How Should We Build A Benchmark? Revisiting 274 Code-Related Benchmarks For LLMs

Anonymous ACL submission

Abstract

Various benchmarks have been proposed to assess the performance of large language models (LLMs) in different coding scenarios. We refer to them as code-related benchmarks. However, there are no systematic guidelines by which such a benchmark should be developed to assure its quality, reliability, and reproducibility. We propose ***How2Bench*** comprising a 55-criteria checklist as a set of guidelines to comprehensively govern the development of code-related benchmarks. Using HOW2BENCH, we profiled 274 code-related benchmarks released within the past decade and found concerning issues. Nearly 70% of the benchmarks did not take measures for data quality assurance; over 10% did not even open source or only partially open source. Many highly cited benchmarks have loopholes, including duplicated samples, incorrect reference codes/tests/prompts, and unremoved sensitive/confidential information. Finally, we conducted a human study involving 49 participants and ***revealed significant gaps in awareness*** of the importance of data quality, reproducibility, and transparency. For ease of use, we provide a ***printable version*** of HOW2BENCH in Appendix F.

1 Introduction

> ❝ *Awareness is the beginning of action; action is the fulfillment of awareness.* ❞
> — Yangming Wang (1472 - 1529)

Recent large language models (LLMs) have shown remarkable capabilities across various domains such as software development (Chen et al., 2021a), question answering (Rogers et al., 2023), and math reasoning (Imani et al., 2023). Various ***benchmarks*** (Chen et al., 2021a; Jimenez et al., 2024; Austin et al., 2021; Yue et al., 2024; Du et al., 2023a) are proposed to evaluate LLMs' effectiveness and limitations from multiple perspectives in different application scenarios.

However, ***doubts regarding the quality, reliability, and transparency*** of various code-related benchmarks arise. For example, a recent study pointed out that "current programming benchmarks are inadequate for assessing the actual correctness of LLM-generated code" (Liu et al., 2023a). Other accusations, including ***irreproducible*** (Reuel et al., 2024), ***closed*** data sources (Cao et al., 2024b), ***low quality*** (Qiu et al., 2024a; Yadav et al., 2024a), and ***inadequate validation measures*** (Liu et al., 2023a), were also raised, undermining the credibility of these benchmarks and thereby their subsequent evaluation results. This motivates the need for rigorous and thorough ***guidelines*** to govern code-related benchmark development.

In this paper, we introduce **HOW2BENCH, a comprehensive guideline** consisting of a 55-criteria checklist specially designed for code-related benchmarks. This checklist ***covers the entire lifecycle*** of benchmark development, from *design* and *construction* to *evaluation*, *analysis*, and *release* as shown in Figure 1. It underwent multiple iterations — we initiated a draft inspired by open-source software guidelines (Fogel, 2005) and classical measurement theory (Suppes et al., 1962). We refined it through iterative discussions with practitioners, leading to the finalization of these criteria. HOW2BENCH emphasizes ***reliability***, ***validity***, ***open access***, and ***reproducibility*** in the benchmark development, ensuring high standards and fostering a more reliable and transparent environment.

Following HOW2BENCH, we conducted an in-depth profiling of 270+ code-related benchmarks developed over the past decade (2014 - 2024). The extent of criteria violations by the profiled benchmarks is concerning:

- Almost 70% of the benchmarks did not take any measures for ***data quality assurance***;
- Over 90% did not consider ***code coverage*** when use passing test cases as an oracle;
- Over half of the benchmarks did not provide

the essential information (e.g., experiment setup, prompts) for ***reproducibility***;

• Over 10% are ***not open source*** or only partially open source.

We observed that even ***highly cited benchmarks have loopholes***, including duplicated samples, incorrect reference/tests, unclear displays, and unremoved sensitive/confidential information. We also observed these loopholes can ***propagate***. Over 18% of the benchmarks serve as data sources for subsequent benchmarks (Figure 8). Therefore, the data quality of benchmarks affects their credibility and likely impacts future benchmarks.

To understand the usefulness of the criteria in How2Bench, we conducted a human study involving 49 participants through questionnaires. All participants ***concurred on the necessity*** of having a checklist for benchmark construction to enhance quality. Nearly all participants with experience in benchmark development ***acknowledged the importance of all these 55 criteria***. The study also exposed ***gaps in quality awareness***: 16% of participants were unaware of the necessity for data denoising, and over 40% were not aware that the experimental setup and environment could impact the reproducibility and transparency.

Given the identified severe situation, this paper calls for attention to the quality of LLM benchmarks due to insufficient rigor and lack of systematic, reproducible evaluation workflows. We noticed that this issue not only stems from the significant effort required to construct a rigorous benchmark, but also from a lack of awareness regarding how crucial a rigorous development process is to the quality of benchmarks and the reliability of evaluations. Without being aware of and addressing the quality of code-related benchmarks (as well as other kinds of benchmarks for LLMs), the reliability and reproducibility of benchmark results remain compromised, misleading research directions and hindering meaningful progress.

Therefore, we advocate for establishing comprehensive, transparent, and enforceable guidelines for benchmark development and quality assurance, and took an initial step, introducing HOW2BENCH. This intention also echoed several prior works (McIntosh et al., 2025; Reuel et al., 2024; Holloway et al., 1999; Zairi, 2010; Francis and Holloway, 2007; Wu et al., 2025). We hope HOW2BENCH could be helpful to elevate the standards and trustworthiness of future benchmarks.

This paper makes contributions in five aspects:

- ***Novelty***. We introduce HOW2BENCH, a comprehensive set of guidelines packaged as a 55-criteria checklist that covers the lifecycle of code-related benchmark development.
- ***Significance***. HOW2BENCH presents the first comprehensive set of actionable guidelines for developing high-quality benchmarks, striving to create a more reliable and transparent environment. The human study also highlighted the demand for such a detailed guideline.
- ***Usefulness***. HOW2BENCH serves as a guideline for practitioners before/during developing code-related benchmarks, and a checklist for evaluating existing benchmarks after their release. For ease of use, we also provide a ***printable version*** of HOW2BENCH on Appendix F.
- ***Generalizability***. Most criteria listed in HOW2BENCH can be adopted or adapted to other benchmarks such as Question-answering, mathematical reasoning, and multi-modal benchmarks.
- ***Long-term Impact***. Our statistics alert the community to the severity and prevalence of non-standard practices in benchmark development. It ultimately improves the overall quality of benchmarks due to the propagation effect among them.

2 Background

2.1 Code-related Benchmarks

Benchmarks for coding tasks like code generation (Chen et al., 2021a; Austin et al., 2021), defect detection (Just et al., 2014; Gao et al., 2023b; Liu et al., 2024c), and program repair (Jimenez et al., 2024; Risse and Böhme, 2024) are increasingly common, reflecting the growing needs for using LLMs for coding tasks. Recent studies have highlighted various issues with these benchmarks, ranging from design inconsistencies to scope and applicability limitations. For example, (Liu et al., 2023a) found that even some widely used benchmarks, such as HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021), contains a non-trivial proportion of bugs in implementation, documentation, and test cases. Our work, in comparison, introduces a detailed guideline that ***guides the benchmark development*** during the entire lifecycle.

2.2 Related Studies and Surveys

Several recent surveys and empirical studies have profiled the status quo of LLM development. These

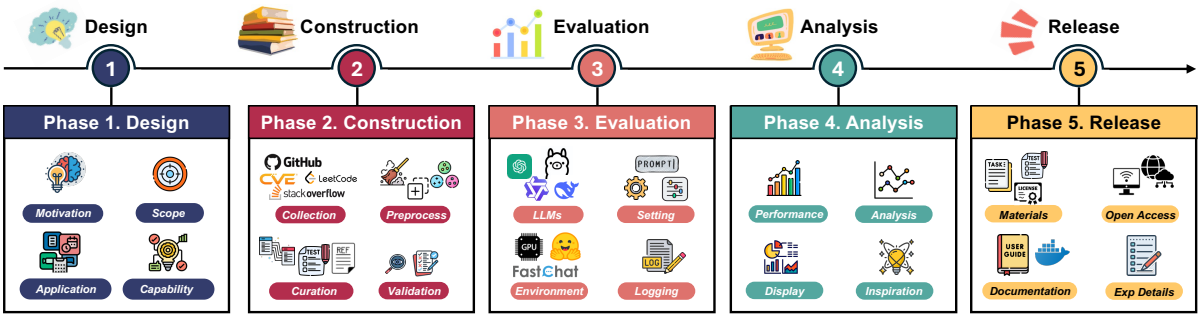

Figure 1: Lifecycle of Benchmark Development

studies either explore the overall performance for certain areas such as software engineering (Hou et al., 2023; Wang et al., 2024a) or investigate the capabilities of LLMs on specific tasks such as code generation (Dou et al., 2024; Yu et al., 2024) and test generation (Schäfer et al., 2024; Yuan et al., 2024b, 2023a). A survey (Chang et al., 2024) about how to evaluate LLMs was proposed to answer what/where/how to evaluate LLMs. This paper differs from these studies in its purpose and perspectives. Unlike these benchmarks, our work proposes guidelines for future benchmark development and provides a checklist to assess the quality of these existing benchmarks.

Recently, BetterBench (Reuel et al., 2024) is a concurrent work assessing the AI benchmarks against 46 criteria. Then, it scored 24 AI benchmarks in various domains and ranked them. BetterBench differs from this paper in several key aspects: scope (general benchmarks vs. code-related benchmarks), lifecycle division (it addresses benchmark retirement, while How2Bench focuses on benchmark evaluation, analysis, and release), and objectives (scoring benchmarks vs. offering comprehensive guidelines for future benchmark development). Additionally, the study in this paper was conducted on a much larger scale (24 vs. 274 benchmarks), statistically highlighting the prevalent issues in existing benchmarks.

3 Guideline Design

3.1 The Lifecycle of Benchmark Development

Code-related benchmark development comprises five typical phases (Phase 1 - 5), as shown in Figure 1, explained in detail as follows.

Phase 1. Design. At the beginning of benchmark development, it is vital to identify the motivation, the ***scope*** and the ***capabilities*** required by the ***application*** scenario of interest. To achieve this objective, one needs to carefully consider the application scenarios, making sure these scenarios align with real-world demands (Moriarty, 2011). Also, it is necessary to assess whether other benchmarks already exist that address similar tasks, and to identify any shortcomings they may possess (McIntosh et al., 2025; Malode, 2024). Furthermore, this new benchmark should be designed to evaluate specific LLMs' capabilities; the crafted tasks are expected to reflect these capabilities (Hodak et al., 2023).

Phase 2. Construction. Benchmark Construction phase moves from design to execution. Typically, data is ***collected*** from public coding websites such as GitHub, LeetCode, and StackOverflow. This is followed by preprocessing, which includes filtering, cleaning (e.g., deduplication, denoising), and ***curation*** (e.g., aligning tests with corresponding code). The phase usually ends with a ***validation*** process, which can be manual or automated (McIntosh et al., 2025).

Phase 3. Evaluation. Once the benchmark is available, the next step is to apply it to LLMs, validating if it can effectively measure the intended LLM capabilities. Essential considering factors include ***selecting a representative array of LLMs***, configuring ***settings*** like prompts and hyperparameters for consistency, choosing appropriate experimental ***environments*** to meet LLM requirements, and implementing thorough ***logging*** to ensure dependable and reproducible results.

Phase 4. Analysis. After evaluation, experimental results are analyzed, drawing conclusions on LLMs' capabilities. This phase involves comparing each LLM's ***performance*** to identify standout or underperforming models. Then, proper visual aids such as bar charts and tables can be used to ***display*** the experimental results, presenting clearer observation and deeper ***inspiration***, such as the correlations between models, the correlations with re-

lated benchmarks, or performance in upper-/downstream tasks (Hendee and Wells, 1997). Indeed, a thorough analysis helps pinpoint areas for improvement and guides future LLM enhancements.

Phase 5. Release. The final phase is to make the benchmark open-accessible. This phase involves meticulously preparing all ***materials*** associated with the benchmark, ensuring they are ready for ***open access*** to foster widespread adoption and collaboration. Clear, comprehensive ***documentation*** is provided to guide users on effectively utilizing the benchmark. Additionally, all logged ***experiment details*** are made available, enhancing the reproducibility and transparency of the benchmark.

3.2 Study Design

Our study consists of four steps (Figure 2). All steps are explained as follows.

Step 1. Guideline Construction. To begin with, we ***sketched the initial guidelines*** for each phase in the benchmark development lifecycle (Section 3.1, Figure 1) by reviewing existing literature (Suppes et al., 1962; Zheng et al., 2023b; Schäfer et al., 2024; Reuel et al., 2024) and brainstorming. After that, we ***refined the guidelines*** through a series of interviews with various stakeholders, including model developers and benchmark builders, allowing for the addition, deletion, or modification of criteria based on expert feedback and practical insights. In order to allow more flexibility and increase the practicality of the guideline, we prioritized them with ★★★ (Highly important), ★★ (Important), and ★ (Optional). By doing so, the highly-recommended criteria are essential to comply with when setting up a new benchmark, while other criteria allow compromise, making the guideline simpler for benchmark developers to follow.

Step 2. Literature Profiling. This step begins by ***collecting related benchmarks*** according to their publication time, venue, and coding tasks, and then employing techniques like ***snowballing*** to ensure a comprehensive collection. This step leads to 274 code-related benchmarks for study. The detailed statistics can be found in the Appendix E. This step is followed by ***profiling*** each selected benchmark through a thorough review of *corresponding papers* and examination of *the released artifacts* or homepages associated with these benchmarks. The phase is completed by ***reporting statistics*** that highlight overall trends, pros, and cons identified during the profiling, providing a structured overview of existing benchmarks.

Step 3. Focused Case Study. After obtaining an overall impression of existing benchmarks, we ***selected 30 (= 5 * 6) representative benchmarks*** from top-5 tasks, with top-5 highly-cited benchmarks plus the latest 1 benchmark (Appendix D). Each selected benchmark is then ***analyzed against*** HOW2BENCH, examining how well they meet the established criteria, studying their overall statistics, and identifying both exemplary and poor cases. Insights and references from existing literature are also incorporated to enrich the analysis, providing a deeper understanding of the benchmarks' performance and areas for improvement.

Step 4. Human Study. The final step is a human study that evaluates the importance and practicality of HOW2BENCH. This involves ***designing a questionnaire*** by first initiating and iterating to gather diverse, logical insights, which is then ***distributed*** to a targeted audience. After collecting and filtering responses for quality, the data is ***analyzed*** to derive insights. See Appendix C for details.

4 Guideline – "HOW2BENCH"

The completed guideline HOW2BENCH with 55 criteria can be found in the Appendix. Each guideline contains: an actionable check criterion with necessary explanations; a priority indicator, divided into three levels in total, marking the importance of the checklist; and a checkbox for convenience.

4.1 Guideline for Benchmark Design

Explanation – For benchmark design, we listed four essential criteria, as shown in Figure 3. In particular, the guideline starts by recommending that benchmarks should initially assess if they are addressing a ***significant gap*** in existing research, ensuring the relevance and necessity of the benchmark. The ***scope*** of the benchmark is expected to be well-defined, clarifying the ***capabilities*** or characteristics being tested, how these relate to practical scenarios such as programming assistance or automated testing, and the relevance of these capabilities in real-world ***applications***.

■ ***Key Statistics*** – According to our statistics among 270+ benchmarks, ***apparent research bias*** can be observed in terms of coding tasks, programming languages, and code granularities (Appendix B.1). For example, 36.13% (99/274) are code generation benchmarks, followed by program repair, with 9.85% (27/274). Also, more than 30 programming languages and 20 natural languages

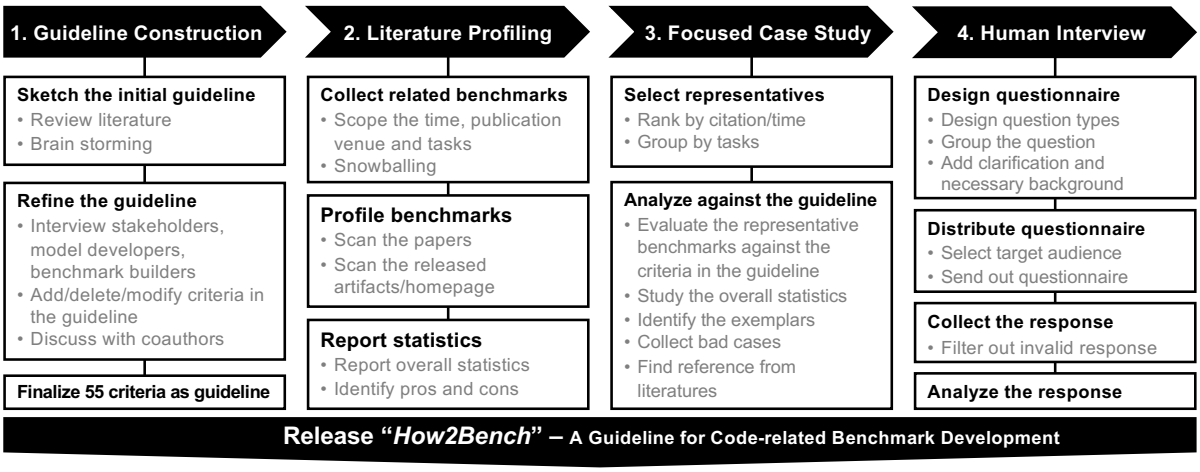

Figure 2: Workflow of study process

are assessed by only one benchmark.

