Coding in a Bubble? Evaluating LLMs in Resolving Context Adaptation Bugs During Code Adaptation

TANGHAORAN ZHANG, National University of Defense Technology, China
XINJUN MAO*, National University of Defense Technology, China
SHANGWEN WANG, National University of Defense Technology, China
YUXIN ZHAO, National University of Defense Technology, China
YAO LU, National University of Defense Technology, China
ZEZHOU TANG, National University of Defense Technology, China
WENYU XU, National University of Defense Technology, China
LONGFEI SUN, National University of Defense Technology, China
CHANGRONG XIE, National University of Defense Technology, China
KANG YANG, National University of Defense Technology, China
YUE YU*, Peng Cheng Laboratory, China

Code adaptation is a fundamental but challenging task in software development, requiring developers to modify existing code for new contexts. A key challenge is to resolve *Context Adaptation Bugs (CtxBugs)*, which occurs when code correct in its original context violates constraints in the target environment. Unlike isolated bugs, *CtxBugs* cannot be resolved through local fixes and require cross-context reasoning to identify semantic mismatches. Overlooking them may lead to critical failures in adaptation. Although Large Language Models (LLMs) show great potential in automating code-related tasks, their ability to resolve *CtxBugs* remains a significant and unexplored obstacle to their practical use in code adaptation.

To bridge this gap, we propose *CtxBugGen*, a novel framework for generating *CtxBugs* to evaluate LLMs. Its core idea is to leverage LLMs' tendency to generate plausible but context-free code when contextual constraints are absent. The framework generates *CtxBugs* through a four-step process to ensure their relevance and validity: (1) Selection of four established context-aware adaptation tasks from the literature, (2) Perturbation via task-specific rules to induce *CtxBugs* from LLMs while ensuring their plausibility, (3) Generation of candidate variants by prompting LLMs without any context constraints and (4) Identification of valid *CtxBugs* through syntactic differencing and test execution in the target context. Based on the benchmark constructed by *CtxBugGen*, we conduct an empirical study with four state-of-the-art LLMs. Our results reveal their

---

*Xinjun Mao and Yue Yu are corresponding authors.

Authors' Contact Information: Tanghaoran Zhang, zhangthr@nudt.edu.cn, National University of Defense Technology, China; Xinjun Mao, xjmao@nudt.edu.cn, National University of Defense Technology, China; Shangwen Wang, wangshangwen13@nudt.edu.cn, National University of Defense Technology, China; Yuxin Zhao, yuxinzhao@nudt.edu.cn, National University of Defense Technology, China; Yao Lu, luyao08@nudt.edu.cn, National University of Defense Technology, China; Zezhou Tang, tangzezhou24@nudt.edu.cn, National University of Defense Technology, China; Wenyu Xu, xuwenyu@nudt.edu.cn, National University of Defense Technology, China; Longfei Sun, lfsun@nudt.edu.cn, National University of Defense Technology, China; Changrong Xie, xiechangrong02@nudt.edu.cn, National University of Defense Technology, China; Kang Yang, yangkang@nudt.edu.cn, National University of Defense Technology, China; Yue Yu, yuy@pcl.ac.cn, Peng Cheng Laboratory, China.

---

unsatisfactory performance in *CtxBug* resolution. The best performing LLM, Kimi-K2, achieves 55.93% on Pass@1 and resolves just 52.47% of *CtxBugs*. The presence of *CtxBugs* degrades LLMs' adaptation performance by up to 30%. Failure analysis indicates that LLMs often overlook *CtxBugs* and replicate them in their outputs. This suggests that LLMs overly focus on the local code correctness of the reused code while ignoring its compatibility in the target context. Our study highlights a critical weakness in LLMs' cross-context reasoning and emphasize the need for new methods to enhance their context awareness for reliable code adaptation. The replication package for this paper is at https://github.com/ztwater/CtxBugGen.

CCS Concepts: • **Software and its engineering** → *Reusability*; **Automatic programming**.

Additional Key Words and Phrases: Code Adaptation, Large Language Models, Context Adaptation Bugs

1 Introduction

Code reuse is a widely adopted practice in software development, significantly boosting productivity by allowing developers to leverage existing solutions rather than building everything from scratch [5, 21, 26, 63]. However, their reused code is inherently designed for its original context and may violate specific requirements or constraints of the target context. Thus, developers must always adapt reused code to fit its new context, ensuring its correctness and quality [58, 67, 70]. While developers are alert to pre-existing bugs in reused code, they often overlook the risk of semantic mismatches. This occurs when code that is bug-free in its original context is integrated into a new context without adequate adaptation. These resulting bugs are defined as **Context Adaptation Bugs (CtxBugs)** [37]. Unlike isolated bugs, resolving *CtxBugs* requires developers to perform cross-context reasoning to align the code's behavior with new context.

Figure 1 illustrates an example of *CtxBug* in code adaptation. Developer Alice needs to implement a feature for combining bitwise status flags in her utility class. She queries a search engine or an AI assistant with "How to combine two status values" and receives a solution using the arithmetic plus operator ("+"). While this numerical addition is a semantically feasible implementation, it introduces a *CtxBug* in Alice's context, where a bitwise OR ("|") is required to merge flags. This mismatch is subtle, because the reused code compiles and even works for cases like 1 + 2 = 3, but it would fail for others like 1 + 1. This example demonstrates that *CtxBugs* are not intrinsic and isolated bugs in the reused code but semantic mismatches arising from integration, making their resolution critical during the adaptation process.

Recent advancements in large language models (LLMs) have shown great potential in automating various software engineering tasks, including program repair, code optimization, as well as code adaptation [20, 60, 61, 68, 71]. This capability stems from their powerful prompt-based learning ability [7, 33], which allows them to perform unseen tasks based on instructions provided in a prompt. However, despite their general applicability, a recent study found that LLMs perform significantly worse on code adaptation compared to code generation, with a reported performance decrease of 15% [71]. We hypothesize that this weakness is relevant to LLMs' limited ability to resolve *CtxBugs* during adaptation. To demonstrate this, we apply four state-of-the-art LLMs to the adaptation task presented in Fig. 1. In line with the experimental design of Zhang et al. [71], each LLM was prompted with the requirement, the reused code, and the target context. Our result indicates that all LLMs failed to resolve the *CtxBug* and expressed high confidence in generating the incorrect operator. Their consistent failures suggest a fundamental limitation in their current cross-context reasoning capability. Consequently, we argue that rigorously evaluating
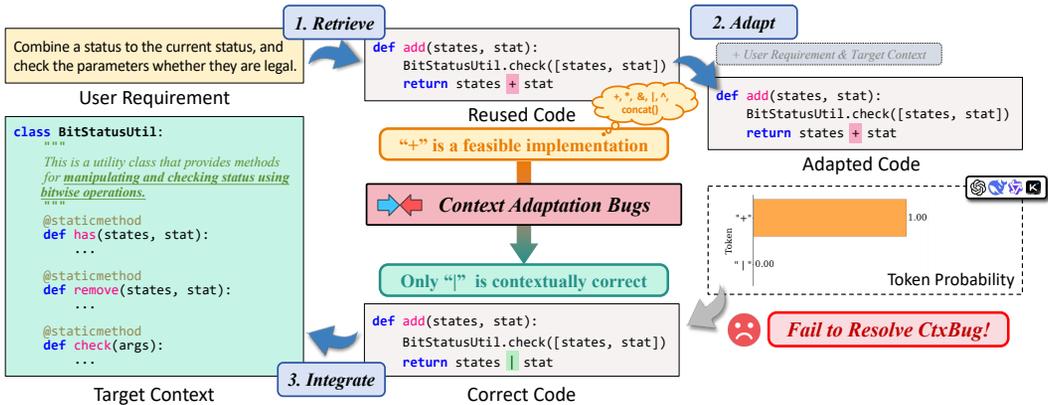
Fig. 1. Example of a *CtxBug* in code adaptation. All evaluated state-of-the-art LLMs failed to resolve it.

their effectiveness in resolving *CtxBugs* is a critical prerequisite for their reliable adoption in automated code adaptation.

Despite the significance of *CtxBug* resolution, the lack of dedicated adaptation benchmarks hinders research in this domain. To address this gap, we propose *CtxBugGen*, a novel framework to generate *CtxBugs* for adaptation evaluation. Its core insight is that code generated by LLMs with no contextual constraints is inherently general but plausible, making it likely to introduce semantic mismatches for a specific target context, thereby creating *CtxBugs*. To ensure the relevance and validity of generated *CtxBugs*, *CtxBugGen* employs a four-step pipeline: *(1) Adaptation Task Selection*: We select four context-aware adaptation tasks from current literature to ground our *CtxBugs* in realistic developer needs; *(2) Task-Specific Perturbation*: To ensure the plausibility of LLMs' generation, we preserve the overall code structure and perturb minimal but targeted code elements for each adaptation task. *(3) LLM-Based Variant Generation*: We instruct LLMs to generate code at the perturbed locations without any contextual constraints. This produces candidates that are semantically correct in isolation but may introduce *CtxBugs*; and *(4) CtxBug Identification*: We combine syntactic differencing with test execution in the target context to identify code with valid *CtxBugs*. Finally, we manually validate their quality and obtain a benchmark with 3,683 *CtxBugs*.

Based on our benchmark, we empirically evaluate four state-of-the-art LLMs regarding their *CtxBug* resolution capability. Results demonstrate unsatisfactory performance across all models. The best performing LLM, Kimi-K2, achieves only 55.93% on Pass@1 and resolves just 52.47% of *CtxBugs*. Their performance further drops to 36.65% Pass@1 and 27.24% resolution rate when specifically evaluated on the *Functionality Customization* task. Comparative analysis indicates that the presence of *CtxBugs* leads to LLMs' performance degradation of up to 30%, suggesting their disruptive impact on adaptation. LLMs can even handle isolated bugs more effectively than *CtxBugs* during adaptation. Through failure analysis, we observe that LLMs frequently overlook *CtxBugs* and directly replicate them in their outputs. This suggests that LLMs prioritize the local semantic correctness of code while ignoring their contextual constraints.

This paper makes the following contributions:

- We present the first comprehensive study to investigate the effectiveness of LLMs in resolving *CtxBugs*, which is a critical challenge that limits their utilization in code adaptation.
- We propose *CtxBugGen*, a novel framework for effective *CtxBug* generation, and use it to construct the first benchmark for four established adaptation tasks leveraging four LLMs.

- Our experimental results reveal the limitations of state-of-the-art LLMs in handling *CtxBugs* and their negative impacts on performance, highlighting the critical need to improve cross-context reasoning in LLMs.

