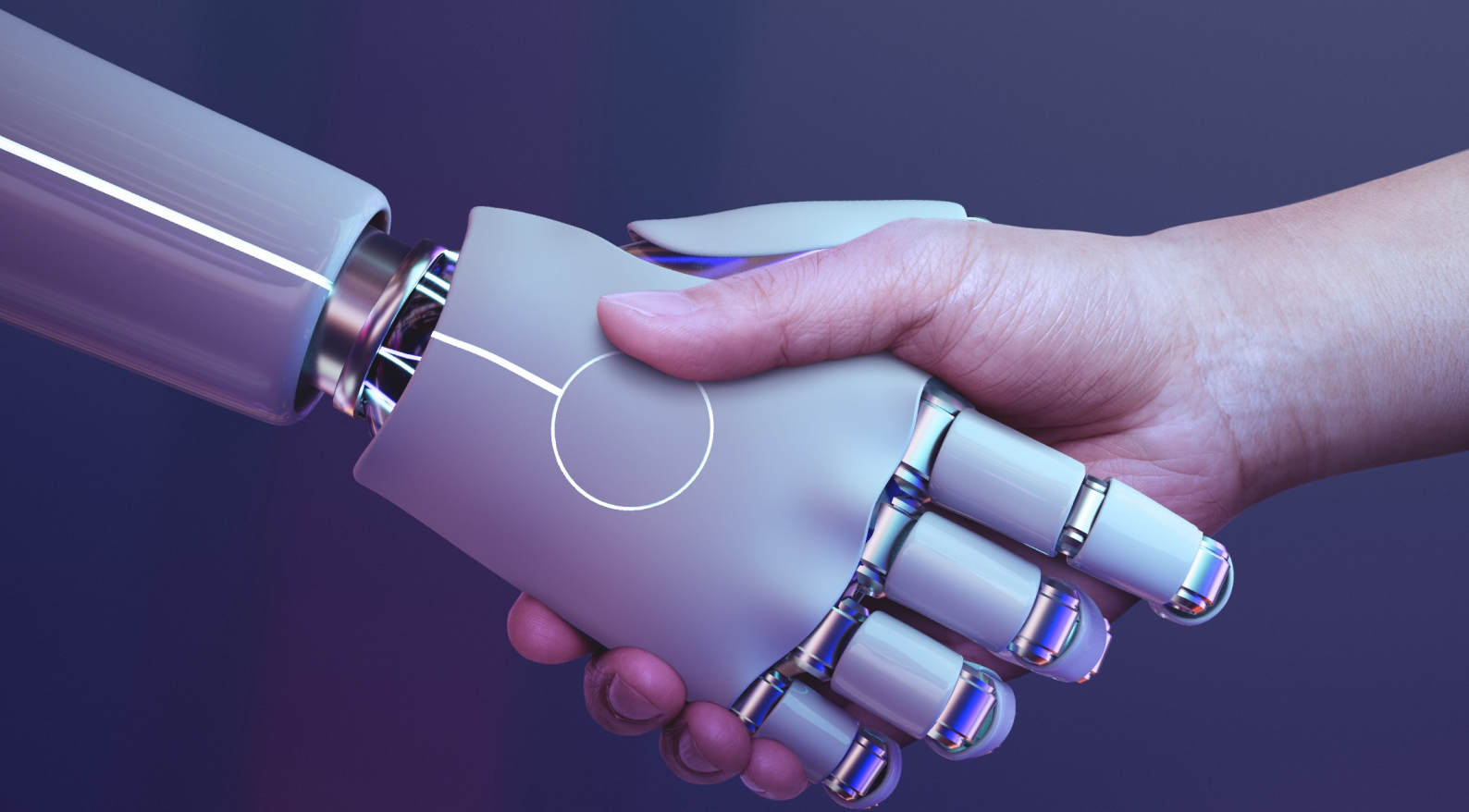


AI for Software Engineering

Reviewing and Improving Benchmarking Practices

Philippe de Bekker

MSc Thesis — Computer Science — Software Technology



AI FOR SOFTWARE ENGINEERING (AI4SE):
REVIEWING AND IMPROVING BENCHMARKING
PRACTICES

A thesis submitted to the Delft University of Technology in partial fulfillment
of the requirements for the degree of

Master of Science

by

Philippe de Bekker

10 July 2024

© Philippe de Bekker. All rights reserved. Contact me for inquiries about copying or redistributing this work.

The work in this thesis was made under guidance and collaboration with:



Software Engineering Research Group
Computer Science - Software Technology
Faculty of Electrical Engineering, Mathematics & Computer Science
Delft University of Technology

Thesis advisor: Prof. Dr. Arie van Deursen
Daily supervisor: Dr. Maliheh Izadi
Co-reader: Dr. Maria Soledad Pera

ABSTRACT

Artificial Intelligence (AI) has rapidly advanced, significantly impacting software engineering through AI-driven tools like ChatGPT and Copilot. These tools, which have garnered substantial commercial interest, rely heavily on the performance of their underlying models, assessed via benchmarks. However, the current focus on performance scores has often overshadowed the quality and rigor of these benchmarks, as emphasized by the absence of studies on this topic. This thesis addresses this gap by reviewing and improving benchmarking practices in the field of AI for software engineering (AI4SE).

First, a categorized overview and analysis of nearly a hundred prominent AI4SE benchmarks from the past decade are provided. Based on this analysis, several challenges and future directions are identified and discussed, including quality control, programming and natural language diversity, task diversity, purpose alignment, and evaluation metrics. Lastly, a significant contribution of this work is the introduction of HUMANEVALPRO, an enhanced version of the original HumanEval benchmark. HUMANEVALPRO incorporates more rigorous test cases and edge cases, providing a more accurate and challenging assessment of model performance. The findings demonstrate substantial drops in pass@1 scores for various large language models, highlighting the necessity for well-maintained and comprehensive benchmarks.

This thesis aims to set a new standard for AI4SE benchmarks, providing a foundation for future research and development in this rapidly evolving field.

PREFACE

This thesis represents a culmination of my journey into the field of Artificial Intelligence (AI) for software engineering (AI4SE). Although AI was not initially my area of expertise, my fascination with its potential drove me to explore and delve deep into this transformative field. Through dedicated coursework and active participation in research projects, I developed a keen interest in the benchmarking practices within AI4SE, particularly after my involvement in a project that resulted in the publication of the HumanEval-Haskell benchmark [81]. This thesis aims to address critical gaps in existing AI4SE benchmarks and proposes improvements to enhance their reliability and effectiveness. A significant contribution of this thesis is the introduction of HUMANEVALPRO, an enhanced version of the original HumanEval benchmark. HUMANEVALPRO includes more rigorous test cases and edge cases, providing a more accurate and challenging assessment of model performance.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and guidance of several individuals.

First and foremost, I would like to express my deepest gratitude to my thesis advisor, Prof. Dr. Arie van Deursen, and my daily supervisor, Dr. Maliheh (Mali) Izadi.

Mali, all your insights, kindness, and unwavering support throughout the process have been invaluable. Your dedication, especially in arranging numerous opportunities to collaborate with companies and other people, has significantly enriched my thesis experience.

Secondly, I extend my gratitude to my co-reader, Dr. Maria Soledad Pera, for her support and most importantly, the highly uplifting energy throughout my study. Unfortunately, PDFs do not support GIFs yet – but a *Thank You* Olaf from Frozen GIF would be suiting!

Furthermore, I would like to acknowledge the AI-enabled Software Engineering (AISE) research lab for the knowledge and fun shared during all meetings, ranging from thesis meetings to the weekly reading club. Your companionship and intellectual discussions have been a big motivation and inspiration.

In addition, I would like to thank Roham Koohestani for his meticulous peer review of the HUMANEVAL-PRO benchmark and Fabio Salerno for co-leading the development of a reasoning benchmark in collaboration with Google DeepMind and other members from AISE.

Lastly, I would like to express my deepest appreciation to my parents, sister, and other close people for their unconditional support throughout this thesis and my entire education. Your encouragement and belief in me have been a constant source of strength.

CONTENTS

1	Introduction	1
2	Current Landscape of AI4SE Benchmarks	3
2.1	General Outline of Collected AI4SE Benchmarks	3
2.2	Detailed Overview of Categorised AI4SE Benchmarks	5
3	A Taxonomy of Challenges and Solution Directions in AI4SE Benchmarks	17
3.1	Quality Control	17
3.2	Preventing Data Contamination	18
3.3	Programming language diversity	19
3.4	Natural Language Diversity	20
3.5	Task Diversity	21
3.6	Purpose Alignment	23
3.7	Evaluation Techniques	24
3.7.1	Evaluation Techniques based on Similarity	24
3.7.2	Evaluation Techniques based on Functional Correctness	28
3.7.3	Recommended Evaluation Techniques for AI4SE Benchmarks	30
4	Towards Better Evaluation of LLMs for Software Engineering: Introducing HUMANEVALPRO	32
4.1	Introduction	33
4.2	Related Work	34
4.3	Approach	34
4.3.1	Standardized Observations in Current HumanEval Benchmarks	35
4.3.2	Modifications in HUMANEVALPRO	37
4.3.3	Peer Review Process of HUMANEVALPRO	38
4.4	Experimental Setup	39
4.4.1	Benchmark Details	39
4.4.2	Models	40
4.4.3	Configuration	41
4.4.4	Post-processing Completions	41
4.4.5	Metrics	42
4.5	Results	44
4.6	Discussion	48
4.7	Conclusion and Future Work	49
5	Summary	50
A	Illustration of Shortcomings in HumanEval	62

1

INTRODUCTION

Artificial Intelligence (AI) has become the word of the year [60], consistently making headlines and revolutionizing various domains, including software engineering. The emergence of AI-driven tools such as ChatGPT¹ and Copilot² has significantly transformed engineering workflows and developed intense commercial interest amongst leading companies such as OpenAI, Google and Meta. As performance, which translates into commercial viability, is measured by benchmarks, achieving the top score has become a battleground where the true quality of benchmarks has been neglected: at the outset of this study, there is a troubling absence of rigorous studies evaluating the quality of benchmarks and the practices in the field.

Among these benchmarks, HumanEval [12] has distinguished itself as the most popular benchmark over the recent years, becoming synonymous with claims of superiority and commercial promise within the field of AI for software engineering (AI4SE). During my previous work of manually translating this benchmark into Haskell [81], many notable problems were discovered, prompting the need for a thorough re-evaluation of both this benchmark and the field of AI4SE benchmarks in general. Subsequently, this thesis aims to address this gap by focusing on the following two objectives and associated research questions:

Research Objective 1

Review and improve benchmarking practices for AI for Software Engineering (AI4SE).

↳ Research Question 1.1

What is the current landscape of AI4SE benchmarks?

↳ Research Question 1.2

What are the challenges and takeaways from the existing pool of AI4SE benchmarks?

Research Objective 2

Establish an enhanced foundation for HumanEval.

¹ <https://chat.openai.com/>

² <https://copilot.github.com/>

By addressing these objectives, this research aims to move towards better evaluation of LLMs for software engineering and contribute to more accurate and reliable tools and practices in the industry.

The structure of this thesis is as follows:

- [Chapter 2](#): Provides a rigorous and extensive survey of existing AI4SE benchmarks, addressing Research Question 1.1.
- [Chapter 3](#): Builds upon the findings of [Chapter 2](#), offering a detailed taxonomy of the challenges in current benchmarks and proposing solution directions, addressing Research Question 1.2. Together with [Chapter 2](#), [Chapter 3](#) addresses Research Objective 1.
- [Chapter 4](#): Introduces HUMANEVALPRO, a reusable, full-fledged solution designed to enhance the foundation of the HumanEval benchmark, addressing Research Objective 2.

Finally, the thesis concludes with a summary in [Chapter 5](#).

2 | CURRENT LANDSCAPE OF AI4SE BENCHMARKS

This chapter outlines the existing pool of notable AI4SE benchmarks. Based upon researching these benchmarks, an analysis can be made as to whether AI4SE benchmarks truly provide meaningful assessments or what exactly is being signified when researchers train models to excel on these benchmarks. Furthermore, by identifying the current challenges in the AI4SE benchmarks, directions can be formed to improve the AI4SE benchmarks in the future (Chapter 3).

2.1 GENERAL OUTLINE OF COLLECTED AI4SE BENCHMARKS

The AI4SE benchmarks considered in this paper are the most relevant, popular, recent and enhanced benchmarks from the last decade (2014 – 2024), as depicted in Figure 2.2, based on the search process shown in Figure 2.1. For each benchmark, the following details were collected (if applicable):

- **General information:** DOI (Digital Object Identifier) paper, date, author affiliations.
- **Details:** task description, prediction goal and provided context, number of problems, number and type of tests, (programming and natural) languages present in the benchmark.
- **Discussion:** bugs, shortcomings and other reviews.
- **Data:** accessibility/reproducibility, reference(s) to stored data (e.g. GitHub [23], Hugging Face [33] and Google Drive [24]), sources regarding the data (e.g. data originates from StackOverflow [79]).

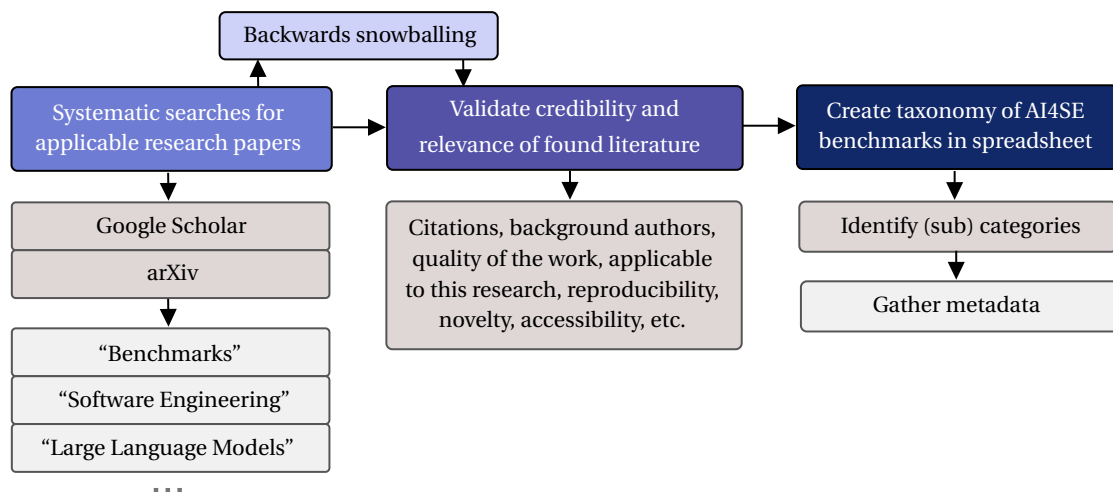


Figure 2.1: Schematic representation of the search process behind the collection of AI4SE benchmarks.

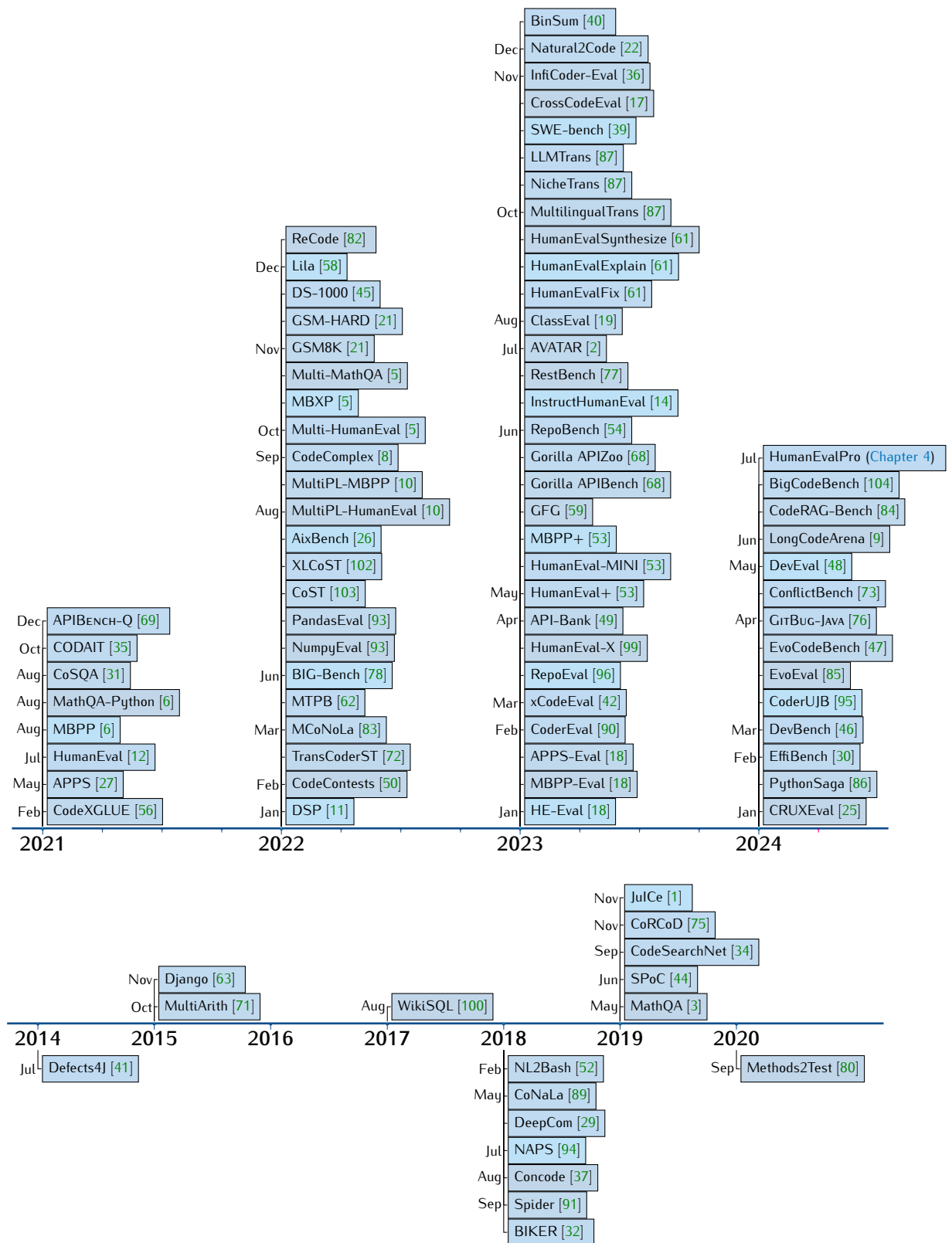


Figure 2.2: Timeline of notable AI4SE benchmarks from 2014 to mid 2024.

2.2 DETAILED OVERVIEW OF CATEGORISED AI4SE BENCHMARKS

Currently, one of the most popular AI4SE benchmarks is HumanEval [12], used to evaluate the performance of many notable models with software engineering capabilities (e.g. Codex [12], Gemini [22] and GPT-4 [64]). This benchmark is used mainly for code synthesis, though there also exist some variations for code repair and code explanation [61]. The family of HumanEval benchmarks is depicted in Table 2.1. After an in-depth analysis of these benchmarks, which can be found in Appendix A, it becomes clear that this family of benchmarks suffers from the following issues: incorrect tests, lack of proper test coverage, incorrect canonical solutions, and imprecise problem definitions. While there are versions that have improved the language support [5, 10, 61, 99] and test coverage [18, 53], there is no version that contains all the improvements combined nor fixed the original issues. The issues for enhancing the original dataset can be generalised as follows:

- Variants that cover multiple languages have duplicated the original issues.
- Variants that added tests used the original incorrect solutions to generate the output.
- Variants based on human corrections or translations are inconsistent.

Furthermore, prompting production systems, such as ChatGPT-3.5, a detail of an incorrect problem in the original HumanEval benchmark yields the exact same mistake in the response (see Figure A.2), revealing a high likelihood of production systems being contaminated with the data of this benchmark, albeit in a near-identical form, potentially due to its high popularity, rendering the benchmark outdated.

