Code-SPA: Style Preference Alignment to Large Language Models for Effective and Robust Code Debugging

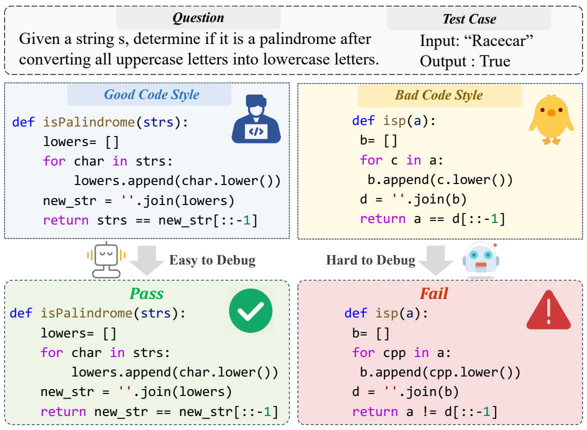
Tengfei Wen[1,2], **Xuanang Chen**[2,*], **Ben He**[1,2], **Le Sun**[2]
[1]School of Computer Science and Technology, University of Chinese Academy of Sciences
[2]Chinese Information Processing Laboratory, Institute of Software, Chinese Academy of Sciences
wentengfei23@mails.ucas.ac.cn, chenxuanang@iscas.ac.cn
benhe@ucas.ac.cn, sunle@iscas.ac.cn

Abstract

Large language models (LLMs) have demonstrated impressive capabilities in coding tasks like code generation and debugging. However, code from real-world users is often poorly styled, containing various types of noise, such as structural inconsistencies, stylistic deviations and flawed test cases. To investigate this, we first simulate poorly styled code using eight types of code perturbations, and then demonstrate that the debugging performance of existing LLM-based methods significantly declines on such inputs. Furthermore, to address this, we propose a novel debugging method called Code-SPA, which aligns noisy code with the well-structured style familiar to LLMs, mitigating the impact of stylistic inconsistencies. Specifically, Code-SPA extracts the model's preferred coding style from a reference snippet, then adjusts the input code by Concrete Syntax Tree (CST)-based transformations and LLM-assisted refinements before debugging. By aligning the code style preference, Code-SPA enhances the debugging performance of both code-specific and general-purpose LLMs on both poorly and well-styled code across the HumanEval, MBPP and EvalPlus datasets.

1 Introduction

In recent years, large language models (LLMs) have achieved great success in programming tasks, such as Deepseek-Coder (Guo et al., 2024), CodeLlama (RoziÃĺre et al., 2024), and GPT (Brown et al., 2020). These models have significantly accelerated the productivity of software developers by automating complex coding tasks, such as code generation (Austin et al., 2021; Chen et al., 2021b; Lai et al., 2022), debugging (Tian et al., 2024), and translation (Yan et al., 2023). Despite their impressive capabilities, LLMs still face significant robustness challenges (Shirafuji et al., 2023; Lad et al., 2024; Ma et al., 2024; Wang et al., 2024c; Singh et al., 2024). Meanwhile, recent studies on code generation have shown that LLMs are sensitive to variations in prompts (Zhang et al., 2024; Chen et al., 2024; Yang et al., 2024). As code generation tasks typically rely on task descriptions as the primary input, most of perturbations in recent studies focus on natural language.



```
def isPalindrome(strs):
    lowers= []
    for char in strs:
        lowers.append(char.lower())
    new_str = ''.join(lowers)
    return strs == new_str[::-1]
```

```
def isp(a):
 b= []
 for c in a:
  b.append(c.lower())
 d = ''.join(b)
 return a == d[::-1]
```

```
def isPalindrome(strs):
    lowers= []
    for char in strs:
        lowers.append(char.lower())
    new_str = ''.join(lowers)
    return new_str == new_str[::-1]
```

```
def isp(a):
 b= []
 for cpp in a:
  b.append(cpp.lower())
 d = ''.join(b)
 return a != d[::-1]
```

Figure 1: The impact of code style on LLM code understanding and debugging. Good code style (left) makes it easier for LLMs to understand the code logic, thus enabling more accurate and efficient error location and fixing, while bad code style (right) hinders LLM's comprehension, increasing debugging difficulty.

However, as debugging tasks involve more complex inputs (Olausson et al., 2024; Gu et al., 2024), typically consisting of the task description, buggy code, and failing test cases, this added complexity makes debugging tasks inherently different from code generation and other natural language tasks. Furthermore, current debugging methods (Chen et al., 2023; Hu et al., 2024; Zhong et al., 2024; Wang et al., 2024a) evaluate performance based on the model's initial output, which tends to be well-structured. This assumption of ideal code does not reflect the reality of programming. In practice, code is often written by different developers with varying coding styles, leading to inconsisten-

[*]Corresponding author.

cies in naming conventions, indentation patterns, and overall code structure. The variation in coding style can lead to significant difficulties in code comprehension. Such challenges can directly impact the performance of debugging tasks, with models struggling to handle inconsistencies in code style, as shown in Figure 1.

In this work, we analyze the impact of various code perturbations on debugging performance, our findings show that introducing such perturbations consistently reduces debugging performance, highlighting the sensitivity of models to code noise. Based on these insights, we propose the **Code** **S**tyle **P**reference **A**lignment (Code-SPA), which is designed to improve debugging performance by aligning the style of the input code with the preferred style of an LLM. The process involves extracting the style features from a reference code snippet, aligning the formatting through concrete syntax tree (CST) transformations, and refining deeper style features such as naming conventions and docstrings using LLM-assisted alignment before inputting the aligned code for debugging. By addressing the challenges posed by different coding styles, Code-SPA enhances the model's ability to better understand and debug code, even when facing noisy or inconsistent inputs.

Our contributions are three-fold: 1) We investigate the impact of code style on the debugging performance of LLMs, an area that has not been largely explored. 2) We propose a novel method called Code-SPA[1] to improve the robustness of LLM-based debugging through a style preference alignment mechanism. 3) Extensive experiments demonstrate that Code-SPA improves debugging performance not only on poorly styled code but also on well-styled code, highlighting its general effectiveness and adaptability.

2 Preliminaries

2.1 Code Debugging

We formulate the input of the code debugging task as a triplet (Q, C_{error}, T), where Q represents the question, C_{error} denotes the buggy code that fails to solve the question, and T refers to the test case that fails. A typical code debugging process by a large language model (LLM) can be expressed as:

$$C_{\text{correct}} = \text{LLM}(Q, C_{\text{error}}, T) \tag{1}$$

[1]Our code and data are publicly available at https://github.com/IsshikiIr0ha/codespa.