2 Background and Motivation

This section introduces the background of code adaptation and its recent advances in LLM-based approaches, followed by motivating examples showing the impact of *CtxBugs* on LLMs' adaptation.

2.1 Code Adaptation

Code adaptation is a fundamental activity in software development. Although open-source platforms such as Stack Overflow (SO) provide millions of ready-to-use code snippets, integrating them often requires extensive manual efforts [58, 67]. The growing adoption of AI-generated code makes it more challenging, as such code can be difficult to understand and debug [4, 31].

Early research, such as Jigsaw [12], attempts to automate code adaptation using syntactic similarity. Subsequent studies aim to improve the compilability and reusability of reused code [51, 62]. For instance, Terragni et al. introduce "APIzation" [52], which converts code snippets into well-formed, reusable methods, while Zhang et al. [70] leverage historical adaptations to generate reusable templates. However, these approaches largely overlook the semantic differences across contexts. This gap is identified by Mondal et al. [37], who introduce the concept of *Context Adaptation Bugs* (*CtxBugs*) that occur when developers clone correct code without proper adaptation. While their study focuses on the evolution of *CtxBugs* in intra-project code clones, it does not explore broader code reuse scenarios, leaving a critical gap in addressing such semantic mismatches during code adaptation.

With the emergence of LLMs, researchers have started to investigate their potential in automating various code adaptation tasks [36, 54, 68, 71]. The large-scale parameters and extensive training of LLMs bring them emergent abilities such as prompt-based learning [33]. LLMs are allowed to perform unseen tasks which previously require architecture changes and further fine-tuning [30, 32]. One of the most motivational studies is conducted by Zhang et al. [71], who first investigate LLMs' capabilities in end-to-end adaptation tasks. Their prompt-based approach leverages both the pretrained knowledge and contextual code understanding of LLMs, making it applicable to a wide range of adaptation scenarios. However, their results also reveal that LLMs remain less effective at adaptation tasks than they are at code generation. To further reveal LLMs' strengths and limitations in this task, our study focuses on evaluating their effectiveness in resolving *CtxBugs*.

2.2 A Motivating Example

Fig. 2 illustrates the impact of *CtxBugs* on LLMs' adaptation, which could be misleading and more harmful than isolated bugs. Based on the setup of Fig. 1, we employ LLMs for adaptation under three scenarios: code with a *CtxBug*, code without *CtxBugs*, and code with an isolated bug. To simulate these, we replace the bug-inducing operator "+" with a placeholder "*<INFILL>*" for the bug-free case, and introduce an isolated bug using "-" (which violates status combination requirement) in the third case. Then we compare LLMs' outputs and token distributions across all cases.

In Case ❶, the LLM (e.g., GPT-4o) fails to adapt code with *CtxBug*, assigning 100% probability to the incorrect token. In contrast, Case ❷ shows that the LLM correctly adapts the code when no *CtxBug* is present. Case ❸ further shows that the LLM can also correctly handle the isolated bug under the same setting. In both correct scenarios, the LLM shows no tendency to generate the incorrect token, while it is misled by the *CtxBug* in the first case. These results suggest that LLMs are susceptible to *CtxBugs*, likely because *CtxBugs* appear semantically consistent with the local context, such as the method name "*add*". This hinders LLMs from capturing global context constraints
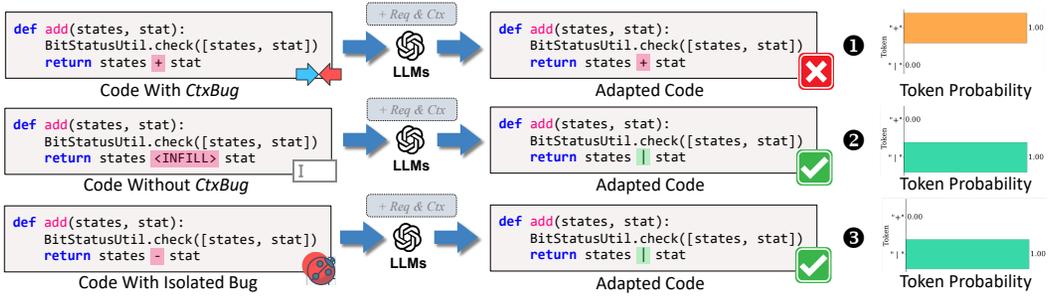
Fig. 2. A motivating example to demonstrate how a *CtxBug* can mislead LLMs during adaptation. Bar charts at the right side show LLM-generated token distributions at our concerned position.
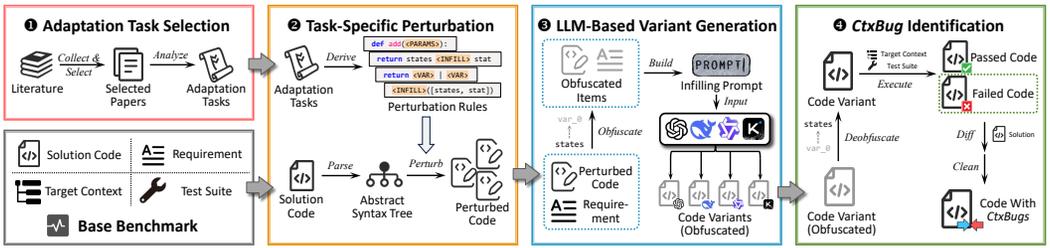

Fig. 3. The benchmark construction process through the *CtxBugGen* framework.

(e.g., bitwise state operations) and leads to their poor performance in code adaptation. The above observation motivates our study to systematically evaluate LLMs' capabilities in resolving *CtxBugs*.

3 *CtxBugGen* Framework

This section introduces *CtxBugGen*, a framework for constructing evaluation benchmarks containing *CtxBugs*. Its core idea is to leverage LLMs to generate code by deliberately withholding contextual constraints. This elicits semantically plausible but contextually incorrect outputs, thereby introducing *CtxBugs*. An overview of *CtxBugGen* is shown in Fig. 3. It includes four steps: adaptation task selection (Section 3.1), task-specific perturbation (Section 3.2), LLM-based variant generation (Section 3.3) and *CtxBug* identification (Section 3.4). Each step is detailed as follows.

3.1 Adaptation Task Selection

To ensure our benchmark reflects practical adaptation challenges, we systematically analyze the literature to answer a key question:

What are the primary code adaptation issues identified in existing studies?

Following guidelines for mapping studies in software engineering [28, 41], we first conduct an explicit literature collection and selection process to retrieve relevant papers. We then extract and analyze all data items related to code adaptation issues through thematic analysis [6], and finally report our results. All data and materials are available in our replication package.

3.1.1 Literature Collection and Selection. As shown in Fig. 4, our literature collection and selection process consists of three steps: (1) keyword search, (2) selection by inclusion and exclusion criteria and (3) snowballing. The detail of each step is as follows.
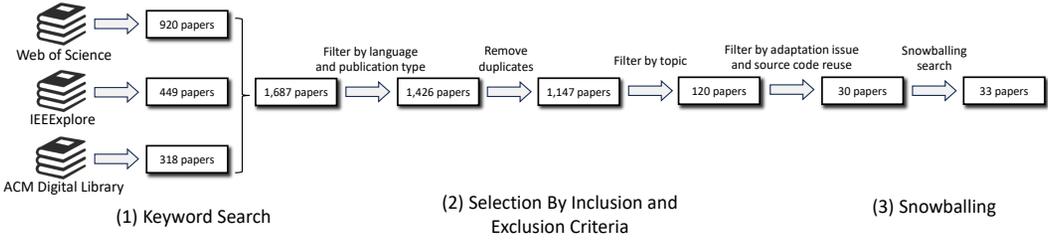
Fig. 4. Our literature collection and selection process.

Keyword Search. In line with prior surveys [65, 66], we select Web of Science, IEEEXplore and ACM Digital Library as our databases. As we aim to cover papers about code adaptation as comprehensive as possible, we also include *code reuse* and *code integration* along with the most precise term, *code adaptation* in our keywords. To the best of our knowledge, the first related study on source code adaptation was Jigsaw [13], which was published in 2008. Hence, the time interval of collection is set from 2008 to 2025, lasting for 17 years. This step yields 1,687 papers in total.

Selection By Inclusion and Exclusion Criteria. This step aims to further evaluate the quality and relevance of our collect papers. Our criteria are shown in Table 1. The selection is conducted by two researchers with rich experience in software engineering and code adaptation. We first use automatic filters to apply *Exclusion Criteria 1-3*, resulting in 1,147 papers. Then we manually read the title, abstract and keywords of each paper to determine whether they fit our topic according to *Inclusion Criteria 1*. After this step, we narrow down to 120 papers. Subsequently, we look through these papers to identify source code reuse and related adaptation issues (*Inclusion Criteria 2 & Exclusion Criteria 4*), resulting in 30 papers.

Table 1. Inclusion and exclusion criteria of our literature selection.

Criteria	Explanation
Inclusion	1. The paper should be related to the topic of code reuse or adaptation. 2. The paper should focus on at least one code adaptation issue derived from empirical evidences.
Exclusion	1. Non-English written paper should be discarded. 2. Books, chapters, thesis, or grey publications should be discarded. 3. Duplicated papers included in multiple databases or similar studies with different versions should be discarded. 4. The paper without including *source code reuse* should be discarded, e.g., component, library or non-code artifact reuse.

Snowballing. To identify additional relevant studies, we conduct a snowballing search in the last step. By looking through the reference and citation list of above 30 papers, we manually add three papers which are missed by our keyword search. Additionally, we ask another researcher to double-check our final list of papers to ensure that they satisfy our criteria.

3.1.2 Data Item Extraction and Analysis. In this step, we extract the essential data items from our selected papers, which helps us summarize current adaptation issues. Two researchers annotate data items following the principles of thematic analysis [6]. It includes two phases: (1) independently code each data item with its representative information, e.g., what property of source code is mentioned; and (2) work together to discuss about initial codes and group them into themes with descriptive names. To avoid bias, we invite another researcher to reassign the established themes to each item for cross validation. Its Cohen's Kappa [10] is 0.932, which indicates an almost perfect agreement between annotators. Results are shown in Table 2, including five categories (themes) of adaptation issues. Their detailed explanations are in the following.

Table 2. Adaptation issues identified by existing code adaptation literature.

