Fine-tuning Language Models for Joint Rewriting and Completion of Code with Potential Bugs

Dingmin Wang[1*] **Jinman Zhao**[2*†] **Hengzhi Pei**[2] **Samson Tan**[3] **Sheng Zha**[3]
[1]University of Oxford [2]Amazon Web Services [3]Amazon AGI
`dingmin.wang@cs.ox.ac.uk`
`{jinmaz,philepei,samson,zhasheng}@amazon.com`

Abstract

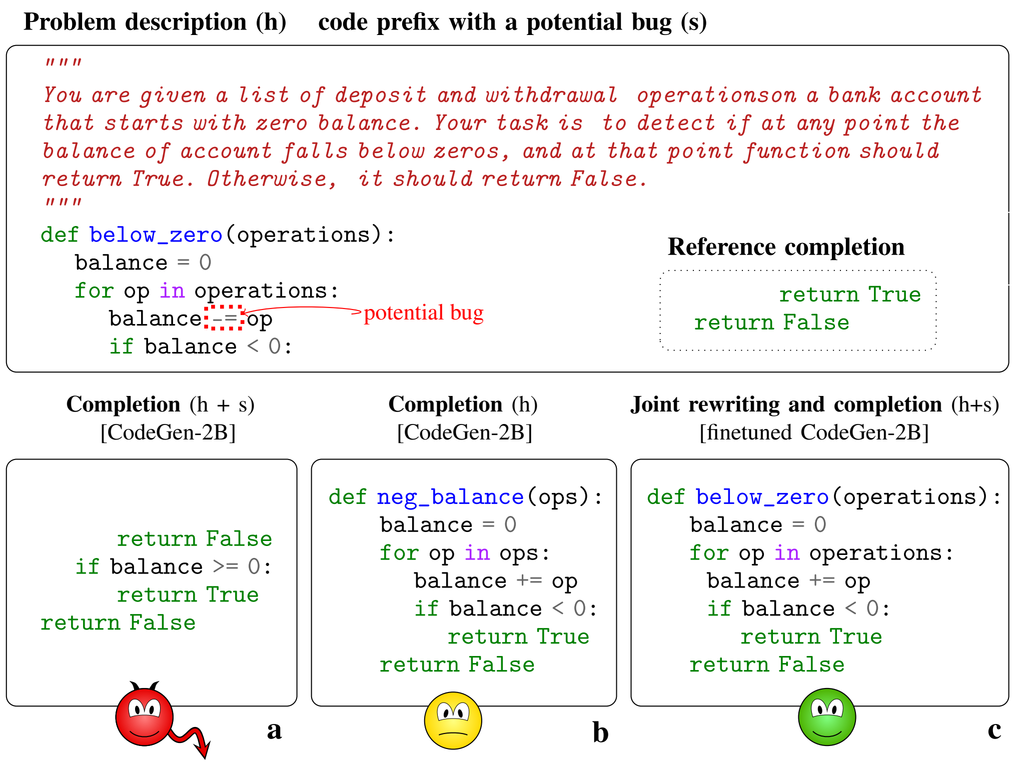
Handling *drafty* partial code remains a notable challenge in real-time code suggestion applications. Previous work has demonstrated shortcomings of large language models of code (CodeLLMs) in completing partial code with potential bugs. In this study, we view partial code as implementation hints and fine-tune CodeLLMs to jointly rewrite and complete partial code into functional full programs. We explore two strategies: *one-pass* generation and *multi-pass* iterative refinement. We construct new training and testing datasets using semantic-altering code transformations and iterative self-generations. We conduct comprehensive experiments over three representative open-sourced CodeLLMs – InCoder, CodeGen, and StarCoder. Results show that CodeLLMs fine-tuned using our approach achieve superior pass rates compared to the previous baselines across existing and newly-created benchmarks, effectively handle both potentially buggy and clean code, and largely preserve the integrity of the original partial implementations. We further present findings on the properties of the potential bugs we tested and on the design choices of our methods.

1 Introduction

Large language models of code (CodeLLMs), i.e. large language models trained on vast repositories of code, have demonstrated remarkable progress in program synthesis and other code-related tasks (Chen et al., 2021; Nijkamp et al., 2022; Li et al., 2022, 2023). Such models are commonly trained on finished and reviewed code, e.g. GitHub crawls of open-source projects. However, one common application of such code intelligence is suggesting implementations for in-progress code, which tends to be less refined and more error-prone. Previous work (Dinh et al., 2023) framed this inference challenge as "buggy-code completion" – completing code with potential bugs, where a potential bug is a piece of undesirable code in a code prefix with respect to some reference completion (see Section 2). They showed that CodeLLMs, albeit achieving a high pass rate over code generation benchmarks, fail to generate functional code in the presence of potential bugs. Figure 1 illustrates such an example. The marked potential bug in the partial code makes producing a functional completion a more challenging task. In this case, the completion model (CodeGen-2B) attempted to accommodate the potential bug by deviating from the reference completion, but the resulting completion is still functionally incorrect (panel **a**). Besides, we also show the output of the completion model (CodeGen-2B) using only the problem description (panel **b**). We can observe that the generated completion code segment accurately addresses the problem, but it deviates from the given partial code in terms of function names and certain variables, which may disalign with user expectations. Typically, users prefer suggestions that closely match their existing code structure and naming conventions. In this paper, we propose to fine-tune CodeLLMs to jointly rewrite and complete partial code. As shown in Figure 1, the fine-tuned CodeGen-2B gives a correct and full program (panel **c**). From the resulting completion, we can observe that the program aligns well with the given partial code while fixing the potential bugs.

We hypothesize that a user's partial code offers valuable information about their intent. It first helps clarify the problem, providing a richer specification than mere descriptions. It at the same time helps narrow down which solution idea, among a possibly vastly diverse solution space, the user wants to use to address the problem. Thirdly, it provides an implementation sketch for the solution idea, presumably making the task of finishing a functional program easier. It thus becomes more surprising to see that completing with the presence of potential

*Equal contribution.
†Currently at Morph.ai, `jinman@morph.ai`.