In evaluation datasets, the code C_{error} is typically clean and well-organized, making it easier for the model to understand, as it is trained on high-quality code. However, in real-world applications, the code that requires debugging often contains various types of noise, such as inconsistent variable naming, incorrect indentation, missing or excessive comments, and issues with test cases. These elements can complicate the debugging process, requiring models to differentiate between actual errors and extraneous noise for effective debugging.

2.2 Noise Simulation

Unlike previous works that focus on perturbations in natural language (Yang et al., 2024; Zhang et al., 2024), we introduce several types of perturbations to the buggy code C_{error} and test case T to simulate real-world noise.

1) **NoSems / PartSems: variable naming perturbations.** NoSems removes semantic meaning from variable names by mapping them to arbitrary lowercase letters. PartSems abbreviates variable names, retaining partial semantic structure.

2) **IndDisp / NonStdInd: indentation perturbations.** IndDisp alters indentation to modify the hierarchical structure of the code. NonStdInd replaces standard indentation lengths with non-standard.

3) **CmtRem: code comment perturbation.** CmtRem removes all comments, including single-line, multi-line comments, and docstrings.

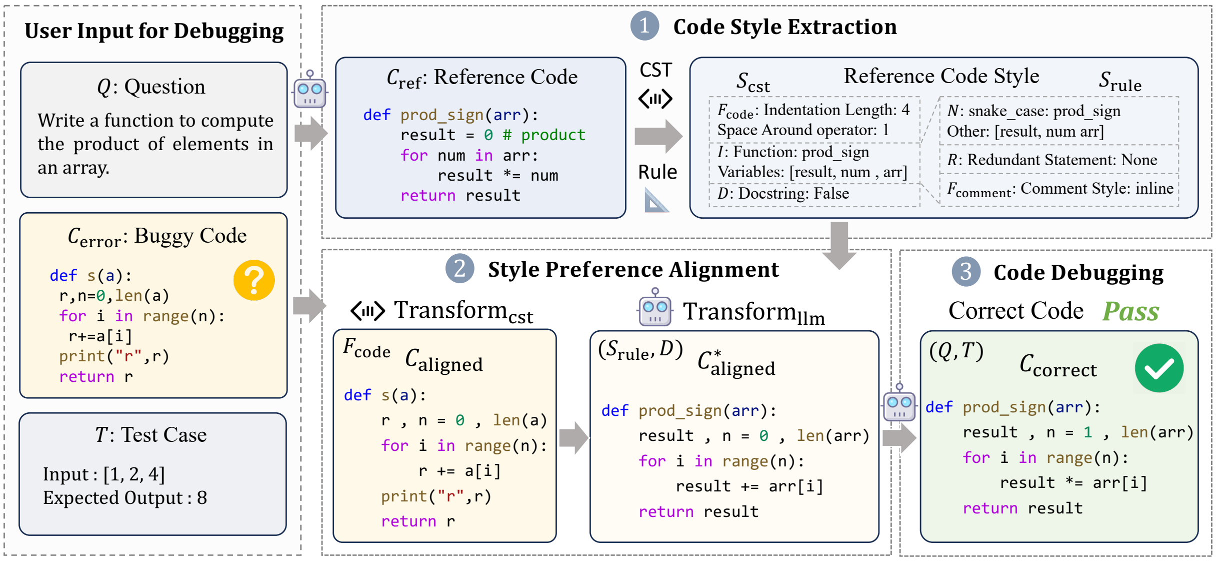
4) **RedStmts: redundancy perturbation.** RedStmts inserts redundant statements that do not affect the functionality of the code.

5) **NoGT / RandGT: test case perturbations.** NoGT removes the ground truth for the test case output, RandGT randomizes the ground truth.

More perturbation details of each simulated code noise, and examples of each type of code noise are summarized in Appendix A.

2.3 Code Style

Code style constitutes a set of conventions governing source code to enhance readability and consistency. Widely-recognized style guides include language-specific style guides like Python's PEP 8 (Guido van Rossum, 2001) and Go's gofmt, as well as organizational standards such as Java Code Conventions and Google C++ Style Guide. Notably, large language models (LLMs) also tend to develop distinctive coding styles (Wang et al.,


User Input for Debugging
Q: Question
Write a function to compute the product of elements in an array.
C_{error}: Buggy Code
T: Test Case
Input : [1, 2, 4]
Expected Output : 8
1 Code Style Extraction
C_{ref}: Reference Code
CST
Rule
S_{cst} Reference Code Style S_{rule}
F_{code}: Indentation Length: 4
Space Around operator: 1
I: Function: prod_sign
Variables: [result, num , arr]
D: Docstring: False
N: snake_case: prod_sign
Other: [result, num arr]
R: Redundant Statement: None
F_{comment}: Comment Style: inline
2 Style Preference Alignment
Transform$_{\text{cst}}$
Transform$_{\text{llm}}$
F_{code} C_{aligned}
(S_{rule}, D) C^{*}_{aligned}
3 Code Debugging
Correct Code Pass
(Q, T) C_{correct}

```
def prod_sign(arr):
    result = 0 # product
    for num in arr:
        result *= num
    return result
```

```
def s(a):
 r,n=0,len(a)
 for i in range(n):
  r+=a[i]
 print("r",r)
 return r
```

```
def s(a):
    r , n = 0 , len(a)
    for i in range(n):
        r += a[i]
    print("r",r)
    return r
```

```
def prod_sign(arr):
    result , n = 0 , len(arr)
    for i in range(n):
        result += arr[i]
    return result
```

```
def prod_sign(arr):
    result , n = 1 , len(arr)
    for i in range(n):
        result *= arr[i]
    return result
```

Figure 2: The core workflow of Code-SPA consists of three key stages: 1) Code Style Extraction, where the LLM's preferred code style features are automatically extracted from a reference code snippet; 2) Style Preference Alignment, where the user-input buggy code is aligned with the extracted style to normalize it according to the LLM's preferences; and 3) Code Debugging, where the style-aligned code is then used for debugging.

2024b), these stylistic variations may pose challenges in downstream tasks, as inconsistencies in code style can hinder code comprehension and impact task performance.

