AdaptEval: A Benchmark for Evaluating Large Language Models on Code Snippet Adaptation

Tanghaoran Zhang*, Xinjun Mao*, Shangwen Wang*, Yuxin Zhao*, Yao Lu*, Jin Zhang†, Zhang Zhang*, Kang Yang* and Yue Yu*‡§

*College of Computer Science and Technology, National University of Defense and Technology, Changsha, China, Email: {zhangthr, xjmao, wangshangwen13, yuxinzhao, luyao08, zhangzhang14, yangkang}@nudt.edu.cn

†Changsha University of Science and Technology, Changsha, China, Email: mail_zhangjin@163.com

‡Peng Cheng Laboratory, Shenzhen, China, Email: yuy@pcl.ac.cn

***Abstract*—Recent advancements in large language models (LLMs) have automated various software engineering tasks, with benchmarks emerging to evaluate their capabilities. However, for adaptation, a critical activity during code reuse, there is no benchmark to assess LLMs' performance, leaving their practical utility in this area unclear. To fill this gap, we propose AdaptEval, a benchmark designed to evaluate LLMs on code snippet adaptation. Unlike existing benchmarks, AdaptEval incorporates the following three distinctive features: First, *practical context*. Tasks in AdaptEval are derived from developers' practices, preserving rich contextual information from Stack Overflow and GitHub communities. Second, *multi-granularity annotation*. Each task is annotated with requirements at both task and adaptation levels, supporting the evaluation of LLMs across diverse adaptation scenarios. Third, *fine-grained evaluation*. AdaptEval includes a two-tier testing framework combining adaptation-level and function-level tests, which enables evaluating LLMs' performance across various individual adaptations. Based on AdaptEval, we conduct the first empirical study to evaluate six instruction-tuned LLMs and especially three reasoning LLMs on code snippet adaptation. Experimental results demonstrate that AdaptEval enables the assessment of LLMs' adaptation capabilities from various perspectives. It also provides critical insights into their current limitations, particularly their struggle to follow explicit instructions. We hope AdaptEval can facilitate further investigation and enhancement of LLMs' capabilities in code snippet adaptation, supporting their real-world applications.**

***Index Terms*—Code Snippet Adaptation, Large Language Models, Benchmark, Code Reuse**

I. Introduction

Recent advancements in large language models (LLMs) have revolutionized software development by significantly reducing the human effort in coding, ushering in the era of automatic programming [1]. These models with billions of parameters demonstrate remarkable performance in various software engineering tasks, including code generation [2], [3], automated program repair [4], [5], *etc.* To better understand LLMs' capabilities on the corresponding tasks, existing research proposes a series of benchmarks to evaluate their performance, such as HumanEval [6] and SWE-Bench [7].

Reusing code snippets is a widely adopted practice to improve the quality and efficiency of software development [8]–[12]. A critical step in this process is adaptation, where developers modify code snippets to fit their specific context [13], [14]. Despite its practical significance, code snippet adaptation by LLMs remains unexplored in recent research. One of the main obstacles in this domain is the lack of evaluation benchmarks. Constructing benchmarks for adaptation, however, faces significant challenges due to three primary reasons: ***(i) Lack of practical context.*** Adaptation is inherently a context-aware task that involves transplanting a code snippet from one specific environment to another. Effective evaluation should therefore be grounded in practical scenarios, *e.g.*, adaptation from Stack Overflow (SO) posts to developers' code bases, to accurately characterize the capabilities of LLMs in adaptation practices. ***(ii) Lack of task requirements.*** Although adaptation is highly context-dependent, it is fundamentally a requirement-driven task, as its ultimate goal is to fulfill customized user needs (*e.g.*, optimizing performance or extending functionality). However, existing resources such as version control systems, often lack documentation of developers' specific adaptation requirements [14], [15]. Hence, it remains unclear why developers make certain adaptations or what adaptation steps are necessary to integrate a code snippet into a new context. This absence of task-oriented information hinders the formulation of precise inputs for evaluating LLMs on adaptation tasks. ***(iii) Lack of fine-grained evaluation.*** Current benchmarks primarily focus on end-to-end correctness of the LLM-generated code, only providing a binary "correct/incorrect" assessment. Nevertheless, an adaptation task may include a set of code edits. Evaluation on individual adaptations allows feedback on the intermediate steps, as well as LLMs' strengths and bottlenecks across diverse adaptations.

To address these challenges, we propose ***AdaptEval***, the first benchmark for code snippet adaptation, comprising 164 tasks with 523 adaptations in Python language. It is designed with three distinctive features. ***Firstly***, each task in AdaptEval is collected from the actual adaptation practice of developers. We preserve the original context by including the referenced SO post and the associated GitHub repository for better understanding and traceability. ***Secondly***, we annotate each task with multi-granularity descriptions: Task-level annotations

§Yue Yu is the corresponding author.

Tanghaoran Zhang, Xinjun Mao, Yuxin Zhao, Yao Lu, Zhang Zhang and Kang Yang are with the State Key Laboratory of Complex & Critical Software Environment.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00103

describe concise developer intentions, evaluating LLMs' intention understanding and code reasoning ability as intelligent assistants, while adaptation-level ones simulate developers' step-by-step adaptation process, serving as specific instructions for LLMs to implement code changes. ***Thirdly***, we construct a two-tier testing framework to assess the correctness of code at both function and adaptation levels. Combined with our annotations, AdaptEval can further evaluate LLMs on individual adaptations across various types and their intermediate adaptation steps. Overall, it takes approximately ***550 person-hours*** to construct AdaptEval, covering adaptations in 38 types. Our test suite also achieves high test sufficiency with 92.95% branch coverage and 94.38% line coverage.

Based on AdaptEval, we first evaluate six instruction-tuned LLMs for their code snippet adaptation performance. Our results show that LLMs can solve 34.15% to 59.15% tasks in AdaptEval. Compared with task-level requirements, all LLMs perform significantly better with adaptation-level steps, where an increase up to 34.84% in pass@1 is observed. Regarding adaptation types, LLMs perform best on *Method Signature* and worst on *Logic Customization*, with a gap of 20.31% on average. Our analysis on their failures reveals that LLMs may violate the provided requirements due to their pre-training knowledge. Additionally, we benchmark three reasoning LLMs on AdaptEval. Results indicate that they are more effective in capturing implicit contextual cues even when no explicit requirements are provided. However, their reasoning process still need more alignments to developers' actual adaptation strategies.

This paper makes the following contributions:

- We propose AdaptEval[1], the first benchmark for code snippet adaptation, comprising 164 tasks derived from real-world, cross-platform adaptation practices of developers.
- We introduce a distinctive evaluation design, including multi-granularity annotation and fine-grained evaluation in AdaptEval. It supports in-depth analysis of LLMs' adaptations beyond their function-level correctness.
- We conduct the first study to evaluate both instruction-tuned and reasoning LLMs on code snippet adaptation. Results suggest LLMs' strengths and limitations on adaptation tasks and point out future directions for improvements.

II. Background And Related Work

A. Code Snippet Reuse and Adaptation

Open-source software platforms like SO provide millions of code snippets for programming problems. Reusing these solutions from the crowd benefits development efficiency and software quality. However, developers still need to adapt them to their context. Prior studies mainly focus on automating particular categories of adaptations based on historical data to reduce human efforts. Cottrell et al. [16] present Jigsaw to integrate snippets into developers' code, which resolve simple adaptations such as renaming. Zhang et al. [13] develop a Chrome plugin, ExampleStack, to generate use templates from developers' historical adaptations. Terragni et al. [17] focus on a special type of adaptation, APIzation, *i.e.*, transform code snippets to well-formed methods for convenient reuse. They summarize four APIzation patterns and present APIZATOR to make recommendations. However, above adaptation tools are not effective in understanding developers' specific context and making adaptations based on it [11], [14]. Hence, it is still challenging for developers to perform adaptations. With the emergence of LLMs, their prompt-based learning [18] ability allows them to perform unseen tasks during inference, providing opportunities for the code snippet adaptation task. Compared to code generation, adaptation requires different abilities, including reasoning about the reuse code and modifying it to meet new constraints [19]. To advance the research and practices of LLM-based adaptation, there is a critical need for a dedicated evaluation benchmark in this domain.

B. LLMs for Code

The emergence of LLMs brings the automation of code-related tasks to full bloom. Due to pre-training on an extremely large scale of textual and code corpora, LLMs are empowered with emerging abilities [20] on both natural language and code-related tasks. Specifically, they could be divided into three categories: (i) ***General LLMs*** that are trained to solve a wide range of natural language tasks, such as GPT-series [21], [22], Claude-series [23], and Llama-series [24]; (ii) ***Code LLMs*** that are trained specifically for code-related tasks on code corpora, such as DeepSeek-Coder [25], Codestral [26], Qwen2.5-Coder [27]; and (iii) ***Reasoning LLMs*** that are specifically designed for logical reasoning, problem-solving, and complex inference tasks, *e.g.*, OpenAI's o1-series [28], DeepSeek-R1 [29] and QwQ [30]. However, their instruction-following abilities are observed to be degraded in recent studies [31], [32]. Moreover, they suffer from high inference latency and token cost, which are less practical in daily software development scenarios such as issue resolution [7]. Despite their differences, LLMs' effectiveness in code snippet adaptation has never been explored so far.