Another AI4SE benchmark, highly similar in style and popularity compared to HumanEval, is MBPP [6]: Mostly Basic Python Problems. It contains nearly a thousand crowdsourced problems, where almost half of it is sanitized and separately released. Furthermore, several enhancements have been published for MBPP, see Table 2.2. Upon a more in-depth analysis of MBPP and its family of benchmarks, there are many signs suggesting deficient quality. One notable problem is the lack of proper testing, as MBPP originally only has three (rather trivial) tests per problem – which are all revealed in the prompt as well. With such a test suite in place, evaluation metrics become unstable and insignificant for proper comparison. The strength of the written tests and solutions themselves is not only troublesome in the original data but also the sanitized data features many flaws (even in *corrected* variants [53]). From negligible observations such as poor syntax (e.g. too many spaces, Python method names starting with a capital – this is a common convention to only use for classes) to uncaught bugs and edge cases that break the implementation. While there are enhancements that improve the language support and extend the test cases, they are all built upon inadequate foundations, which renders any MBPP benchmark suboptimal for properly assessing the capabilities of and between AI4SE models. In addition, a pattern of problems is starting to emerge for AI4SE benchmarks based on the in-depth inspection of HumanEval and MBPP. Combined with the inspection of other benchmarks in this chapter, these observed issues form the basis for the challenges and takeaways for AI4SE benchmarks, outlined in Chapter 3.

Table 2.1: Overview of AI4SE benchmarks stemming from HumanEval [12]. Note, the *# Tests* column denotes the number of tests included in the respective benchmarks. This can either be an average per problem, or a scale compared to the complete original HumanEval benchmark.

Category	Name	Language(s)	# Tests	Task	Data
Original	HumanEval [12]	Python	Avg. 7.7	Code Synthesis	🔄, 😊
Improved Language Support	MultiPL-HumanEval [10]	Bash, C++, C#, D, Go, Java, JavaScript, Julia, Lua, Perl, PHP, R, Racket, Ruby, Rust, Scala, Swift, TypeScript	Avg. 7.7	Code Synthesis	🔄, 😊
	HumanEval-Fix [61]	Python, JavaScript, Java, Go, C++, Rust	Avg. 7.7	Code Repair	😊
	HumanEval-Explain [61]	Python, JavaScript, Java, Go, C++, Rust	Avg. 7.7	Code Explanation	😊
	HumanEval-Synthesize [61]	Python, JavaScript, Java, Go, C++, Rust	Avg. 7.7	Code Synthesis	😊
	HumanEval-X [99]	Python, C++, Java, JavaScript, Go	Avg. 7.7	Code Synthesis	🔄, 😊
	Multi-HumanEval [5]	C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Ruby, Scala, Swift, TypeScript, Python	Avg. 7.7	Code Synthesis	🔄
Improved Testing	HumanEval+ [53]	Python	Scaled ×80	Code Synthesis	🔄
	HumanEval-MINI [53]	Python	Scaled ×47	Code Synthesis	🔄
	HE-Eval [18]	Python	Scaled ×14	Code Synthesis	🔄
Instruction-based	InstructHumanEval [14]	Python	Avg. 7.7	Code Synthesis	😊
Extended	EvoEval [85]	Python	Avg. 773.2 (<i>subtle</i>), Avg. 49.2 (<i>difficult</i>), Avg. 43.1 (<i>creative</i>), Avg. 51.8 (<i>combine</i>), Avg. 51.3 (<i>tool_use</i>)	Code Synthesis	🔄, 😊 (Only 100 problems per category released currently)

Table 2.2: Overview of AI4SE benchmarks stemming from MBPP [6].

Category	Name Version	Language(s)	# Problems # Tests per problem	Task	Data
Original	<u>MBPP [6]</u> Original	Python	974 3 (exposed in prompt)	Code Generation	🔄, 😊
Improved Language Support	<u>MultiPL- MBPP [10]</u> Sanitized	Bash, C++, C#, D, Go, Java, JavaScript, Julia, Lua, Perl, PHP, R, Racket, Ruby, Rust, Scala, Swift, TypeScript	382 (Bash), 397 (C++), 386 (C#), 358 (D), 374 (Go), 386 (Java), 390 (Julia), 397 (JavaScript), 397 (Lua), 397 (PHP), 396 (Perl), 397 (Python), 397 (R), 397 (Ruby), 397 (Racket), 354 (Rust), 396 (Scala), 396 (Swift), 390 (TypeScript) 3.1 on avg. (tests can be hidden or exposed in prompt, configurable doctests)	Code Generation	🔄, 😊
	<u>MBXP [5]</u> Original	C++, C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, TypeScript	848 (C++), 968 (C#), 939 (Go), 966 (Java), 966 (JavaScript), 966 (Kotlin), 966 (Perl), 966 (PHP), 974 (Python), 966 (Ruby), 966 (Scala), 966 (Swift), 968 (TypeScript) 3 (exposed in prompt)	Code Generation	🔄, 😊
Improved Testing	<u>MBPP+ [53]</u> Sanitized	Python	427 Scaled ×35 (for avg. of 3.1)	Code Generation	🔄
	<u>MBPP- Eval [18]</u> Original	Python	974 101.7 on avg. (hidden in prompt)	Code Generation	🔄

Besides HumanEval and MBPP, the standardized benchmarks for code synthesis evaluation, there are many more considerable benchmarks for assessing various categories of tasks within the field of software engineering. As a guide for finding specific AI4SE benchmarks, several additional categorized tables have been listed below, highlighting the most notable benchmarks for each in detail. [Table 2.3](#) features benchmarks with competitive programming as their root, benchmarks used for understanding code complexity and efficiency, and benchmarks related to data science. To assess the mathematical reasoning capabilities of AI4SE models, see [Table 2.5](#). Besides numbers and code, natural language is also a key component in AI4SE. From supporting instruction-tuned AI4SE models, which align more with the human brain [7], that aim to accomplish question and answering (QA) similar to the widely recognized platform StackOverflow, to summarizing code and generating tags, [Table 2.6](#) and [Table 2.7](#) feature AI4SE benchmarks including natural language: text-to-code, code-to-text and text-to-text (code related).

Table 2.3: Overview of various AI4SE benchmarks categories: competitive programming, code complexity and code efficiency.

Category	Name	Language(s)	# Tests	Comment	Data
Competitive Programming	CodeContests [50]	C++, C#, Go, Java, JavaScript, Lua, PHP, Python, Ruby, Rust, Scala, TypeScript	Avg. 203.7	13,610 problems	🔄, 😊
	APPS [27]	Python	Avg. 13.2	10,000 problems	🔄, 😊
Code Complexity	CoRCoD [75]	Java	932	Time: $O(1), O(\log n), O(n), O(n \log n), O(n^2)$	🔄
	GeeksForGeeks (GFG) [59]	C++, Python	$\pm 1,400$ / language & categ.	Time: $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3)$, NP-HARD Space: $\uparrow \setminus O(n^3)$	Reproducible via appendix
	CODAIT [35]	Python	4,000,000	sub-polynomial, polynomial, above-polynomial	Private
	CodeComplex [8]	Java, Python	4,900 / language	Time: $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3)$, NP-HARD	🔄, 😊
	PythonSaga [86]	Python	?	185 prompts with balanced code complexity, spanning 38 programming concepts	Unreleased
Code Efficiency	EffiBench [30]	Python	Self-defined, default avg. 100	1,000 Leetcode efficiency-critical problems	😊

Table 2.4: Overview AI4SE benchmarks beneficial for data science capabilities.

Name	Language(s)	# Tests	Comment	Data
DS-1000 [45]	Python	Avg. 1.6	NumPy, Pandas, Pytorch, Scipy, Sklearn, Tensorflow, Matplotlib	🔄, 😊
NumpyEval [93]	Python	Avg. 20 functions Avg. 1 variables	NumPy (101 problems)	🔄
PandasEval [93]	Python	Avg. 20 functions Avg. 1 variables	Pandas (101 problems)	🔄
JuICe [1]	Python, Jupyter Notebooks	✗	Cell completion (1.5M training & 3.7K test samples)	🔄
DSP [11]	Python, Jupyter Notebooks	✓	Cell completion (1,119 problems)	Cannot load data (🔄)
BIG-bench [78] <i>list_functions</i>	Numeric, JSON, Python	250	Infer and compute functions over lists of natural numbers	🔄

Table 2.5: Overview of **math-related** benchmarks useful for AI4SE. For more datasets, not necessarily bound to programming or core math, please refer to the survey by Lu et al. [55].

Category	Name	Language(s)	# Problems	Data
	MathQA [3]	English	37,297	😊
	MathQA-Python [6]	Python	23,914	Reproducible (🔄)
	MathQA-X [5]	Python, Java, Javascript	1,883 / language	🔄, 😊
Mathematical Reasoning	LILA [58]	Python	133,815 questions 358,769 programs	🔄
	MultiArith [71]	English	600	😊
	GSM8K [21]	English	1,320	🔄
	GSM-HARD [21]	English	1,320	🔄
	TheoremQA [13]	English	800	🔄, 😊
	BIG-bench [78] <i>see 'math' keyword</i>	Python	Various tasks	🔄

Table 2.6: Overview of AI4SE benchmarks specifically focused on the inclusion of **natural language** (part I).

Category	Name	Language(s)	No. of Problems	Tests	Task	Data
Text2Code	CoNaLa [89]	English → Python	2,879	Unit tests ✗	Stack Overflow Q&A	😊
	MCoNaLa [83]	Spanish, Japanese, Russian → Python	896 (341 + 210 + 345)	Unit tests ✗	Stack Overflow Q&A	🔄, 😊
	APPS [27]	English → Python	10,000	Avg. 13.2	Code Generation	🔄, 😊
	APPS-Eval [18]	English → Python	10,000	Avg. 181	Code Generation	🔄
	AixBench [26]	English, Chinese → Java	175	✓ / ✗ (paper claims unit tests)	Method Generation	🔄 (tests not listed in /resources)
	Natural2Code [22]	English → Python	?	✓	Code Generation	Private
	CoSQA [31]	English → Python	20,604	Unit tests ✗	Code Search, Q&A	🔄
	WebQueryTest [56]	English → Python	1,046	Unit tests ✗	Code Search, Q&A	🔄
	AdvTest [56]	English → Python	280,634	Unit tests ✗	Code Search	🔄
	CONCODE [37]	English → Java	104,000	Unit tests ✗	Generate Class Member Functions	🔄
	XLCoST [102]	English → C, C++, C#, Java, JavaScript, PHP, Python	509K & 58K	Unit tests ✗	Snippet & Program Synthesis + Code Search	🔄
	MTPB [62]	English → Python	115	Avg. 5	Multi-step Code Generation	Reproducible from their Appendix D
	Text2Text (about code)	xCodeEval [42]	English → C, C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust	5,539,899	✓	Code Generation
BIG-bench [78] <small>programming_challenge</small>		English → Python	7 (very easy), 14 (easy), 14 (medium), 7 (hard)	Labeled Pairs ✓	Code Generation	🔄
InfiCoder-Eval [36]		English → English (featuring: Bash, C, C++, C#, CSS, Dart, Go, HTML, Java, JavaScript, Kotlin, PHP, Python, R, Ruby, Rust, Swift, VBA)	270	Labeled Pairs ✓ (50% not released currently)	Free-form Q&A ability, e.g. Code Completion, Knowledge Q&A, Code Debugging	🔄
CodeXGLUE [56] / MicrosoftDocs [57]		Chinese, Norwegian, Danish, Latvian ↔ English	52K (Chinese), 46K (Norwegian), 45K (Danish), 21K (Latvian)	Labeled Pairs ✓	Code Document Translation	🔄

Table 2.7: Overview of AI4SE benchmarks specifically focused on the inclusion of **natural language** (part II).

Category	Name	Language(s)	No. of Problems	Tests	Task	Data
Code2Text	CodeXGLUE [56]/ CodeSearchNet [34]	Go, Java, JavaScript, PHP, Python, Ruby → English	280,652 (Go), 180,253 (Java), 65,201 (JS), 39,588 (PHP), 28,588 (Python), 27,588 (Ruby)	Labeled Pairs ✓	Code Summarization, Comment Generation	🔄
	DeepCom [29]	Java → English	588,108	Labeled Pairs ✓	Code Summarization	🔄
	BinSum [40]	Binary functions (x64, x86, ARM, MIPS) → English	557,664	Labeled Pairs ✓	Binary Code Summarization	🔄
	XLCoST [102]	C, C++, C#, Java, JavaScript, PHP, Python → English	509K & 58K	Labeled Pairs ✓	Snippet & Program Summarization	🔄
	xCodeEval [42]	C, C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust → English	5,587,437	Labeled Pairs ✓	Tag Classification	🔄, 😊
	BIG-bench [78] <small>code_line_description</small>	Python → English	60	Labeled Pairs ✓	Code line description	🔄
	Long Code Arena [9]	Python → English	163	Labeled Pairs ✓	Commit message generation	😊
	Long Code Arena [9]	Python → English	216	Labeled Pairs ✓	Module sum- marization	😊

While translating natural language is more trivial nowadays, translating code remains challenging due to various reasons (e.g. versioning, semantics, dependencies). With the lack of diversity in language support for AI4SE benchmarks and also benefiting numerous other SE tasks, Table 2.8 features an overview of resources that can support the ongoing development of code translation.

Table 2.8: Overview of useful resources regarding **code translation** for AI4SE benchmarks.

Category	Name	Language(s)	# Samples	Avg. # Chars	Data
Programming Languages	CodeTrans [56]	C#, Java	11,800	±205	🔄
	TransCoder-ST [72]	C++, Java, Python	437,030	–	Reproduction Package (🔄)
	CoST [103]	C, C++, C#, Java, JavaScript, PHP, Python	16,738	±600	🔄
	XLCoST [102]	C, C++, C#, Java, JavaScript, PHP, Python	122,151	±640	🔄
	AVATAR [2]	Java, Python	7.133 (Train), 476 (Dev), 1,906 (Test) (note: 3,391 parallel functions and 250 unit tests)	±689	🔄
	Multilingual-Trans [87]	C, C++, C#, Go, Java, PHP, Python, Visual Basic	19,115 (Train), 3,759 (Dev), 7,545 (Test)	1,099 (Train), 1,135 (Dev), 1,358 (Test)	🔄
	NicheTrans [87]	Ada, Arturo, AutoHotKey, AWK, BBC Basic, Clojure, COBOL, Common Lisp, D, Delphi, Elixir, Erlang, Factor, F#, Forth, Fortran, Groovy, Haskell, Icon, J, Julia, Lua, Mathematica, MATLAB, Nim, OCaml, Pascal, Perl, PowerShell, R, Racket, Ruby, Rust, Scala, Swift, Tcl	165,457 (Train), 23,509 (Dev), 47,502 (Test)	785 (Train), 995 (Dev), 1,372 (Test)	🔄
	LLMTrans [87]	C, C++, C#, Go, Java, PHP, Python, Visual Basic	350	745	🔄
	xCodeEval [42]	C, C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust	5,538,841 (Train), 7,474 (Dev), 20,356 (Test)	–	🔄, 😊
	Libraries	DLTrans [87]	PyTorch, TensorFlow, MXNet, Paddle	282 (Train), 36 (Dev), 90 (Test)	1,318 (Train), 2,441 (Dev), 1,841 (Test)
Language Conversion Frameworks	MultiPLE [10]	Bash, C++, C#, D, Go, Java, JavaScript, Julia, Lua, Perl, PHP, Python, R, Racket, Ruby, Rust, Scala, Swift, TypeScript	–	–	🔄
	MultiEval [5]	C++, C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, TypeScript	–	–	🔄

Notably, many benchmarks do not reflect real-world code usage scenarios. Hence, several benchmarks catering to these scenarios (e.g. featuring a wider context) are denoted in Table 2.9.

Table 2.9: Overview of AI4SE benchmarks simulating more real-to-life scenarios. For example, by providing wider context than standalone functions or actual project issues instead of illustrative tasks.

Name	Language(s)	No. of Problems	Tests	Task	Data
ClassEval [19]	Python	100	Avg. 33.1	Class-level code generation	🔄, 😊
CrossCodeEval [17]	C#, TypeScript, Java, Python	2,665 (Python), 2,139 (Java), 3,356 (TypeScript), 1,768 (C#)	✗	Cross-file contextual code completion	🔄
CoderEval [90]	Java, Python	230 (Java), 230 (Python)	✓ (source repositories)	Context-dependent code generation	🔄
SWE-bench [39]	Python	19,008 (Train), 225 (Dev), 2,294 (Test)	Avg. resolved tests: 20.9 (Dev), 9.1 (Test) Avg. regression tests: 86.3 (Dev), 111.7 (Test) Median resolved tests: 2 (Dev), 1 (Test), Median regression tests: 42 (Dev), 51 (Test)	Solve pull request (PR) of a GitHub project, unit test verification using post-PR behaviour as the reference solution	🔄, 😊
RepoBench [54]	Python, Java	Cross-file: 8,033 (<i>first</i>), 7,618 (<i>random</i>) In-file: 7,910 Median tokens: For each, ±10K	✗	Repository-level context retrieval and code completion	🔄, 😊
RepoEval [96]	Python	1,600 (<i>line</i>), 1,600 (<i>API</i>), 373 (<i>function</i>)	Line, API: ✗ Function: ✓ (source repositories)	Repository-level code completion, various granularities: line, function & API invocation completion	🔄
CONCODE [37]	English, Java	104,000	✗	Class-level code generation	🔄
CoderUJB [95]	Java	2,239	✓	Based on 17 real projects: code generation (function), code-based and issue-based test generation, defect detection, automated program repair	🔄
EvoCodeBench [47]	Python	275	✓	Based on 25 repositories: code generation	🔄, 😊
BigCodeBench [104]	Python	1,140	Avg. 5.6	Code generation with function calls and complex instructions	🔄, 😊
DevEval [48]	Python	1,874	✓	Repository-level code generation	🔄
DevBench [46]	Python, C/C++, Java, JavaScript	22 repositories	LLM-as-a-judge or Avg. # Unit Tests: Python: 12.4 C/C++: 11.8 Java: 8.2 JavaScript: - (functional correctness is not applicable to pure static web pages)	Software development lifecycle, including software design, environment setup, implementation, acceptance testing, and unit testing	🔄
Long Code Arena [9]	Python	150	✓	Library-based code generation	😊
Long Code Arena [9]	Python	Per context size: Huge: 270 Large: 270 Medium: 224 Small: 144	✓ (≈ completion lines)	Project-level code completion	😊

Table 2.10: Overview of AI4SE resources focusing on leveraging the power of APIs or other external sources.





















Name	Sources/ API(s)	No. of Problems	Tests	Task	Data
RestBench [77]	Spotify, TMDB	57, 100	Golden solution path	Predict API path(s) for realistic user instructions	
APIBENCH-Q [69]	StackOverflow, Tutorial Websites	6,563 (Java), 4,309 (Python)	Corresponding APIs, API classes and source	Predict API for code-related questions	
BIKER [32]	StackOverflow	33,000	Verified subset: 413 pairs	Predict candidate APIs for query	
Gorilla APIBench [68]	HuggingFace, TensorHub, TorchHub	925, 696, 94	10 instruction references per API call	Predict API based on user question prompts	
Gorilla APIZoo [68]	Open submissions, including APIs of Google, YouTube, Zoom and more		Instruction reference per API call	Predict API based on user question prompts	
API-Bank [49]	73 commonly used APIs	753	Chinese instruction reference per API call	Planning, retrieving and calling API tools	
CodeRAG- Bench [84]	Competition solutions, online tutorials, library documentation, StackOverflow, GitHub	25,859	Ground truths available	Basic programming, open-domain, repository-level and code retrieval	

Table 2.11: Overview of AI4SE benchmarks related to pseudocode.

Name	Language(s)	No. of Problems	Tests	Task	Source	Data
SPoC [44]	C++	18,356	Avg. 38.6	Pseudocode to Code	Crowdsourced	
NAPS [94]	Java/UAST	17,477	Avg. 7.5	Pseudocode to Code	Generated	
Django [63]	Python, English & Japanese	18,805 (Train), 1,000 (Dev), 1,805 (Test)	–	Code to Pseudocode	Generated	 , 

Additionally, the utilization of APIs play a significant role in AI4SE benchmarks, specifically for models with Retrieval Augmented Generation (RAG) capabilities. In [Table 2.10](#), prominent benchmarks focusing on leveraging the power of APIs are denoted. Furthermore, [Table 2.11](#) lists benchmarks related to pseudocode, followed by an overview of notable crowd-sourced AI4SE resources in [Table 2.12](#).

Table 2.12: Overview of notable crowd-sourced AI4SE resources.