Also, during the focused case study (listed in Appendix D), we identified that 10% benchmarks have not explicitly specified the capabilities (e.g., intention understanding, program synthesis) to be evaluated, and ***30% have not specified application scenarios*** the benchmark targets.

Besides, we also identified a case in MBPP (Austin et al., 2021) where a case fell out of the target evaluation capabilities (Appendix B.2). Indeed, clearly defining the application scenarios/scopes/capabilities could help benchmark constructors establish precise goals for the design and development of the benchmark, ensuring accuracy in the evaluation.

Lastly, Figure 12 shows that 58% (158/274) code-related benchmarks involve Python, followed by 39% (107/274) involving Java. Yet, 31 programming languages are only covered by one benchmark, and less than five benchmarks cover other 19 programming languages. This observation consolidates the observation from previous works (Cao et al., 2024a; Hou et al., 2023) on a larger scale.

▲ **Severity –** Current benchmarks exhibit ***an apparent imbalance*** in coding tasks and programming languages dominated by code generation and Python, leaving research blank to be filled. Also, even highly cited benchmarks may have samples that do not fall into the examined capabilities.

4.2 Guideline for Construction

☞ ***Explanation –*** Figure 5 shows 19 criteria for benchmark construction. Essentially, for **data source**, the key considerations include verifying the traceability and quality of the data source, addressing potential *data contamination* (Sainz et al., 2023), and ensuring that the ***data sampling processes*** are scientifically robust and rigorous. Also, for **data representativeness**, it also guides through specific checks to ensure the benchmark's scope is strictly adhered to, such as making sure every data point falls within the targeted scope and that the data can cover all studied capabilities, domain knowledge, and application scenarios.

For data preprocessing and cleaning, it also stresses handling code-specific aspects, such as compilability and execution, along with cleaning and ***manually reviewing*** data for quality assurance. Output validation methods and evaluation metrics must be carefully designed and reviewed to ensure they effectively measure the benchmark's goals. Lastly, it suggests considering additional evaluation perspectives, such as safety (Wei et al., 2024; Yuan et al., 2024a) checks, ensuring the code does not contain sensitive information.

■ ***Key Statistics –*** According to our statistics (Appendix B.3), the 270+ benchmarks exhibit ***numerous irregularities*** in their implementation, which could significantly threaten the reliability of the benchmarks. Surprisingly, **62% of benchmarks did not deduplicate** or did not mention. **Near 80% benchmarks did not consider or handle data contamination threats**. **About 70% of the benchmarks did not go through any quality assurance checks** such as manual checks and code execution. In particular, we summarized the ***commonly-used data quality assurance metrics*** and their frequency: manual check (22.6%), code execution

(2.2%), LLM check (1.5%), others (*e.g.* the number of stars or heuristic rules, 5.8%).

Also, since we focus on code-related benchmarks, which usually accompany test cases, ***test coverage*** also needs to be considered. As pointed out by a prior study (Liu et al., 2023a), inadequate test coverage can lead to inflated evaluation results. However, we observed that only 8.7% of benchmarks have considered test coverage when using test cases as oracles (Appendix B.3). It severely affects the reliability of findings on these benchmarks, potentially misguiding future research and applications based on these flawed assessments.

▲ **Severity –** Most benchmarks display ***severe loopholes*** in data preparation and curation, *i.e.*, only 31% of benchmarks went through a quality assurance check, and only 17% of them considered and handled data contamination threats.

4.3 Guideline for Evaluation

☞ ***Explanation –*** Guidelines for benchmark evaluation focus on the rigorousness and reliability of the evaluation. HOW2BENCH provides 12 criteria for benchmark evaluation, as shown in Figure 4. It mainly focuses on the comprehensive evaluation processes for benchmarks involving LLMs. For evaluation design, it stresses the importance of assessing a sufficient and ***representative range of LLMs*** to ensure the benchmark's applicability across various model families and configurations, both open and closed-source. Figure 29 and Figure 30 show the distribution of numbers of LLMs studied and the most exercised LLMs.

Also, ***prompting*** has a direct impact on the quality of the LLMs' output results (Wei et al., 2022; He et al., 2024a; Jin et al., 2024; Ye et al., 2023). As pointed out by a recent study, up to 40% performance gap could be observed in code translation when prompts vary (He et al., 2024b).

Additionally, ***the experiment environment*** is essential for reproducibility and transparency. Indeed, the hardware, software, and platform environments used during experiments might influence the outcomes (Ghosh, 2024). Furthermore, because of the nondeterministic nature of LLMs, experiments should be repeated, and randomization strategies should be used to mitigate the effects of randomness and parameter configuration biases. Lastly, ***meticulously documented logs*** of the experimental process is advised to facilitate transparency and reproducibility, detailing everything from parameter settings to the specific LLM pipelines such as vLLM (Kwon et al., 2023) used.

■ ***Key Statistics –*** Among the 274 benchmarks, 183 of them are evaluated over LLMs. According to our statistics (Figure 29), over 34% of the benchmarks were evaluated on fewer than 3 LLMs, with **11.48% benchmarks only evaluated on one LLM.** Such evaluation results can hardly be generalized to other LLMs. Furthermore, ***more than half of the benchmarks studied fewer than 6 LLMs*** (51% = (21 +22 + 20 + 4 + 12+15)/183).

For reference, we listed the top 10 most studied LLM families in Figure 30. Among them, the GPT series from OpenAI is the most extensively studied, accounting for 63% (116/183). Under the constraints of time and available resources, it is beneficial to evaluate more representative LLMs.

The prompt quality also significantly impacts the LLM evaluation (He et al., 2024b). According to a recent study, up to 40% of performance variation could be observed in the code translation task (He et al., 2024b). So, carefully designing a prompt needs consideration. However, ***73.3%*** representative benchmarks (Appendix D) do not validate whether the prompts they used are well-designed (Appendix B.4). Similarly, though 94.9% benchmarks were evaluated in a zero-shot manner, only 21.2% benchmarks were evaluated under few-shot, 8.8% under Chain-of-Thought and 2.6% under RAG (Appendix B.4). However, as shown in Figure 34, 73.3% representative benchmarks (Appendix D) do not validate whether the prompt they used is well-designed.

Regarding the evaluation process, our statistics exposed that **only 35.4% of benchmark evaluations have been repeated** (Appendix B.4). Also, regarding the transparency and matriculated documents, the observation is not optimistic – **Only 3.6% benchmarks provided their experiment environment**. **More than 50% of benchmarks did not provide reproducible instructions** such as prompts, examples for few-shot learning, or content for retrieval (Figure 39). **Less than half (42.7%) provide hyperparameters** such as temperature for reproduction.

▲ **Severity –** Over 60% of evaluations have not been repeated to eliminate the impact of randomness. **Less than 3.6% provided the complete and necessary information for reproducibility** such as prompts.

4.4 Guideline for Evaluation Analysis

☞ ***Explanation –*** The analysis of the experiment results is expected to be objective and comprehensive, hopefully providing insights or actionable advice. So, we listed 10 criteria for the evaluation analysis phase, as shown in Figure 6. Regarding **the perspectives of analysis**, inspired by classic measurement theory (Suppes et al., 1962), we suggest four essential perspectives, including ***difficulty*** (whether a benchmark is appropriately challenging for LLMs), ***stability*** (whether the results are consistent through repeated trials), ***differentiability*** (whether benchmarks can differentiate the strengths and weaknesses of various LLMs), and ***inspiration*** (e.g., the correlations between the upper-/downstream coding tasks and LLM scores).

Moreover, effective ***presentation of results*** using clear visual and textual descriptions could ensure the findings are understandable and actionable. The phase concludes with the suggestion to interpret and explain the results comprehensively, providing a basis for future enhancements.

■ ***Key Statistics –*** Because experimental analysis is relatively subjective and cannot be obtained through mechanical scanning, we focus on 30 representative focus benchmarks (Appendix D), covering the highest cited and latest benchmarks in top five tasks. Figure 37 shows an example from CruxEval (Gu et al., 2024) where the experimental scores can hardly be read from the figures.

Also, ***explaining experiment results*** is crucial for other practitioners to understand what the outcomes mean in the context of the research questions. According to our statistics (Appendix B.5), 70% benchmarks have detailed explanations and analyses of their evaluation results, while still ***30% have not***. Indeed, an explanation contributes to the body of knowledge by making it possible to understand and compare results with previous studies, promoting transparency within the community.

▲ **Severity –** The analysis of experimental data and the clarity of data presentation may receive less attention and worth consideration. Even in papers cited 1k+ times like MBPP (Austin et al., 2021), there are instances of***unclear evaluation analysis and display***.

4.5 Guideline for Benchmark Release

☞ ***Explanation –*** Finally, releasing a benchmark for open access also needs careful consideration. We offered 10 suggestions for this step, as shown in Figure 7, to highlight essential steps for public release preparation, emphasizing accessibility and ethical compliance. This includes setting an appropriate **license** to clarify usage rights, conducting a thorough review to **eliminate sensitive or harmful content** such as the API keys to access LLMs, the personal emails or toxic code comments (Miller et al., 2022) unless they are a part of the benchmark, and ensuring **transparency and reproducibility** by making all related materials openly available. **Detailed prompts** and clear descriptions of the experimental setup are advised to facilitate replication. Additionally, providing user manuals and evaluation interfaces is crucial for effective user engagement with the benchmark, enhancing its reliability and value for the research community.

■ ***Key Statistics –*** The final step involves the release of the benchmark. The fundamental requirement for releasing a benchmark is that it must be open-sourced. However, surprisingly, we observed that 5.1% of the benchmarks are only partially open-sourced (e.g., missing some subjects or tests), and ***5.8% are not open-sourced at all*** (e.g., links/web pages are no longer active). ***19.3% have not properly set up the license***. Furthermore, prompts, which are necessary for reproducibility, are not disclosed in 52.6% of the benchmarks (Figure 39). Not to mention the lack of public information on experimental settings (Figure 32 and Figure 31) and experimental parameters (Figure 43). What is worse, 19.3% benchmarks do not setup licenses (Figure 44). The absence of licensing may lead to severe legal and ethical issues, potentially resulting in unauthorized use and distribution of proprietary technologies. Additionally, only 16.7% of the benchmarks make their logged experimental results publicly available (Appendix B.6).

▲ **Severity –** The release of existing benchmarks exhibits several issues. For example, over 10% of the benchmarks are either not open to public access or are only partially open-sourced. Only 47.4% of benchmarks are released with replicable prompts.

5 Human Study

To delve deeper into the integration of knowledge and action, we ***surveyed 49 global researchers*** in AI (42.6%) and SE (57.14%), as shown in Figure 50. Each participant had published at least one

research paper, and about ***half had constructed code-related benchmarks***. See Appendix C.

First, **all participants agreed** that having a checklist for benchmark construction would contribute to the quality of the benchmark. 47/55 criteria in HOW2BENCH are deemed important by more 80% participants. Additionally, among the 21 participants who have constructed code-related benchmarks, ***53 out of 55 criteria were deemed important by all benchmark developers***; only two criteria (criteria 3 and 4 in Section 4) were considered unimportant by a few individuals (3 and 2 participants, respectively). Additionally, we received two valuable suggestions that draw importance to recording ***the time/monetary costs*** of constructing the benchmark and conducting the experiments.

However, we also identified some ***notable gaps in awareness***. First, regarding the ***data preparation***, more than 15% of participants were not aware that the selection of data should consider the target scope of the evaluation set (i.e., the data must be representative), and 16% of participants were ***unaware of the need for data denoising***. This oversight can significantly affect the validity and generalizability of experimental results, underscoring the importance of a comprehensive understanding of data handling for reliable research outcomes. Second, regarding ***evaluation replicability and reliability***. Over 40% of participants believe that recording and publicizing the hardware and software environments, software versions, and libraries used in experiments is not important, with more than 20% still considering it unimportant despite already done so. This reveals ***a significant lack of awareness*** about the impact that experimental environments can have on ***the reliability, reproducibility, and stability of evaluation results***. In fact, various studies have demonstrated that different experimental environments, parameters, and prompts can lead to substantial variations in outcomes (Xiao et al., 2024; Wang et al., 2019, 2023a).

6 Discussion

6.1 Trade-offs Between Benchmark Rigor and Development Efficiency

Despite all the above arguments, we admit that constructing rigorous benchmarks often entails a significant investment of time and human effort, which can lead to reduced efficiency in the development and evaluation process. Indeed, ensuring data quality, implementing thorough validation procedures, and designing comprehensive evaluation protocols requires non-trivial effort and may slow down the pace of research. It also echoed our human study. Through our human study, we found that researchers are often aware of the importance of several criteria (*e.g.*, data denoise, and repeating the experiments), but did not implement them due to time constraints or other limitations.

However, this trade-off between rigor and efficiency is necessary to guarantee the reliability, reproducibility, and scientific value of benchmark results. While faster, less rigorous benchmarks might accelerate short-term experimentation, they risk producing misleading or non-generalizable findings that ultimately hinder long-term progress. Therefore, each checklist in HOW2BENCH is labeled with a **priority**: ★★★ (Highly important), ★★ (Important), and ★ (Optional). We hope the design of this indicator may better help benchmark developers balance efficiency with rigor.

6.2 Awareness and Action

It is also worth recapping the notable gaps in awareness of the importance of data preparation and reproducibility (Section 5). For example, 16% of participants were unaware of the need for data denoising; 40% of participants believe that recording and publicizing the hardware and software environments, software versions, and libraries used in experiments is not important.

However, without being aware of and addressing the quality of LLM benchmarks, the reliability and reproducibility of benchmark results remain compromised. This observation should be a call to action for the research community to strengthen education and awareness around best practices in benchmark development.

7 Conclusion

This paper proposes a rigorous guideline consisting of 55 checklists covering the benchmark development lifecycle. After investigating over 270 code-related benchmarks, we exposed their merits and limitations and provided suggestions for improving them. Finally, our human study reveals the neglect of details that may affect the benchmark's reliability. In the long run, HOW2BENCH helps to improve the overall quality of benchmarks in the community due to the propagation among benchmarks.

Limitations

This paper has two primary limitations that offer avenues for future research. First, ***the collection of code-related benchmarks may be incomplete***. To minimize this limitation, we covered papers published over the last decade, and conducted multiple rounds of snowballing. Ultimately, we collected 274 benchmarks, which is comparable to the number included in recent surveys (Hou et al., 2023; Schäfer et al., 2024) in the field. Second, ***the study involved substantial manual analysis***, which could lead to oversight and discrepancies in the statistical results. To mitigate this issue, we ensured that each benchmark was double-checked by at least two authors and underwent multiple rounds of iteration. Third, ***the guidelines may not cover all the details***. Constructing a code-related benchmark involves numerous details, and some criteria are task-specific. To overcome this limitation, we iteratively refined the guidelines, interviewed practitioners, and tried to cover the entire benchmark development process as thoroughly as possible. Last, ***the human study participants may exhibit subjectivity***. To address this limitation, we endeavored to include a broad range of practitioners and seasoned researchers with experience in both AI and SE, aiming for the least biased results possible.

References

Yash Agarwal, Devansh Batra, and Ganesh Bagler. 2020. Building hierarchically disentangled language models for text generation with named entities. In *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 26–38. International Committee on Computational Linguistics.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 5435–5445. Association for Computational Linguistics.

Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-guided decoding of code lms with static analysis of repository context. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A parallel corpus for java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 2268–2281. Association for Computational Linguistics.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 27865–27876.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2022. Multi-lingual evaluation of code generation models.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2024. Studenteval: A benchmark of student-written prompts for large language models of code. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 8452–8474. Association for Computational Linguistics.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE):675–698.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017, Volume 2: Short Papers*, pages 314–319. Asian Federation of Natural Language Processing.

Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 780–791. PMLR.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long code arena: a set of benchmarks for long-context code models. *CoRR*, abs/2406.11612.

Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024a. Can AI beat undergraduates in entry-level java assignments? benchmarking large language models on javabench. *CoRR*, abs/2406.12902.

Jialun Cao, Wuqi Zhang, and Shing-Chi Cheung. 2024b. Concerned with data contamination? assessing countermeasures in code language model. *arXiv preprint arXiv:2403.16898*.

Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Yuchen Mao, Wenjing Hu, Tianbao Xie, Hongshen Xu, Danyang Zhang, Sida Wang, Ruoxi Sun, Pengcheng Yin, Caiming Xiong, Ansong Ni, Qian Liu, Victor Zhong, Lu Chen, Kai Yu, and Tao Yu. 2024c. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? *CoRR*, abs/2407.10956.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Software Eng.*, 48(9):3280–3296.