3 Method

3.1 Overview of Code-SPA

Variations in code style, whether among human programmers, across different LLMs due to variations in training data and architectures, or between humans and LLMs, pose significant challenges for code-related tasks, including debugging. To address this, we propose Code-SPA, a three-step approach designed to align the code style with the LLM's preferences. As illustrated in Figure 2, we first generate a reference code snippet from the LLM based on the given debugging task and automatically extract its code style features. Then, we align the style of the code to be debugged with the extracted style, mitigating the interference caused by stylistic discrepancies. Finally, the aligned code is input into the LLM for debugging.

3.2 Code Style Extraction

To extract the coding style from an LLM, we turn to summarize the code features from its generated code, by adopting an approach that combines Concrete Syntax Tree (CST)[2] analysis with rule-based pattern matching. Specifically, given a question Q, we first generate a reference code snippet C_{ref}:

$$C_{\text{ref}} = \text{LLM}(Q) \tag{2}$$

This code snippet C_{ref} serves as the textbook for style extraction and subsequent alignment. Using a CST parser, we represent C_{ref} as a concrete syntax tree T_{ref} that preserves all formatting details:

$$T_{\text{ref}} = \text{Parser}_{\text{cst}}(C_{\text{ref}}) \tag{3}$$

wherein the CST parser $\text{Parser}_{\text{cst}}$ analyzes the code structure and generates T_{ref}, from this representation, we extract various style features:

$$S_{\text{cst}} = \text{Extract}(T_{\text{ref}}) = [F_{\text{code}}, I, D] \tag{4}$$

wherein F_{code} represents formatting details, I is the set of identifiers (variable and function names), and D indicates the presence of a docstring.

After that, to complement the CST analysis, we employ a rule-based component that applies regular expressions and predefined patterns to capture additional style characteristics. Specifically, we define a set of regular expressions RE that match patterns in the code, allowing us to identify features. The rule-based extraction process can be formally represented as:

$$S_{\text{rule}} = \text{RE}(C_{\text{ref}}, I) = [N, R, F_{\text{comment}}] \tag{5}$$

wherein I and C_{ref} are the inputs, I used to extract the naming conventions N, R and F_{comment} are obtained through rule-baed matching from C_{ref}.

[2]https://github.com/Instagram/LibCST

Reference Code	Code Style Symbol	Code Style Value
Question: Compute factorial	*- Style features from CST analysis:* S_{cst}	
Code:	F_{code}: code formatting	{"Indentation Length": 4, "Space Around Operator" : 1}
`def compute_fact(n):`	I: identifiers	{"Function": `compute_fact`, "Variables" : `[n, i, res]`}
`    res =  1 # factorial`	D: docstring presence	Docstring : False
`    for i in range(1, n+1):`	*- Style features from rule-based pattern matching:* S_{rule}	
`        res  *=  i`	N: naming conventions	{"snake_case": `compute_fact`, "Other": `[n, i, res]`}
`    return res`	R: redundant statements	{"Redundant Statements" : None}
	F_{comment}: comment style	{"Comment Style" : "inline"}

Table 1: An example of code style extracted in Code-SPA method.

Finally, the reference code style S_{ref} is the combination of the CST-derived style features S_{cst} and the rule-based style features S_{rule}. As shown in Table 1, we have illustrated these code style features by an actual code. For example, "Indentation Length" being 4 indicates the indentation length for each level is set to 4, and "Space Around Operator" being 1 means that one space are added around operators, like `res = 1`, "Redundant Statements" being None means that there are no functionally irrelevant statements, such as print statement.

3.3 Style Preference Alignment

After extracting the code style features from the LLM, we proceed to the style preference alignment phase, where the goal is to harmonize the style of the code to be debugged with the extracted style template S_{ref} from the LLM. This alignment process involves two steps: direct alignment and LLM-assisted alignment.

Direct Alignment. The first part of the alignment process focuses on deterministically adjustable format features F_{code}, such as indentation length and the number of whitespace around operator. We apply a CST transformation using the LibCST library to adjust these formatting details of the code C_{error} being debugged, aligning them with the desired code formatting F_{code}:

$$C_{\text{aligned}} = \text{Transform}_{\text{cst}}(C_{\text{error}}, F_{\text{code}}) \tag{6}$$

By performing direct modifications, the input code is adjusted to a standard formatting style, ensuring that structural inconsistencies are eliminated before further debugging steps.

LLM-assisted Alignment. While CST transformation is a static method that focuses on syntactic structure and cannot handle the semantic nuances involved in naming conventions N, and redundant statements R and comment style F_{comment} in S_{rule}, and docstring presence D in S_{cst}. To address these challenges, we employ an LLM to assist in aligning these deeper style features. Formally, the LLM-assisted transformation refines the aligned code C_{aligned} in Eq. 6 based on semantical styles from reference code:

$$C^*_{\text{aligned}} = \text{Transform}_{\text{llm}}(C_{\text{aligned}}, S_{\text{rule}} + D) \tag{7}$$

This process ensures that, beyond mere structural formatting, the code is semantically aligned with the desired style template, thereby facilitating more effective debugging by harmonizing both syntactic and semantic style elements.

3.4 Code Debugging

After aligning the input buggy code C_{error} with the LLM's preferred style and generating the style-aligned code C^*_{aligned}, we proceed to the debugging phase. The primary goal of this step is to enable the LLM to debug the code with minimal distractions from stylistic inconsistencies, allowing the model to focus on identifying and correcting logical errors. The core idea is that by aligning the code's style, we reduce the noise introduced by variations in naming conventions, indentation, and comment styles, which could otherwise hinder the model's ability to accurately detect and fix errors.

Specifically, we utilize the triplet formulation (Q, C_{error}, T) introduced in Section 2.1 to represent the debugging task. The debugging process is formalized by substituting the original buggy code C_{error} with its style-aligned counterpart C^*_{aligned} as input to the LLM, as shown in the following equation:

$$C_{\text{correct}} = \text{LLM}(Q, C^*_{\text{aligned}}, T) \tag{8}$$

wherein C_{correct} is the corrected code output by the LLM. By providing the LLM with the style-aligned code, we hypothesize that the model will be better

