# Can docstring reformulation with an LLM improve code generation?

**Nicola Dainese** and **Alexander Ilin** and **Pekka Marttinen**
Department of Computer Science
Aalto University
nicola.dainese@aalto.fi

## Abstract

Generating code is an important application of Large Language Models (LLMs) and the task of function completion is one of the core open challenges in this context. Existing approaches focus on either training, fine-tuning or prompting LLMs to generate better outputs given the same input. We propose a novel and complementary approach: to optimize part of the input, the docstring (summary of a function's purpose and usage), via reformulation with an LLM, in order to improve code generation. We develop two baseline methods for optimizing code generation via docstring reformulation and test them on the original HumanEval benchmark and multiple curated variants which are made more challenging by realistically worsening the docstrings. Our results show that, when operating on docstrings reformulated by an LLM instead of the original (or worsened) inputs, the performance of a number of open-source LLMs does not change significantly. This finding demonstrates an unexpected robustness of current open-source LLMs to the details of the docstrings. We conclude by examining a series of questions, accompanied by in-depth analyses, pertaining to the sensitivity of current open-source LLMs to the details in the docstrings, the potential for improvement via docstring reformulation and the limitations of the methods employed in this work.

## 1 Introduction

Large Language Models for coding (code LLMs) emerged in recent years as the dominant approach to code generation (Chen et al., 2021; Nijkamp et al., 2023b,a; Li et al., 2023; Rozière et al., 2023). The research community proposed various benchmarks to systematically evaluate the code generation abilities of LLMs (Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021; Lai et al., 2022). One of the grand challenges of code generation is the task of function completion: given its definition and an explanation of the desired behaviour, write the body of the desired function. Current methods in code generation focus on multiple directions: better pre-training datasets and procedures for more performant base models (Li et al., 2023; Rozière et al., 2023), specific fine-tuning methods for coding models (Le et al., 2022; Chaudhary, 2023; Luo et al., 2023), test-time improvements via self-testing (Chen et al., 2022; Ni et al., 2023), solutions reranking (Zhang et al., 2022), self-repair (Chen et al., 2023b; Olausson et al., 2023) or other techniques (Shi et al., 2022; Shinn et al., 2023; Zelikman et al., 2023).

In this work we explore the idea of input transformations as a new, complementary approach to improving code generation abilities of LLMs. In the context of function completion tasks, the input can be formulated as a function declaration followed by a docstring, which explains in words what the function is supposed to do. As illustrated in Figure 1, we consider a specific input transformation for this task: docstring reformulation.

The motivation for this approach is the following: first, optimizing the docstrings could provide benefits that are model-agnostic, that is, benefit all code LLMs. We hypothesize that this could be the case, drawing inspiration from the success of prompting techniques, such as Chain-of-Thought (Wei et al., 2023) and similar works (Zhou et al., 2023), that demonstrate efficacy across a broad class of models. Second, optimizing docstrings, while maintaining their readability, can provide interpretable insights on the influence of docstrings' properties on code LLMs in the context of code generation.

This work makes the following contributions:

1. We introduce a novel framework for optimizing code generation through docstring reformulation with LLMs. Additionally, we propose two baseline methods for the docstring reformulation task.



*Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 296–312
March 21-22, 2024 ©2024 Association for Computational Linguistics

**Input with Original Docstring**

```python
def simplify(x, n):
    """Your task is to implement a function that will simplify
    the expression x * n. The function returns True if x * n
    evaluates to a whole number and False otherwise. Both x
    and n, are string representation of a fraction, and have
    the following format, <numerator>/<denominator> where both
    numerator and denominator are positive whole numbers.

    You can assume that x, and n are valid fractions, and do
    not have zero as denominator.

    simplify("1/5", "5/1") = True
    simplify("1/6", "2/1") = False
    simplify("7/10", "10/2") = False
    """
```

Original code LLM completion

```python
    # Convert the input strings to fractions
    x_frac = Fraction(x)
    n_frac = Fraction(n)

    # Multiply the fractions
    product = x_frac * n_frac

    # Check if the product is a whole number
    return product.denominator == 1
```

Not defined!

Passes all tests ✗

**Input with Reformulated Docstring**

```python
def simplify(x, n):
    """
    This function takes two string representations of fractions,
    x and n, and returns True if x * n
    evaluates to a whole number and False otherwise.

    Both x and n are in the format of <numerator>/<denominator>,
    where both numerator and denominator
    are positive whole numbers.

    The function simplifies the expression x * n by converting
    the string representations to integers,
    multiplying them, and checking if the result is a whole
    number.

    simplify("1/5", "5/1") = True
    simplify("1/6", "2/1") = False
    simplify("7/10", "10/2") = False
    """
```

New code LLM completion

```python
    # Convert the string representations to integers
    x_numerator, x_denominator = map(int, x.split('/'))
    n_numerator, n_denominator = map(int, n.split('/'))

    # Multiply the fractions
    result_numerator = x_numerator * n_numerator
    result_denominator = x_denominator * n_denominator

    # Check if the result is a whole number
    return result_numerator % result_denominator == 0
```

Passes all tests ✓

Figure 1: Motivating example of docstring reformulation with LLMs for improving code generation. One can learn to optimize docstrings by fine-tuning an LLM on successful docstring reformulations. The reformulations can improve the usefulness of the docstring in many ways, e.g. by adding hints on how to implement the function as in this example, or by fixing faults in the docstring.

2. We assess the effectiveness of the proposed methods on multiple variants of the HumanEval benchmark, finding a limited improvement to code generation.

3. We present a thorough analysis of the limitations of the models used, the methods proposed and the experimental setup. We find evidence for a significant margin of potential improvement in code generation via docstring reformulation when using oracle reformulations, and highlight key obstacles hindering this potential.

4. We independently replicate the performance of multiple open-source code LLMs on the HumanEval benchmark. Additionally, we release all code necessary for experiment reproduction and share the novel curated variants of HumanEval featuring faulty docstrings.

## 2 Related work

**Prompt optimization** Prompt optimization techniques have garnered significant attention in recent research. Li and Liang (2021) propose prefix-tuning, an alternative to fine-tuning, which focuses on optimizing task-specific vectors while keeping the language model parameters fixed. Lester et al. (2021) present prompt-tuning as a simplification of prefix tuning, involving the addition of small task-specific prompts for each task. Additionally, Liu et al. (2021) introduces P-tuning, a continuous optimization technique for mapping context to target output using prompts. Qin and Eisner (2021) also explore soft prompts, emphasizing their relevance in various NLP applications.

Reinforcement Learning (RL) has also been employed in prompt optimization. Deng et al. (2022) discuss the challenges associated with soft prompts and propose a method that employs RL to decode discrete prompts token-by-token. Zhang et al. (2023) leverage RL to dynamically construct instance-specific discrete prompts, enhancing task performance through query-dependent prompts.