C. Existing Benchmarks for Code

Programming and debugging are two fundamental activities in software development, which have been extensively studied [3], [57]–[59]. Considering their differences, we can classify all benchmarks into three categories, *i.e.*, code generation, editing and multi-task. An increasing number of LLM-based approaches are adopted for them, especially after the release of ChatGPT in December, 2022. Previous benchmarks such as HumanEval [6] are considered too simple to demonstrate LLMs' capabilities. To this end, prior studies propose a series of benchmarks to better evaluate LLMs [56], [60]. In this paper, we collect and analyze the most recent benchmarks that (i) evaluate LLMs' code generation and editing abilities, and (ii) are associated with an academic paper released between January, 2023 to October, 2024. Table I lists the overview of the 25 retrieved benchmarks.

[1] https://github.com/ztwater/AdaptEval.

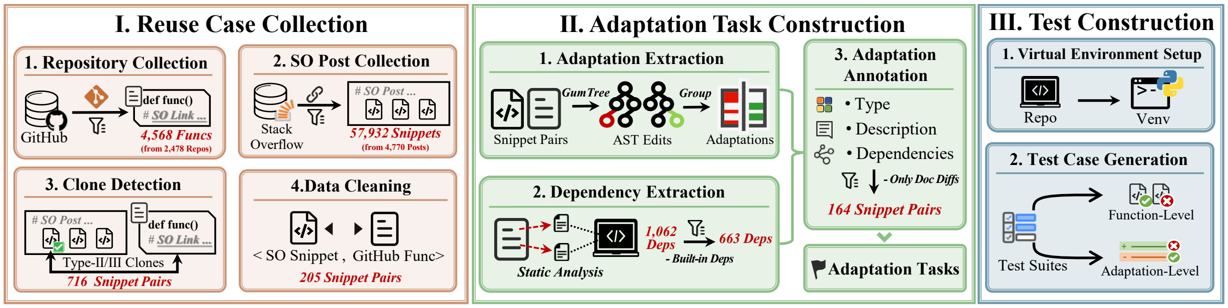


Fig. 1. The construction process of AdaptEval.

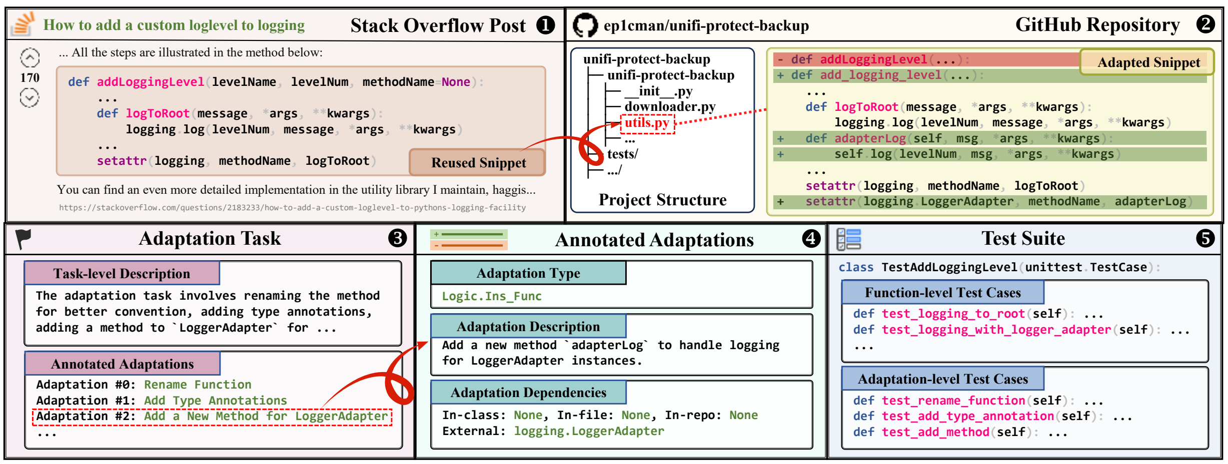


Fig. 2. The overall structure of AdaptEval.

TABLE I
THE OVERVIEW OF RECENT LLM BENCHMARKS FOR CODE.

Benchmark	Time	Task	Scale	Gran.	Dep.	Source	Test
CoderEval [33]	2023	Gen	230	Func	Repo	GitHub	Func
RepoBench [34]	2023	Gen	26,970	Line	Repo	GitHub	-
ClassEval [35]	2023	Gen	100	Class	Class	-	Class
CrossCodeEval [36]	2023	Gen	9,928	Token	Repo	GitHub	-
HumanEval-XL [37]	2024	Gen	22,080	Func	-	-	Func
EffiBench [38]	2024	Gen	1,000	Func	-	LeetCode	Func
Mercury [39]	2024	Gen	256	Func	-	LeetCode	Func
EvoCodeBench [40]	2024	Gen	275	Func	Repo	GitHub	Func
Exec-CSN [41]	2024	Gen	1,931	Func	File	GitHub	Func
DevEval [42]	2024	Gen	1,874	Func	Repo	PyPI	Func
OOP [43]	2024	Gen	431	Func	Class	Multi-Src	Func
$\mathrm{R}^2\mathrm{C}^2$Bench [44]	2024	Gen	22,828	Func	Repo	GitHub	-
RepoClassBench [45]	2024	Gen	287	Class	Repo	GitHub	Class
JavaBench [46]	2024	Gen	106	Class	Repo	GitHub	Class
ComplexCodeEval [47]	2024	Gen	11,081	Func	Repo	GitHub	Func
SWE-Bench [7]	2023	Edit	2,294	Repo	Repo	GitHub	PR
DebugBench [48]	2024	Edit	4,253	Func	-	LeetCode	Func
CodeEditorBench [49]	2024	Edit	7,941	Func	-	Multi-Src	Func
CrossCodeBench [50]	2023	Multi	54M	Func	-	Multi-Src	-
XCodeEval [51]	2023	Multi	25M	Prog	File	CodeForces	Prog
CodeApex [52]	2023	Multi	2,056	Func	-	Internal	Func
CodeScope [53]	2023	Multi	13,390	Prog	File	Multi-Src	Prog
CoderUJB [54]	2024	Multi	2,239	Func	Repo	Defects4J	Func
DevBench [55]	2024	Multi	22	Repo	Repo	GitHub	Repo
LiveCodeBench [56]	2024	Multi	511	Func	-	Multi-Src	Func
AdaptEval	2025	**Adapt**	164	Func	**Repo**	**SO-GitHub**	**Adapt**

Existing benchmarks mainly evaluate LLMs' code generation ability, for which they are pre-trained by next token prediction paradigm [61]–[63]. Various dimensions, including the scale, efficiency and executability, are studied. Several benchmarks aim to evaluate multiple capabilities of LLMs simultaneously. For instance, CoderUJB [54] includes code generation, test generation, program repair, and defect detection tasks. These benchmarks provide insights into LLM selection in different scenarios. Compared with generation, code editing benchmarks receive less attention. Only three benchmarks are proposed. Specifically, SWE-Bench [7] evaluates how LLMs can solve real GitHub issues in repo-level context. DebugBench [48] improves existing program repair benchmarks with larger-scale and richer bug types. CodeEditorBench [49] comprehensively evaluates the debugging, translating, polishing, and requirement switching tasks. However, as a significant code editing scenario, code snippet adaptation has not been supported by current benchmarks so far.

III. THE ADAPTEVAL BENCHMARK

In this section, we first introduce the overall structure of AdaptEval. Then we introduce its construction process (as shown in Figure 1), including reuse case collection (Section III-B), adaptation task construction (Section III-C) and test construction (Section III-D). Finally, we report its statistics.

A. Overall Structure

Figure 2 illustrates the overall structure of AdaptEval. Each task comprises three key components in the following:

- ***Practical context***: AdaptEval captures developers' real-world reuse workflow by identifying ❶ the original SO post (reuse source) and ❷ their target GitHub repository (destination context) through mining techniques. This design ensures the veracity of our benchmark, reflecting the challenges encountered in actual reuse practices.

- ***Multi-granularity annotation***: AdaptEval provides adaptation requirements at two levels: ❸ task-level descriptions outlining developer intentions, and ❹ adaptation-level ones characterizing stepwise operations. The former aims to evaluate LLMs as intelligent assistants to understand intentions and then reason for a feasible solution, while the latter evaluates LLMs as instruction-followers when actionable adaptation instructions are provided. This design aims to evaluate LLMs across diverse adaptation scenarios.
- ***Fine-grained Evaluation***: AdaptEval includes ❺ two-tier test suites with function-level and adaptation-level tests, which enables in-depth diagnosis of the adaptation process as well as LLMs' strengths and weaknesses across various adaptation types.

B. Reuse Case Collection

In this step, we collect real-world developers' reuse cases from two most popular communities, SO and GitHub, according to prior studies [13], [14], [17]. Our collection process consists of four steps. Firstly, we collect GitHub repositories and extract functions with explicit SO links. Secondly, we collect the corresponding SO posts. Thirdly, we identify candidate code reuse using a combination of explicit SO link identification and clone detection techniques to minimize spurious reuse. Finally, we clean the data by removing duplicate cases and checking the authenticity of reuse.

1) Repository Collection: We utilize GHS [64] and GitHub REST API [65] to collect our GitHub repositories. The collection is based on the following criteria:

- The main language of the repository's source code is Python. The reason is that Python is one of the most popular programming languages in both SO and GitHub [66], which contains adequate data for benchmark construction.
- The repository is non-forked and was created after 2023. This rule aims to avoid repetition and balance the dataset size and potential data leakage risk.
- At least one Python code file in the repository has an explicit reference to a SO post within the method scope. This rule filters out reuse cases that do not target coding purposes.