Settings	Code Noise	Vanilla Debug	Self-Debug (Simple.)	Self-Debug (UT + Expl.)	INTERVENOR	Code-SPA (Ours)
w/o *Noise*	ORIGINAL	64.6	63.2	63.2	61.9	**65.3**
w/ *Single Noise*	NOSEMS	61.9 / -4.2%	62.2 / -1.6%	61.6 / -2.5%	60.6 / -2.1%	**64.8** / -0.8%
	PARTSEMS	61.9 / -4.2%	63.0 / -0.3%	61.9 / -2.1%	62.4 / +0.8%	**64.8** / -0.8%
	INDDISP	63.8 / -1.2%	62.7 / -0.8%	61.4 / -2.9%	60.6 / -2.1%	**65.3** / -0.0%
	NONSTDIND	63.5 / -1.7%	63.8 / +0.9%	61.4 / -2.9%	62.4 / +0.8%	**64.8** / -0.8%
	CMTREM	63.2 / -2.2%	61.9 / -2.1%	62.2 / -1.6%	59.5 / -3.9%	**64.8** / -0.8%
	REDSTMTS	61.9 / -4.2%	59.8 / -5.4%	59.5 / -5.9%	58.7 / -5.2%	**64.8** / -0.8%
	NOGT	**64.8** / +0.3%	63.2 / 0.0%	61.9 / -2.1%	61.9 / 0.0%	64.6 / -1.1%
	RANDGT	63.5 / -1.7%	63.2 / 0.0%	62.2 / -1.6%	62.2 / +0.5%	**65.1** / -0.3%
w/ *Mixed Noise*	NOISE-1	62.7 / -1.9%	62.7 / -1.9%	62.7 / -1.9%	60.4 / -4.0%	**65.3** / +0.7%
	NOISE-2	62.2 / -2.4%	61.9 / -2.7%	60.6 / -4.0%	61.4 / -3.0%	**64.8** / +0.2%
	NOISE-3	61.6 / -3.0%	62.7 / -1.9%	62.7 / -1.9%	61.4 / -3.0%	**64.8** / +0.2%
	NOISE-4	62.2 / -2.4%	61.6 / -3.0%	62.7 / -1.9%	57.7 / -6.9%	**63.8** / -0.8%
	NOISE-5	61.1 / -3.5%	60.6 / -4.0%	61.1 / -3.5%	59.0 / -5.6%	**63.8** / -0.8%
	NOISE-6	60.3 / -4.3%	58.7 / -5.9%	60.3 / -4.3%	56.9 / -7.5%	**63.5** / -1.1%

Table 2: Debugging performance of the DeepSeek-Coder-6.7B-Instruct model on the MBPP+ dataset under single- and mixed-type noise. In this table, scores presented as a / b signify the Pass@1 (a) for a given noise condition and its percentage change (b) relative to that method's ORIGINAL (noise-free) score.

able to focus on logical errors rather than being distracted by stylistic discrepancies. This approach aims to enhance debugging performance by aligning the code with a consistent style, reducing noise, and enabling the LLM to concentrate on resolving the core errors in the code.

4 Experiments

4.1 Experimental Settings

Datasets We evaluate our approach on four widely recognized datasets: HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021) and EvalPlus (Liu et al., 2023), which collectively provide a comprehensive benchmark for code generation and code repair. HumanEval consists of hand-crafted coding problems designed to assess functional correctness and the ability of models to generate syntactically and semantically correct programs. MBPP includes a collection of diverse Python programming problems, focusing on tasks of varying complexity that are suitable for evaluating general-purpose code understanding and generation. EvalPlus is an extension of the MBPP and HumanEval datasets, incorporating additional test cases to enhance robustness and challenge the debugging models with a wider range of scenarios.

Initial Code to Debug Existing studies on code debugging typically initiate the process with the model's first generated output. Since the debugging input and the initial output are both generated by the same model, they inherently share a consistent style and task-relevant variable semantics. However, this assumption oversimplifies real-world scenarios, where code often originates from diverse sources, leading to variations in style and potential semantic misalignments. In contrast, our debugging process is initialized with pre-generated code samples provided by the EvalPlus[3] dataset, which were originally produced by the CodeLlama-7B-Instruct model. Notably, this model does not participate in the debugging process itself. For functions containing only pass statements, we regenerate meaningful implementations to ensure a valid starting point for debugging. This approach better reflects real-world scenarios, where debugging often starts with diverse and stylistically inconsistent code rather than outputs from the same model. Subsequent perturbation experiments are also conducted on this set of pre-generated code.

Metrics To evaluate the performance of our approach, we use the widely adopted Pass@1 metric (Chen et al., 2021b). Our experiments include evaluating the impact of single noise sources as well as combined noise scenarios to simulate real-world complexities.

Baselines To evaluate the impact of noise on debugging performance and the effectiveness of our approach, we compare it against a range of

[3]`https://github.com/evalplus/evalplus/releases/tag/v0.2.0`

Base Model	Method	MBPP		MBPP+		HumanEval		HumanEval+	
		Ori.	Noise-3	Ori.	Noise-3	Ori.	Noise-3	Ori.	Noise-3
Deepseek-Coder (6.7B-Instruct)	Vanilla-Debug	74.6	73.0	64.6	61.6	74.4	67.7	68.9	63.4
	Self-Debug (Simple)	73.8	**73.8**	63.2	62.7	73.8	67.7	67.7	61.6
	Self-Debug (UT+Expl.)	74.6	73.0	63.2	62.7	75.6	67.7	71.3	64.0
	INTERVENOR	73.0	72.2	61.9	61.4	72.6	65.9	67.1	61.0
	Code-SPA (Ours)	**76.5**	**73.8**	**65.3**	**64.8**	**78.0**	**75.6**	**75.0**	**72.6**
CodeLlama (13B-Instruct)	Vanilla-Debug	64.3	62.7	55.3	54.2	45.7	43.9	42.1	41.5
	Self-Debug (Simple)	63.8	64.3	55.3	52.9	45.7	45.7	45.7	43.9
	Self-Debug (UT+Expl.)	66.9	66.4	55.8	55.3	48.8	45.7	46.3	44.5
	INTERVENOR	65.3	65.3	56.3	51.9	48.2	45.7	45.1	43.3
	Code-SPA (Ours)	**67.5**	**67.5**	**57.4**	**57.4**	**53.0**	**50.6**	**47.6**	**46.3**
Llama3 (8B-Instruct)	Vanilla-Debug	72.5	70.1	**65.3**	58.7	65.9	62.8	**63.4**	57.9
	Self-Debug (Simple)	71.4	68.8	63.0	59.0	65.2	62.2	61.6	57.9
	Self-Debug (UT+Expl.)	73.5	71.4	64.6	61.6	62.8	61.0	62.2	56.7
	INTERVENOR	**74.9**	73.5	64.6	62.4	68.3	65.2	61.6	59.1
	Code-SPA (Ours)	**74.9**	**74.3**	**65.3**	**64.8**	**69.5**	**68.3**	62.8	**60.4**
Qwen2.5 (7B-Instruct)	Vanilla-Debug	75.9	72.8	67.7	65.3	77.4	73.2	72.6	67.1
	Self-Debug (Simple)	75.1	74.3	66.4	64.6	76.8	74.4	68.9	65.9
	Self-Debug (UT+Expl.)	78.0	74.1	66.9	64.8	77.4	74.4	74.4	67.7
	INTERVENOR	75.4	74.9	63.5	63.2	70.7	71.3	67.7	64.6
	Code-SPA (Ours)	**80.7**	**80.4**	**69.0**	**68.8**	**80.5**	**79.9**	**75.6**	**75.0**

