugs utilizing ChatGPT. It is essential to note, however, that the components designed for PATCH can be effectively applied to various programming languages, integrated with multiple LLMs, and utilized in a wider range of bug-fixing scenarios. In Section 5.2, we present a case study evaluating PATCH in fixing single-function multi-hunk bugs through prompt adaptation. In future work, we aim to expand the evaluation scope of bug fixing to comprehensively assess the generalizability of PATCH.

6 RELATED WORK

6.1 Automatic Bug Fixing

Over the last decade, automatic bug fixing has emerged as a promising research topic, garnering significant attention from both the SE and AI communities. Traditional approaches can be broadly divided into two categories: search-based [19, 20, 33, 35, 44, 71] and semantics-based [1, 8, 34, 52, 93]. Search-based approaches typically rely on predefined bug-fixing patterns mined from historical open-source software repositories to generate patches, whereas semantics-based approaches generate patches by solving repair constraints derived from test suite specifications.

With the rapid advancement of DL techniques, there has been an increasing focus on neural-based approaches [104, 112], which have shown remarkable potential in enhancing bug-fixing performance. Unlike traditional bug-fixing approaches, learning-based techniques can automatically capture semantic relationships between parallel bug-fixing pairs, enabling the generation of more effective and context-aware patch solutions. However, candidate patches generated by pre-trained models are typically not evaluated against a test suite or subjected to other automated verification strategies. Consequently, these patches may encounter issues related to compilability. In contrast to existing learning-based studies that typically use only static source code as input, SelfAPR [96] extracts test execution diagnostics and encodes them into the input representation, guiding neural models in fixing specific bugs. Furthermore, while most existing approaches are designed to fix bugs at a single location, several multi-hunk bug-fixing methods have been proposed, utilizing either a divide-and-conquer strategy [41] or an iterative fixing paradigm [99].

Recently, researchers have explored the feasibility of employing LLMs for automatic bug fixing. LLMs have demonstrated the capability to directly generate correct patches based on the surrounding context, obviating the necessity for fine-tuning. Despite the unprecedented outcomes achieved by LLM-based approaches [28, 66, 75, 89], these techniques primarily focus on the buggy code and treat the bug fixing process as a single-stage task, neglecting the interactive and collaborative nature inherent in bug resolution. Nevertheless, recent advancements have shifted towards multi-step approaches. For example, ChatRepair [91] employs a conversational-driven approach, iteratively querying the LLM based on relevant test failure information derived from previous fix attempts. Similarly, ThinkRepair [100] tackles bug fixing through a two-step process: first performing few-shot learning using retrieved examples, followed by automatic interactions with LLMs, supplemented by feedback from test results. In contrast, RepairAgent [5] allows LLMs to interact with predefined tools that assist in the bug-fixing process. This paper introduces a stage-wise framework comprising multiple ChatGPT agents, each assigned to distinct stages within the bug management process using specific prompts. To the best of our knowledge, this is the first attempt to enhance the bug-fixing capabilities of LLMs through the guidance of programmer intent and the interactive simulation of collaborative behavior.

6.2 Large Language Model

Recent advancements in generative AI have led to a significant increase in the performance and widespread adoption of LLMs [108]. LLMs undergo unsupervised training using billions of open-source text/code tokens to achieve comprehensive language modeling. Due to the utilization of diverse data sources and their general design to acquire cross-domain knowledge, researchers can subsequently utilize LLMs for specific downstream tasks (e.g., improving the efficiency of programmers in writing, editing, and reviewing code [37, 82, 103]) by providing tailored prompts or a few demonstrations of the task being solved as input [45]. Among LLMs, the GPT family developed by OpenAI [6, 9] is particularly notable for its popularity and prowess. Additionally, numerous attempts have been made to reproduce similar open-source LLMs such as CodeGPT [46], LLaMA [80], and others. Despite their robust performance, LLMs sometimes struggle to produce accurate results when faced with complex tasks. In response, researchers have proposed the use of advanced prompting techniques (e.g.,

Chain of Thoughts (CoT) [86] and Tree of Thoughts (ToT) [95]) to enhance the reasoning capability of LLMs in natural language processing tasks. These techniques involve a sequence of intermediate reasoning steps in natural language that culminate in the final output. More recently, researchers have proposed LLMs trained using reinforcement learning to better align with human preferences. Examples of such models include InstructGPT [62] and ChatGPT [59], which are initially initialized from a pre-trained model on autoregressive generation and then fine-tuned using reinforcement learning from human feedback (RLHF) [116]. This fine-tuning process, which incorporates human preferences, has significantly enhanced the ability of these LLMs to comprehend input prompts and follow instructions to perform complex tasks [2]. Notably, ChatGPT has achieved superior performance in various SE tasks [12, 75]. The objective of this paper is to draw insights from effective bug management practices to enhance the capabilities of existing LLMs in the task of bug fixing. Our experimental results demonstrate that such alignment enables ChatGPT to interact and collaborate, significantly outperforming traditional LLMs.