Problem description (h)
code prefix with a potential bug (s)
potential bug
Reference completion
Completion (h + s) [CodeGen-2B]
Completion (h) [CodeGen-2B]
Joint rewriting and completion (h+s) [finetuned CodeGen-2B]
a
b
c

```
"""
You are given a list of deposit and withdrawal  operationson a bank account
that starts with zero balance. Your task is  to detect if at any point the
balance of account falls below zeros, and at that point function should
return True. Otherwise,  it should return False.
"""
def below_zero(operations):
    balance = 0
    for op in operations:
        balance -= op
        if balance < 0:
```

```
            return True
    return False
```

```
            return False
        if balance >= 0:
            return True
    return False
```

```
def neg_balance(ops):
    balance = 0
    for op in ops:
        balance += op
        if balance < 0:
            return True
    return False
```

```
def below_zero(operations):
    balance = 0
    for op in operations:
        balance += op
        if balance < 0:
            return True
    return False
```

Figure 1: Buggy-code completion with three different paradigms: completion with both the problem description (h), completion with only the problem description (h), and the code prefix (s) and joint rewriting with both the problem description (h) and the code prefix (s). **a** is incorrect, **b** is correct but disaligns with the given partial prefix (the function name and the variable names are different), and **c** is correct and align with the given partial prefix.

bugs gives an even worse pass rate than generating code from only specifications (Dinh et al., 2023).

In this paper, we focus on tuning existing CodeLLMs with the task of generating a full functional programs given a problem specification and a code prefix, in which potential bugs may present (Section 2). The intuition is to treat code prefixes as implementation hints and allow deviations from it. We explore two strategies of joint rewriting and completion of partial code (Section 3): *one-pass* generation where a model directly attempts to generate a functional full program from a code prefix, and *multi-pass* iterative refinement where a model iteratively refines its previous outputs. Regarding the construction of the required training datasets, we design an automatic way to injecting multiple types of semantic-altering code transformations to code prefixes and use the original reference code as generation targets (Section 4). We evaluate our models in terms of pass rates against **b-HumanEval** and **b-FixEval** benchmarks from Dinh et al. (2023), as well as two newly constructed evaluation datasets **b-MBPP** and **b^2-MBPP** (Section 4.4). Extensive results (Section 5) using CodeGen-(2B, 16B) (Nijkamp et al., 2022), InCoder-(1B, 6B) (Fried et al., 2022), and StarCoder (15.5B) (Li et al., 2023) suggest that the *multi-pass* iterative inference strategy outperforms the *one-pass* generation strategy, which in turn surpasses previous *post-hoc* baseline mitigations (Dinh et al., 2023) in terms of pass rates on both buggy and reference code prefixes and in terms of preserving the given partial code. As illustrated in Figure 1, our approach not only provides a correct solution that aligns with the given partial code but also rectifies the potential bug, thus resulting in a more desirable outcome.

Contributions We (i) first attempt the joint rewriting and completion of code with potential bugs using two inference strategies facilitated by fine-tuning, (ii) construct 2 training datasets and 2 new benchmark datasets via code transformation, (iii) improve the performance of all tested CodeLLMs in terms of pass rates, preservation of partial code and performance on reference code, and (iv) provide various findings on the property of potential bugs and design choices of our methods.

2 Task

In this section, we look into the task of transforming a code prefix s into a functional full program t, conditioning on a task specification h. Motivated by the scenario of real-time code suggestion, where a user's in-progress code can be less refined and more error-prone than the final revised code, the code prefix here may contain potential bugs.

A potential bug is a span u in a code prefix s, such that, compared to the reference program $\hat{t} := \hat{s} :: \hat{c}$ that implements h, s and $\hat{s}$ only differs in u and that program $t := s :: \hat{c}$ fails h. $::$ denotes code concatenation. In other words, u is a bug in program t. We call $\hat{s}$ a reference prefix, $\hat{c}$ a reference completion, and s a (code) prefix with potential bugs, or a (potentially) buggy prefix. Note that, although we refer to s as "buggy prefix" for brevity, s is not buggy per se without a reference completion, because "bugginess" is only defined over a full program. The task was previously studied as "buggy-code completion" by Dinh et al. (2023, Section 2). In particular, since the code prefix s manifests a user's implementation intent, a desideratum is to minimize deviations from s. We quantify this by the similarity between generated solution t and the reference solution $\hat{t}$ (see Section 5.2), because the prefix-completion boundary is not always well-defined in the generated program t after the joint rewriting and completion.

3 Method

We describe two inference strategies for joint rewriting and completion of code, *one-pass* generation and *multi-pass* iterative refinement, as well as how we fine-tune a CodeLLM with such strategies.

3.1 One-Pass Generation

Given a problem specification h and a code prefix s as input, the code completion model decodes a full program t autoregressively in one pass.

On a dataset $\mathcal{D}_1$ that consists of triplets $(h, s, \hat{t})$, where $\hat{t}$ is a reference (full) solution, we fine-tune a base CodeLLM using the auto-regressive language modelling loss (supervised fine-tuning, SFT). That is, we maximize

$$\Pr(\hat{t} \mid h, s) = \prod_{i=1}^{|\hat{t}|} p(\hat{t}_i \mid h, s, \hat{t}_{<i}). \tag{1}$$

The resulting model is denoted as $\mathcal{M}_1$. The construction of $\mathcal{D}_1$ is detailed in Section 4.

3.2 Multi-pass Iterative Refinement

CodeLLMs might not generate a good solution on their first try, especially when the potential bugs are difficult to identify. Motivated by how humans proofread and refine their written texts, we propose to improve the generated solution using the *iterative refinement* strategy, a fundamental characteristic of human problem-solving (Flower and Hayes, 1981). *Iterative refinement* has demonstrated efficacy across many code-related tasks (Madaan et al., 2023; Reid and Neubig, 2022; Welleck et al., 2022). In our setting, the primary objective is to allow the model multiple opportunities to detect potential bugs and to generate suitable corrections (**rewritings**). Once the potential bugs are fixed, corrected partial code will enable the model to produce accurate and relevant **completions**.

Inference For this strategy, we let a model refine its solution iteratively by generating a new solution based on its previous generated output together with the original specification. Specifically, given a specification h and a code prefix s, model $\mathcal{M}$ generates a solution $t = t^{(J)}$ following

$$t^{(j)} = \mathcal{M}(h, \text{truncate}_s(t^{(j-1)})) \tag{2}$$

for $j = 1, \ldots, J$, where $t^{(0)} = s$, J is the number of iterations and $\text{truncate}_s(\cdot)$ modifies the generated solution $t^{(j-1)}$ by truncating it to a partial prefix, which retains the same number of lines of code as specified by s.

Training We use a two-phase fine-tuning approach to facilitate a model's ability to refine its output. The training process is illustrated in Figure 2. In the first phase, we obtain fine-tuned CodeLLM $\mathcal{M}_1$ following Section 3.1. Then, we use $\mathcal{M}_1$ to generate dataset $\mathcal{D}_2$ for the second phase. Specifically, for every sample $(h, s, \hat{t}) \in \mathcal{D}_1$, we create new training samples by collecting model outputs during iterative refinement as input code prefixes.

$$\mathcal{D}_2 = \bigcup_{(h,s,\hat{t}) \in \mathcal{D}_1} \{(h, \text{truncate}_s(t^{(j)}), \hat{t})\}_{j=1}^{J} \tag{3}$$

where $t^{(j)}$ is obtained following Equation (2) using $\mathcal{M}_1$, and J and $\text{truncate}_s(\cdot)$ carry the same meaning as before. After that, we start the second-phase fine-tuning over $\mathcal{M}_1$ using the newly constructed $\mathcal{D}_2$, resulting in a new model $\mathcal{M}_2$.