**Instruction generation** As more and more LLMs are aligned to follow instructions, e.g., via instruction fine-tuning or reinforcement learning from human feedback, an open question is how to generate instructions in natural language in order to increase the likelihood of producing with an LLM the desired output for a given input. Zhou et al. (2023) introduce the Automatic Prompt Engineer (APE), framing instruction generation as a natural language program synthesis problem and propose search methods to find approximate so-

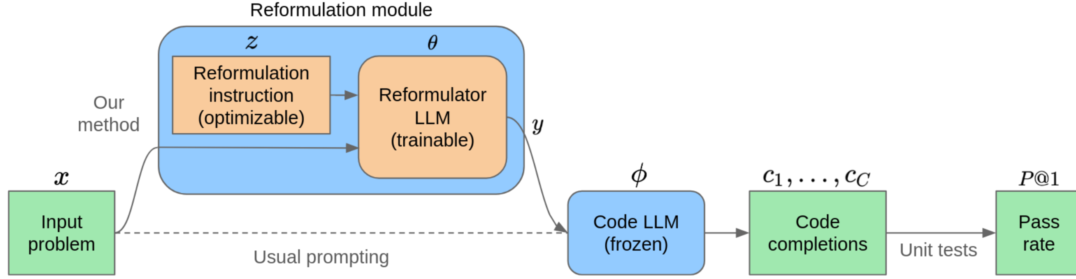


Figure 2: High-level view of the proposed method. We apply a transformation via a reformulation module $(\theta, z)$ to the input problem $x$ to obtain a reformulation $y$ of it. We then use the reformulated problem as input for any downstream code LLM $\phi$. The reformulation module is optimized to reformulate the docstring of the function to be completed, in order to increase the unit test pass rate (pass@1) of LLM-generated code completions for the target function.

lutions. Pryzant et al. (2023) propose Automatic Prompt Optimization (APO). APO leverages data minibatches to create natural language "gradients," representing linguistic attempts at achieving what mathematical gradients do. These language gradients critique the existing prompt and are integrated into it through steps akin to "gradient descent". Most relevant to our work, Yang et al. (2023) introduce Optimization by PROmpting (OPRO), a method for optimizing tasks specified in natural language. Their approach involves generating new solutions from a prompt containing previously generated solutions and their corresponding values, which are subsequently evaluated and incorporated into the prompt for further optimization.

**Evolutionary methods** Evolutionary methods involving language models have also been explored. Xu et al. (2023) expand the self-instruct method by Wang et al. (2023) with instruction evolution, employed in the creation of high-quality instruction-tuning datasets. Meyerson et al. (2023) investigate the use of language models as variation operators in evolutionary algorithms, enabling tasks such as symbolic regression and sentiment modification. Lehman et al. (2022) combine evolution through large models with MAP-Elites (Mouret and Clune, 2015) to generate functional examples of Python programs in the Sodarace domain, a task unseen during pre-training. Chen et al. (2023a) focus on producing valid neural network architectures for neural architecture search using LLMs.

## 3 Methodology

In this section, we first introduce the function completion task as formulated in benchmarks such as HumanEval, then we formalise the docstring reformulation task and finally we present our baseline methods for optimising docstring reformulations.

### 3.1 Code generation and function completion task

Current state-of-the-art code generation methods (Chen et al., 2021; Nijkamp et al., 2023b,a; Li et al., 2023; Rozière et al., 2023) use decoder-only Transformer architectures with auto-regressive probabilistic modeling of the next token to be generated. In this work we denote as code LLMs any LLM which during pre-training has been trained on a non-negligible amount of code data.

In the context of function completion tasks, given an input problem $x$ and a code LLM with parameters $\phi$, the code completion $c$ is typically obtained by generating one token at a time until the end-of-sentence (EOS) token is sampled or a maximum sequence length L is reached:

$$p_\phi(c|x) = \prod_{l=1}^{L} p_\phi(c_l|x, c_{<l}). \tag{1}$$

Given a dataset $D$ consisting of pairs of input problems and unit tests $(x, T)$, and a code LLM $\phi$, the performance $J(\phi, D)$ of the code LLM is:

$$J(\phi, D) = \mathop{\mathbb{E}}_{\substack{(x,T)\sim D,\\ c\sim p_\phi(c|x)}} [T(c)] \,. \tag{2}$$

$T(c)$ is a binary variable, with value 1 if the code completion $c$ passes all the unit tests $T$, and 0 if $c$ fails at least one unit test. $J(\phi, D)$ in the literature is also called 'pass@1' and is the main metric to evaluate code LLMs' performance on benchmarks such as HumanEval (Chen et al., 2021).

### 3.2 Docstring reformulation task

We consider the task of improving code generation abilities of code LLMs. In particular we are

interested in optimizing functions' docstrings to increase the probability of sampling a correct code solution from any code LLM; this is akin to treating the docstring as a prompt and optimizing the prompt for the given task.

We formalise the task as follows: given the dataset $D$ of input problem and unit test pairs $(x, T)$, and a set $\Phi$ of code LLMs, for each $x$ generate a reformulation $y_x$ to maximize the probability of sampling a correct code completion $c$ for $(x, T)$ with a code LLM $\phi$ uniformly sampled from $\Phi$:

$$\max_{y_x} \mathop{\mathbb{E}}_{\substack{(x,T)\sim D,\\ \phi\sim U(\Phi),\\ c\sim p_\phi(c|y_x)}} [T(c)]\,. \tag{3}$$

In this work, we frame this problem as a *docstring reformulation* task. For each code function $f$, there exist multiple possible docstrings to document it and we hypothesize that certain docstrings are more effective than others in guiding the code generation as desired. Furthermore, we speculate that the effectiveness of a docstring has a model-agnostic component, possibly because different code LLMs share pre-training data and would respond similarly to the same input patterns.

To reformulate docstrings, we rely on an instruction-tuned LLM $\theta$, which we denote *reformulator*. We provide the reformulator with the *original problem input* $x$ to be reformulated and with a *reformulation instruction* $z$, expressing how the reformulation task should be done.

The main reasons for the introduction of the instruction $z$ is the following. Docstring reformulation is a problem that requires some exploration mechanism, as the search space is huge and the original docstring provided in $x$ can be assumed to be a local maximum in the performance landscape. We hypothesize that using instructions to steer the reformulations in different directions is much more sample-efficient than relying only on stochastic sampling to search for the best reformulation.

Under this framework, the docstring reformulation task can be expressed as follows:

$$\max_{\theta,z} \mathop{\mathbb{E}}_{\substack{(x,T)\sim D,\\ \phi\sim U(\Phi),\\ y\sim p_\theta(y|x,z),\\ c\sim p_\phi(c|y)}} [T(c)]\,. \tag{4}$$

To evaluate the quality of a reformulation $y$ of a problem $x$, we decode it with a code LLM and test if the code sample $c$ passes the unit tests $T$. We can use this evaluation to rank multiple reformulations for the same input problem in terms of performance, which serves as the basis for any learning algorithm.

