Investigating Reproducibility Challenges in LLM Bugfixing on the HumanEvalFix Benchmark

Balázs Szalontai , Balázs Márton , Balázs Pintér * , Tibor Gregorics

Posted Date: 29 May 2025

doi: 10.20944/preprints202505.2321.v1

Keywords: reproducibility; bugfixing; humanevalfix; ml4code

 doi:10.20944/preprints202505.2321.v1

Article

Investigating Reproducibility Challenges in LLM Bugfixing on the HumanEvalFix Benchmark

Balázs Szalontai, Balázs Márton, Balázs Pintér * and Tibor Gregorics

Department of Software Technology, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
* Correspondence: pinter@inf.elte.hu

Abstract: Benchmark results for Large Language Models often show inconsistencies across different studies. This paper investigates the challenges of reproducing these results in automatic bugfixing using LLMs, on the HumanEvalFix benchmark. To determine the cause of the differing results in the literature, we attempted to reproduce a subset of them by evaluating 11 models in the DeepSeekCoder, CodeGemma, and CodeLlama model families, in different sizes and tunings. A total of 32 unique results were reported for these models across studies, of which we successfully reproduced 16. We identified several relevant factors that influence the results. Base models can be confused with their instruction-tuned variants, making their results better than expected. Incorrect prompt templates or generation length can decrease benchmark performance, as well as using 4-bit quantization. Using sampling instead of greedy decoding can increase variance, especially with higher temperature values. We found that precision and 8-bit quantization have less influence on benchmark results.

Keywords: reproducibility; bugfixing; humanevalfix; ml4code

1. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in software development [1–3], which includes their ability to detect and resolve bugs. Various benchmarks are designed to evaluate these capabilities, such as QuixBugs [4], HumanEvalFix [1], BugsInPy [5] and MDEVAL [6].

The most widely acknowledged bugfix benchmark is probably HumanEvalFix, a component of the HumanEvalPack [1]. In this benchmark, each of the 164 canonical solutions from the original HumanEval code generation benchmark [7] has been manually corrupted by introducing a bug. The task of the LLM is to repair the buggy function. The modified output is then verified using the original test cases.

Although reliable reports of benchmark results are essential to compare models, it is not uncommon to see differing benchmark results for the same model throughout studies [8–10]. In the broader community, researchers can struggle to reproduce the officially reported benchmark results. The difference between reported and reproduced results can be minor, but it can also be significant. This not only complicates comparison of models but also questions the validity of the currently published benchmark results.

In this paper, we address the issue of inconsistent benchmark results on the HumanEvalFix benchmark. We look for discrepancies in a broad range of results published in the literature, considering models with scores reported in two or more studies. To identify the underlying reasons of these discrepancies, we conduct our own evaluations using 11 models of various sizes and tunings: DeepSeekCoder (1.3B and 6.7B, base and instruct) [3], CodeLlama (7B and 13B, base and instruct) [2], and CodeGemma (2B base, 7B base and instruct) [11]. Our findings are as follows:

- We reproduce 16 of the 32 scores reported for the evaluated models. This implies that many of the discrepancies in reported results can be attributed to using different, or possibly incorrect evaluation settings.

 doi:10.20944/preprints202505.2321.v1

- We quantify the impact of modifying evaluation settings individually. For instance, maximum generation length, sampling strategy, 4-bit quantization, and prompt template choice significantly influence the results, whereas precision and 8-bit quantization do not.
- We identify instances of possibly misreported results, likely due to confusion between base and instruction-tuned models.

2. Method

We begin by introducing the benchmark we focus on, HumanEvalFix, as well as the evaluation framework commonly used to run it. Then we identify the discrepancies across reported results by conducting a literature review of the benchmark scores. Finally, we analyze GitHub issues about unsuccessful reproduction of published results to determine possible reasons for these discrepancies and our experimental setup.

2.1. Benchmarking with HumanEvalFix

The authors of the HumanEvalFix benchmark have introduced a bug into each of the 164 canonical solutions in the HumanEval benchmark, thereby turning it into a bugfixing benchmark. These bugs involve missing logic, extra logic, or incorrect logic. The benchmarked LLM is prompted with the incorrect function, along with unit tests, and is tasked with repairing the function. The output code is then validated using the same unit tests found in the original HumanEval benchmark. Since the Python version of this benchmark is the most widely used, we will also focus on this variant.

For conducting our evaluations, we use the *Code Generation LM Evaluation Harness* [12] framework, provided by the *BigCode Project*. This framework includes multiple benchmark implementations, such as HumanEval, MBPP, and QuixBugs, as well as the HumanEvalFix benchmark. Most of these benchmarks evaluate functional correctness of code outputs using the pass@k metric, which indicates the percentage of outputs that pass the tests at least once in k attempts. Users of the framework can adjust a range of evaluation settings, such as the maximum generation length, batch size, and sampling parameters.

2.2. Review of Reported Benchmark Results

We conduct a review of 18 papers that contain results of the HumanEvalFix benchmark [1,13–28], focusing on their reported HumanEvalFix results. These papers report benchmark results for a total of 117 different models (including variants in sizes and tunings). Out of the 117 models mentioned, 26 appeared in at least two papers. These models form the basis of our investigation. Their reported results are visualized in Figure 1, as well as in Table 1 with all relevant citations.