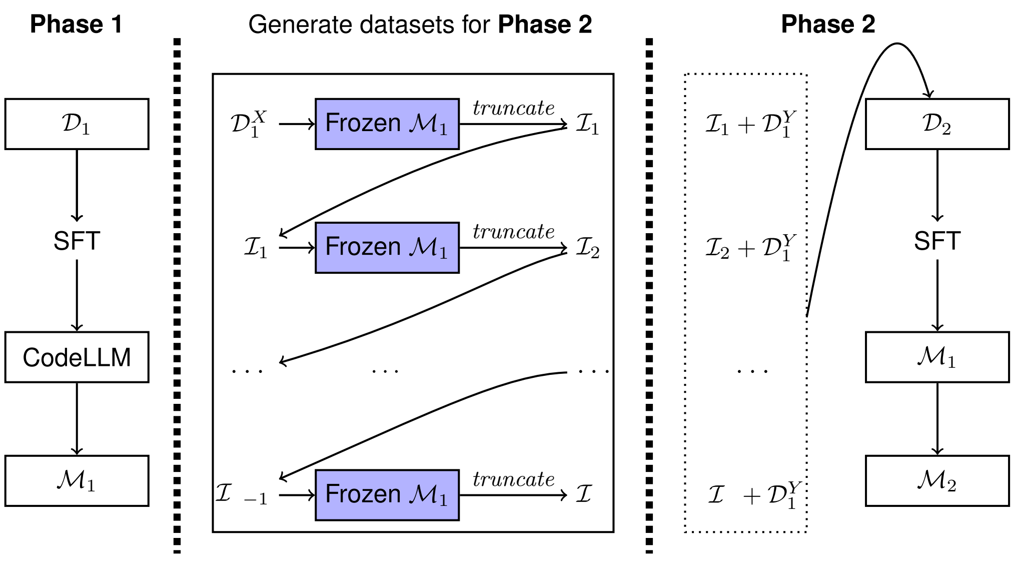


Figure 2: Overview of the two-phase training framework. $\mathcal{D}_1$ consists of triplets consists of triplets $(h, s, \hat{t})$, and $\mathcal{D}_1^X$ represents the input part of $\mathcal{D}_1$, each of which is of the form (h, s), and $\mathcal{D}_1^Y$ represents the output part of $\mathcal{D}_1$, each of which is of the form $(\hat{t})$. $\mathcal{I}_i + \mathcal{D}_1^Y$ represents the concatenating the generated input part with the target output to form the triplet $(h, s, \hat{t})$ for the fine-tuning.

Choice of J We fix the number of iterations J to be 3, which is determined by our preliminary experimental results. An adaptive choice of J based on the input may offer better trade-offs between quality and efficiency. We leave further explorations of the choice of J to future work.

4 Construction of Datasets

We create labeled instances for joint rewriting and completion of partial code using truncation and semantic-altering code transformations. Section 4.1 – 4.3 describe the construction of $\mathcal{D}_1$. Section 4.4 describes testing datasets.

4.1 Buggy Code Prefixes

An overview of the construction process is depicted in Figure 3. Starting with a reference program $\hat{t}$ comprising $|\hat{t}|$ lines of code, we first parse[1] it into an abstract syntax tree (AST). Next, we apply a semantic-altering AST transformation $\mathcal{F}$ that yields a valid AST. We then unparse it to get a buggy program t, where the bug is marked by the span corresponding to transformed AST nodes. This ensures that t remains syntactical because it still corresponds to a valid AST. We locate the smallest L such that line 1 to L contains the bug span. Finally, we take the first S lines of t as a potentially buggy prefix s, where S is randomly chosen from $\{L, L+1, \ldots, |\hat{t}|\}$.

[1] docs.python.org/3/library/ast.html

Choice of transformations Given the diversity of potential bugs, it is impractical to attempt cataloging every conceivable error. We identify common bug types by analyzing both successful and failed submissions in CodeNet (Puri et al., 2021), a large-scale code datasets consists of user submissions to coding problems. We randomly selected submissions from $1,000$ different users to uncover some common potential bugs types. In total, we have identified seven distinct types of potential bugs. Correspondingly, we devised 7 AST-based transformations for injecting bugs and obtaining buggy partial code, namely Variable Renaming (`v-Rename`), Keywords Removal (`k-Remove`), Numerical Value Change (`nv-Change`), While and If Swapping (`wi-Swap`), Condition Removal (`c-Remove`), Branch Removal (`b-Remove`) and Operator Change (`o-Change`). Details can be found in Appendix A.1.

Discussions Although we only applied the construction procedure to Python code, however, the operation of code transformation and truncation is generic and can be easily adapted for other programming languages. The spectrum of potential bugs that our proposed transformations can simulate is not exhaustive, owing to the complexities previously discussed. Despite this, our dataset encompasses the majority of bug types documented in bug-fixing literature, such as the work by Karampatsis and Sutton (2020). Furthermore,

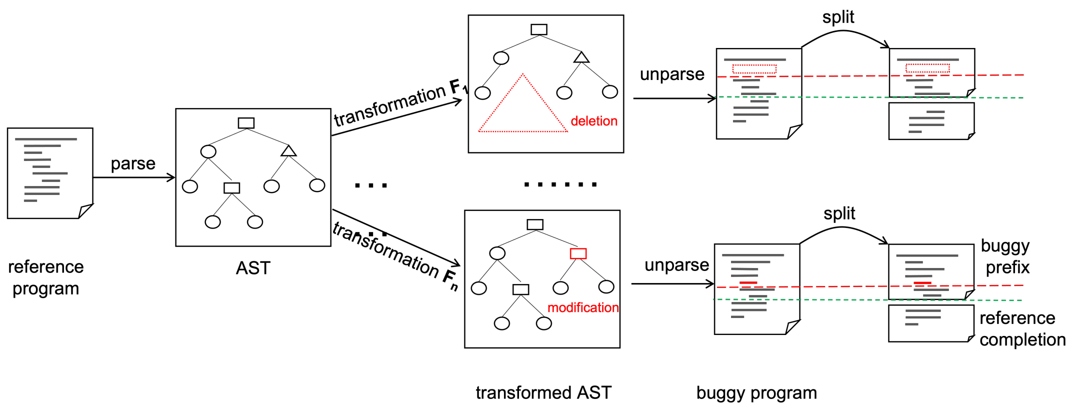


Figure 3: Obtaining buggy partial code via bug injection. The dotted red lines represent the last lines of bug spans. The dotted green lines represent the splitting locations, which is randomly chosen after the bug spans. Given that our focus is on code completion from partial code inputs, we have opted to discard solutions with only one-line code.

there is a significant overlap with types of bugs found in established testing datasets like Buggy-HumanEval (Dinh et al., 2023), which affirms the relevance and utility of our approach.