### 3.3 Proposed methods

We propose two main methods to optimize the reformulations:

1. **Supervised fine-tuning on the best instruction (SFT)**: maintaining the instruction $z$ fixed, fine-tune the reformulator $\theta$ on the best reformulation $y^*$ for each $x$.
2. **Instruction optimization via OPRO (OPRO)**: keeping the reformulator $\theta$ fixed, generate new instructions $z$ with a pre-trained LLM, denoted as *instruction optimizer*, conditioned on the past instructions and their pass rates following the OPRO method in Yang et al. (2023).

**SFT** In this first approach, for each input problem $x$ in the dataset, we generate $R$ ($R \geq 2$) reformulations $y_1, \ldots, y_R$ as:

$$y_1, \ldots, y_R \sim p_\theta(y|x, z). \tag{5}$$

In the SFT method, we consistently use the following hand-written instruction $z$:

```
"Improve the docstring of the following function
    using the best coding conventions."
```

The reformulation instruction $z$ and the input problem $x$ are presented to the reformulator using an instruction-following template adapted from Luo et al. (2023) and reported in Appendix B.1. For each reformulation $y_i$ we then generate $C$ code completions $c_1, \ldots, c_C$ using a code LLM. Each code completion is evaluated against the problem's unit tests T and the result is either pass (1) or fail (0). We define the best reformulation of the problem input $x$ as:

$$y^* = \mathop{\mathrm{argmax}}_{y\in\{y_1,\ldots,y_R\}} \sum_{j=1}^{C} T(c_j(y)). \tag{6}$$

We then perform supervised fine-tuning of the reformulator $\theta$ on the $(x, y^*)$ pairs, formatted with the same template used during reformulation and we compute the loss only for the tokens corresponding to $y^*$. In summary, at every iteration of the algorithm, we generate $R$ reformulations of each input problem in the training set, $C$ code completions of

each reformulation, evaluate all code completions against the corresponding tests and perform supervised fine-tuning on the pairs of input problems and best reformulations. We continue the training for multiple iterations and use the final reformulator model in the evaluation phase. During training, we use a single, fixed code LLM, the coder model, to generate the code based on the reformulations. During evaluation we use different coder models to study the generalizability of the benefits from the reformulated docstrings.

**OPRO** In this second approach, at every iteration, we generate $Z$ instructions $z_1, \ldots, z_Z$ by prompting the reformulator with a specific instruction generation template and then we form all possible combinations of reformulation instructions and input problems. For each combination, we generate one reformulation, using as input the same reformulation template as in the SFT method, but with a different $z$, and use the reformulation as input to the coder model to produce $C$ code completions. We score each reformulation instruction with the pass@1 metric of all the code completions associated with it. We start the first iteration with $Z$ hand-written instructions (see Appendix B). From the second iteration, to generate new instructions, we do the following:

1. Sample $n$ (instruction, score) pairs from all instructions evaluated so far;
2. Sort them in ascending order of score;
3. Format them according to the instruction generation template (described below);
4. Generate a continuation of the template with the *instruction optimizer* LLM and parse out the new instruction (until the first newline character).

$n$ is a hyper-parameter of the algorithm, and how to set its value is further discussed in the Appendix B.

We use the following instruction generation template, adapted from Yang et al. (2023):

```
Your task is to generate the next instruction to
achieve a higher score. The instructions should
ask to change, improve or rewrite the function
documentation or docstring. The instructions
should not ask to write new functions, add new
arguments or change the output of the given
function. Below are some previous instructions
with their scores. The score ranges from 0.0 to
1.0.
Instruction 1: {instruction_1}
Score 1: {score_1}
⋮
Instruction n: {instruction_n}
Score n: {score_n}
Instruction n+1:
```

## 4 Experiments and Results

**Datasets** In this work we consider HumanEval as a dataset on which to test the efficacy of the docstring reformulation, as it is one of the most used benchmarks in coding. To further study the influence of the docstrings and their reformulations in various scenarios, we curate four other versions of HumanEval, where we manually edited all input problems, introducing the respective faults:

- **Misspelling**: a character was either added, subtracted or changed in one of the most important words of every docstring.
- **Ambiguity**: all examples of input-output behaviour, hints and edge-case specifications are removed from every docstring.
- **Distractor**: a sentence out of context is inserted at the beginning or at the end of every docstring.
- **Bad formatting**: all type hints in the function declaration, blank lines and ">>>" symbols in front of examples are removed.

The motivation for introducing errors in the docstrings was to explore the potential for improvement by reformulation when the docstring is initially imperfect. We aimed at introducing errors similar to those that could potentially happen by human coders. We report examples of all the faults introduced in Appendix E.

**Experimental setup** For every variant of HumanEval, we run the SFT and the OPRO methods for 10 iterations. The SFT method uses 2 reformulations per input problem and 2 code completions per reformulation, while the OPRO method uses 5 reformulation instructions per input problem, one reformulation and one code completion.

During training, we use the WizardCoder-Python-7B (Luo et al., 2023) as the reformulator and coder model and, for the OPRO method, Llama-2-7b-chat (Touvron et al., 2023) as the instruction optimizer model. For the evaluation setup,

Table 1: **Results for SFT method.** We report the pass@1 results on the four selected variants of HumanEval, corresponding to the original problem and three modified versions with manually worsened docstrings. *Initial* column shows the performance on the initial input problem (original or worsened). *Reformulated* column shows the performance after applying the trained reformulator model to the corresponding initial input.

| Models | Original | | Misspelling | | Ambiguity | | Distractor | |
|---|---|---|---|---|---|---|---|---|
| | Initial | Reformulated | Initial | Reformulated | Initial | Reformulated | Initial | Reformulated |
| open_llama_7b_v2 | 13.4 (2.7) | 14.0 (2.7) | 13.4 (2.7) | 14.6 (2.8) | 15.2 (2.8) | 12.8 (2.6) | 10.3 (2.4) | 14.6 (2.8) |
| mpt-7b | 16.4 (2.9) | 15.9 (2.9) | 17.7 (3.0) | 14.6 (2.8) | 16.4 (2.9) | 17.7 (3.0) | 10.3 (2.4) | 17.7 (3.0) |
| starcoder | 33.5 (3.7) | 33.5 (3.7) | 35.4 (3.7) | 32.3 (3.7) | 30.5 (3.6) | 32.9 (3.7) | 31.1 (3.6) | 32.9 (3.7) |
| WizardCoder-3B | 35.4 (3.7) | 32.9 (3.7) | 29.2 (3.6) | 31.1 (3.6) | 30.5 (3.6) | 31.1 (3.6) | 33.5 (3.7) | 34.8 (3.7) |
| WizardCoder-Python-7B* | 53.0 (3.9) | 56.1 (3.9) | 46.3 (3.9) | 54.3 (3.9) | 53.7 (3.9) | 52.4 (3.9) | 54.8 (3.9) | 53.0 (3.9) |
| WizardCoder-15B | 57.9 (3.9) | 57.9 (3.9) | 56.1 (3.9) | 54.3 (3.9) | 51.8 (3.9) | 50.6 (3.9) | 54.2 (3.9) | 53.0 (3.9) |
| Average | 34.9 | **35.1** | 33.0 | **33.5** | **33.0** | 32.9 | 32.4 | **34.3** |

* WizardCoder-Python-7B is used as coder model during training.