Six out of 26 models have identical reported results across studies (e.g. InstructCodeT5+, StarCoder). This suggests that the initially published result was cited by other authors without trying to reproduce it. For **4**/26 models, results differ only slightly (e.g. OctoCoder, Llama3-Inst-8B). Small differences are accepted in many cases as they can be due to minor differences in configuration settings. However for most of the models, more precisely in **16**/26 cases, the results differ significantly (e.g. CodeLlama, DeepSeekCoder). For example, the reported results for DeepSeekCoder-Instruct (1.3B) range from 9.1% to 48.9%. Similarly, for CodeLlama-Instruct (13B), six benchmark results are reported, ranging from 15.2% to 30.5%, nearly evenly spread. According to one reported result, DeepSeekCoder-Instruct (33B) should be one of the top models for bugfixing with an exceptionally high score of 81.0%, yet two other studies report a much lower score of 47.5%.

 doi:10.20944/preprints202505.2321.v1

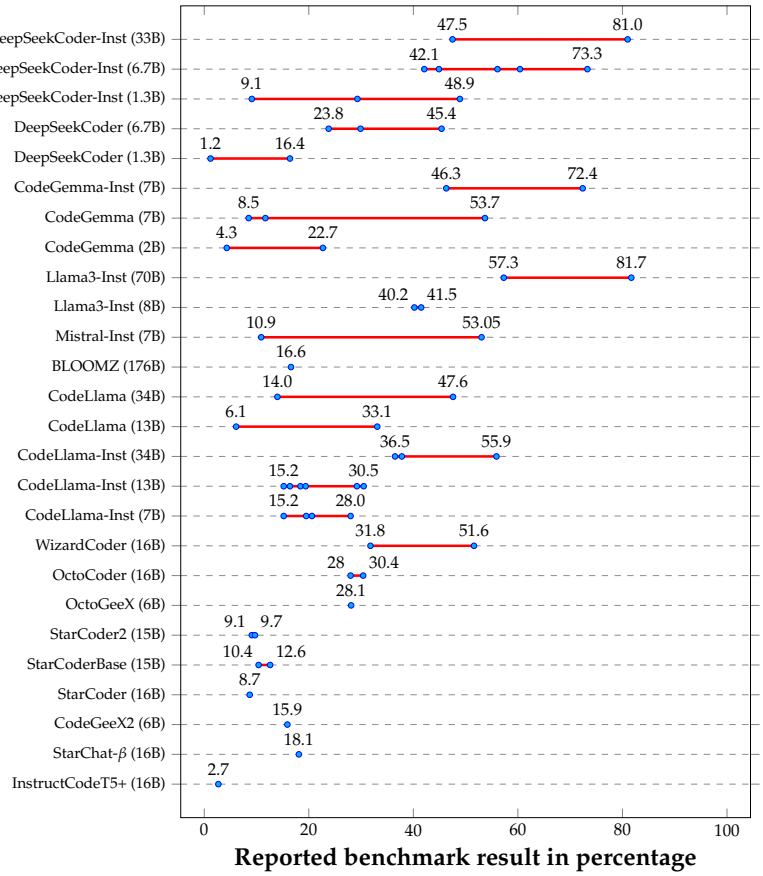


Figure 1. Reported HumanEvalFix results of LLMs. Only models with reported results from at least 2 papers are included. The labeled blue dots represent the results. Red lines are used in between results to visualize the discrepancy between the minimum and maximum reported score. In the original papers, multiple prompts were used for StarCoderBase and StarCoder2; however, we include only the results from the instruct prompt, as this is the one used by other authors.

As the benchmark dataset size is exactly 164, the benchmark results should be percentages that are calculated through dividing by 164. Since in most cases this is an infinitely long fraction, reported values are usually rounded or floored to one or two decimals. Notably, we have observed that in certain cases, the reported result cannot be obtained either by flooring or rounding. For example, the reported pass@1 score of 44.9% (for DeepSeekCoder-Inst 6.7B) cannot be correct: if 73 fixes were successful, the score would be 44.51%, and if 74 fixes were successful, it would be 45.12%. Throughout the reported scores across the analyzed papers (from Figure 1 and Table 1), 34 reports have such an unobtainable score out of 85. Considering the unique scores only, 20 scores have this property out of 57. Such scores are hard to explain. They could be a result of a simple mistake, such as inaccurate rounding/flooring, or a more severe one, such as confusion with a different benchmark score.

 doi:10.20944/preprints202505.2321.v1

Table 1. All reported HumanEvalFix results of LLMs considered in this paper. Citations are provided for studies introducing models and for ones that evaluate them.

Model	Reported results
InstructCodeT5+ (16B) [29]	**2.7** [1,18,22]
StarChat-β (16B) [30]	**18.1** [1,22]
CodeGeeX2 (6B) [31]	**15.9** [1,18,22]
StarCoder (16B) [32]	**8.7** [1,18,22,28]
StarCoderBase (15B) [32]	**10.4** [21], **12.6** [20,23]
StarCoder2 (15B) [20]	**9.1** [21], **9.7** [20,23]
OctoGeeX (6B) [1]	**28.1** [1,22]
OctoCoder (16B) [1]	**28** [21], **30.4** [1,18,20,22,23,28]
WizardCoder (16B) [33]	**31.8** [1,18,22,28], **51.6** [25]
CodeLlama-Inst (7B) [2]	**15.2** [22], **19.5** [21], **20.6** [25], **28.0** [28]
CodeLlama-Inst (13B) [2]	**15.2** [18], **16.4** [22], **18.9** [21], **19.4** [20], **29.2** [28], **30.5** [25]
CodeLlama-Inst (34B) [2]	**36.5** [13,20], **37.8** [21], **55.9** [25]
CodeLlama (13B) [2]	**6.1** [21], **33.1** [25]
CodeLlama (34B) [2]	**14.0** [21], **47.6** [25]
BLOOMZ (176B) [34]	**16.6** [1,18,23]
Mistral-Inst (7B) [35]	**10.9** [25], **53.05** [16]
Llama3-Inst (8B) [36]	**40.2** [21], **41.5** [25]
Llama3-Inst (70B) [36]	**57.3** [21], **81.7** [25]
CodeGemma (2B) [11]	**4.3** [21], **22.7** [25]
CodeGemma (7B) [11]	**8.5** [21], **11.7** [25], **53.7** [14]
CodeGemma-Inst (7B) [11]	**46.3** [21], **72.4** [25]
DeepSeekCoder (1.3B) [3]	**1.2** [19], **16.4** [25]
DeepSeekCoder (6.7B) [3]	**23.8** [19], **29.9** [28], **45.4** [25]
DeepSeekCoder-Inst (1.3B) [3]	**9.1** [13], **29.3** [19], **48.9** [25]
DeepSeekCoder-Inst (6.7B) [3]	**42.1** [19], **44.9** [13,20], **56.1** [28], **60.4** [14], **73.3** [25]
DeepSeekCoder-Inst (33B) [3]	**47.5** [13,20], **81.0** [25]