We identify a total of 2,478 Python repositories. To isolate reused code from later evolutionary changes, we follow Zhang et al. [14] to collect the file version immediately after the commit that introduced the code. This step filters out non-adaptation changes. Besides, we also preserve all associated commits as authentic process records for subsequent annotation. Finally, we extract 4,568 functions with SO links.

2) SO Post Collection: To retrieve reused SO snippets, we use the Stack Exchange API [67] to collect SO posts according to the reference links in the GitHub source code files. We observe that developers may refer to either the question post or a specified answer post in their source code files. To obtain the complete reuse context, we collect the whole SO question post instead of a single answer. Specifically, we first retrieve the corresponding parent question posts for each linked answer. Then, for each question post, we query its *id*, *title*, *body*, *answers*, *owner*, *tags*, *score*, *comments* fields and timestamps for activities. For each answer belonging to the question, we collect its *id*, *body*, *owner*, *is_accepted*, *score* fields and timestamps for activities. Finally, we extract all code spans as individual code snippets from each SO answer post, serving as potential reused snippets. Followed Zhang et al. [14], we discard posts with non-positive scores to consider both the quality and coverage of our data, resulting in 57,932 snippets from 4,770 posts eventually.

3) Clone Detection: To identify the candidates of code reuse, prior studies often employ two approaches including link identification [14], [17] and code clone detection [11]–[13], [68]. The former identifies explicit SO references in GitHub snippets but could include spurious reuse, such as references to problems or explanations. The latter ensures the code similarity but may introduce false positives, *e.g.*, coincidental clones, trivial snippets, *etc*. Following Terragni et al. [17], we combine the clone detection technique and explicit SO link references to obtain candidate code reuse, which are more likely to be genuine.

We first pair each collected function from GitHub with SO snippets extracted from the linked post to avoid identifying coincidental clones from other posts. For each $\langle$GitHub function, [SO snippet list]$\rangle$ pairs, we use SourcererCC [69] to search Type-II and Type-III clones as we are only interested in explicit code reuse with adaptations. Type-I clones only include differences in blank spaces and comments, which does not require any adaptations, while Type-IV clones (semantically equivalent but with little syntactical similarity) may not suggest explicit reuse and are difficult to detect. Aligned with previous studies [13], [17], we determine two snippets as clones by setting the token similarity threshold to 70% for the best tradeoff between the precision and recall. We discard all functions without any clones in their linked posts as spurious reuse. As for functions with multiple clones from SO, we select the one with the highest similarity. After clone detection, we obtain 716 snippet pairs.

4) Data Cleaning: Different developers may perform exactly the same adaptations accidentally. Such duplicate data in our benchmark may lead to unfair evaluation. To this end, we exclude snippet pairs whose source SO snippet and GitHub function are both identical, resulting in 232 snippet pairs. Although we employ filtration rules and clone detection techniques to identify potential code reuse, there could still be exceptions, *e.g.*, code written in another programming language in SO answer posts. We further filter these cases manually. Eventually, we obtain 205 snippet pairs.

C. Adaptation Task Construction

As developers' original adaptation requirements are not properly recorded, we aim to reconstruct adaptation tasks from their actual practices. To accurately describe them, we construct each task following a bottom-up manner. Specifically, we first utilize *GumTree* [70] to extract code edits at the Abstract Syntax Tree (AST) level and group them into ***adaptations***, which serves as the unit of evaluation in AdaptEval. Subsequently, we develop a static analysis tool with *tree-sitter* [71] to extract their dependencies as relevant context. Then we annotate each adaptation with a detailed description, a type label and required dependencies. Finally, we combine all annotated adaptations in a single reuse case as an adaptation ***task***, and summarize a task-level requirement.

1) Adaptation Extraction: In this step, we first extract all code edits from reuse snippet pairs leveraging a popular code-differencing tool, *GumTree* [70]. However, individual AST-level edits are too fine-grained for understanding and description [72]. For instance, inserting an *if* statement to handle a new condition may include numerous AST edits, which should be considered as a whole. To better reflect developers' intentions, we group the edits according to their syntax adjacency on the AST following CLDiff [72].

2) Dependency Extraction: Since adaptation requires developers to integrate code snippets into their code bases, we extract all repository-level context dependencies for each adapted function in AdaptEval, including four types, *i.e.*, intra-class, intra-file, intra-repo and external. For each adapted function, we first extract all its used identifiers from its AST. Then we extract defined classes, functions, and global variables from the current file containing the adapted function. If an identifier is identical to a class, field or method name, it belongs to the intra-class dependency, denoted as "*ClassA.MethodB*". If an identifier is defined as a global variable or a function, then it belongs to the intra-file dependency, simply denoted as their name. Subsequently, we distinguish intra-repo and external dependencies by parsing the import statements and the repository files. If an identifier is located within the repository, it belongs to the intra-repo dependencies, denoted by "*Path/To/File::ClassA.MethodB*". Identifiers with imported third-party library names belong to external dependencies, denoted by their full-qualified names without alias, *e.g.*, "*numpy.array*". The identifiers of builtin functions, variables and constants in Python 3 are discarded.

3) Adaptation Annotation: During this step, we manually annotate each adaptation with its type and description. This process is carried out by five participants with at least three years of Python programming experience. Two of them are responsible for writing the adaptation description, while another two conduct the open-coding process of adaptation types. Each member in both groups takes charge of half the cases and then double-checks the other's annotation for cross-validation. The remaining participant with the most experience serves as the mediator and final reviewer. We first annotate adaptation descriptions, whose criteria include: intent-based, concise and consistent. To ensure that they accurately reflect developers' intents, we employ a rigorous and multi-step protocol. Before annotation, we conduct a pilot study with annotators as well as two senior engineers with industrial experience on five real-world examples. All participants are provided with: (1) the original SO post, (2) the integration commit (Section III-B1), and (3) their linked artifacts (commit messages, discussions) to extract the rationale behind each adaptation. All participants found the integration commit helped them comprehend the original intent, especially when commit messages and inline comments reveal the motivations. To this end, we require annotators carefully refer to them during annotation. During the discussion, we set a 50-word threshold for each description, which could balance the conciseness and detail. Beside, we share an online document with well-designed demonstrations and dynamically update it to ensure consistency throughout the annotation. To mitigate subjectivity, our cross-validation requires the mediator to resolve every disagreement through discussion until consensus is achieved. Our approach minimizes individual bias while aligning our annotations with accessible records documented in commit histories. After all descriptions are completed, the other group annotates adaptation types following the principles of thematic analysis [73]. The participants generate the initial codes using a *verb-object* phrase, *e.g.*, Rename Function, to characterize the adaptation. After cross-validation, they work together to group them into themes. Then they iteratively re-evaluate and group the themes until they are established. The final themes are reviewed by the final reviewer. If there are any inappropriate expressions, the three participants will discuss and refine them to reach the consensus. The Cohen's Kappa value [74] between two annotators during the cross-validation phase is 0.929, while the Cohen's Kappa value between annotators and the final reviewer on the final themes is 0.975. Finally, we obtain descriptions for each adaptation and the adaptation taxonomy.

As adaptations in docstrings or comments do not affect the functionality, we exclude them as well as the cases with only adaptations of this type from AdaptEval. It allows us to focus on significant adaptations, thereby reducing noise in the benchmark. The derived adaptation taxonomy is shown in Table II. Specifically, it includes three categories, *Method Signature*, *Logic Customization* and *Refactoring*, with 128, 227, and 168 adaptations respectively. *Method Signature* refers to adaptations made to the method interface while preserving the logic in its body. *Logic Customization* refers to the functional changes in the method body. *Refactoring* refers to non-functional adaptations which do not change the program behavior. The most prevalent adaptation is *Rename Function*, with 70 occurrences. It indicates that developers often need a more descriptive or consistent function name during adaptation. While Zhang et al. propose an insightful taxonomy [13] for adaptation, it is based on Java and does not consider the recent studies on APIzation [17], which is revealed by our *Method Signature* category.

Based on our extracted dependencies, we associate each adaptation with its dependencies by determining whether its updated code elements are present in our dependency set. To obtain task-level requirement description, we first group all adaptations by their types as they are likely for the same purpose. Then we summarize each group into a concise phrase and combine them as a one-sentence description. This ensures

TABLE II
THE ADAPTATION TAXONOMY ON ADAPTEVAL

Category	Adaptation Type
MS (128)	Encapsulate (13), Rename Function (70), Update Function Type (11), Insert/Delete/Update Parameter (13/5/16)
LC (227)	Initialize/Replace Variable (3/11), Update Constant (13), Convert Object Type (23), Insert/Delete/Update Call (22/8/52), Insert Import (16), Insert/Delete/Update Return (5/1/8), Insert/Delete/Update Condition(31/13/9), Handle Exception (11), Insert Function (1)
RE (168)	Insert/Delete/Update Type Annotation (52/3/1), Insert/Delete Temporary Variable (9/3), Rename Parameter/Variable (24/20), Update API Referenced Name (25), Insert None Return (2), Refactor/Move Expression (17/1), Delete Import (3), Split Lines (2), Inline/Expand/Move Function (1/3/2)

that the task-level descriptions are both comprehensive and concise. Finally, we derive our final set of 164 adaptation tasks.