Table 2: **Results for OPRO method.** We report the pass@1 results on the four selected variants of HumanEval, corresponding to the original problem and three modified versions with manually worsened docstrings. *Initial* column shows the performance on the initial input problem (original or worsened). *Reformulated* column shows the performance after applying the reformulator model with the optimized reformulation instruction to the corresponding initial input.

| Models | Original | | Misspelling | | Ambiguity | | Distractor | |
|---|---|---|---|---|---|---|---|---|
| | Initial | Reformulated | Initial | Reformulated | Initial | Reformulated | Initial | Reformulated |
| open_llama_7b_v2 | 13.4 (2.7) | 14.0 (2.7) | 13.4 (2.7) | 15.9 (2.9) | 15.2 (2.8) | 18.9 (3.1) | 10.3 (2.4) | 12.8 (2.6) |
| mpt-7b | 16.4 (2.9) | 17.1 (2.9) | 17.7 (3.0) | 15.9 (2.9) | 16.4 (2.9) | 18.9 (3.1) | 10.3 (2.4) | 14.0 (2.7) |
| starcoder | 33.5 (3.7) | 32.3 (3.7) | 35.4 (3.7) | 34.8 (3.7) | 30.5 (3.6) | 34.8 (3.7) | 31.1 (3.6) | 32.9 (3.7) |
| WizardCoder-3B | 35.4 (3.7) | 32.3 (3.7) | 29.2 (3.6) | 31.7 (3.6) | 30.5 (3.6) | 34.8 (3.7) | 33.5 (3.7) | 33.5 (3.7) |
| WizardCoder-Python-7B* | 53.0 (3.9) | 56.1 (3.9) | 46.3 (3.9) | 53.0 (3.9) | 53.7 (3.9) | 53.7 (3.9) | 54.8 (3.9) | 50.0 (3.9) |
| WizardCoder-15B | 57.9 (3.9) | 54.9 (3.9) | 56.1 (3.9) | 53.0 (3.9) | 51.8 (3.9) | 48.8 (3.9) | 54.2 (3.9) | 51.8 (3.9) |
| Average | **34.9** | 34.5 | 33.0 | **34.1** | 33.0 | **35.0** | 32.4 | **32.5** |

* WizardCoder-Python-7B is used as coder model during training.

in addition to the original coder model, we consider 5 other LLMs with model sizes ranging from 3B to 15B parameters: OpenLlama-2-7B-V2 (Geng and Liu, 2023; TogetherComputer, 2023), MPT-7B (MosaicML, 2023), starcoder (15B) (Li et al., 2023), WizardCoder-3B and WizardCoder-15B (Luo et al., 2023). These models were selected as a representative subset of the open-source LLM landscape. The selection criteria are discussed in Appendix C.

During the evaluation, we use the reformulator to produce one reformulation per each input problem via greedy decoding and pass each reformulation to all six coder models to compute the pass@1 metric for each model. As a baseline, we compute the pass@1 of each model using the non-reformulated problems as inputs to the coder models and following the same exact evaluation procedure.

**Results** We report in Table 1 the results for SFT method and in Table 2 the ones for the OPRO method. Additionally, we report the results for one more HumanEval variant (the bad formatting one) in Appendix D, as the average model performance did not decrease after introducing this type of fault, thus raising doubts about its relevance. We also report in parentheses the estimated errors for the models' performances as $\sqrt{p(1-p)/N}$, assuming a Binomial distribution of the successful code completions, with $p$ the pass rate (pass@1) and $N = 164$ the number of problems in the HumanEval dataset.

For both methods, we can notice two main trends: first, the average performance on the faulty variants of HumanEval across the coder models decreases slightly (roughly 2 percentage points), and second, the average performance when using reformulations rather than the initial (possibly faulty) input problems does not increase significantly. All models obtain mixed results, increasing performance on some reformulated variants and losing it on others, with the only exception of WizardCoder-15B, whose performance consistently decreases on reformulations.

## 5 Discussion and Conclusions

In the following section we raise a series of questions about the docstring reformulation framework, the SFT and OPRO methods and the experimental setup. We address these questions with ulterior argumentations and analyses, before drawing the conclusions from this study. A more extended discussion is presented in Appendix A.

**Q1. Limitations of docstring optimization for code generation:** *How capable are the code LLMs considered in this work to leverage docstrings' improvements?*

We use the following working definition of docstring improvement: *An increase in the information that the docstring contains about the body of the function to be completed.* This definition is model-agnostic, as it does not make reference to the performance of any model; rather, we expect that LLMs can leverage the increased information in the docstrings for better code generation.

First, we show in Table 3 in Appendix A that, if docstrings are completely removed from the input problems, the performance drops dramatically across all models, demonstrating that docstrings serve a key role in accurate function completion.

Then, we compare the performance of the various LLMs when evaluated on the faulty variants of HumanEval versus the original dataset, which can be considered an improved version of them. We find that five out of six models surprisingly increased performance on at least one of the four faulty variants of HumanEval, indicating that a docstring improvement does not necessarily benefit code generation and it can even hurt performance.

Finally we evaluate the coder models on two new sets of strongly improved docstrings for the HumanEval problems, produced while having access to the ground-truth function completions (oracle docstrings). The first set of docstrings is produced by GPT-4[1] with access to the ground-truth solution (*'Oracle Hints'*), asking the model to give detailed hints on how to implement the function. The second set contains the true body of the function to be completed (*'Oracle Solutions'*), so that the task of the coder models simplifies to copy-pasting the solution. Our results, presented in Table 4 in Appendix A, show that both Oracle Hints and Oracle Solutions docstrings greatly improve the performnace of all coder models.

[1] In all the experiments with GPT-4 we use GPT-4 Turbo, also referred as `gtp-4-1106-preview` in OpenAI API.

