Automated Feedback for Variable Name Semantics: A Large Language Model Approach to Enhancing Code Style

Koutaro Yumiba
University of Auckland
Auckland, New Zealand
kyum151@aucklanduni.ac.nz

Maxine Yang
University of Auckland
Auckland, New Zealand
myan565@aucklanduni.ac.nz

Owen Li
University of Auckland
Auckland, New Zealand
oli444@aucklanduni.ac.nz

Abstract

Providing students with effective guidance on descriptive variable names can be a challenging task, typically requiring manual feedback from instructors. The emergence of large language models (LLMs) offers a potential solution to this problem, enabling automated feedback on the semantics of the code, which would otherwise not have been possible with traditional linters and automated feedback tools. This paper investigates the extent to which LLMs can provide actionable feedback on variable naming. Our results demonstrate that LLM-generated feedback led to significant improvements in variable name quality, with consistently high ratings for both the actionability and justification of the feedback. These findings highlight the potential of LLMs as a scalable solution for providing automated semantic code style feedback in educational settings.

Keywords

Code Quality, Code Style, Code Feedback, Variable Naming, LLMs

ACM Reference Format:
Koutaro Yumiba, Maxine Yang, and Owen Li. 2025. Automated Feedback for Variable Name Semantics: A Large Language Model Approach to Enhancing Code Style. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Good code style ensures that software is readable, maintainable, and efficient [15]. Understanding good effective code style practices is essential for novice programmers, as it fosters habits that help them write clear and organized code [22, 23]. However, learning these code style practices often requires assistance, such as in the form of actionable feedback from instructors. Receiving feedback, whether from instructors or automated tools, enables students to better identify areas for improvement and refine their skills [12]. While student programmers have relied on instructor feedback to learn good code style practices, such approaches are often impractical in large classes due to the time-intensive nature of providing individualized, manual feedback [26].

In recent years, automated feedback tools have become increasingly prominent in helping developers and students improve their code style, especially given their ability to automate some of the workload faced by instructors in beginner programming courses [13, 26]. Most of these automated tools use static code analysis to examine the syntax of the submitted code and provide feedback accordingly [11, 22–24]. However, because static code analyzers tend to only examine syntax, they are unable to provide sufficient feedback on the semantic characteristics of a given program [18]. As a result, existing automated feedback tools tend to focus primarily on measures of correctness and identifying errors, rather than offering actionable feedback on context-dependent aspects of code quality such as clarity and intent [18, 26].

One particular aspect of code style that is difficult for automated tools to evaluate is variable naming. Meaningful variable names can greatly improve the readability and maintainability of code [8]. Programmers can more easily understand and modify code when they have descriptive variable names that convey their purpose. On the other hand, poor variable names may lead to confusion and errors, especially in collaborative environments. Teaching novices the importance of good variable naming is thus essential, but current tools cannot accurately evaluate whether the name sufficiently conveys the purpose of that variable. This highlights the need for more advanced tools that can provide automated feedback on the semantic aspects of code.

LLMs provide a unique capability in that, unlike rule-based automatic tools, they have the ability to account for semantics and context [6]. This allows them to assess whether the name of a variable accurately reflects its purpose in the program. This evaluation can be done automatically and we would be able to provide natural language explanations for the feedback which may be more accessible for novice programmers [4]. However, the research surrounding the usage of LLMs as feedback tools for code quality is limited and the accuracy and actionability of such feedback for variable names remains unclear. To improve our understanding of the capability of LLMs to provide automated feedback on variable names, we focus on the following research questions:

RQ1 *To what extent does LLM-generated feedback improve variable names?*
RQ2 *How actionable and well-justified is the feedback provided by LLMs on variable naming?*
RQ3 *How does the presence of broader context influence the effectiveness of LLM-generated feedback?*

2 Related Works

2.1 Background on Code Style

Kirk et al. defines code style as "aspects of software quality that can be determined by looking at the source code, constrained to understanding and changing code" [19]. According to this definition, code style is about the visual and structural aspects of code rather than the execution of the code. It emphasizes how code should be written in order to allow other developers to read, understand and easily maintain it in the future. Kirk et al.'s definition of code style is complemented by Wiese et al. who defined code style as code that is "elegant, efficient, idiomatic, and revealing of the design intent" [35].