Category	Description	Related Studies	Num	Ctx-Based
Interface Specification	Mismatches between the *interface* (e.g., signature) of the reused code and its specifications in the target context.	[36, 52, 55, 62, 67, 71]	6	✔
Functionality Customization	Mismatches between the *internal logic* (e.g., conditionals) of reused code and its desired behavior in the target context.	[15, 44, 47, 51, 58, 62, 64, 67, 70, 71]	10	✔
Identifier Reference	Incorrect *identifier references* to the local (e.g., local variables) or contextual entities (e.g., user-defined methods).	[1, 13, 15, 18, 24, 27, 34, 45, 47, 51, 54, 57, 62, 67, 70, 71]	16	✔
Dependency Constraint	Mismatches and misuses of *dependencies* that are incompatible with the target context.	[3, 15, 24, 25, 38, 43, 45, 51, 67, 69, 71]	11	✔
Miscellaneous	Reuse code without providing *proper attribution* to its origin.	[2, 11, 35, 43, 48]	5	✗
	Vulnerabilities could spread through code reuse if not resolved.	[19, 46, 64]	3	✗

Interface Specification issues refer to mismatches between the interface of the reused code and its required specifications in the target context. Specifically, developers often reuse code from sources like Stack Overflow (SO), where snippets may lack proper method signatures or contain dangling code. Hence, they have to perform a specific type of adaptation, which is called "APIization" [36, 52], to encapsulate such code into well-formed methods with appropriate parameters and return statements. Besides, developers may also modify the parameter list by parameterizing some local variables or hard-coded values to improve reusability [67].

Functionality Customization issues refer to mismatches between the internal logic of reused code and its desired behavior in the target context. Prior studies find that code snippets from platforms like SO are often illustrative and may be semantically incomplete or not directly usable [51, 62, 70]. Hence, developers must frequently customize their functionality [58, 70]. Even when code is complete and executable, it may still require adaptations to satisfy specific needs, e.g., extracting relevant logic from code with tangled features [67]. This task covers adaptations of a wide range of code elements, including constants, types, conditionals, and API usage [47, 70].

Identifier Reference issues involve incorrect references to local or contextual entities and fall into two sub-categories: (1) misuse of bounded identifiers (already defined in the target context), and (2) misuse of free identifiers (instantiated within the current scope). For bounded identifiers, previous studies identify a common adaptation pattern called *Code Wiring*, where developers replace undeclared or conflicting variables with contextually appropriate ones [54, 67]. Similar adaptations are also applied to method identifiers to align with existing implementations. For free identifiers, adaptations often address variable misuse issues, which occur when integrating code with duplicate variable names or confusing variables of similar types and purposes [1, 34].

Dependency Constraint issues refer to mismatches and misuses of dependencies that are incompatible with the target context, including two main types: (1) missing or inconsistent imports: Code snippets from sources like SO often lack necessary imports or use incorrect library names or aliases, requiring developers to align the dependencies with the target context [25, 36, 45, 51] (2) version and compatibility issues: Rapid evolution of third-party libraries often leads to altered, replaced, or removed APIs. Incompatible or deprecated APIs can cause failures when reused code and the target context depend on different library versions [38]. Reusing outdated snippets or those containing API violations [43, 69] without appropriate adaptation may also introduces risks.

We also identify two **Miscellaneous** issues from the literature: (1) license violations, where code is reused without proper attribution to its original source [2, 35, 43, 48]; and (2) spread of vulnerabilities, where vulnerabilities in reused code may spread to the target system if not detected and resolved during adaptation [19, 46]. Although these issues are critically important in practice,

they fall beyond the scope of our study on *CtxBugs*. Licensing is primarily a legal concern rather than a semantic one, while vulnerabilities are generally context-agnostic. They reflect inherent flaws in the code, rather than bugs that emerges from integrating code into a new context. Therefore, we focus on the first four context-dependent issues in *CtxBug* generation and LLM evaluation, which constitute our ***adaptation tasks***.

3.2 Task-Specific Perturbation

In this step, we transform solution code from an existing benchmark into perturbed code templates for LLM-based variant generation. Specifically, we first derive ten perturbation rules based on our adaptation tasks. Subsequently, we parse the solution code into an abstract syntax tree (AST) using *tree-sitter-python*. By traversing the AST, we replace the target code elements with their corresponding placeholders according to our derived rules.

3.2.1 Selected Base Benchmark. To support *CtxBug* generation, a base benchmark should include the following components: (1) comprehensive requirement specification to describe the intended functionality of reused code, (2) rich code context to simulate the target environment, and (3) high-quality test suites to detect the constraint violation in the target context. To this end, we select a typical context-aware benchmark, ClassEval [16], as our base benchmark. Specifically, it is a class-level code generation benchmark widely adopted for evaluating LLMs in code generation [8] and also adaptation [71]. It includes detailed descriptions and test cases for each class and method, as well as rich intra-class interactions and diverse library usage. Besides, ClassEval also covers a broad range of programming topics that reflect real-world development scenarios. Following Zhang et al. [71], we focus on integrating function-level code within class-level contexts. It is worth noting that our framework is generalizable and can be applied to other benchmarks.

3.2.2 Perturbation Rules. A straightforward approach to generating *CtxBugs* is to mask entire methods and use LLMs to generate code from scratch without contextual constraints. However, this often results in implausible or functionally divergent outputs, e.g., implementing real mail-sending logic instead of a simulation. Besides, developers also prefer reusing near-correct code to avoid costly adaptations in practices [58, 67], hence our approach preserves the overall code structure and applies minimal perturbations to elements relevant to specific adaptation tasks. Guided by the adaptation taxonomy and syntactic rules from Zhang et al. [70], we design ten perturbation rules at two granularities: token-level and expression-level. To prevent LLMs from trivially inferring correct values from local context, e.g., deducing a boolean value from a contradictory condition or replicating a preceding arithmetic operation, we perturb all instances of a specific token type or all occurrences of a particular expression in a single perturbation. This allows a freer generation while preserving semantic plausibility. The detailed rules are described below.

Interface Specification. This task involves adapting the inputs (parameters) and outputs (return statements) of a method. As these elements are semantically distant and define distinct aspects of the interface, we perturb them individually to induce *CtxBugs*. It is worth noting that masking occurrences of a specific return statement may also allow LLMs to bypass the reasoning process, e.g., by inferring a *return True* from a visible *return False* statement. To this end, we mask all return statements simultaneously using explicit "*<RETURN>*" placeholders. This not only eliminates LLMs' shortcuts in reasoning but also constrains them to focus exclusively on generating return values.

> **[Rule 1 Parameter]:** *Replace the parameter list with a "<PARAMS>" placeholder.*
> **[Rule 2 Return Statement]:** *Replace all return statements with "<RETURN>" placeholders.*

Functionality Customization. This task focuses on adapting the internal code logic to aligned with the desired behavior in the context. Prior work identifies that such adaptations commonly involve constants, operations, conditionals, and API calls [70]. Implanting errors to these elements is also an established practice for creating program repair benchmarks, e.g., DebugBench [53], CodeEditorBench [22]. Thus, we perturb them individually to induce functionality-related *CtxBugs*.

Constants represent critical but context-specific values for code logic, e.g., array indices, API options. Since they are defined at the token level, we perturb all instances of a given constant type (e.g., all integers or strings) to induce *CtxBugs* that violate hard-coded contextual constraints.

> **[Rule 3 Constant]:** *Replace all constants of a specific type with "<INFILL>" placeholders.*

Operations are critical for controlling program behavior. To induce more *CtxBugs* in their semantics, we perform perturbations at both granularities. Firstly, we perturb all token-level operators as they encode the core logic of any operation. Secondly, we perturb each right-hand side of assignment statements individually to introduce higher-level semantic differences.

> **[Rule 4 Operator]:** *Replace all operators with "<INFILL>" placeholders.*
> **[Rule 5 Operation]:** *Replace each right-value operation with a "<INFILL>" placeholder.*

Since token-level perturbations above also cover conditionals and function calls, the following rules focus on their expression-level perturbations. Based on our criteria, we perturb their each occurrence individually. For library API calls, this rule focuses solely on their functional purposes, rather than on environmental constraints. Hence, we do not perturb other library usage simultaneously to restrict LLMs, which is different from **Rule 10**. Additionally, methods defined in the target contexts are excluded, as bounded identifier usage is specifically addressed in **Rule 8**.

> **[Rule 6 Conditional]:** *Replace each conditional with a "<INFILL>" placeholder.*
> **[Rule 7 Function Call]:** *Replace each function call with a '<INFILL>' placeholder.*

Identifier Reference. This task focuses on whether identifiers are correctly resolved within the target context. Due to the differences of bounded and free identifiers described in Section 3.1, we address them separately. For bounded identifiers, their references include attributes, methods and global identifiers. We perturb all their occurrences to prevent LLMs from accessing their contextual information. For free identifiers, we perturb all occurrences of local variable identifiers except their first declaration, in line with the variable misuse task defined in [1, 34]. Since this task specifically targets variables, we use explicit "<VAR>" placeholders to guide LLMs' generation.

> **[Rule 8 Bounded Identifier]:** *Replace all bounded identifiers with "<INFILL>" placeholders.*
> **[Rule 9 Free Identifier]:** *Replace all local variable identifiers with "<VAR>" placeholders.*

Dependency Constraint. This task focuses on resolving incompatible library usage arising from environmental factors. To force LLMs to generate feasible alternatives that may lead to *CtxBugs*, we perturb all attributes and API calls from third-party libraries simultaneously.

> **[Rule 10 Library Usage]:** *Replace all library attributes or API calls with "<INFILL>" placeholders.*

By applying above rules, we derive 3,527 perturbed code instances from 410 methods in the base benchmark. While some perturbed elements may be repetitive or overlapping, the goal of this step is to better induce *CtxBugs* rather than ensure uniqueness. Any duplicate variants generated by LLMs will be further filtered during the identification process (Section 3.4).

3.3 LLM-Based Variant Generation

In this step, we leverage LLMs to generate code variants containing potential *CtxBugs* from our perturbed code. However, since our base benchmark ClassEval was released in 2023, its inclusion in the latest LLMs' training corpora creates a risk of data leakage. A direct consequence is that LLMs may simply output the solution code even when provided only with the perturbed code and no target context. This bypasses their reasoning and results in no *CtxBugs* being generated.

To ensure that LLMs' generation adheres strictly to our requirements and is not influenced by prior exposure to the base benchmark, we employ code obfuscation techniques [29, 72] to mitigate the potential data leakage issue. Specifically, we rename all the identifiers in the perturbed code and their corresponding mentions in the requirement descriptions, e.g., $state \rightarrow var\_0$. This requires LLMs to complete the code according to requirement rather than relying on memorized data. This obfuscation step does not harm our purpose, as the obfuscated code is only textually different while preserving the original semantics, ensuring its validity during LLMs' inference phase.