Name	Language(s)	No. of Problems	Tests	Source	Year	Data
WikiSQL [100]	Natural language → SQL query	80,654	Pairs	Amazon MT	2017	 (deprecated)
Spider [91]	Natural language → SQL query	10,181	5,683	11 Yale students	2018	 , 
NL2Bash [52]	Natural language → Bash	9,305	Pairs	Upwork	2018	
NAPS [94]	Java/UAST → Pseudocode	17,477	Avg. 7.5	Self-hosted crowd-sourcing platform, participants competitive programming community	2018	
SPoC [44]	C++	18,356	Avg. 38.6	Participants of competitive programming websites	2019	
MBPP [6]	Python	974	Avg. 3	Internal pool of crowdworkers, Google Research	2021	 , 
BIG-bench [78]	Python	200+ tasks	Pairs	Open-source contributions via GitHub	2022	

With AI4SE models mainly being utilised for program synthesis, it remains relatively questionable how effective these models are in generating tests and repairing bugs, as it is unclear whether these models *truly* understand code. For example, Siddiq et al. [74] observed Codex [12] being able to get above 80% coverage for HumanEval [12], yet many test smells were discovered and for another dataset, no higher than 2% coverage was attained. This reveals the importance of benchmarking AI4SE models' capabilities in test generation, bug repair and understanding. Below, in Table 2.13, several benchmarks are listed that make an effort to assess the aforementioned.

Table 2.13: Overview of AI4SE benchmarks related to bug repair, test generation and understanding.

Name	Language	No. of Samples	Context	Task	Data
Defects4J [41]	Java	835 (at the time of writing, it is open for extension)	Bug and single commit fix, open-source program, accompanied by a test suite	Automated program repair (APR), fault localisation (FL)	🔄
METHODS2TEST [80]	Java	780,944	Unit tests mapped to methods (also class-level)	Automated unit test case generation	🔄
BIG-bench [78] <small>code_line_description</small>	Python	20 (numeric), 15 (string), 10 (collection), 15 (logical), 6 (type)	List of inputs/outputs	Inductive code synthesis (satisfy I/O relationship)	🔄
BIG-bench [78] <small>auto_debugging</small>	Python	34	Program with question and answer pair(s)	Program state analysis, automatic debugging	🔄
BIG-bench [78] <small>simp_turing_concept</small>	Python	6,390 queries, 426 concepts	Few I/O examples	Turing-complete concept learning	🔄
ConflictBench [73]	Java	180	Conflicting chunk reported by <code>git-merge</code> and merged version for validation	Merge conflict repair	🔄
CRUXEVAL [25]	Python	800	Input-output pair for a short (3-13 lines) function	Test input and output prediction, code understanding	🔄, 😊, 🔗
Long Code Arena [9]	Java, Kotlin, Python	5.04k, 1.24k, 8.68k	Bug issue description, repositories, list of corresponding files	Bug localization	😊
GITBUG-JAVA [76]	Java	199	Repository, validated by subsequent version	APR & FL	🔄
Long Code Arena [9]	Python	78	Logs, commit, workflows, repository snapshot, CI builds repair changed files	CI builds repair	😊

3 | A TAXONOMY OF CHALLENGES AND SOLUTION DIRECTIONS IN AI4SE BENCHMARKS

By researching the existing pool of AI4SE benchmarks in the previous chapter ([Chapter 2](#)), many insights have been gathered on current practices within the field. Unfortunately, not all works follow best practices, or in some cases, no best practices even exist – revealing that there is still much work to be done in the field of AI4SE benchmarks. To adequately shape this evolving landscape, this chapter aims to provide a structured taxonomy of critical challenges and actionable solution directions that are essential for advancing AI4SE benchmarks towards greater reliability and relevance.

3.1 QUALITY CONTROL

Quality control is paramount in ensuring that AI4SE benchmarks accurately measure the performance of models. As observed in several studied benchmarks in [Chapter 2](#), such as HumanEval and the family of benchmarks around it, almost none mention quality control measures and data used for evaluation oftentimes contains significant errors – confirming the lack of actual quality control being implemented. Even beyond the original authors, newer papers extending these benchmarks (with the aim to *enhance* them) also repeatedly lack rigorous quality control; many fail to thoroughly review the source data before adding new layers. While reviewing and correcting benchmark data is labor-intensive, it is essential for maintaining the integrity and usefulness of AI4SE benchmarks.

Some AI4SE engineers argue that errors do not affect model comparisons since all models contend with the same flawed data. However, data accuracy is crucial for several reasons. Firstly, inaccuracies compromise the validity of these comparisons – especially when considering a lack of test coverage. Secondly, incorrect data can lead to misleading conclusions about model performance, setting an inaccurate performance standard. Lastly, reliable benchmarks build trust within the research community, encouraging their adoption for further research and development. To mitigate any issues associated with erroneous benchmark data, recommendable strategies are provided below.

Incorporate a review process in benchmark creation. The review process can be tiered based on the level of scrutiny required, with the following options available (listed in descending order of quality assurance):

- **Independent peer review:** Engage experts with domain knowledge to conduct a thorough review of the benchmark data. This is the highest quality review method but can be resource-intensive.

- **Crowdsourced review:** Utilize platforms such as MTurk¹ or Prolific² to gather feedback and corrections from a large pool of reviewers. This method can easily provide diverse insights and is cost-effective.
- **LLM-based review:** Leverage LLMs to automatically detect and suggest corrections for errors in the data. While not as reliable as human review, LLMs can serve as an initial filter to catch obvious mistakes.

For research insights and transparency within the community, it is imperative to publish any data or details regarding these changes. In current AI4SE benchmarks that do mention quality control, the extent of effort or impact of these measures remains unclear due to the lack of detailed documentation and dissemination of this information.

Release updated versions. While continuously releasing minor updates (e.g. v1.4.7) for a benchmark may be impractical or impossible to maintain, extensively accumulated community feedback (e.g. GitHub pull requests or discussions) can warrant a higher quality version release at some point. Ideally, leaderboards would facilitate more frequent versioned releases of benchmarks, automatically re-running evaluations for all models, as this bottleneck remains the most influential reason for not updating benchmarks. However, this approach is resource-intensive, thus, it remains best to ensure that the original version attains a certified level of quality.

3.2 PREVENTING DATA CONTAMINATION

The observation that smaller models exhibit significant benchmark performance on certain leaderboards prompted a closer investigation into data contamination. Confirming the likelihood of data contamination, the upcoming [Chapter 4](#) shows how some AI4SE models completely drop in performance upon changing the data in a benchmark compared to other model and benchmark results. Ensuring the integrity and reliability of AI4SE models is crucial for producing valid and reproducible results, thus it is important to understand different ways to prevent data contamination. Otherwise, misleading conclusions are made and could eventually render all results of certain benchmarks unreliable. Below, several strategies are denoted that can be employed to mitigate the risk of data contamination or possibly detect contaminated models.

Temporal data splits. Temporal data splitting involves partitioning the dataset based on time. By ensuring that training data precedes testing data chronologically, it is possible to avoid using future information to predict past events, which is unrealistic in real-world scenarios. This method helps maintain a clear distinction between training and evaluation phases, thus preventing leakage of information from the test set into the training set.

Using obfuscated data. Obfuscated data involves modifying the data to retain its original structure and semantics while making it unrecognizable compared to its previous state. Techniques such as shuffling or rephrasing instruction sentences, renaming variables, obscuring code using state-of-the-art obfuscating tools, or even employing LLM-augmented obfuscation can be used. This obfuscation ensures that models do not rely on mistakenly memorized patterns from contaminated data but must

¹ <https://www.mturk.com>

² <https://www.prolific.com>

generalize on “unseen” data. Note, bias may be introduced when models receive differently obfuscated data, yet their performance results are directly compared. Thus, this approach should mainly be used to detect data contamination amongst models.

API-based testing. API-based testing involves evaluating models using a standardized API that provides access to the test set of the benchmark. This approach ensures that the test data remains hidden from the developers and the models being tested. The API returns evaluation metrics without exposing the underlying test data, thus preventing any form of data contamination. However, the reliability of this method depends on the trustworthiness and quality of the API implementation.

Detecting near-duplicates. While this is rather difficult in practice, correct near-duplicate detection could ensure that similar or identical examples do not appear in both training and test sets. By eliminating these instances, the risk of models overfitting to tested patterns is reduced, thereby enhancing the accuracy of performance results.

Opt out. As a final measure, benchmark developers can take proactive steps to ensure their data is explicitly marked as unavailable for scraping from specific websites, such as licenses on GitHub [23], or for inclusion in widely used large-scale data collections like The Stack [43], which are commonly employed in training large language models (LLMs). In addition, marking benchmark data with certain tokens simplifies the identification and exclusion of non-consensual data for model developers that aim to evaluate on a specific benchmark.

In cases of suspected data contamination, comparing trendlines between results from different benchmarks and models can help identify outliers that may exhibit unusually high performance due to contamination.

3.3 PROGRAMMING LANGUAGE DIVERSITY

The rapid evolution of the AI4SE field has resulted in a significant deficiency in programming language diversity within its benchmarks. Upon a careful analysis of the AI4SE benchmarks listed in [Chapter 2](#), a rather substantial imbalance in the distribution of programming languages becomes visible – see [Figure 3.1](#). Almost 80% of the analysed benchmarks support Python and roughly 40% supports Java. The next most popular programming languages are C++, C#, JavaScript, Go and PHP, being supported by roughly 20% of the AI4SE benchmarks. Beyond these languages, support dwindles dramatically, underscoring the urgent necessity for greater programming language diversity in the field.

The importance of supporting a broader range of programming languages in AI4SE benchmarks is twofold. Firstly, the current distribution of programming language support is misaligned with the actual usage patterns across different domains and industries. Secondly, programming languages exhibit significant variations in their use cases and structural paradigms. For instance, functional languages, which are scarcely represented in current benchmarks [81], offer different computational models compared to imperative or object-oriented languages, making AI4SE tools rather incapable in supporting these languages. Similarly, database languages such as SQL, which are vital for data manipulation and querying, also receive inadequate representation.

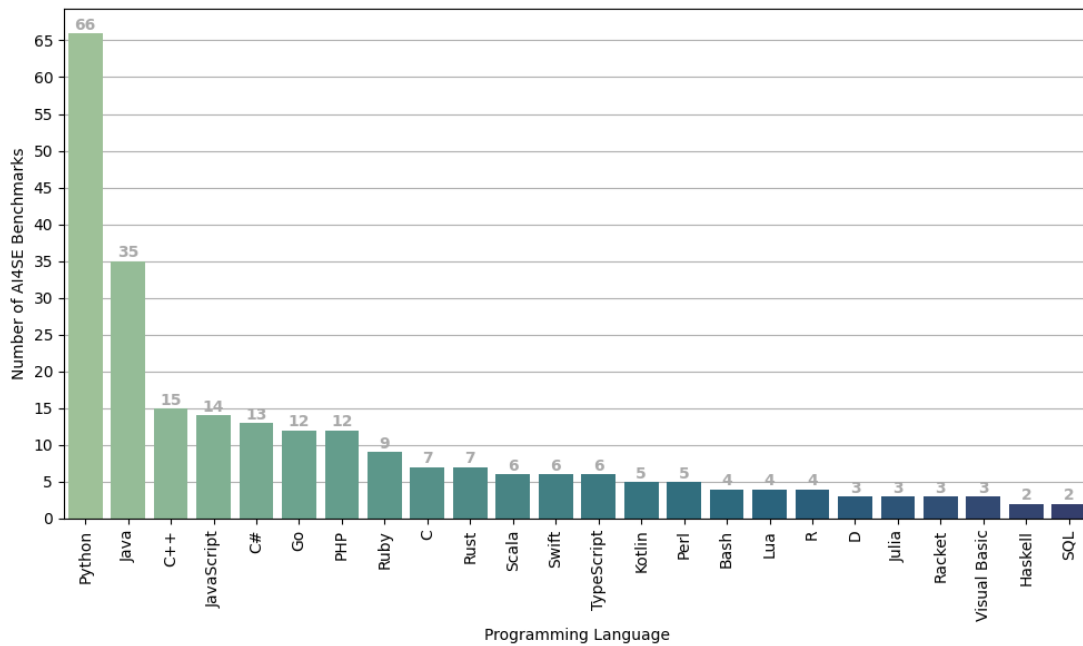


Figure 3.1: Lack of diversity in programming languages based on 84 prominent AI4SE benchmarks from [Chapter 2](#). In this diagram, only languages with 2 or more benchmarks are considered.

All other programming languages not listed in [Figure 3.1](#) though supported, albeit by only one AI4SE benchmark (predominantly NicheTrans [87]), include: AWK, Ada, Arturo, AutoHotKey, BBC Basic, Binary, COBOL, CSS, Clojure, Common Lisp, Dart, Delphi, Elixir, Erlang, F#, Factor, Forth, Fortran, Groovy, HTML, Icon, J, MATLAB, Mathematica, Nim, OCaml, Pascal, PowerShell, Tcl, and UAST.

There are promising language conversion frameworks, as listed in [Table 2.8](#), applicable for some benchmarks. However, ideally, benchmarks should be manually constructed or reviewed for specific languages, as there is a high chance of potential mismatches in the translated problems and common usage of a language.

3.4 NATURAL LANGUAGE DIVERSITY

In addition to the lack of programming language diversity discussed in [Section 3.3](#), the diversity of natural languages in AI4SE benchmarks is even more limited. To summarize, all prominent AI4SE benchmarks are primarily in English. Cassano et al. [10] highlighted in 2022 that MCoNoLa is the only benchmark potentially suitable for evaluating code generation from multiple natural languages, specifically Spanish, Japanese, and Russian. Despite the passage of several years, there has been minimal progress in this area.

Based on the careful analysis in [Chapter 2](#), only a few other benchmarks are revealed which support natural languages other than English (covering at least 95% of renowned AI4SE benchmarks). AixBench [26] includes Chinese for method generation tasks. Additionally, Django [63] supports Japanese

for code-to-pseudocode conversion tasks. For code document translation, the Microsoft Docs benchmark [57] in CodeXGLUE [56] supports Chinese, Norwegian, Danish, and Latvian. Lastly, API-Bank [49] features Chinese instruction references amongst 73 commonly used APIs. Despite these examples, the support for diverse natural languages remains alarmingly low, underscoring the urgent need for expansion in this area to ensure that AI4SE tools are accessible and effective for a global user base.

The inclusion of a broader range of natural languages in AI4SE benchmarks is crucial for several reasons. Firstly, it ensures that the tools and models developed are inclusive and useful to non-English-speaking users, thereby democratizing access to advanced AI capabilities. Secondly, different natural languages present unique challenges in terms of syntax, semantics, and idiomatic usage, which can significantly impact the performance and robustness of AI models. For example, languages with complex grammatical structures or those that are significantly different from English, such as Chinese or Japanese, may reveal different strengths and weaknesses in AI models that are not apparent when evaluating them solely on English tasks.

3.5 TASK DIVERSITY

There are two primary levels of concern regarding the diversity of tasks in the domain of AI4SE. At a macro level, the literature predominantly focuses on a few key tasks: code generation, code completion, code summarization, and program repair [28]. This trend is corroborated by the analysis presented in [Chapter 2](#), which accentuates the need for a broader spectrum of AI4SE benchmarks encompassing a wider array of software engineering tasks. At a micro level, a more granular examination within these benchmarks reveals that the individual tasks, or specific problems, exhibit significant diversity issues themselves.

AI4SE benchmarks frequently possess a generic label describing their scope, such as ‘code generation’ or algorithmic coding tasks. As a result, almost all lack clarity on the diversity of tasks encompassed. This ambiguity poses significant challenges for benchmark users, as it requires high effort to review the benchmark data manually, leading towards benchmark users erroneously trusting the benchmark on their novelty. Consequently, certain benchmarks have become the de facto standard for evaluating domain-specific expertise, without clear indications of what excelling on these benchmarks signifies. For instance, HumanEval is widely used to gauge straightforward coding capabilities of models and was often regarded by large companies as the benchmark for signifying state-of-the-art performance.

Recent work by Yadav and Singh [86] confirms this hypothesis by looking into the difficulties within HumanEval and MBPP. By creating a taxonomy of diverse programming concepts and manually annotating the difficulty and category of tasks within the benchmark, results show roughly 80% of problems belonging to the basic category. Altogether, this work highlights a strong imbalance in difficulty level and task diversity for these benchmarks, illustrating the need for change. Three recommendations to move the field towards trustworthy and balanced benchmarks are delineated below.

Providing an overview of difficulty levels for all problems in the benchmark. Numerous papers present evaluations of their benchmarks using a broad array of LLMs [30, 53], sometimes exceeding 50 models [85]. While this provides an overview of the models’ capabilities, it fails to offer insights into the difficulty levels of individual tasks within the benchmark. If manually annotating theoretical

difficulty is overly labor-intensive, the evaluation of various LLMs offers a new opportunity to easily create a difficulty overview for all benchmark problems: by accumulating the number of models that pass each problem. This can be mathematically defined as follows:

$$S(p_i) = \sum_{k=1}^n \text{pass-metric}(p_i) \quad (3.1)$$

where:

- $S(p_i)$ represents the simplicity score, scaled over all n models, for problem p_i
- $\text{pass-metric}(p_i)$ denotes the $[0, 1]$ score of model k for problem p_i according to a pass metric such as $\text{pass}@k$ or a more granular pass metric that considers the ratio of tests passing for problem p_i .

For a difficulty score, $D(p_i)$, the inverse of $S(p_i)$ could be considered, where $D(p_i) + S(p_i) =$ the total number of evaluated models.

To illustrate this concept, consider the following scenarios. For a balanced benchmark, a wide variety of models with different capability levels should exhibit a clear diagonal line in a bar chart based on sorted values. An illustrative example is shown below in [Figure 3.2](#).

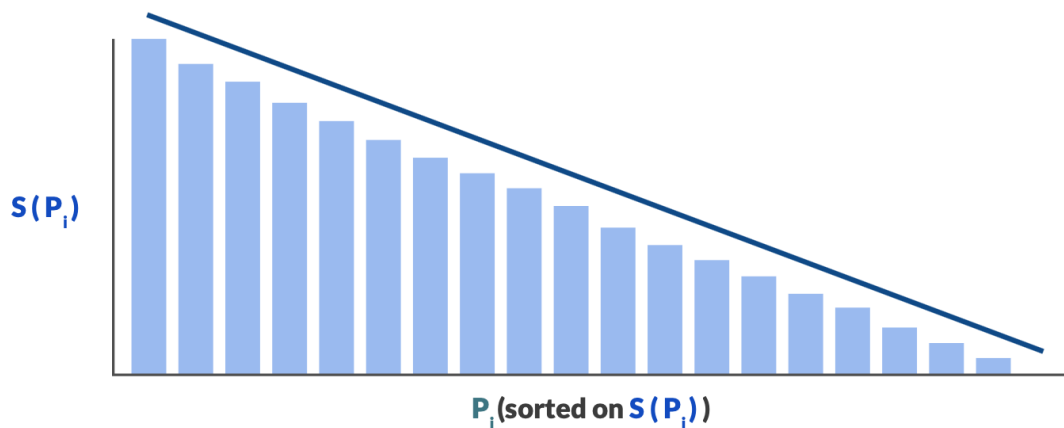


Figure 3.2: Illustration of difficulty levels for a balanced benchmark based on [Equation 3.1](#).

In contrast, for a difficult benchmark, the diagonal line would start off-axis and remain low, indicating that many problems are too complex for most models to solve, even the simplest ones. An illustrative example of this scenario is depicted below in [Figure 3.3](#).

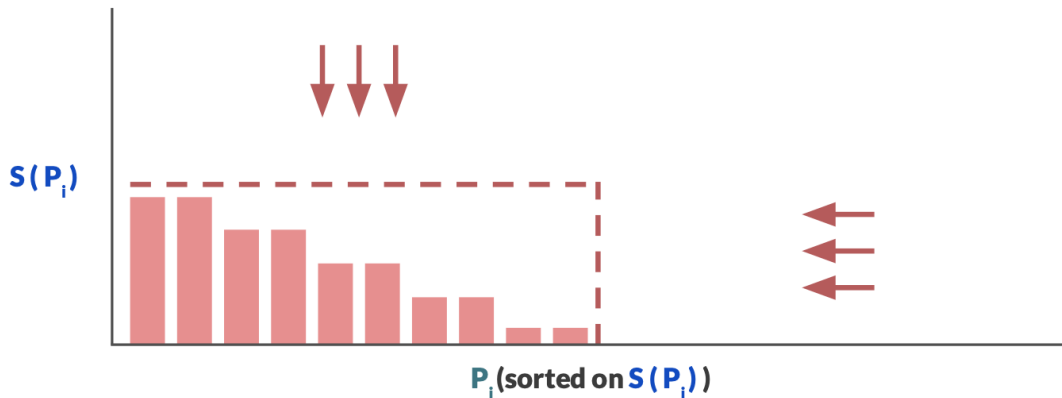


Figure 3.3: Illustration of difficulty levels for a rather challenging balanced benchmark based on Equation 3.1.