Both Kirk et al. and Wiese et al.'s perspectives on code style highlight the importance of writing code that is both readable and maintainable. Beyond theoretical definitions, other studies have explored the practical implications of code style in real-world contexts. Bafatakis et al. [3] conducted a study where they analyzed over 407,097 Python code snippets from over 1.9 million Stack Overflow posts. They found a correlation between code style compliance and Stack Overflow post score, indicating that well-styled code may improve how clearly the code's purpose is communicated to others.

Given the practical importance of code style, it is crucial for programmers to start learning its principles early in their education. Izu et al. suggested that teaching and providing feedback on code quality should be taught to CS1-level students [14]. However, providing detailed and individualized feedback on student code can be an overwhelming task for instructors, particularly in large introductory programming courses. This manual feedback process can be time-consuming and unsustainable [26]. This is further backed by Ala-Mutka et al. [1], who found that manual feedback did not have a significant effect on students, mainly due to classes being too large and tutors not having enough time to check each student's submissions manually. One possible solution to this problem is to use automated feedback tools which can help provide timely feedback to CS1-level students at scale.

2.2 Current Existing Tools for Code Style

Over the past few years, a wide range of tools have been developed to automatically assess code quality and code style, particularly in the context of introductory programming courses. A systematic review by Keuning et al. [18] analyzed 101 automated feedback tools and found that the vast majority of the existing tools focus on identifying and correcting programming mistakes. In contrast, only a small fraction of the tools provide feedback that utilize knowledge about task constraints (14.9%) or conceptual understanding (16.8%). Among the mistake-focused feedback, solution errors were the most common category (59.4%), followed by test failures (52.5%), compiler errors (34.7%), and code style issues (29.7%). In terms of methodology, the review identified that automated testing remains the most commonly used technique (58.4%), followed by static code analysis (36.6%) while intention-based diagnosis is utilized in only 20.8% of the tools.

Table 1 summarizes a selection of tools that provide automated feedback on code style. Most of these tools, such as PythonTA [22] and TA-Bot [11], are built on top of existing static analysis tools, such as pylint, and serve primarily as wrappers that reformat the technical outputs of these linters into more accessible messages for students and novice programmers. While these tools were found to be effective in detecting structural stylistic issues, such as indentation or formatting errors, these tools generally fall short when it comes to evaluating the semantic quality of code, for example, whether the overall logic is clearly expressed or whether variable names effectively reflect their role in the program [11, 17, 22, 24, 28, 30].

Some progress has been made toward overcoming these limitations. For instance, Keuning et al. later introduced RefactorTutor [17], a tool capable of providing semantic feedback in a small number of well-defined contexts. It utilizes a custom abstract syntax tree that can be used to detect changes in the code supplied and give feedback based on what the student refactored. While RefactorTutor demonstrates potential for a more comprehensive feedback, its reliance on hardcoded templates and narrowly scoped examples limits its generalizability across varied student submissions.

Other efforts have experimented with custom configurations or machine learning based models to provide more nuanced feedback, though these approaches often require significant data curation or task-specific tuning [25, 26]. A recent and promising direction involves leveraging LLMs to bridge the gap between surface-level analysis and deeper code understanding. One such study by Woodrow et al. [36] applied LLMs to generate semantic feedback, including suggestions on variable naming and logic clarity. Their findings demonstrate the feasibility of LLMs in interpreting code intent, indicating a shift toward intent-aware tools that can provide semantic level feedback.

2.3 Variable Naming

Variable naming is an vital aspect of code style that is often overlooked by existing linters and traditional automated feedback tools. Variable names play an important role in code readability and comprehension, influencing how developers understand, maintain, and evaluate source code. Unlike syntactical aspects of code, variable naming requires understanding of the relationship between the variable and its purpose within the context of the code, which extends beyond the scope of most feedback tools.

Research demonstrates that there is a significant influence of variable naming on code comprehension. A controlled experiment carried out by Avidan et al. [2] involved nine developers who were tasked with understanding six methods extracted from production utility classes. The methods presented to participants contained either their original variable names or intentionally meaningless ones. Their findings revealed that code comprehension time was significantly reduced when participants read code containing the original variable names, as opposed to abstract identifiers such as a, b or c. Moreover, the study showed that misleading variable names - those that convey incorrect or deceptive information - hindered comprehension to an even greater extent than abstract names, emphasizing the critical role of variable names in code readability. Interestingly, the study also found that three of the six methods led to comprehension errors even when they used variable names that were not completely meaningless, such as single-letter identifiers. This suggests that poor naming, even if it is not reduced to arbitrary characters, can still significantly hinder code comprehension, even