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45.

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

Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. 2021b. Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 2169–2181. Association for Computational Linguistics.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*, pages 654–668. ACM.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. 2024. MHPP: exploring the capabilities and limitations of language models beyond basic code generation. *CoRR*, abs/2405.11430.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 1337–1350. Association for Computational Linguistics.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair,

David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2024a. Vulnerability detection with code language models: How far are we? *CoRR*, abs/2403.18624.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024b. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 3433–3445. ELRA and ICCL.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023a. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023b. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *CoRR*, abs/2308.01861.

Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From commit message generation to history-aware commit message completion. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 723–735. IEEE.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir R. Radev. 2018. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 351–360. Association for Computational Linguistics.

Karl Fogel. 2005. *Producing open source software: How to run a successful free software project*. " O'Reilly Media, Inc.".

Graham Francis and Jacky Holloway. 2007. What have we learned? themes from the literature on best-practice benchmarking. *International Journal of Management Reviews*, 9(3):171–189.

Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, Yifan Liu, Jingkuan Wang, Siyuan Qi, Kangning Zhang, Weinan Zhang, and Yong Yu. 2023. Codeapex: A bilingual programming evaluation benchmark for large language models. *CoRR*, abs/2309.01940.

Yanjun Fu, Ethan Baker, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *CoRR*, abs/2405.00218.

Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. 2022. Measuring and improving compositional generalization in text-to-sql via component alignment. In *Findings of the Association for Computational Linguistics: NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, pages 831–843. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-sql models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 2505–2515. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. Exploring underexplored limitations of cross-domain text-to-sql generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8926–8931. Association for Computational Linguistics.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023a. PAL: program-aided language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023b. How far have we gone in vulnerability detection using large language models. *CoRR*, abs/2311.12420.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. Deepperf: A deep learning-based approach for improving software performance. *CoRR*, abs/2206.13619.

Bijit Ghosh. 2024. Changing your gpu changes your llm behavior.

Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *CoRR*, abs/2308.08493.

Jing Gong, Yanghui Wu, Linxi Liang, Zibin Zheng, and Yanlin Wang. 2024. Cosqa+: Enhancing code search dataset with matching code. *CoRR*, abs/2406.11589.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 933–944. ACM.

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. *CoRR*, abs/2404.03543.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1345–1351. AAAI Press.

Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. 2024. On the impacts of contexts on repository-level code generation. *Preprint*, arXiv:2406.11927.

Patrick Haller, Jonas Golde, and Alan Akbik. 2024. PECC: problem extraction and coding challenges. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 12690–12699. ELRA and ICCL.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and Wei He. 2022. Aixbench: A code generation benchmark dataset. *CoRR*, abs/2206.13179.

Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. Fixeval: Execution-based evaluation of program fixes for programming problems. In *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*, pages 11–18. IEEE.

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. Codesc: A large code-description parallel dataset. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 210–218. Association for Computational Linguistics.

Moshe Hazoom, Vibhor Malik, and Ben Bogin. 2021. Text-to-sql in the wild: A naturally-occurring dataset based on stack exchange data. *CoRR*, abs/2106.05006.

Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. 2024a. Does prompt formatting have any impact on llm performance? *Preprint*, arXiv:2411.10541.

Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. 2024b. Does prompt formatting have any impact on llm performance? *arXiv preprint arXiv:2411.10541*.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

William R Hendee and Peter NT Wells. 1997. *The perception of visual information*. Springer Science & Business Media.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193.

Miro Hodak, David Ellison, Chris Van Buren, Xiaotong Jiang, and Ajay Dholakia. 2023. Benchmarking large language models: opportunities and challenges. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 77–89. Springer.

Jacky Holloway, Matthew Hinton, Graham Francis, and David Mayle. 1999. *Identifying best practice in benchmarking*. CIMA Publishing.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *CoRR*, abs/2308.10620.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension,*

ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, pages 200–210. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Refactoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398.

Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin B. Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *CoRR*, abs/2211.09374.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and Weizhu Chen. 2024. Competition-level problems are effective LLM evaluators. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 13526–13544. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. *Preprint*, arXiv:2303.05398.

Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 955–963, New York, NY, USA. Association for Computing Machinery.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1430–1442. IEEE.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1529–1541. IEEE.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1646–1656. ACM.

Mingyu Jin, Qinkai Yu, Dong Shu, Haiyan Zhao, Wenyue Hua, Yanda Meng, Yongfeng Zhang, and Mengnan Du. 2024. The impact of reasoning step length on large language models. *arXiv preprint arXiv:2401.04925*.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM.

Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Do Long, Weishi Wang, Md. Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 6766–6805. Association for Computational Linguistics.

Rahul Kumar, Amar Raja Dibbu, Shrutendra Harsola, Vignesh Subrahmaniam, and Ashutosh Modi. 2024. Booksql: A large scale text-to-sql dataset for accounting domain. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 497–516. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Beck LaBash, August Rosedale, Alex Reents, Lucas Negritto, and Colin Wiel. 2024. RES-Q: evaluating code-editing large language model systems at the repository scale. *CoRR*, abs/2406.16801.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 795–806. IEEE / ACM.

Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. CS1QA: A dataset for assisting code-based question answering in an introductory programming course. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, pages 2026–2040. Association for Computational Linguistics.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. Kaggledbqa: Realistic evaluation of text-to-sql parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 2261–2273. Association for Computational Linguistics.

Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. 2023. EHRSQL: A practical text-to-sql benchmark for electronic health records. *CoRR*, abs/2301.07695.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024a. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint*. ArXiv:2405.19856 [cs].

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023a. Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Kaixin Li, Qisheng Hu, James Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024b. Instructcoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024 - Student Research Workshop, Bangkok, Thailand, August 11-16, 2024*, pages 50–70. Association for Computational Linguistics.

Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, and Jing Ma. 2024c. Mmcode: Evaluating multi-modal code large language models with visually rich programming problems. *CoRR*, abs/2404.09486.

Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. 2024d. Infibench: Evaluating

the question-answering capabilities of code large language models. *Preprint*, arXiv:2404.07940.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023b. TACO: topics in algorithmic code generation dataset. *CoRR*, abs/2312.14852.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zehan Li, Jianfei Zhang, Chuantao Yin, Yuanxin Ouyang, and Wenge Rong. 2024e. Procqa: A large-scale community-based programming question answering dataset for code search. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 13057–13067. ELRA and ICCL.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018a. Sysevr: A framework for using deep learning to detect software vulnerabilities. *CoRR*, abs/1807.06756.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018b. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A 3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware. *IEEE Transactions on Software Engineering*.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 55–56. ACM.

Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2019. Deep learning-based vulnerable function detection: A benchmark. In *Information and Communications Security - 21st International Conference, ICICS 2019, Beijing, China, December 15-17, 2019, Revised Selected Papers*, volume 11999 of *Lecture Notes in Computer Science*, pages 219–232. Springer.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier Y. de Vel, Paul Montague, and Yang Xiang. 2021. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Trans. Dependable Secur. Comput.*, 18(5):2469–2485.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier Y. de Vel, and Paul Montague. 2018. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Informatics*, 14(7):3289–3297.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5):88:1–88:21.

Chenxiao Liu and Xiaojun Wan. 2021. Codeqa: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2618–2632. Association for Computational Linguistics.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024a. Repoqa: Evaluating long context code understanding. *CoRR*, abs/2406.06025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2022. ATOM: commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Trans. Software Eng.*, 48(5):1800–1817.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024b. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. 2024c. Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models. *CoRR*, abs/2406.07595.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zhengliang Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023c. Ml-bench: Large language models leverage open-source libraries for machine learning tasks. *CoRR*, abs/2311.09835.

Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 373–384. ACM.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 304–315. IEEE / ACM.

Vishal Manjunatha Malode. 2024. *Benchmarking public large language model*. Ph.D. thesis, Technische Hochschule Ingolstadt.

Timothy R McIntosh, Teo Susnjak, Nalin Arachchilage, Tong Liu, Dan Xu, Paul Watters, and Malka N Halgamuge. 2025. Inadequacies of large language model benchmarks in the era of generative artificial intelligence. *IEEE Transactions on Artificial Intelligence*.

Courtney Miller, Sophie Cohen, Daniel Klug, Bogdan Vasilescu, and Christian KaUstner. 2022. "did you miss my comment or what?": understanding toxicity in open source discussions. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 710–722, New York, NY, USA. Association for Computing Machinery.

Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2241–2252. ACM.

John P Moriarty. 2011. A theory of benchmarking. *Benchmarking: An International Journal*, 18(4):588–611.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David A. Sontag. 2024. The realhumaneval: Evaluating large language models' abilities to support programmers. *CoRR*, abs/2404.02806.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 819–830. IEEE / ACM.

Daniel Nichols, Joshua Hoke Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2024, Pisa, Italy, June 3-7, 2024*, pages 281–294. ACM.

Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2111–2123. IEEE.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. Crossvul: a cross-language vulnerability dataset with commit

data. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1565–1569. ACM.

Wonseok Oh and Hakjoo Oh. 2022. Pyter: effective program repair for python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 922–934. ACM.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *CoRR*, abs/2305.15334.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Sceneval: A benchmark for scenario-based evaluation of code generation. In *IEEE International Conference on Artificial Intelligence Testing, AITest 2024, Shanghai, China, July 15-18, 2024*, pages 55–63. IEEE.

Nathaniel Ross Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks. *CoRR*, abs/2408.11053.

Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can openai's codex fix bugs?: An evaluation on quixbugs. In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*, pages 69–75. IEEE.

Julian Aron Prenner and Romain Robbes. 2023. Runbugrun - an executable dataset for automated program repair. *CoRR*, abs/2304.01102.

Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024a. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*.

Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024b. How efficient is llm-generated code? A rigorous & high-standard benchmark. *CoRR*, abs/2406.06647.

Anka Reuel, Amelia Hardy, Chandler Smith, Max Lamparth, Malcolm Hardy, and Mykel J. Kochenderfer. 2024. Betterbench: Assessing ai benchmarks, uncovering issues, and establishing best practices. *Preprint*, arXiv:2411.12990.

Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association.

Anna Rogers, Matt Gardner, and Isabelle Augenstein. 2023. Qa dataset explosion: A taxonomy of nlp resources for question answering and reading comprehension. *ACM Comput. Surv.*, 55(10).

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*, pages 757–762. IEEE.

Jaehee Ryu, Seonhee Cho, Gyubok Lee, and Edward Choi. 2024. Ehr-seqsql : A sequential text-to-sql dataset for interactively exploring electronic health records. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 16388–16407. Association for Computational Linguistics.

Oscar Sainz, Jon Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10776–10787, Singapore. Association for Computational Linguistics.

Irina Saparina and Mirella Lapata. 2024. AMBROSIA: A benchmark for parsing ambiguous questions into database queries. *CoRR*, abs/2406.19073.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Software Eng.*, 50(1):85–105.

Maximilian Schall, Tamara Czinczoll, and Gerard de Melo. 2024. Commitbench: A benchmark for

commit message generation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 728–739. IEEE.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Chufan Shi, Cheng Yang, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, Gongye Liu, Xiaomei Nie, Deng Cai, and Yujiu Yang. 2024a. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *CoRR*, abs/2406.09961.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024b. Can language models solve olympiad programming? *CoRR*, abs/2404.10952.

Tianze Shi, Chen Zhao, Jordan L. Boyd-Graber, Hal Daumé III, and Lillian Lee. 2020. On the potential of lexico-logical alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1849–1864. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023a. Repofusion: Training code models to understand your repository. *CoRR*, abs/2306.10998.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2code: How far are we from automating front-end engineering? *CoRR*, abs/2403.03163.

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. 2024. Nofuneval: Funny how code lms falter on requirements beyond functional correctness. *CoRR*, abs/2401.15963.

Patrick Suppes, Joseph L Zinnes, et al. 1962. *Basic measurement theory*.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 476–480. IEEE Computer Society.

Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark B Gerstein. 2024. Biocoder: a benchmark for bioinformatics code generation with large language models. *Bioinformatics*, 40(Supplement_1):i266–i276.

Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2022. A large-scale empirical study of commit message generation: models, datasets and evaluation. *Empir. Softw. Eng.*, 27(7):198.

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. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 4173–4198. Association for Computational Linguistics.

Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. METHODS2TEST: A dataset of focal methods mapped to test cases. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 299–303. ACM.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29.

Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024. Vhdl-eval: A framework for evaluating large language models in VHDL code generation. *CoRR*, abs/2406.04379.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 397–407. ACM.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024a. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Ping Wang, Tian Shi, and Chandan K. Reddy. 2020. Text-to-sql generation for question answering on electronic medical records. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 350–361. ACM / IW3C2.

Shuai Wang, Liang Ding, Li Shen, Yong Luo, Bo Du, and Dacheng Tao. 2024b. OOP: object-oriented programming evaluation benchmark for large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 13619–13639. Association for Computational Linguistics.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024c. TESTEVAL: benchmarking large language models for test case generation. *CoRR*, abs/2406.04531.

Yibo Wang, Ying Wang, Tingwei Zhang, Yue Yu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023a. Can machine learning pipelines be better configured? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 463–475, New York, NY, USA. Association for Computing Machinery.

Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking tpu, gpu, and cpu platforms for deep learning. *Preprint*, arXiv:1907.10701.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2023b. Mconala: A benchmark for code generation from multiple natural languages. In *Findings of the Association for Computational Linguistics: EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pages 265–273. Association for Computational Linguistics.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024d. Coderag-bench: Can retrieval augment code generation? *CoRR*, abs/2406.14497.

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1398–1409. ACM.

Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. 2024. Jailbroken: How does llm safety training fail? *Advances in Neural Information Processing Systems*, 36.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*.

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Typet5: Seq2seq type inference using static analysis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. 2024a. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *CoRR*, abs/2405.07990.

Minghao Wu, Weixuan Wang, Sinuo Liu, Huifeng Yin, Xintong Wang, Yu Zhao, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. 2025. The bitter lesson learned from 2,000+ multilingual benchmarks. *Preprint*, arXiv:2504.15521.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. 2024b. Versicode: Towards version-controllable code generation. *CoRR*, abs/2406.07411.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024a. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via LLM. *CoRR*, abs/2403.19114.

Yinghui Xia, Yuyan Chen, Tianyu Shi, Jun Wang, and Jinsong Yang. 2024b. Aicodereval: Improving AI domain code generation of large language models. *CoRR*, abs/2406.04712.

Jie Xiao, Qianyi Huang, Xu Chen, and Chen Tian. 2024. Large language model performance benchmarking on mobile platforms: A thorough evaluation. *Preprint*, arXiv:2410.03613.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2024. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. *CoRR*, abs/2408.13001.

Ankit Yadav, Himanshu Beniwal, and Mayank Singh. 2024a. Pythonsaga: Redefining the benchmark to evaluate code generating llms. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 17113–17126.

Ankit Yadav, Himanshu Beniwal, and Mayank Singh. 2024b. Pythonsaga: Redefining the benchmark to evaluate code generating llms. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 17113–17126. Association for Computational Linguistics.

Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 344–354. IEEE.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 5067–5089. Association for Computational Linguistics.

Hongyu Yang, Liyang He, Min Hou, Shuanghong Shen, Rui Li, Jiahui Hou, Jianhui Ma, and Junda Zhao. 2024a. Aligning llms through multi-perspective user preference ranking-based feedback for programming question answering. *CoRR*, abs/2406.00037.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. 2024b. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 11789–11804. Association for Computational Linguistics.

Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1693–1703. ACM.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *Preprint*, arXiv:2303.10420.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 126–173. Association for Computational Linguistics.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander R. Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter S. Lasecki, and Dragomir R. Radev. 2019a. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 1962–1979. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir R. Radev. 2019b. Sparc: Cross-domain semantic parsing in context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4511–4523. Association for Computational Linguistics.

Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. 2024. Fight fire with fire: How much can we trust chatgpt on source code-related tasks? *arXiv preprint arXiv:2405.12641*.

Xiaojing Yu, Tianlong Chen, Zhengjie Yu, Huiyu Li, Yang Yang, Xiaoqian Jiang, and Anxiao Jiang. 2020. Dataset and enhanced model for eligibility criteria-to-sql semantic parsing. In *Proceedings of The 12th Language Resources and Evaluation Conference, LREC 2020, Marseille, France, May 11-16, 2020*, pages 5829–5837. European Language Resources Association.

Youliang Yuan, Wenxiang Jiao, Wenxuan Wang, Jen tse Huang, Pinjia He, Shuming Shi, and Zhaopeng Tu. 2024a. Gpt-4 is too smart to be safe: Stealthy chat with llms via cipher. *Preprint*, arXiv:2308.06463.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024b. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023a. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023b. No more manual tests? evaluating and improving chatgpt for unit test generation. *CoRR*, abs/2305.04207.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2024. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. *Preprint*, arXiv:2311.16502.

Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Qazim Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, Timothy Baldwin, Zhengzhong Liu, Eric P. Xing, Xiaodan Liang, and Zhiqiang Shen. 2024. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *CoRR*, abs/2406.20098.