2.3. Experimental Setup

The goal of our experiments is twofold. First, we examine the effect of a wide range of settings on the results of models mentioned in multiple studies. Then, we try to reproduce results reported in the literature by finding out the evaluation settings they could have used. In this section, we determine the experimental setup for the evaluations.

To explore potential causes of differences in benchmark results, we reviewed the reported issues about unsuccessful attempts at reproducing published results in the GitHub repository of the benchmarking framework[1]. At the time of investigation, 145 issues were published (47 open and 98 closed)[2]. Thirteen out of the 145 issues (6 open and 7 closed) were about unsuccessful reproduction of officially reported results. We analyzed the issues that were either closed or had helpful answers to them (10 in total), and found these causes:

- A differently tuned version of the model was used (e.g. the base model instead of its instruction-tuned variant), causing a large difference in benchmark result. (2 issues)
- The wrong variant of the same benchmark was used: MBPP instead of MBPP+, and HumanEval instead of its unstripped variant. (2 issues)
- The temperature was set incorrectly: while generating one sample for each prompt, the temperature $T = 0.8$ turned out to be an improperly large value. (1 issue)
- An incorrect prompt was used, causing more than 6% of a decrease in pass@1 performance. (1 issue)

[1] https://github.com/bigcode-project/bigcode-evaluation-harness
[2] We reviewed these issues on the 5th of November (2024)

 doi:10.20944/preprints202505.2321.v1

- The reproduced results were different by only a few percentage points compared to the officially reported ones. Such a discrepancy was considered negligible, as this can be due to minor variations in evaluation settings, minor updates in model versions, or in hardware configurations. (4 issues)

We chose 11 models for evaluation, most of which have multiple reported results across papers. We evaluated these models on the HumanEvalFix benchmark, in different settings. The models used are: DeepSeekCoder (1.3B and 6.7B), DeepSeekCoder-Instruct (1.3B and 6.7B), CodeLlama (7B and 13B), CodeLlama-Instruct (7B and 13B), CodeGemma (2B and 7B) and CodeGemma-Instruct (7B). In our experiments, we consider these evaluation settings and values:

Prompt template The evaluation framework defaults to the *instruct* prompt template, with only the context and instruction (the program and the instruction to fix it). The default prompt template lacks model-specific formatting suggested by model authors. We evaluate models both with their suggested prompt template as well as the default *instruct* setting.

Sampling vs. greedy decoding The default behavior in the framework is not to use greedy decoding, but to apply sampling with the temperature $T = 0.2$. Alongside greedy decoding, we conduct experiments using sampling with temperatures $T = 0.2$ and $T = 0.8$.

Limiting generation length The default value of 512 tokens is insufficient, as it can lead to unfinished programs and reduced benchmark scores. We conduct our experiments using lengths of 512 and 2048. Since the *max length generation* parameter considers both the prompt and the generated output, it must be large enough to prevent cutting the output short. We found 2048 to be a good choice on this benchmark, as tokenizers typically fit the inputs within 1024 tokens, leaving enough space for the generated output.

Precision and quantization The majority of LLMs are released using bf16 precision, making it the preferred choice for precision. We evaluate models using all three common precision formats: fp16, fp32, and bf16. While quantization is a useful technique to lower memory usage when running models, it's generally best to use models without it. In addition to our tests without quantization, we also run experiments using 4-bit and 8-bit quantization (with fp16 precision).

To assess the effect of each parameter setting individually, we select a configuration as the baseline. Then we compare the evaluation results of this baseline configuration with those obtained when only one setting differs. In the baseline setting, the prompt format follows the recommendations of the model's authors, no sampling is applied, the maximum generation length is set to the sufficiently large value of 2048, and bf16 precision is used without quantization.

Next, we attempt to reproduce the results reported in the literature. To do this, we run an exhaustive grid search for each model, with all possible combinations of the evaluation settings. As we have 2 options for prompt, 3 for sampling and temperature, 2 for generation length, and 5 for precision and quantization, we evaluate every single model in 60 settings.

While our main focus is on instruction-tuned models – as these are more frequently evaluated throughout studies –, we also take results of base models into consideration to assess whether they reproduce results originally attributed to their instruction-tuned counterparts.

The experiments are performed on A100 GPUs with 40GB VRAM each. We set *max_memory_per_gpu* to the value of *auto*, which distributes the model on the assigned GPUs. We utilize only as many GPUs as required to load the model and execute the benchmark, which is typically just one. Where multiple GPUs were used, we also measured whether modifying the number of assigned GPUs has an influence on results; we have not seen any variation in benchmark results when increasing the number of utilized GPUs.

3. Experimental Results

In this section, we present the results of our evaluations. First, we examine the impact of each evaluation parameter individually, compared to the baseline configuration. Next, we perform an

 doi:10.20944/preprints202505.2321.v1

exhaustive grid search for each model and assess whether the reported results can be reproduced. All evaluation results are summarized in Table 2.