Table 1: Existing Tools For Code Style Feedback

Author	Year	Tool	Analysis Used	Type of Feedback	Variable Naming
Ala-Mutka et al. [1]	2004	Style++	custom AST	Syntax/Structural	
Nutbrown and Higgins [28]	2016	Web-CAT	PMD	Syntax/Structural	
Keuning et al. [17]	2020	RefactorTutor	custom AST	Semantics	
Luukkainen et al. [24]	2022	ASPA	custom AST	Syntax/Structural	
Forden et al. [11]	2023	TA-Bot	pylint	Syntax/Structural	
Saliba et al. [30]	2024	ccheck-style	custom PST	Syntax/Structural	
Liu et al. [22]	2024	PythonTA	pylint, pycodestyle	Syntax/Structural	
Woodrow et al. [36]	2024	RTSF	ChatGPT	Semantics	✓

more so than minimalistic naming conventions such as single-letter variable names.

While Avidan et al.'s findings highlight the negative impact of meaningless or unclear variable names on code comprehension, other studies have argued that single-letter variable names may be appropriate in certain contexts. Beniamini et al. [5] analyzed variable names extracted from 1000 popular GitHub projects and found that single-letter variable names are still widely used. In order to test for their impact on code readability, they conducted an experiment where participants were shown different versions of the same functions with changed variable names. They found that single-letter variable names had no significant effect on the code comprehension of the participants, suggesting that in certain contexts, single-letter variable names can be used without hindering code readability.

Expanding beyond the specific impact of single-letter variable names, other research has taken a broader look at the influence of identifier naming on perceived code style. For example, Butler et al.'s [7] study analyzed eight open-source Java applications and found that identifiers with poor-quality names tend to be linked with code that is more complex, harder to read and more difficult to maintain.

Ultimately, effective variable naming is a highly subjective task that depends on context and semantics, presenting a challenge for traditional static analysis tools. This semantic gap reveals the need for more advanced tools capable of accounting for prior context and understanding variable relationships. As such, the task of providing variable naming feedback creates a compelling use case for Large Language Models (LLMs).

2.4 Usage of LLMs for Feedback and Variable Name Generation

The usage of LLMs and generative AI in educational contexts has become an increasingly relevant topic in recent years, with much discussion focused on how these models may influence how students learn to program [9]. The rapid advancement of these tools offers unprecedented opportunities for computing education, with LLM tools even being capable of solving introductory programming exam questions [10]. LLMs are particularly adept at handling contextual and semantic context within various tasks [6], and are capable of generating natural-language explanations on a variety of topics. As such, they provide a promising opportunity for streamlining the feedback process in CS1 courses for variable naming. However, their ability to provide 'good quality' or 'actionable' feedback on variable naming has not been properly evaluated and remains underexplored.

Several studies have investigated the usage of LLMs in generating code revisions [33] or error message explanations in developer and university course contexts to varying degrees of success, providing insights that can inform our approach to evaluating variable name feedback. One such study found that feedback from LLM-generated data did not significantly outperform stock compiler error messages [31]. However, they noted that handwritten error messages crafted by experts were substantially more effective, suggesting that the quality of the explanation is crucial for student comprehension. In contrast, Widjojo and Treude demonstrated that ChatGPT outperforms Stack Overflow in explaining compiler messages, with LLM-enhanced explanations being consistently more relevant [34]. These findings demonstrate that there is a need for systematic evaluation frameworks to assess the quality of LLM-generated explanations, especially when extending beyond the context of error messages to stylistic conventions.

To develop a framework for feedback evaluation, inspiration can be drawn from these error message studies, such as from Leinonen et al.'s methods for analysis of LLM-enhanced programming error messages [21]. Leinonen et al. examined multiple dimensions for LLM messages versus the original stock compiler messages, such as whether the provided explanation is comprehensible, necessary, explained and explained correctly, and improved over the original. Although focused on error messages, this rubric provides an adaptable foundation for evaluating feedback on code style and variable naming.