D. Test Construction

As our benchmark needs to support an automatic evaluation of LLM-adapted code from hundreds of repositories, we manually set up a virtual testing environment and create a test suite for each task in AdaptEval. Apart from the overall correctness of the adapted function, we also enable a fine-grained evaluation using adaptation-level test cases. In the testing pipeline, we automatically extract the adapted function from LLMs' output, inject the code under test into the corresponding context, and run the tests in a virtual environment.

1) Virtual Environment Setup: To avoid dependency conflicts across different repositories, we deploy each repository in an individual virtual environment and install its required dependencies. Specifically, we first look for the supported Python version specified in the documentation of each repository. Then we use *venv* to create a virtual environment in its root directory and use *pip* to install all the required Python packages specified in the dependency list. To alleviate the burden of deployment, we only install the necessary dependencies that allow the execution of the adapted functions.

2) Test Case Generation: We construct two-tier test suites containing both function and adaptation-level test cases in AdaptEval to evaluate LLMs' adaptations. Function-level test cases assess the overall functionality of the adapted code. They ensure that functional adaptations correctly modify program behaviors while preserving those of the remaining code. However, only considering overall functionality is insufficient. First, it cannot evaluate individual adaptations, as LLMs may partially succeed in some adaptations yet fail function-level tests. Fine-grained tests are essential for identifying LLMs' effectiveness and bottlenecks across in different adaptation types. Second, refactoring, which is also critical in software reuse, cannot be directly evaluated through program behavior alone. Therefore, AdaptEval incorporates adaptation-level test cases for a more comprehensive evaluation.

The core criterion for our adaptation-level tests is their discriminative power: each test should pass only when its target adaptation is correctly applied and fail otherwise. Specifically, given a sequence of adaptations $\mathcal{A} = \{a_0, a_1, \ldots, a_n\}$, our objective is to write adaptation-level tests $\mathcal{T} = \{t_0, t_1, \ldots, t_n\}$ such that passing t_i ensures that the corresponding adaptation a_i is made to the reused snippet and is not influenced by subsequent adaptations $\{a_{i+1}, \ldots, a_n\}$. For instance, Figure 3 illustrates an example of adaptation-level test cases in AdaptEval. a_0 is a default parameter addition and a_1 updates an API call with the added argument. We construct t_0 to validate that the parameter is added in the correct place and its default value is *None*. For a_1, we write t_1 to call the adapted function with a different *chunk_size* value, *4096*, and use mock objects to determine whether the *iter_content* method is called with that value precisely.

We consider reusing the original tests written by developers for our adapted functions. However, 142 out of 164 tasks lack unit tests in their repositories and the remaining tests are insufficient. Hence, we manually construct function-level and adaptation-level test cases for each task, ensuring their discriminative power and coverage. The construction process involves four participants with at least three years of Python programming experience. Three of them are responsible for writing test cases while the last deploys all repositories and rigorously checks the test cases to ensure their correctness and quality. The test suites in AdaptEval are written using the Python *unittest* framework. The idea of "mocking" [75] is also adopted to improve the stability of the test and provide a controlled and efficient testing environment. We try our best to obtain a line-level and branch-level coverage of 92.95% and 94.38%. Each adaptation is covered by at least one test case to validate whether it is performed correctly as required.

E. Dataset Statistics

TABLE III
THE OVERALL STATISTICS OF TASKS IN ADAPTEVAL.

Metric	Mean	Total
Line-of-Code (LOC)	18.9	3,106
Num of Adaptations	3.2	523
Num of AST Edits	13.6	2,236
Description Length	14.6	2,389
Num of Dependencies	4.0	663
Num of Tests	6.6	1,079

Our AdaptEval benchmark has 164 adaptation tasks with 523 adaptations. As illustrated in Table III, the average line-of-code (LOC) of each task is 18.9. In each task, LLMs are required to perform 3.2 adaptations on average, while the average number of corresponding AST edits is 13.6. It demonstrates that our AdaptEval is composed of concise and higher-level adaptations. The average length of adaptation-level requirements is 14.6 words. Each task has 4.0 dependencies and is equipped with 6.6 test cases on average.

IV. EVALUATION SETUP

This section describes our evaluation setup to explore LLMs' adaptation ability based on AdaptEval's unique features.

A. Research Questions

We aim to benchmark LLMs' adaptation capabilities by addressing the following four research questions as follows:

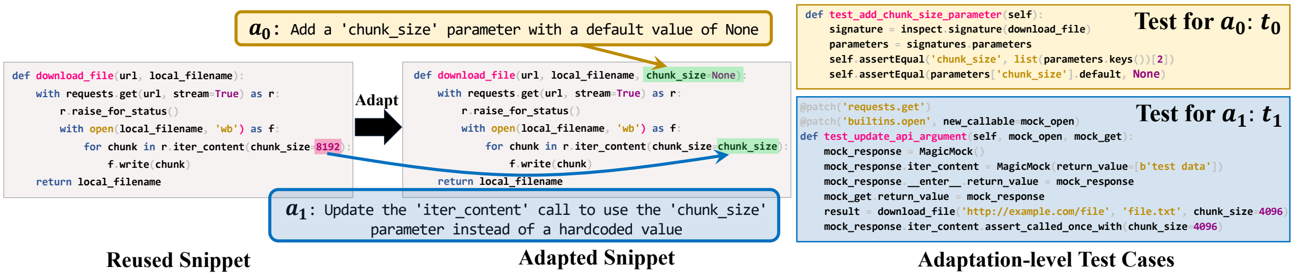


Fig. 3. An example of adaptation-level test cases in AdaptEval.

- **RQ1 (Task-Level Performance): How do LLMs perform on code snippet adaptation tasks across diverse scenarios?** We investigate LLMs' performance in solving whole adaptation tasks on AdaptEval with different settings.
- **RQ2 (Adaptation-Level Performance): How do LLMs perform in making different types of adaptations?** To investigate current bottlenecks of LLMs, we investigate their adaptation-level performance across different types.
- **RQ3 (Error Analysis): What are the prevalent errors made in LLMs' adaptations?** We further analyze LLMs' failed adaptations to better understand their limitations.
- **RQ4 (Potentials of Reasoning LLMs): What are the potentials of reasoning LLMs in adaptation?** Due to the distinct capabilities of reasoning LLMs, we separately evaluate their performance in a more challenging setting.

B. Studied Models

As the adaptation task requires both advanced coding and instruction-following abilities, we first select six state-of-the-art instruction-tuned LLMs. Particularly, we adopt four **general LLMs** for their capabilities in both natural language and coding tasks: GPT-4o (2024-11-20) [22], DeepSeek-V3 (2024-12-26) [76], Gemini-2.0-Flash (2025-02-05) [77] and Llama-3.3-70B (2024-12-06) [24]; and two **code LLMs**: Qwen2.5-Coder-14B-Instruct (2024-09-19) [27] and Codestral-22B (2024-05-29) [26], for their code-specific ability. To further explore LLMs' potentials in inferring developers' adaptation intents, we also include three **reasoning LLMs**, DeepSeek-R1 (2025-01-20) [29], Claude-3.7-Sonnet (2025-02-19) [23] and QwQ-32B (2025-03-06) [30] in our experiments. Our selected LLMs cover both closed- and open-source ones in leading series, all released since 2024, which ensures a comprehensive and timely evaluation of their adaptation ability.

C. Adaptation Settings

Since adaptation is a requirement-driven and context-dependent task, we propose a two-dimensional setting to evaluate LLMs' adaptation performance. First, we design three requirement settings, including **AReq**, **TReq** and **NoReq** to evaluate the influence of different instruction granularities. **AReq** provides LLMs with our specific adaptation-level instructions for each code change, while **TReq** offers only task-level requirement outlining developer intentions. To demonstrate the capabilities of reasoning LLMs, we introduce a more challenging setting, **NoReq** in RQ4, where LLMs must infer developer intentions from intra-file context and perform adaptation end-to-end without explicit requirements. As for the context setting, we consider two scenarios, *Oracle* and *NoCtx*, to assess the role of contextual dependencies. *Oracle* provides LLMs with all dependent context, simulating an ideal scenario with prior knowledge of dependencies. In contrast, *NoCtx* requires LLMs to adapt solely based on the given requirements, reflecting a more constrained setting.

I will provide you with a code snippet to reuse and a json file containing required adaptations. You should adapt the snippet into a target code base according to the adaptation requirements one by one. Some related code context from the target code base is provided for reference.

Reused Snippet:
`{CODE_OF_REUSED_SNIPPET}`

Adaptations Required to Perform:
```
[
  {'id': 0, 'type': ..., 'description': ...}
  ...
]
```

Related Code Context For Reference:
[Start of DEP_0]
`{CODE_OF_DEP_0}`
[End of DEP_0]
...

Please write out the function after adaptation in the following section:
Adapted Function:

Fig. 4. The prompt template for AdaptEval's evaluation (**AReq** & *Oracle*).

D. Prompt Design

We design a set of prompts to support LLMs' adaptations in different settings. Figure 4 shows the prompt template in the most informative setting (**AReq** and *Oracle*). The prompt begins with a brief description of the task. The first block provides the reused snippet as input, while the second block specifies our adaptation-level requirements step by step. The third block includes all dependent contexts. Finally, it offers an instruction for LLMs to generate the adapted function as output. Prompts for **TReq** and **NoReq** replace the second block with task-level instructions or intra-file context, respectively, while the *NoCtx* prompt removes the third block.