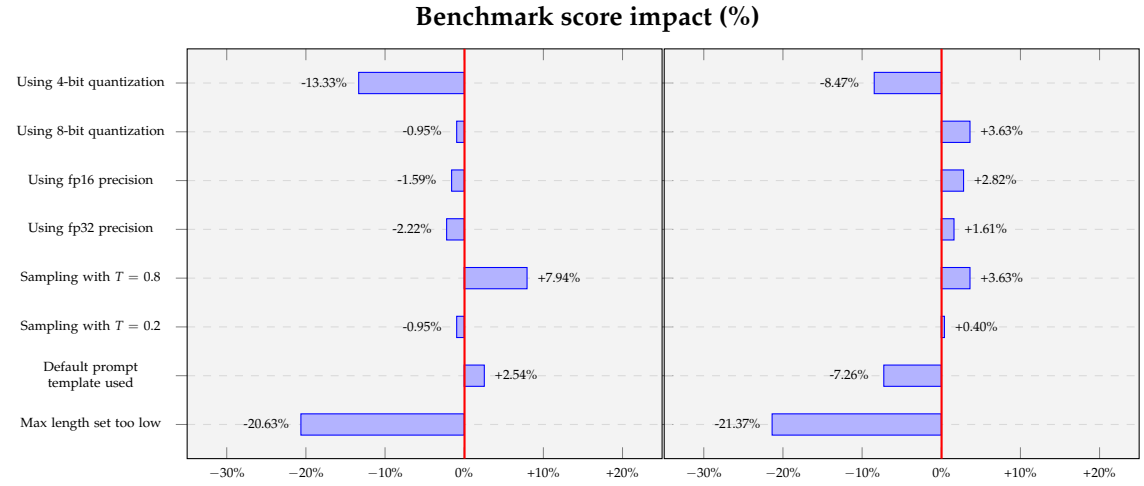
Table 2. All of our evaluation results and the reported results for the evaluated models (on the right). Succesful or close reproductions of reported scores are highlighted with bold and underlined font.