To our knowledge, only a few studies have actively investigated the use of LLMs in providing feedback on programming [20, 29], of which only one specifically accounts for both student contexts and code style-related feedback. Woodrow et al. [36] leveraged LLMs instead of traditional linters to successfully provide feedback on some semantic aspects of students' code, marking a shift towards a more meaningful automated analysis. However, this study did not delve into evaluating the quality or actionability of the feedback provided, but rather focused on the impact of the feedback on student outcomes. We have not found an educational study that actively focuses on evaluating the quality and actionability of feedback on variable naming or other code style conventions. This gap highlights the need for further research into the evaluation and analysis of LLM-generated feedback on code style, particularly for novice programmers.

3 Data Generation

In this study, we decided to use synthetic student data to evaluate the effectiveness of LLMs in providing feedback on variable names. Using synthetic data helps mitigate the potential biases present in real-world datasets and allows us to simulate real student code which may be scarce or hard to collect [16]. Previous research also supports the use of synthetic data in modeling student behaviour for educational research [27]. To generate our synthetic data, we opted to use OpenAI's GPT-4o-mini, a cost-effective but powerful large language model.

CS1-level programming courses are introductory and structured to teach the basic elements of programming. As such, these courses typically revolve around small, self-contained logic exercises with a focus on core programming constructs, such as variables, loops, conditionals, and basic functions, rather than complex design or object-oriented principles [14]. To ensure that our dataset closely reflects the kind of code typically written by students, we prompted the LLM to generate Python code snippets limited to 10-20 lines in length that use only basic programming constructs. Python was selected for this task due to its status as one of the most widely-used languages in introductory programming courses [32], and its suitability for demonstrating basic programming concepts in a clear manner. The generated code snippets were typically short, self-contained and focused on implementing simple algorithms, relying only on a minimal set of programming constructs. This ensured that the problem contexts and generated solutions remained within the scope of an introductory programming course.

We initially experimented with prompts based on specific programming tasks sourced from platforms such as Leetcode[1]. However, these prompts produced highly uniform outputs, limiting the variety of code snippets needed for our study. To address this, we moved away from task-specific prompts and instead allowed the LLMs to freely generate functions to any problem, which significantly improved the diversity of the generated dataset.

The final version of the prompt used in this study is structured as follows (our exact prompt can be found in Figure 1):

- A directive prompt is used to guide the generation of functions. To ensure variability in the output, a seed value is incorporated into the prompt. The prompt instructs the LLM to generate a function that addresses a random, commonly encountered beginner programming task. Additionally, it specifies the various styles of variable naming to be used.
- The output format is explicitly defined to structure the LLM's response in a consistent and parsable manner, thereby facilitating the extraction of functions for analysis.
- The notes section outlines specific considerations that the LLM should take into account during the function generation process.

Student Code Generation Prompt

Generate 4 Python code snippets that demonstrate varied degrees of identifier naming quality based on this seed {SEED}. Each snippet should be based on a random common introductory programming task.

- Misleading: Use variable names that are deceptive or totally unrelated to their function.
- Bad: Use vague or nondescriptive variable names.
- Good: Use decent, generally descriptive variable names.
- Perfect: Use highly descriptive and meaningful variable names.

Label each function with its respective identifier naming quality.

# Output Format

- Only return the code with no comments except for the label indicating the naming quality.
- Label each function with one of the following: misleading, bad, good, or perfect.
- Each code snippet must be between 10-20 lines long.
- All snippets should have exactly the same implementation but differ in identifier naming quality.
- Return a JSON output that looks like:
  {{
  Problem: {{problem here}}
  Code: {{code snippet here}}
  Identifier Naming Quality: {{label here}}
  }}

# Notes

- Enforce identical implementation for all snippets to highlight identifier naming differences.
- Each snippet must strictly adhere to the 10-20 line range.
- Focus on the quality of the identifier names with increasing quality from misleading to perfect as specified.
- It is mandatory for each snippet to be long enough, specifically within the 10-20 lines range, even when implementing misleading or bad identifier names.

Figure 1: The exact prompt used in this study.

The fully constructed prompt was then fed into the GPT-4o-mini model using a Python script[2]. This script automated the generation process, enabling us to produce a large number of function samples efficiently. To make the feedback generation on these samples easier, we also prompted the LLM to output responses in a structured JSON format. This approach allowed us to maintain consistent metadata across examples and made it straightforward to align each

[1]https://leetcode.com

[2]https://github.com/koutaroyumiba/var-namer

student code snippet with its respective context and corresponding feedback.

For this study, a total of 100 function samples were generated for analysis, evenly divided across four quality categories: misleading, poor, good, and perfect, with 25 samples in each. This number was selected based on a power analysis to ensure sufficient statistical power for comparing performance across the different categories.