Subsequently, we construct a set of code infilling prompts, each containing only the requirement description and a perturbed code snippet, to guide LLMs in generating code variants without exposing any contextual constraints. For the *Dependency Constraint* task, we include an additional instruction to explicitly restrict the use of correct libraries: "*However, you are not allowed to use the following libraries: [LIB_DEPS]*" where *LIB_DEPS* denotes all third-party libraries used in the solution. To ensure the robustness of our framework, we employ four different LLMs for *CtxBug* generation, which are the same models used in our subsequent evaluation (Section 4.2).

3.4 *CtxBug* Identification

This step employs a hybrid approach to identify *CtxBugs* from LLM-generated code variants. Since *CtxBugs* should be contextually incorrect, we first deobfuscate all variants to restore original identifier names and test them within the target context. However, test execution alone is insufficient, because it cannot determine whether a failure stems from an incorrect completion at the perturbed location or from irrelevant changes introduced elsewhere by LLMs. Relying

Fig. 5. Details of *CtxBug* identification. The numbers represent the remaining cases after each filtering step.

solely on execution could result in *CtxBugs* that do not align with the intended adaptation task.

To address this, we complement execution-based identification with syntactic differencing. This technique ensures that the generated code differs from the solution only at the intentionally perturbed locations. The algorithm for this process is described as follows. We first parse both the LLM-generated variant and solution code into ASTs. We then use GumTree [17], a widely-used code differencing tool, to obtain AST node matches and an edit script between two ASTs. For each perturbed node in the solution code (obtained in Section 3.2), we identify its corresponding matching node in the variant. A *CtxBug* is considered valid if either: (1) a matching node is found and its textual content differs from the original, or (2) the edit script indicates that the perturbed node is deleted. If all perturbed nodes are found to be textually identical in the variant, the case is filtered out as it does not contain a *CtxBug*. This process confirms that any functional incorrectness arises from a genuine *CtxBug* at the intended location, rather than from arbitrary or extraneous modifications.

However, our hybrid identification approach cannot handle empty outputs. Furthermore, different perturbation rules may generate identical code variants containing the same *CtxBug*. To prevent
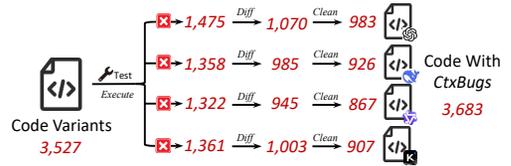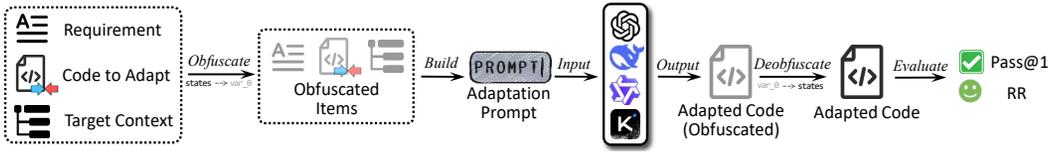
Fig. 6. The evaluation process for LLMs' adaptation.

these issues from affecting our evaluation, we perform a data cleaning step to filter out repetitive or empty outputs. Detailed statistics for the identification process for each LLM are provided in Fig. 5.

To ensure quality, we further conduct manual validation for 348 snippets containing *CtxBugs* by employing a stratified sampling (with 95% CI and MOE=5%). Two annotators independently evaluate each sample against our criteria for *CtxBugs*: (1) whether the code is plausible regarding the requirement, (2) whether it is incorrect in the target context, and (3) whether generated *CtxBugs* are located at the intended locations. Since criterion 2 is validated by test execution, our validation focuses on criteria 1 and 3. For any instance failing them, the annotators document the specific reason. Our validation results demonstrate the high quality of the benchmark. 90.80% of the sampled *CtxBugs* satisfy all criteria. Only 30 cases (8.62%) contain isolated bugs and 4 cases (1.15%) have the location issue. The inter-annotator agreement is nearly perfect, with Cohen's Kappa values [10] of 0.866 (criterion 1) and 0.886 (criterion 3). The statistics of our benchmark (*w/ CtxBugs*) are presented in Table 3. The full benchmark and our annotated data are available in the replication package.

4 Experimental Setup

This section describes our experimental setup to evaluate the effectiveness of LLMs in resolving *CtxBugs*. Specifically, we structure our evaluation through the following three research questions.

RQ1: How effective are LLMs at resolving *CtxBugs* during code adaptation? This RQ aims to benchmark and compare the performance of state-of-the-art LLMs in resolving *CtxBugs* generated from *CtxBugGen*. Besides, it also investigates whether their effectiveness is impacted by the source model that introduces *CtxBugs*.

RQ2: To what extent do *CtxBugs* impact code adaptation performance of LLMs? This RQ compares LLMs' performance across three settings: adapting code with *CtxBugs*, code without *CtxBugs* and code with isolated bugs. The comparison aims to quantify the impact of *CtxBugs* and highlight their differences from isolated bugs during adaptation.

RQ3: What are the limitations of LLMs in resolving *CtxBugs* during adaptation? This RQ involves a qualitative analysis of LLMs' failures when resolving *CtxBugs*. We investigate their erroneous handling actions and common *CtxBug* types in different adaptation tasks. The findings aim to provide insights for improving LLMs' capabilities and their practical use in code adaptation.

Our evaluation process is illustrated in Fig. 6. To mitigate data leakage issue, we employ a class-level code obfuscation for LLMs' inputs. Then we evaluate four state-of-the-art LLMs with a designed adaptation prompt through execution and syntactic metrics. The details are as follows.

4.1 Baselines

To measure the impact of *CtxBugs* on LLMs' adaptation, we design two baselines: (1) *Code Without CtxBugs* and (2) *Code With Isolated Bugs (IsoBugs)*, whose construction processes are illustrated in Fig. 7. For *Code Without CtxBugs*, we directly replace all *CtxBug* locations with "*<INFILL>*" placeholders. This represents the setting without distractions from their semantics, forcing LLMs to merely reason with context. This approach aligns with the established program repair practices [59]. However, the placeholders may leak positional information by signaling to LLMs which locations
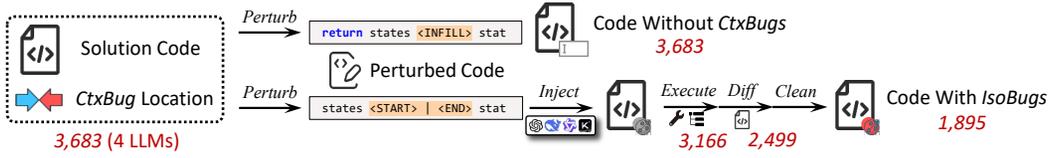
Fig. 7. The data construction process for baselines.

require completion. To address this issue, we introduce a second baseline, *Code With IsoBugs*. Its construction leverages LLMs' ability to implant bugs [22, 53]. Specifically, we first enclose each code span where *CtxBug* is located within "*<START>*" and "*<END>*" tokens in the solution code. This design enables a direct paired comparison of bug resolution performance at identical locations. We then prompt LLMs to introduce isolated bugs within the marked span. The generated candidates undergo the same identification process as *CtxBugs*, including test execution, syntactic differencing, and data cleaning, as their only difference lies in local correctness. To ensure quality, we also conduct manual validation by sampling 320 snippets (with 95% CI, MOE=5%). Two annotators independently evaluate each sample against two criteria: (1) whether the code contains a bug in isolation, and (2) whether generated *IsoBugs* are located at the target location. Our validation results show that 92.81% (297/320) of the generated *IsoBugs* are both locally incorrect and correctly located, with Cohen's Kappa values [10] of 0.832 (criterion 1) and 1.0 (criterion 2), respectively, indicating nearly perfect agreement. The detailed statistics are listed in Table 3. The baseline benchmarks and our annotated data are available in the replication package.

Table 3. The statistics of generated benchmarks. *Code with* and *without CtxBugs* have identical sizes, as they differ only at the perturbed locations. *Code with IsoBugs* undergoes an extra generation and filtration process and thus has a smaller size. Values in parentheses indicate the total number of bugs for each category.

Model	Setting	Interface	Functionality	Identifier	Dependency	All Tasks
GPT-4o	w/(o) CtxBugs	**259** (320)	**379** (497)	**297** (835)	**48** (109)	**983** (1,761)
	w/ IsoBugs	**112** (161)	**291** (389)	**208** (583)	**45** (106)	**656** (1,239)
DS-V3	w/(o) CtxBugs	**242** (286)	**366** (491)	**271** (785)	**47** (125)	**926** (1,687)
	w/ IsoBugs	**100** (133)	**238** (339)	**132** (368)	**35** (103)	**505** (943)
Qwen3	w/(o) CtxBugs	**234** (278)	**350** (506)	**238** (644)	**45** (128)	**867** (1,556)
	w/ IsoBugs	**118** (150)	**230** (348)	**101** (247)	**24** (72)	**473** (817)
Kimi-K2	w/(o) CtxBugs	**229** (280)	**354** (466)	**280** (793)	**44** (122)	**907** (1,661)
	w/ IsoBugs	**39** (80)	**88** (165)	**109** (324)	**25** (84)	**261** (653)

4.2 Studied LLMs

To evaluate the performance of state-of-the-art LLMs in resolving *CtxBugs*, we include four leading LLMs in our study. Since code adaptation requires both advanced natural language understanding and coding capabilities, our selection encompasses three general-purpose LLMs, including GPT-4o (2024-11-20) [39], DeepSeek-V3 (2025-03-24) [14] and Kimi-K2 (2025-07-11) [50], based on their exceptional results across various benchmarks, as well as one specialized code model, Qwen3-Coder-Plus (2025-07-22) [42], recognized for its extraordinary performance in coding tasks. An additional selection criterion is the accessibility of token generation probabilities, which allows us to analyze LLMs' confidence during generation.

4.3 Prompt Design

In line with the effective prompt design established by Zhang et al. [71], our adaptation prompt comprises three parts: (1) the user requirement, (2) the reused code for adaptation, and (3) the target class-level context. To simulate a realistic adaptation scenario, the targeted method itself and its callers are removed from this context. This structure provides LLMs with the essential contextual information required for successful code adaptation. Prompt templates and concrete examples are available in our replication package.

4.4 Evaluation Metrics

Simiar to our *CtxBug* identification, we adopt both execution-based and syntactic-based metrics to evaluate the effectiveness of LLMs in resolving *CtxBugs*.

Pass@k metric estimates the likelihood that an LLM produces a correct solution within k attempts based on unit test execution. In our study, we set $k = 1$ to assess the correctness of the LLM's output. This selection is motivated by both the high computational cost of generating multiple solutions and its alignment with developer behavior, as studies show practitioners typically consider only the first suggestion from AI coding assistants [4, 49].