Table 3: Pass@1 scores achieved by Code-SPA and baseline methods on the original code (i.e. Ori.) and code injected with three types of noise (i.e. Noise-3) across the MBPP, MBPP+, HumanEval and HumanEval+ datasets.

established baseline methods. The **Vanilla Debug** baseline represents the simplest setup, where the debugging model directly takes the input tuple (Q, C_{error}, T). We also assess three debugging methods: **Self-Debug (Simple.)** (Chen et al., 2023) uses a straightforward prompt informing the model that the provided code contains errors. **Self-Debug (UT + Expl.)** (Chen et al., 2023) enhances the debugging process by including a failed test case in the input and prompting the model to perform a line-by-line explanation of the code before attempting to fix it. **INTERVENOR** (Wang et al., 2024a) utilizes a teacher-student framework where the teacher provides a modification method based on error messages, and the student applies the suggested modifications to correct the code.

4.2 Results

To evaluate the impacts of code noise on debugging performance, and demonstrate the effectiveness of our proposed debugging method Code-SPA, we carry out a series of experiments, as summarized in Tables 2 & 3. From these results, we can obtain the following conclusions:

The impact of single noise types varies, with variable name perturbations having the most significant effect. Our evaluation of individual noise types on debugging performance (Table 2) reveals three key patterns: 1) Perturbations affecting test case outputs, such as NoGT and RandGT, have minimal impact on Pass@1. 2) Noises that alter code semantics, like Total Semantic Removal (NoSems) and Redundant Statements (RedStmts), lead to the largest performance declines. 3) Noise types affecting code structure or non-semantic elements, such as Partial Semantic Preservation (PartSems), Comment Removal (CmtRem), and Indentation Variations (IndDisp, NonStdInd), result in smaller performance variations. Compared to baseline methods, Code-SPA consistently outperforms all noise types, showcasing its resilience to both semantic and structural noise.

Increasing the number of mixed noise types leads to greater degradation in debugging performance. We evaluate the impact of multiple simultaneous perturbations on LLM debugging performance, as shown in Table 2. We randomly combine single perturbations, observing a general trend of decreasing performance as the number of perturbations increases. However, this decrease is not strictly monotonic, suggesting potential interactions and non-linear effects between different perturbation types. Furthermore, under multiple perturbations, Code-SPA maintains higher Pass@1 scores across all noise levels, demonstrating its robustness to combined perturbations.

Method	MBPP	MBPP+
Code-SPA	**80.4**	**68.8**
Resampling	79.4	67.5
w/o reference code	74.1	63.2
w/o code style class	77.2	66.7
w/o CST	78.0	66.4
w/ internal alignment	72.8	63.2
w/o alignment	73.0	63.2

Table 4: Ablation study results of Code-SPA method on both MBPP and MBPP+ datasets using Qwen-2.5-7B-Instruct model.

Model	Method	MBPP+	HumanEval+
Deepseek-Coder (6.7B-Instruct)	NOISE-3	61.6	63.4
	w/ PEP8	61.6	64.0
	w/ Code-SPA	**64.8**	**72.6**
CodeLlama (13B-Instruct)	NOISE-3	54.2	41.5
	w/ PEP8	56.3	45.7
	w/ Code-SPA	**57.4**	**46.3**
Llama 3 (8B-Instruct)	NOISE-3	58.7	57.9
	w/ PEP8	60.1	56.7
	w/ Code-SPA	**64.8**	**60.4**
Qwen2.5 (7B-Instruct)	NOISE-3	65.3	67.1
	w/ PEP8	65.3	69.5
	w/ Code-SPA	**68.8**	**75.0**

Table 5: Comparisons of debugging results (Pass@1) on the NOISE-3 code, and code formatted by PEP8 guide or our Code-SPA alignment, across different LLMs on MBPP+ and HumanEval+ datasets.

Code-SPA outperforms other baselines in debugging noisy code across different datasets and base models. We present a comprehensive evaluation of Code-SPA's performance across various perturbation scenarios and base models, demonstrating its robustness and effectiveness in enhancing LLM debugging capabilities. Moreover, evaluations under the three mixed settings across four different models on the MBPP, MBPP+, HumanEval, and HumanEval+ datasets as shown in Table 3 demonstrate Code-SPA consistently outperforms or matches the best baseline method across all models and datasets, clearly showcasing Code-SPA's broad applicability.

Code-SPA also enhances performance on original noise-free code. As seen in Table 3, Code-SPA delivers consistent performance gains even on noise-free source code. This improvement arises from the mismatch in coding style between the code generation and debugging models. Since the debugging model differs from CodeLlama-7B which generates the initial error code to debug, their inherent stylistic differences persist even without explicit noise. Code-SPA effectively bridges this gap, leading to performance gains.

4.3 Analysis

Ablation Study To better understand the effectiveness of the Code-SPA framework, we conduct a series of ablation experiments on the MBPP and MBPP+ datasets. The results of these experiments are presented in Table 4. We define the following ablation settings: a) **Resampling**: The reference code is directly treated as the final solution; b) **w/o reference code**: The model generates the code style based on its own understanding of the "code style" class, without leveraging the reference code for guidance; c) **w/o code style class**: The reference code is provided to the model, but the model aligns its generation to the reference without incorporating the "code style" class for structured style alignment. d) **w/ internal alignment**: The model aligns the code style without relying on reference code or any predefined "code style" class; e) **w/o CST**: The model is given extracted code style information but without CST-based structured processing; f) **w/o alignment**: The reference code and extracted code style are provided as guidance, but the buggy code is not aligned to these sources, leaving its original structure unchanged. Based on above ablation study, we can derive the following conclusions:

Reference code is crucial for debugging performance. The results clearly show that using the reference code achieves the highest performance. Moreover, even ablation settings that retain the reference code, such as w/o code style and w/o CST, maintain relatively strong performance, reinforcing its crucial role in guiding the debugging process. The reference code provides essential guidance, this aligns with prior work (Gu et al., 2024) on code repair, which found that resampling often outperforms targeted methods by bypassing the complexities of code understanding.