4.2 Clean Code Prefixes

Dinh et al. (2023, Section 4.4) showed that improving the performance on buggy code prefixes can degrade the performance on reference code prefixes. To prevent such performance degradation, we additionally include the prefixes from reference program for training. For a specification h and a reference program $\hat{t}$, we take the first $\lceil r|\hat{t}|\rceil$ lines as code prefix s and form a training sample $(h, s, \hat{t})$, where ratio r takes from $\{0.2, 0.4, 0.6, 0.8\}$, generating 4 distinct training samples.

4.3 Training Dataset

To align CodeLLMs with the capability to handle common formats of coding solutions, we take problems and reference solutions from CodeContests (Li et al., 2022) for the *Standard Input Format* where a solution is a standalone program that interacts with standard input-output, and from MBPP (Austin et al., 2021) for the *Function-Call based format* where a solution is a function implementation. See examples in Appendix A.3. Both datasets contains coding problems and test cases for each problem. CodeContests contains user submitted solutions to each problem; while MBPP comes with a reference solution to each problem. From CodeContests, we randomly sample 3,000 problems; and for each of problem, we randomly sample one accepted solution as the reference solution. From MBPP, we use 875 problems for constructing training instances and reserve the remaining 100 problems for constructing testing instances (see below). Then, we apply the aforementioned procedures to obtain training dataset $\mathcal{D}_1$, which contains 100k samples including both buggy and clean training samples.

4.4 Testing Dataset

To facilitate direct comparisons, we test our methods on benchmarks from Dinh et al. (2023): **b-HumanEval** constructed by injecting operator flips to HumanEval reference solutions and **b-FixEval** constructed by contrasting rejected and accepted human solutions from FixEval (Haque et al., 2023). In addition, we construct two new testing datasets for evaluating joint rewriting and completion of code with potential bugs using the procedure described in Section 4.1. **b-MBPP** and **b^2-MBPP** are both constructed from 100 MBPP (Austin et al., 2021) samples held out from the training set construction. **b-MBPP** contains only instances with only one potential bugs while **b^2-MBPP** contain instances with two potential bugs.

Details of training and testing dataset statistics used in our experiments are listed in Table 1. Note that the size of $\mathcal{D}_2$ is not three times the size $\mathcal{D}_1$. This is because, for the clean instances, we construct only one pair per instance.

5 Experiments

5.1 Experimental Setting

Base completion models We select several public CodeLLMs as base code completion models, including CodeGen-(2B, 16B) (Nijkamp et al., 2022),

	$\mathcal{D}_1$	$\mathcal{D}_2$	**b-HumanEval**	**b-FixEval**	**b-MBPP**	**b^2-MBPP**
# of total instances	100,000	207,272	1,896	292	976	500
# of buggy instances	53,636	128,774	1,896	292	650	500
# of clean instances	46,364	78,498	0	0	326	0
# of potential bug(s)	0/1	0/1	1	-	1	2

Table 1: Statistics of our training and testing datasets. **b-FixEval** is constructed from the real submission, so the number of potential bugs could not be quantified.

autoregressive language models released for program synthesis; InCoder-(1B, 6B) (Fried et al., 2022), decoder-only Transformer models trained on code using a causal-masked objective; and StarCoder (15.5B) (Li et al., 2023), a recently released decoder-only transformer model achieved by fine-tuned StartCoderBase for 35B Python tokens.

Experimental settings We fine-tune base CodeLLMs on our constructed datasets. We use the AdamW optimizer (Loshchilov and Hutter, 2017) with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$, a per-GPU batch size ranging from 2 to 32. We fine-tune for 5 epochs. We use DeepSpeed (Rasley et al., 2020) together with the ZeRO optimizer (Rajbhandari et al., 2020) to reduce memory consumption while training large models. Models are fine-tuned on Amazon p4de.24xlarge with 8 A100 GPUs.

Baseline methods Beyond the naiv̈e generation from code context, we adopt three post-hoc mitigation methods from Dinh et al. (2023) as baselines: removal $\rightarrow$ completion mitigates the negative effect of potential bugs by directly removing the partial code from input. While it guarantees that the input contains no bug, it sacrifices useful context information; completion $\rightarrow$ rewriting uses RealiT (Richter and Wehrheim, 2022), a program repair model, to fix the generated solution from a CodeLLM; rewriting $\rightarrow$ completion uses InCoder-6B (Fried et al., 2022) and a likelihood-based heuristic to identify and rewrite potential bugs before the code prefix was sent for completion.

Evaluation metrics We measure the functionality of a completed program by executing it against the provided test cases on the testing problems. Following the common protocol for evaluating code generation (e.g. Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Fried et al., 2022), we measure the `pass@k` ($\uparrow$) for each instance as follows:

$$\texttt{pass@}k := 1 - \binom{n-c}{k} \Big/ \binom{n}{k}$$

where n generated solutions are sampled from a model and c of them pass all the tests. We use $n = 100$ and $k = 1, 10, 100$. Unless otherwise specified, we use the default HuggingFace generation parameters for all the methods. As a reference solution to a problem can derive multiple input cases, to avoid the dominance of certain problems, we average the $\texttt{pass@}k$ numbers within each problem and then average across all problems.

5.2 Experimental Results

Performance on buggy code prefixes. Table 2 shows the pass@1 results on completing buggy code prefixes on the five testing datasets. Our fine-tuned models consistently outperform the other baseline models across different model sizes. It indicates that task-specific fine-tuning can help the models better leverage the useful information from the buggy partial codes and improve the quality of the generated code solutions. By contrast, rewriting $\rightarrow$ completion and completion $\rightarrow$ rewriting cannot effectively leverage the useful information from buggy prefixes. When the model is large, e.g. CodeGen-16B and StarCoder, removal $\rightarrow$ completion, which disregards code prefixes, can even significantly outperform rewriting $\rightarrow$ completion and completion $\rightarrow$ rewriting. Meanwhile, the iterative refinement model $\mathcal{M}_2$ consistently outperforms the one-pass generation model $\mathcal{M}_1$, which proves the effectiveness of the iterative refinement strategy.