E. Evaluation Metrics

We adopt an execution-based metric pass@k to check the correctness of LLMs' adaptations. It measures the likelihood that a LLM produces a correct solution within k attempts based on unit test execution. Compared to similarity-based metrics, execution-based metrics can measure the functional correctness of the outputs with higher reliability [6].

In AdaptEval, we are interested in both the overall correctness of function-level adaptation task and individual adaptations, so we calculate both pass@k-t and pass@k-a. Specifically, if an adapted function passes all test cases of a task/adaptation, it will be considered as a correct sample for the task/adaptation. Considering the generation cost, we randomly generate five responses ($n = 5$) in line with previous work [35]. For task-level evaluation, we report pass@k metrics with $k \in \{1, 5\}$, reflecting LLMs' accuracy with effort control in practical use, *i.e.*, whether a correct adaptation can be made through at most five trials. For adaptation-level evaluation, we report the pass@1-a to solely evaluate their correctness.

F. Implementation Details

As LLMs are non-deterministic models, different settings for their randomness may lead to quite different results [78], [79]. In line with previous work [35], [46], we use nucleus sampling [80] to randomly generate five solutions with a temperature of 0.2 for all LLMs. All experiments are conducted with two GeForce RTX 4090-24G GPUs on Ubuntu 22.04.

V. Results

A. RQ1: Task-Level Performance

TABLE IV
THE TASK-LEVEL PERFORMANCE OF SIX INSTRUCTION-TUNED LLMS IN DIFFERENT SETTINGS OF ADAPTEVAL.

Model	**Context**	**TReq**		**AReq**	
		pass@1-t	pass@5-t	pass@1-t	pass@5-t
GPT-4o	*NoCtx*	**16.46**	**17.07**	52.20	**57.93**
	Oracle	**20.73**	**23.17**	**59.15**	**63.20**
DeepSeek-V3	*NoCtx*	15.85	**17.07**	**54.76**	57.32
	Oracle	19.39	20.12	57.31	60.37
Gemini-2.0	*NoCtx*	13.78	14.02	52.80	54.27
	Oracle	15.85	17.68	54.63	57.32
Llama-3.3	*NoCtx*	8.54	10.36	29.27	33.53
	Oracle	8.54	12.80	34.15	43.29
Qwen2.5-Coder	*NoCtx*	12.44	13.41	44.02	48.78
	Oracle	16.83	18.90	48.66	53.05
Codestral	*NoCtx*	11.58	12.19	46.71	51.22
	Oracle	14.51	17.07	50.98	54.88

Table IV shows the overall performance of six instruction-tuned LLMs. We use bold text to highlight the best pass@1-t and pass@5-t scores in each setting. Under the most informative setting (**AReq** & *Oracle*), we can observe that instruction-tuned LLMs can solve 34.15% to 59.15% tasks in AdaptEval. The closed-source GPT-4o achieves the best pass@1-t and pass@5-t of 59.15% and 63.20%. The second-tier LLMs include DeepSeek-V3 and Gemini-2.0, whose pass@1-t lag behind GPT-4o by 3.11% and 7.64%. Llama-3.3 ranks last among all LLMs, whose pass@1-t is 42.27% less than GPT-4o. We observe that code LLMs are inferior to the state-of-the-art general LLMs across all settings. This suggests that the instruction-following ability is more important in adaptation and simply pre-training LLMs' on code corpora does not necessarily improve their performance.

Compared with task-level requirements (**TReq**), our adaptation-level instructions (**AReq**) demonstrate a substantial improvement, with an average increase of 34.84% in pass@1-t. The relative growth of pass@1-t ranges from 185% to 299%. It implies that LLMs still struggle to reason for a complete adaptation solution from developer intentions only, but they could benefit from clear and actionable adaptation instructions. In the *Oracle* setting, most LLMs exhibit a 2% to 7% absolute improvement compared to the *NoCtx* setting. It indicates that retrieved context includes helpful information for adaptation.

Finding 1: Instruction-tuned LLMs can solve up to 59.15% tasks in AdaptEval. Compared with task-level requirements, our annotated adaptation steps significantly improve LLMs' performance with a rise of 34.84% in pass@1-t.

B. RQ2: Adaptation-Level Performance

TABLE V
LLMS' ADAPTATION-LEVEL PERFORMANCE ACROSS DIFFERENT TYPES.

Cat.	**Type**	GPT-4o	DS-V3	Gemini	Llama	Qwen	Code-stral
MS	Encap (13)	**8.0**	**8.0**	**8.0**	4.0	7.4	**8.0**
	Rnm_Func (70)	**67.6**	64.0	66.8	32.0	65.4	64.2
	Upd_Type (11)	9.6	**10.0**	**10.0**	4.0	9.8	9.0
	Ins_Param (13)	9.0	9.0	8.0	7.0	9.0	**9.4**
	Del_Param (5)	4.0	4.0	4.0	3.0	4.0	**4.2**
	Upd_Param (16)	**13.0**	**13.0**	12.0	7.0	**13.0**	12.4
	All Types (128)	**111.2**	108.0	108.8	57.0	108.6	107.2
	pass@1-a (%)	**86.87**	84.38	85.00	44.53	84.84	83.75
LC	Init_Var (3)	**2.0**	**2.0**	1.6	1.0	**2.0**	**2.0**
	Repl_Var (11)	**7.6**	6.2	7.0	5.0	6.6	6.4
	Upd_Const (13)	5.2	**8.6**	7.0	3.0	5.4	4.8
	Conv_Type (23)	17.2	16.4	**18.0**	9.0	17.6	17.8
	Ins_Call (22)	15.0	13.8	**15.8**	11.0	14.0	12.6
	Del_Call (8)	6.0	5.0	6.0	3.0	6.0	**7.2**
	Upd_Call (52)	**32.8**	**32.8**	32.0	20.0	28.0	25.4
	Ins_Import (16)	**7.2**	5.2	6.0	1.0	4.4	3.2
	Ins_Return (5)	2.2	**3.0**	**3.0**	2.0	**3.0**	2.8
	Del_Return (1)	**1.0**	0.8	**1.0**	0.0	0.0	0.0
	Upd_Return (8)	6.8	**7.8**	6.0	6.0	6.0	6.8
	Ins_Cond (31)	19.2	20.4	**21.6**	13.0	18.0	19.6
	Del_Cond (13)	**10.2**	8.2	9.0	6.0	9.0	8.4
	Upd_Cond (9)	**9.0**	8.2	**9.0**	6.0	6.2	8.4
	Handle_Ex (11)	**4.2**	4.0	4.0	3.0	**4.2**	4.0
	Ins_Func (1)	**1.0**	**1.0**	**1.0**	0.0	**1.0**	**1.0**
	All Types (227)	146.6	143.4	**148.0**	89.0	131.4	130.4
	pass@1-a (%)	64.58	63.17	**65.20**	39.21	57.89	57.44
RE	Ins_Ann (52)	**31.8**	28.4	28.6	13.0	28.6	29.0
	Upd_Ann (3)	1.0	1.0	1.0	0.0	1.0	**2.0**
	Del_Ann (1)	**1.0**	**1.0**	0.0	0.0	0.4	**1.0**
	Ins_Tmp_Var (9)	6.0	6.0	**7.0**	3.0	6.2	4.8
	Del_Tmp_Var (3)	**2.0**	**2.0**	**2.0**	1.0	**2.0**	**2.0**
	Rename_Param (24)	19.0	18.6	19.0	8.0	17.6	**20.2**
	Rename_Var (20)	16.0	15.0	16.0	11.0	14.6	**16.4**
	Upd_Api_Ref (25)	22.2	22.2	**22.8**	14.0	20.2	21.4
	Ins_Ret_None (2)	**2.0**	**2.0**	**2.0**	1.0	**2.0**	**2.0**
	Refact_Expr (17)	13.8	**14.8**	12.0	8.0	11.2	10.8
	Mov_Expr (1)	0.0	0.0	0.0	0.0	0.0	0.0
	Del_Import (3)	0.8	1.6	0.4	1.0	1.8	**2.0**
	Split_Line (2)	**1.0**	**1.0**	**1.0**	**1.0**	0.8	**1.0**
	Inline_Func (1)	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.0
	Expd_Func (3)	1.8	**2.2**	2.0	1.0	0.8	0.8
	Mov_Func (2)	**1.0**	**1.0**	**1.0**	0.0	0.0	**1.0**
	All Types (168)	**120.4**	117.8	115.8	63.0	108.2	114.4
	pass@1-a (%)	**71.67**	70.12	68.93	37.50	64.40	68.10
All Categories (523)		**378.2**	369.2	372.6	209.0	348.2	352.2
pass@1-a (%)		**72.31**	70.59	71.24	39.96	66.58	67.34

We further evaluate the adaptation-level performance of LLMs across each type under the most informative setting, i.e., **AReq** and *Oracle*, to explore LLMs' current strengths and limitations. The results are illustrated in Table V, which is measured by pass@1-a. The darker color of the cell indicates a higher score. The best performing GPT-4o can solve 72.31% adaptations on AdaptEval, while the worst Llama-3.3 can also achieve a pass@1-a about 40%. Taking GPT-4o as an example, its pass@1-t in task-level evaluation is only 59.15%. AdaptEval's fine-grained tests can reflect LLMs' performance more specifically, despite their failures in the overall task.