Assuming a large effect size (Cohen's f = 0.4) and a standard significance level of $\alpha = 0.05$, a power analysis for one-way ANOVA (four groups) suggests that a total sample size of approximately 72 is required to achieve 80% power. By choosing 100 samples, we ensured a conservative buffer that strengthens the robustness of our results and permits exploratory subgroup analysis if needed. Additionally, generating 25 examples per category allows for representative coverage of code variations in each category while maintaining a feasible scope for manual validation and qualitative analysis.

We list an example from each of the four categories below (taken from seed 7):

Example 1: Misleading Variable Names

```python
def calculate_banana_count(running_days):
    total_apples = 0
    for day in range(running_days):
        if day % 2 == 0:
            total_apples += 1
    return total_apples
```

Example 2: Bad Variable Names

```python
def func(x):
    y = 0
    for i in range(x):
        if i % 2 == 0:
            y += 1
    return y
```

Example 3: Good Variable Names

```python
def count_even_days(total_days):
    even_day_count = 0
    for day in range(total_days):
        if day % 2 == 0:
            even_day_count += 1
    return even_day_count
```

Example 4: Perfect Variable Names

```python
def count_even_days_in_interval(days):
    even_day_counter = 0
    for current_day in range(days):
        if current_day % 2 == 0:
            even_day_counter += 1
    return even_day_counter
```

These examples are deliberately simple and match the structure of typical CS1 student code. For instance, in a generated example involving a simple counting task:

- The misleading version sets the count function name as `calculate_banana_count()` despite the actual count being named `total_apples`. The name does not accurately reflect the purpose of the function within context, and thus misleads the reader, hindering readability.
- The bad version names the function `func()` and uses numerous single-letter names, reflecting that beginner error of prioritizing brevity over clarity.
- The good and perfect versions improve on clarity with increasing specificity of the roles that each function and variable plays.
- The exercise itself is algorithmically simple and matches the context of a CS1 course, being a single function making use of lists and conditionals.

Since there is no guarantee that the LLM-generated student code accurately reflects real student submissions, we manually reviewed each generated code snippet to ensure its appropriateness for an introductory programming course. In particular, we checked for the presence of advanced constructs (e.g., lambda functions) that are unlikely to appear at this level, as well as snippets that were unreasonably short or too long in length.

This approach enables us to simulate code snippets that have misleading or perfect variable names which can often be rare in student submissions. By using this synthetic dataset, we are able to explore our research question in a controlled yet realistic context, ensuring that the evaluation of LLM-generated feedback remains relevant and applicable to actual classroom settings.

4 Method

4.1 Feedback Generation

We utilized LLMs to provide feedback on student code and suggest better variable names. Through prompt engineering, we developed two versions of the prompt: one that included the contextual information about the programming problem and another that excluded it. The purpose of using these two prompts was to evaluate whether the inclusion of the problem context would result in higher-quality feedback from the LLM.

The two prompts followed a similar structure (The exact prompt can be found in Figure 2.):

- A directive prompt is used to guide the evaluation of student code with a strict focus on variable naming quality. The prompt instructs the LLM to provide actionable feedback on variable names based on their clarity, descriptiveness, and consistency, while explicitly ignoring loop variables such as "i", "j", etc.
- The output format is explicitly defined to return a structured JSON object, enabling consistent parsing and downstream analysis.
- In the 'no context' group, the Problem section was removed from the prompt (all other text remained the same).

We used these two prompts on each of the 100 synthetic student code snippets to generate a total of 200 feedback samples using a Python script[3] (100 samples that incorporated the context of

[3]https://github.com/koutaroyumiba/var-namer/blob/main/variableFeedbackGenerator.py

the problem and 100 samples generated without any contextual information). An example of the feedback generated is presented below.

Feedback Generation Prompt

Please provide feedback on a student's code with a focus ONLY on variable names, listing specific suggestions for improvement.
Examine the given code and focus on evaluating variable names. Do not critique variables used in loops such as "i". Consider clarity, descriptiveness, and consistency of the variable names in your feedback. Provide feedback based on the initial problem the code is trying to solve.

# Output Format

- Provide the feedback in a bullet point list format.
- Use concise sentences, highlighting areas of success and potential improvement in variable naming.
- Output the format in a JSON format:
  {{
  feedback: list of feedback here
  mapping: (key: value of old variable name: new variable name
  }}

Problem:
<PROBLEM>

Student code:
<CODE HERE>

Figure 2: The exact feedback prompt (with context).