Sampling + Temperature:				Greedy decoding				Sampling with T = 0.2				Sampling with T = 0.8				Reproduction of reports
Max generation length:				2048		512		2048		512		2048		512		match / no match
Prompt type:				proper	instruct	proper	instruct	proper	instruct	proper	instruct	proper	instruct	proper	instruct	close match (±1%)
Model			Precision													(close) match in other tuning
DeepSeekCoder	Instruct	1.3B	fp16	21.34%	20.73%	**16.46%**	18.90%	22.56%	20.12%	**16.46%**	18.90%	**29.27%**	25.00%	17.68%	18.29%	9.1%
			fp32	21.34%	20.73%	**16.46%**	18.90%	21.34%	20.12%	**16.46%**	18.90%	25.00%	25.61%	17.68%	23.17%	29.3%
			bf16	20.73%	22.56%	15.85%	20.73%	20.73%	23.78%	17.68%	19.51%	25.61%	24.39%	17.68%	21.95%	48.9%
			fp16 (8bit)	20.73%	19.51%	FAILED	FAILED	20.12%	24.39%	FAILED	FAILED	20.73%	28.05%	FAILED	FAILED	
			fp16 (4bit)	21.95%	20.12%	FAILED	FAILED	22.56%	20.73%	FAILED	FAILED	20.12%	23.78%	FAILED	FAILED	
		6.7B	fp16	48.78%	**45.12%**	31.71%	37.20%	51.22%	**45.12%**	32.93%	37.20%	49.39%	43.29%	31.71%	33.54%	42.1%
			fp32	48.78%	**45.12%**	31.71%	37.20%	48.78%	**44.51%**	32.93%	37.80%	50.00%	**42.68%**	34.76%	35.37%	44.9%
			bf16	50.00%	46.34%	32.93%	37.20%	50.61%	**45.73%**	32.32%	37.80%	50.61%	**44.51%**	34.76%	34.76%	56.1%
			fp16 (8bit)	50.61%	**44.51%**	FAILED	FAILED	52.44%	**44.51%**	FAILED	FAILED	52.44%	40.24%	FAILED	FAILED	60.4%
			fp16 (4bit)	**44.51%**	39.02%	FAILED	FAILED	43.29%	39.02%	FAILED	FAILED	**44.51%**	39.63%	FAILED	FAILED	73.3%
	Base	1.3B	fp16	0.61%	**1.22%**	0.00%	0.61%	0.61%	2.44%	0.00%	1.83%	3.66%	8.54%	1.83%	6.71%	1.2%
			fp32	0.61%	**1.22%**	0.00%	0.61%	0.61%	2.44%	0.00%	1.83%	4.27%	10.98%	1.83%	8.54%	16.4%
			bf16	0.61%	**1.22%**	0.00%	0.61%	0.61%	1.83%	0.00%	1.83%	3.66%	11.59%	1.83%	7.93%	
			fp16 (8bit)	0.61%	3.66%	FAILED	FAILED	0.61%	4.27%	FAILED	FAILED	2.44%	4.88%	FAILED	FAILED	
			fp16 (4bit)	1.83%	6.10%	FAILED	FAILED	3.05%	6.10%	FAILED	FAILED	3.05%	10.98%	FAILED	FAILED	
		6.7B	fp16	30.49%	19.51%	25.61%	17.07%	31.10%	20.12%	25.61%	18.29%	32.32%	27.44%	**24.39%**	25.00%	23.8%
			fp32	31.10%	19.51%	26.22%	17.07%	31.10%	20.12%	25.61%	18.29%	32.93%	27.44%	26.83%	19.51%	29.9%
			bf16	31.71%	20.73%	25.61%	17.68%	**29.88%**	21.34%	**24.39%**	19.51%	34.15%	**24.39%**	25.00%	25.61%	45.4%
			fp16 (8bit)	30.49%	20.12%	FAILED	FAILED	33.54%	20.73%	FAILED	FAILED	**24.39%**	25.61%	FAILED	FAILED	
			fp16 (4bit)	22.56%	17.07%	FAILED	FAILED	21.34%	21.34%	FAILED	FAILED	35.37%	**23.17%**	FAILED	FAILED	
CodeLlama	Instruct	7B	fp16	**19.51%**	24.39%	17.68%	22.56%	18.29%	23.78%	15.85%	21.95%	**19.51%**	21.95%	17.07%	25.00%	15.2%
			fp32	**19.51%**	24.39%	17.68%	22.56%	18.29%	23.78%	16.46%	21.95%	21.34%	21.95%	16.46%	23.78%	19.5%
			bf16	**19.51%**	24.39%	17.07%	22.56%	18.90%	22.56%	15.85%	23.17%	17.07%	22.56%	15.85%	**20.73%**	20.6%
			fp16 (8bit)	21.34%	25.61%	FAILED	FAILED	21.95%	24.39%	FAILED	FAILED	24.39%	24.39%	FAILED	FAILED	28.0%
			fp16 (4bit)	**15.24%**	**15.24%**	FAILED	FAILED	**15.24%**	**15.24%**	FAILED	FAILED	14.02%	21.34%	FAILED	FAILED	
		13B	fp16	**18.90%**	12.80%	17.68%	12.20%	18.29%	12.20%	**16.46%**	13.41%	**19.51%**	21.34%	20.12%	13.41%	15.2% 30.5%
			fp32	**18.90%**	12.80%	17.68%	12.20%	18.29%	12.20%	17.07%	13.41%	**19.51%**	23.17%	18.29%	17.07%	16.4%
			bf16	**18.90%**	12.80%	17.68%	12.20%	**19.51%**	14.02%	17.68%	13.41%	21.95%	**19.51%**	20.73%	**15.24%**	18.9%
			fp16 (8bit)	**18.90%**	13.41%	FAILED	FAILED	20.12%	15.85%	FAILED	FAILED	21.95%	20.73%	FAILED	FAILED	19.4%
			fp16 (4bit)	12.20%	10.37%	FAILED	FAILED	12.80%	11.59%	FAILED	FAILED	14.63%	17.07%	FAILED	FAILED	29.2%
	Base	7B	fp16	0.61%	15.85%	0.61%	14.63%	0.61%	14.63%	0.61%	14.02%	2.44%	17.68%	1.83%	15.85%	30.0%
			fp32	0.61%	15.85%	0.61%	14.63%	0.61%	14.63%	0.61%	14.02%	1.83%	15.85%	1.83%	19.51%	
			bf16	0.61%	15.24%	0.61%	14.02%	0.61%	14.02%	0.61%	14.63%	0.00%	15.24%	1.83%	16.46%	
			fp16 (8bit)	0.61%	14.63%	FAILED	FAILED	0.61%	15.85%	FAILED	FAILED	3.05%	21.34%	FAILED	FAILED	
			fp16 (4bit)	0.61%	14.63%	FAILED	FAILED	0.61%	13.41%	FAILED	FAILED	1.22%	18.90%	FAILED	FAILED	
		13B	fp16	0.61%	4.27%	0.61%	14.63%	0.61%	3.05%	0.61%	3.66%	3.66%	12.20%	3.05%	13.41%	6.1%
			fp32	0.61%	3.66%	0.61%	14.63%	0.61%	3.05%	0.61%	3.66%	4.88%	12.20%	3.05%	13.41%	33.1%
			bf16	0.61%	5.49%	0.61%	4.88%	0.61%	**6.10%**	0.61%	5.49%	3.66%	9.15%	2.44%	17.07%	
			fp16 (8bit)	0.61%	7.32%	FAILED	FAILED	0.61%	4.88%	FAILED	FAILED	4.88%	15.24%	FAILED	FAILED	
			fp16 (4bit)	0.61%	7.32%	FAILED	FAILED	0.00%	6.71%	FAILED	FAILED	3.66%	15.24%	FAILED	FAILED	
CodeGemma	Instruct	7B	fp16	46.95%	34.15%	38.41%	29.27%	45.73%	33.54%	39.02%	29.27%	45.12%	32.93%	34.15%	26.83%	46.3%
			fp32	45.12%	33.54%	37.20%	28.66%	**46.34%**	33.54%	36.59%	28.05%	44.51%	35.98%	37.80%	28.05%	72.4%
			bf16	42.07%	34.15%	35.37%	28.66%	42.07%	29.88%	38.41%	27.44%	41.46%	33.54%	31.71%	26.83%	
			fp16 (8bit)	45.12%	32.93%	FAILED	FAILED	48.78%	32.93%	FAILED	FAILED	44.51%	35.98%	FAILED	FAILED	
			fp16 (4bit)	44.51%	29.27%	FAILED	FAILED	45.12%	32.32%	FAILED	FAILED	43.29%	30.49%	FAILED	FAILED	
	Base	2B	fp16	0.61%	**4.27%**	0.61%	**4.27%**	0.00%	4.88%	0.00%	4.88%	3.66%	7.32%	2.44%	6.71%	4.3%
			fp32	0.61%	4.88%	0.61%	4.88%	0.00%	4.88%	0.00%	3.66%	3.66%	10.37%	3.05%	6.10%	22.7%
			bf16	1.83%	**4.27%**	1.83%	**4.27%**	1.83%	**4.27%**	1.22%	4.88%	3.66%	9.15%	**4.27%**	12.20%	
			fp16 (8bit)	0.61%	4.88%	FAILED	FAILED	0.00%	4.88%	FAILED	FAILED	1.22%	9.15%	FAILED	FAILED	
			fp16 (4bit)	2.44%	4.88%	FAILED	FAILED	2.44%	4.88%	FAILED	FAILED	3.66%	9.76%	FAILED	FAILED	
		7B	fp16	0.61%	7.32%	0.61%	6.71%	1.22%	9.76%	0.61%	**8.54%**	7.93%	14.02%	6.10%	**12.20%**	8.5%
			fp32	0.61%	**8.54%**	0.61%	7.93%	1.22%	9.76%	0.61%	9.15%	4.88%	17.07%	5.49%	15.24%	11.7%
			bf16	5.49%	9.76%	4.88%	**8.54%**	4.88%	10.98%	3.05%	10.98%	5.49%	15.24%	6.10%	12.80%	53.7%
			fp16 (8bit)	0.61%	7.32%	FAILED	FAILED	1.22%	10.37%	FAILED	FAILED	6.71%	12.80%	FAILED	FAILED	
			fp16 (4bit)	0.00%	1.22%	FAILED	FAILED	0.61%	2.44%	FAILED	FAILED	0.00%	1.83%	FAILED	FAILED	