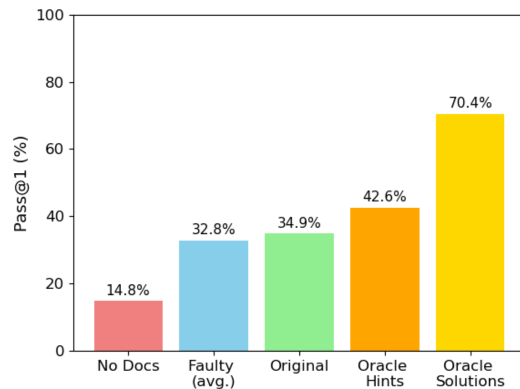


Figure 3: **Performance of different docstrings.** We report the pass@1 averaged across the six coder models for different kinds of docstrings: no docstrings at all ('No docs'), faulty docstrings ('Faulty (avg.)', average across Misspelling, Ambiguity and Distractor variants), original docstrings ('Original'), oracle docstrings with hints ('Oracle Hints') and with solutions ('Oracle Solutions').

Figure 3 summarises our findings on the ability of the considered LLMs in leveraging docstrings for code generation: coder models may not reliably leverage small improvements in the docstrings, but clearly benefit from the overall information included in them. In particular, the performance with Oracle Hints can be considered as a good estimate of the potential of optimizing code generation via docstring reformulation; how to achieve such performance without access to the ground-truth solution to generate hints remains an open question.

**Q2. Limitations of the docstring reformulation methods in principle:** *Are the methods proposed guaranteed to improve the performance of the coder model used during training? Is there any guarantee that the improvement will transfer to other coder models?*

We identify the following challenges that any method for docstring reformulation faces: exploration, noisy learning signal and learning rule, overfitting and generalization.

The exploration challenge is about searching for the best docstring for a given input problem. Both the SFT and the OPRO methods rely on stochastic sampling of the reformulations with an inductive bias, encoded as extra information $z$ in the prompt. However, this doesn't guarantee to find the best docstrings, as the search space is huge.

The challenge with noisy learning signals lies in the high variance of the pass rate metric, which is due to the stochasticity in the coder model. This can impair the stability of the optimization in the SFT method, as it is not well suited for dealing with noisy feedback. OPRO's learning rule, on the other hand, while more robust to noise, is reliant on the instruction optimizer LLM and as such, it also doesn't guarantee any convergence of the method.

Finally, if reformulations exclusively boost a particular coder model's performance while decreasing performance for most other models, they overfit to that model. Conversely, if reformulations enhance the performance of diverse coder models without specific tailoring, they demonstrate generalization across coder models. Empirically, we do not observe any sign of overfitting. We attribute this to the lack of backpropagation through the coder model in the proposed methods, which, we speculate, acts as a regularizer over the optimised reformulations and improves their generalizability.

We conclude that the proposed methods face key shortcomings in exploring the reformulation space and in learning from a noisy feedback signal.

**Q3. Limitations of the docstring reformulation methods in practice:** *Are there further practical considerations about our experimental setup that could affect the methods' success?*

In addition to the limitations discussed in **Q2**, the proposed methods may be limited by practical implementation choices. Initial experiments indicated that the choice of method parameters, as well as the language generation parameters and prompt templates for LLMs, does not strongly influence the results. Consequently, we run additional experiments to ablate the role of capability of the models employed as:

1. **Reformulator:** We evaluate the coder models on reformulations produced by GPT-4, instead of WizardCoder-Python-7B, prompted with the same reformulation instruction as in the SFT method. We use the original HumanEval dataset for this experiment.

2. **Instruction optimizer:** We reproduce the OPRO experiments for the original HumanEval dataset using GPT-4 as instruction optimizer model, instead of Llama-2-7b-chat.

Regarding the experiments on the reformulator, reported in Table 5 in Appendix A, we find no significant difference in performance between the two models for the given reformulation instruction; our qualitative inspection of the generated reformulations supports the conclusion that the selected open-source model can generate docstring reformulations on par with GPT-4 in this specific context.

In the case of the instruction generator experiment, the results for GPT-4, presented in Table 6 in Appendix A, are significantly worse than the ones obtained with the selected instruction optimizer. Qualitatively, GPT-4 suggests verbose reformulation instructions, often leading the reformulator to include in the documentation hallucinated information, e.g. about possible invalid inputs. This results in incorrect handling of edge cases in generated code completions and performance degradation.

In summary, our ablation studies in this section show that the limitations of the proposed methods are not linked with the quality of the models selected as reformulator and instruction optimizer, but rather to the points described in **Q2**.

## Conclusions

Code generation is crucial for diverse real-world applications, and accurate function completion poses a key challenge in this context. In this work we introduce docstring reformulation with an LLM as a novel approach to improve code generation for function completion and propose two methods to solve the task. When testing them on multiple variants of the HumanEval benchmark, we find limited improvements to code generation.

In our discussion, we first show that the considered coder models may not reliably leverage small improvements in the docstrings, but clearly benefit from the overall information included in them. Leveraging oracle reformulations, we then provide evidence that the more information the docstrings contain about the solutions, the more beneficial they are, regardless of the model. Finally, we argue that the proposed methods face key shortcomings in exploring the reformulation space and in learning from a noisy feedback signal, while we exclude limitations linked to our implementation choices.

Interesting future directions are to investigate more efficient ways of searching for promising reformulations, for example by reflecting on previous candidates, and to examine RL-based algorithms, such as RLHF (Christiano et al., 2023) and DPO (Rafailov et al., 2023), as alternatives to supervised fine-tuning.

## Limitations

The main limitation of this work is that we train and test our methods on the same input problems, i.e. the ones of HumanEval. The choice is due to the fact that not many benchmarks exist for function completion, as we require verified unit tests for each input problem in the benchmark. Further, HumanEval is arguably the most used coding benchmark at the time of writing and this facilitated verifying the performance of a large amount of open-source LLMs as the starting point of this work, which wouldn't have been possible otherwise.

However, we argue that this doesn't impair our results for the following reasons: First, we evaluate our methods also on different LLMs than the one used during training, in contrast with prior work, such as Pryzant et al. (2023) and Yang et al. (2023), that focuses on optimizing prompts for a single model. Second, we constrain the optimization to be done via a language prompt (the reformulation) and we only use a non-differentiable scalar feedback to score the reformulations; this is a setup very similar to the ones in bandit problems, where the reformulation serves as the action, the scalar feedback as reward and the performance is assessed on the training distribution. Third, we never let our methods see the code completions nor the true solutions to the input problems. However, future work should definitely focus on cross-dataset generalization of prompt reformulation. The other limitations are addressed in the main text, in Section 5.

## Acknowledgements

We are grateful to Minttu Alakuijala and Hans Moen for the insightful conversations during the development of this research. This research has been funded by the Academy of Finland Flagship program: Finnish Center for Artificial Intelligence (FCAI). We acknowledge the computational resources provided by the Aalto Science-IT project and by CSC – IT Center for Science, Finland. We also acknowledge CSC for awarding this project access to the LUMI supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CSC (Finland) and the LUMI consortium through Finland.

## References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Angelica Chen, David M. Dohan, and David R. So. 2023a. Evoprompting: Language models for code-level neural architecture search.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests.

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug.

Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2023. Deep reinforcement learning from human preferences.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. 2022. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3369–3391, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Xinyang Geng and Hao Liu. 2023. Openllama: An open reproduction of llama.