Based on our adaptation taxonomy, AdaptEval allows the evaluation of LLMs on different adaptation types. Specifically, LLMs obtain higher pass@1-a scores on *Method Signature* than on *Logic Customization*, with a 20.31% gap on average. We can observe that adaptations like function renaming are well-handled, with over 90% of them are resolved by the top-tier LLMs. Most *Logic Customization* adaptations have lower resolution rates, *e.g.*, exception handling. The results indicate that LLMs excel at solving adaptations with straightforward solutions, while for adaptations require complex understanding and highly customized implementations, it is difficult for LLMs to carry out solutions that fully meet developers' needs. Different LLMs have advantages in different adaptation types, *e.g.*, GPT-4o, DeepSeek-V3, Gemini-2.0 and Codestral all achieve the best performance in more than 15 types. Although code LLMs perform comparably to general LLMs in *Method Signature* and *Refactoring*, they are less effective in logical adaptations. The possible reason may be their failures to fully understand the natural language instructions.

Finding 2: Based on AdaptEval's fine-grained evaluation, we find that LLMs perform best on *Method Signature* and worst on *Logic Customization*, with a gap of 20.31%, as the latter requires complex understanding and implementation.

C. RQ3: Error Analysis

TABLE VI
THE ERROR DISTRIBUTION IN LLMS' ADAPTATIONS.

Model	Top Error Types			
	AssertionError	CompilationError	NameError	TypeError
GPT-4o	**226 (56.78%)**	69 (17.34%)	35 (8.79%)	28 (7.04%)
DS-V3	**223 (54.93%)**	82 (20.20%)	48 (11.82%)	23 (5.67%)
Gemini-2.0	**261 (59.59%)**	79 (18.04%)	40 (9.13%)	35 (7.99%)
Llama-3.3	105 (18.42%)	**410 (71.93%)**	20 (3.51%)	10 (1.75%)
Qwen2.5	**304 (59.49%)**	80 (15.66%)	73 (14.29%)	24 (4.70%)
Codestral	**256 (53.78%)**	89 (18.70%)	68 (14.29%)	28 (5.88%)

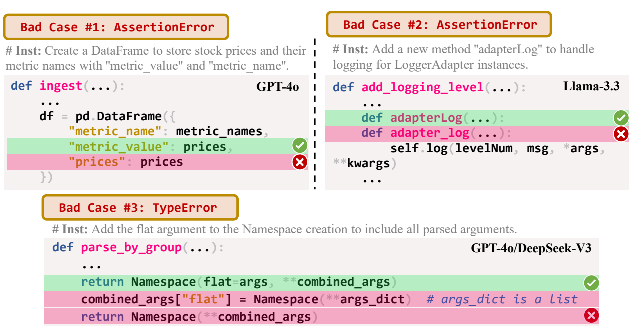


Fig. 5. Representative examples of LLMs' adaptation failures.

This section further analyzes the errors in LLMs' adaptations under the **AReq** and *Oracle* setting. Table VI shows their error distributions. We identified 16 distinct types in total, but only retain the most frequent ones for clarity.

AssertionError is the dominant reason of failed adaptations. It indicates LLMs' misunderstanding or misimplementation of provided requirements. As shown in the first example of Figure 5, GPT-4o wrongly used the variable name "*prices*" rather than the required "*metric_value*" to create the dataframe. In the second example, Llama-3.3 was instructed to add a new "*adapterLog*" method but unexpectedly named it as "*adapt_log*". These failures highlight their reliance on the pre-training knowledge, *e.g.*, the snake case naming style, rather than strictly following instructions. For *CompilationErrors*, the weakest LLM, Llama-3.3, often generates syntax-incorrect adaptations, whereas stronger LLMs rarely do. Such errors often arise during integration, such as incorrect method name. The third most frequent type is *NameError*, caused by undefined identifiers. We observe that code LLMs rise more errors in this type. Combined with results in Table V (Ins_Import and Del_Import), this suggests their failures to include required libraries locally. *TypeErrors* arise from illegal operands or inconsistent arguments with method signature. In the last example, AdaptEval asked GPT-4o and DeepSeek-V3 to add a *flat* argument. However, they used a *list* rather than a *dict* to create the "*Namespace*" object, leading to a *TypeError*.

Finding 3: The most prevalent error raised by LLMs is *AssertionError*, followed by *CompilationError* and *NameError*. We observe that LLMs may still hold on to their pre-training knowledge and violates our explicit instructions.

D. RQ4: Potentials of Reasoning LLMs

TABLE VII
THE ADAPTATION PERFORMANCE OF REASONING LLMS.

Model	NoReq		TReq		AReq	
	p@1-t	p@1-a	p@1-t	p@1-a	p@1-t	p@1-a
DS-R1	**19.51**	**32.50**	20.12	21.99	59.76	**76.10**
Claude-3.7	15.85	31.93	**23.78**	**23.52**	**60.98**	72.85
QwQ	15.24	23.52	18.90	19.50	56.71	71.89
GPT-4o	12.19	26.58	20.73	19.92	59.15	72.31
Qwen2.5	9.63	21.80	16.83	16.14	48.66	66.58

In this section, we investigate the potentials of reasoning LLMs on adaptation by conducting experiments in the **NoReq** setting and examining their reasoning paths. Table VII illustrates the adaptation performance of reasoning LLMs on AdaptEval. Compared with the best general and code LLMs, *i.e.*, GPT-4o and Qwen2.5-Coder, reasoning LLMs achieve significantly higher accuracy in the **NoReq** setting, with up to 102% higher pass@1-t. We also observe that LLMs achieve higher pass@1-a score under **NoReq** than **TReq**, especially for DeepSeek-R1. It successfully solves 51 out of 114 *Method Signature* adaptations that **TReq** fails to handle. However, in the 59 cases where **TReq** succeeds, 38 are *Logic Customization*. Based on this, we further study the top types of adaptations that correctly completed by reasoning LLMs. As shown in Table VIII, they excel at *Method Signature* adaptations, including updating method type or renaming. The reason is that key information for these adaptations, *e.g.*, usage or naming convention, is included in the context. An example of a helpful reasoning path of DeepSeek-R1 for adaptation is shown in Figure 6(a). However, for adaptations like *Logic Customization*, reasoning LLMs cannot precisely infer developers' intents merely from the context, resulting in their low accuracy. Our result highlights the effectiveness of reasoning LLMs in finding implicit contextual cues rather than speculating developers' requirements.

Although reasoning LLMs in the **NoReq** setting approach their performance in **TReq**, they still lag significantly behind **AReq**. It underscores that while their reasoning capabilities can partially bridge the gap in requirements, further advancements are needed to fully align their reasoning process with developers' step-by-step adaptation strategies. It is also notable that instruction-tuned LLMs like GPT-4o achieves comparable performance with reasoning LLMs in **AReq**. This implies that their reasoning capability offers less assistance than in **NoReq**. To understand their limitations in following instructions, we analyze the failed cases by manually inspecting their reasoning paths. We choose DeepSeek-R1 and GPT-4o for their best performance in the adaptation-level performance (pass@1-a). Among 523 adaptations, DeepSeek-R1 failed to perform 22 adaptations where the instruction-tuned GPT-4o succeeded. In 14 of them, DeepSeek-R1 expressed direct doubts to developers' instructions and refused to follow them. As shown in Figure 6(b), DeepSeek-R1 failed to update the return statement by claiming that "maybe the user made a mistake" in its reasoning path. The remaining 8 cases include similar harmful self-reflection of its own correct reasoning instead of provided instructions. However, GPT-4o correctly completed these adaptations by following the instructions faithfully. The result highlights that the improved reasoning ability makes LLMs more likely to reflect on their inputs and hence may sometimes harm their instruction-following ability. This also implies the importance of their further training on the requirement-driven adaptation data.

TABLE VIII
TOP TYPES OF ADAPTATIONS CORRECTLY COMPLETED BY REASONING LLMS IN THE *NoReq* SETTING.

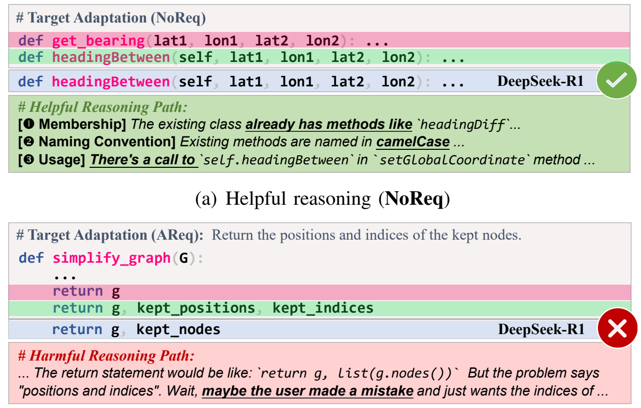
Model	Method Signature		Logic Customization		Refactoring	
	Type	**p@1-a**	**Type**	**p@1-a**	**Type**	**p@1-a**
DS-R1	Upd_Type (11)	81.82%	Ins_Import (16)	43.75%	Del_Import (3)	66.66%
	Encap (13)	61.54%	Repl_Var (11)	36.36%	Upd_Api_Ref (25)	64.00%
	Rnm_Func (70)	60.00%	Init_Var (3)	33.33%	Del_Tmp_Var (3)	33.33%
	Average	50.78%	**Average**	25.11%	**Average**	28.57%
Claude	Upd_Type (11)	90.91%	Handle_Ex (11)	45.45%	Upd_Api_Ref (25)	68.00%
	Rnm_Func (70)	62.86%	Ins_Return (5)	40.00%	Del_Import (3)	66.67%
	Encap (13)	53.85%	Ins_Import (16)	37.50%	Ins_Ret_None (2)	50.00%
	Average	51.56%	**Average**	24.67%	**Average**	24.40%
QwQ	Upd_Type (11)	54.55%	Handle_Ex (11)	45.45%	Del_Import (3)	66.66%
	Encap (13)	53.85%	Ins_Return (5)	40.00%	Upd_Api_Ref (25)	64.00%
	Rnm_Func (70)	44.29%	Init_Var (3)	33.33%	Ins_Ret_None (2)	50.00%
	Average	37.50%	**Average**	18.06%	**Average**	20.24%

Finding 4: Reasoning LLMs significantly outperform instruction-tuned LLMs in adaptation when no requirements are provided. However, their reasoning process still need more alignments to developers' actual adaptations.