3.1. The Effect of Individual Evaluation Settings

We measure the effect of individual evaluation settings by comparing the baseline configuration against variants in which evaluation parameters are modified individually. The effect of each parameter change on the benchmark scores is analyzed in isolation. The impact of these modifications is visualized in Figure 2.

 doi:10.20944/preprints202505.2321.v1



Subfigure 1: All evaluated models **Subfigure 2:** Instruction-tuned models

Figure 2. Effects of individually altering each evaluation setting relative to the baseline, showing both the average change in overall benchmark score and standard deviation.

3.1.1. Temperature: Greedy Evaluation and Sampling

Considering all models, sampling with temperature T=0.2 has a slight impact on average performance: -0.95%. The change in performance is more significant when using T=0.8: $+7.94\%$. However, by only considering the instruction-tuned models, this change is less notable: $+0.40\%$ (with T=0.2) and $+3.63\%$ (with T=0.8).

For the baseline of greedy decoding, the standard deviation of scores across precision and quantization settings is 1.58 for all models and 1.98 for instruction-tuned models. When we switch to sampling at T=0.2, the standard deviation rises to 1.94 ($+22.99\%$) for all models and 2.48 ($+25.03\%$) for the instruction-tuned ones. Increasing the temperature further to T=0.8 increases standard deviation even more, to 2.37 ($+50.40\%$) for all models and 3.03 ($+52.78\%$) for the instruction-tuned models.

3.1.2. Maximum Generation Length

In the case of DeepSeekCoder-Instruct (1.3B) (evaluated using fp16), we see a notable decrease in performance (21.34% $\rightarrow$ 16.46%) when limiting the length of generation to the default value (to 512). This is a significant effect with a relative drop of 22.87%. The effect also applies more generally: across all models, the average performance drop is 20.63%. Similarly, for the instruction-tuned models, this drop is 21.37%.

3.1.3. Using an Incorrect Prompt Template

Considering the instruction-tuned models, switching from the suggested template to the default *instruct* prompt template resulted in average relative drop of 7.26% in model performance. This phenomenon is not observable when base models are also included in the calculation: the average performance even increases by 2.54%.

3.1.4. Precision and Quantization

By switching from bf16 to fp32 or fp16 precision, performance drops by 2.22% (fp32) and 1.59% (fp16) for all models, and increases by 1.61% (fp32) and 2.82% (fp16) for instruction-tuned models. Similarly, 8-bit quantization seems to have a rather small effect with changes of -0.95% (all models) and $+3.63\%$ (instruction-tuned). However, using 4-bit quantization decreases overall scores by a significant margin: -13.33% for all models and -8.47% for the instruction-tuned ones.

3.2. Reproducibility of Results in Existing Research

We attempt to reproduce existing results in the literature by trying out all possible combinations of the experimental settings. Our main goal is to identify potential misconfigurations in the original setups. All results from our evaluations as well as scores from existing studies can be seen in Table 2.

 doi:10.20944/preprints202505.2321.v1

3.2.1. CodeLlama

The results reported in studies for CodeLlama-Instruct (7B) range from 15.2% to 28.0%. This is in line with our evaluations, in which this model achieves scores from 14.02% to 25.61%. We reproduced all of the reported scores with one exception, indicating that the differences in reported results are indeed caused by variations in the evaluation settings. For CodeLlama-Instruct (13B), reported results range from 15.2% to 30.5%. Our evaluation yielded considerably lower scores: 10.37% to 23.17%. We reproduced 4 out of 6 results.

The base CodeLlama (13B) model has two reported scores across papers: 6.1%, which we reproduced, and an unexpectedly high score of 33.1%. The latter score is very close to the upper range of scores reported for CodeLlama-Instruct (13B).

3.2.2. DeepSeekCoder

The reported results for the DeepSeekCoder (1.3B and 6.7B) base models go up as high as 16.4% and 45.4% for the 1.3B and 6.7B variants, respectively. These scores are unusually high, especially when considering our own evaluations, which never exceed 11.59% (1.3B) and 35.37% (6.7B).

DeepSeekCoder (1.3B) for example is measured to have both 1.2% and 16.4% pass@1 score, which represents more than a 10X difference. We evaluated this model and its instruction-tuned variant and reproduced both reported results: 1.2% was reproduced using the base model, while the score of 16.4% was only reproducible with the instruction-tuned variant. Using the 6.7B base model, we reproduced two of the reported results, but not the very high score of 45.4%. Its instruction-tuned variant however, has yielded multiple results close to this value, with the closest being 45.12%.