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

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000:

A natural and reliable benchmark for data science code generation.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. 2022. Evolution through large models.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. Gpt understands, too.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct.

Elliot Meyerson, Mark J. Nelson, Herbie Bradley, Arash Moradi, Amy K. Hoover, and Joel Lehman. 2023. Language model crossover: Variation through few-shot prompting.

MosaicML. 2023. Introducing mpt-7b: A new standard for open-source, commercially usable llms.

Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying gpt self-repair for code generation.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search.

Guanghui Qin and Jason Eisner. 2021. Learning how to ask: Querying LMs with mixtures of soft prompts. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5203–5212, Online. Association for Computational Linguistics.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

TogetherComputer. 2023. Redpajama-data: An open source recipe to reproduce llama training dataset.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions.

Tianjun Zhang, Xuezhi Wang, Denny Zhou, Dale Schuurmans, and Joseph E. Gonzalez. 2023. TEMPERA: Test-time prompt editing via reinforcement learning. In *The Eleventh International Conference on Learning Representations*.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder reviewer reranking for code generation.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*.

# A Extended discussion

**Q1. Limitations of the models considered:** *How capable are the code LLMs that we considered in this work to leverage improvements in the docstrings?*

We use the following working definition of docstring improvement: *An increase in the information that the docstring contains about the body of the function to be completed*. This definition is model-agnostic, as it does not make reference to the performance of any model; rather, we expect that LLMs can leverage the increased information in the docstrings for better code generation.

First, we show in Table 3 that, if docstrings are completely removed from the input problems, the performance drops dramatically across all models, demonstrating that docstrings serve a key role in accurate function completion.

Then, we compare the performance of the various LLMs when evaluated on the faulty variants of HumanEval versus the original dataset, which can be considered an improved version of them. We find that five out of six models surprisingly increased performance on at least one of the four faulty variants of HumanEval, indicating that a docstring improvement does not necessarily benefit code generation and it can even hurt performance (see Table 1 and Table 14, *Initial* columns).

Finally we evaluate the coder models on two new sets of strongly improved docstrings for the HumanEval problems, produced while having access to the ground-truth function completions (oracle docstrings). The first set of docstrings is produced by GPT-4[2] with access to the ground-truth solution (*'Oracle Hints'*), asking the model to give detailed hints on how to implement the function. The second set contains the true body of the function to be completed (*'Oracle Solutions'*), so that the task of the coder models simplifies to copy-pasting the solution. Our results, presented in Table 4, show that, both the Oracle Hints and the Oracle Solutions reformulations largely increase the performance of all models, with the Oracle Solutions being always superior to the Oracle Hints. Interestingly, no model achieves 100% pass rate even with the ground truth solution in-context, highlighting that current LLMs have strong limitations in using the information provided without any additional fine-tuning.

Figure 3 summarises our findings on the ability of the considered LLMs in leveraging docstrings for code generation: coder models may not reliably leverage small improvements in the docstrings, but clearly benefit from the overall information included in them. In particular, the performance with Oracle Hints can be considered as a good estimate of the potential of optimizing code generation via docstring reformulation; how to achieve such performance without access to the ground-truth solution to generate hints remains an open question.

[2]In all the experiments with GPT-4 we use GPT-4 Turbo, also referred as `gtp-4-1106-preview` in OpenAI API.

Table 3: **Performance with or without docstrings on the HumanEval benchmark.** These results highlight the importance of docstrings in the context of function completion.

| Models | With docs | Without docs |
|---|---|---|
| open_llama_7b_v2 | 13.4 (2.7) | 7.3 (2.0) |
| mpt-7b | 16.4 (2.9) | 9.8 (2.3) |
| starcoder | 33.5 (3.7) | 15.9 (2.9) |
| WizardCoder-3B | 35.4 (3.7) | 15.9 (2.9) |
| WizardCoder-Python-7B | 53.0 (3.9) | 19.5 (3.1) |
| WizardCoder-15B | 57.9 (3.9) | 20.1 (3.1) |
| Average | **34.9** | 14.8 |

**Q2. Limitations of the docstring reformulation methods in principle:** *Are the methods proposed guaranteed to improve the performance of the coder model used during training? Is there any guarantee that the improvement will transfer to other coder models?*

We identify the following challenges that any method for docstring reformulation faces: exploration, noisy learning signal and learning rule, overfitting and generalization.

The exploration challenge is about searching for the best docstring for a given input problem. The search space is huge, as generating docstrings is an open-ended text generation problem. In both the SFT and the OPRO methods we have an exploration process based on the stochastic sampling of the reformulations with an inductive bias encoded as extra information (the reformulation instruction $z$) in the prompt. This unfortunately doesn't provide any strong guarantees in the ability of finding the best docstrings, thus efficient exploration remains a key obstacle in the proposed methods.

The main signal for learning is the pass rate of the code completions generated by the coder model when receiving as input a given reformulation. We use the pass rate metric to identify the best reformulations and then we either explicitly increase their probability under the reformulator model, in the SFT method, or we reward the reformulation instruction associated with them, in the OPRO method. The challenge of noisy learning signal is that the coder model is stochastic and as such there is a lot of variance associated to the pass rate metric. Furthermore, the metric is computationally intensive to calculate, as one needs to invoke an LLM (the coder model) to obtain one or more code completions. Thus it might be too costly to arbitrarily reduce the noise if we have a tight computational budget.

Linked to the noise concern within the learning signal is the task of formulating a stable and efficient learning rule that harnesses the learning signal to maximize the task objective.

In the SFT method, for each input problem, we select the best reformulation and increase its probability. This encounters two main problems: first, with high noise levels, we can mistakenly select a non-optimal reformulation as the best one; second, it might be that none of the reformulations is as good as the initial docstring and this can lead to training instabilities. Thus, while in principle this method is quite flexible and it can perform a fine-grained optimization at the reformulation level, it is not well suited to learn from noisy signals.

In the OPRO method, given $n$ instructions with their corresponding scores, we prompt an LLM model to produce a better instruction; therefore, the learning rule is effectively a black box, where the new "learned" instruction is the one sampled by the model. This method can deal quite well with the noise in the learning signal, because each instruction's score is the pass rate averaged over all problems, rather than the pass rate for a single one. However, the OPRO method is reliant on an LLM for implementing the learning rule, which doesn't provide any guarantee of improvement.

Thus, how to design a learning rule to efficiently and robustly learn from a noisy signal is another open challenge in the docstring reformulation task.

Finally, if reformulations exclusively boost a particular coder model's performance while decreasing performance for most other models, they overfit to that model. Conversely, if reformulations enhance the performance of diverse coder models without specific tailoring, they demonstrate generalization across coder models. Empirically, we do not observe any sign of overfitting. We attribute this to the lack of backpropagation through the coder model in the proposed methods, which, we speculate, acts as a regularizer over the optimised reformulations and improves their generalizability. However, we do not have any theoretical guarantees against overfitting, nor in support of the generalizability of the optimized reformulations.