7 CONCLUSION AND FUTURE WORK

This paper introduces a stage-wise framework aimed at enhancing the bug-fixing capabilities of LLMs in an interactive manner. We explore the potential of ChatGPT by simulating the behavior of human programmers engaged in bug management. Specifically, we augment the BFP benchmark by providing contextual information to better guide LLMs in generating the correct patches. Moreover, we decompose the bug-fixing task into four distinct stages and employ three ChatGPT agents to collectively produce candidate patches for bug resolution using devised prompts. We conduct extensive experiments to demonstrate the effectiveness of PATCH. We firmly believe that aligning the collaborative problem-solving skills of programmers with LLMs represents a pivotal stride towards intelligent SE research.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments. This work was partially supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant Nos. XDA0320100 and XDA0320102), the National Natural Science Foundation of China (Grant Nos. 62192731, 62192733, and 62192730), the Major Program (JD) of Hubei Province (Grant No. 2023BAA024), the Major Project of ISCAS (Grant No. ISCAS-ZD-202302), the Basic Research Project of ISCAS (Grant No. ISCAS-JCZD-202403), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (Grant Nos. Y2022044 and 2023121), and the Fundamental Research Funds for the Central Universities (Grant No. JK2024-28).

REFERENCES

[1] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Trans. Software Eng.* 47, 10 (2021), 2162–2181.

[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 (IJCNLP'23)*. Association for Computational Linguistics, Nusa Dua, 675–718.

[3] Muneera Bano, Rashina Hoda, Didar Zowghi, and Christoph Treude. 2024. Large Language Models for Qualitative Research in Software Engineering: Exploring Opportunities and Challenges. *Autom. Softw. Eng.* 31, 1 (2024), 8.

[4] Sid 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. *CoRR* abs/2204.06745 (2022).

[5] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR* abs/2403.17134 (2024).

[6] 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 Annual Conference on Neural Information Processing Systems (NeurIPS'20)*. , Virtual Event.

[7] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE, Melbourne, 443–455.

[8] Liushan Chen, Yu Pei, and Carlo A. Furia. 2021. Contract-Based Program Repair Without The Contracts: An Extended Study. *IEEE Trans. Software Eng.* 47, 12 (2021), 2841–2857.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR* abs/2107.03374 (2021).

[10] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *Proceedings of the 12th International Conference on Learning Representations (ICLR'24)*. OpenReview.net, Vienna.

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

[12] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* 33, 7 (2024), 189:1–189:38.

[13] Tore Dybå, Vigdis By Kampenes, and Dag I. K. Sjøberg. 2006. A Systematic Review of Statistical Power in Software Engineering Experiments. *Inf. Softw. Technol.* 48, 8 (2006), 745–755.

[14] Çagri Eren, Kerem Sahin, and Eray Tüzün. 2023. Analyzing Bug Life Cycles to Derive Practical Insights. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE'23)*. ACM, Oulu, 162–171.

[15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. ACM, Vasteras, 313–324.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, Virtual Event, 1536–1547.

[17] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR'23)*. OpenReview.net, Kigali.

[18] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *CoRR* abs/2101.00027 (2021).

[19] 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 (ISSTA'19)*. ACM, Beijing, 19–30.

[20] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72.

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL'22)*. Association for Computational Linguistics, Dublin, 7212–7225.

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR'21)*. OpenReview.net, Virtual Event.

[23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024).

[24] Zeyu He, Chieh-Yang Huang, Chien-Kuang Cornelia Ding, Shaurya Rohatgi, and Ting-Hao Kenneth Huang. 2024. If in a Crowdsourced Data Annotation Pipeline, a GPT-4. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI'24)*. ACM, Honolulu, HI, 1040:1–1040:25.

[25] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, Luxembourg, 1162–1174.
[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019).
[27] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *CoRR* abs/2310.06825 (2023).
[28] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. ACM, Melbourne, 1430–1442.
[29] 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 IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, Melbourne, 1251–1263.
[30] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*. IEEE, Madrid, 1161–1173.
[31] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, San Jose, CA, 437–440.
[32] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The Stack: 3 TB of Permissively Licensed Source Code. *Trans. Mach. Learn. Res.* 2023 (2023).
[33] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.
[34] Xuan-Bach Dinh 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 (ESEC/FSE'17)*. ACM, Paderborn, 593–604.
[35] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. IEEE Computer Society, Suita, Osaka, 213–224.
[36] Jae Yong Lee, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2024. The GitHub Recent Bugs Dataset for Evaluating LLM-Based Debugging Applications. In *Proceedings of the 17th IEEE Conference on Software Testing, Verification and Validation (ICST'24)*. IEEE, Toronto, ON, 442–444.
[37] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. CodeEditor: Learning to Edit Source Code with Pre-trained Models. *ACM Trans. Softw. Eng. Methodol.* 32, 6 (2023), 143:1–143:22.
[38] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE, Melbourne, 155–166.
[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023).
[40] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.
[41] 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 IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. ACM, Pittsburgh, PA, 511–523.
[42] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH'17)*. ACM, Vancouver, BC, 55–56.