Moreover, our methods are less affected by the number of potential bugs in the partial codes. On the **b^2-MBPP** dataset where the partial code contains two bugs, the rewriting $\rightarrow$ completion and completion $\rightarrow$ rewriting methods significantly perform worse than they do on the **b-MBPP** dataset. For example, when the base model is InCoder-6B,

	Methods	b-FixEval	b-HumanEval	b-MBPP	b^2-MBPP
CodeGen-2B	removal → completion	8.6	9.3	6.7	6.7
	naive completion	4.3	3.1	4.9	4.4
	rewriting → completion	7.2	**24.9**	8.9	5.4
	completion → rewriting	4.7	23.6	12.8	3.8
	One-pass Generation ($\mathcal{M}_1$)	8.1	19.2	17.4	10.4
	Iterative Refinement ($\mathcal{M}_2$)	**9.6**	20.4	**19.3**	**14.2**
InCoder-6B	removal → completion	6.4	6.9	9.1	9.1
	naive completion	1.8	1.0	2.4	1.5
	rewriting → completion	5.1	16.4	11.2	6.2
	completion → rewriting	3.0	25.2	19.2	4.9
	One-pass Generation ($\mathcal{M}_1$)	7.3	25.5	21.6	18.7
	Iterative Refinement ($\mathcal{M}_2$)	**8.8**	**26.8**	**23.3**	**21.3**
CodeGen-16B	removal → completion	28.2	30.4	35.6	35.6
	naive completion	16.4	17.5	13.1	12.6
	rewriting → completion	21.6	24.5	27.9	24.9
	completion → rewriting	18.3	27.1	26.6	23.6
	One-pass Generation ($\mathcal{M}_1$)	29.1	31.3	37.2	34.9
	Iterative Refinement ($\mathcal{M}_2$)	**31.2**	**33.2**	**39.1**	**37.1**
StarCoder	removal → completion	29.9	35.4	40.5	40.5
	naive completion	17.1	18.2	14.2	13.8
	rewriting → completion	22.1	26.4	30.1	22.2
	completion → rewriting	21.0	25.2	28.2	18.6
	One-pass Generation ($\mathcal{M}_1$)	30.1	36.1	41.8	41.0
	Iterative Refinement ($\mathcal{M}_2$)	**31.8**	**37.8**	**43.9**	**42.6**

Table 2: Pass@1 (%) on five benchmarks with buggy partial codes. The best result on each benchmark are in bold and the second best result are underlined under the same baseline model.

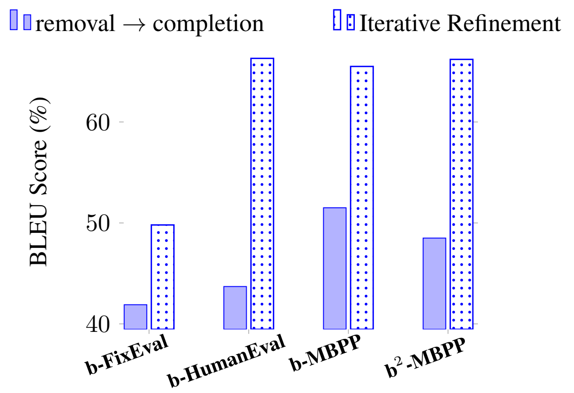


Figure 4: BLEU scores for removal → completion and our fine-tuned model (CodeGen-16B) over four benchmarks. The solutions used for the calculation are all correct , namely, passing the test cases.

	Methods	r-FixEval	r-HumanEval
CodeGen-2B	naive completion	**37.8**	**54.9**
	rewriting → completion	31.0	49.6
	completion → rewriting	19.4	22.7
	Iterative Refinement ($\mathcal{M}_2$)	37.2	52.1
CodeGen-16B	naive completion	**50.2**	**63.4**
	rewriting → completion	32.7	51.5
	completion → rewriting	20.9	24.2
	Iterative Refinement ($\mathcal{M}_2$)	45.9	59.7

Table 3: Pass@1 (%) on benchmarks with clean partial code settings. The best result on each benchmark are in bold and the second best result are underlined.

rewriting → completion and completion → rewriting become worse than removal → completion on the **b^2-MBPP** dataset. However, our methods still perform the best on the **b^2-MBPP** dataset.

We note from Table 2 that when the model size is large, the removal → completion method shows comparable performance to our fine-tuned models. However, we find that the solutions generated by the removal → completion method often exhibit significant deviations from the given code prefixes.

We quantify this misalignment using the BLEU score. Figure 4 shows that when using CodeGen-16B as the base model, iterative refinement consistently produces the solutions with higher BLEU scores than those generated by removal → completion on all test sets. This further proves our method's ability to generate solutions better aligned with the provided partial code, which we consider a desired property in practice to cause less surprise and interruption when a user gets code suggestions.

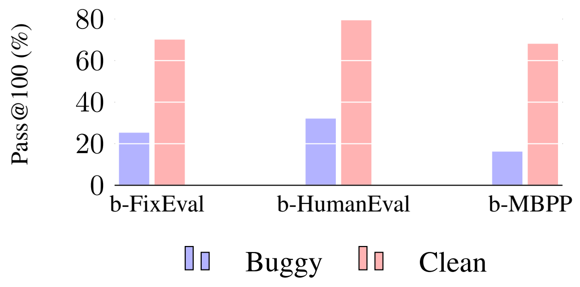


Figure 5: Pass@100 (%) on three benchmarks with naive completion using CodeGen-16B.

Performance on reference code prefixes We recognize that partial code may not always contain potential bugs, and expect that whatever methods for this task should still perform well when potential bugs are not present. Table 3 presents the pass@1 results of various methods on **r-FixEval** and **r-HumanEval** benchmarks where the methods are provided with the reference code prefixes. We observe that for both CodeGen-2B and CodeGen-16B, completion $\rightarrow$ rewriting and rewriting $\rightarrow$ completion perform significantly worse than the naive completion in this setting. rewriting $\rightarrow$ completion may do wrong localization when the code prefix does not contain a bug and may introduce new errors during its infilling-based line replacement. As for completion $\rightarrow$ rewriting, we find that the code repairer may inadvertently transform an already correct solution to an incorrect one. Our method shows desirable behavior that its performance is much closer to that of the naive completion.

6 Analysis

Are buggy code prefixes recoverable? A partial code snippet, in itself, is not inherently buggy, as the notion of bugginess is ill-defined without the context of a complete program. To quantify the extent to which a partial code can be considered truly "buggy", we measure the `pass@100` scores for the code completion results from a substantial large CodeLLM (CodeGen-16B). If none of 100 generated solutions is able to pass all the test cases, it indicates that the partial code is impossible, or at least very difficult, to reach a correct solution by merely appending a suffix. In other words, these potential bugs are considered as "not recoverable". Modifications over the given partial code may be necessary to reach a correct solution. Figure 5 shows that even CodeGen-16B, a substantially large language model for code with up to 16.1B parameters, has lower `pass@100` for the buggy partial code than that for the clean partial code. This suggests that obtaining a fully functional program merely by appending a suffix to the given buggy partial code is extremely challenging. To some extent, it indicates that rewriting the partial code with potential bugs fixed before the completion operation is necessary as to achieve a functional program.