Resolution Rate (RR) measures whether each *CtxBug* has been correctly resolved by performing an exact match against the reference solution. For each AST node in the solution code at the *CtxBug* location, we find its corresponding node in the LLM-generated code using GumTree. A *CtxBug* is considered resolved only if a matching node is found and its textual content is identical to the solution. This metric is applicable for evaluating LLMs' adaptation of code without *CtxBugs* or with *Isobugs*, as their locations for the required adaptations are consistent.

4.5 Implementation Details

LLMs' different randomness settings can lead to different outcomes [40]. In our evaluation, we are interested in the most preferred solution generated by the LLMs without any scaling. Therefore, we set the temperature to 0 for all LLMs to eliminate the randomness. This configuration is also an established best practice when using the Pass@1 to measure only one generated solution [9, 33, 49]. All experiments are conducted with four GeForce RTX 4090-24G GPUs on Ubuntu 22.04.

5 Results

5.1 RQ1: Effectiveness of LLMs

Table 4 presents the performance of each LLM in resolving *CtxBugs* on our benchmark constructed by *CtxBugGen*. To ensure a fair evaluation, each LLM is tested on *CtxBugs* generated by all four models. We also report their overall average performance weighted by benchmark size. Kimi-K2 achieves the best performance among all LLMs, with a Pass@1 of 55.93% and an RR of 52.47% respectively. Across four adaptation tasks, LLMs perform better at resolving *Interface Specification* and *Identifier Reference CtxBugs*. This is likely because these

Fig. 8. Relative *Pass@1* ranking of LLMs.

tasks are more syntax-aware, allowing models to rely on textual patterns. In contrast, *Functionality Customization* proves most challenging, with all LLMs achieving under 40% Pass@1 and under 30% RR. The reason is that this task requires deep understanding of code semantics and accurate identification of contextual constraints in both code and natural language. Regarding *CtxBugs* in the *Dependency Constraint* task, Qwen3-Coder-Plus exhibits a particular strength, achieving over
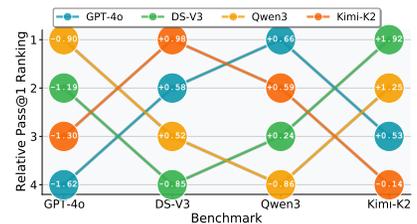
Table 4. LLMs' performance in *CtxBug* resolution. **Model-G** and **Model-T** represent the generator model and the tested model respectively.

Model-G	Model-T	Interface		Functionality		Identifier		Dependency		All Tasks	
		Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR
GPT-4o	GPT-4o	67.18	67.81	33.77	21.93	67.34	**60.96**	43.75	44.04	53.20	**50.14**
	DS-V3	66.02	67.81	34.04	23.94	66.33	58.68	47.92	41.28	52.90	49.46
	Qwen3	**68.73**	**69.69**	35.36	24.75	68.35	57.84	45.83	41.28	**54.63**	49.63
	Kimi-K2	66.80	67.50	**35.36**	**25.35**	**69.02**	58.44	**52.08**	**44.95**	**54.63**	49.91
DS-V3	GPT-4o	68.18	65.73	**38.25**	**28.51**	69.00	**62.93**	44.68	56.00	55.40	52.87
	DS-V3	66.94	66.43	35.52	27.70	65.31	61.40	51.06	56.80	53.24	52.10
	Qwen3	68.60	65.03	36.34	28.31	70.48	61.78	**61.70**	**61.60**	56.05	52.58
	Kimi-K2	**70.25**	**67.83**	37.70	28.31	**70.85**	62.42	57.45	56.80	**56.91**	**52.99**
Qwen3	GPT-4o	70.09	65.47	37.43	29.05	**68.49**	62.89	51.11	57.81	55.48	**51.93**
	DS-V3	70.51	66.19	35.43	29.84	66.39	59.78	53.33	54.69	54.33	50.77
	Qwen3	65.81	62.59	**39.43**	**31.62**	66.39	58.07	53.33	**60.94**	54.67	50.51
	Kimi-K2	**70.94**	**66.91**	38.86	29.84	68.07	61.18	**57.78**	57.81	**56.52**	51.74
Kimi-K2	GPT-4o	68.12	67.50	33.62	22.32	**72.86**	68.98	55.27	53.28	55.35	54.49
	DS-V3	**71.18**	68.21	33.62	22.75	72.50	68.60	52.27	54.10	56.01	54.61
	Qwen3	68.12	66.43	**35.59**	**24.03**	**72.86**	**69.23**	**65.91**	**60.66**	**56.78**	**55.45**
	Kimi-K2	69.87	**69.64**	33.90	22.75	**72.86**	68.98	50.00	58.20	55.79	55.33
Average	GPT-4o	68.36	66.67	35.75	25.51	69.43	**63.95**	48.54	53.10	54.82	52.33
	DS-V3	68.57	67.18	34.65	26.12	67.68	62.18	51.09	52.07	54.09	51.72
	Qwen3	67.84	66.07	**36.65**	**27.24**	69.61	61.85	**56.52**	**56.61**	55.53	52.03
	Kimi-K2	**69.40**	**67.95**	36.44	26.63	**70.26**	62.77	54.35	54.75	**55.93**	**52.47**

60% on both Pass@1 and RR for benchmarks generated by DeepSeek-V3 and Kimi-K2. Besides, the gap between Pass@1 and RR is more pronounced for *Functionality Customization* and *Identifier Reference*. Since their cases may contain multiple interdependent *CtxBugs* (introduced by Rule 3, 4, 8 & 9), failing to identify the complete set prevents LLMs from resolving any of them, leading to a severe drop in RR.

Besides, we evaluate differences of LLMs' performance across *CtxBugs* generated by different LLMs. Considering their inherent differences in capabilities, we compute each LLM's relative Pass@1 and its corresponding ranking on each benchmark, as shown in Fig. 8. A key finding is that all four LLMs rank last on the data generated by themselves, which is statistically significant ($p = 0.0039$ via Fisher's Sign Test [23]). This suggests that LLMs perceive their own generated tokens as more plausible, making them more susceptible to such *CtxBugs*. We also observe a symmetrical performance pattern across model pairs, e.g., GPT-4o performs best on data generated by Qwen3, and vice versa. This implies that their biases on semantic correctness are also symmetrical.

> **Finding 1:** LLMs can perform correct adaptations in up to 55.93% cases and resolve at most 52.47% *CtxBugs*. *CtxBugs* in *Functionality Customization* are the most difficult to resolve, with a RR of 27.24%. Besides, LLMs are less effective in resolving *CtxBugs* generated by themselves.

5.2 RQ2: Impact of *CtxBugs*

Table 5 compares LLMs' performance on adaptation with or without *CtxBugs*. The presence of *CtxBugs* causes an average decrease ratio of 23.16% in Pass@1 and 27.53% in RR, suggesting that LLMs are easily misled by *CtxBugs* and perform worse than completing code from scratch. Across task types, *Functionality Customization* shows the most severe degradation (34.99% in Pass@1,

Table 5. Comparison of LLMs' Performance in Adapting *Code w/ CtxBugs* and *Code w/o CtxBugs*.

Model	Setting	Interface		Functionality		Identifier		Dependency		All Tasks	
		Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR
GPT-4o	*w/ CtxBugs*	67.18	67.81	33.77	21.93	67.34	60.96	43.75	44.04	53.20	50.14
	w/o CtxBugs	74.52	76.56	53.03	46.68	83.50	84.43	68.75	67.89	68.67	71.32
	Relative %	9.85↓	11.43↓	36.32⇓	53.02⇓	19.35↓	27.80⇓	36.36⇓	35.13⇓	22.53↓	29.70⇓
DS-V3	*w/ CtxBugs*	66.94	66.43	35.52	27.70	65.31	61.40	51.06	56.80	53.24	52.10
	w/o CtxBugs	77.27	73.43	53.83	50.71	88.93	84.20	70.21	72.00	71.06	71.72
	Relative %	13.37↓	9.53↓	34.01⇓	45.38⇓	26.56⇓	27.08⇓	27.28⇓	21.11↓	25.08⇓	27.36⇓
Qwen3	*w/ CtxBugs*	65.81	62.59	39.43	31.62	66.39	58.07	53.33	60.94	54.67	50.51
	w/o CtxBugs	77.78	72.66	58.29	54.94	86.55	82.76	68.89	78.91	71.86	71.59
	Relative %	15.39↓	13.86↓	32.36⇓	42.45⇓	23.29↓	29.83⇓	22.59↓	22.77↓	23.92↓	29.45⇓
Kimi-K2	*w/ CtxBugs*	69.87	69.64	33.90	22.75	72.86	68.98	50.00	58.20	55.79	55.33
	w/o CtxBugs	75.55	73.57	53.95	47.00	88.57	86.25	68.18	77.05	70.78	72.43
	Relative %	7.52↓	5.34↓	37.16⇓	51.60⇓	17.74↓	20.02⇓	26.66⇓	24.46↓	21.18↓	23.61↓
Average	*Relative %*	11.52↓	10.08↓	34.99⇓	48.04⇓	21.60↓	26.03⇓	28.35⇓	25.55⇓	23.16↓	27.53⇓

48.04% in RR), while *Interface Specification* is less affected. This is largely because that functionality-related *CtxBugs* require deeper semantic reasoning, whereas interface violations often provide clearer syntactic clues for resolution.

Table 6. Comparison of LLMs' Performance in Adapting *Code w/ CtxBugs* and *Code w/ IsoBugs*.