We conclude that the proposed methods face core shortcomings in exploring the reformulation

Table 4: **Performance with Oracle docstrings.** Model Performance when including in the docstring GPT-4-generated hints based on the ground truth solution (*'Oracle Hints'*) and when including in the docstring the ground truth solution (*'Oracle Solutions'*). We compare them with the performance of the coder models on the original HumanEval (*'Original'*).

| Models | Original | Oracle Hints | Oracle Solutions |
|---|---|---|---|
| open_llama_7b_v2 | 13.4 (2.7) | 22.6 (3.3) | 65.9 (2.7) |
| mpt-7b | 16.4 (2.9) | 27.4 (3.5) | 53.0 (2.9) |
| starcoder | 33.5 (3.7) | 40.2 (3.8) | 54.9 (3.7) |
| WizardCoder-3B | 35.4 (3.7) | 48.2 (3.9) | 78.7 (3.7) |
| WizardCoder-Python-7B | 53.0 (3.9) | 57.3 (3.9) | 79.9 (3.9) |
| WizardCoder-15B | 57.9 (3.9) | 59.8 (3.8) | 90.0 (3.9) |
| Average | 34.9 | 42.6 | 70.4 |

space and in learning from a noisy feedback signal.

**Q3. Limitations of the docstring reformulation methods in practice:** *Are there further practical considerations about our experimental setup that could affect the methods' success?*

In addition to the limitations discussed in **Q2**, the proposed methods may be limited by:

- The performance of the initial models used as reformulator as instruction optimizer.
- The choice of the methods' parameters, such as the amount $R$ of reformulations per input problem and the amount $C$ of code completions per reformulations.
- The hyperparameters used for language generation with LLMs and, for the SFT method only, the hyperparameters for the fine-tuning of the reformulator model.
- The specific prompt templates employed.

We pose special emphasis on the first point, as preliminary experiments ruled out a strong dependence from the other points. We run additional experiments to ablate the role of capability of the models employed as:

1. **Reformulator:** We evaluate the coder models on reformulations produced by GPT-4, instead of WizardCoder-Python-7B, prompted with the same reformulation instruction as in the SFT method. We use the original HumanEval dataset for this experiment.
2. **Instruction optimizer:** We reproduce the OPRO experiments for the original HumanEval dataset using GPT-4 as instruction optimizer model, instead of Llama-2-7b-chat.

Regarding the experiments on the reformulator, reported in Table 5, we find no significant difference in performance between the two models for the given reformulation instruction; our qualitative inspection of the generated reformulations supports the conclusion that the selected open-source model can generate docstring reformulations on par with GPT-4 in this specific context.

In the case of the instruction generator experiment, the results for GPT-4, presented in Table 6, are significantly worse than the ones obtained with the selected instruction optimizer. Qualitatively, GPT-4 suggests verbose reformulation instructions, often leading the reformulator to include in the documentation hallucinated information, e.g. about possible invalid inputs. This results in incorrect handling of edge cases in generated code completions and performance degradation.

In summary, our ablation studies in this section show that the limitations of the proposed methods are not linked with the quality of the models selected as reformulator and instruction optimizer, but rather to the points described in **Q2**.

Table 5: **Reformulator model ablation.** We compare the performance of our reformulator model, WizardCoder-Python-7B, without any SFT training against the one of GPT-4.

| Models | Reformulated by | |
|---|---|---|
| | WizardCoder | GPT-4 |
| open_llama_7b_v2 | 14.6 (2.8) | 15.9 (2.9) |
| mpt-7b | 16.4 (2.9) | 18.3 (3.0) |
| starcoder | 30.5 (3.6) | 28.7 (3.5) |
| WizardCoder-3B | 33.5 (3.7) | 37.2 (3.8) |
| WizardCoder-Python-7B | 52.4 (3.9) | 51.2 (3.9) |
| WizardCoder-15B | 54.2 (3.9) | 49.4 (3.9) |
| Average | **33.6** | 33.5 |

Table 6: **Instruction optimizer model ablation for OPRO method.** We compare the performance of our instruction optimizer model, Llama2-7B-chat, against the one of GPT-4, when utilising the OPRO method for 10 iterations. The results for GPT-4 are significantly worse than the ones obtained with the selected instruction optimizer.

| Models | Instruction optimizer | |
|---|---|---|
| | Llama2 | GPT-4 |
| open_llama_7b_v2 | 14.0 (2.7) | 8.0 (2.1) |
| mpt-7b | 17.1 (2.9) | 13.4 (2.7) |
| starcoder | 32.3 (3.7) | 25.0 (3.4) |
| WizardCoder-3B | 32.3 (3.7) | 30.5 (3.6) |
| WizardCoder-Python-7B | 56.1 (3.9) | 50.6 (3.9) |
| WizardCoder-15B | 54.9 (3.9) | 50.0 (3.9) |
| Average | **34.5** | 29.6 |

## B Hyper-Parameters used for the experiments

In the following section we report all the hyper-parameters used in our experiments. In Table 7 we report the parameter values for the SFT method, while in Table 8 the ones for the ORPO method. Furthermore, in Table 9 we report the parameters used for generating the reformulations, in Table 10 we report the ones for generating the code completions and finally in Table 11 the PEFT parameters for the SFT method.

In OPRO we also use a variable amount $n$ of past instructions and scores pairs, starting at an arbitrary value of $min(4, Z)$, where Z is the number of instructions used per iteration of the method, and increasing $n$ of 1 at every iteration. While we haven't ablated this choice, we speculate that smaller $n$ favour exploration by reducing the amount of patterns available to the instruction optimizer, while larger $n$ favour exploitation of features in common between successful past instructions.

Table 7: **Parameter values for SFT method.**

| Parameter | Value |
|---|---|
| Reformulation instructions ($Z$) | 1 |
| Reformulations per problem ($R$) | 2 |
| Code completions per reformulation ($C$) | 2 |
| Method iterations ($I$) | 10 |

Table 8: **Parameter values for OPRO method.**

| Parameter | Value |
|---|---|
| Reformulation instructions ($Z$) | 5 |
| Reformulations per problem ($R$) | 1 |
| Code completions per reformulation ($C$) | 1 |
| Method iterations ($I$) | 10 |
| Instruction optimizer temperature | 1.0 |
| Instruction optimizer top-$p$ | 0.8 |
| Instruction optimizer max tokens | 200 |

Table 9: **Parameters for generating reformulations with an LLM.** The OPRO method is always using the evaluation setting for the reformulations, while the SFT method uses the training and evaluation settings in the respective phases. Notice that the batch size does not affect performance and its choice depends on the hardware at disposal and the size of the reformulator model.