Impact of different types of potential bugs Figure 6 shows the results for different bug types obtained through three distinct methods. Notably, completion $\rightarrow$ rewriting exhibits an exceptionally low `pass@1` rate for the three bug types: `k-Remove`, `c-Remove`, and `b-Remove`. These particular bug types are attributed to code removal operations. The bug fixer, RealiT (Richter and Wehrheim, 2022), employs a localization and repair module, which first identifies the bug's location and then seeks a suitable replacement token for repair. Consequently, for the bugs arising from the removal of code segments or entire lines, the localization module struggles to pinpoint the bug. In cases where the code removal operation occurs at the last line of the partial code, there is no bug to fix, which explains why completion $\rightarrow$ rewriting still manages to generate solutions satisfying the specification in some instances. Similar challenges are observed for rewriting $\rightarrow$ completion, as it also needs to identify lines of code containing bugs before utilizing the infilling operation. Therefore, both of these post-hoc baseline methods have limitations in addressing specific bug types.

By contrast, our fine-tuned model generates solutions from scratch while adhering to the given specification. It treats the provided partial code as a *soft constraint* to guide the generation process. This approach offers greater flexibility and wider applicability across various bug types.

7 Related Works

7.1 LLMs for code completion

Large Language Models (LLMs) (Touvron et al., 2023; Team et al., 2023; Achiam et al., 2023) have recently received tremendous attention and shown impressive success. By leveraging extensive pre-training on vast amounts of internet data and further fine-tuning with human-annotated instruction data (Ouyang et al., 2022), these models achieved state-of-the-art (SOTA) zero-shot performance across diverse tasks. This trend is also

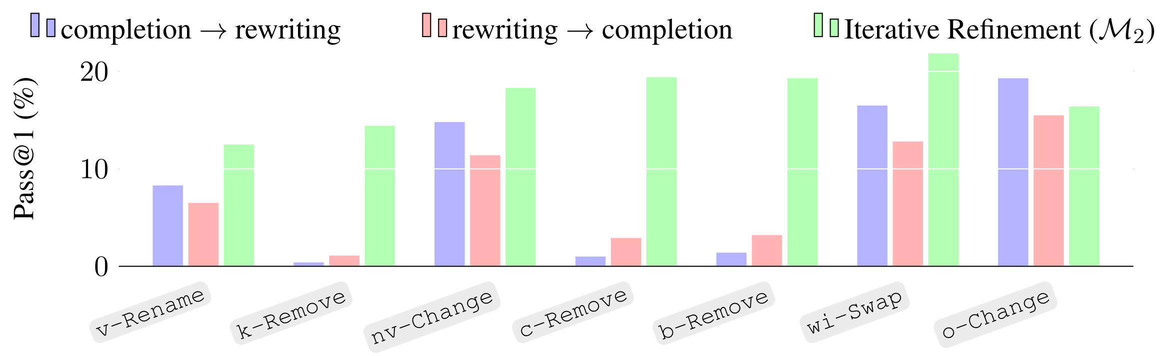


Figure 6: Pass@1 for different type of bugs under three different methods (completion → rewriting, rewriting → completion, Iterative Refinement ($\mathcal{M}_2$)). The baseline model is CodeGen-2B.

observed in the code domain, where numerous large language models for code (CodeLLMs) have been trained to tackle the challenges associated with code-related tasks. These CodeLLMs leverage billions of trainable parameters and access to vast repositories of publicly available source code. For example, AlphaCode (Li et al., 2022) claimed to outperform half of the human participants in real-world programming competitions, while CodeX (Chen et al., 2021) empowers Copilot, a widely used AI programming assistant, to offer real-time coding suggestions. Other noteworthy open-source CodeLLMs include WizardCoder (Luo et al., 2023), StarCoder (Li et al., 2023), CodeT (Wang et al., 2021), CodeGEN (Nijkamp et al., 2022), and InCoder (Fried et al., 2022). In this paper, we harness the capabilities of these open-source pre-trained code language models and fine-tune them using a collection of code datasets automatically constructed to enhance code generation, particularly in scenarios involving potential bugs.

7.2 Automatic program repair

Repairing for complete code The research on automatic bug detection and fixing (Vasic et al., 2018; Chen et al., 2019; He et al., 2022; Karampatsis and Sutton, 2020; Richter and Wehrheim, 2022) aims at relieving the programmers from the enormous effort of finding and fixing programming bugs. Fine-tuning pretrained code language models has proven to be an effective strategy. For example, Mashhadi and Hemmati (2021) fine-tuned CodeBERT (Feng et al., 2020) to automatically generate the fix codes, and He et al. (2022) fine-tuned CuBERT (Kanade et al., 2020) in a two-phase strategy in order to reduce the distribution shift existing in learning-based bug detectors. Allamanis et al. (2021) proposed BUGLAB, an approach to co-train bug detection and repair models in a supervised way. Despite the similarity, these previous studies mainly focus on code repairing, targeted at fixing bugs from complete programs, while we study potential bugs from partial code.

Repairing for in-progress code Unlike program repair for complete code, addressing issues in works-in-progress poses a relatively ambiguous challenge and remains largely unexplored in the field. In a pioneering study, Li et al. (2021) considered the tasks of localizing and repairing variable misuse for work-in-process code. More recently, Dinh et al. (2023) experimented several post-hoc methods to mitigate the adverse effects when CodeLLMs try to complete partial code with potential bugs. However, those methods either lack the ability to rewrite the partial code, or lack the adaptations needed to better accustom to the presence of potential bugs in partial code. Our proposed fine-tuning for joint rewriting and completion addresses these two limitations and shows improvements in pass rate and preservation of the original code.

8 Conclusion

In this paper, we study the task of generating a complete functional program given a specification and an in-progress code prefix with potential bugs. To align CodeLLMs with the task of jointly rewriting and completing such code prefixes, we construct a new training dataset by injecting various potential bugs. Beyond the standard fine-tuning and generation workflow, we apply an *multi-pass* iterative refinement strategy to enhance code generation performance. Our experiments demonstrate that our methods outperform previous baseline methods on both existing evaluation benchmarks and the new testing datasets we have constructed.