Lastly, it could be argued that any relatively non-linear line indicates an imbalance in the benchmark which needs to be carefully regarded, assuming a wide variety of AI4SE models. A real application of the above, based on 10 LLMs, is presented in Figure 4.7.

Providing a clear scope of the benchmark data, for example, via a taxonomy. Consumers of the benchmark should precisely know what it signifies when they excel on the benchmark, thus, a clear representation of the scope of the data of the benchmark should be provided. The most ideal way would be creating a list of (sub)domains (e.g. in the field of software engineering this could be file handling, dynamic programming, divide and conquer, bit manipulation, etc.). These (sub)domains can then be categorized into difficulty levels, such as basic, intermediate, and advanced. Subsequently, a bar chart depicting the number of problems per difficulty level and/or (sub)domain can be drafted, providing a clear overview of the tasks' difficulty and scope. Such an overview also facilitates the creation of more balanced benchmarks by developers.

Comparing features of benchmark data with real-life data. To ensure that a benchmark accurately reflects the diversity of real-world data, various features of the benchmark could be calculated and compared to those of actual data. For instance, comparing the code and dependency distributions of the benchmark to an aggregate of popular and valuable repositories can provide insights into its representativeness. Recent work of Li et al. [47, 48] exemplifies this approach, demonstrating the effectiveness of such comparative analyses in confirming the diversity and applicability of AI4SE benchmarks by providing such an overview.

3.6 PURPOSE ALIGNMENT

The alignment of AI4SE benchmarks with their intended purposes is a critical aspect that warrants careful consideration. This goes hand in hand with the emphasis on scope clarity in Section 3.5. *What (real-world) challenges does my benchmark accurately represent?* Realistic and purpose-aligned benchmarks ensure that the evaluation of AI models is both meaningful and indicative of their performance in real-world scenarios. At the outset of this research, state-of-the-art benchmarks were often mis-

aligned with real-to-life developer scenarios, urgently necessitating a paradigm shift to properly advance AI4SE [47]. However, given the rapid progression in this field, a shift is already underway with new benchmarks increasingly providing the complexity and contextual richness reflective of actual software engineering tasks, or calculating their alignment with real-world code repositories [47, 48].

State-of-the-art AI4SE benchmarks exemplifying this shift include SWE-bench [39] and Long Code Arena [9], offering enormous context to models. Furthermore, with the aim of moving towards Artificial General Intelligence (AGI) [77], benchmarks leveraging APIs are quickly emerging (Table 2.10), allowing rapid advancement in Retrieval-Augmented Generation (RAG) capabilities for AI4SE models. Additionally, the rise of autonomous program improvement has started, with many newer AI4SE models showing increasingly promising results on these larger benchmarks [15, 98].

Despite these advancements in a satisfactory direction, there remains significant future work to be done. The current progress is in its earliest stages, there still is a pressing need for more benchmarks with larger contexts featuring complex software engineering tasks (e.g. non-trivial code that requires cross-file references). Additionally, the integration of multimodality in AI4SE benchmarks is an area that requires substantial development. Effective utilization of multimodality, where models can process and integrate data from various sources such as video, images, and speech, remains one of the critical research gaps in AI4SE benchmarks. Currently, no prominent AI4SE benchmarks incorporate this capability, which limits the comprehensive evaluation of newer AI models with multimodal capabilities (e.g. GPT-4V [65], GPT-4o [66], Gemini [22], Claude 3 [4], and more).

3.7 EVALUATION TECHNIQUES

As with any benchmark, including AI4SE benchmarks, robust metrics are needed to measure the performance, for which it is essential to capture the right aspects. This section will describe the most widely used evaluation techniques regarding code generation, based on similarity (3.7.1) and functional correctness (3.7.2), along with a discussion of their respective advantages and disadvantages, to determine which techniques can be considered as best practices in Section 3.7.3. Additionally, providing a more fine-grained perspective, a new metric for assessing the functional correctness of code is introduced: `latest-pass@n`.

3.7.1 Evaluation Techniques based on Similarity

In earlier research, evaluation metrics were mainly focused on similarity. In some cases, these metrics are still valuable to include in your evaluation. Below, the most prevalent ones are listed.

Exact Match (EM). This metric, also referred to as (perfect) ‘accuracy’ at times, measures the percentage of generated code sequences that *exactly* match the reference code sequences. Formally, let \hat{y}_i be the generated code and y_i the reference code for instance i , then EM is defined as:

$$\text{EM} = \frac{1}{N} \sum_{i=1}^N \mathbf{1}(\hat{y}_i = y_i) \quad (3.2)$$

where N is the total number of instances, and $\mathbf{1}(\cdot)$ is the indicator function.

Advantages:

- Simple to compute.
- Easily interpretable: a high EM score can be thought of as having high precision.
- Directly measures exact correctness.

Disadvantages:

- Stringent criterion; does not account for minor acceptable variations.
- Sensitive to superficial differences like variable naming and formatting.

Edit Similarity (ES). A more granular variant of EM: ES evaluates the similarity between generated and reference code by computing the edit distance, which is the minimum number of operations (insertions, deletions, substitutions) required to transform one sequence into the other. The normalized edit similarity, hereafter referred to as ES, is given by:

$$ES = 1 - \frac{d_{\text{edit}}(\hat{y}, y)}{\max(|\hat{y}|, |y|)} \quad (3.3)$$

where $d_{\text{edit}}(\hat{y}, y)$ is the edit distance (Levenshtein), and $|\cdot|$ denotes the length of the sequence.

Advantages:

- Captures subtle differences.
- Less sensitive to small, irrelevant changes.

Disadvantages:

- May not correlate with functional correctness.
- Computationally expensive for long sequences.

BLEU. The **Bi**Lingual **E**valuation **U**nderstudy (BLEU) score is a precision-based metric, originally designed for machine translation, adapted for code generation [67]. For context, the definition of an n -gram is needed first: a contiguous sequence of n items from a given sample of text or code – see Figure 3.4 for an example.

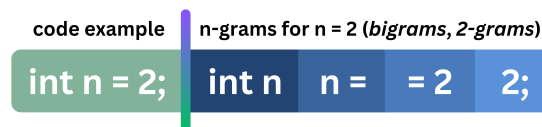


Figure 3.4: Illustrative example regarding the definition of n-grams.

BLEU computes the geometric mean of the precision of n -grams with a length penalty to favour longer sequences (since it could be that the generated code contains all the n -grams of the solution, but does not match the repeated n -grams). The metric is defined as follows:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^N w_n \log p_n\right) \quad (3.4)$$

where BP is the brevity penalty, p_n is the precision of n -grams, and w_n are positive weights summing to 1.

Advantages:

- Well-established in natural language processing.
- Captures semantic relations by taking into account the n -gram overlap.

Disadvantages:

- May not fully capture syntactic and semantic correctness.
- Sensitive to exact token matching.

METEOR. The **M**etric for **E**valuation of **T**ranslation with **E**xplicit **O**rding (METEOR) is designed to improve upon BLEU by considering synonymy, stemming, and paraphrasing.

Advantages:

- Accounts for synonyms, structure and morphological variations.
- Better at sentence level than BLUE, which makes it more interesting for line completion.

Disadvantages:

- More complex and computationally intensive than BLEU.
- Originally designed for natural language, less commonly used for code.
- Deficient for evaluating larger bodies of text, due to the coherence and cohesion.
- Language dependent (e.g. different synonym database and stemmer needed).

CodeBLEU. This metric leverages the strengths of BLEU in matching n -grams and makes it more suitable for code evaluation by incorporating code syntax through abstract syntax trees (AST) and code semantics through data flow analysis, resulting in a higher correlation with human evaluation scores [70]. The authors of this metric suggest using five times the weight for keywords compared to the weight of other tokens. The metric can be simplified as follows:

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{Syntax} + \gamma \cdot \text{DataFlow} \quad (3.5)$$

where α , β , and γ are weights for the BLEU score, syntax match, and data flow match, respectively.

Advantages:

- Considers structural and semantic aspects of code, which together with the similarity-based approach of BLEU results in a more genuine assessment of the code quality.

Disadvantages:

- More complex to compute than BLEU.
- Requires additional resources like parsers.
- Still, no functional correctness is guaranteed.

ROUGE. The **R**ecall-**O**riented **U**nderstudy for **G**isting **E**valuation (ROUGE) is a recall-based metric that evaluates the overlap of n-grams (ROUGE-1 and ROUGE-2) and longest common subsequences (ROUGE-L) between the generated and reference texts [51]. These metrics are defined as follows:

$$\text{ROUGE-1} = \frac{\sum_{\text{unigrams}} \min(\text{count}_{\text{gen}}, \text{count}_{\text{ref}})}{\sum_{\text{unigrams}} \text{count}_{\text{ref}}} \quad (3.6)$$

where the numerator sums the minimum counts of each unigram (single word or token) in the generated text ($\text{count}_{\text{gen}}$) and reference text ($\text{count}_{\text{ref}}$), capturing the correctly recalled unigrams. The denominator sums the counts of unigrams in the reference text, normalizing the score to reflect recall.

$$\text{ROUGE-2} = \frac{\sum_{\text{bigrams}} \min(\text{count}_{\text{gen}}, \text{count}_{\text{ref}})}{\sum_{\text{bigrams}} \text{count}_{\text{ref}}} \quad (3.7)$$

which is similar to ROUGE-1, however, instead of unigrams, bigrams (pair of consecutive words or tokens) are considered.

$$\text{ROUGE-L} = \frac{LCS(\hat{y}, y)}{|y|} \quad (3.8)$$

where $LCS(\hat{y}, y)$ is the length of the longest common subsequence between \hat{y} and y , capturing the longest sequences of tokens that appear in both the generated and reference texts in the same order. This metric reflects the recall of the longest matching sequence, normalized by the length of the reference text.

Advantages:

- Measures recall, useful for partial correctness.
- Robust to minor lexical variations.

Disadvantages:

- May not correlate with functional correctness.
- Does not consider order beyond subsequences.

While there are improvements for ROUGE, such as ROUGE-N+Synonyms and ROUGE-TopicUniq [20], they do not transfer well to the coding domain, as semantic similarity in programming languages relies on deeper underlying logic and structural relationships compared to natural languages.

CodeBERTScore. This metric, derived from BERTScore [97], evaluates the similarity between generated and reference code by leveraging the contextual embeddings from pretrained models. Unlike traditional similarity metrics that rely on token overlap, CodeBERTScore encodes both the natural language input and the generated code to assess consistency. This yields the highest correlation with human judgments of functional correctness, making it the most optimal choice when a test-based evaluation is not available. For the detailed equations and methodology used to calculate this metric, refer to the work of Zhou et al. [101].

Advantages:

- Captures semantic similarity, making it robust to minor lexical differences.
- Highest correlation with human preferences and functional correctness [101].
- Supports multiple programming languages, including Python, Java, C, C++, and JavaScript.

Disadvantages:

- Computationally intensive due to the use of large pretrained models.
- Requires model fine-tuning for optimal performance in specific programming languages.

3.7.2 Evaluation Techniques based on Functional Correctness

Recently, metrics based on functional correctness have emerged as the gold standard for evaluating code generation models, in particular `pass@k`. These metrics offer a robust framework for assessing whether generated code meets the specified functional requirements through testing. This section provides an overview of the currently available functional correctness metrics and introduces the novel `latest-pass@n` metric, designed to offer an even more nuanced evaluation of model performance. Note, for all functional correctness metrics, the evaluation can become rather expensive computationally and the outcomes are dependent on the quality and comprehensiveness of test cases.

Pass@k. This metric measures the functional correctness by evaluating whether at least one of the top k generated code samples passes all test cases. It is defined as:

$$\text{pass@k} = \frac{1}{N} \sum_{i=1}^N f(n_i, c_i, k) \quad (3.9)$$

where:

- $f(n_i, c_i, k) = \begin{cases} 1 & \text{if } n_i - c_i < k \\ 1 - \frac{\binom{n_i - c_i}{k}}{\binom{n_i}{k}} & \text{otherwise} \end{cases}$
- n_i is the total number of predictions for problem i .

- c_i is the number of correct predictions (viz. all tests pass for the prediction) for problem i .
- k is the number of top predictions considered.
- $\binom{n_i}{k}$ is the binomial coefficient, representing the number of ways to choose k items from n_i items.
- $\binom{n_i - c_i}{k}$ is the binomial coefficient, representing the number of ways to choose k items from $n_i - c_i$ items.

Advantages:

- Directly measures functional correctness.
- Reflects practical utility in competitive coding environments.

Disadvantages:

- The binary nature may not reflect partial correctness.

Pass-ratio@n. Instead of using a binary approach like pass@k (either all tests in a test suite pass $\rightarrow 1$ or otherwise $\rightarrow 0$), Yeo et al. [88] introduce a more granular approach, which captures the accuracy according to the pass rate of the test cases in the range of $[0, 1]$ as follows:

$$\text{pass-ratio}_i = \left(\frac{\text{the number of passed test cases for problem } i}{\text{the number of test cases}} \right)^2 \quad (3.10)$$

$$\text{pass-ratio@n} = \frac{\sum_{i=1}^n (\text{pass-ratio}_i)}{n} \quad (3.11)$$

Advantages:

- Directly measures functional correctness.
- More informative for understanding the performance distribution across test cases compared to the traditional pass@k metric.

Disadvantages:

- Does not clearly extend to multiple top- k samples, appearing primarily suited for $k = 1$.
- Although squaring the pass-ratio values is intended to reflect higher degrees of accuracy, this approach lacks insight into the distribution of test case difficulties, which may lead to the potential undervaluation or overvaluation of the scores.

Latest-pass@n. To address the potential undervaluation or overvaluation of scores in pass-ratio@n, we introduce the latest-pass@n metric. This metric evaluates the ratio of *consecutively passed* test cases, thereby offering a more nuanced measure of performance. It allows for each test to be valued

appropriately by designing the test suite in an order of interest – inspired by the Ideal Discounted Cumulative Gain (IDCG) metric in the field of information retrieval [38]. Formally, `latest-pass@n` is defined as follows:

$$\text{latest-pass}_i = \frac{\text{number of consecutively passed test cases for problem } i}{\text{total number of test cases}} \quad (3.12)$$

$$\text{latest-pass@n} = \frac{\sum_{i=1}^n (\text{latest-pass}_i)}{n} \quad (3.13)$$

To maximize the insight provided by this metric, the execution order of the tests must be carefully considered. For example, if the tests are sorted by increasing complexity, this metric can indicate how far a model can progress through the problem set, offering a more cautious yet granular evaluation compared to `pass-ratio@n`. Additionally, if certain tests are of pivotal importance, such as edge cases or safety tests, this metric facilitates a benchmark design that enables the accurate evaluation of model capabilities according to specific expectations. To further mitigate the chances of undervaluation or overvaluation, an alternative approach would be to incorporate weighted scoring for the tests, thereby providing a more balanced assessment of model performance.

3.7.3 Recommended Evaluation Techniques for AI4SE Benchmarks

Building on the comprehensive analysis of the discussed metrics above, a strategic recommendation is provided below with the aim of advancing future AI4SE benchmarking practices. To illustrate the limitations and potential solutions, consider the example [Figure 3.5](#).

This example highlights the constraints of similarity-based metrics, as they tend to be overly restrictive towards correct solutions that differ syntactically from the canonical answer. While metrics such as CodeBLUE attempt to mitigate these limitations by considering code structure, they still fall short in certain aspects. When similarity-based metrics are the sole feasible option, employing a multi-metric approach is recommended for a more comprehensive assessment – as most metrics capture different facets of code quality and can be susceptible to certain biases or shortcomings [92]. However, this changed with the introduction of CodeBERTScore – if applicable for the programming language in use. This metric is the new state-of-the-art in the field, providing superior performance by leveraging contextual embeddings from pretrained models, capturing semantic similarity more effectively, and correlating highly with human judgments of functional correctness [101], making it the most recommendable metric for similarity-based evaluations.

Ultimately, the most robust evaluation technique hinges on actual functional correctness, typically assessed through unit tests. This method is highly recommended, as it is the only reliable means to verify the validity of a candidate solution. However, functional correctness is not without challenges, notably the need for meticulously curated unit tests for each problem. Additionally, functional correctness metrics often lack granularity in depicting varying degrees of correctness. To address this, new research should consider variants such as `pass-ratio@n` and `latest-pass@n`, which provide more nuanced insights into model performance. Additionally, for newer AI4SE benchmarks using cross-references, consider `recall@k` mentioned in the work of Li et al. [47, 48].

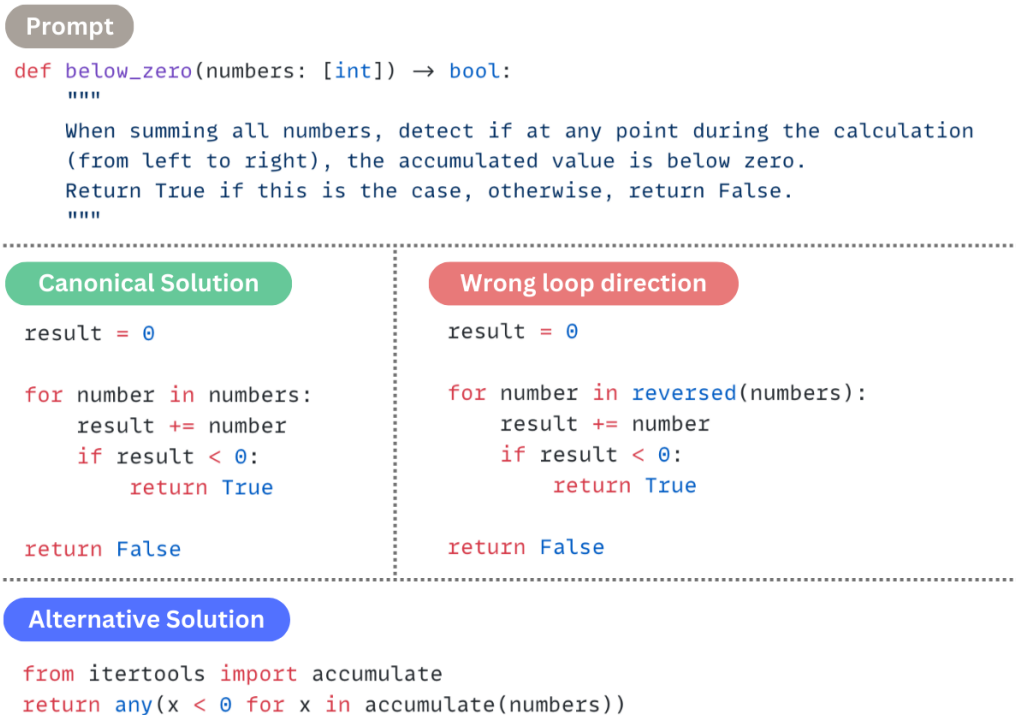


Figure 3.5: Illustration of the pitfall behind using text similarity metrics for code synthesis evaluation. Any incorrect solution that is highly similar to the canonical solution will score higher than alternative-looking solutions that do capture the semantical purpose of the function.

In summary, the selection of evaluation metrics should align with the specific requirements of the code generation task, whether it be exactness, syntactic accuracy, or functional correctness. Each metric offers a unique perspective on performance, and often a combination of metrics is necessary for a thorough evaluation. Currently, CodeBERTScore and pass@k are the state-of-the-art metrics in the field, excelling in capturing functional correctness. In the next chapter, the latest functional correctness metrics will be further explored in practice.

4

TOWARDS BETTER EVALUATION OF LLMs FOR SOFTWARE ENGINEERING: INTRODUCING HUMANEVALPRO

This chapter encapsulates, in a similar format to a research paper, the work behind the newly introduced benchmark HUMANEVALPRO, established by the findings within earlier chapters in this study.