[43] Yngve Lindsjørn, Dag I. K. Sjøberg, Torgeir Dingsøyr, Gunnar R. Bergersen, and Tore Dybå. 2016. Teamwork Quality and Project Success in Software Development: A Survey of Agile Development Teams. *J. Syst. Softw.* 122 (2016), 274–286.

[44] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. ACM, Beijing, 31–42.

[45] 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 (2023), 195:1–195:35.

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks'21)*. , Virtual Event.

[47] 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 (ISSTA'20)*. ACM, Virtual Event, 101–114.

[48] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, Hangzhou, 468–478.

[49] Antonio Mastropaolo, Camilo Escobar-Velásquez, and Mario Linares-Vásquez. 2024. The Rise and Fall(?) of Software Engineering. *CoRR* abs/2406.10141 (2024).

[50] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320.

[51] Ian R. McChesney and Séamus Gallagher. 2004. Communication and Co-Ordination Practices in Software Engineering Projects. *Inf. Softw. Technol.* 46, 7 (2004), 473–489.

[52] 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 (ICSE'16)*. ACM, Austin, TX, 691–701.

[53] Cory Merow, Josep M. Serra-Diaz, Brian J. Enquist, and Adam M. Wilson. 2023. AI Chatbots Can Boost Scientific Coding. *Nat. Ecol. Evol.* , (2023), 1–3.

[54] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A Study of Repetitiveness of Code Changes in Software Evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE, Silicon Valley, CA, 180–190.

[55] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *CoRR* abs/2305.02309 (2023).

[56] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI'22)*. ijcai.org, Vienna, 5546–5555.

[57] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. ACM, Pittsburgh, PA, 2228–2240.

[58] Masao Ohira, Ahmed E. Hassan, Naoya Osawa, and Ken-ichi Matsumoto. 2012. The Impact of Bug Management Patterns on Bug Fixing: A Case Study of Eclipse Projects. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*. IEEE, Trento, 264–273.

[59] OpenAI. 2022. *Introducing ChatGPT*. Technical Report. . https://openai.com/blog/chatgpt

[60] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[61] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. 2014. Mining Frequent Bug-Fix Code Changes. In *Proceedings of the 2014 IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*. IEEE, Antwerp, 343–347.

[62] Long Ouyang, Jeffrey 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 F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS'22)*. , New Orleans, LA.

[63] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Softw.* 40, 3 (2023), 4–8.

[64] Rishov Paul, Md. Mohib Hossain, Masum Hasan, and Anindya Iqbal. 2023. Automated Program Repair Based on Code Review: How do Pre-Trained Transformer Models Perform? *CoRR* abs/2304.07840 (2023).

[65] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Softw. Pract. Exp.* 46, 9 (2016), 1155–1179.

[66] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An Evaluation on QuixBugs. In *Proceedings of the 3rd IEEE/ACM International Workshop on Automated Program Repair (APR@ICSE)*. IEEE, Pittsburgh, PA, 69–75.

[67] Peter C. Rigby, Brendan Cleary, Frédéric Painchaud, Margaret-Anne D. Storey, and Daniel M. Germán. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Softw.* 29, 6 (2012), 56–61.

[68] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389.

[69] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code LLaMA: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).

[70] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. ACM, Gothenburg, 10–13.

[71] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE / ACM, Montreal, QC, 13–24.

[72] Jessica Shieh. 2023. *Best Practices for Prompt Engineering with OpenAI API.* Technical Report. . https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

[73] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *CoRR* abs/2312.15698 (2023).

[74] Ajay S. Singh and Micah B. Masuku. 2014. Sampling Techniques & Determination of Sample Size in Applied Statistics Research: An Overview. *International Journal of Economics, Commerce and Management* 2, 11 (2014), 1–22.

[75] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *Proceedings of the 4th IEEE/ACM International Workshop on Automated Program Repair (APR@ICSE)*. IEEE, Melbourne, 23–30.

[76] Diomidis Spinellis. 2000. The Pragmatic Programmer: From Journeyman to Master. *IEEE Softw.* 17, 6 (2000), 108–110.