9 Limitations

In this study, we evaluate methods on benchmarks constructed from synthetic bugs or user submission to coding problems, which does not reflect the full complexity of practical applications. For example, actual software development can work with multiple files within a complex project environment. We only conduct experiments in Python. While our methods do not rely on any language-specific features, experiments on other programming languages would provide evidence for cross-language generalizability of our findings.

We measure the similarity between a generated solution and its reference solution as a proxy for preservation of implementation intent. The text level similarity may not reflect high-level alignment in semantics or pragmatics. Current users generally do not expect a coding assistant to overwrite their code when making completion suggestions. What behavior should we target to increase users' productivity and satisfaction remains not only a science but also a design problem.

In real-life coding scenarios, the occurrence of bugs in an in-progress code can differ among programmers. As a preliminary exploration, this paper focuses on a relatively straightforward scenario wherein only one or two potential bugs are considered. Besides, the seven transformations are our concerted effort to cover and replicate realist bugs. They are proposed based on our heuristic analysis and insights from surveys, with some of them already identified in prior related studies (Karampatsis and Sutton, 2020; He et al., 2022).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.

Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. *arXiv preprint arXiv:2306.03438*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Linda Flower and John R Hayes. 1981. A cognitive process theory of writing. *College composition and communication*, 32(4):365–387.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

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

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. 2022. On distribution shift in learning-based bug detectors. In *International Conference on Machine Learning*, pages 8559–8580. PMLR.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR.

Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xuechen Li, Chris J Maddison, and Daniel Tarlow. 2021. Learning to extend program graphs to work-in-progress code. *arXiv preprint arXiv:2105.14038*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

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. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE.

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

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506.

Machel Reid and Graham Neubig. 2022. Learning to model editing processes. *arXiv preprint arXiv:2205.12374*.

Cedric Richter and Heike Wehrheim. 2022. Can we learn from developer mistakes? learning to localize and repair real bugs from real bug fixes. *arXiv preprint arXiv:2207.00301*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2018. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*.

A Details of Data Construction

A.1 AST Transformations

Variable Renaming (`v-Rename`) In coding, it is common for us to inadvertently introduce errors when reusing a previously defined variable. These errors can arise from a lack of attention, leading to typographical mistakes like 'friend' becoming 'friends', or they may occur due to incorrect auto-completion suggestions from coding assistant tools. In our method, we implement a process where we randomly select a variable and intentionally introduce a potential bug. This is achieved either by substituting it with another variable defined before it [2] or by creating a new variable with the addition or the deletion of the character 's' at the end of the variable.

Keywords Removal (`k-Remove`) Keywords such as 'continue', 'break', and 'return' are frequently used and serve a crucial role in ensuring the correctness of functions or programs. The absence of these keywords can lead to programs or functions failing to terminate properly. Identifying such bugs can be challenging and may consume a significant amount of a programmer's time for the debugging.

Numerical Value Change (`nv-Change`) Numeric values, real and integer numbers such as 3.68 or 52, are commonplace elements in our code that require meticulous attention to prevent challenging-to-detect programming errors. For instance, on a keyboard, adjacent numbers are closely positioned, making it easy to unintentionally input incorrect values by pressing the wrong key. These errors can go unnoticed until we begin running our code. Specifically, when dealing with single-digit numeric values, we replace them with their neighboring digits (for 1, we use 2 as the replacement, and for 0, we use 9); for multi-digit numeric values, we randomly select a digit to replace, following the same principle, e.g., we replace 3.45 with 3.35, in which the second digit 4 in 3.45 is replaced with 3.

While and If Swapping (`wi-Swap`) A 'while' block is employed to repeatedly execute a loop operation until the specified condition is no longer met, whereas an 'if' block is executed only once when the specified condition is satisfied. In this method, we replace 'while' with 'if' while preserving the existing code within the block. Such a programming bug could easily go unnoticed without proper debugging.

Condition Removal (`c-Remove`) An 'if' or 'while' block might have multiple conditions, such as 'if left == len(nums) or nums[left] != target:'. Eliminating any one of these conditions might lead to the occurrence of loose conditions, potentially resulting in incorrect outcomes. We inject bugs by picking 'if' or 'while' blocks with more than one condition and then randomly removing one of conditions.

Branch Removal (`b-Remove`) In the programming, we often use different branches to cope with different situations, making the 'if-elif-...-else' structure a frequent choice. However, in cases where possible conditions are complex, it's easy to overlook some conditions. Recognizing this issue, we propose to inject potential bugs by randomly removing an 'elif' or 'else' branch from the original code containing the 'if-elif-...-else' structure.

Operator Change (`o-Change`) We extend the operator swapping operation in (Dinh et al., 2023) with more operators, e.g., membership operators (in and $not\ in$), bitwise operators ($\&, |, \hat{}\ , <<, >>$), logical operators (and, or, not) and identity operators (is and $is\ not$). We introduce artificial bugs by changing the operator to another operator with the same type.

A.2 Examples of Each Bug Injection Method

Here, for each method, we show an example of the clean partial and its converted buggy partial code.

[2]We do not differentiate the relative positions or frequencies of variables, deferring their exploration to future research.

Reference Partial Code

```python
def has_close_elements(numbers: List[float],
                       threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each other
    than given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
```

v-Rename

```python
def has_close_elements(numbers: List[float],
                       threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each other
    than given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(number):
            if idx != idx2:
                distance = abs(elem - elem2)
```

Reference Partial Code

```python
"""Problem Statement:Karen is getting ready
for a new school day! It is currently hh:mm,
given in a 24-hour format.  As you know,
Karen loves palindromes, and she believes
that it is good luck to wake up when the
time is a palindrome.

What is the minimum number of minutes
she should sleep, such that, when she wakes
up, the time is a palindrome?
......
"""
h, m = map(int, input().split(':'))
for i in range(999):
    if h == 10 * (m % 10) + m // 10:
        print(i)
        break
    if m < 59:
        h, m = (h, m + 1)
```

k-Remove

```python
"""Problem Statement:Karen is getting ready
for a new school day! It is currently hh:mm,
given in a 24-hour format.  As you know,
Karen loves palindromes, and she believes
that it is good luck to wake up when the
time is a palindrome.

What is the minimum number of minutes
she should sleep, such that, when she wakes
up, the time is a palindrome?
......
"""
h, m = map(int, input().split(':'))
for i in range(999):
    if h == 10 * (m % 10) + m // 10:
        print(i)
        
    if m < 59:
        h, m = (h, m + 1)
```