ABSTRACT

The HumanEval benchmark is widely recognized as the most popular tool for assessing the software engineering capabilities of large language models (LLMs) in a lightweight manner. However, its widespread use has revealed significant flaws, including incorrect tests, suboptimal or wrong canonical solutions, and imprecise problem definitions – issues that have propagated into all ‘enhanced’ variants of the benchmark. Serving as a new foundation, we introduce HUMANEVALPRO, a thoroughly refined benchmark that addresses these deficiencies through rigorous peer review and comprehensive modifications. Key improvements include fixing erroneous solutions and problem definitions, enhancing support for language conversion, doubling the number of manually crafted tests, and elevating the benchmark’s difficulty to ensure ongoing relevance and impact, given the high data leakage risk and the presence of near-duplicates in training datasets. Additionally, we present a novel metric, namely `latest_pass`, to provide a more granular evaluation of performance. Our extensive experiments reveal a substantial 31.22% average and 26.02% median drop in `pass@1` scores compared to HumanEval, highlighting the increased challenge and accuracy of HUMANEVALPRO. This work underscores the critical need for high-quality, peer-reviewed benchmarks to advance the evaluation of LLMs in software engineering.

4.1 INTRODUCTION

The HumanEval benchmark [12] has become a cornerstone in the evaluation of large language models (LLMs) with software engineering capabilities, such as Codex [12], Gemini [22], and GPT-4 [64]. Despite its widespread use, numerous mistakes and suboptimalities are present in the benchmark, particularly evident during our manual translation of HumanEval-Haskell [81]. These issues have prompted a thorough re-evaluation of the entire suite of benchmarks related to HumanEval.

Although several enhanced versions of HumanEval have been developed (see Table 2.1 & Section 4.2), such as improved language support [5, 10, 61, 99] and improved test coverage [18, 53], all variants suffer from fundamental flaws. These include incorrect tests, inadequate test coverage, incorrect canonical solutions, and imprecise problem definitions. Specifically, our analysis (refer to Section 4.3.1 for more details) identified the following general issues across current HumanEval variants:

- Variants covering multiple languages have duplicated the original issues.
- Variants adding new tests used the original incorrect canonical solutions to generate the output.
- Variants based on human corrections or translations are inconsistent.

Moreover, attempted LLM-augmented strategies for benchmark improvement lack rigorous quality control, perpetuating original errors and suboptimalities. Ultimately, this analysis underscores the necessity of peer-reviewing benchmarks, even though this remains a high manual effort. Given the limitations of HumanEval, such as high data leakage risk and the presence of near-duplicates in training datasets, there is also a pressing need to elevate the benchmark's difficulty. Transforming problems into more challenging variants with additional edge cases will ensure the benchmark's ongoing relevance and impact.

To address the above issues, we introduce **HUMANEVALPRO**¹ (🍷), an enhanced foundation for the family of benchmarks around HumanEval. Moreover, we demonstrate a new metric for assessing functional correctness in a more fine-grained manner, `latest_pass`, allowing for more precise comparisons between LLMs with regard to the underlying complexity of the problems.

The outline of this chapter is as follows. Section 4.2 discusses the related work on HumanEval. Section 4.3 highlights the approach for the HUMANEVALPRO benchmark, describing the observed issues, modifications and peer-review process. Subsequently, an evaluation on various LLMs is performed, for which the experimental setup is presented in Section 4.4, the results in Section 4.5 and discussion in Section 4.6. Lastly, Section 4.7 concludes and mentions future work.

¹ <https://github.com/AISE-TUDeLft/HumanEvalPro>

4.2 RELATED WORK

Throughout the lifespan of HumanEval [12], many efforts have been made to enhance the popular benchmark. These enhancements can be categorized as follows: improved language support, different task and instruction setups, improved testing, and extended problem sets.

For improved (programming) language support, Cassano et al. [10] present MultiPL-HumanEval, based on their newly introduced language conversion framework for benchmark tests: MultiPL-E. Through translation of unit tests and function headers, in addition to changing language-specific terminology in the prompt, it extends HumanEval to 18 other programming languages. However, canonical solutions are not translated. Consequently, Athiwaratkun et al. [5] proposes a new language conversion framework, where canonical solutions are also included, creating Multilingual HumanEval, a variant covering 12 languages. Besides automated conversion, Zheng et al. [99] translate the full benchmark using human-crafted samples into 6 programming languages: HumanEval-X. This variant has been evaluated and modified by Meunnighoff et al. [61], performing additional cleaning and support for Rust, resulting in HumanEvalPack.

HumanEvalPack also provides different tasks besides code synthesis (HumanEval-Synthesize), namely HumanEval-Fix for code repair and HumanEval-Explain for code explanation. Similarly, Instruct-HumanEval [14] provides a different setup, namely catering the prompts towards a more instruction-based format, covering a human-assistant style, for improved interpretation for instruction-based models.

As the original HumanEval lacked test coverage, Liu et al. [53] introduce HUMAN-EVAL⁺, with 80 times the number of tests as the original. For efficiency, they also release HUMAN-EVAL⁺-MINI, with 47 times the number of tests. The test suite is extended by EvalPlus, an automatic test input generator, using LLM- and mutation-based strategies. Dong et al. [18] release HE-Eval, with 14 times the number of tests.

Lastly, Xia et al. [85] introduce EVO-EVAL, which evolves the HumanEval problem set by extending it with five new domains: *subtle*, *creative*, *difficult*, *combine* and *tool use*. Using the original problems as seeds, the problems have been transformed into different domains using LLMs – also using EvalPlus to generate additional test cases.

4.3 APPROACH

As mentioned, the original and variants of the HumanEval benchmark all contain flaws. To establish a proper new foundation for this family of benchmarks, we pursue the following approach – also illustrated in Figure 4.1. First, we initiate a comprehensive code review, leading to standardized observations (Section 4.3.1). Then, we address these identified issues through a series of modifications (Section 4.3.1), followed by a peer review (Section 4.3.3) to ensure accuracy and reliability. Finally, in the upcoming sections, we will experiment with the revised benchmark to evaluate and discuss the results. Below, more details are provided for specific steps in this approach.

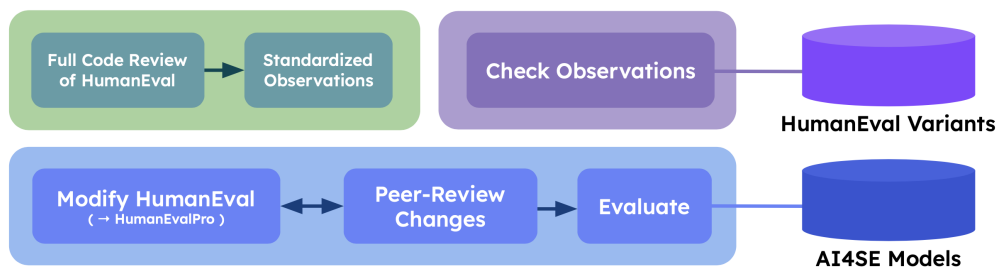


Figure 4.1: Outline of the general approach behind HUMANEVALPRO.

4.3.1 Standardized Observations in Current HumanEval Benchmarks

Upon careful examination, manual reconstruction of canonical solutions, and experimentation with the test suites of various HumanEval benchmarks, we identified several consistent issues across all problems:

Incorrect and Suboptimal Code

- Canonical solutions are incorrect (see *testing* point below) and inefficient.
- Canonical solutions do not account for assumptions stated in the problem description.
- Only a small subset of problems includes type annotations in the function headers.

Lack of Quality Testing

- Example tests in the prompt overlap with those in the test suite. This makes test suites with 5 assertions rather ineffective when the model is already given 3 tests in the problem description.
- Incorrect canonical solutions pass the tests.
- Mistakes in the tests of the problem description, where the input of test examples does not produce the stated output upon executing the canonical solution.

Poor Language Quality

- Grammar mistakes, poor explanations.
- Ambiguous instructions leading to mismatches between the canonical solution and the tests.
- Formatting of the problem description is inconsistent, particularly the test examples.

Suboptimal Support for Language Conversion Frameworks

- MultiPL-E, the only ready-to-use language conversion framework with the best language support, currently only supports equality assertions, whereas many problems feature an incompatible setup.

For a more precise overview of the observed issues in altered versions of the HumanEval benchmark, see [Table 4.1](#).

Table 4.1: Overview of the issues present in AI4SE benchmarks stemming from HumanEval [12]. The example issues mentioned originate from further insights provided in [Appendix A](#).

Category	Name	Issues	Note
Original	HumanEval [12]	✓	<p>Incorrect tests (e.g. #47)</p> <p>Incorrect canonical solutions (e.g. #95)</p> <p>Lack of proper test coverage (e.g. #95)</p> <p>Imprecise docstrings (e.g. #163)</p>
Improved Language Support	MultiPL-HumanEval [10]	✓	<p>Similar issues as original, besides:</p> <p>No incorrect canonical solutions (e.g. #95), as only unit tests are translated and used in MultiPL-E [10].</p>
	HumanEval-Fix [61]	✓	Similar issues as original
	HumanEval-Explain [61]	✓	Similar issues as original
	HumanEval-Synthesize [61]	✓	Similar issues as original
	HumanEval-X [99]	✓	<p>Similar issues as original, besides:</p> <p>Inconsistent manual translation, e.g. JavaScript fixed the docstring issue of #47, Rust does not even provide examples, the other languages still have the issue. For #95, the incorrect original solution is not fixed for all languages (Python and Go).</p> <p>Suboptimal manual translation, e.g. best time-complexity is also worst in #95 for JavaScript and Rust. Furthermore, for most languages in #95, the translators are only focused on returning the correct output and write semantically suboptimal code ("hacking" the logic by setting the state to "mixed" instead of returning false when the wrong type is given, as done originally).</p>
	Multi-HumanEval [5]	✓	<p>Similar issues as original, besides:</p> <p>No incorrect canonical solutions (e.g. #95), as they are not provided (besides Python).</p>
Improved Testing	HumanEval+ [53]	✓	<p>Similar issues as original, besides:</p> <p>Inconsistent manual corrections, some implementations have been altered such that the mistakes are fixed (e.g. #95), yet other alterations have worsened the quality and still lack proper test coverage (e.g. #163).</p>
	HumanEval-MINI [53]	✓	Similar issues as HumanEval+ , though features fewer tests (tradeoff between performance and coverage).
	HE-Eval [18]	✓	Similar issues as original
Instruction-based	InstructHumanEval [14]	✓	Similar issues as original

4.3.2 Modifications in HUMANEVALPRO

In HUMANEVALPRO, we address all the above issues by manually modifying all problems in the original HumanEval benchmark. To summarize, these are the general changes made in HUMANEVALPRO and their benefits:

- **Fixing all suboptimal and incorrect canonical solutions**, previously uncaught due to the lack of testing and overall quality review.
- **Adding type annotations** to all problems, providing useful context and facilitating easier translation to other programming languages. The original HumanEval benchmark only featured type annotations for the first 30 problems, which is 18% of all problems (164).
- **Better support for language conversion frameworks.** HUMANEVALPRO has improved language support for frameworks such as MultiPL-E [10], which can translate to 18 other programming languages, by changing all tests to equality assertions if possible. This reduces the number of incompatible problems by roughly ten times.
- **Adding edge cases for each problem** (e.g. negative numbers, zero cases, empty input, non-alphanumeric characters), such that high-quality AI4SE benchmarks can only pass if they carefully consider and handle different scenarios similarly to the intuition of excellent engineers.
- Similarly, **adding assertions** in the code whenever the problem description mentions constraints. High-quality AI4SE models should not neglect this information and adding this increases the data provided in the benchmark.
- **Improving the test examples** in the problem description to enhance the quality of the evaluation - performance is sensitive to prompt examples [6]. Test examples in the problem description are also not solely used anymore for testing the generated code. In addition, for some problem descriptions with a large number of test examples, the number has been reduced to a minimum, such that the evaluation of the model is distributed more fairly and critical.
- **Removing any spelling errors** in the descriptions of the problems, **consistently formatting** the descriptions and making sure any **description matches the implementation** yet also leaves adequate room to test the model on its capabilities to solve the problem intuitively, as expected for high-quality AI4SE models.
- **Elevated difficulty** by adding **more edge cases** and altering the various problems. By adding difficult edge cases, this benchmark expects models to perform similar to intuitiveness of actual engineers. Furthermore, more challenging and differently phrased problems aim to solve the issues around the high risk of data leakage with the original benchmark in addition to the saturated performance scores in the HumanEval leaderboards.

To illustrate the modifications of the tests in HUMANEVALPRO, consider [Table 4.2](#) below.

Table 4.2: Comparison of test statistics between HumanEval (based on `human-eval-v2-20210705.json` listed as `HumanEvalOriginal.json` in the repository) and HUMAN-EVAL-PRO.

Metric	HumanEval (Original)	HUMAN-EVAL-PRO	Δ
Total number of asserts	1325	2551	$\times 1.92$
Average number of asserts	8	16	$\times 2$
Median number of asserts	7	11	$\times 1.57$
Min. number of asserts	1	4	+3
Problems with < 5 asserts	34	2	-94%

4.3.3 Peer Review Process of HUMAN-EVAL-PRO

To back up our commitment to accuracy and reliability, all the changes made during the creation of the initial version of HUMAN-EVAL-PRO have been repeated by an independent reviewer. In short, the following was carefully checked:

- **Clarity of docstrings:** Each problem’s docstring, viz. the problem description, is reviewed for clarity, and completeness.
- **Consistency between docstring and canonical solution:** Verification of the alignment between the canonical solution and the problem’s docstring. This step involves checking that the tests provided are consistent with both the docstring and the canonical solution, ensuring there are no discrepancies.
- **Correctness and efficiency of canonical solutions:** The canonical solution is assessed for correctness. If inefficiencies are easily identifiable, suggestions for optimization are provided. This ensures that the solutions are not only correct but also optimize for efficiency.
- **Comprehensive test cases:** The test cases are reviewed to identify any missing expected behavior or scenarios where faulty solutions might still pass. This step is crucial for ensuring that the tests are robust and cover a wide range of edge cases.

While the initial crafting of this benchmark took over hundred hours, the additional review only required another 16 hours to complete. The peer-review of the 164 problems (100%) resulted in the following modifications:

- Redesign of 2 problems (1%), both sharing a similar structure. This modification primarily accounts for the observed changes in the new results.
- A few additional test cases for 15 problems (9%).
- Simple grammar or clarity enhancements, e.g. adding ‘the’ before certain words, for 25 problems (15%).

All suggested changes were clearly documented and reviewed by the original author, with all suggestions being implemented or improved differently. The data behind this peer-review process is also available, upon request.

Since there are changes beyond the test suites, implementing the peer-review suggestions also required re-running the completions for all models. Notably, the results gathered before the peer-review

process already demonstrated great similarity with the results based on the current version of HUMAN-EVALPRO, as the `pass@1` scores did not result in a substantial change for most considered models nor did the order of the results change. To be more specific, this was the detailed impact on the `pass@1`-scores of the models:

- For 40% of the models, there was no change in score.
- For 50% of the models, the change was at most 1-2%.
- The remaining 10%, initially top-performing in the original HumanEval benchmark, dropped 5% in score.

Note, these percentages are values regarding the absolute change in score, not the relative percentages to the original scores. For example, the writing style above denotes a drop from 45% to 40% as -5% , not $\frac{(40-45)}{45} \approx -11\%$.

Overall, the peer-review process confirmed the quality of the initial modifications and provided final enhancements. In the upcoming sections, the experimental setup and results of the peer-reviewed version, i.e. the released version of HUMAN-EVALPRO, will be showcased in more detail.

4.4 EXPERIMENTAL SETUP

This section outlines all the data, layers, and components in HUMAN-EVALPRO that contribute to its evaluation or the understanding thereof. First, context is provided regarding the data and setup of HUMAN-EVALPRO in [Section 4.4.1](#). Then, an overview of the models and their systematic configuration is given in [Section 4.4.2](#) and [4.4.3](#). Lastly, the metrics, including the newly introduced `latest_pass` metric, are described in [Section 4.4.5](#). With all these elements combined, the results presented in [Section 4.5](#) can be accurately interpreted.

4.4.1 Benchmark Details

Understanding the root elements of HUMAN-EVALPRO is crucial, particularly its data layout and setup. The benchmark comprises 164 unique problems, each identified by a task ID. Each problem consists of executable code files, broken down into separate parts and stored in *.json*-format. This structure of the code is as follows:

- **Imports** – these are rarely needed and hence often not included.
- **Function Header** – provides the model with specific parameters for the problem, including the types, also of the expected output.
- **Function Description** – Explanation of the problem to solve in the function body.
- **Canonical Solution** – The answer to the above, in the form of the function body. Mainly provided for completeness as this is replaced by model completions when benchmarking.
- **Test Suite** – To check the function body, a series of input-output pairs formatted as assertions is supplied.

During inference, an instructional preamble is prepended to each prompt (= imports + function header + function description), clearly defining the task for the models:

General Instruction

“Your task is to finish the implementation of the function below according to the docstring. Keep in mind all possible edge cases. Only provide the implementation of the function.”

To better visualise the above data and other benchmark details, refer to [Figure 4.2](#) below.

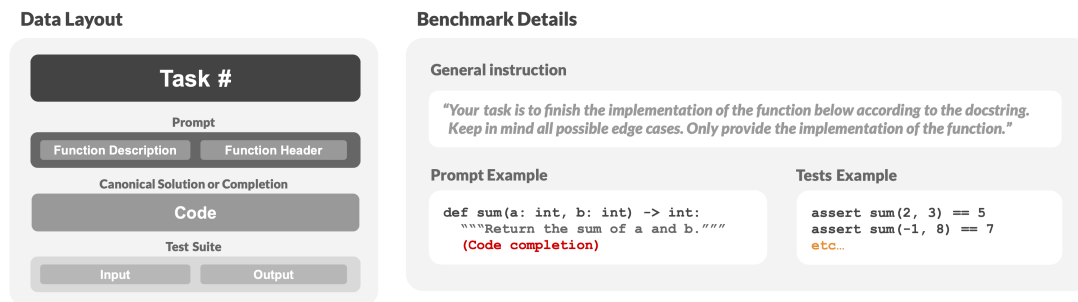


Figure 4.2: Illustrative overview of the data layout and benchmark details of HUMANEVALPRO.

4.4.2 Models

To get a general overview of the impact of HUMANEVALPRO, experiments with various LLMs are needed. Ultimately, 10 LLMs have been selected, conforming to the following:

- Available via the Hugging Face module `transformers`².
- `pass@1` values are made available, either in the published paper or via popular and well-maintained leaderboards such as BigCode³ and EvalPlus⁴.
- A wide spectrum of performance is covered.

To facilitate the automated use of these models, a ‘model hub’ has been developed (see [Figure 4.3](#)). It automatically configures each model correctly by setting up the model endpoint, specifying the prompt instruction format, and potentially including a system message – all defined in the documentation of the models. This abstraction effectively separates the concerns of gathering completions accurately for each model, also making it applicable in a broader scope within this field. To utilize this functionality, one simply needs to import the wrapper classes provided in the HUMANEVALPRO GitHub repository ([🔗](#)).

² <https://huggingface.co/docs/transformers>

³ <https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>

⁴ <https://evalplus.github.io/leaderboard.html>

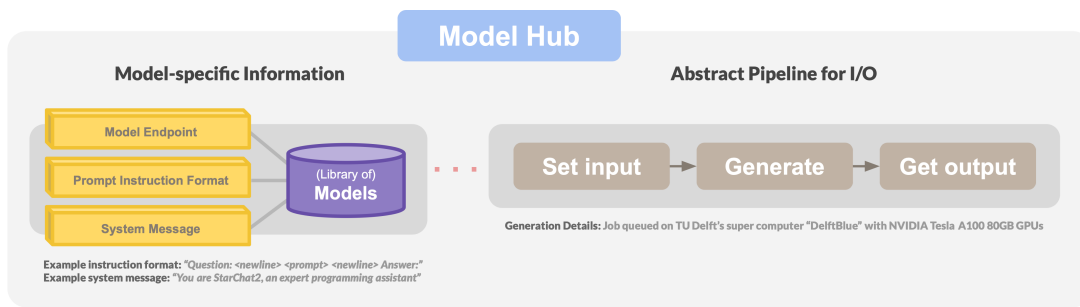


Figure 4.3: The *Model Hub* abstraction, designed to easily yet accurately gather completions of various models.

4.4.3 Configuration

The configuration of the models and the execution environment is as follows:

- In gathering completions across different models, a deterministic approach is employed for each model, utilizing at most 512 tokens and generating only a single sample. Models are run on DelftBlue, a supercomputer made available by Delft High Performance Computing Centre (DHPC) [16], utilising one NVIDIA A100 80GB GPU and 32 CPU cores.
- During test execution, a timeout limit of 15 seconds is utilised per function call to disregard completions that are potentially looping forever or are considered overly inefficient with regard to the canonical solutions. Each test is executed in an evaluation suite using the precautions deployed by OpenAI, e.g. a reliability guard disabling various destructive functions, as documented in `execution.py` from the repository of the original HumanEval benchmark (🔗).