Lack of explicit style alignment reduces debugging performance. Our experiments show that both self-alignment and no alignment significantly hinder performance. In the self-alignment setting, the model relies solely on its internal mechanisms to standardize its generated style, often resulting in inconsistent style. Similarly, without alignment, the buggy code retains its original, unstandardized style, making it harder for the model to focus on the underlying logic. These findings highlight the

Method	HumanEval	HumanEval+
Deepseek-Coder-6.7B-Instruct		
Initial Code to Debug	76.2	70.1
Vanilla-Debug	77.4	72.0
Self-Debug(UT+Expl.)	78.7	73.8
Code-SPA	**79.3**	**74.4**
CodeLlama-13B-Instruct		
Vanilla-Debug	73.2	67.7
Self-Debug(UT+Expl.)	73.2	68.9
Code-SPA	**75.0**	**70.7**

Table 6: Pass@1 results for the same-model and cross-model debugging performance on the NOISE-3 code across the HumanEval and HumanEval+ datasets.

Model	Method	HumanEvalPack-Fix
Deepseek-Coder (6.7B-Instruct)	Self-Debug (UT+Expl.)	67.7
	Code-SPA	**72.6**
CodeLlama (13B-Instruct)	Self-Debug (UT+Expl.)	46.3
	Code-SPA	**50.6**
Llama 3 (8B-Instruct)	Self-Debug (UT+Expl.)	64.6
	Code-SPA	**67.7**
Qwen2.5 (7B-Instruct)	Self-Debug (UT+Expl.)	76.8
	Code-SPA	**77.4**

Table 7: Debugging performance (Pass@1) of Code-SPA compared to the Self-Debug (UT+Expl.) baseline on the HumanEvalPack-Fix dataset across various large language models.

importance of explicit style alignment in improving debugging accuracy.

Comparisons with PEP8 We further evaluate a standard code style normalization approach based on PEP8 under three mixed settings, and present the result in Table 5. We applied autopep8 toolkit[4] to format the code before presenting it to the LLMs. It's crucial to emphasize that PEP8 focuses solely on stylistic conventions, such as indentation and spacing, it improves code readability for humans by enforcing consistent style, it does not address the semantic ambiguities or logical flaws that can hinder LLM understanding. Therefore, Code-SPA focus on style alignment, which goes beyond superficial formatting, is crucial for enhancing LLM debugging capabilities.

Impact of Style Consistency We examine how style consistency affects debugging performance by comparing two scenarios: (1) Deepseek-Coder-6.7B-Instruct debugging its own code (same-model) and (2) CodeLlama-13B-Instruct debugging Deepseek-Coder-6.7B-Instruct's code (cross-model). As shown in Table 6, the baseline method performs well in the same-model setting, likely due to the inherent style consistency between the generated and debugged code. However, in the cross-model scenario, where style discrepancies arise, Code-SPA provides a more substantial improvement, demonstrating its effectiveness in handling stylistic misalignment.

[4]https://pypi.org/project/autopep8/

Model	Method	HumanEval-X-C++
Deepseek-Coder (6.7B-Instruct)	Self-Debug (UT+Expl.)	59.7
	Code-SPA	**61.6**
CodeLlama (13B-Instruct)	Self-Debug (UT+Expl.)	38.6
	Code-SPA	**42.7**
Llama 3 (8B-Instruct)	Self-Debug (UT+Expl.)	40.2
	Code-SPA	**40.4**

Table 8: Debugging performance (Pass@1) of Code-SPA compared to the Self-Debug (UT+Expl.) baseline on the HumanEval-X-C++ dataset.

Effectiveness on Real-World Human-Authored Bugs To evaluate Code-SPA's effectiveness on real-world debugging tasks, we analyzed its performance on human-authored errors from the HumanEvalPack-Fix dataset (Muennighoff et al., 2024). The comparative results presented in Table 7 demonstrate Code-SPA's consistent advantage over the Self-Debug (UT+Expl.) baseline across a diverse set of large language models. These findings across multiple models highlight Code-SPA's robustness against human-authored bugs. Human coding styles are diverse and can differ from an LLM's optimal processing patterns, potentially creating comprehension hurdles. Code-SPA's improvements indicate its style alignment effectively mitigates this stylistic mismatch, thereby enhancing the LLM's ability to parse, understand, and rectify subtle human errors.

Generalization to Other Programming Languages To investigate Code-SPA's generalizability beyond Python, we evaluated it on C++ using the HumanEval-X-C++ dataset (Zheng et al., 2024). For C++ code processing, we utilized `tree-sitter` to apply rule-based stylistic perturbations, analogous to our `libcst`-based Python approach. The results in Table 8 demonstrate Code-SPA's effectiveness and applicability in the C++ context. This suggests that challenges from stylistic inconsistencies, and the advantages of Code-SPA in mitigating them, are not Python-specific, and that our approach holds promise for LLM-based debugging across more programming languages.

5 Related Work

Code Large Language Models Early code large language models, such as Codex (Chen et al., 2021a) and CodeGen (Nijkamp et al., 2023), were among the first to demonstrate the potential of LLMs in code generation, some models also unify code representation and generation (Wang et al., 2021; Guo et al., 2022; Wang et al., 2023). Recent advancements in LLMs have given rise to specialized models for code generation and understanding. Models such as DeepSeek-Coder (Guo et al., 2024), Qwen-Coder (Hui et al., 2024) and GPT-4 (OpenAI et al., 2024c) have been explicitly designed to handle coding-related tasks. Besides, reasoning models like OpenAI-o1 (OpenAI et al., 2024b) and DeepSeek-R1 (DeepSeek-AI et al., 2025), have also demonstrated impressive coding performance due to their long reasoning ability.

Debugging with LLMs Large Language Models have shown significant potential in enhancing debugging tasks by leveraging interpreter outputs (Chen et al., 2023; Hu et al., 2024), integrating with external debugging tools (Zhong et al., 2024), and decomposing complex debugging problems into smaller, more manageable steps (Zhang et al., 2023; Madaan et al., 2023; Shinn et al., 2023; Zhou et al., 2023; Wang et al., 2024a). Beyond proposing debugging methods, recent studies have also investigated the effectiveness and limitations of LLMs in code repair (Tyen et al., 2024; Gu et al., 2024; Kamoi et al., 2024; Olausson et al., 2024).