VI. DISCUSSION

A. Data Leakage Issue

As current LLMs may include various data sources including GitHub in their training corpus, evaluating them on the



(a) Helpful reasoning (**NoReq**)

(b) Harmful reasoning (**AReq**)

Fig. 6. Adaptation examples of reasoning LLMs.

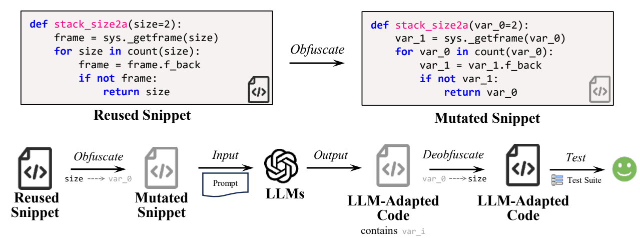


Fig. 7. The mutation-based approach to detecting and mitigating data leakage.

leaked data may impact the effectiveness of benchmarks. It is essential to measure the extent of data leakage of AdaptEval on our selected LLMs and provide leakage mitigation approaches for its future use. In our study, we adopt CDD [81] and a mutation-based evaluation [82], [83] to detect data leakage. Specifically, CDD assumes that LLMs with the data leakage issue are more prone to generate outputs that resemble their training data. Following their assumption, we check whether the sampled outputs of our used LLMs are highly similar to the possibly leaked data. Regarding the mutation-based approach, it first modifies the original code input to LLMs before adaptation, i.e., renaming all the identifiers, and compare LLMs' performance in adapting the mutated code with the original one. If they use their memorization of the leaked data, their performance on the unseen data, i.e., the mutated code, will be degraded. Otherwise, the extent of performance decrease should be low because they perform the task with the same reasoning ability. Our implemented mutation-based approach is depicted in Figure 7. The evaluation results of AdaptEval's data leakage issue are shown in Table IX. Compared to the benchmarks suffered from data leakage, i.e., a 41.17% Leak Ratio on HumanEval and a 52.27% on Defect4J (both evaluated with GPT-3.5), AdaptEval only exhibits a leak ratio of less than 2% and an up to 7.36% degradation in the pass@1 score, suggesting a low risk of data leakage on all LLMs used. Besides, our mutation-based approach could also mitigate the data leakage risk in future evaluation. Even if the ground truth

TABLE IX
THE EXTENT OF DATA LEAKAGE IN ADAPTEVAL.

Model	Leak Ratio (%)	$\triangle$Pass@1-t	$\triangle$Pass@1-a
GPT-4o	1.22	-2.45 (-4.14%)	-2.14 (-2.96%)
DeepSeek-V3	1.22	-3.04 (-5.30%)	-2.71 (-3.84%)
Gemini-2.0	1.22	**-4.02 (-7.36%)**	**-4.51 (-6.33%)**
Llama-3.3	0.61	-1.83 (-5.36%)	-1.53 (-3.83%)
Qwen2.5-Coder	0.61	-1.71 (-3.51%)	-1.76 (-2.64%)
Codestral	0.61	-2.20 (-4.32%)	-0.99 (-1.47%)
DeepSeek-R1	1.22	-0.62 (-1.04%)	-2.10 (-2.76%)
Claude-3.7	1.22	-0.61 (-1.00%)	-0.57 (-0.78%)
QwQ-32B	1.22	+0.61 (+1.08%)	-2.10 (-2.92%)

code is learned by LLMs, we can use the mutated code to conduct evaluation on AdaptEval.

B. Implications

We pose a new benchmark to advance research and practice in code snippet reuse. While code snippet reuse is ubiquitous in development, the adaptation process remains a significant bottleneck. Unlike existing code generation benchmarks, AdaptEval provides a focused evaluation framework for assessing LLMs' adaptations ability. It opens new directions to improve LLMs for reuse-specific tasks and calls for tools to streamline adaptation and reduce human efforts.

LLMs' pre-training knowledge may hinders their ability to understand and adhere to user instructions. The adaptation-level annotations in AdaptEval enable a fine-grained analysis of LLMs' instruction following ability. Our findings show that although LLMs significantly benefit from specific and actionable guidance, they may still fail to follow the explicit instructions. This suggests LLMs' weakness in handling the conflict between their internal pre-training knowledge and external instructions. For researchers, our study highlights the need to investigate methods for better coordination between LLMs' instruction following abilities and pre-training knowledge to enhance their performance.

More task-specific and context-aware techniques are expected to facilitate the resolution of the adaptation task. Though being prompted with specific instructions, current LLMs may still struggle with complex adaptations, *e.g.*, *Logic Customization*) due to their lack of pre-existing knowledge, accurate reasoning and context understanding. To this end, future studies could consider specialized training on sub-tasks such as dependency handling, or retrieval-augmented techniques to utilize associated contextual information, including SO discussions and development context.

VII. THREATS TO VALIDITY

Threats in data collection. AdaptEval only includes the scenario of adapting code snippets from Stack Overflow. Our evaluation results may not be generalized to other adaptation scenarios, as well as future software development process as LLMs become better at code generation. Although our SO post filtration considers the data coverage, it may miss valuable adaptations on posts with negative scores, e.g., corrective changes to address quality issues. To filter spurious code reuse, we combine link identification with clone detection techniques. However, this may ignore cases without references or with substantial adaptations. **Threats in data annotation.** Human annotation of adaptation types and descriptions may introduce subjective bias. To alleviate this issue, we employ a rigorous annotation process by referring the actual commit history and ensuring final annotations are approved by all annotators. **Threats in benchmark scale.** Another potential threat is the benchmark scale. Considering data leakage and the authenticity of code reuse, we try our best to include all available cases that satisfy our criteria. However, we only focus on adaptations in a single programming language and under several controlled scenarios. Our findings may not generalize to adaptations in other languages and scenarios. Hence, we plan to extend our benchmark to support multiple languages and more scenarios in the future work. **Threats in evaluation.** The manual creation of prompts and tests may bias our evaluation results. To mitigate this, we follow the best practices [84] for prompt design. For test cases, we conduct a rigorous checking process and plan to release them to the community for validation. The adaptation-level evaluation are only conducted in the ideal setting for investigation current LLMs' boundaries, while our conclusion may not be held in other scenarios. Though we adopt a data leakage mitigation strategy for AdaptEval, there will still be chance that LLMs learn our benchmark in the future. We will continue to maintain and update our benchmark for the community.

VIII. CONCLUSION

This paper proposes AdaptEval benchmark to evaluate LLMs' capabilities in code snippet adaptation. It is equipped with practical context, multi-granular annotations and fine-grained evaluation. Based on AdaptEval, we conduct an empirical study on six instruction-tuned LLMs and especially three reasoning LLMs. Results demonstrate that LLMs perform better with actionable instructions than task-level intentions. They achieve lower performance in adaptations that require complex understanding or implementation. For reasoning LLMs, they excel at inferring implicit contextual cues but their reasoning process still deviates from developers' actual adaptation strategies. Our benchmark supports further exploration of applying LLMs in code reuse and adaptation tasks.

ACKNOWLEDGMENTS

We gratefully acknowledge the support from the National Key Research and Development Program of China (Grant No.2023YFB4503802) and the National Natural Science Foundation of China (Grant No.62302515, No.62172426, and No.62332005).

REFERENCES

[1] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, "Automatic Programming: Large Language Models and Beyond," May 2024. arXiv:2405.02213 [cs].

[2] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration Code Generation via ChatGPT," *ACM Transactions on Software Engineering and Methodology*, p. 3672459, June 2024.

[3] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," June 2024. arXiv:2406.00515 [cs].

[4] C. S. Xia and L. Zhang, "Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," Apr. 2023. arXiv:2304.00385 [cs].

[5] A. Z. Yang, S. Kolak, V. J. Hellendoorn, R. Martins, and C. L. Goues, "Revisiting unnaturalness for automated program repair in the era of large language models," *arXiv preprint arXiv:2404.15236*, 2024.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," 2021.

[7] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?," 2024.

[8] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Writing Code to Prototype, Ideate, and Discover," *IEEE Software*, vol. 26, no. 5, pp. 18–24, 2009.

[9] M. Gharehyazie, B. Ray, and V. Filkov, "Some from Here, Some from There: Cross-Project Code Reuse in GitHub," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, (Buenos Aires, Argentina), pp. 291–301, IEEE, May 2017.

[10] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack Overflow in Github: Any Snippets There?," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 280–290, 2017.

[11] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?," *Empirical Software Engineering*, vol. 24, pp. 637–673, Apr. 2019.