4.4.4 Post-processing Completions

As some models may continue generating output beyond the required function implementation, or other noise, a systematic approach to extract the relevant data from the model completions has been developed. The two main interests are:

- The function body, i.e. the required completion of the prompt.
- Added imports within the generated data – possibly declared outside the function body.

Below, insight is given as to how these two parts within the generated data can be collected in an automated manner. For the evaluation of HUMAN EVAL PRO, manual verification was also performed, however, a significant speed-up in the process was obtained via the use of tools based on the methods described below.

Extracting Only the Function Body from Completions

To extract only the function body from the model completions, the following approach is used:

1. **Identify the function declaration:** The process begins by locating the earliest instance of the keyword `def`, which marks the start of the function definition. Any text preceding this declaration is considered noise and is removed. It is important to take the earliest `def`, as functions might have other functions declared within them.

2. **Remove extraneous content after the function body:** To ensure that only the relevant part of the function is retained, the process continues by identifying the latest return statement within the function. All lines following this statement are considered unnecessary and are removed. In practice, oftentimes models continue with writing tests, following the structure of the HumanEval data, most likely due to data leakage.
3. **Store and alert for truncated completions:** If no return statement is found, we store this information, as it might provide model-specific insights during the evaluation step.

Combining the above into a rule-based method ensures that the extracted function body is as free from any noise or extraneous content as possible, making the evaluation process more efficient and accurate.

Extracting imports from completions

Given the response of a model, it could be the case that the model has imported certain libraries. Since we are only interested in the completion of the function, i.e. the function body, and the imports could be defined above the function header whenever the model returns the entire function or amidst other noise, we need to extract the imports from the response separately.

The algorithm to extract the imports is rather simplistic:

1. Look for lines that contain the keyword `import`.
2. Remove any trailing spaces or tabs.
3. If the line starts with `import` or `from` (reducing the chance of including other sentences using `'import'`), attach the line to the problem-specific list of imports for the respective model.

Then, during the evaluation of the completion, the stored imports can be included before the function declaration, such that the completion is evaluated correctly instead of wrongly judging functional code due to preventable execution errors. In practice, this post-processing step is mainly used to extract `import hashlib` for Task ID 162. Other tasks should not require other imports, however, this function allows for flexibility if needed.

4.4.5 Metrics

To assess the impact of HUMANEVALPRO, functional correctness is used as the evaluation score of the benchmark. Instead of only using the `pass@k` metric, two other variants are also used to get a more fine-grained perspective into the performance of models, namely the `average_pass` and `latest_pass`. The overall score for the benchmark can be calculated as follows:

$$\text{Evaluation score benchmark} = \mathbb{E}_{\text{problems}} [\textit{pass-metric}] \quad (4.1)$$

where *pass-metric* can be substituted for `pass@k`, `average_pass` and `latest_pass`, all defined below.

The `pass@k` metric is introduced by the authors of the original benchmark, HumanEval [12], as follows:

$$\text{pass}@k = \begin{cases} 1 & \text{if } n - c < k \\ 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} & \text{otherwise} \end{cases} \quad (4.2)$$

where:

- n is the total number of predictions.
- c is the number of correct answers (viz. all tests pass for the prediction).
- k is the number of top predictions considered.
- $\binom{n}{k}$ is the binomial coefficient, representing the number of ways to choose k items from n items.
- $\binom{n-c}{k}$ is the binomial coefficient, representing the number of ways to choose k items from $n - c$ items.

Since only one sample is generated using greedy decoding, `pass@1` can be simplified to a binary value, where ‘1’ denotes all tests passing and ‘0’ otherwise. For the entire benchmark, this indicates the ratio of completely passed problems. However, a shortcoming in this approach is the lack of insight into the varying degrees of correctness. To provide more nuance in the assessment of models, the following two variants will also be reviewed:

$$\text{average_pass} = \frac{p_t}{t} \quad (4.3)$$

$$\text{latest_pass} = \frac{p_c}{t} \quad (4.4)$$

where:

- p_t is the total number of passed tests.
- p_c is the number of consecutively passed tests, until the first non-passing test is reached.
- t is the number of available tests.

To gain maximum insight from the `latest_pass` metric, the execution order of the tests needs to be considered. For example, if the tests are sorted on increasing complexity, this metric can represent how far a model can solve the problem set in a more careful, yet still granular, way compared to `average_pass`. Given that HUMAN-EVAL-PRO considers new edge cases, including simpler ones such as empty input, the `latest_pass` metric can properly penalize models according to this design by executing such tests early on. Meanwhile, `average_pass` could still score relatively high by continuing to check all test cases, ultimately denoting the expected average pass ratio with no insight into the varying degrees of underlying complexity within the test suite.

4.5 RESULTS

After running the experiments as described in [Section 4.3](#) and [4.4](#), the main finding is an **average drop of 31.22%** and **median drop of 26.02%** (both in absolute percentages) in pass@1 scores of HumanEval compared to the newly introduced HUMANEVALPRO benchmark, based on 10 LLMs. The results per model are shown in [Figure 4.5](#), with a summarized illustration of the performance drops in [Figure 4.4](#) below.

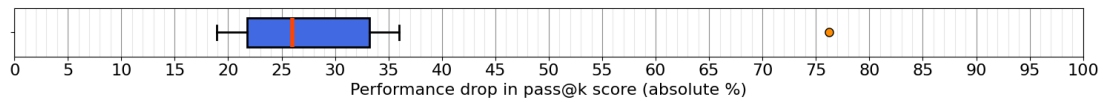


Figure 4.4: Boxplot depicting the distribution of absolute drops in pass@1 score between HumanEval and the newly introduced HUMANEVALPRO benchmark, based on 10 LLMs ([Figure 4.5](#)).

Furthermore, specifically for the HUMANEVALPRO benchmark, novel variants of the pass@ k metric, namely `average_pass` and `latest_pass`, have been investigated and the results are shown in [Figure 4.6](#). The variants display a wider spectrum of scores compared to pass@1. In addition, the scores always rank in the following order: `pass@1 < latest_pass < average_pass`.

Lastly, insights regarding the overall complexity of problems within HUMANEVALPRO are visualised in [Figure 4.7](#). The theory behind this figure is explained in more detail in [Section 3.5](#).

Note, all scripts to develop or gather any data in the results are available in the GitHub repository ([🔗](#)).

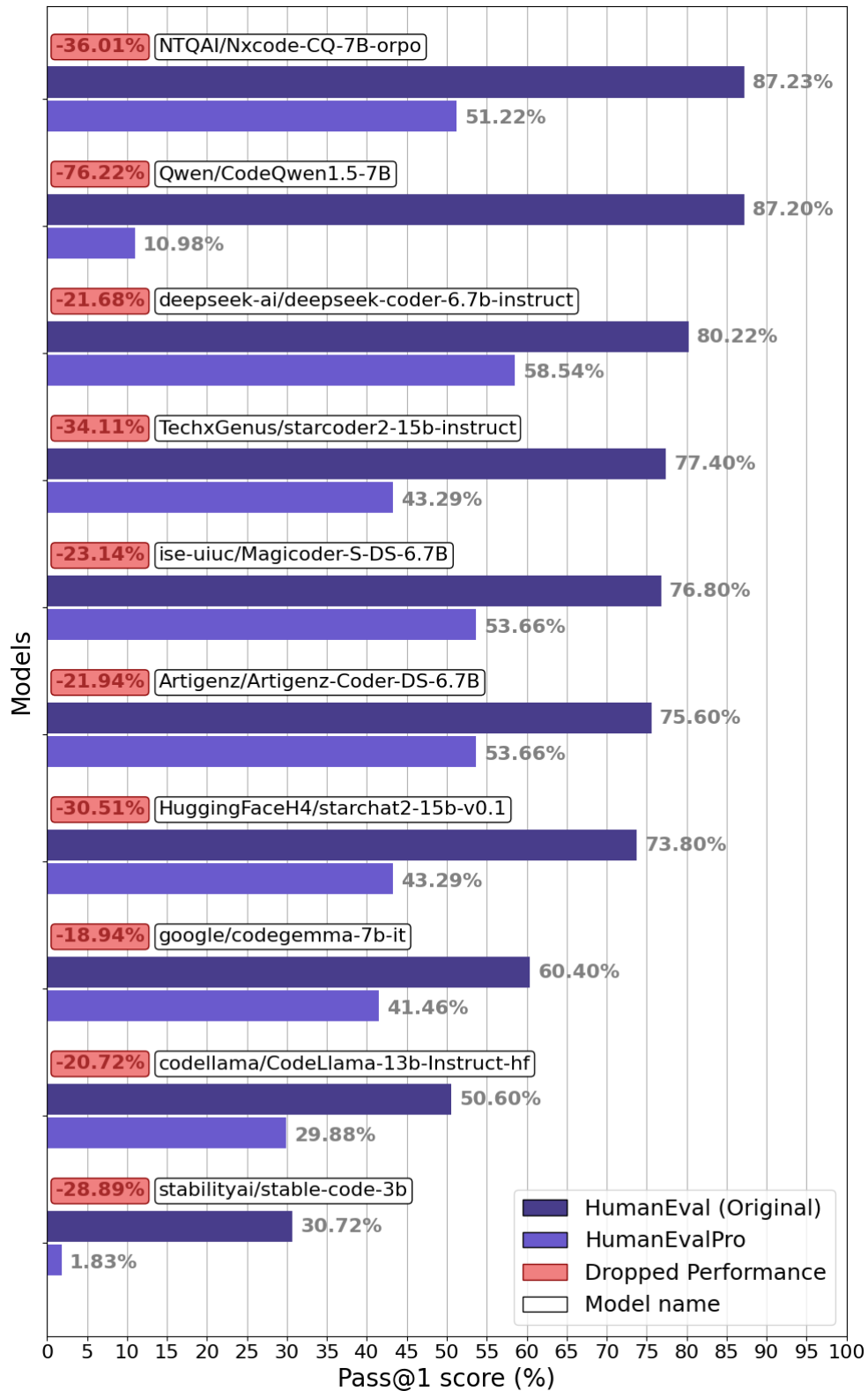


Figure 4.5: Comparison of pass@1 scores between the original HumanEval benchmark and HUMANEVALPRO.

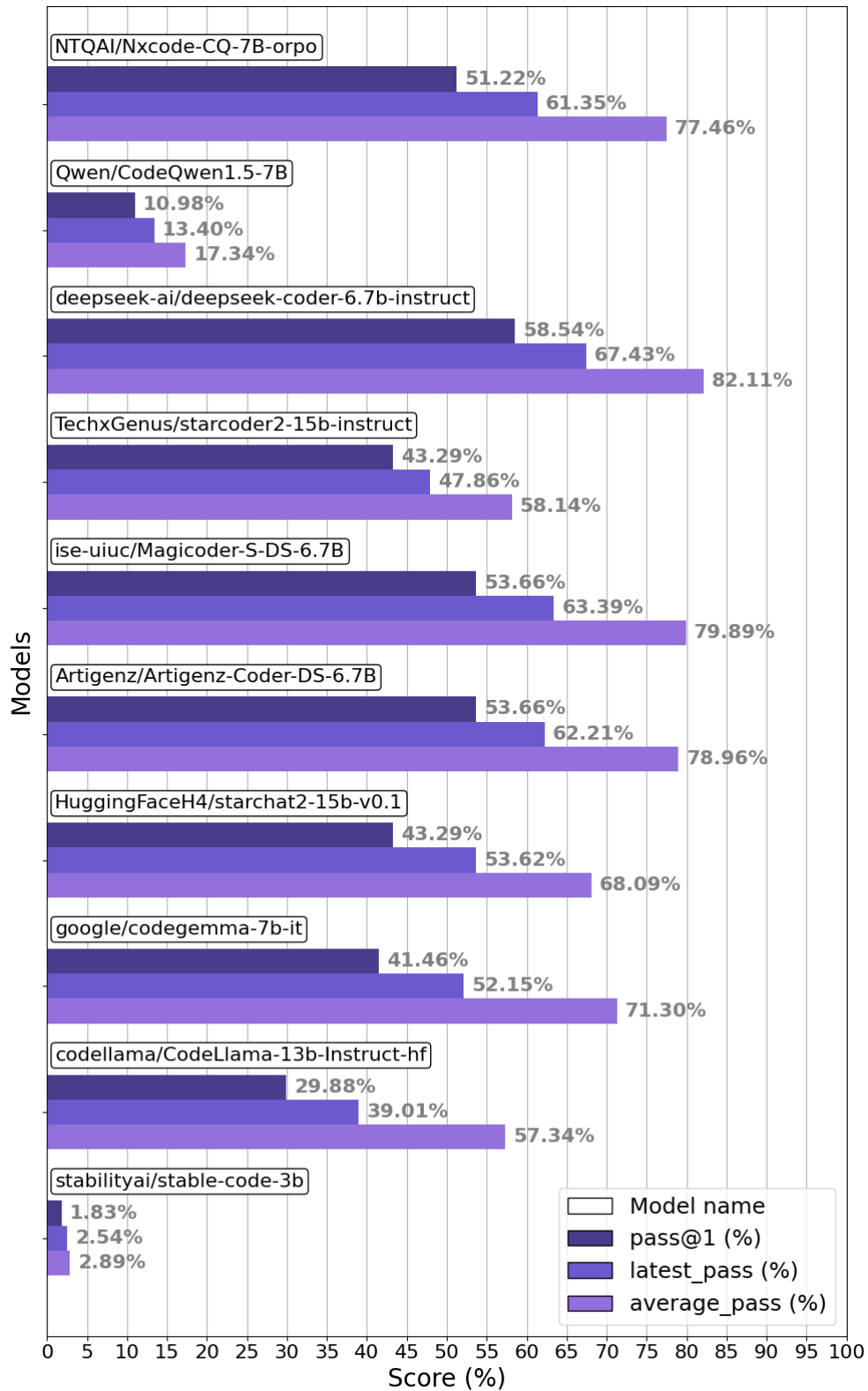


Figure 4.6: Comparison of different *pass*-metrics on HUMAN EVAL PRO.

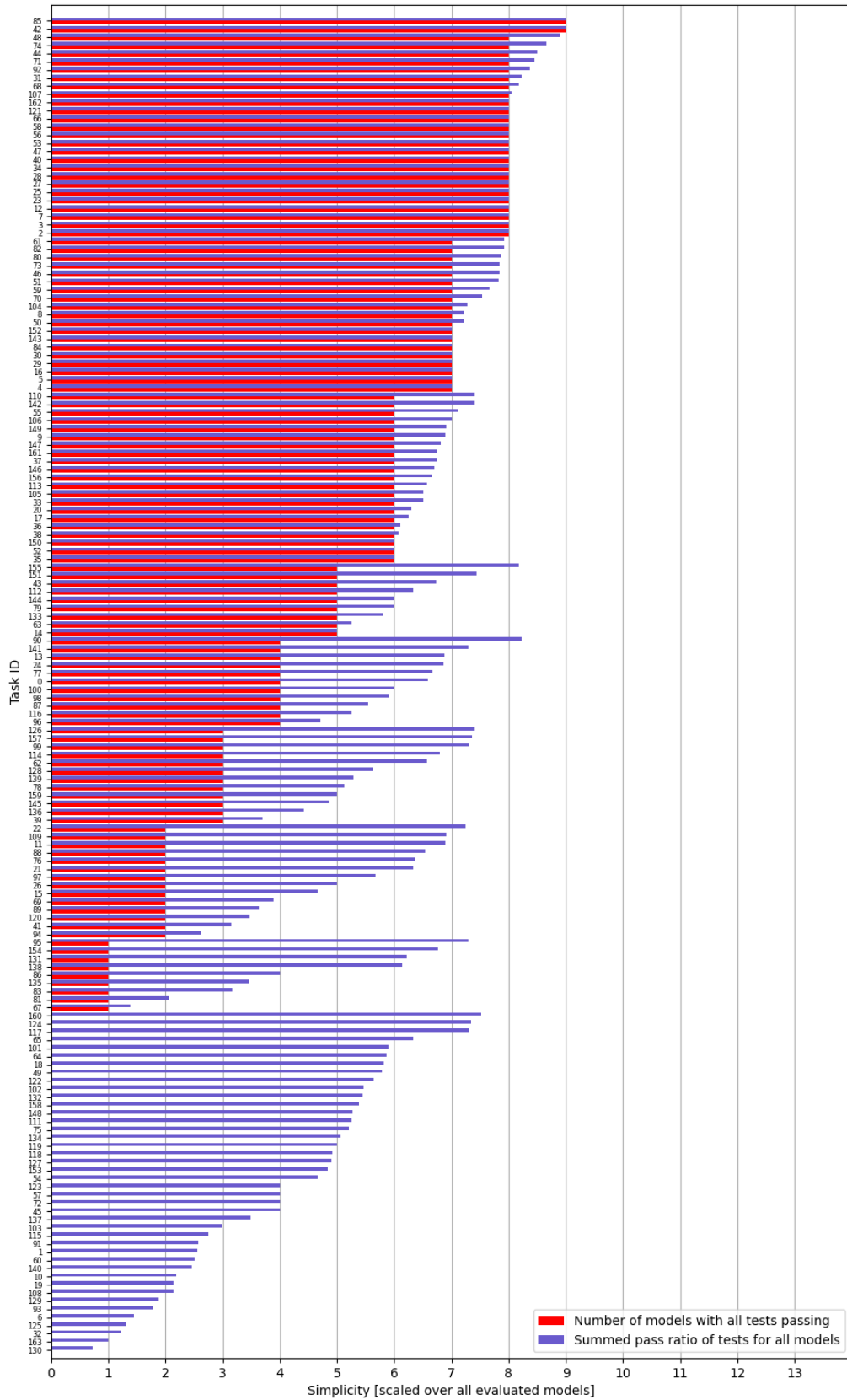


Figure 4.7: Distribution of the increasing “difficulty” (or in inverse, the decreasing “simplicity”) for HUMANEVAL-PRO problems based on evaluating 10 LLMs on functional correctness.

4.6 DISCUSSION

Overall, the results listed in [Section 4.5](#) demonstrate a notable decline in model performance when evaluated with HUMANEVALPRO compared to the original HumanEval benchmark. This trend underscores the increased difficulty and improved assessment accuracy provided by HUMANEVALPRO, which even features more robust environments with type annotations and clearer instructions.

A closer look reveals that the top-performing models from the original HumanEval benchmark do not maintain their standings in HUMANEVALPRO. For example, while HUMANEVALPRO consistently ranks `deepseek-ai/deepseek-coder-6.7b-instruct` as the top performer across all metrics, previous leaders like `NTQAI/Nxcode-CQ-7B-orpo` and `Qwen/CodeQwen1.5-7B` show a significant drop in their rankings. Specifically, `NTQAI/Nxcode-CQ-7B-orpo` falls from an impressive 87.23% `pass@1` in HumanEval to 51.22% in HUMANEVALPRO, and `Qwen/CodeQwen1.5-7B` plummets from 87.2% to 10.98%. This sharp decline suggests that certain models may have benefited from data leakage or other issues in the original HumanEval benchmark, indicating the necessity for a more rigorous benchmark like HUMANEVALPRO to accurately assess model performance.

Especially the resilience of models, e.g. `deepseek-ai/deepseek-coder-6.7b-instruct`, can be confirmed by evaluating the model on the new challenges presented in HUMANEVALPRO. When models also score relatively well in this benchmark, it highlights the models' ability to adapt and perform competently under more demanding conditions, making HUMANEVALPRO a great benchmark to reveal the reliability of the capabilities of models. This finding also emphasizes the need for regular updates to benchmarks, as reliance on outdated benchmarks can misrepresent model capabilities over time, even when models claim not to train on the test data.

Furthermore, an interesting observation emerges when analyzing model size and performance: bigger is not always better. Larger models such as `TechxGenus/starcoder2-15b-instruct` and `HuggingFaceH4/starchat2-15b-v0.1` do not consistently outperform smaller models. This observation suggests that model size alone is not a definitive predictor of success in complex, edge-case-inclusive benchmarks like HUMANEVALPRO. For instance, looking at `pass@1` scores, `TechxGenus/starcoder2-15b-instruct` scores 77.4% on HumanEval but drops to 43.29% on HUMANEVALPRO, while `ise-uiuc/Magicoder-S-DS-6.7B` scores 76.8% on HumanEval (lower) but only drops to 53.66% on HUMANEVALPRO (higher), highlighting that increased model size does not necessarily equate to better performance in more challenging assessments.