For the instruction-tuned 1.3B model, one of three values was reproduced. The score of 9.1% might come from the base-model, as some of our evaluations yielded values close to it. Our evaluation results of the 6.7B instruction-tuned model range from 31.71% and 52.44%. These are partially in line with the reported results, with two very close results.

3.2.3. CodeGemma

Our evaluations of the CodeGemma (2B) base model attained up to 12.20% for the 2B model with one of the two reported scores reproduced. Using the 7B base model, values go up to 17.07% with one exact match and one near match to the reported scores; the third score was significantly higher. In our evaluations of CodeGemma-Instruct (7B), we reproduced one of the two reported results.

4. Discussion

The results of our reproducibility experiments highlight several possible reasons for the discrepancies between results in the literature. They also point to some strategies that could help prevent these discrepancies.

In some cases, we see surprisingly large scores for base models. We think this is due to confusing the base model with its instruction-tuned variant. It is crucial to ensure that the intended model variant is used, not one with different tuning. Furthermore, it is necessary to confirm that the correct model name and type is specified when reporting evaluation results.

The pass@1 scores do not change significantly when switching from greedy evaluation to sampling, especially for instruction-tuned models. However, the standard deviation increases, especially with higher temperature values. Thus, we observe that greedy evaluation should be preferred over sampling when calculating the pass@1 performance on this task.

Restricting the models to a short generation length affects the results negatively, resulting in large drops in performance. In order to obtain accurate results, a proper limit should be chosen to avoid cutting possibly correct generations. In the case of our benchmark of focus, HumanEvalFix, 2048 is such a value, but this might differ for other benchmarks.

 doi:10.20944/preprints202505.2321.v1

Using the proper prompt template (defined by model authors) is necessary to ensure stable and reliable evaluation results. Without it, the evaluation may not reflect accurate results. This only applies to instruction-tuned models, as base models were not fine-tuned to process such templates.

We noticed only a slight variation when modifying precision settings or using 8-bit quantization, suggesting these factors do not strongly account for differing results across papers. However, 4-bit quantization does have a notable effect.

5. Conclusion

In this paper, we highlighted the issue of inconsistent benchmark results in the literature on the HumanEvalFix benchmark, one of the most widely used benchmarks for evaluating bugfixing. We found that for most of the models mentioned in multiple papers, the reported benchmark results can vary significantly. We conducted evaluations to determine the effect of various evaluation settings, and to reproduce different scores reported for the same models, in order to uncover the reasons for these inconsistencies.

Through a series of emprirical evaluations, using multiple models in different sizes and tunings, we identified the set of factors influencing benchmark results. Our experiments revealed that factors such as the prompt template, maximum generation length and decoding strategies had a notable influence on benchmark results, whereas precision and 8-bit quantization did not. We have also found cases where results were likely misattributed between instruction-tuned and base models. We wish to emphasize the importance of using appropriate evaluation settings and including these settings in the studies to ensure reliable and reproducible results.

Author Contributions: Conceptualization, B.Sz.; methodology, B.Sz.; software, B.M.; investigation, B.Sz. and B.M.; writing—original draft preparation, B.Sz.; writing—review and editing, B.Sz., B.M, B.P and T.G.; visualization, B.M.; supervision, B.P. and T.G.; project administration, T.G.; funding acquisition, T.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by EKÖP-24 University Excellence Scholarship Program of the Ministry for Culture and Innovation from the Source of the National Research, Development and Innovation Fund.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Muennighoff, N.; Liu, Q.; Zebaze, A.; Zheng, Q.; Hui, B.; Zhuo, T.Y.; Singh, S.; Tang, X.; von Werra, L.; Longpre, S. OctoPack: Instruction Tuning Code Large Language Models, 2024, [arXiv:cs.CL/2308.07124].
2. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code Llama: Open Foundation Models for Code, 2024, [arXiv:cs.CL/2308.12950].
3. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, 2024, [arXiv:cs.SE/2401.14196].
4. Lin, D.; Koppel, J.; Chen, A.; Solar-Lezama, A. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In Proceedings of the Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, New York, NY, USA, 2017; SPLASH Companion 2017, p. 55–56. https://doi.org/10.1145/3135932.3135941.
5. Widyasari, R.; Sim, S.Q.; Lok, C.; Qi, H.; Phan, J.; Tay, Q.; Tan, C.; Wee, F.; Tan, J.E.; Yieh, Y.; et al. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In Proceedings of the Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 2020; ESEC/FSE 2020, p. 1556–1560. https://doi.org/10.1145/3368089.3417943.
6. Liu, S.; Chai, L.; Yang, J.; Shi, J.; Zhu, H.; Wang, L.; Jin, K.; Zhang, W.; Zhu, H.; Guo, S.; et al. MdEval: Massively Multilingual Code Debugging, 2024, [arXiv:cs.CL/2411.02310].
7. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code, 2021, [arXiv:cs.LG/2107.03374].