Robustness of LLMs While Large Language Models have demonstrated impressive capabilities across various tasks, they still suffer from robustness issues (Lad et al., 2024; Ma et al., 2024; Wang et al., 2024c; Singh et al., 2024), including in coding-related tasks (Zhang et al., 2024; Yang et al., 2024; Chen et al., 2024). Beyond evaluation, some works have explored techniques to enhance robustness (Zhang et al., 2024; Zhao et al., 2024). Different from most existing robustness evaluations in coding-related tasks, which primarily emphasize natural language, we focus specifically on the challenges posed by code perturbations.

6 Conclusion

In this work, we investigated the impact of stylistic variations on LLM debugging performance, highlighting the challenges LLMs face when dealing with diverse coding styles. Through a comprehensive set of perturbation experiments, including single and multiple perturbations across various base models and datasets, we demonstrated the significant influence of stylistic factors on debugging accuracy. To address these challenges, we proposed Code-SPA, a novel approach designed to enhance LLM robustness to stylistic variations. Our experiments demonstrate that Code-SPA effectively mitigates the negative impact of perturbations, achieving superior or comparable performance to baselines across all evaluated scenarios. This demonstrates the effectiveness of style alignment in enhancing LLM debugging capabilities.

Limitations

There are two primary limitations in our work due to the lack of real-world code data and the constraints of computational resources. Firstly, our experiments were conducted using synthetically perturbed code with poor styling, which, while useful for isolating specific issues, may not fully capture the complexity and diversity of real-world coding scenarios, where irregularities can stem from a broader range of sources and patterns. Secondly, although we utilized existing models in our experiments, we did not explore the potential of larger or more advanced models, such as GPT-4o (OpenAI et al., 2024a) or DeepSeek-R1 (DeepSeek-AI et al., 2025), which could offer improved performance and robustness, especially in handling more complex and diverse code-related tasks.

Acknowledgments

We sincerely thank the reviewers for their insightful comments and valuable suggestions. This work is supported by the National Natural Science Foundation of China (62272439) and the Fundamental Research Funds for the Central Universities.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, et al. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. 2024. Nlperturbator: Studying the robustness of code llms to natural language variations. *Preprint*, arXiv:2406.19783.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021a. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021b. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael SchÃd'rli, and Denny Zhou. 2023. Teaching large language models to self-debug. *Preprint*, arXiv:2304.05128.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X. Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *Preprint*, arXiv:2402.19475.

Alyssa Coghlan Guido van Rossum, Barry Warsaw. 2001. Pep 8 - style guide for python code.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, et al. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024. Leveraging print debugging to improve code generation in large language models. *Preprint*, arXiv:2401.05319.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, et al. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can LLMs actually correct their own mistakes? a critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12:1417–1440.

Vedang Lad, Wes Gurnee, and Max Tegmark. 2024. The remarkable robustness of llms: Stages of inference? *Preprint*, arXiv:2406.19384.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, et al. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

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

Xinbei Ma, Tianjie Ju, Jiyang Qiu, Zhuosheng Zhang, Hai Zhao, Lifeng Liu, and Yulong Wang. 2024. On the robustness of editing large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16197–16216, Miami, Florida, USA. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. *Preprint*, arXiv:2308.07124.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *Preprint*, arXiv:2203.13474.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, et al. 2024a. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, et al. 2024b. Openai o1 system card. *Preprint*, arXiv:2412.16720.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, et al. 2024c. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Baptiste RoziÃÍre, Jonas Gehring, Fabian Gloeckle, Sten Sootla, et al. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki. 2023. Exploring the robustness of large language models for solving programming problems. *Preprint*, arXiv:2306.14583.

Ayush Singh, Navpreet Singh, and Shubham Vatsal. 2024. Robustness of llms to perturbations in text. *Preprint*, arXiv:2407.08989.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, et al. 2024. DebugBench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand. Association for Computational Linguistics.

Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. 2024. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13894–13908, Bangkok, Thailand. Association for Computational Linguistics.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024a. INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2081–2107, Bangkok, Thailand. Association for Computational Linguistics.

Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, and Zibin Zheng. 2024b. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Preprint*, arXiv:2407.00456.

Yifei Wang, Dizhan Xue, Shengjie Zhang, and Shengsheng Qian. 2024c. BadAgent: Inserting and activating backdoor attacks in LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9811–9827, Bangkok, Thailand. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. 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, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *Preprint*, arXiv:2403.07506.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. Algo: Synthesizing algorithmic programs with generated oracle verifiers. In *Advances in Neural Information Processing Systems*, volume 36, pages 54769–54784. Curran Associates, Inc.

Yuansen Zhang, Xiao Wang, Zhiheng Xi, Han Xia, Tao Gui, Qi Zhang, and Xuanjing Huang. 2024. RoCoIns: Enhancing robustness of large language models through code-style instructions. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 14186–14203, Torino, Italia. ELRA and ICCL.

Yukun Zhao, Lingyong Yan, Weiwei Sun, Guoliang Xing, Shuaiqiang Wang, Chong Meng, Zhicong Cheng, Zhaochun Ren, and Dawei Yin. 2024. Improving the robustness of large language models via consistency alignment. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8931–8941, Torino, Italia. ELRA and ICCL.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *Preprint*, arXiv:2303.17568.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

A Noise Simulation Details

In this section, we provide a detailed explanation of the noise simulation process used in our experiments. To illustrate the implementation of each perturbation, we apply them to example code snippets from our dataset. These examples are drawn from real-world scenarios, and both the original and modified versions are shown to demonstrate how each type of noise affects the code structure and readability. This approach highlights the practical impact of noise in real-world programming environments.