Model	Setting	Interface		Functionality		Identifier		Dependency		All Tasks	
		Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR	Pass@1	RR
GPT-4o	*w/ CtxBugs*	59.82	49.69	37.46	23.65	61.06	57.80	46.67	45.28	49.39	44.96
	w/ IsoBugs	60.71	58.39	47.77	46.53	65.87	74.27	48.89	64.15	55.79	62.63
	Relative %	1.47↓	14.90↓	21.58↓	49.17⇓	7.30↓	22.18↓	4.54↓	29.42⇓	11.47↓	28.21⇓
DS-V3	*w/ CtxBugs*	59.00	57.89	33.61	25.96	59.85	55.98	51.43	61.17	46.73	46.02
	w/ IsoBugs	55.00	57.14	37.82	41.59	65.91	75.27	68.57	81.55	50.69	61.29
	Relative %	7.27↑	1.31↑	11.13↓	37.58⇓	9.19↓	25.63⇓	25.00↓	24.99↓	7.81↓	24.91↓
Qwen3	*w/ CtxBugs*	55.93	51.33	38.70	29.60	61.39	57.09	41.67	48.61	47.99	43.57
	w/ IsoBugs	55.08	56.67	44.78	50.57	72.28	78.13	41.67	65.28	53.07	61.32
	Relative %	1.54↑	9.42↓	13.58↓	41.47⇓	15.07↓	26.93⇓	0.00=	25.54⇓	9.57↓	28.95⇓
Kimi-K2	*w/ CtxBugs*	48.72	61.25	30.68	21.21	71.56	69.44	48.00	60.71	52.11	55.13
	w/ IsoBugs	51.28	65.00	40.91	55.15	71.56	82.72	56.00	71.43	56.70	72.13
	Relative %	4.99↓	5.77↓	25.01⇓	61.54⇓	0.00=	16.05↓	14.29↓	15.01↓	8.10↓	23.57↓
Average	*Relative %*	1.49↑	7.82↓	16.83↓	45.49⇓	7.73↓	22.48↓	11.13↓	24.09↓	9.56↓	26.69⇓

To ensure a fair comparison between LLMs' performance in resolving *CtxBugs* and *IsoBugs*, we only sample *CtxBug* instances that also appear in the *IsoBug* benchmark. Results are shown in Table 6. The average decrease of Pass@1 is 9.56%, while RR still drops by 26.69%. This indicates that LLMs are better at fixing individual *IsoBugs* than *CtxBugs*, while their resolution for both remain incomplete. The most significant gap occurs in *Functionality Customization*, with an average decrease of 16.83%. Interestingly, *IsoBugs* in interface are found as challenging as *CtxBugs*, with even lower Pass@1. The reason is that LLMs often introduce unused default parameters as *IsoBugs*, which are easily overlooked since they lack functional effects. We also observe performance differences across LLMs. For instance, DeepSeek-V3 struggles with dependency-related *CtxBugs*, while Kimi-K2 shows the smallest gap in identifiers.
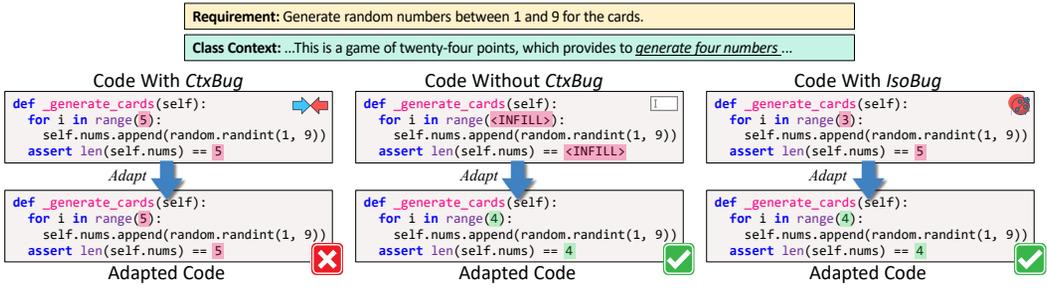
Fig. 9. A failure example of adapting code with *CtxBugs*.

We further illustrate these scenarios with the example in Fig. 9. The task requires adapting a random number generation method for a Twenty-Four Point Game, which should generates exactly four numbers. Our framework introduces a *CtxBug* that generates five numbers and includes an assertion for a length of five. In contrast, the *CtxBug*-free code contains two placeholders to be filled, and the *IsoBug* version incorrectly produces three numbers while asserting a length of five, creating an inherent inconsistency. The required adaptations are of similar complexity (i.e., two integers), ensuring a fair comparison. In this case, LLMs failed to resolve the *CtxBug* but succeeded in the other two scenarios. This result indicates that the inconsistency of *IsoBug* is locally detectable within the method, while the *CtxBug* is locally correct but violates the game's requirement. LLMs still lack cross-context reasoning to identify such semantic mismatch during adaptation.

Finding 2: LLMs' adaptation performance is significantly impacted by *CtxBugs*, with a degradation of 23.16% and 27.53% in Pass@1 and RR respectively. Besides, *CtxBugs* are more difficult for LLMs to resolve than isolated errors.


5.3 RQ3: Failure Analysis

We employ stratified sampling to select erroneous adaptations that failed both test execution and syntactic differencing checks. The sample is drawn from the outputs of four evaluated LLMs across the whole benchmark (with 95% CI, 5% MOE). This process yields a set of 1,250 failed instances for manual analysis. Four annotators are invited to characterize each output from two dimensions: (1) the LLM's handling action for the *CtxBug* and (2) the specific type of *CtxBug* that caused failure in adaptation. The annotated data are available in the replication package.

For the first dimension, we enhance the taxonomy from Zhang et al. [71] to classify LLMs' handling actions into four categories, including:

- **Overlooked**: The LLM failed to identify the *CtxBug* and replicated it in its output.
- **Invalid**: The LLM identified the *CtxBug* but applied an incorrect adaptation.
- **Unexpected**: The LLM resolved the *CtxBug* but also introduced new bugs elsewhere.
- **Others**: LLM's output is unrelated to the *CtxBug* resolution, e.g., empty responses, reimplementations or code with major deletions.

Table 7. The Average Token Probability (ATP) of LLMs' Overlooked *CtxBugs*.

Model	GPT-4o	DeepSeek-V3	Qwen3-Coder-Plus	Kimi-K2	**Average**
ATP	0.982	0.988	0.978	0.984	**0.983**

(a) Distribution of LLMs' handling actions

(b) *Unexpected* handling action of *CtxBug*
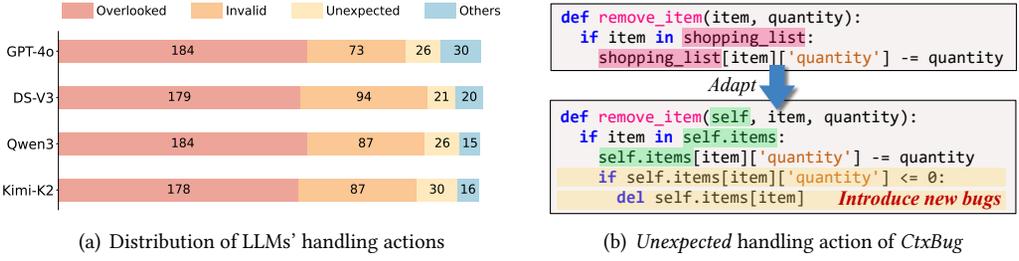
Fig. 10. LLMs' erroneous handling actions for *CtxBugs*.

The distribution of LLMs' handling actions is summarized in Fig. 10(a). In nearly 60% of cases, LLMs completely overlooked *CtxBugs*, directly replicating the buggy code in their output. To further investigate this replication behavior, we calculate the Average Token Probability (ATP) for LLM-generated token sequences of the *CtxBugs*. ATP measures the LLMs' intrinsic confidence in their generations [49]. We observe that LLMs all exhibit high confidence with an average ATP of 0.983, indicating they consider the bug-inducing tokens to be correct. In up to 30% of cases, LLMs performed adaptations that are insufficient to fully resolve the *CtxBugs*. Besides, in up to 9.65% of cases, LLMs' adaptations to fix the original *CtxBug* resulted in introducing new bugs. As shown in Fig. 10(b), an LLM correctly replaced an undefined variable with a class attribute but then erroneously removed a key when its value is non-positive, which is a unspecified behavior. This suggests that LLMs sometimes inappropriately rely on patterns from their pretraining knowledge (e.g., a rule for cleaning up zero-value items) when the target context provides no explicit guidance.

Table 8. The distribution of *CtxBug* types that LLMs failed to resolve in different adaptation tasks.

Task	Bug Type	Associated Rules	Model				Average
			GPT-4o	DS-V3	Qwen3	Kimi-K2	
Interface	Incorrect Parameter	Rule 1	15 (22.73%)	28 (32.56%)	30 (30.93%)	38 (40.86%)	111 (32.46%)
	Incorrect Return	Rule 2	51 (77.27%)	58 (67.44%)	67 (69.07%)	55 (59.14%)	231 (67.54%)
Functionality	Incorrect Condition	Rule 3, 4, 6	51 (29.31%)	58 (32.22%)	56 (30.43%)	52 (29.55%)	217 (30.39%)
	Incorrect Argument	Rule 3, 7	25 (14.37%)	32 (17.78%)	23 (12.50%)	25 (14.20%)	105 (14.71%)
	Incorrect Initialization	Rule 3, 5	23 (13.22%)	38 (21.11%)	31 (16.85%)	26 (14.77%)	118 (16.53%)
	Incorrect Calculation	Rule 3, 4, 5	21 (12.07%)	21 (11.67%)	34 (18.48%)	27 (15.34%)	103 (14.43%)
	Incorrect Data Operation	Rule 5, 7	29 (16.67%)	19 (10.56%)	31 (16.85%)	35 (19.89%)	114 (15.97%)
	Incorrect Type Operation	Rule 5, 7	11 (6.32%)	4 (2.22%)	1 (0.54%)	6 (3.41%)	22 (3.08%)
	Incorrect Indexing	Rule 3	14 (8.05%)	8 (4.44%)	8 (4.35%)	5 (2.84%)	35 (4.90%)
Identifier	Misassigned Identifier	Rule 9	10 (17.54%)	9 (15.00%)	7 (14.00%)	7 (12.07%)	33 (14.67%)
	Missing Context	Rule 8	36 (63.16%)	29 (48.33%)	28 (56.00%)	28 (48.28%)	121 (53.78%)
	Context Misuse	Rule 8	11 (19.30%)	22 (36.67%)	15 (30.00%)	23 (39.66%)	71 (31.56%)
Dependency	Missing Library	Rule 10	18 (60.00%)	18 (58.06%)	14 (56.00%)	9 (64.29%)	59 (59.00%)
	Library Misuse	Rule 10	12 (40.00%)	13 (41.94%)	11 (44.00%)	5 (35.71%)	41 (41.00%)

Table 8 summarizes the type distribution of *CtxBugs* that LLMs failed to resolve. Since *CtxBugs* are generated for specific adaptation tasks, our taxonomy and analysis are correspondingly scoped to these tasks individually. The detailed definition of each type is available in our replication package. *Incorrect Return* is the most prevalent *CtxBug* type in the *Interface Specification* task. LLMs often failed to identify the inconsistencies between the outputs of the reused code and its context. A representative example is shown in Fig.11(a), where Kimi-K2 overlooked the missing semicolon in a return value, even though the context provided clear examples of the correct formatting in sibling methods. *CtxBugs* in conditions are the most challenging to resolve in the *Functionality*