Reference Partial Code

```python
"""Problem Statement:Karen is getting ready
for a new school day! It is currently hh:mm,
given in a 24-hour format.  As you know,
Karen loves palindromes, and she believes
that it is good luck to wake up when the
time is a palindrome.

What is the minimum number of minutes
she should sleep, such that, when she wakes
up, the time is a palindrome?
......
"""
h, m = map(int, input().split(':'))
for i in range(999):
    if h == 10 * (m % 10) + m // 10:
        print(i)
        break
    if m < 59:
        h, m = (h, m + 1)
```

nv-Change

```python
"""Problem Statement:Karen is getting ready
for a new school day! It is currently hh:mm,
given in a 24-hour format.  As you know,
Karen loves palindromes, and she believes
that it is good luck to wake up when the
time is a palindrome.

What is the minimum number of minutes
she should sleep, such that, when she wakes
up, the time is a palindrome?
......
"""
h, m = map(int, input().split(':'))
for i in range(999):
    if h == 10 * (m % 10) + m // 10:
        print(i)
        break
    if m < 60:
        h, m = (h, m + 1)
```

Reference Partial Code

```python
def prime_fib(n: int):
    """
    prime_fib returns n-th number that is a
    Fibonacci number and it's also prime.
    """
    import math
    def is_prime(p):
        if p < 2:
            return False
        for k in range(2, min(int(math.sqrt(p))
                              + 1, p - 1)):
            if p % k == 0:
                return False
        return True
    f = [0, 1]
    while True:
        f.append(f[-1] + f[-2])
        if is_prime(f[-1]):
            n -= 1
```

wi-Swap

```python
def prime_fib(n: int):
    """
    prime_fib returns n-th number that is a
    Fibonacci number and it's also prime.
    """
    import math
    def is_prime(p):
        if p < 2:
            return False
        for k in range(2, min(int(math.sqrt(p))
                              + 1, p - 1)):
            if p % k == 0:
                return False
        return True
    f = [0, 1]
    if True:
        f.append(f[-1] + f[-2])
        if is_prime(f[-1]):
            n -= 1
```

Reference Partial Code

```
def find_Min(arr,low,high):
    """
    Write a python function to find the
    minimum element in a sorted and rotated array.
    >>> find_Min([1,2,3,4,5],0,4)
    1
    >>> find_Min([4,6,8],0,2)
    4
    >>> find_Min([2,3,5,7,9],0,4)
    2
    """
    while (low < high):
        mid = low + (high - low) // 2;
        if (arr[mid] == arr[high]):
            high -= 1
        elif (arr[mid] > arr[high]):
            low = mid + 1
        else:
            high = mid
```

b-Remove

```
def find_Min(arr,low,high):
    """
    Write a python function to find the
    minimum element in a sorted and rotated array.
    >>> find_Min([1,2,3,4,5],0,4)
    1
    >>> find_Min([4,6,8],0,2)
    4
    >>> find_Min([2,3,5,7,9],0,4)
    2
    """
    while (low < high):
        mid = low + (high - low) // 2;
        if (arr[mid] == arr[high]):
            high -= 1

        else:
            high = mid
```

Reference Partial Code

```
def sum(a,b):
    """
    Write a python function to find the sum of
    common divisors of two given numbers.
    >>> sum(10,15)
    6
    >>> sum(100,150)
    93
    """
    sum = 0
    for i in range (1,min(a,b)):
        if (a % i == 0 and b % i == 0):
            sum += i
```

c-Remove

```
def sum(a,b):
    """
    Write a python function to find the sum of
    common divisors of two given numbers.
    >>> sum(10,15)
    6
    >>> sum(100,150)
    93
    """
    sum = 0
    for i in range (1,min(a,b)):
        if (a % i == 0             ):
            sum += i
```

Reference Partial Code

```
def has_close_elements(numbers: List[float],
              threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each other
    than given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
```

o-Change

```
def has_close_elements(numbers: List[float],
              threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each other
    than given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                 distance = abs(elem + elem2)
                 if distance < threshold:
```

A.3 Example of Two Input Formats

There are mainly two types of formats: *Function-Call based format* and *Standard Input Format*:

Table 4 shows an example in *Function-Call based format*: The input contains a function header and a docstring, and the model is expected to generate a solution to be provided as the function's return value. The test cases used for checking accuracy of the generate codes are provided as *Assertion Statements*.

Table 5 shows an example in *Standard Input Format*: The input contains a problem description without a starter code (e.g., a header function). The model is expected to generate a complete solution. When provided an input in the location of input-required function (e.g., input()), the model is expected to output is answers to the STDOUT stream, such as by using print statements. The test cases used for checking the accuracy of the generated codes are provided as In/Out files, namely, given an input file, the model is expected to output the content shown in the Out file.

Prompt: (A function header + a docstring):

```
def has_close_elements(numbers: List[float],
    threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each
    other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0],
                                     0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0,
                   4.0,5.0, 2.0], 0.3)
    True
    """
```

Canonical Solution:

```
    for idx, elem in enumerate(numbers):
    for idx2, elem2 in enumerate(numbers):
        if idx != idx2:
            distance = abs(elem - elem2)
            if distance < threshold:
                return True
    return False
```

Test Format:

```
def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0,
                     5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0,
                     5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9,
                     4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9,
                     4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0,
                     5.0, 2.0], 0.1) == True
```

Notes: We can call the function by assign the *candidate* with the corresponding function name (e.g., has_close_elements in this example). Then, we can use the *exec* to execute the full code. If no errors are present, it indicates that the generated code has successfully passed the tests; otherwise, it has not.

Table 4: A Function-Call based Example from HumanEval.

Prompt: (A problem description):

```
"""Problem Statement: We have two distinct
integers  A and B.  Print the integer K
such that |A - K| = |B - K|. If such an
integer does not exist,  print IMPOSSIBLE.
Constraints:
    All values in input are integers.
    0 \leq A,\ B \leq 10^9
    A and B are distinct.
Input: Input is given from Standard Input
in the following format: A B

Output: Print the integer K satisfying the
condition. If such an integer does not exist,
print IMPOSSIBLE instead. """
```

Canonical Solution:

```
a,b = map(int, input().split())
diff = b - a
if diff % 2 == 0:
  print(b - diff // 2)
else:
  print('IMPOSSIBLE')
```

Test Format:

```
Input:  2 16
Output: 9
-----------------------------
Input:  999999999 1000000000
Output: IMPOSSIBLE
-----------------------------
Input: 3833907 617067938
Output: IMPOSSIBLE
-----------------------------
Input: 232334010 962483782
Output: 597408896
-----------------------------
```

Notes: The test cases of the form "Input -Output " need to run the generated code as a script. For example, we can save the generated code in *a.py* and an Input in *input1.txt*, then we can run

```
python a.py < input1. txt  > out1.txt
```

If the *out1.txt* has the same content as the gold output, we say that the generated code pass the test case.

Table 5: A Standard Input Example from FixEval.