Mohamed Zairi. 2010. *Benchmarking for best practice*. Routledge.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022a. When language model meets private library. *arXiv preprint arXiv:2210.17236*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022b. Cert: continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888*.

Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. 2024. Coderujb: An executable and unified java benchmark for practical programming scenarios. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 124–136. ACM.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 13643–13658. Association for Computational Linguistics.

Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023a. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *CoRR*, abs/2310.08879.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024b. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts. *CoRR*, abs/2405.04520.

Yusen Zhang, Jun Wang, Zhiguo Wang, and Rui Zhang. 2023b. Xsemplr: Cross-lingual semantic parsing in multiple natural languages and meaning representations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 15918–15947. Association for Computational Linguistics.

Ziyin Zhang, Lizhen Xu, Zhaokun Jiang, Hongkun Hao, and Rui Wang. 2024c. Multiple-choice questions are efficient and robust LLM evaluators. *CoRR*, abs/2405.11966.

Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2024. Towards more realistic evaluation of llm-based code generation: an experimental study and beyond. *CoRR*, abs/2406.06918.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023a. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

Yunhui Zheng, Saurabh Pujar, Burn L. Lewis, Luca Buratti, Edward A. Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 111–120. IEEE.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *CoRR*, abs/2206.08474.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. Domaineval: An auto-constructed benchmark for multi-domain code generation. *CoRR*, abs/2408.13204.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2020. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *CoRR*, abs/2001.02334.

A Detailed Guidance in HOW2BENCH

We present the detailed guidance in each phase in this section.

A.1 Guideline for Benchmark Design

	Phase 1. Benchmark Design	Priority	☑
1	**Consider whether the benchmark can fill the gap in related research.**	★★★	☐
2	**Consider what is the expected scope of the benchmark set** (e.g., what natural languages, programming languages, task granularity).	★★★	☐
3	**Consider the expected application scenario of this benchmark** (e.g., programming assistant, automated tester).	★★★	☐
4	**Consider the LLMs' capabilities** (e.g., understanding, reasoning, calculation) **and domain knowledge** (e.g., OOP, memory management, fault localization, process scheduling) **that the benchmark hopes to evaluate.**	★★★	☐

Figure 3: **Guideline for Benchmark Design**

A.2 Guideline for Benchmark Evaluation

	Phase 3. Benchmark Evaluation	Priority	☑
24	Consider whether **sufficient LLMs are evaluated.**	★	☐
25	**Consider whether representative LLMs** (e.g., covering latest/classical LLM families, small/large LLMs, and open-/closed-source LLMs) **are evaluated.**	★	☐
26	**Consider whether the prompt is of high quality** (e.g., the instruction and intent are clear).	★★★	☐
27	The prompts have been **validated by humans or LLMs** (e.g., evaluated or discussed by participants or preliminarily tried out on several LLMs).	★	☐
28	**Try different paraphrases of the prompt.**	★	☐
29	**Try different prompting strategies to observe the impact on the evaluation results** (e.g., in-context learning, chain-of-thought).	★★	☐
30	**Pay attention to the hardware environment** (such as GPU card, storage size, etc.) of the experiment.	★★★	☐
31	**Pay attention to the operating system and software environment** (e.g. operating system, version, etc.) **used for the experiment.**	★★★	☐
32	Pay attention to the off-the-shelf **platforms, frameworks, or libraries for LLM evaluation** (e.g., fast chat, vllm, huggingface) **that are used.**	★★	☐
33	**Repeat the experiment multiple times to reduce the impact of randomness on the evaluation.**	★	☐
34	**Consider various randomization strategies** (e.g., trying various temperature parameters) **to reduce the impact of parameter configuration on the evaluation.**	★★	☐
35	**Record the experimental process in detail** (e.g., parameter settings, running time, input/output pairs, etc.).	★★★	☐

Figure 4: Guideline for Benchmark Evaluation

A.3 Guideline for Benchmark Construction

	Phase 2. Benchmark Construction (1/2)	Priority	☑
5	**Consider whether the data source of the benchmark is traceable.**	★	☐
6	**Consider whether the data source of the benchmark is of high quality** (e.g., stars, downloads, last update times, number of forks).	★★	☐
7	**Consider whether the benchmark's data source is representative** (e.g., choose an open-source community or code hosting platform that matches the task, capability, and scope under study)	★	☐
8	**Consider data contamination issues during the benchmark collection** (e.g., considering the upload time of the source code or checking whether the data source is included in the training data of LLMs).	★	☐
9	**If data sampling is needed, consider whether the choice of sample size is scientific** (e.g., considering the confidence level/margin of error/sampling proportion, etc.).	★	☐
10	**If data sampling is needed, consider whether the sampling process is rigorous** (e.g., random sampling, stratified sampling, etc.).	★	☐
11	**Ensure each data point in the benchmark falls into the targeted scope** (e.g., checking each data point's evaluated capabilities or domain knowledge).	★	☐
12	**Consider whether the data in the benchmark can cover the studied capabilities/domain knowledge/application scenarios.**	★★★	☐
13	**Consider whether there is a standard answer for each sample in the benchmark** (such as reference code, etc.).	★★	☐
14	**For code, consider whether the code is compilable/executable.**	★	☐
15	**Consider the possibility of noise in the data and perform denoise.**	★★	☐
16	**Consider the possibility of duplication in the data and deduplicate them.**	★★	☐
17	**Clean the sensitive information** (such as data desensitization and anonymization) **unless the benchmark is deliberately designed so.**	★★	☐
18	**Manually review some or all of the data in the benchmark to ensure its quality.**	★★	☐
19	**Use LLMs to review some or all of the data in the benchmark to ensure its quality.**	★	☐
20	**Design appropriate output validation methods for the benchmark** (e.g., using exact matching or designing test cases).	★★★	☐
21	**Design appropriate evaluation metrics for the evaluation set** (e.g., precision, accuracy, pass@K, recall).	★★★	☐
22	**Consider the adequacy of the evaluation metrics** (e.g., is the code coverage high enough).	★★★	☐
23	**Consider if there are any other evaluation perspectives** (e.g., readability, efficiency, safety, security).	★	☐

Figure 5: Guideline for Benchmark Construction

A.4 Guideline for Benchmark Analysis

	Phase 4. Benchmark Analysis	Priority	☑
36	**Observe the difficulty of the benchmark, checking if the benchmark is too hard or too easy for LLMs** (i.e., most LLMs score too high/low).	★★	☐
37	**Consider whether the benchmark can distinguish the pros and cons of different LLMs.**	★	☐
38	**If the experiment is repeated several times, consider the stability of the benchmark** (i.e., whether the experimental results vary too much in the repeated experiments).	★	☐
39	**Analyze the correlation between the data and their score**. For example, if there is a correlation between the data (such as similar difficulty and knowledge required), then the scores should also be correlated.	★★	☐
40	Compare the performance of LLMs on this benchmark with their performance on **other related benchmarks**.	★	☐
41	**Consider presenting the experiment results in an appropriate way** (e.g., table, line graph, pie chart, etc.).	★★★	☐
42	**Consider presenting the experiment results clearly** (e.g., distinguishable colors/labels/shapes, etc.).	★★★	☐
43	**Explain the experiment results.**	★★★	☐
44	**Observe correlations via multiple perspectives from the experimental results** (e.g., performance is correlated with model size or amount of context).	★	☐

Figure 6: **Guideline for Evaluation Analysis**

A.5 Guideline for Benchmark Release

	Phase 5. Benchmark Release	Priority	☑
45	**The analysis of the evaluation results will be inspiring** (e.g., shed light on future direction, make actionable advice, etc.).	★	☐
46	**Set the appropriate license for the benchmark.**	★★★	☐
47	**Review the released benchmark or other artifacts to ensure they do NOT contain sensitive information** (e.g., API keys, usernames, passwords, etc.).	★★★	☐
48	**review the released benchmark or other artifacts to ensure they do NOT contain toxicity information** (e.g., abusive comments/identifiers).	★★★	☐
49	**Make sure the benchmark is open-accessible.**	★★★	☐
50	**Make sure the test cases or reference data are open and accessible.**	★★★	☐
51	**Provide prompts used in the experiment to ensure the experiments are reproducible.**	★★★	☐
52	**Disclose the experimental environment** (e.g., hardware, operating system, software version, framework platform) **to ensure the reproducibility of the experiment.**	★★★	☐
53	**Make the detailed experimental results public for verification.**	★★★	☐
54	**Ensure the quality of the user manual** such as README (e.g., it contains necessary benchmark introduction, executable scripts, etc.).	★★	☐
55	**Provide convenient evaluation interfaces for the released benchmark** (e.g., providing a command line interface, docker, etc.).	★★	☐

Figure 7: **Guideline for Benchmark Release**

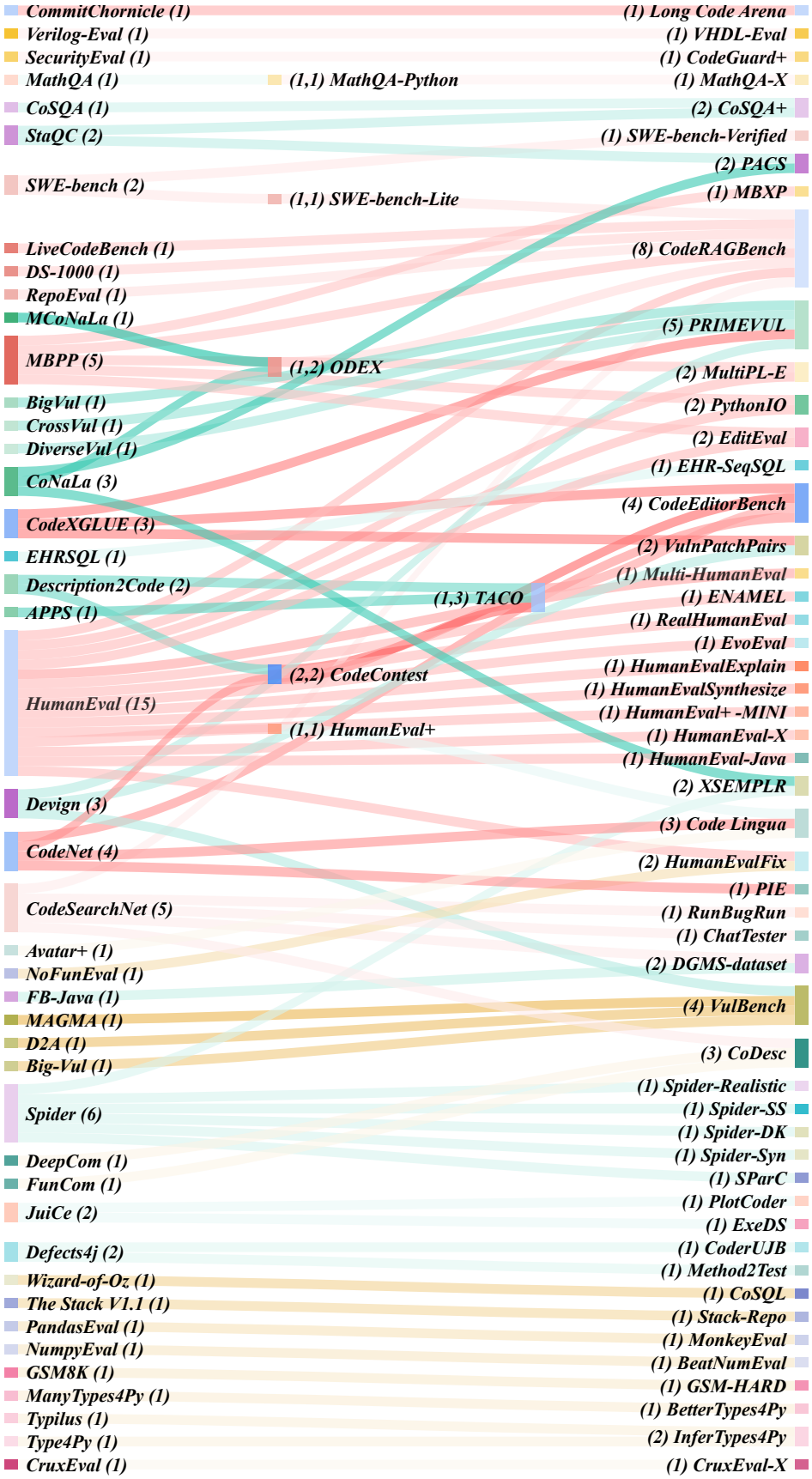

Figure 8: Relationships between Benchmarks

B Statistics of studied benchmarks

In this section, we conducted a comprehensive and detailed statistical analysis of the 274 benchmarks collected.

B.1 Profile of Studied Benchmarks

We first show the trend in the development of benchmarks from 2014 to 2024. As shown in Figure 9, the data shows a modest beginning, with only a handful of benchmarks created annually until 2017. From 2018 onwards, there is a noticeable uptrend in benchmark creation, culminating in a significant jump to 149 benchmarks in 2024. This sharp increase indicates a recent heightened interest and demand for comprehensive code-related benchmarks for LLMs, reflecting the evolving complexities and expanding requirements of automated software engineering.

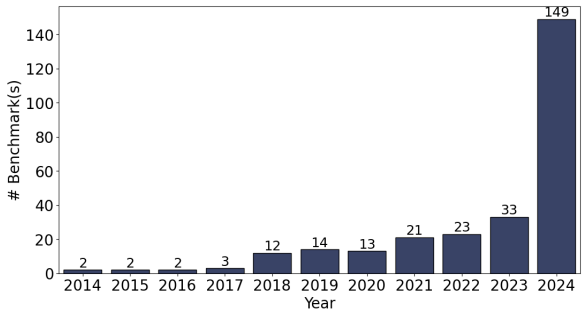

Figure 9: Benchmark Distribution over Years

Hierarchy of Benchmarks. Figure 8 visualize the inheritance relationships among benchmarks, indicating that the benchmarks on the ***left serve as sources*** for those on the right. It highlights that ***18% (50 out of 274) of benchmarks act as data sources***, continuously benefiting the construction of subsequent benchmarks.

Figure 8 reveals that ***HumanEval*** (Chen et al., 2021a), as the ***most significant source*** benchmark, benefits at least ***15 downstream benchmarks***, followed by MBPP (Austin et al., 2021) and CodeSearchNet (Husain et al., 2019). From the right side of the figure, some benchmarks, like VulBench (Gao et al., 2023b), incorporate methodologies or data from 4 previous benchmarks, and codeRagBench (Wang et al., 2024d) integrates elements from 8 prior benchmarks.

This hierarchical structure among benchmarks also alerts us that the ***data quality of a benchmark not only affects its own credibility but can continue to impact others*** if it serves as a source. This underscores the importance of adhering to stringent guidelines during benchmark development and highlights the crucial role of ***establishing standards*** to ensure the integrity and utility of benchmark data across research and development efforts.

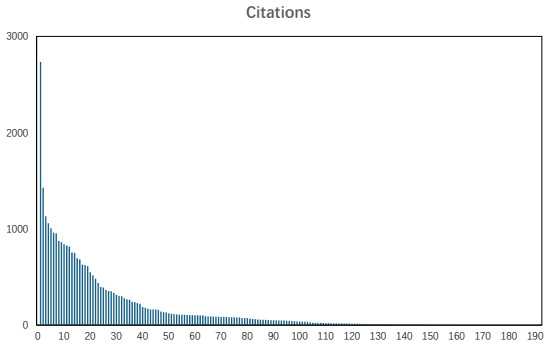

Figure 10: Citation Distribution of Benchmarks

Annual Trend. Regarding the ***coding tasks***, Figure 11 illustrates the distribution of various coding tasks across benchmarks. It is clear that the task of ***Code Generation*** is most prevalent, with 99 benchmarks focusing on this area, according to 36% (99/274) of studied benchmarks, indicating a significant interest in generating code automatically. Program Repair and Defeat Detection are well-represented, with 27 and 25 benchmarks, respectively, reflecting the importance of correcting code and detecting defects.

Citation distribution. We also visualized the citations of 274 code-related benchmarks. The citation statistics were collected on September 1st, 2024. From Figure 10, we can see a clear long-tail trend of the citations, from the highest 2735 (HumanEval (Chen et al., 2021a)) to the lowest 0.

Coding Task. Tasks like Code Summarization and Text2SQL are similarly significant, each with 25 and 22 benchmarks. These tasks focus on making code more understandable and converting natural language queries into SQL queries. Other tasks, such as Code Retrieval, Code Reasoning, and Code Translation, are represented with 18, 17, and 16 benchmarks, respectively. Lesser-represented benchmarks are Test Generation, Code Optimization, and Code Completion, each represented by 8 and 7 benchmarks, indicating the inadequacy of these tasks.