NoSems No semantic information in variable names. We sequentially maps variables to lowercase letters, completely eliminating any semantic information encoded in the original names. This process ensures that all variables lose their original descriptive meaning while maintaining code functionality. This method effectively simulates the impact of poor coding standards where developers, neglecting clear naming conventions, might use overly simplistic or cryptic variable names. As a result, the readability and interpretability of the code are significantly reduced, making it harder to infer its intended behavior. In the example code below, `max_index` is truncated to `m`.

```
# Original Code
def can_arrange(arr):
    max_index = -1
    for i in range(1, len(arr)):
        if arr[i] < arr[i-1]:
            max_index = i
    return max_index - 1
----------------------------------------
# NoSems's Code
def can_arrange(a):
    b = -1
    for i in range(1, len(a)):
        if a[i] < a[i-1]:
            b = i
    return b - 1
```

PartSems Only partial semantic information in the variable names. We abbreviate variable names by extracting the first letter of each word separated by underscores or capitalization; otherwise, only the first letter is retained. For example, both `naturalLanguage` and `natural_language` become `nl`. This simulates scenarios where variable names are abbreviated or use common acronyms, retaining some structural and word-boundary information. In the following example code, `count_dict` becomes `cd`, and `result` becomes `r`.

```
# Original Code
def remove_duplicates(numbers):
    count_dict = {}
    result = []
    for num in numbers:
        if num not in count_dict:
            count_dict[num] = 1
            result.append(num)
        else:
            count_dict[num] += 1
    return result
----------------------------------------
# PartSems's Code
def remove_duplicates(n):
    cd = {}
    r = []
    for m in n:
        if m not in cd:
            cd[m] = 1
            r.append(m)
        else:
            cd[m] += 1
    return r
```

RedStmts Redundant statements. We randomly insert redundant statements, such as `print` statements and logging statements, into the code. The added redundancy does not affect the core functionality, meaning that for any test case T, the output of both the original and modified erroneous code remains identical.

```
# Original Code
def can_arrange(arr):
    max_index = -1
    for i in range(1, len(arr)):
        if arr[i] < arr[i-1]:
            max_index = i
    return max_index - 1
----------------------------------------
# RedStmts's Code
def can_arrange(arr):
    max_index = -1
    for i in range(1, len(arr)):
        print(f"Checking: {arr[i]}")
        if arr[i] < arr[i-1]:
            max_index = i
    print("Final max_index:",max_index)
    return max_index - 1
```

IndDisp Indentation displacement. We adjust the indentation of code to modify its hierarchical structure or alignment. This simulates lost or redundant indentation caused by copy-pasting or editing. This manipulation simulates common formatting errors that programmers might introduce, such as losing indentation when copy-pasting code blocks or accidentally adding extra, unnecessary indentation during editing. While in some languages this might only affect readability, in others where indentation is syntactically significant (like Python), such displacement can lead to errors or change the program's logic.

```
# Original Code
def same_chars(s0: str, s1: str):
    return sorted(s0) == sorted(s1)
----------------------------------------
# IndDisp's Code
def same_chars(s0: str, s1: str):
return sorted(s0) == sorted(s1)
```

NONSTDIND Non-standard indentation lengths. We replace the standard indentation like four-space or tab convention to other non-standard indentation length like two-space. These perturbations aim to evaluate model robustness against non-standard code formatting. In the following example, the indentation length of each level is changed from the standard 4 spaces to 1 space.

```
# Original Code
def iscube(a):
    x = round(a ** (1. / 3))
    return x ** 3 == a
----------------------------------------
# NonStdInd's Code
def iscube(a):
 x = round(a ** (1. / 3))
 return x ** 3 == a
```

CMTREM Comment removal. To eliminate the influence of comments on the model, we remove all comments from the code, including single-line comments, multi-line comments, and docstrings. This simulates a scenario where comments are omitted due to a lack of effort in writing them during code development. We removed all comments in the example, including docstring and line comments.

```
# Original Code
def add(lst):
    """
    Given a non-empty list
    of integers lst.
    Calculate the sum of
    even elements at odd
    indices.

    Examples:
    add([4, 2, 6, 7]) ==> 2
    """
    # calculate the sum.
    return sum(lst[i] for i in
        range(1, len(lst), 2)
----------------------------------------
# CmtRem's Code
def add(lst):
    return sum(lst[i] for i in
        range(1, len(lst), 2)
```

NOGT / RANDGT No or randomized ground truth for the test case. To evaluate the impact of test case T outputs on debugging performance, we either remove the ground truth output entirely or replace it with a random value. This simulates scenarios where the ground truth is unavailable or deliberately omitted, for instance, when the user neglects to provide the correct output. In the example, under the NOGT setting, we remove the ground truth output of the test case. Under the RANDGT setting, we replace the ground truth output `-9` with a random value, `100`.

```
# Original Code
def prod_signs(arr):
    if not arr:
        return None
    sum_magnitudes = 0
    product_signs = 1
    for num in arr:
        sum_magnitudes += abs(num)
        if num < 0:
            product_signs *= -1
    return (sum_magnitudes *
            product_signs)
----------------------------------------
# Original Assertion
The code fails on this test case:
assert prod_signs([1, 2, 2, -4]) == -9
----------------------------------------
# NoGT
The code fails on this test case:
    prod_signs([1, 2, 2, -4])
----------------------------------------
# RandGT
The code fails on this test case:
assert prod_signs([1, 2, 2, -4]) == -100
```

B Prompts

This subsection provides a detailed exposition of the prompt templates crafted for these key functionalities within Code-SPA. The structure and content of these prompts are crucial for providing the necessary context as illustrated in Figure 3. The `{task}` placeholder in Reference Code Generation and Debug Prompts specifies the code's objective. Within the Rewrite Prompt, specific placeholders are used to convey detailed stylistic preferences to the model, aligning with style dimensions (e.g., naming conventions, comment style) illustrated by the extracted features in Table 1. Within the Debug Prompt, `{aligned buggy code}` represents buggy code that has undergone a style alignment process but remains functionally incorrect. The `{test case}` placeholder provides crucial context by detailing the concrete failure scenario. This includes the specific inputs, the expected correct outputs.

Reference Code Generation:

Task:

`{task}`

Please give me the correct code.

Rewrite Prompt:
Please rewrite the given Python code to significantly improve its readability and comprehensibility. While adhering to the stylistic conventions below, focus on making the logic clear and explicit. The rewritten code should be easy for another developer to pick up and understand quickly. Functional equivalence with the original code must be maintained.

Code Style:
Naming Convention: `{variable map}`,
Comment Style: `{comment}`,
Docstring: `{docstring}`,
Reduntant Statement: `{statement}`

Code to be modified:
`{buggy code}`

Debug Prompt:

Task:

`{task}`

The following code provided fails on the task.

[InCorrect Code]

`{aligned buggy code}`

[/InCorrect Code]

And fail on these cases:

[Case]

`{test case}`

[/Case]

Please give me correct code.

Figure 3: Prompts used for Code-SPA.