Evaluating additional pass-metrics beyond the commonly used `pass@1` metric provides the following insights:

- **Wider spectrum of percentages:** The inclusion of metrics such as `latest_pass` and `average_pass` reveals a broader spectrum of performance percentages, enabling clearer differentiation between models. This wider range helps highlight subtle performance differences that `pass@1` alone might not capture.
- **Model ranking variations:** Different metrics can result in varied model rankings. While the top-tier and worst-performing models consistently rank across all pass-metrics in HUMANEVALPRO, a different order in ranking is obtained for the mid-tier models like `TechxGenus/starcoder2-15b-instruct`, `HuggingFaceH4/starchat2-15b-v0.1` and `google/codegemma-7b-it`. These shifts show that `average_pass` and `latest_pass` are able to capture different

facets of model performance compared to the commonly used `pass@k` metric – indicating a need for more data and research on these metrics.

- **Resemblance to `pass@1`:** If only more nuance is desired, based on these results, the `latest_pass` metric aligns more closely with `pass@1` compared to `average_pass`.
- **Effectiveness of `latest_pass`:** The `latest_pass` metric proves effective for providing a detailed view of a model's performance, particularly in handling initial, simpler test cases, as this was the common order of tests throughout HUMANEVALPRO. However, its effectiveness might be limited if the order of test cases skews results, necessitating a critical evaluation of this metric to ensure it accurately reflects model capabilities.

Lastly, ranking problems based on their difficulty using accumulated pass metrics per problem (Figure 4.7), confirms that HUMANEVALPRO presents a well-distributed range of complexity over the complete set of challenges. It also shows the increased difficulty of the benchmark, where even the easiest problems are not universally passed, with roughly 30% of the models still failing to solve them. Altogether, this distribution highlights HUMANEVALPRO's effectiveness in providing a thorough assessment environment, making it an excellent tool for evaluating straightforward coding capabilities of LLMs in a lightweight manner - the main reason behind the popularity of the original HumanEval benchmark and its variants.

4.7 CONCLUSION AND FUTURE WORK

The introduction of HUMANEVALPRO represents a significant step forward in the field of AI4SE benchmarks, showcasing the effect of a lack of rigorous quality control. By addressing the limitations of the original HumanEval benchmark and incorporating more rigorous test cases, HUMANEVALPRO provides a more accurate and challenging assessment of model performance for evaluating straightforward coding tasks in a lightweight manner. With substantial drops in score, 10 LLMs showcasing a 31.22% average and 26.02% median drop in `pass@1` scores compared to HumanEval, the results confirm the need for comprehensive benchmarks that are well-maintained. Hence, we urge the authors of the benchmarks based on HumanEval to update their benchmarks with this new seed, leveraging the benefits of this new foundation, and any other developers of AI4SE benchmarks to rigorously focus on quality control - even though it can be a high manual effort, this is of pivotal importance for the development of more robust and reliable models.

Future work on HUMANEVALPRO will focus on expanding to additional programming languages, which it already has been optimized for, yet these variants have not been produced or evaluated. Additionally, further exploration of various setups of the `latest_pass` metric could yield significant insights into the evaluations of model performance. This paper already provides a glimpse of how more nuanced metrics regarding functional correctness can lead to a more comprehensive understanding of model capabilities, yet there needs to be more data and research to move the field forward. Lastly, while evaluating 10 LLMs provides valuable insights, it remains unclear how larger, top-performing models behind paywalls, such as GPT and Gemini, would perform; unfortunately, due to financial constraints, these models were excluded from this study.

5 | SUMMARY

Artificial Intelligence (AI) has rapidly advanced, profoundly impacting numerous domains, including software engineering. AI-driven tools such as ChatGPT and Copilot have revolutionized engineering workflows, garnering significant commercial interest from leading companies like OpenAI, Google, and Meta. The performance of such tools depends on the underlying model, which is assessed using benchmarks, leading to a competitive environment where achieving top scores is paramount. However, this emphasis on performance has overshadowed the importance of the quality and rigor of these benchmarks. With the absence of such studies, this thesis addresses this gap by focusing on two main objectives: reviewing and improving benchmarking practices in the field of AI for software engineering (AI4SE) and establishing an enhanced foundation for HumanEval – the most prominent, yet erroneous, AI4SE benchmark in the recent years of the AI4SE field developing.

By analyzing nearly a hundred of the most prominent AI4SE benchmarks from the past decade, a comprehensive categorization of these benchmarks has been established. The identified categories are as follows:

- Benchmarks derived from **HumanEval**.
- Benchmarks derived from **MBPP**.
- Benchmarks focusing **competitive programming, code complexity, code efficiency**.
- Benchmarks related to **data science**.
- Benchmarks concentrating on **mathematical problems**.
- Benchmarks involving **code translation** and frameworks useful for benchmark conversion.
- Benchmarks based on **real-to-life scenarios**, which include wider context and common software engineering tasks.
- Benchmarks that leverage the power of **APIs**, particularly relevant for Retrieval-Augmented Generation (RAG) models.
- Benchmarks incorporating **natural language processing** tasks (e.g., text to code, code to text, and text to text).
- Benchmarks related to **pseudocode**.
- Noteworthy **crowd-sourced benchmarks**.
- Benchmarks focused on **bug repair, test generation, and understanding**.

Furthermore, by analyzing these benchmarks, several key challenges and takeaways have been identified to improve the future of AI4SE benchmarks:

- **Quality Control.** Benchmarks rarely denote or incorporate quality control in their processes. While it can be labor-intensive, it is essential for maintaining the integrity and usefulness of AI4SE benchmarks. Quality control can be achieved through independent peer review, crowd-sourced review, LLM-based review, or by releasing new versions upon accumulation of external feedback.
- **Programming Language Diversity.** With approximately 80% of AI4SE benchmarks supporting Python and roughly 40% Java, followed by a substantial drop for other languages, there is an urgent need for greater programming language diversity in the AI4SE field.
- **Natural Language Diversity.** With only five benchmarks supporting different natural languages, there is a clear need for the inclusion of a broader range of natural languages in AI4SE benchmarks.
- **Task Diversity.** At a macro level, a broader spectrum of AI4SE benchmarks should be considered, as current benchmarks mostly focus on code generation, completion, summarization, and program repair, leaving many other software engineering tasks underrepresented. At a micro level, a more granular examination of tasks within benchmarks is needed. Solutions include providing an overview of difficulty levels for all problems in the benchmark, creating a taxonomy to provide a clear scope of the data, and comparing features of the data with real-life data to ensure balanced task diversity.
- **Purpose Alignment.** At the outset of this study, AI4SE benchmarks did not align well with real-to-life software engineering scenarios, necessitating a shift. However, as of yet, this advancement is well underway where benchmarks accurately start representing real-world challenges. Still, much future work is needed, including the integration of multimodality in AI4SE benchmarks.
- **Evaluation Techniques.** The selection of evaluation metrics should align with the specific requirements of the code generation task, whether it be exactness, syntactic accuracy, or functional correctness. Especially when only similarity-based metrics are applicable, a multi-metric approach is needed, as each metric offers a unique perspective on performance (viz. similarity-metrics are inadequate when not considered together with other insights). However, if applicable for the programming language in use, `CodeBERTScore` is recommended as the state-of-the-art metric for similarity-based evaluations, capturing semantic similarity more effectively and correlating highly with human judgments of functional correctness. The future of the field mainly lies in evaluation using functional correctness, primarily through `pass@k`, with newer, more granular approaches emerging, such as the introduced `latest-pass@n` metric.

Lastly, a significant contribution of this thesis is the introduction of `HUMANEVALPRO` (🔗), an enhanced version of the original `HumanEval` benchmark. `HUMANEVALPRO` addresses the many limitations of `HumanEval`:

- **Adding Type Annotations:** In `HUMANEVALPRO`, we introduced comprehensive type annotations across all problems. This inclusion not only provided essential context for each problem but also facilitated easier translation of benchmarks into multiple programming languages. Unlike the original benchmark, which only included type annotations for a limited subset of problems, `HUMANEVALPRO` ensures consistency and clarity throughout all 164 problems.

- **Enhanced Language Support for Frameworks:** Recognizing the importance of language diversity, HUMAN EVAL PRO significantly improved support for language conversion frameworks such as MultiPL-E. By standardizing tests to equality assertions wherever feasible, we reduced compatibility issues across various programming languages by a considerable margin.
- **Inclusion of Edge Cases:** Each problem in HUMAN EVAL PRO now includes carefully crafted edge cases. These additions, such as scenarios involving negative numbers, zero values, empty inputs, and non-alphanumeric characters, ensure that AI4SE models are rigorously tested against a broader spectrum of realistic conditions. This enhancement encourages models to handle diverse inputs akin to proficient software engineers.
- **Integration of Assertions:** Building on problem constraints specified in the descriptions, HUMAN EVAL PRO incorporates explicit assertions within the code. This practice underscores the importance of adhering to given constraints, thereby enriching the benchmark's evaluation criteria and promoting robust model performance.
- **Improvement of Test Examples:** We refined the test examples provided within each problem description to enhance evaluation quality. By optimizing the use of test examples and reducing their number in certain cases, we aimed to ensure a fair and unbiased assessment of model capabilities, aligning closely with current best practices in benchmark design.
- **Enhanced Descriptive Consistency:** HUMAN EVAL PRO underwent thorough editing to rectify spelling errors and maintain consistent formatting across all problem descriptions. This consistency not only improves readability but also ensures that the problem descriptions accurately reflect the intended implementations, fostering a more intuitive evaluation process for AI4SE models.
- **Increased Difficulty and Complexity:** By introducing more challenging problem variations and nuanced phrasing, HUMAN EVAL PRO raises the benchmark's overall difficulty level. These modifications address concerns regarding data leakage risks associated with the original benchmark and aim to mitigate inflated performance scores often observed in HumanEval leaderboards.

Most importantly, the more rigorous test cases (twice as many compared to the original) and edge cases offer a more accurate and challenging assessment of model performance on straightforward coding tasks. The results demonstrate substantial drops in pass@1 scores for 10 LLMs, with an average and median drop of **31.22%** and **26.02%**, respectively. These findings underscore the need for comprehensive and well-maintained benchmarks, with the rest of the thesis providing the necessary material for achieving this.

To summarize, the taxonomy of challenges and proposed solution directions, alongside the introduction of HUMAN EVAL PRO, mark a significant refinement in AI4SE benchmarking practices. Emphasizing these insights will be crucial for guiding future advancements in the field and fostering meaningful progress.

BIBLIOGRAPHY

- [1] R. Agashe, S. Iyer, and L. Zettlemoyer. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In K. Inui, J. Jiang, V. Ng, and X. Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446, Hong Kong, China, Nov. 2019. Association for Computational Linguistics.
- [2] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang. AVATAR: A parallel corpus for Java-python program translation. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2268–2281, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [3] A. Amini, S. Gabriel, P. Lin, R. Koncel-Kedziorski, Y. Choi, and H. Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. <https://arxiv.org/abs/1905.13319>, 5 2019.
- [4] ANTHROPIC. Introducing the next generation of Claude. <https://www.anthropic.com/news/claude-3-family>, 3 2024.
- [5] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xi-ang. Multi-lingual evaluation of code generation models. <https://arxiv.org/abs/2210.14868>, 10 2022.
- [6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program Synthesis with Large Language Models. <https://arxiv.org/abs/2108.07732>, 8 2021.
- [7] K. L. Aw, S. Montariol, B. Alkhamissi, M. Schrimpf, and A. Bosselut. Instruction-tuning aligns llms to the human brain. <https://arxiv.org/abs/2312.00575>, 2023.
- [8] S.-Y. Baik, M. Jeon, J. Hahn, J. Kim, Y.-S. Han, and S.-K. Ko. Codecomplex: A time-complexity dataset for bilingual source codes. <https://arxiv.org/abs/2401.08719>, 2024.
- [9] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi, and T. Bryksin. Long code arena: a set of benchmarks for long-context code models. <https://arxiv.org/abs/2406.11612>, 2024.
- [10] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. MULTIPL-E: a scalable and extensible approach to benchmarking neural code generation. <https://arxiv.org/abs/2208.08227>, 8 2022.
- [11] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan. Training and evaluating a jupyter notebook data science assistant. <https://arxiv.org/abs/2201.12901>, 1 2022.

- [12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. <https://arxiv.org/abs/2107.03374>, 7 2021.
- [13] W. Chen, M. Yin, M. Ku, P. Lu, Y. Wan, X. Ma, J. Xu, X. Wang, and T. Xia. TheoremQA: A theorem-driven question answering dataset. In H. Bouamor, J. Pino, and K. Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901, Singapore, Dec. 2023. Association for Computational Linguistics.
- [14] codeparrot. Instructhumaneval. <https://huggingface.co/datasets/codeparrot/instructhumaneval>, 6 2023.
- [15] cognition. Introducing Devin, the first AI software engineer. <https://www.cognition.ai/blog/introducing-devin>, 3 2024.
- [16] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). <https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2>, 2024.
- [17] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatta, D. Roth, and B. Xiang. CrossCodeEval: a diverse and multilingual benchmark for Cross-File code completion. <https://arxiv.org/abs/2310.11248>, 10 2023.
- [18] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin. CodeScore: Evaluating code generation by learning code execution. <https://arxiv.org/abs/2301.09043>, 1 2023.
- [19] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. ClassEval: a Manually-Crafted benchmark for evaluating LLMs on class-level code generation. <https://arxiv.org/abs/2308.01861>, 8 2023.
- [20] K. Ganesan. Rouge 2.0: Updated and improved measures for evaluation of summarization tasks. <https://arxiv.org/abs/1803.01937>, 2018.
- [21] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. PAL: Program-aided Language Models. <https://arxiv.org/abs/2211.10435>, 11 2022.
- [22] Gemini Team, Google. Gemini: A Family of Highly Capable Multimodal Models. https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf, 12 2023.
- [23] GitHub. Github. <https://github.com/>, 2008.
- [24] Google. Google drive. <https://drive.google.com>, 2012.
- [25] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. <https://arxiv.org/abs/2401.03065>, 2024.

- [26] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei. AixBench: a Code Generation Benchmark Dataset. <https://arxiv.org/abs/2206.13179>, 6 2022.
- [27] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. <https://arxiv.org/abs/2105.09938v3>, 5 2021.
- [28] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. <https://arxiv.org/abs/2308.10620>, 2024.
- [29] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC '18*, page 200–210, New York, NY, USA, May 2018. Association for Computing Machinery.
- [30] D. Huang, J. M. Zhang, Y. Qing, and H. Cui. Effibench: Benchmarking the efficiency of automatically generated code. <https://arxiv.org/abs/2402.02037v2>, 2 2024.
- [31] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan. CoSQA: 20,000+ web queries for code search and question answering. In C. Zong, F. Xia, W. Li, and R. Navigli, editors, *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, Aug. 2021. Association for Computational Linguistics.
- [32] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18*, page 293–304, New York, NY, USA, 2018. Association for Computing Machinery.
- [33] Hugging Face, Inc. Hugging face. <https://huggingface.co>, 2016.
- [34] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. CodeSearchNet Challenge: Evaluating the state of semantic code Search. <https://arxiv.org/abs/1909.09436>, 9 2019.
- [35] IBM CODAIT. AI for Code: Predict Code complexity using IBM's CodeNet Dataset. <https://community.ibm.com/community/user/datascience/blogs/sepideh-seifzadeh1/2021/10/05/ai-for-code-predict-code-complexity-using-ibms-cod>, 10 2021.
- [36] InfiCoderTeam. Infocoder-eval: Systematically evaluating question-answering for code large language models. <https://infi-coder.github.io/infocoder-eval/>, 2023.
- [37] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2018.
- [38] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, oct 2002.
- [39] C. E. Jiménez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-Bench: Can language models resolve Real-World GitHub Issues? *arXiv (Cornell University)*, 10 2023.

- [40] X. Jin, J. Larson, W. Yang, and Z. Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. <https://arxiv.org/abs/2312.09601>, 12 2023.
- [41] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, page 437–440, New York, NY, USA, 7 2014. Association for Computing Machinery.
- [42] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. <https://arxiv.org/abs/2303.03004>, 3 2023.
- [43] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries. The stack: 3 tb of permissively licensed source code. <https://arxiv.org/abs/2211.15533>, 2022.
- [44] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [45] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W.-T. Yih, D. Fried, S. Wang, and Y. Chen. DS-1000: A natural and reliable benchmark for data science code Generation. *arXiv (Cornell University)*, 11 2022.
- [46] B. Li, W. Wu, Z. Tang, L. Shi, J. Yang, J. Li, S. Yao, C. Qian, B. Hui, Q. Zhang, Z. Yu, H. Du, P. Yang, D. Lin, C. Peng, and K. Chen. Devbench: A comprehensive benchmark for software development. <https://arxiv.org/abs/2403.08604>, 2024.
- [47] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. <https://arxiv.org/abs/2404.00599>, 2024.
- [48] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, and Y. Li. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. <https://arxiv.org/abs/2405.19856>, 2024.
- [49] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li. Api-bank: A comprehensive benchmark for tool-augmented llms. <https://arxiv.org/abs/2304.08244>, 4 2023.
- [50] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. De Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. De Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 12 2022.
- [51] C.-Y. Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

- [52] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. Technical report, 2018.
- [53] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. <https://arxiv.org/abs/2305.01210>, 5 2023.
- [54] T. Liu, C. Xu, and J. McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. <https://arxiv.org/abs/2306.03091>, 6 2023.
- [55] P. Lu, L. Qiu, W. Yu, S. Welleck, and K.-W. Chang. A survey of deep learning for mathematical reasoning. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14605–14631, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [56] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. <http://arxiv.org/abs/2102.04664>, 2 2021.
- [57] Microsoft. Microsoftdocs. <https://github.com/MicrosoftDocs/>, 2024. Accessed: Feb 2024.
- [58] S. Mishra, M. Finlayson, P. Lu, L. Tang, S. Welleck, C. Baral, T. Rajpurohit, O. Tafjord, A. Sabharwal, P. Clark, and A. Kalyan. LILA: A unified benchmark for mathematical reasoning. In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5807–5832, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics.
- [59] K. Moudgalya, A. Ramakrishnan, V. Chemudupati, and X. H. Lu. Tasty: A transformer based approach to space and time complexity. <https://arxiv.org/abs/2305.05379>, 5 2023.
- [60] D. Mouriquand. ‘AI’ named Word of the Year by Collins Dictionary. 12 2023.
- [61] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre. OctoPack: Instruction Tuning Code Large Language Models. <https://arxiv.org/abs/2308.07124>, 8 2023.
- [62] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. <https://arxiv.org/abs/2203.13474>, 3 2023.
- [63] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, 2015.
- [64] OpenAI. GPT-4 Technical Report. <https://openai.com/research/gpt-4>, 3 2023.
- [65] OpenAI. GPT-4V(ision). <https://openai.com/research/gpt-4v-system-card>, 9 2023. PDF accessible at https://cdn.openai.com/papers/GPTV_System_Card.pdf.