Programming Languages. Figure 12 shows the distribution of benchmarks across various programming languages. The overall trend indicates a strong preference for benchmarking ***Python***, which

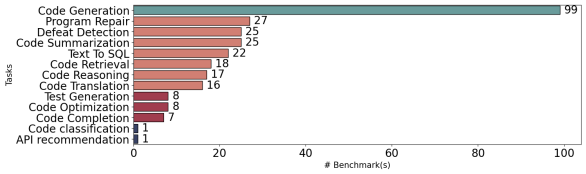

Figure 11: Benchmark Distribution over Tasks

leads with 158 benchmarks, followed by Java and C++, with 107 and 63, respectively. The graph also reveals a diverse range of languages being used. In total, 724 programming languages are studied by these 274 benchmarks. Though some programming languages, such as Kotlin, Swift, and Scala, are less frequently exercised, the benchmarks involving them are tailored to different application needs and technology environments. This distribution shows the existing benchmarks are dominated by three mainstream programming languages, leaving other programming languages less studied and benchmarked.

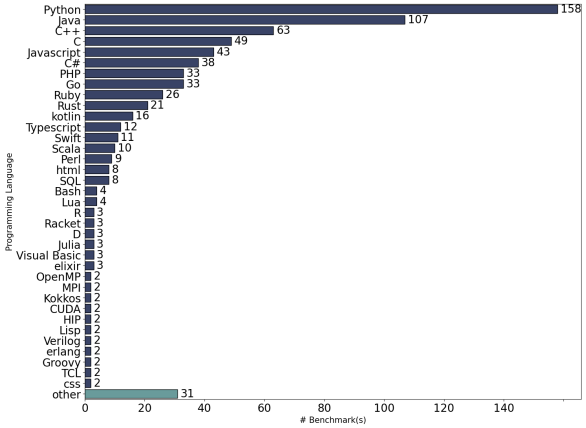

Figure 12: Benchmark Distribution over Programming Language

Natural Language. Figure 13 illustrates the distribution of benchmarks for different natural languages. The bar chart overwhelmingly shows that English is the dominant language, with 192 benchmarks highlighting its ubiquity in research and development. Other languages have significantly fewer benchmarks, with six for Chinese and only two each for Japanese, Russian, and Spanish. The category labeled "Other" includes 20 benchmarks spread across other natural languages, indicating some diversity but limited attention to non-English benchmarks. This distribution highlights the prominence of English in the global research community and also demonstrates the ***uneven representation*** of natural languages in the studied benchmarks.

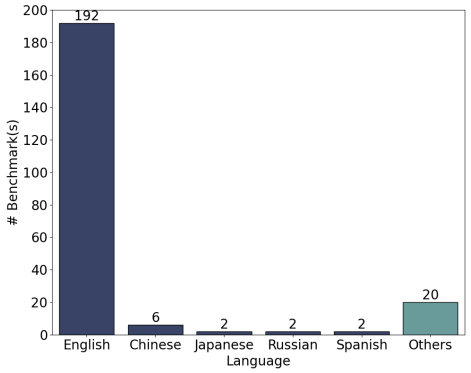

Figure 13: Benchmark Distribution over Natural Language

Modals in the benchmarks. Figure 14 presents the distribution of benchmarks according to the type of language used in their prompts. The chart shows that the majority, at 47.1%, of the benchmarks use a combination of natural language and programming Language, followed by PL only (31.0%) and NL only (21.9%).

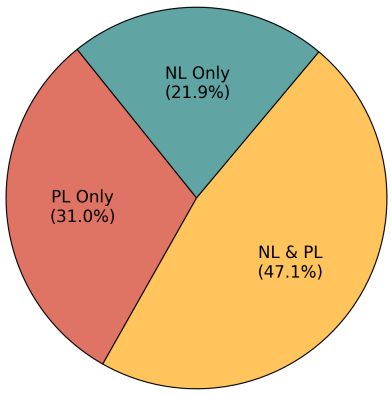

Figure 14: Benchmark Distribution over Modal in Prompt

Granularity. The code snippet in a code-related benchmark varies from statement-level (i.e., one line of code. For example, CoNaLa (Yin et al., 2018) and Math-QA (Amini et al., 2019)), function-level (i.e., a function unit of code. For example, HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021)), class-level (i.e., a class with multiple function units of code. For example, ClassEval (Du et al., 2023b)) and project-level (i.e., a project with multiple classes or modules. For example, DevEval (Li et al., 2024a) and JavaBench (Cao et al., 2024a)). Figure 15 illustrates the granularity levels at which benchmarks are typically conducted. The chart shows that the majority of bench-

marks, comprising ***71.8%, focus on the function level***. Projects constitute 15.0% of the benchmarks. Class-level granularity is the least represented at 2.6%.

> The majority of benchmarks are currently at the function level (70+%), followed by the project level (15+%). This indicates that the current major demand is for ***assessing individual functions within a single task***, followed by the demand for evaluating functionalities more aligned with ***actual project-level code development***.

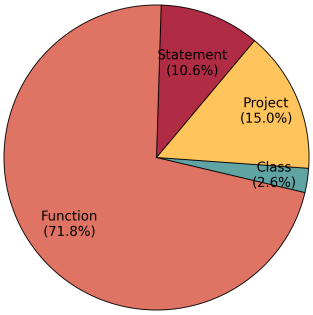

Figure 15: Benchmark Distribution over Granularity

B.2 Statistics about Benchmark Design

Design of Studied Capabilities. To understand whether benchmark developers recognize the capabilities of LLMs they aim to evaluate, we carefully analyzed 30 representative benchmarks (Appendix D) to see if they clearly specify the capabilities being assessed by their benchmarks. As shown in Figure 16, 90% of benchmarks explicitly specify the capabilities (e.g., intention understanding, problem solving, testing, debugging capabilities)to be evaluated, while 10% do not. The statistics show that the most highly cited benchmarks clearly define the assessment capabilities.

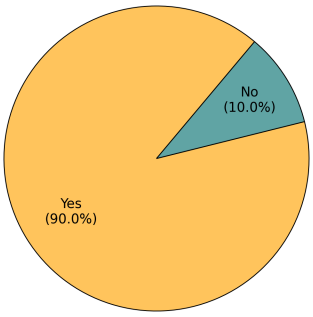

Figure 16: Benchmark Distribution Over Capabilities Consideration

Furthermore, we investigated the 30 focused benchmarks and identified a case (Figure 17) from MBPP (Austin et al., 2021) where the case is likely to fall outside of the targeted capability of the benchmark. In particular, MBPP (Austin et al., 2021) aims to "*measure the ability of these models to synthesize short Python programs from natural language descriptions*" for "*entry-level programmers*". As we can see from Figure 17, the prompt requires LLMs to "*Write a function to calculate the dogs' years.*" Simply from this description, an entry-level programmer is unlikely to write a correct program without knowing the conversion equation from dogs' year to dogs' age. In other words, this case is more about assessing whether LLMs have acquired this specific knowledge rather than evaluating the most fundamental programming skills.

Out of Targeted Capabilities

```
{
  'source_file': 'Benchmark Questions
                  Verification V2.ipynb',
  'task_id': 264,
  'prompt': 'Write a function to calculate
             a dog's age in dog's years.'
  'test_list': [ "assert dog_age(12)==61",
                 "assert dog_age(15)==73",
                 "assert dog_age(24)==109" ]
}
```

Figure 17: An Example of Out-of-capability Case from MBPP (Austin et al., 2021).

Design of Studied Application Scenarios. Similarly, to understand whether benchmark developers scoped the application scenarios of LLMs they aim to evaluate, we carefully analyzed 30 representative benchmarks (Appendix D) to see whether they explicitly specify the application scenarios their benchmarks target. As shown in Figure 18, 70% representative benchmarks have clearly specified application scenarios (e.g., programming assistant), while the rest do not. Indeed, clearly defining the application scenarios could help benchmark constructors establish precise goals for the design and development of the benchmark, ensuring accuracy in the evaluation.

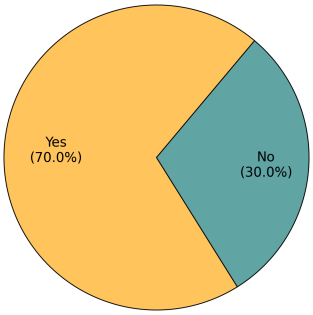

Figure 18: Benchmark Distribution Over Expected Application Scenario Consideration

B.3 Statistics about Data Preparation

B.3.1 Data Preprocessing

Data Deduplication. During benchmark preparation, data cleaning and preprocessing are necessary. However, as shown in Figure 19, only ***38% benchmarks have deduplicated*** the collected data. More than half of them didn't mention this process.

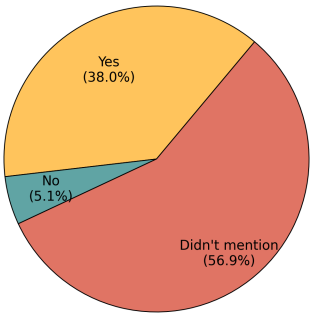

Figure 19: Benchmark Distribution over Deduplication

To investigate the situation, we went through the 30 representative benchmarks (Listed in Appendix D) and found two duplicated subjects in MBPP (Austin et al., 2021). Tasks with id `71` and `141` examined the same functionality, i.e., "*Write a function to sort a list of elements*.", collected from the same source.

> The significance of data preprocessing, such as ***deduplication***, is frequently ***overlooked*** by benchmark builders, leading to data duplication even in highly cited benchmarks.

Data Quality Assurance. Ensuring data quality for the benchmark is essential. However, our statistics (Figure 21) show disappointing results. ***67.9% of benchmarks do NOT take any measures for data quality assurance***. Among those benchmarks that do incorporate data quality measures, the majority rely on manual checks, which accounts for 22.6%. Other countermeasurements, such as code execution, constitute only 2.2%, while verification using LLMs accounts for 1.5%. Additional methods, such as using download counts as a basis, are also employed.

Duplicated Data

```
1  {
2    'source_file': 'Mike's Copy of Benchmark
3                    Questions Verification V2.ipynb',
4    'task_id': 71,
5    'prompt': 'Write a function to sort a list of elements. '
6  }
7
8  {
9    'source_file': 'Mike's Copy of Benchmark
10                   Questions Verification V2.ipynb',
11   'task_id': 141,
12   'prompt': 'Write a function to sort a list of elements. '
13 }
```

Figure 20: A Counterexample of Rule 16 from MBPP (Austin et al., 2021).

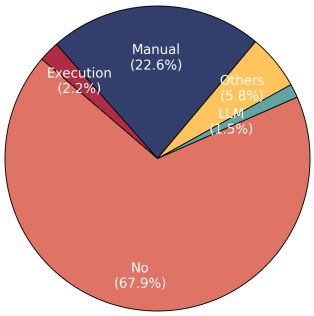

Figure 21: Benchmark Distribution over Quality Assurance Method

Additionally, we dived into the 30 representative benchmarks (Listed in Appendix D) and identified an example where the code cannot be executed successfully. As shown in Figure 22, the function `swap()` in line 7 has not been defined, so the execution of the code would fail if the code has been executed. This highlights a significant gap in ensuring the reliability and validity of benchmark data, underscoring the need for more rigorous and automated data quality assurance practices.

Data Contamination Resolution. Data contamination (Golchin and Surdeanu, 2023; Cao et al., 2024b) threat has been widely discussed. A benchmark with contaminated data may yield overclaimed results, misleading the understanding of the LLMs' capabilities. According to our statistics (Figure 23 on benchmarks from the year

Problem in Execution

```
Import math
def  min_Operations (A, B):
    """ Write a python function to find
    the minimum operations required
    to make two numbers equal. """
    if (A > B):
        swap(A,B)
    B = B // math.gcd(A,B);
    return B - 1
```

Figure 22: An Example from MBPP (Austin et al., 2021) that failed to be executed.

2023 to 2024 (the duration when most LLMs were launched), most (81.8 %) benchmarks were not aware of and have not taken any measures to alleviate data contamination, being vulnerable to data contamination threat.

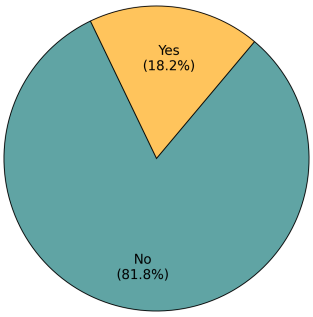

Figure 23: Benchmark Distribution over Quality Assurance on Data Contamination

B.3.2 Statistics about Data Curation

Ground truth solutions. Figure 24 shows that although the majority (92.3%) of benchmarks provide reference code as ground truth, there are 5% of benchmarks without reference code. Although it is not compulsory as long as object measurements (e.g., test cases) are provided, ***a reference code is still recommended.*** Indeed, if a benchmark provides reference code, its reliability tends to be better because it ensures that there are feasible solutions for the tasks involved. This guarantees that the tasks are theoretical and practically solvable, enhancing the benchmark's usefulness and credibility in real-world applications.

Additionally, ***the correctness of the ground truth solution*** should also be noted. Figure 25 shows an ***incorrect*** code solution provided in HumanEval (Chen et al., 2021a). This should draw

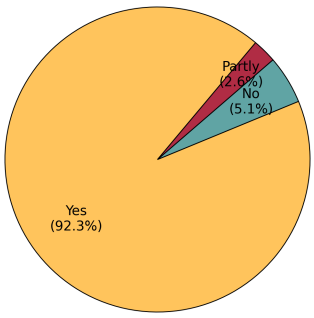

Figure 24: Benchmark Distribution over Solution

benchmark constructors' attention to the correctness of the benchmark reference code.

Incorrect Ground-truth

```
def check_dict_case(dict):
    """ Given a dictionary, return True if all keys are strings
    in lower case or all keys are strings in upper case, else
    return False. The function should return False is the
    given dictionary is empty. """
    if len(dict.keys()) == 0:
        return False
    else:
        state = "start"
        for key in dict.keys():
            if isinstance(key, str) == False:
                state = "mixed"
                break
            if state == "start":
                if key.isupper():
                    state = "upper"
                elif key.islower():
                    state = "lower"
                else:
                    break
            elif (state == "upper" and not key.isupper())
                    or (state == "lower" and not
                        key.islower()):
                state = "mixed"
                break
            else:
                break
    return state == "upper"  or state == "lower"
```

Figure 25: An Example from HumanEval (Chen et al., 2021a) which shows ***an incorrect solution*** provided in the benchmark.

Oracle. An oracle (Barr et al., 2014) is a way to determine whether the output is correct or not. For example, assume the output of LLMs is in the form of code, then an oracle could be running tests against the code and see whether the code can pass all the tests. Figure 26 shows the distribution of types of oracle that are used in these benchmarks. Clear that the ***exact match*** 41.97% (115/274) and ***test case passing*** (114/274) 41.6% are the most common oracle used in code-related benchmarks, followed by thresholds (i.e., similarities smaller than a specific threshold).

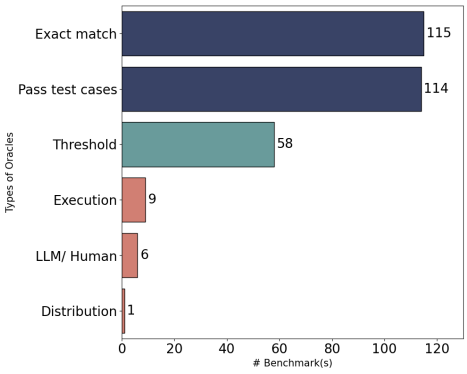

Figure 26: Benchmark Distribution over Test Orcale

Code test coverage (Ivanković et al., 2019), as a common oracle for code-related benchmarks, has been widely adopted to determine the output correctness. It should be considered if a benchmark uses test case passing as a criterion for the correctness of the generated code. Otherwise, a test could be too weak to detect the existence of a defect in the generated code. For example, as pointed out by prior work (Liu et al., 2023a), existing benchmarks such as HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021) still suffer from "insufficient tests", allowing incorrect code to pass all the tests without capturing the bugs.

Despite its importance, as shown in Figure 27, among the benchmarks that use test cases as the oracle, only 8.7% considered and reported "test coverage" explicitly in their papers, while 87.8% ignored the test coverage.

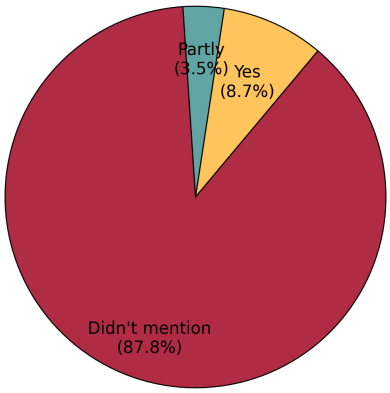

Figure 27: Benchmark Distribution over Test Coverage

Furthermore, we dived into 30 representative benchmarks (Listed in Appendix D) and identified an example (Figure 28) from MBPP (Austin et al., 2021) where the test is incorrect. It alerts us that both the quality of the test and the test adequacy (e.g., code coverage) should be considered.

Wrong Example Tests

```
{
  'source_file': 'charlessutton@: Benchmark
                  Questions Verification V2.ipynb',
  'task_id': 461,
  'prompt': 'Write a python function to count the
             upper case characters in a given string.'
  'test_list': [ "assert upper_ctr('PYthon') == 1",
                 "assert upper_ctr('BigData') == 1",
                 "assert upper_ctr('program') == 0" ]
}
```

Figure 28: An Example of Incorrect Tests from MBPP (Austin et al., 2021).