[77] Shin Hwei Tan, Ziqiang Li, and Lu Yan. 2024. CrossFix: Resolution of GitHub Issues via Similar Bugs Recommendation. *J. Softw. Evol. Process.* 36, 4 (2024).

[78] Yu Tang, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Muyun Yang. 2021. Grammar-Based Patches Generation for Automated Program Repair. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP*. Association for Computational Linguistics, Virtual Event, 1300–1305.

[79] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message?. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. ACM, Pittsburgh, PA, 2389–2401.

[80] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

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

[82] Rosalia Tufano, Ozren Dabic, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. 2024. Code Review Automation: Strengths and Weaknesses of the State of the Art. *IEEE Trans. Software Eng.* 50, 2 (2024), 338–353.

[83] Jing Wang, Patrick C. Shih, Yu Wu, and John M. Carroll. 2015. Comparative Case Studies of Open Source Software Peer Review Practices. *Inf. Softw. Technol.* 67 (2015), 1–12.

[84] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP'21)*. Association for Computational Linguistics, Punta Cana, 8696–8708.

[85] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, Melbourne, 349–360.

[86] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS'22)*. , New Orleans, LA.

[87] 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 (ESEC/FSE'23)*. ACM, San Francisco, CA, 172–184.

[88] W. Eric Wong, Xue-Lin Li, and Phillip A. Laplante. 2017. Be More Familiar with Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures. *J. Syst. Softw.* 133 (2017), 68–94.

[89] 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 IEEE/ACM International Conference on Software Engineering (ICSE'23)*. ACM, Melbourne, 1482–1494.

[90] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. ACM, Singapore, 959–971.

[91] 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 (ISSTA'24)*. ACM, Vienna, 819–831.

[92] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit Message Generation for Source Code Changes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. ijcai.org, Macao, 3975–3981.

[93] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55.

[94] Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi. 2016. Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS'16)*. IEEE, Vienna, 191–201.

[95] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS'23)*. , New Orleans, LA.

[96] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. ACM, Rochester, MI, 92:1–92:13.

[97] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated Patch Assessment for Program Repair at Scale. *Empir. Softw. Eng.* 26, 2 (2021), 20.

[98] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. ACM, Pittsburgh, PA, 1506–1518.

[99] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24)*. ACM, Lisbon, 10:1–10:13.

[100] 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 (ISSTA'24)*. ACM, Vienna, 1274–1286.

[101] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Broomfield, CO, 249–265.

[102] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-Trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. ACM, Virtual Event, 39–51.

[103] Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Jia Allen Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, and Zhi Jin. 2024. Deep Learning for Code Generation: A Survey. *Sci. China Inf. Sci.* [online first] (2024). https://doi.org/10.1007/s11432-023-3956-3

[104] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-Based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69.

[105] Tao Zhang, He Jiang, Xiapu Luo, and Alvin T. S. Chan. 2016. A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions. *Comput. J.* 59, 5 (2016), 741–773.

[106] Yuwei Zhang. 2024. *Replicate Package of PATCH.* Zenodo. https://doi.org/10.5281/zenodo.14257480

[107] Yuwei Zhang, Ge Li, Zhi Jin, and Ying Xing. 2023. Neural Program Repair with Program Dependence Analysis and Effective Filter Mechanism. *CoRR* abs/2305.09315 (2023).

[108] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023).

[109] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning (ICML'21)*. PMLR, Virtual Event, 12697–12706.

[110] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'23)*. ACM, Long Beach, CA, 5673–5684.

[111] Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. 2022. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated*

Software Engineering (ASE'22). ACM, Rochester, MI, 97:1–97:13.
[112] Wenkang Zhong, Chuanyi Li, Jidong Ge, and Bin Luo. 2022. Neural Program Repair: Systems, Challenges and Solutions. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware (Internetware'22)*. ACM, Hohhot, 96–106.
[113] Wenkang Zhong, Chuanyi Li, Kui Liu, Jidong Ge, Bin Luo, Tegawendé F. Bissyandé, and Vincent Ng. 2024. Benchmarking and Categorizing the Performance of Neural Program Repair Systems for Java. *ACM Trans. Softw. Eng. Methodol.* [online first] (2024). https://doi.org/10.1145/3688834
[114] 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. ACM, Athens, 341–353.
[115] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. ACM, Melbourne, 1443–1455.
[116] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul F. Christiano, and Geoffrey Irving. 2019. Fine-Tuning Language Models from Human Preferences. *CoRR* abs/1909.08593 (2019).
[117] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Software Eng.* 36, 5 (2010), 618–643.
[118] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Trans. Software Eng.* 46, 8 (2020), 836–862.

Received 17 August 2024; revised 4 December 2024; accepted 13 February 2025