| Parameter | Training | Evaluation |
|---|---|---|
| Batch size | 32 | 32 |
| Temperature | 0.2 | 0 |
| Top-$p$ | 0.95 | N/A |
| Max tokens | 512 | 512 |

Table 10: **Parameters for generating code completions with an LLM.** The OPRO method is always using the evaluation setting for the reformulations, while the SFT method uses the training and evaluation settings in the respective phases. At evaluation time, the batch size is adjusted depending on the size of the coder model (however it does not affect performance).

| Parameter | Training | Evaluation |
|---|---|---|
| Batch size | 32 | 'custom' |
| Temperature | 0.2 | 0 |
| Top-$p$ | 0.95 | N/A |
| Max tokens | 768 | 768 |

Table 11: **PEFT parameters for training the reformulator in the SFT method.**

| Parameter | Value |
|---|---|
| LoRA r | 8 |
| LoRA $\alpha$ | 32 |
| LoRA dropout | 0.1 |
| Batch size | 4 |
| Gradient accumulation steps | 8 |
| Max sequence length | 768 |
| Learning rate | $2 \times 10^{-5}$ |
| Number of training epochs | 1 |

### B.1 Prompt templates

In both the SFT and the OPRO methods, we present the reformulation instruction $z$ and the input problem $x$ to the reformulator using the following template:

```
Below is an instruction that describes a task,
paired with an input that provides further
context. Write a response that appropriately
completes the request.
### Instruction:
{z}
### Input:
{x}
### Response:
{imports_and_def}
    """
```

This was used by Luo et al. (2023) to train the WizardCoder suite of code LLMs and was adopted in this work because, after preliminary tests, a WizardCoder model was chosen as our reformulator.

We append at the end of the template the library imports and the function definition (until the """ that marks the start of the docstring in Python programs) contained in $x$ to facilitate the reformulation task. This is because both the imports and the function definition are meant to remain fixed in the task and they provide additional context to generate a high-quality docstring in the reformulation.

The OPRO method uses also the following seed reformulation instructions:

1. `Improve the docstring of the following function using the best coding conventions.`
2. `Rephrase the following python code maintaining the function name and signature:`
3. `Clarify the following python function by rewriting the docstring:`
4. `Expand the documentation of the following python function suggesting how to implement it:`
5. `Rewrite the documentation of the following function:`

## C Independent replication of open-source LLMs on HumanEval

For the evaluation setup, in addition to the original coder model, we consider 5 other LLMs with model sizes ranging from 3B to 15B parameters: OpenLlama-2-7B-V2 (Geng and Liu, 2023; TogetherComputer, 2023), MPT-7B (MosaicML, 2023), starcoder (15B) (Li et al., 2023), WizardCoder-3B and WizardCoder-15B (Luo et al., 2023). These models were selected as a representative subset of the open-source LLM landscape and had to satisfy the following criteria:

1. They had to be available on Hugging Face.
2. Their prior performance on HumanEval had to be reported online from the authors of the models or from a trustworthy third party.
3. Their performance on HumanEval had to be reproducible by the authors of this work within reasonable limits.

The results for the models of which we could reproduce the reported performance are presented in Table 12, while the results for the models whose performance couldn't be replicated are presented in Table 13. Importantly, we do not imply that these models cannot yield the reported performance, but rather that model performance depends on many undocumented factors, such as the prompting strategy, the post-processing of the model output and the versions of the libraries used in the implementation.

We also tried to download teknium/Replit-v1-CodeInstruct-3B and Salesforce/xgen-7b-8k-base as other performing models whose performance was replicated with open-source code, but we encountered errors in using them with version 4.31 of the HuggingFace`transformers` library. Different library versions caused other LLMs to drop in performance, thus we ran all experiments with this version of the library.

## D Results for bad formatting HumanEval

We report in Table 14 the results for the SFT and the ORPO methods on the bad formatting variant of HumanEval.

## E Faulty variants of HumanEval

In Figure 4 we report an example of an input problem together with all the four different faults that we implement in this work.

### Original

```
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
```

### Misspelling

```
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XORg on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
```

### Distractor

```
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    Returns:
        list[int]: A list of integers
    """
```

### Ambiguity

```
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    """
```

### Bad formatting

```
def string_xor(a, b):
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    string_xor('010', '110')
    '100'
    """
```

Figure 4: **Examples of faulty docstrings.** Example of an input problem from the original HumanEval benchmark and of its faulty versions.

Table 12: **Independent verification of selected LLMs performance on HumanEval.**

| Model | Our Setup | Best Reproducible (GitHub) | Best Reported (Paper) |
|---|---|---|---|
| open_llama_7b_v2 | 13.4 | 14.0 | N/A |
| mpt-7b | 16.4 | 15.9 | 18.3 |
| starcoder | 33.5 | 34.6 | 33.6 |
| WizardCoder-3B | 35.4 | N/A | 34.8 |
| WizardCoder-Python-7B | 53.0 | N/A | 55.5 |
| WizardCoder-15B | 57.9 | 57.0 | 59.8 |

Table 13: **Independent verification of excluded LLMs performance on HumanEval.** WizardCoder-1B and the base model of Llama-2-7b work to some extent, but are not close enough to the reference values to be selected for the main studies. We were not able to make the CodeLlama family of models work and it is not clear where the big gap in performance comes from.

| Model | Our Setup | Best Reproducible (GitHub) | Best Reported (Paper) |
|---|---|---|---|
| WizardCoder-1B-V1.0 | 18.9 | N/A | 23.8 |
| Llama-2-7b-hf | 11.6 | 13.1 | 12.8 |
| CodeLlama-Python-7b | 3.0 | N/A | 38.4 |
| CodeLlama-Python-13b | 5.5 | N/A | 43.3 |
| CodeLlama-Instruct-7b | 7.9 | N/A | 34.8 |
| CodeLlama-Instruct-13b | 4.3 | N/A | 42.7 |

Table 14: **Results for HumanEval with bad formatting fault.** We removed these results from the main results because the fault introduced did not affect on average the performance of the selected LLMs.
* is used as coder model by SFT and OPRO also during training.

| Models | Initial | Reformulated | |
|---|---|---|---|
| | | SFT | OPRO |
| open_llama_7b_v2 | 12.1 (2.5) | 13.4 (2.7) | 11.6 (2.5) |
| mpt-7b | 17.7 (3.0) | 17.7 (3.0) | 16.5 (2.9) |
| starcoder | 35.4 (3.7) | 34.8 (3.7) | 32.9 (3.7) |
| WizardCoder-3B | 32.9 (3.7) | 31.7 (3.6) | 32.9 (3.7) |
| WizardCoder-Python-7B | 53.0 (3.9) | 56.1 (3.9) | 57.3 (3.9) |
| WizardCoder-15B | 58.5 (3.8) | 55.5 (3.9) | 55.5 (3.9) |
| Average | 34.9 | 34.9 | 34.5 |