B.4 Statistics about Evaluation

Studied LLMs. We summarize the number of LLMs that have been evaluated in each benchmark evaluation. Among the 274 benchmarks, 183 of them are evaluated over LLMs, so we show the statistics over them. As shown in Figure 29, over 34% of the benchmarks were evaluated on fewer than 3 LLMs, with 11.48% benchmarks only evaluated on one LLM. Such evaluation results can hardly be generalized to other LLMs. Furthermore, ***more than half of the benchmarks studied fewer than 6 LLMs*** (51% = (21 +22 + 20 + 4 + 12+15)/183).

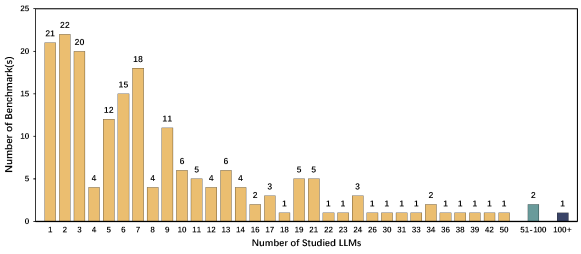

Figure 29: Benchmark Distribution over LLM Experimented

Additionally, we listed the top-10 LLMs by the number of code-related benchmarks they have been evaluated, as shown in Figure 30. GPT series leads significantly with 116 benchmarks, suggesting its widespread adoption and possibly its versatility or superior performance in handling code-related tasks. The rest, including CodeLlama, StarCoder, CodeGen, and others, show varying degrees of involvement, with numbers ranging from 60 down to 24 benchmarks for Claude. This figure may provide a reference for choosing which model to

evaluate. In addition, it is worth mentioning that different LLMs should be considered considering different coding tasks.

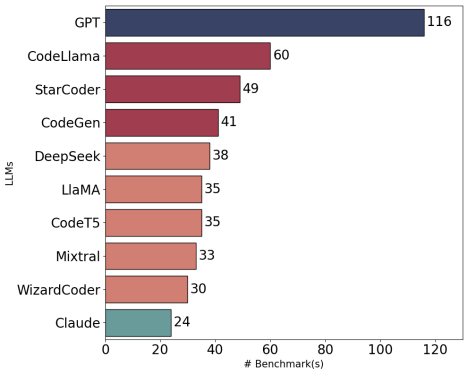

Figure 30: Top-10 Studied LLMs for Code-related Benchmarks

Experiment Environments. The experimental environment (such as the operating system and hardware) is important for the reproduction of the experiment. However, Figure 32 and Figure 31 highlight a significant gap. A mere 27.4% of benchmarks document the devices used in their experiments, leaving a substantial 72.6% that do not. The situation appears even more dire when considering os, with only 3.6% of benchmarks documenting the OS used, while a staggering 96.4% neglect to record this information.

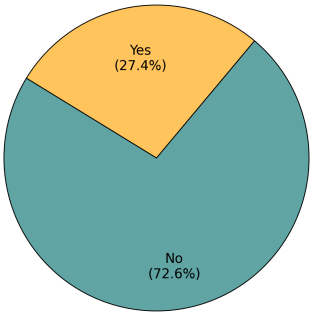

Figure 31: Benchmark Distribution over Recording Experiment Devices

Prompting and Prompting Strategies Prompting has a direct impact on the quality of the LLMs' output results (Wei et al., 2022; He et al., 2024a; Jin et al., 2024). So, we summarized whether different prompting strategies have been evaluated and statistics the distribution. Figure 33 shows the usage of four kinds of prompts: zero-shot, few-shot, chain-of-thought, and retrievals (RAG). From Figure 33, we can see that a vast majority (94.9%) benchmarks were evaluated in a zero-shot context setting, while only 21.2% benchmarks were evaluated in a few-shot manner. Even fewer benchmarks were evaluated under the COT and RAG settings, utilized by only 8.8% and 2.6%.

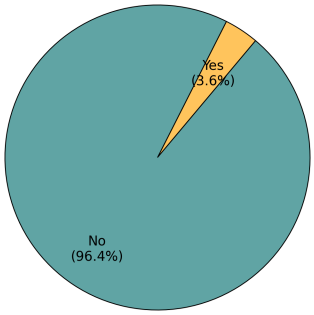

Figure 32: Benchmark Distribution over Recording Experiment OS

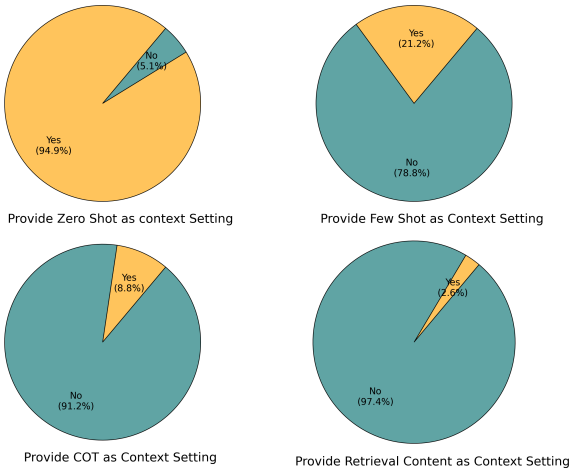

Figure 33: Benchmark Distribution over Context Setting

Prompt Quality The prompt quality also greatly impacts the LLM evaluation (He et al., 2024b). So, carefully designing a prompt needs consideration. However, as shown in Figure 34, 73.3% representative benchmarks (Appendix D) do not validate whether the prompt they used is well-designed.

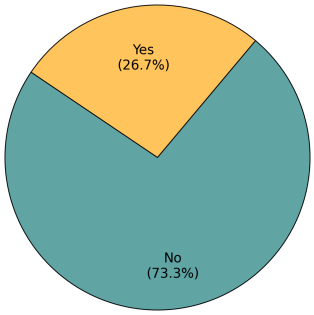

Figure 34: Benchmark Distribution Over Validation of Prompts

Repeated Experiment Given the random na-

ture of LLMs, the experiments are expected to repeat, ensuring the stability and reliability of the results. However, as shown in Figure 35, ***only 35.4% benchmarks went through a repeated experiment***, while a majority of 64.6% opted against repeating the experiment. This reflects a lack of awareness regarding the stability and reproducibility of evaluations.

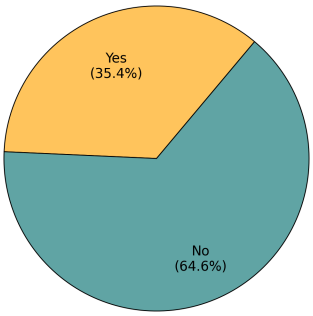

Figure 35: Benchmark Distribution over Repeating the Experiment

B.5 Statistics about Analysis

Experiment Explanation. Explaining experiment results is crucial for other practitioners to understand what the outcomes mean in the context of the research questions. So, we investigate whether the representative benchmarks (Appendix D) have explained the experiment results. As shown in Figure 36, 70% benchmarks have detailed explanations and analyses of their evaluation results, while still 30% have not. Indeed, an explanation contributes to the body of knowledge by making it possible to understand and compare results with previous studies, promoting transparency within the community.

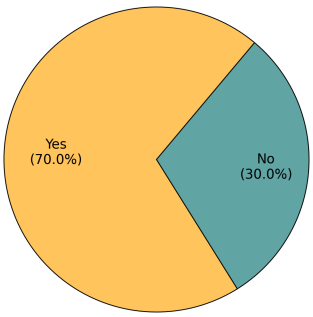

Figure 36: Benchmark Distribution Over Explaining the Experiment

A clear and precise presentation of experimental results is important for enabling robust interpretation and comparison across benchmarks. However, further examination of the 30 representative benchmarks (listed in Appendix D) revealed notable deficiencies in result visualization. As shown in Figure 37, CruxEval (Gu et al., 2024) exhibits unclear experimental result presentation. Specifically, the scatter plot suffers from ambiguous labeling, poor readability of axis values, and inconsistent marker encoding, making it difficult for researchers to extract meaningful insights. Such presentation shortcomings obscure the performance relationships between methods and compromise the benchmark's usability for fair evaluation. To address these issues, benchmarks should adopt standardized and well-documented visualization practices, ensuring results are interpretable, accessible, and reproducible.

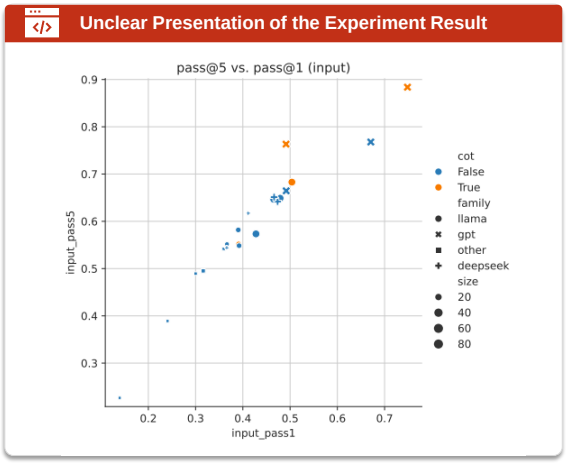

Figure 37: An Example of Unclear Experiment Analysis and Display from CruxEval (Gu et al., 2024)

B.6 Statistics about Release

Data Accessibility. The fundamental requirement for releasing a benchmark is that it must be open-sourced. However, surprisingly, as shown in Figure 38, we observed that 5.1% of the benchmarks are only partially open-sourced (e.g., missing some subjects or tests), and ***5.8% are not open-sourced at all*** (e.g., links/web pages are no longer active).

Prompt Accessibility. Detailed prompts are essential for ensuring the reproducibility and transparency of code-related benchmarks. Yet, Figure 39 indicates that ***52.6% of benchmarks do not provide detailed prompts***, limiting the ability to accurately replicate and evaluate the performance of LLMs. Lacking prompt disclosure highlights a gap in benchmark design practices, as prompts are often indispensable for understanding model perfor-

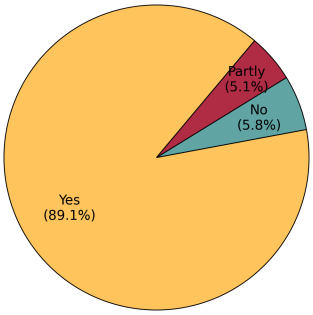

Figure 38: Benchmark Data Availability

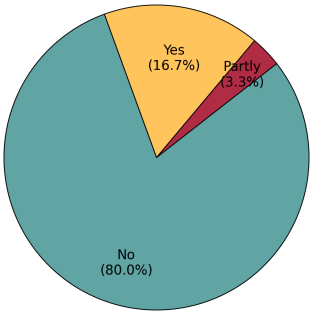

Figure 40: Availability of Logging Information

mance under specific conditions. While 47.4% of benchmarks include such prompts, the absence of comprehensive prompt documentation in over half of the cases raises concerns about the consistency and reproducibility of reported results.

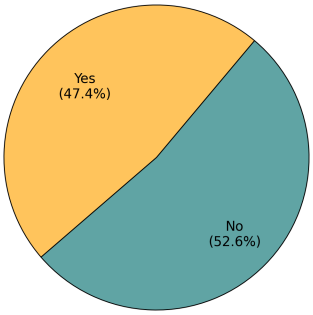

Figure 39: Availability of Prompts

Logging Info Accessibility. Providing detailed logging information, including comprehensive experimental results, is essential for ensuring transparency, verifiability, and reproducibility in benchmarking research. However, as shown in Figure 40, only ***16.7% of the benchmarks make their experimental results publicly available***, while 80.0% fail to disclose this critical information. Alarmingly, an additional 3.3% provide only partial logging details, further complicating result verification. The absence of complete logging information creates significant barriers for researchers attempting to reproduce experiments or validate reported findings, thereby undermining the reliability of benchmarks. To address this, we emphasize the necessity of making detailed logging information, including intermediate results and metrics, publicly accessible to uphold rigorous scientific standards and foster trustworthy comparisons across models.

User Manual Accessibility. A high-quality user manual, such as a well-documented README file, is crucial for enhancing benchmark usability, enabling users to understand the dataset, execute provided scripts, and reproduce results efficiently. However, our analysis revealed that a significant number of benchmarks lack comprehensive user manuals, hindering accessibility and adoption. As depicted in Figure 41, poorly structured or incomplete manuals often omit essential components such as benchmark introductions, usage instructions, and evaluation scripts. This creates unnecessary barriers for researchers who rely on these manuals for setup and experimentation. To address this, we advocate for benchmarks to include clear, standardized user manuals that provide an overview of the benchmark, step-by-step execution guides, and troubleshooting instructions, ensuring a seamless and reproducible user experience.

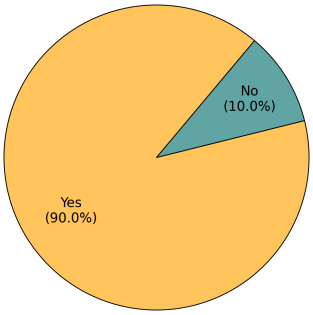

Figure 41: Availability of User Manual

Convenient Evaluation Interface Availability. Providing convenient evaluation interfaces is essential for enhancing the usability and accessibility of benchmarks, enabling researchers to easily reproduce results and compare models. As shown in Figure 42, ***16.7% of benchmarks fail to offer any evaluation interfaces***, imposing significant barriers to usability. While a majority of benchmarks (83.3%) provide such interfaces, including command-line tools, Docker images, or scripts, the absence of standardized and user-friendly evaluation tools in a notable minority of cases highlights an area for improvement. Benchmarks without

convenient evaluation interfaces require users to spend additional effort in setup and result verification, which can discourage adoption and hinder reproducibility. To address this, we emphasize the importance of releasing benchmarks with well-documented, ready-to-use evaluation pipelines to promote efficient, reliable, and fair benchmarking practices.

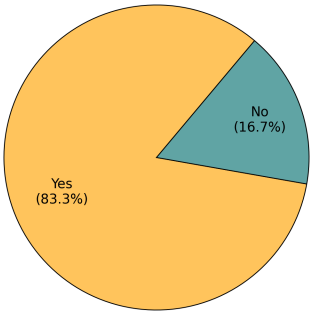

Figure 42: Availability of Convenient Evaluation Interfaces

Temperature Records. One critical parameter for benchmarking is the temperature setting, which influences stochasticity in LLMs. As shown in Figure 43, we observed that ***57.3% of benchmarks fail to record the temperature setting***, hindering reproducibility and fair evaluation. While 42.7% of benchmarks do document this parameter, the majority omission highlights an overlooked yet essential aspect of benchmark transparency.

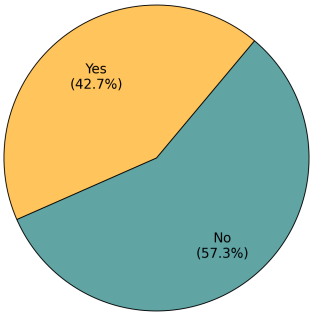

Figure 43: Benchmark Distribution over Recording Temperature

License Provision. Releasing benchmarks under a clear and accessible license is fundamental for legal compliance and ensuring open collaboration. Figure 44 reveals that ***19.3% of benchmarks do not provide a license***, limiting their usability and distribution. Encouragingly, 80.7% of benchmarks do include a license, but the lack of licensing in nearly one-fifth of the benchmarks raises concerns about widespread adoption and usage. These findings emphasize the need for standardized practices in benchmark releases to promote legal clarity and accessibility.

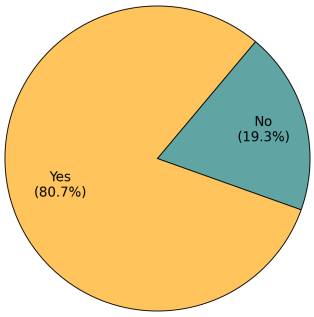

Figure 44: Provision of License

Data Security. Ensuring data security is a critical yet often overlooked aspect of benchmark development. Sensitive information, such as API keys, credentials, or private tokens, should never be included in benchmark releases. However, further investigation into 30 representative benchmarks (listed in Appendix D) revealed instances of sensitive data leakage. As shown in Figure 45, XSemPLR (Zhang et al., 2023b) inadvertently included an ***API key*** in its release, a critical oversight that can expose resources to external exploitation. Similarly, Figure 46 highlights an example from

API Key Leakage

```
#! /bin/bash

# conda activate skg
#export WANDB_API_KEY=**************************
export WANDB_PROJECT=mt5-large_mgeoquery_few-shot
export CUDA_LAUNCH_BLOCKING=1
```

Figure 45: An Example of API Key Leakage in Benchmark Release from XSemPLR (Zhang et al., 2023b).

CrossVul (Nikitopoulos et al., 2021), where ***personal names and email addresses*** were unintentionally disclosed. Such leakage poses risks of unauthorized access and resource misuse, potentially compromising systems and research integrity.