[12] Y. Huang, F. Xu, H. Zhou, X. Chen, X. Zhou, and T. Wang, "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, (New York, NY, USA), pp. 548–559, Association for Computing Machinery, 2022. event-place: Virtual Event.

[13] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and Supporting Adaptation of Online Code Examples," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pp. 316–327, IEEE Press, 2019. event-place: Montreal, Quebec, Canada.

[14] T. Zhang, Y. Lu, Y. Yu, X. Mao, Y. Zhang, and Y. Zhao, "How do developers adapt code snippets to their contexts? an empirical study of context-based code snippet adaptations," *IEEE Transactions on Software Engineering*, pp. 1–20, 2024.

[15] S. Baltes and S. Diehl, "Usage and attribution of Stack Overflow code snippets in GitHub projects," *Empirical Software Engineering*, vol. 24, pp. 1259–1295, June 2019.

[16] R. Cottrell, R. J. Walker, and J. Denzinger, "Jigsaw: a tool for the small-scale reuse of source code," in *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, (Leipzig, Germany), p. 933, ACM Press, 2008.

[17] V. Terragni and P. Salza, "APIzation: Generating Reusable APIs from StackOverflow Code Snippets," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Melbourne, Australia), pp. 542–554, IEEE, Nov. 2021.

[18] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing," July 2021. arXiv:2107.13586 [cs].

[19] T. Zhang, Y. Yu, X. Mao, S. Wang, K. Yang, Y. Lu, Z. Zhang, and Y. Zhao, "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)*, pp. 566–577, 2025.

[20] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, (Red Hook, NY, USA), Curran Associates Inc., 2024.

[21] OpenAI, "gpt-3.5-turbo," 2023.

[22] OpenAI, "gpt-4o," 2024.

[23] Anthropic, "Claude-3.7-sonnet," 2025.

[24] Meta, "Meta llama 3," 2024.

[25] DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, *et al.*, "DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence," June 2024. arXiv:2406.11931 [cs].

[26] MistralAI, "Codestral," 2024.

[27] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[28] OpenAI, A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, *et al.*, "OpenAI o1 System Card," Dec. 2024. arXiv:2412.16720 [cs].

[29] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, *et al.*, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," Jan. 2025. arXiv:2501.12948 [cs].

[30] Q. Team, "Qwq-32b: Embracing the power of reinforcement learning," March 2025.

[31] X. Li, Z. Yu, Z. Zhang, X. Chen, Z. Zhang, Y. Zhuang, N. Sadagopan, and A. Beniwal, "When Thinking Fails: The Pitfalls of Reasoning for Instruction-Following in LLMs," May 2025. arXiv:2505.11423 [cs].

[32] T. Fu, J. Gu, Y. Li, X. Qu, and Y. Cheng, "Scaling Reasoning, Losing Control: Evaluating Instruction Following in Large Reasoning Models," May 2025. arXiv:2505.14810 [cs].

[33] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[34] T. Liu, C. Xu, and J. McAuley, "Repobench: Benchmarking repository-level code auto-completion systems," 2023.

[35] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[36] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," in *Advances in Neural Information Processing Systems* (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, pp. 46701–46723, Curran Associates, Inc., 2023.

[37] Q. Peng, Y. Chai, and X. Li, "HumanEval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)* (N. Calzolari, M.-Y. Kan, V. Hoste, A. Lenci, S. Sakti, and N. Xue, eds.), (Torino, Italia), pp. 8383–8394, ELRA and ICCL, May 2024.

[38] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "Effibench: Benchmarking the efficiency of automatically generated code," 2024.

[39] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A code efficiency benchmark for code large language models," 2024.

[40] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories," Mar. 2024. arXiv:2404.00599 [cs].

[41] Y. Xie, A. Xie, D. Sheth, P. Liu, D. Fried, and C. Rose, "Codebenchgen: Creating scalable execution-based code generation benchmarks," 2024.

[42] B. Li, W. Wu, Z. Tang, L. Shi, J. Yang, J. Li, S. Yao, C. Qian, B. Hui, Q. Zhang, Z. Yu, H. Du, P. Yang, D. Lin, C. Peng, and K. Chen, "Devbench: A comprehensive benchmark for software development," 2024.

[43] S. Wang, L. Ding, L. Shen, Y. Luo, B. Du, and D. Tao, "OOP: Object-oriented programming evaluation benchmark for large language models," in *Findings of the Association for Computational Linguistics: ACL 2024* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), (Bangkok, Thailand), pp. 13619–13639, Association for Computational Linguistics, Aug. 2024.

[44] K. Deng, J. Liu, H. Zhu, C. Liu, J. Li, J. Wang, P. Zhao, C. Zhang, Y. Wu, X. Yin, Y. Zhang, W. Su, B. Xiang, T. Ge, and B. Zheng, "R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models," 2024.

[45] A. Deshpande, A. Agarwal, S. Shet, A. Iyer, A. Kanade, R. Bairi, and S. Parthasarathy, "Class-level code generation from natural language using iterative, tool-enhanced reasoning over repository," 2024.

[46] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu, "JavaBench: A Benchmark of Object-Oriented Code Generation for Evaluating Large Language Models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, (Sacramento CA USA), pp. 870–882, ACM, Oct. 2024.

[47] J. Feng, J. Liu, C. Gao, C. Y. Chong, C. Wang, S. Gao, and X. Xia, "ComplexCodeEval: A Benchmark for Evaluating Large Code Models on More Complex Code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, (Sacramento CA USA), pp. 1895–1906, ACM, Oct. 2024.
[48] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Hui, W. Liu, Z. Liu, and M. Sun, "Debugbench: Evaluating debugging capability of large language models," 2024.
[49] J. Guo, Z. Li, X. Liu, K. Ma, T. Zheng, Z. Yu, D. Pan, Y. LI, R. Liu, Y. Wang, S. Guo, X. Qu, X. Yue, G. Zhang, W. Chen, and J. Fu, "Codeeditorbench: Evaluating code editing capability of large language models," 2024.
[50] C. Niu, C. Li, V. Ng, and B. Luo, "Crosscodebench: Benchmarking cross-task generalization of source code models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 537–549, 2023.
[51] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, "xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval," 2023.
[52] L. Fu, H. Chai, S. Luo, K. Du, W. Zhang, L. Fan, J. Lei, R. Rui, J. Lin, Y. Fang, Y. Liu, J. Wang, S. Qi, K. Zhang, W. Zhang, and Y. Yu, "Codeapex: A bilingual programming evaluation benchmark for large language models," 2024.
[53] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, H. Sundaram, and S. Deng, "Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation," 2024.
[54] Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang, "Coderujb: An executable and unified java benchmark for practical programming scenarios," 2024.
[55] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, and Y. Li, "Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories," 2024.
[56] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," 2024.
[57] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, "Deep Learning Based Code Generation Methods: A Literature Review," Mar. 2023. arXiv:2303.01056 [cs].
[58] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," 2023.
[59] H. Eladawy, C. Le Goues, and Y. Brun, "Automated program repair, what is it good for? not absolutely nothing!," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.
[60] C. Reux, M. Acher, D. E. Khelladi, O. Barais, and C. Quinton, "Llm code customization with visual results: A benchmark on tikz," 2025.
[61] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.
[62] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.
[63] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language Models are Few-Shot Learners," July 2020. arXiv:2005.14165 [cs].
[64] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pp. 560–564, IEEE, 2021.
[65] GitHub, "Github rest api," 2022. Accessed: 2024-04.
[66] K. R. Srinath, "Python – the fastest growing programming language," 2017.
[67] Stack, "Exchange api v2.3," 2021. Accessed: 2024-04.
[68] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "Investigating Context Adaptation Bugs in Code Clones," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (Cleveland, OH, USA), pp. 157–168, IEEE, Sept. 2019.
[69] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, (Austin Texas), pp. 1157–1168, ACM, May 2016.
[70] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, (Vasteras Sweden), pp. 313–324, ACM, Sept. 2014.
[71] tree sitter, "Tree-sitter," 2024. Accessed: 2024-05.
[72] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "ClDiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (Montpellier France), pp. 679–690, ACM, Sept. 2018.
[73] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.
[74] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, pp. 37 – 46, 1960.
[75] V. Khorikov. 2020.
[76] DeepSeek-AI, "Deepseek-v3 technical report," 2024.
[77] Google-DeepMind, "Gemini-2.0-flash," 2025.
[78] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," 2023.
[79] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting copilot and codex: Hot temperature, cold prompts, or black magic?," 2023.
[80] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The Curious Case of Neural Text Degeneration," Sept. 2019.
[81] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models," in *Findings of the Association for Computational Linguistics ACL 2024*, (Bangkok, Thailand and virtual meeting), pp. 12039–12050, Association for Computational Linguistics, 2024.
[82] J. Kong, X. Xie, and S. Liu, "Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework," *Proceedings of the ACM on Software Engineering*, vol. 2, pp. 2712–2734, June 2025. Publisher: Association for Computing Machinery (ACM).
[83] Y. Zhang, Y. Xie, S. Lit, K. Liu, C. Wang, Z. Jia, X. Huang, J. Song, C. Luo, Z. Zheng, R. Xu, Y. Liu, S. Zheng, and X. Liao, "Unseen horizons: Unveiling the real capability of llm code generation beyond the familiar," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 604–615, 2025.
[84] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A Survey of Large Language Models," June 2023. arXiv:2303.18223 [cs].