- [66] OpenAI. Hello GPT-4o. <https://www.anthropic.com/news/claude-3-family>, 5 2024.
- [67] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In P. Isabelle, E. Charniak, and D. Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.
- [68] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis. <https://arxiv.org/abs/2305.15334>, 5 2023.
- [69] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. Lyu. Revisiting, benchmarking and exploring api recommendation: How far are we? <https://arxiv.org/abs/2112.12653>, 2021.
- [70] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. <https://arxiv.org/abs/2009.10297>, 2020.
- [71] S. Roy and D. Roth. Solving general arithmetic word problems. In L. Màrquez, C. Callison-Burch, and J. Su, editors, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1743–1752, Lisbon, Portugal, 9 2015. Association for Computational Linguistics.
- [72] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample. Leveraging automated unit tests for unsupervised code translation. <https://arxiv.org/abs/2110.06773>, 2022.
- [73] B. Shen and N. Meng. Conflictbench: A benchmark to evaluate software merge tools. *Journal of Systems and Software*, 214:112084, 2024.
- [74] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes. Using large language models to generate junit tests: An empirical study. <https://arxiv.org/abs/2305.00418>, 2024.
- [75] J. Sikka, K. Satya, Y. Kumar, S. Uppal, R. R. Shah, and R. Zimmermann. Learning based methods for code runtime complexity prediction. <https://arxiv.org/abs/1911.01155>, 11 2019.
- [76] A. Silva, N. Saavedra, and M. Monperrus. Gitbug-java: A reproducible benchmark of recent java bugs. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 118–122, 2024.
- [77] Y. Song, W. Xiong, D. Zhu, W. Wu, H. Qian, M. Song, H. Huang, C. Li, K. Wang, R. Yao, Y. Tian, and S. Li. RestGPT: Connecting Large Language Models with Real-World RESTful APIs. <https://arxiv.org/abs/2306.06624>, 6 2023.
- [78] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, A. Kluska, A. Lewkowycz, A. Agarwal, A. Power, A. Ray, A. Warstadt, A. W. Kocurek, A. Safaya, A. Tazarv, A. Xiang, A. Parrish, A. Nie, A. Hussain, A. Askell, A. Dsouza, A. Slone, A. Rahane, A. S. Iyer, A. Andreassen, A. Madotto, A. Santilli, A. Stuhlmüller, A. Dai, A. La, A. Lampinen, A. Zou, A. Jiang, A. Chen, A. Vuong, A. Gupta, A. Gottardi, A. Norelli, A. Venkatesh,

A. Gholamidavoodi, A. Tabassum, A. Menezes, A. Kirubarajan, A. Mullokandov, A. Sabharwal, A. Herrick, A. Efrat, A. Erdem, A. Karakaş, B. R. Roberts, B. S. Loe, B. Zoph, B. Bojanowski, B. Özyurt, B. Hedayatnia, B. Neyshabur, B. Inden, B. Stein, B. Ekmekci, B. Y. Lin, B. Howald, B. Orinion, C. Diao, C. Dour, C. Stinson, C. Argueta, C. F. Ramírez, C. Singh, C. Rathkopf, C. Meng, C. Baral, C. Wu, C. Callison-Burch, C. Waites, C. Voigt, C. D. Manning, C. Potts, C. Ramirez, C. E. Rivera, C. Siro, C. Raffel, C. Ashcraft, C. Garbacea, D. Sileo, D. Garrette, D. Hendrycks, D. Kilman, D. Roth, D. Freeman, D. Khashabi, D. Levy, D. M. González, D. Perszyk, D. Hernandez, D. Chen, D. Ippolito, D. Gilboa, D. Dohan, D. Drakard, D. Jurgens, D. Datta, D. Ganguli, D. Emelin, D. Kleyko, D. Yuret, D. Chen, D. Tam, D. Hupkes, D. Misra, D. Buzan, D. C. Mollo, D. Yang, D.-H. Lee, D. Schrader, E. Shutova, E. D. Cubuk, E. Segal, E. Hagerman, E. Barnes, E. Donoway, E. Pavlick, E. Rodola, E. Lam, E. Chu, E. Tang, E. Erdem, E. Chang, E. A. Chi, E. Dyer, E. Jerzak, E. Kim, E. E. Manyasi, E. Zheltonozhskii, F. Xia, F. Siar, F. Martínez-Plumed, F. Happé, F. Chollet, F. Rong, G. Mishra, G. I. Winata, G. de Melo, G. Kruszewski, G. Parascandolo, G. Mariani, G. Wang, G. Jaimovitch-López, G. Betz, G. Gur-Ari, H. Galijasevic, H. Kim, H. Rashkin, H. Hajishirzi, H. Mehta, H. Bogar, H. Shevlin, H. Schütze, H. Yakura, H. Zhang, H. M. Wong, I. Ng, I. Noble, J. Jumelet, J. Geissinger, J. Kernion, J. Hilton, J. Lee, J. F. Fisac, J. B. Simon, J. Koppel, J. Zheng, J. Zou, J. Kocoń, J. Thompson, J. Wingfield, J. Kaplan, J. Radom, J. Sohl-Dickstein, J. Phang, J. Wei, J. Yosinski, J. Novikova, J. Bosscher, J. Marsh, J. Kim, J. Taal, J. Engel, J. Alabi, J. Xu, J. Song, J. Tang, J. Waweru, J. Burden, J. Miller, J. U. Balis, J. Batchelder, J. Berant, J. Frohberg, J. Rozen, J. Hernandez-Orallo, J. Boudeman, J. Guerr, J. Jones, J. B. Tenenbaum, J. S. Rule, J. Chua, K. Kanclerz, K. Livescu, K. Krauth, K. Gopalakrishnan, K. Ignatyeva, K. Markert, K. D. Dhole, K. Gimpel, K. Omondi, K. Mathewson, K. Chiafullo, K. Shkaruta, K. Shridhar, K. McDonell, K. Richardson, L. Reynolds, L. Gao, L. Zhang, L. Dugan, L. Qin, L. Contreras-Ochando, L.-P. Morency, L. Moschella, L. Lam, L. Noble, L. Schmidt, L. He, L. O. Colón, L. Metz, L. K. Şenel, M. Bosma, M. Sap, M. ter Hoeve, M. Farooqi, M. Faruqui, M. Mazeika, M. Baturan, M. Marelli, M. Maru, M. J. R. Quintana, M. Tolkiehn, M. Giulianelli, M. Lewis, M. Potthast, M. L. Leavitt, M. Hagen, M. Schubert, M. O. Baitemirova, M. Arnaud, M. McElrath, M. A. Yee, M. Cohen, M. Gu, M. Ivanitskiy, M. Starritt, M. Strube, M. Swędrowski, M. Bevilacqua, M. Yasunaga, M. Kale, M. Cain, M. Xu, M. Suzgun, M. Walker, M. Tiwari, M. Bansal, M. Aminnaseri, M. Geva, M. Gheini, M. V. T. N. Peng, N. A. Chi, N. Lee, N. G.-A. Krakover, N. Cameron, N. Roberts, N. Doiron, N. Martinez, N. Nangia, N. Deckers, N. Muennighoff, N. S. Keskar, N. S. Iyer, N. Constant, N. Fiedel, N. Wen, O. Zhang, O. Agha, O. Elbaghdadi, O. Levy, O. Evans, P. A. M. Casares, P. Doshi, P. Fung, P. P. Liang, P. Vicol, P. Alipoormolabashi, P. Liao, P. Liang, P. Chang, P. Eckersley, P. M. Htut, P. Hwang, P. Miłkowski, P. Patil, P. Pezeshkpour, P. Oli, Q. Mei, Q. Lyu, Q. Chen, R. Banjade, R. E. Rudolph, R. Gabriel, R. Habacker, R. Risco, R. Millière, R. Garg, R. Barnes, R. A. Saurous, R. Arakawa, R. Raymaekers, R. Frank, R. Sikand, R. Novak, R. Sitelew, R. LeBras, R. Liu, R. Jacobs, R. Zhang, R. Salakhutdinov, R. Chi, R. Lee, R. Stovall, R. Teehan, R. Yang, S. Singh, S. M. Mohammad, S. Anand, S. Dillavou, S. Shleifer, S. Wiseman, S. Gruetter, S. R. Bowman, S. S. Schoenholz, S. Han, S. Kwatra, S. A. Rous, S. Ghazarian, S. Ghosh, S. Casey, S. Bischoff, S. Gehrmann, S. Schuster, S. Sadeghi, S. Hamdan, S. Zhou, S. Srivastava, S. Shi, S. Singh, S. Asaadi, S. S. Gu, S. Pachchigar, S. Toshniwal, S. Upadhyay, Shyamolima, Debnath, S. Shakeri, S. Thormeyer, S. Melzi, S. Reddy, S. P. Makini, S.-H. Lee, S. Torene, S. Hatwar, S. Dehaene, S. Divic, S. Ermon, S. Biderman, S. Lin, S. Prasad, S. T. Piantadosi, S. M. Shieber, S. Mishnerghi, S. Kiritchenko, S. Mishra, T. Linzen, T. Schuster, T. Li, T. Yu, T. Ali, T. Hashimoto, T.-L. Wu, T. Desbordes, T. Rothschild, T. Phan, T. Wang, T. Nkinyili, T. Schick, T. Kornev, T. Tunduny, T. Gerstenberg, T. Chang, T. Neeraj, T. Khot, T. Shultz, U. Shahan, V. Misra, V. Demberg, V. Nyamai, V. Raunak, V. Ramasesh, V. U. Prabhu, V. Padmakumar, V. Srikumar, W. Fedus, W. Saunders, W. Zhang, W. Vossen, X. Ren, X. Tong, X. Zhao, X. Wu, X. Shen,

- Y. Yaghoobzadeh, Y. Lakretz, Y. Song, Y. Bahri, Y. Choi, Y. Yang, Y. Hao, Y. Chen, Y. Belinkov, Y. Hou, Y. Hou, Y. Bai, Z. Seid, Z. Zhao, Z. Wang, Z. J. Wang, Z. Wang, and Z. Wu. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models, 6 2022.
- [79] Stack Exchange, Inc. Stack overflow. <https://stackoverflow.com/>, 2008.
- [80] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan. Unit Test Case Generation with Transformers and Focal Context. <https://arxiv.org/abs/2009.05617>, 9 2020.
- [81] T. van Dam, F. van der Heijden, P. de Bekker, B. Nieuwschepen, M. Otten, and M. Izadi. Investigating the performance of language models for completing code in functional programming languages: a haskell case study. <https://arxiv.org/abs/2403.15185>, 2024.
- [82] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, and B. Xiang. ReCode: Robustness Evaluation of code generation models. <https://arxiv.org/abs/2212.10264>, 12 2022.
- [83] Z. Wang, G. Cuenca, S. Zhou, F. F. Xu, and G. Neubig. MCoNaLa: A Benchmark for Code Generation from Multiple Natural Languages. <https://arxiv.org/abs/2203.08388>, 3 2022.
- [84] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried. Coderag-bench: Can retrieval augment code generation? <https://arxiv.org/abs/2406.14497>, 2024.
- [85] C. S. Xia, Y. Deng, and L. Zhang. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. <https://arxiv.org/abs/2403.19114>, 4 2024.
- [86] A. Yadav and M. Singh. Pythonsaga: Redefining the benchmark to evaluate code generating llm. <https://arxiv.org/abs/2401.03855v2>, 1 2024.
- [87] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. <https://arxiv.org/abs/2310.04951>, 2023.
- [88] S. Yeo, Y. Ma, S. C. Kim, H. Jun, and T. Kim. Framework for evaluating code generation ability of large language models. *ETRI Journal*, 46(1):106–117, Feb. 2024.
- [89] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 476–486, New York, NY, USA, 2018. Association for Computing Machinery.
- [90] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Y. Liang, Y. Liu, T. Xie, and Q. Wang. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv (Cornell University)*, 2 2023.
- [91] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev. Spider: a Large-Scale Human-Labeled dataset for complex and Cross-Domain semantic parsing and Text-to-SQL task. <https://arxiv.org/abs/1809.08887>, 9 2018.
- [92] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. <https://arxiv.org/abs/2306.16171>, 2023.

- [93] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou. Cert: Continual pre-training on sketches for library-oriented code generation. <https://arxiv.org/abs/2206.06888>, 6 2022.
- [94] M. Zavershynskiy, A. Skidanov, and I. Polosukhin. NAPS: Natural Program Synthesis Dataset. <https://arxiv.org/abs/1807.03168>, 7 2018.
- [95] Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang. Coderujb: An executable and unified java benchmark for practical programming scenarios. <https://arxiv.org/abs/2403.19287>, 3 2024.
- [96] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. RepoCoder: Repository-Level Code completion through iterative retrieval and generation. <https://arxiv.org/abs/2303.12570>, 3 2023.
- [97] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. Bertscore: Evaluating text generation with bert. <https://arxiv.org/abs/1904.09675>, 2020.
- [98] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. <https://arxiv.org/abs/2404.05427>, 2024.
- [99] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. <http://arxiv.org/abs/2303.17568>, 3 2023.
- [100] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. <https://arxiv.org/abs/1709.00103>, 8 2017.
- [101] S. Zhou, U. Alon, S. Agarwal, and G. Neubig. Codebertscore: Evaluating code generation with pretrained models of code. <https://arxiv.org/abs/2302.05527>, 2023.
- [102] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. <https://arxiv.org/abs/2206.08474>, 6 2022.
- [103] M. Zhu, K. Suresh, and C. K. Reddy. Multilingual code snippets training for program translation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):11783–11790, June 2022.
- [104] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. V. Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. <https://arxiv.org/abs/2406.15877>, 2024.

A

ILLUSTRATION OF SHORTCOMINGS IN HUMAN EVAL

This appendix illustrates some shortcomings of HumanEval [12]. Note, this is not an exhaustive list of all problems, rather it is a list to indicate that important issues exist in (popular) benchmarks.

Some of the main findings during the analysis of the raw HumanEval data are as follows:

- Incorrect (example) test - see [Figure A.1](#). Notably, ChatGPT, made by the authors of the benchmark, produces the same mistake in production, as depicted in [Figure A.2](#).
- Incorrect canonical solution and lack of proper test coverage - see [Figure A.3](#).
- Mismatch (or lack of clarity) in docstring and canonical solution - see [Figure A.4](#).

Furthermore, these issues can be found in the altered versions of this benchmark as well, see [Table 4.1](#) for a detailed description of the issues per benchmark. This strongly indicates that in general, no proper quality checks are taking place within the field of AI4SE benchmarks.

```
1 def median(l: list):
2     """
3     Return median of elements in the list l.
4     >>> median([3, 1, 2, 4, 5])
5     3
6     >>> median([-10, 4, 6, 1000, 10, 20])
7     15.0
8     """
```

Figure A.1: Prompt of Problem #47 of HumanEval [12]. The docstring contains an incorrect example. The median of $[-10, 4, 6, 1000, 10, 20]$ should be 8 instead of 15, as 6 and 10 are the two middle values in this list of even length – not 10 and 20.

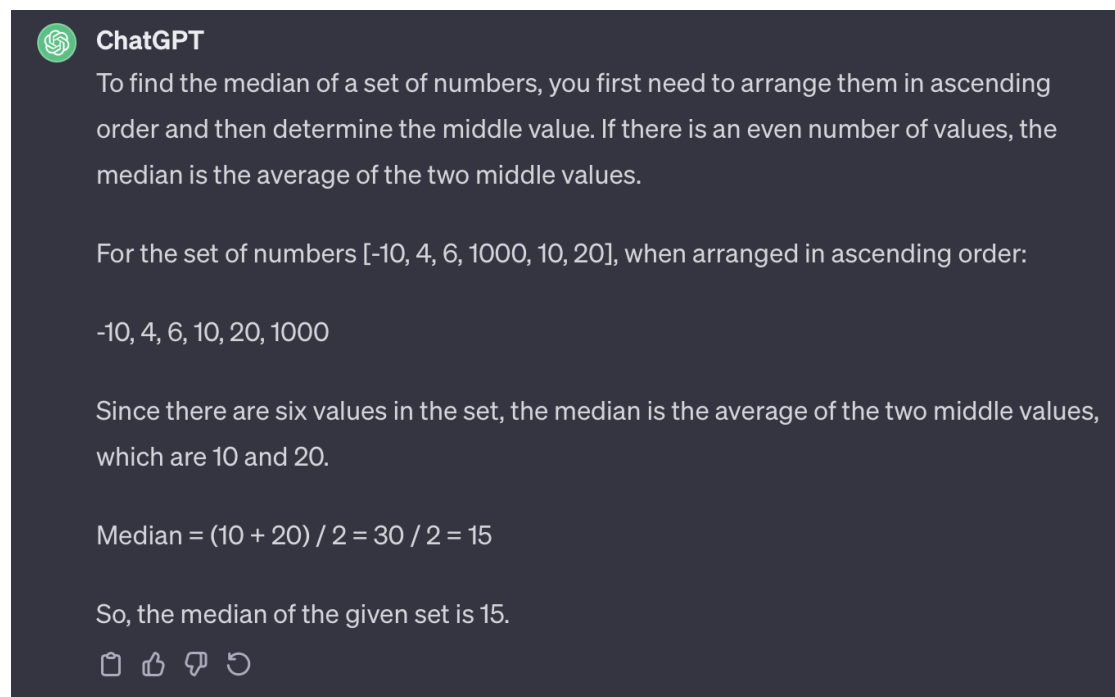


Figure A.2: Production system (ChatGPT-3.5, 6 Dec 2023) providing an incorrect response exactly similar to the mistake included in the HumanEval benchmark [12], as seen in [Figure A.1](#). The prompt for this response was: “What is the median of [-10, 4, 6, 1000, 10, 20]?”. This emphasizes the critical role of high-quality training data in developing AI4SE models and underscores the importance of maintaining accurate benchmarks to avoid misleading conclusions.

```

1 def check_dict_case(dict):
2     """
3     Given a dictionary, return True if all keys are strings in lower
4     case or all keys are strings in upper case, else return False.
5     The function should return False if the given dictionary is empty.
6     Examples:
7     check_dict_case({"a":"apple", "b":"banana"}) should return True.
8     check_dict_case({"a":"apple", "A":"banana", "B":"banana"}) should return
9     False.
10    check_dict_case({"a":"apple", 8:"banana", "a":"apple"}) should return
11    False.
12    check_dict_case({"Name":"John", "Age":"36", "City":"Houston"}) should
13    return False.
14    check_dict_case({"STATE":"NC", "ZIP":"12345" }) should return True.
15    """
16    if len(dict.keys()) == 0:
17        return False
18    else:
19        state = "start"
20        for key in dict.keys():
21
22            if isinstance(key, str) == False:
23                state = "mixed"
24                break
25            if state == "start":
26                if key.isupper():
27                    state = "upper"
28                elif key.islower():
29                    state = "lower"
30            else:
31                break
32            elif (state == "upper" and not key.isupper()) or (state ==
33            "lower" and not key.islower()):
34                state = "mixed"
35                break
36        else:
37            break
38    return state == "upper" or state == "lower"

```

Figure A.3: Prompt and canonical solution of Problem #95 of HumanEval [12]. The function needs to check if all keys are upper or lower case in a dictionary (data structure with keys mapped to values), yet it breaks in line 31 (instead of `continue`) when the state variable is "upper" or "lower" and the current key matches this capitalization. Instead of `continue`ing to check the rest of the keys, the function breaks and returns true, which could be wrong, as later keys might have incorrect capitalization (meaning the state would have been "mixed"). For example, the following input yields True: `check_dict_case({"abcd":"d", "abce":"e", "abcF":"f"})`, as the first two keys, i.e. `abcd` and `abce`, result in a "lower" state and then break out of the program in the incorrect `else`-statement. Meanwhile, the dictionary contained inconsistent capitalization, indicating a lack of test coverage or problematic test generation for enhanced variations of the benchmark using the canonical solution as a way to generate the "correct" output.

```

1 def generate_integers(a, b):
2     """
3     Given two positive integers a and b, return the even digits between a
4     and b, in ascending order.
5
6     For example:
7     generate_integers(2, 8) => [2, 4, 6, 8]
8     generate_integers(8, 2) => [2, 4, 6, 8]
9     generate_integers(10, 14) => []
10    """
11    lower = max(2, min(a, b))
12    upper = min(8, max(a, b))
13
14    return [i for i in range(lower, upper+1) if i % 2 == 0]

```

Figure A.4: Prompt and canonical solution of Problem #163 of HumanEval [12]. The description within the docstring does not match the implementation or example tests. In addition, the problem lacks proper test coverage. Firstly, the description mentions *between* the arguments a and b, yet the implementation and example tests include a and b in the output. Instead of `range(lower, upper+1)`, where the first argument (start) is included and the second (stop) is excluded from the range, the implementation should use `range(lower+1, upper)`. Furthermore, 0 is considered to be a digit (0–9), yet the implementation does not account for this. Given two positive integers a and b where the numbers in between cross 0, e.g. `range(8, 12)` crosses 10, the implementation should include 0 in the output based on the description. Lastly, as the implementation mentions the digits between a and b, it should consider all the digits of the numbers in between (or the instruction lacks clarity in this regard). Digits are not only the last digit of a number. For example, number 2024 contains digits 0, 2, and 4. Hence, the example test `generate_integers(10, 14) => []` could be deemed mistaken compared to the instruction, as the numbers in between, i.e. 11, 12, and 13, contain the even digit 2, meaning the output should be `[2]`.

COLOPHON

This document was typeset using \LaTeX . The document layout was generated using a manually edited version of the MSc Geomatics Thesis Template on Overleaf by Hugo Ledoux¹, which is based on the `arclassica` package by Lorenzo Pantieri, which is an adaption of the original `classicthesis` package from André Miede.

¹ <https://www.overleaf.com/latex/templates/msc-geomatics-thesis-template-tu-delft/yvjkwvtkrwz>