Usability. Clear and comprehensive documentation is crucial for ensuring the usability of benchmarks, as poorly written instructions can significantly hinder adoption and reproducibility. We dived into the 30 representative benchmarks (listed in Appendix D) and identified an example where

Name or Email Leakage

```
10942  Individual \fIreadline\fP initialization file
10943  .PD
10944  .SH AUTHORS
10945  Brian Fox, Free Software Foundation
10946  .br
10947  bfox@gnu.org
10948  .PP
10949  Chet Ramey, Case Western Reserve University
10950  .br
10951  chet.ramev@case.edu
```

Figure 46: An Example of Name & Email Leakage in Benchmark Release from CrossVul (Nikitopoulos et al., 2021).

the README file provided insufficient and unclear information. As shown in Figure 47, VulDeePecker (Li et al., 2018b) includes a less-than-ideal ReadMe file, which lacks essential usage instructions and evaluation guidelines, making the benchmark difficult to understand and deploy.

A Less-than-ideal Readme File

README Apache-2.0 license

Database of "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection" (NDSS'18)

Code Gadget Database (CGD) focuses on two types of vulnerabilities in C/C++ programs, buffer error vulnerability (CWE-119) and resource management error vulnerability (CWE-399). Each code gadget is composed of a number of program statements (i.e., lines of code), which are related to each other according to the data flow associated to the arguments of some library/API function calls.

Based on the National Vulnerability Database (NVD) and the NIST Software Assurance Reference Dataset (SARD) project, we collect 520 open source software program files with corresponding diff files and 8,122 test cases for the buffer error vulnerability, and 320 open source software program files with corresponding diff files and 1,729 test cases for the resource management error vulnerability.

In total, the CGD database contains 61,638 code gadgets, including 17,725 code gadgets that are vulnerable and 43,913 code gadgets that are not vulnerable. Among the 17,725 code gadgets that vulnerable, 10,440 corresponds to buffer error vulnerabilities and the rest 7,285 corresponds to resource management error vulnerabilities.

Explanation

This Readme file only provides limited information of the dataset.

Figure 47: An Example of Unreadable and Hard-to-Use README in Benchmark Release from VulDeePecker (Li et al., 2018b).

Convenient Evaluation Interfaces

First generate the code outputs

```
python3 generate_gpt_codes.py  --save /path/to/save_dir
```

Second evaluate the accuracy of the outputted code

```
python3 test_one_solution.py --save /path/to/save_dir
# because the above may fail on account of poorly generated python programs
# we suggest to run a for loop for each problem index against the "all_codes.json"
for i in {0..#Num_Problems#} ; do
  python3 test_one_solution.py --save /path/to/save_dir -i $i ;
done
```

The above will output the accuracy but to run it again once the evaluations have completed execute the line below:

```
python3 test_one_solution.py --save /path/to/save_dir --print_results
```

Third evaluate the bleu scores of the outputted code

```
python3 eval_bleu.py --save /path/to/save_dir
```

Figure 48: A Good Example of Easy-to-Read README in Benchmark Release from APPS (Hendrycks et al., 2021).

In contrast, Figure 48 highlights APPS (Hendrycks et al., 2021), which provides well-structured and easy-to-follow documentation. The APPS benchmark includes step-by-step instructions for generating, evaluating, and analyzing results, enabling users to efficiently reproduce experiments.

These observations emphasize the importance of high-quality documentation for benchmarks to enhance accessibility, reduce friction in usage, and foster reproducible research.

C Details of Human Study

C.1 Interviewee Selection

The selection of interviewees is pivotal to ensuring the representativeness and relevance of the data collected. This involves identifying individuals with the knowledge or experience pertinent to the research theme.

To this end, we chose graduate students from SE or AI fields who have published at least one paper. This criterion ensures that participants have research experience and judgment capabilities. The focus on SE and AI fields is due to their likely interest in code benchmarks. Particularly, we aimed to recruit individuals who have published papers on code benchmarks to obtain firsthand feedback from experienced benchmark developers.

C.2 Survey Question Design

Questions. The body of the survey was divided into five stages of benchmark development (following Figure 1), with necessary background information provided for each stage. Each criterion in HOW2BENCH was slightly modified to be in the first-person perspective, making it easier for interviewees to empathize and answer the questions from their own viewpoint. Finally, to facilitate comprehension, questions and instructions were translated into both English and Chinese.

Question Setting. To minimize the effort required from respondents, we designed ***single-choice questions*** with four options:

❒ I found it **important**, and I **have done** it.
❒ I found it **important**, although I **haven't done** it.
❒ I found it **not important**, but I **have done** it.
❒ I found it **not important**, and I **wouldn't** do it.

This format is intended to orthogonally explore the correlation between ***awareness*** and ***behavior***.

C.3 Interview Process

Questionnaire Distribution The questionnaire was distributed via online platforms, targeting academic and professional networks related to SE and AI. *The distribution started on October 27, 2024, and ended on November 27th, 2024*, lasting one month.

Results Collection The responses were automatically collected through the online platform used for distribution.

Survey Screening Since the requirement was for participants who have published papers, responses from those selecting "No" to having published a paper were excluded. Also, incomplete surveys where not all questions were answered were also considered invalid and excluded from the analysis.

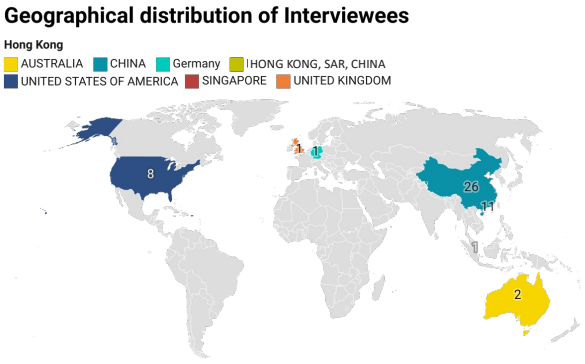

Figure 49: Geographical Distribution of Interviewees

C.4 Interview Result Analysis

In total, we collected 50 responses. The respondents were from seven regions, including the United States, the United Kingdom, Germany, Australia, China, and others, as shown in Figure 49. Only one survey was invalid due to the respondent selecting "have not published a paper", leaving **49 valid surveys** for analysis. A breakdown of the respondents' demographics is shown in Figure 50. The detailed responses for all 55 criteria in HOW2BENCH are shown in Figure 51 and Figure 52.

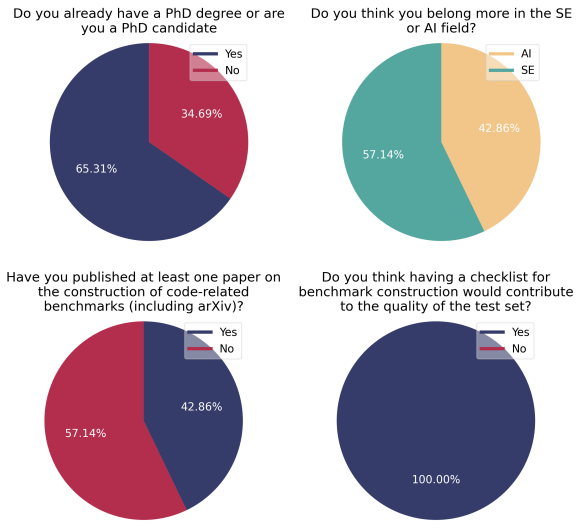

Figure 50: Demography of Interviewees

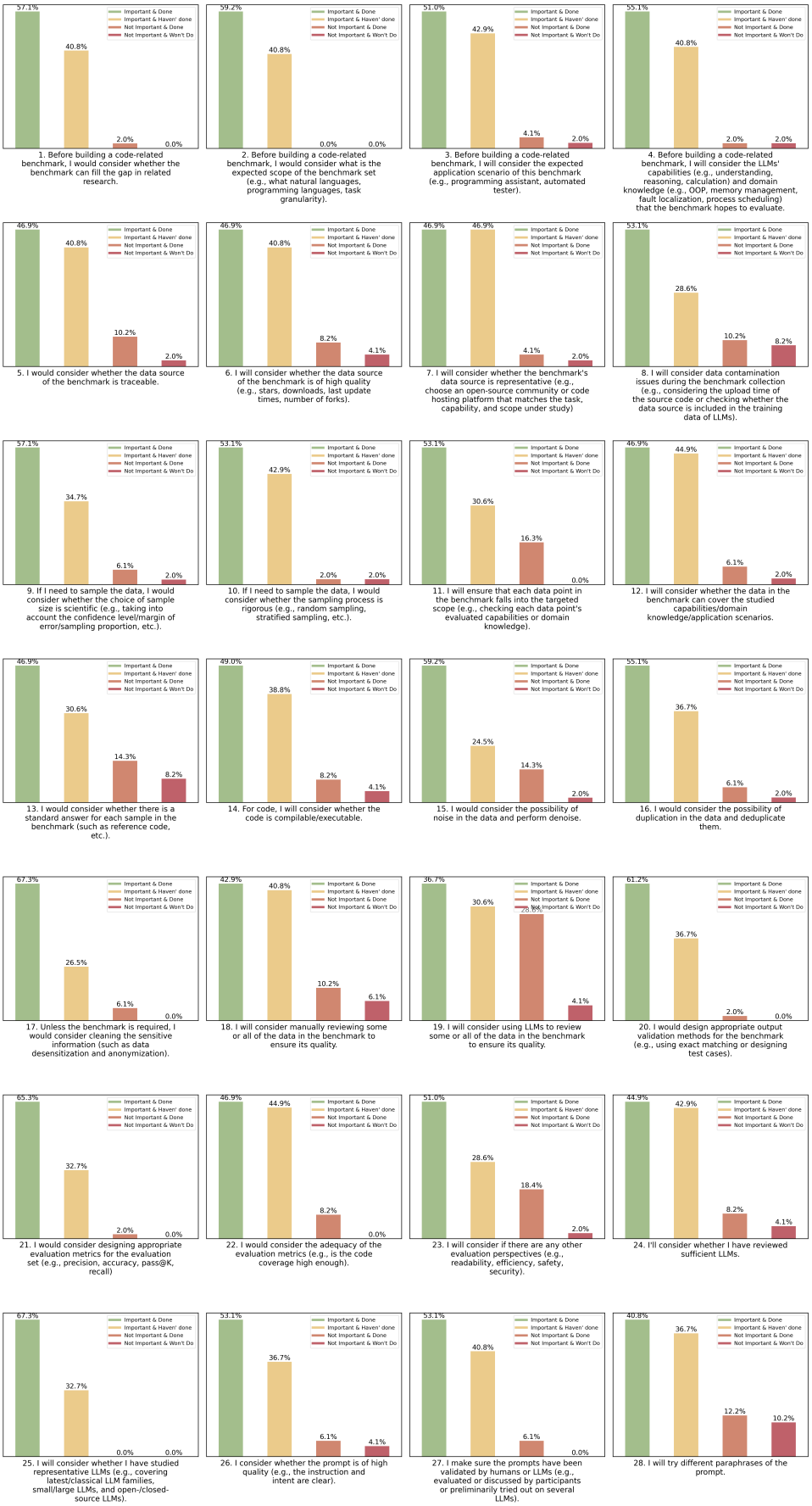

Figure 51: Results of Human Study (Questions 1 - 28

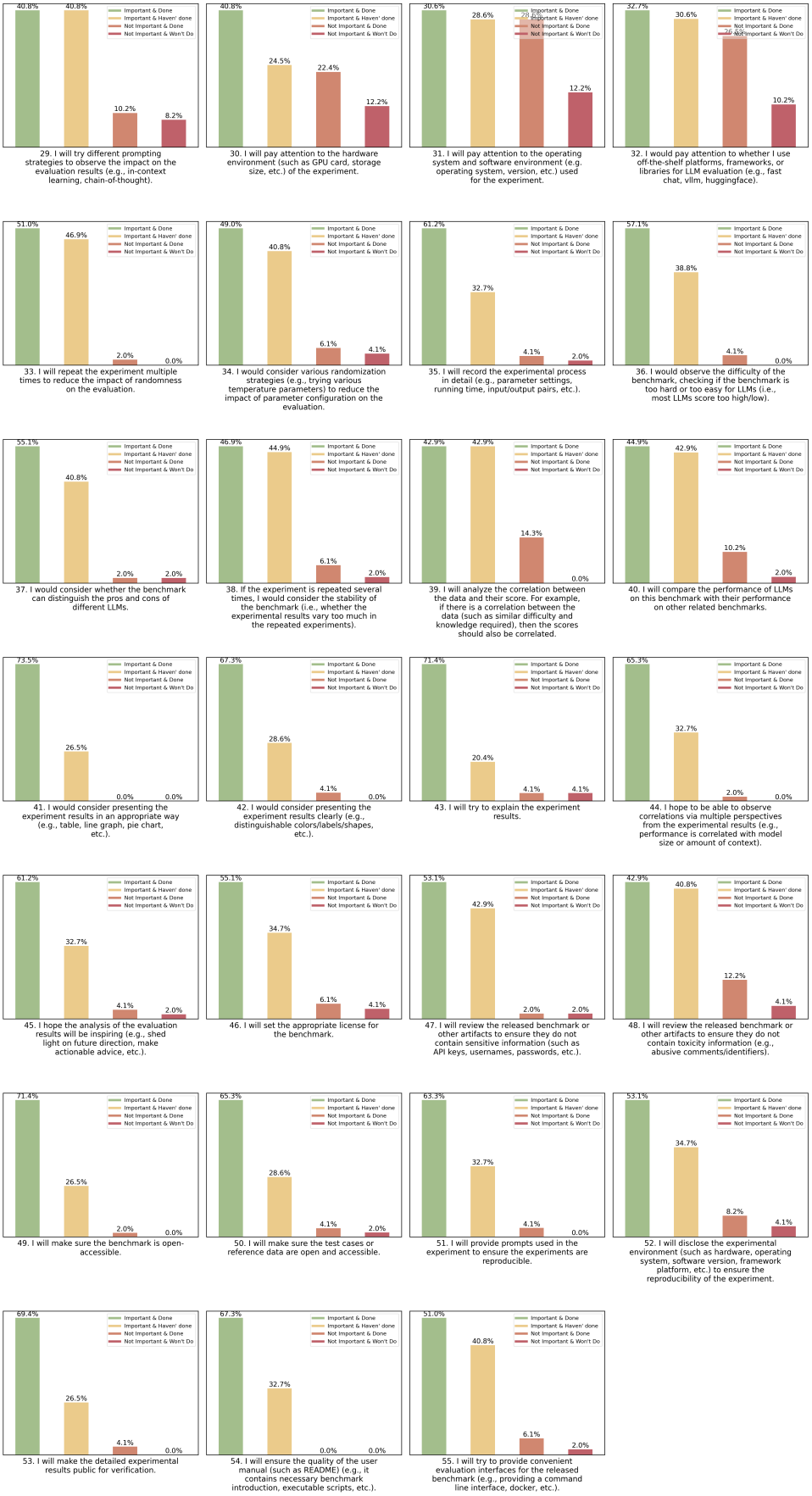

Figure 52: Results of Human Study (Questions 29 - 55

D List of Studied Benchmarks (Focused Ones)

Code Generation: Five with top citations:

- HumanEval (Chen et al., 2021a)
- MBPP (Austin et al., 2021)
- CodeContest (Li et al., 2022)
- leetcodehardgym (Shinn et al., 2023)
- APPS (Hendrycks et al., 2021)

The latest one as of 31/8/2024:

- VerilogEval (Pinckney et al., 2024)

Defect Detection: Five with top citations:

- VulDeePecker (Li et al., 2018b)
- Devign (Zhou et al., 2019)
- Chromium and Debian (Chakraborty et al., 2022)
- μVulDeePecker (Zou et al., 2020)
- Synthetic Dataset (Hellendoorn et al., 2020)

The latest one as of 31/8/2024:

- VulDetectBench (Liu et al., 2024c)

Program Repair: Five with top citations:

- Defects4J (Just et al., 2014)
- BFP (Tufano et al., 2019)
- MANYBUGS, INTROCLASS (Le Goues et al., 2015)
- HumanEval-Java (Jiang et al., 2023)
- QuixBugs (Prenner et al., 2022)

The latest one as of 31/8/2024:

- DebugBench (Tian et al., 2024)

Code Summarization: Five with top citations:

- CODE-NN (Iyer et al., 2016)
- Java-small/med/large (Alon et al., 2019)
- code-summarization-public (Wan et al., 2018)
- HumanEvalPack (Muennighoff et al., 2024)
- Shrivastava et al. (Shrivastava et al., 2023b)

The latest one as of 31/8/2024:

- Long Code Arena (Bogomolov et al., 2024)

Text To SQL: Five with top citations:

- WikiSQL (Zhong et al., 2017)
- Spider (Yu et al., 2018)
- Advising (Finegan-Dollak et al., 2018)
- BIRD (Li et al., 2023a)
- Spider-Realistic (Deng et al., 2021)

The latest one as of 31/8/2024:

- AMBROSIA (Saparina and Lapata, 2024)

E List of Studied Benchmarks (Full)

We collected and studied 274 code-related benchmarks. We then listed and grouped them by year.