Customization, with a proportion of about 30% across all LLMs. As shown in Fig.11(b), Qwen3 accepted the plausible but contextually incomplete condition (e.g., checking if a standard deviation is zero) instead of performing the necessary cross-context reasoning on the callee's output, whose edge case returns *None*. For identifiers, LLMs often ignore the pre-existing implementations to resolve *CtxBugs*. For example, in a Snake game (Fig.11(c)), Kimi-K2 failed to use the "*eat_food()*" method in the context (which handles the snake length and score increment) and instead copied the *CtxBug* that merely added coordinates to the snake's position. Finally, for dependency-related bugs, LLMs frequently missed using a required library or misused its API. Figure11(d) illustrates a *CtxBug* where GPT-4o failed to use the provided library *json* in its implementation. The provided context "*load_cookies*" reads persisted cookie data in the JSON format with the required library. However, GPT-4o overlooked this dependency and maintained the incompatible data serialization in a line-by-line string format instead.
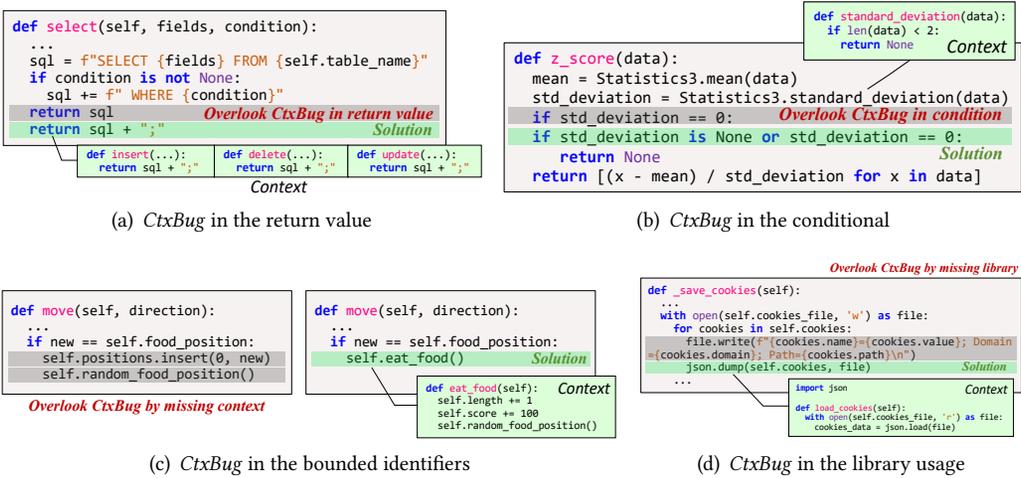

(a) *CtxBug* in the return value

(b) *CtxBug* in the conditional

(c) *CtxBug* in the bounded identifiers

(d) *CtxBug* in the library usage

Fig. 11. Examples of LLMs' failures in adapting code with *CtxBugs*.

To investigate why LLMs overlooked *CtxBugs*, we further prompt them to explain their reasoning alongside their code adaptations[1]. We manually analyze the outputs of 50 samples to determine whether the failures originate from LLMs' inability to identify *CtxBugs* or their inability to perform the repair. Our results show that in 98% of cases, LLMs failed to identify *CtxBugs* entirely. For instance, in the case shown in Fig.11(b), the LLM claimed "*No adaptations have been made because the provided code is correct...*" In the remaining one case, LLMs successfully identified the *CtxBug* and performed correct adaptations, likely due to the Chain-of-Thought (CoT) effects [56] induced by the explanation.

> **Finding 3:** In nearly 60% of cases, LLMs overlook *CtxBugs* and replicate them as correct code. Their weaknesses lie in resolving *CtxBugs* related to return values, conditionals, and the incorporation of pre-existing implementations.

---

[1]The explanation prompt is provided in our replication package.

6 Implication

Mind *CtxBugs* and Consider Cross-Context Semantics in Code Reuse. While fixing standalone bugs has long been a focus in software engineering [60], our study emphasizes that resolving *CtxBugs* is equally critical and often more challenging due to their semantic subtleties. Developers should carefully examine whether reused code aligns with contextual constraints. To support this need, we propose *CtxBugGen* and a dedicated benchmark for systematically evaluating *CtxBug* resolution, providing a foundation for future research and tool development in this domain.

Double Check When Using LLMs for Code Adaptation. Our findings reveal that LLMs often overlook and replicate *CtxBugs*, generating code that is locally plausible but contextually incompatible. Practitioners using LLMs for adaptation should therefore employ rigorous quality assurance, to validate their outputs. It is particularly necessary for functionality interacting with contextual elements, e.g., pre-existing methods, to avoid introducing subtle semantic errors.

Develop Methods to Enhance Cross-Context Reasoning. Our study suggests LLMs currently lack of robust cross-context reasoning, i.e., they overly rely on local code correctness and fail to consider broader contextual constraints. Future work should explore: (1) *Fine-tuning* on cross-context tasks like adaptation; (2) *Prompting strategies* that explicitly guide contextual reasoning through chain-of-thoughts or self-reflection; and (3) *Hybrid techniques* combining LLMs with static analysis to enhance their contextual awareness. Advancing these areas is essential for LLMs application in code adaptation tasks.

7 Threats to Validity

Threats to Internal Validity. Threats may arise from our adaptation task selection and benchmark construction. Although we derive four key tasks from literature review, they may not cover all adaptation scenarios (e.g., optimizations for efficiency). Future work should evaluate LLMs on a broader range. For benchmark construction, we use LLMs to generate both *CtxBugs* and *IsoBugs*, which may affect data quality (e.g., introduces randomness or model-specific noises and may not represent their real-world complexity). To mitigate this, we conduct a rigorous identification combining test execution and syntactic differencing. Our manual validation further confirms the quality of LLM-implanted bugs, which achieves high validity and strong inter-annotator agreements. Additionally, Table 9 reports LLMs' performance on our manually validated *CtxBugs* subset. These results align closely with the full benchmark, confirming the reliability of our findings. Regarding the potential data leakage risk, we address it by applying code obfuscation techniques during benchmark construction and evaluation.

Table 9. LLMs' performance on the full LLM-generated benchmark and our validated *CtxBugs*.

Data	Model-T	Interface		Functionality		Identifier		Dependency		All Tasks	
		Pass@1	*RR*	*Pass@1*	*RR*	*Pass@1*	*RR*	*Pass@1*	*RR*	*Pass@1*	*RR*
Full	GPT-4o	68.36	66.67	35.75	25.51	69.43	**63.95**	48.54	53.10	54.82	52.33
	DS-V3	68.57	67.18	34.65	26.12	67.68	62.18	51.09	52.07	54.09	51.72
	Qwen3	67.84	66.07	**36.65**	**27.24**	69.61	61.85	**56.52**	**56.61**	55.53	52.03
	Kimi-K2	**69.40**	**67.95**	36.44	26.63	**70.26**	62.77	54.35	54.75	**55.93**	**52.47**
Validated	GPT-4o	72.53	71.19	34.78	26.25	**75.53**	**66.89**	50.00	45.45	**58.54**	**55.90**
	DS-V3	67.03	66.95	**37.39**	**29.38**	74.47	66.22	50.00	48.48	57.59	55.74
	Qwen3	68.13	67.80	36.52	28.12	73.40	60.87	**56.25**	**51.52**	57.59	53.11
	Kimi-K2	**73.63**	**72.03**	33.91	28.12	**75.53**	66.56	37.50	36.36	57.91	**55.90**

Threats to External Validity. The generalizability of our findings may be limited by the selection of LLMs and base benchmarks. While we evaluate four state-of-the-art LLMs based on

their performance, accessibility, and specialty to highlight common bottlenecks, our results may not apply to other untest or emerging models. Another limitation arises from our reliance on ClassEval as the base benchmark. Since ClassEval is restricted to Python and focuses exclusively on class-level contexts, our findings may not generalize to other languages (e.g., Java, Rust) or broader contexts (e.g., system-level adaptations). However, our proposed *CtxBugGen* framework is designed to be language-agnostic. It can be adapted to these scenarios by integrating language-specific syntactic parsing and perturbation rules. In our future work, we plan to extend *CtxBugGen* to multi-task, multi-language and multi-context scenarios.

8 Conclusion

This paper proposes *CtxBugGen*, the first framework for generating *CtxBugs* to evaluate LLMs on code adaptation tasks. Our empirical study with four state-of-the-art LLMs reveals their unsatisfactory performance in resolving *CtxBugs*. A key finding is that LLMs frequently overlook these bugs and replicate them in their outputs, suggesting an over-reliance on local code correctness while ignoring contextual constraints. Future work should extend beyond training and evaluating LLMs on standalone coding tasks and prioritize enhancing their cross-context reasoning abilities, which is critical for their reliable application in real-world code reuse and adaptation.

9 Data Availability

The replication package including all the data (benchmarks, human annotation for literature review, benchmark validation and failure analysis), code (for benchmark construction and empirical evaluation) and related materials (prompts, intermediate results, etc.) is available here[2]. The detailed guides are included in the README.md file under each folder.

Acknowledgments

References

[1] Miltiadis Allamanis and Marc Brockschmidt. 2017. SmartPaste: Learning to Adapt Source Code. http://arxiv.org/abs/1705.07867 arXiv:1705.07867 [cs].

[2] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. 2017. Stack Overflow: A code laundering platform?. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, Klagenfurt, Austria, 283–293. https://doi.org/10.1109/SANER.2017.7884629

[3] Ameer Armaly, Casey Ferris, and Collin McMillan. 2014. Reusable execution replay: execution record and replay for source code reuse. In *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, Hyderabad India, 480–483. https://doi.org/10.1145/2591062.2591130

[4] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85–111. https://doi.org/10.1145/3586030 Publisher: Association for Computing Machinery (ACM).

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (2009), 18–24. https://doi.org/10.1109/MS.2009.147

[6] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa arXiv:https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp063oa

---

[2]https://github.com/ztwater/CtxBugGen.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. http://arxiv.org/abs/2005.14165 arXiv:2005.14165 [cs].

[8] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1869–1881. https://doi.org/10.1109/ICSE55347.2025.00012

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[10] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20 (1960), 37 − 46.

[11] Gaia Colombo, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2025. On the Possibility of Breaking Copyleft Licenses When Reusing Code Generated by ChatGPT. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. 1–12. https://doi.org/10.1109/ICPC66645.2025.00060 ISSN: 2643-7171.