 doi:10.20944/preprints202505.2321.v1

8. Pimentel, M.A.; Christophe, C.; Raha, T.; Munjal, P.; Kanithi, P.K.; Khan, S. Beyond Metrics: A Critical Analysis of the Variability in Large Language Model Evaluation Frameworks, 2024, [arXiv:cs.AI/2407.21072].
9. Paul, D.G.; Zhu, H.; Bayley, I. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review, 2024, [arXiv:cs.AI/2406.12655].
10. Wang, S.; Asilis, J.; Ömer Faruk Akgül.; Bilgin, E.B.; Liu, O.; Neiswanger, W. Tina: Tiny Reasoning Models via LoRA, 2025, [arXiv:cs.CL/2504.15777].
11. Team, C.; Zhao, H.; Hui, J.; Howland, J.; Nguyen, N.; Zuo, S.; Hu, A.; Choquette-Choo, C.A.; Shen, J.; Kelley, J.; et al. CodeGemma: Open Code Models Based on Gemma, 2024, [arXiv:cs.CL/2406.11409].
12. Ben Allal, L.; Muennighoff, N.; Kumar Umapathi, L.; Lipkin, B.; von Werra, L. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.
13. Cassano, F.; Li, L.; Sethi, A.; Shinn, N.; Brennan-Jones, A.; Ginesin, J.; Berman, E.; Chakhnashvili, G.; Lozhkov, A.; Anderson, C.J.; et al. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions, 2024, [arXiv:cs.SE/2312.12450].
14. Chae, H.; Kwon, T.; Moon, S.; Song, Y.; Kang, D.; Ong, K.T.i.; Kwak, B.w.; Bae, S.; Hwang, S.w.; Yeo, J. Coffee-Gym: An Environment for Evaluating and Improving Natural Language Feedback on Erroneous Code. In Proceedings of the Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing; Al-Onaizan, Y.; Bansal, M.; Chen, Y.N., Eds., Miami, Florida, USA, nov 2024; pp. 22503–22524.
15. Dehghan, M.; Wu, J.J.; Fard, F.H.; Ouni, A. MergeRepair: An Exploratory Study on Merging Task-Specific Adapters in Code LLMs for Automated Program Repair, 2024, [arXiv:cs.SE/2408.09568].
16. Granite Team, I. Granite 3.0 Language Models, 2024.
17. Campos, V. Bug Detection and Localization using Pre-trained Code Language Models. In *INFORMATIK 2024*; Gesellschaft für Informatik e.V.: Bonn, 2024; pp. 1419–1429. https://doi.org/10.18420/inf2024_124.
18. Jiang, Y.; He, Q.; Zhuang, X.; Wu, Z. Code Comparison Tuning for Code Large Language Models, 2024, [arXiv:cs.CL/2403.19121].
19. Jiang, H.; Liu, Q.; Li, R.; Ye, S.; Wang, S. CursorCore: Assist Programming through Aligning Anything, 2024, [arXiv:cs.CL/2410.07002].
20. Lozhkov, A.; Li, R.; Allal, L.B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; et al. StarCoder 2 and The Stack v2: The Next Generation, 2024, [arXiv:cs.SE/2402.19173].
21. Mishra, M.; Stallone, M.; Zhang, G.; Shen, Y.; Prasad, A.; Soria, A.M.; Merler, M.; Selvam, P.; Surendran, S.; Singh, S.; et al. Granite Code Models: A Family of Open Foundation Models for Code Intelligence, 2024, [arXiv:cs.AI/2405.04324].
22. Moon, S.; Chae, H.; Song, Y.; Kwon, T.; Kang, D.; iunn Ong, K.T.; won Hwang, S.; Yeo, J. Coffee: Boost Your Code LLMs by Fixing Bugs with Feedback, 2024, [arXiv:cs.CL/2311.07215].
23. Nakamura, T.; Mishra, M.; Tedeschi, S.; Chai, Y.; Stillerman, J.T.; Friedrich, F.; Yadav, P.; Laud, T.; Chien, V.M.; Zhuo, T.Y.; et al. Aurora-M: The First Open Source Multilingual Language Model Red-teamed according to the U.S. Executive Order, 2024, [arXiv:cs.CL/2404.00399].
24. Shi, Y.; Wang, S.; Wan, C.; Gu, X. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging, 2024, [arXiv:cs.CL/2410.01215].
25. Singhal, M.; Aggarwal, T.; Awasthi, A.; Natarajan, N.; Kanade, A. NoFunEval: Funny How Code LMs Falter on Requirements Beyond Functional Correctness, 2024, [arXiv:cs.SE/2401.15963].
26. Wang, X.; Li, B.; Song, Y.; Xu, F.F.; Tang, X.; Zhuge, M.; Pan, J.; Song, Y.; Li, B.; Singh, J.; et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024, [arXiv:cs.SE/2407.16741].
27. Yang, J.; Jimenez, C.E.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.; Press, O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, 2024, [arXiv:cs.SE/2405.15793].
28. Yu, Z.; Zhang, X.; Shang, N.; Huang, Y.; Xu, C.; Zhao, Y.; Hu, W.; Yin, Q. WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning, 2024, [arXiv:cs.CL/2312.14187].
29. Wang, Y.; Le, H.; Gotmare, A.D.; Bui, N.D.Q.; Li, J.; Hoi, S.C.H. CodeT5+: Open Code Large Language Models for Code Understanding and Generation, 2023, [arXiv:cs.CL/2305.07922].
30. Tunstall, L.; Lambert, N.; Rajani, N.; Beeching, E.; Le Scao, T.; von Werra, L.; Han, S.; Schmid, P.; Rush, A. Creating a Coding Assistant with StarCoder. *Hugging Face Blog* **2023**. https://huggingface.co/HuggingFaceH4/starchat-beta.
31. Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; et al. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X, 2024, [arXiv:cs.LG/2303.17568].

 doi:10.20944/preprints202505.2321.v1

32. Li, R.; Allal, L.B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: may the source be with you!, 2023, [arXiv:cs.CL/2305.06161].
33. Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; Jiang, D. WizardCoder: Empowering Code Large Language Models with Evol-Instruct, 2023, [arXiv:cs.CL/2306.08568].
34. Workshop, B.; :.; Scao, T.L.; Fan, A.; Akiki, C.; Pavlick, E.; Ilić, S.; Hesslow, D.; Castagné, R.; Luccioni, A.S.; et al. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model, 2023, [arXiv:cs.CL/2211.05100].
35. Jiang, A.Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D.S.; de las Casas, D.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; et al. Mistral 7B, 2023, [arXiv:cs.CL/2310.06825].
36. Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. The Llama 3 Herd of Models, 2024, [arXiv:cs.AI/2407.21783].