2024:

- CodeEditorBench (Guo et al., 2024)
- MHPP (Dai et al., 2024)
- LiveCodeBench (Jain et al., 2024)
- CodeAgentBench (Zhang et al., 2024a)
- CruxEval (Gu et al., 2024)
- BigCodeBench (Zhuo et al., 2024)
- OOPEval (Wang et al., 2024b)
- DevEval (Li et al., 2024a)
- Long Code Arena (Bogomolov et al., 2024)
- CodeRAGBench (Wang et al., 2024d)
- ScenEval (Paul et al., 2024)
- AICoderEval (Xia et al., 2024b)
- VersiCode (Wu et al., 2024b)
- VHDL-Eval (Vijayaraghavan et al., 2024)
- NaturalCodeBench (Zhang et al., 2024b)
- CodeGuard+ (Fu et al., 2024)
- PECC (Haller et al., 2024)
- USACO (Shi et al., 2024b)
- ParEval (Nichols et al., 2024)
- MxEval (Athiwaratkun et al., 2022)
- MMCode (Li et al., 2024c)
- Plot2Code (Wu et al., 2024a)
- ChartMimic (Shi et al., 2024a)
- DebugBench (Tian et al., 2024)
- PythonIO (Zhang et al., 2024c)
- StaCCQA (Yang et al., 2024a)
- RepoQA (Liu et al., 2024a)
- PRIMEVUL (Ding et al., 2024a)
- VulDetectBench (Liu et al., 2024c)
- ProCQA (Li et al., 2024e)
- CoSQA+ (Gong et al., 2024) (Huang et al., 2021)
- JavaBench (Cao et al., 2024a)
- HumanEvo (Zheng et al., 2024)
- REPOEXEC (Hai et al., 2024)
- EHR-SeqSQL (Ryu et al., 2024)
- BookSQL (Kumar et al., 2024)
- AMBROSIA (Saparina and Lapata, 2024)
- WUB, WCGB (Yun et al., 2024)
- RES-Q (LaBash et al., 2024)
- PythonSaga (Yadav et al., 2024b)
- Mercury (Du et al., 2024)
- ENAMEL (Qiu et al., 2024b)
- RealHumanEval (Mozannar et al., 2024)
- CoderUJB (Zeng et al., 2024)
- EvoEval (Xia et al., 2024a)
- ML-Bench (Liu et al., 2023c)
- VerilogEval (Pinckney et al., 2024)
- CodeApex (Fu et al., 2023)
- HumanEvalPack (Muennighoff et al., 2024)
- HumanEval+ (Liu et al., 2023b)
- HumanEval-X (Zheng et al., 2023a)
- XCodeEval (Khan et al., 2024)
- CoderEval (Yu et al., 2023)
- CodeXGLUE (Lu et al., 2021)
- VulnPatchPairs (Risse and Böhme, 2024)
- WikiSQL (Zhong et al., 2017)
- CrossCodeEval (Ding et al., 2023)
- SWE-bench (Jimenez et al., 2024)
- BAIRI et al. (Bairi et al., 2024)
- BioCoder (Tang et al., 2024)
- RepoBench (Liu et al., 2024b)

- NoFunEval (Singhal et al., 2024)
- CoCoMIC (Ding et al., 2024b)
- Java-small/med/large (Alon et al., 2019)
- FixEval (Haque et al., 2023)
- CommitBench (Schall et al., 2024)
- InfiAgent-DABench (Hu et al., 2024)
- InfiBench (Li et al., 2024d)
- Design2Code (Si et al., 2024)
- MatPlotBench (Yang et al., 2024b)
- EditEval (Li et al., 2024b)
- D1, D2, D3 (Huang et al., 2024)
- RepoEval (Liao et al., 2024)
- BetterTypes4Py, InferTypes4Py (Wei et al., 2023)
- HumanEval-Java (Jiang et al., 2023)
- PIE (Shypula et al., 2024)
- EvalGPTFix (Zhang et al., 2023a)
- EHRSQL (Lee et al., 2023)
- Spider2-V (Cao et al., 2024c)
- TESTEVAL (Wang et al., 2024c)
- ChatTester (Yuan et al., 2023b)
- Code Lingua (Pan et al., 2024)
- EffiBench (HUANG et al., 2024)
- CRUXEval-X (Xu et al., 2024)
- DomainEval (Zhu et al., 2024)

2023:

- MCoNaLa (Wang et al., 2023b)
- MultiPL-E (Cassano et al., 2022)
- ODEX (Wang et al., 2022)
- TACO (Li et al., 2023b)
- DOTPROMPTS, MGDMI-CROBENCH (Agrawal et al., 2023)
- StudentEval (Babe et al., 2024)
- CodeTransOcean (Yan et al., 2023)
- G-TransEval (Jiao et al., 2023)
- AVATAR (Ahmad et al., 2023)
- RunBugRun (Prenner and Robbes, 2023)
- VulBench (Gao et al., 2023b)
- DiverseVul (Chen et al., 2023)
- Hellendoorn et al. (Hellendoorn et al., 2020)
- XSemPLR (Zhang et al., 2023b)
- BIRD (Li et al., 2023a)
- Stack-Repo (Shrivastava et al., 2023a)
- RepoEval (Liao et al., 2024)
- MTPB (Nijkamp et al., 2022)
- ARCADE (Yin et al., 2023)
- Shrivastava et al. (Shrivastava et al., 2023b)
- Grag et al. (Garg et al., 2022)
- GSM-HARD (Gao et al., 2023a)
- InferredBugs (Jin et al., 2023)
- LeetcodeHardGym (Shinn et al., 2023)
- APIBench (Patil et al., 2023)
- ClassEval (Du et al., 2023b)
- CommitChronicle (Eliseeva et al., 2023)
- TeCo (Nie et al., 2023)
- TESTPILOT (Schäfer et al., 2024)

2022:

- AixBench (Hao et al., 2022)
- TypeBugs (Oh and Oh, 2022)
- XLCoST (Zhu et al., 2022)
- CS1QA (Lee et al., 2022)
- Chromium and Debian (Chakraborty et al., 2022)
- Spider-Realistic (Deng et al., 2021)
- Spider-SS (Gan et al., 2022)
- DSP (Chandel et al., 2022)

- CodeContest (Li et al., 2022)
- PandasEval, NumpyEval (Zan et al., 2022b)
- TorchDataEval, MonkeyEval, BeatNumEval (Zan et al., 2022a)
- DS-1000 (Lai et al., 2023)
- MCMD (Tao et al., 2022)
- ExeDS (Huang et al., 2022)
- QuixBugs (Prenner et al., 2022)
- ManyTypes4Py v0.7 (Mir et al., 2022)

2021:

- SySeVR (Li et al., 2018a)
- Ling&Wu et al. (Ling et al., 2021)
- Chen et al. (Chen et al., 2021b)
- MBPP, MathQA-Python (Austin et al., 2021)
- HumanEval (Chen et al., 2021a)
- APPS (Hendrycks et al., 2021)
- Berabi et al. (Berabi et al., 2021)
- CrossVul (Nikitopoulos et al., 2021)
- PYPIBUGS, RANDOMBUGS (Allamanis et al., 2021)
- D2A (Zheng et al., 2021)
- CodeQA (Liu and Wan, 2021)
- Spider-DK (Gan et al., 2021b)
- KaggleDBQA (Lee et al., 2021)
- SEDE (Hazoom et al., 2021)
- Spider-Syn (Gan et al., 2021a)
- CoDesc (Hasan et al., 2021)
- Methods2Test (Tufano et al., 2022)
- Rozière et al. (Rozière et al., 2022)

2020:

- Lachaux&Roziere et al. (Rozière et al., 2020)
- μVulDeePecker (Zou et al., 2020)
- CosBench (Yan et al., 2020)
- PACS (Heyman and Cutsem, 2020)
- Criteria2SQL (Yu et al., 2020)
- SQUALL (Shi et al., 2020)
- Hu et al. (Hu et al., 2019)
- CodeSearchNet Challenge (Husain et al., 2019)
- MIMICSQL (Wang et al., 2020)
- Atlas (Watson et al., 2020)
- Liu et al. (Liu et al., 2022)
- Android (Agarwal et al., 2020)
- CCSD (Liu et al., 2021)

2019:

- BFP (Tufano et al., 2019)
- SARD (Lin et al., 2019)
- Spider (Yu et al., 2018)
- JuICe (Agashe et al., 2019)
- Nguyen et al. (Nguyen et al., 2019)
- Lin et al. (Lin et al., 2021)
- Zhou et al. (Zhou et al., 2019)
- CoSQL (Yu et al., 2019a)
- SParC (Yu et al., 2019b)
- Malik (Malik et al., 2019)
- LeClair (LeClair et al., 2019)

2018:

- CoNaLa (Yin et al., 2018)
- DeepCom (Hu et al., 2018a)
- TL-CodeSum (Hu et al., 2018b)
- code-summarization-public (Wan et al., 2018)
- Russell et al. (Russell et al., 2018)
- VulDeePecker (Li et al., 2018b)
- Lin et al. (Lin et al., 2018)
- StaQC (Yao et al., 2018)
- Advising (Finegan-Dollak et al., 2018)

- ConCode (Iyer et al., 2018)
- NNGen (Liu et al., 2018)
- Gu et al. (Gu et al., 2018)

2017:

- QuixBugs (Lin et al., 2017)
- the DeepFix dataset (Gupta et al., 2017)
- Barone et al. (Barone and Sennrich, 2017)

2016:

- CODE-NN (Iyer et al., 2016)
- Mou et al. (Mou et al., 2016)

2015:

- MANYBUGS, INTROCLASS (Le Goues et al., 2015)

2014:

- Defects4j (Just et al., 2014)
- BigCloneBench (Svajlenko et al., 2014)

F Guideline

Finally, for ease of printing and use, we organized the guideline HOW2BENCH into a clear, color-coded checklist (4 pages in total) that is easy to print, attached at the end of the paper.

	HOW-TO-BENCH (1/4)		
	Phase 1. Benchmark Design	**Priority**	☑
1	**Consider whether the benchmark can fill the gap in related research.**	★★★	☐
2	**Consider what is the expected scope of the benchmark set** (e.g., what natural languages, programming languages, task granularity)**.**	★★★	☐
3	**Consider the expected application scenario of this benchmark** (e.g., programming assistant, automated tester).	★★★	☐
4	**Consider the LLMs' capabilities** (e.g., understanding, reasoning, calculation) **and domain knowledge** (e.g., OOP, memory management, fault localization, process scheduling) **that the benchmark hopes to evaluate.**	★★★	☐
	Phase 2. Benchmark Construction (1/2)	**Priority**	☑
5	**Consider whether the data source of the benchmark is traceable.**	★	☐
6	**Consider whether the data source of the benchmark is of high quality** (e.g., stars, downloads, last update times, number of forks).	★★	☐
7	**Consider whether the benchmark's data source is representative** (e.g., choose an open-source community or code hosting platform that matches the task, capability, and scope under study)	★	☐
8	**Consider data contamination issues during the benchmark collection** (e.g., considering the upload time of the source code or checking whether the data source is included in the training data of LLMs).	★	☐
9	**If data sampling is needed, consider whether the choice of sample size is scientific** (e.g., considering the confidence level/margin of error/sampling proportion, etc.).	★	☐
10	**If data sampling is needed, consider whether the sampling process is rigorous** (e.g., random sampling, stratified sampling, etc.).	★	☐
11	**Ensure each data point in the benchmark falls into the targeted scope** (e.g., checking each data point's evaluated capabilities or domain knowledge).	★	☐
12	**Consider whether the data in the benchmark can cover the studied capabilities/domain knowledge/application scenarios.**	★★★	☐
13	**Consider whether there is a standard answer for each sample in the benchmark** (such as reference code, etc.).	★★	☐
14	**For code, consider whether the code is compilable/executable.**	★	☐
15	**Consider the possibility of noise in the data and perform denoise.**	★★	☐
16	**Consider the possibility of duplication in the data and deduplicate them.**	★★	☐
17	**Clean the sensitive information** (such as data desensitization and anonymization) **unless the benchmark is deliberately designed so.**	★★	☐

HOW-TO-BENCH (2/4)			
	Phase 2. Benchmark Construction (2 / 2)	**Priority**	☑
18	**Manually review some or all of the data in the benchmark to ensure its quality.**	★★	☐
19	**Use LLMs to review some or all of the data in the benchmark to ensure its quality.**	★	☐
20	**Design appropriate output validation methods for the benchmark** (e.g., using exact matching or designing test cases).	★★★	☐
21	**Design appropriate evaluation metrics for the evaluation set** (e.g., precision, accuracy, pass@K, recall).	★★★	☐
22	**Consider the adequacy of the evaluation metrics** (e.g., is the code coverage high enough).	★★★	☐
23	**Consider if there are any other evaluation perspectives** (e.g., readability, efficiency, safety, security).	★	☐
	Phase 3. Benchmark Evaluation	**Priority**	☑
24	**Consider whether sufficient LLMs are evaluated.**	★	☐
25	**Consider whether representative LLMs** (e.g., covering latest/classical LLM families, small/large LLMs, and open-/closed-source LLMs) **are evaluated.**	★	☐
26	**Consider whether the prompt is of high quality** (e.g., the instruction and intent are clear).	★★★	☐
27	**The prompts have been validated by humans or LLMs** (e.g., evaluated or discussed by participants or preliminarily tried out on several LLMs).	★	☐
28	**Try different paraphrases of the prompt.**	★	☐
29	**Try different prompting strategies to observe the impact on the evaluation results** (e.g., in-context learning, chain-of-thought).	★★	☐
30	**Pay attention to the hardware environment** (such as GPU card, storage size, etc.) of the experiment.	★★★	☐
31	**Pay attention to the operating system and software environment** (e.g. operating system, version, etc.) **used for the experiment.**	★★★	☐
32	**Pay attention to the off-the-shelf platforms, frameworks, or libraries for LLM evaluation** (e.g., fast chat, vllm, huggingface) **that are used.**	★★	☐
33	**Repeat the experiment multiple times to reduce the impact of randomness on the evaluation.**	★	☐
34	**Consider various randomization strategies** (e.g., trying various temperature parameters) **to reduce the impact of parameter configuration on the evaluation.**	★★	☐
35	**Record the experimental process in detail** (e.g., parameter settings, running time, input/output pairs, etc.).	★★★	☐

HOW-TO-BENCH (3/4)			
	Phase 4. Benchmark Analysis	**Priority**	☑
36	**Observe the difficulty of the benchmark, checking if the benchmark is too hard or too easy for LLMs** (i.e., most LLMs score too high/low).	★★	☐
37	**Consider whether the benchmark can distinguish the pros and cons of different LLMs.**	★	☐
38	**If the experiment is repeated several times, consider the stability of the benchmark** (i.e., whether the experimental results vary too much in the repeated experiments).	★	☐
39	**Analyze the correlation between the data and their score**. For example, if there is a correlation between the data (such as similar difficulty and knowledge required), then the scores should also be correlated.	★★	☐
40	Compare the performance of LLMs on this benchmark with their performance on **other related benchmarks**.	★	☐
41	**Consider presenting the experiment results in an appropriate way** (e.g., table, line graph, pie chart, etc.).	★★★	☐
42	**Consider presenting the experiment results clearly** (e.g., distinguishable colors/labels/shapes, etc.).	★★★	☐
43	**Explain the experiment results**.	★★★	☐
44	**Observe correlations via multiple perspectives from the experimental results** (e.g., performance is correlated with model size or amount of context).	★	☐

HOW-TO-BENCH (4/4)			
	Phase 5. Benchmark Release	**Priority**	☑
45	**The analysis of the evaluation results will be inspiring** (e.g., shed light on future direction, make actionable advice, etc.).	★	☐
46	**Set the appropriate license for the benchmark.**	★★★	☐
47	**Review the released benchmark or other artifacts to ensure they do NOT contain sensitive information** (e.g., API keys, usernames, passwords, etc.).	★★★	☐
48	**review the released benchmark or other artifacts to ensure they do NOT contain toxicity information** (e.g., abusive comments/identifiers).	★★★	☐
49	**Make sure the benchmark is open-accessible.**	★★★	☐
50	**Make sure the test cases or reference data are open and accessible.**	★★★	☐
51	**Provide prompts used in the experiment to ensure the experiments are reproducible.**	★★★	☐
52	**Disclose the experimental environment** (e.g., hardware, operating system, software version, framework platform) **to ensure the reproducibility of the experiment.**	★★★	☐
53	**Make the detailed experimental results public for verification.**	★★★	☐
54	**Ensure the quality of the user manual** such as README (e.g., it contains necessary benchmark introduction, executable scripts, etc.).	★★	☐
55	**Provide convenient evaluation interfaces for the released benchmark** (e.g., providing a command line interface, docker, etc.).	★★	☐