[12] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. 2008. Jigsaw: a tool for the small-scale reuse of source code. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*. ACM Press, Leipzig, Germany, 933. https://doi.org/10.1145/1370175.1370194

[13] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. 2008. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, Atlanta Georgia, 214–225. https://doi.org/10.1145/1453101.1453130

[14] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[15] Christian Doerner, Andrew R. Faulring, and Brad A. Myers. 2014. EUKLAS: Supporting Copy-and-Paste Strategies for Integrating Example Code. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, Portland Oregon USA, 13–20. https://doi.org/10.1145/2688204.2688208

[16] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. https://doi.org/10.1145/3597503.3639219

[17] 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*. ACM, Vasteras Sweden, 313–324. https://doi.org/10.1145/2642937.2642982

[18] Uriel Ferreira Campos, Guilherme Smethurst, Joao Pedro Moraes, Rodrigo Bonifacio, and Gustavo Pinto. 2019. Mining Rule Violations in JavaScript Code Snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 195–199. https://doi.org/10.1109/MSR.2019.00039

[19] Felix Fischer, Konstantin Bottinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 121–136. https://doi.org/10.1109/sp.2017.31

[20] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-Based LLMs for Code Optimization. https://doi.org/10.48550/arXiv.2408.12159 arXiv:2408.12159 [cs].

[21] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from Here, Some from There: Cross-Project Code Reuse in GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, Buenos Aires, Argentina, 291–301. https://doi.org/10.1109/MSR.2017.15

[22] Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi LI, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2024. CodeEditorBench: Evaluating Code Editing Capability of Large Language Models. arXiv:2404.03543 [cs.SE] https://arxiv.org/abs/2404.03543

[23] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.

[24] Reid Holmes and Robert J. Walker. 2012. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology* 21, 4 (Nov. 2012), 1–44. https://doi.org/10.1145/2377656.2377657 Publisher: Association for Computing Machinery (ACM).

[25] Qing Huang, Jiahui Zhu, Zhilong Li, Zhenchang Xing, Changjing Wang, and Xiwei Xu. 2023. PCR-Chain: Partial Code Reuse Assisted by Hierarchical Chaining of Prompts on Frozen Copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, Melbourne, Australia, 1–5. https://doi.org/10.1109/ICSE-Companion58688.2023.00013

[26] Yuan Huang, Furen Xu, Haojie Zhou, Xiangping Chen, Xiaocong Zhou, and Tong Wang. 2022. Towards Exploring the Code Reuse from Stack Overflow during Software Development. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 548–559. https://doi.org/10.1145/3524610.3527923 event-place: Virtual Event.

[27] Patricia Jablonski and Daqing Hou. 2007. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. ACM, Montreal Quebec Canada, 16–20. https://doi.org/10.1145/1328279.1328283

[28] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2 (Jan. 2007).

[29] Jiaolong Kong, Xiaofei Xie, and Shangqing Liu. 2025. Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework. *Proceedings of the ACM on Software Engineering* 2, FSE (June 2025), 2712–2734. https://doi.org/10.1145/3729390 Publisher: Association for Computing Machinery (ACM).

[30] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081

[31] Ze Shi Li, Nowshin Nawar Arony, Ahmed Musa Awon, Daniela Damian, and Bowen Xu. 2024. AI Tool Use and Adoption in Software Development by Individuals and Organizations: A Grounded Theory Study. arXiv:2406.17325 [cs.SE] https://arxiv.org/abs/2406.17325

[32] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1521.

[33] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. http://arxiv.org/abs/2107.13586 arXiv:2107.13586 [cs].

[34] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy. 2022. AdaptivePaste: Code Adaptation through Learning Semantics-aware Variable Usage Representations. http://arxiv.org/abs/2205.11023 arXiv:2205.11023 [cs].

[35] Adriaan Lotter, Sherlock A. Licorish, Bastin Tony Roy Savarimuthu, and Sarah Meldrum. 2018. Code Reuse in Stack Overflow and Popular Open Source Java Projects. In *2018 25th Australasian Software Engineering Conference (ASWEC)*. 141–150. https://doi.org/10.1109/ASWEC.2018.00027 ISSN: 2377-5408.

[36] Yubo Mai, Zhipeng Gao, Xing Hu, Lingfeng Bao, Yu Liu, and JianLing Sun. 2024. Are Human Rules Necessary? Generating Reusable APIs with CoT Reasoning and In-Context Learning. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2355–2377. https://doi.org/10.1145/3660811 Publisher: Association for Computing Machinery (ACM).

[37] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. Investigating Context Adaptation Bugs in Code Clones. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Cleveland, OH, USA, 157–168. https://doi.org/10.1109/ICSME.2019.00026

[38] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A graph-based approach to API usage adaptation. *ACM SIGPLAN Notices* 45, 10 (Oct. 2010), 302–321. https://doi.org/10.1145/1932682.1869486 Publisher: Association for Computing Machinery (ACM).

[39] OpenAI. 2024. *gpt-4o*. https://platform.openai.com/docs/models/gpt-4o

[40] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Nondeterminism of ChatGPT in Code Generation. arXiv:2308.02828 [cs.SE] https://arxiv.org/abs/2308.02828

[41] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (Aug. 2015), 1–18. https://doi.org/10.1016/j.infsof.2015.03.007 Publisher: Elsevier BV.

[42] Qwen3. 2024. *Qwen3-Coder*. https://qwenlm.github.io/blog/qwen3-coder

[43] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2021. Toxic Code Snippets on Stack Overflow. *IEEE Transactions on Software Engineering* 47, 3 (2021), 560–581. https://doi.org/10.1109/TSE.2019.2900307

[44] Brittany Reid, Marcelo d'Amorim, Markus Wagner, and Christoph Treude. 2023. NCQ: Code Reuse Support for Node.js Developers. *IEEE Transactions on Software Engineering* 49, 5 (May 2023), 3205–3225. https://doi.org/10.1109/TSE.2023.3248113

[45] Brittany Reid, Christoph Treude, and Markus Wagner. 2020. Optimising the fit of stack overflow code snippets into existing code. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. ACM, Cancún Mexico, 1945–1953. https://doi.org/10.1145/3377929.3398087

[46] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The extent of orphan vulnerabilities from code reuse in open source software. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2104–2115. https://doi.org/10.1145/3510003.3510216

[47] Qi Shen, Shijun Wu, Yanzhen Zou, and Bing Xie. 2021. Comprehensive Integration of API Usage Patterns. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, Madrid, Spain, 83–93. https://doi.org/10.1109/ICPC52881.2021.00017

[48] Manuel Sojer and Joachim Henkel. 2011. License risks from ad hoc reuse of code from the internet. *Commun. ACM* 54, 12 (Dec. 2011), 74–81. https://doi.org/10.1145/2043174.2043193 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[49] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2025. Calibration and Correctness of Language Models for Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 540–552. https://doi.org/10.1109/ICSE55347.2025.00040

[50] Kimi Team. 2025. Kimi K2: Open Agentic Intelligence. arXiv:2507.20534 [cs.LG] https://arxiv.org/abs/2507.20534

[51] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken Germany, 118–129. https://doi.org/10.1145/2931037.2931058

[52] Valerio Terragni and Pasquale Salza. 2021. APIzation: Generating Reusable APIs from StackOverflow Code Snippets. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 542–554. https://doi.org/10.1109/ASE51524.2021.9678576

[53] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. arXiv:2401.04621 [cs.SE] https://arxiv.org/abs/2401.04621

[54] Taiming Wang, Yanjie Jiang, Chunhao Dong, Yuxia Zhang, and Hui Liu. 2025. Context-Aware Code Wiring Recommendation with LLM-based Agent. https://doi.org/10.48550/arXiv.2507.01315 arXiv:2507.01315 [cs].

[55] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. 2016. Hunter: Next-Generation Code Reuse for Java, T Zimmermann, J ClelandHuang, and Z Su (Eds.). 1028–1032. https://doi.org/10.1145/2950290.2983934

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[57] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. 2012. SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization. ACM, 219–228.

[58] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24, 2 (April 2019), 637–673. https://doi.org/10.1007/s10664-018-9634-5

[59] 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*. ACM, Singapore Singapore, 959–971. https://doi.org/10.1145/3540250.3549101

[60] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. http://arxiv.org/abs/2304.00385 arXiv:2304.00385 [cs].

[61] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-based Program Repair. https://doi.org/10.48550/arXiv.2404.08877 arXiv:2404.08877 [cs].

[62] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, Austin Texas, 391–402. https://doi.org/10.1145/2901739.2901767

[63] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. 2017. Stack Overflow in Github: Any Snippets There?. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 280–290. https://doi.org/10.1109/MSR.2017.13

[64] Mingke Yang, Yuming Zhou, Bixin Li, and Yutian Tang. 2023. On Code Reuse from StackOverflow: An Exploratory Study on Jupyter Notebook. http://arxiv.org/abs/2302.11732 arXiv:2302.11732 [cs].

[65] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A Survey on Deep Learning for Software Engineering. *Comput. Surveys* 54, 10s (Jan. 2022), 1–73. https://doi.org/10.1145/3505243

[66] Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. 2024. A Survey on Modern Code Review: Progresses, Challenges and Opportunities. http://arxiv.org/abs/2405.18216 arXiv:2405.18216 [cs].

[67] Tanghaoran Zhang, Yao Lu, Yue Yu, Xinjun Mao, Yang Zhang, and Yuxin Zhao. 2024. How Do Developers Adapt Code Snippets to Their Contexts? An Empirical Study of Context-Based Code Snippet Adaptations. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2712–2731. https://doi.org/10.1109/TSE.2024.3395519

[68] Tanghaoran Zhang, Xinjun Mao, , Yue Yu, Xinjun Mao, Shangwen Wang, Kang Yang, Yao Lu, Zhang Zhang, and Yuxin
        Zhao. 2025. AdaptEval: A Benchmark for Evaluating Large Language Models on Code Snippet Adaptation. In *2025
        IEEE/ACM 40th International Conference on Software Engineering (ASE)*. 566–577.
[69] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples
        on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *Proceedings of the 40th International
        Conference on Software Engineering*. ACM, Gothenburg Sweden, 886–896. https://doi.org/10.1145/3180155.3180260
[70] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code
        Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 316–327.
        https://doi.org/10.1109/ICSE.2019.00046 event-place: Montreal, Quebec, Canada.
[71] Tanghaoran Zhang, Yue Yu, Xinjun Mao, Shangwen Wang, Kang Yang, Yao Lu, Zhang Zhang, and Yuxin Zhao. 2025.
        Instruct or Interact? Exploring and Eliciting LLMs' Capability in Code Snippet Adaptation Through Prompt Engineering.
        In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 566–577. https://doi.org/10.1109/
        ICSE55347.2025.00104 ISSN: 1558-1225.
[72] Yuanliang Zhang, Yifan Xie, Shanshan Lit, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng
        Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, and Xiangke Liao. 2025. Unseen Horizons: Unveiling the Real
        Capability of LLM Code Generation Beyond the Familiar. In *2025 IEEE/ACM 47th International Conference on Software
        Engineering (ICSE)*. 604–615. https://doi.org/10.1109/ICSE55347.2025.00082