Example 5: Feedback Generated By the LLM

```
{
    "feedback": [
        "The function name 'a' is non-
            descriptive; consider renaming
            it to something that reflects
            its purpose, such as '
            calculate_area'.",
        "The parameter names 'b' and 'c' are
             vague; they should be more
            descriptive, e.g., 'length' and
            'width'.",
        "The variable name 'd' does not
            convey information about its
            content; consider naming it '
            area' to match what it
            represents."
    ],
    "mapping": {
        "a": "calculate_area",
        "b": "length",
        "c": "width",
        "d": "area"
    }
}
```

4.2 Rating Feedback

To evaluate the effectiveness and correctness of LLM-generated feedback on variable naming, a custom rubric was developed that assessed 3 dimensions. We opted for a 4-point ordinal scale for each dimension, where a score of 1 indicated poor quality and a score of 4 indicated excellent quality. Our full rubric is available on our Github[4]. This allowed evaluators to capture varying degrees of quality across the following dimensions:

- **Actionability**: The extent to which the feedback provided clear and specific instructions for how to improve the variable names. A higher score indicated the feedback clearly identified any problematic names and provided precise, understandable, and implementable alternative names.
- **Justification**: The quality of reasoning given for each suggested change. A higher score indicated a well-reasoned and articulated explanation, such as linking to naming conventions or aligning with the variables' roles in the code.
- **Correctness (Before / After)**: The appropriateness and clarity of the variable names, rated both prior to and after LLM-generated feedback was applied to the synthetic student code. A higher score indicated that variable names were consistently descriptive, matched conventions, and reflected their purpose in the code.

The design process for the rubric involved a collaborative effort by the research team (n = 3), all of whom have experience in introductory programming courses. Literature on rubric formation and programming explanations guided rubric structure. In particular, we drew on a study by Leinonen et al. [21] which evaluated the quality of LLM-generated programming error explanations and emphasized dimensions such as clarity, correctness, explanation, and presence of actionable fixes. This greatly informed our dimensions of actionability, justification, and correctness.

To ensure consistency and joint understanding of the dimensions within the rubric, a discussion was held between the research team regarding interpretation and formation. Each researcher was then independently assigned the 200 feedback entries in random order, and rated them according to the rubric. Researchers were blind to condition (context vs. no-context) to minimize bias. This randomization and experiment blinding was done using a Python script[5]. After randomizing and removing all unnecessary information, the script displays each code example individually and prompts the evaluator to rate the corresponding feedback according to the rubric. Following this process, we then aggregated and conducted statistical analyses on the rubric scores as discussed in Section 5.

[4]https://github.com/koutaroyumiba/var-namer/blob/main/rubric.png
[5]https://github.com/koutaroyumiba/var-namer/blob/main/feedbackCoding.py

5 Results

A snippet of our ratings can be found in Tables 3 and 4. Our full ratings can be found on our GitHub[6]. In general, our ratings do show that LLMs can be used to improve variable names with the majority of variable name correctness after the feedback having a score of 4 (172/200 (86%)). Having established that LLM-generated feedback can have a measurable impact on variable names, we next quantify how large that impact is.

5.1 RQ1: Improvement in Variable Name Quality

To evaluate the effectiveness of LLM-generated feedback in improving the correctness of variable names, we compared correctness ratings before and after feedback across two conditions: with problem context and without problem context. Given the paired nature of the samples and the lack of assumptions about their underlying distribution, we used the Wilcoxon signed-rank test to test for significance.

No context: The Wilcoxon signed-rank test found that the rated correctness of variable names in the no context condition after feedback (M = 3.82, SD = 0.48) was significantly higher than before feedback (M = 2.94, SD = 0.98), (V = 844, $p < .001$). A rank-biserial correlation test demonstrated a very large effect size, (r = 0.91, 95% CI [0.88, 0.93]), suggesting that feedback led to a substantial improvement in correctness in the no context condition. A bar plot of correctness ratings pre- and post-feedback in the no context condition is visualized in Figure 3.

Context: The Wilcoxon signed-rank test found that the rated correctness of variable names in the context condition after feedback (M = 3.87, SD = 0.37) was significantly higher than before feedback (M = 2.95, SD = 0.95), (V = 720, $p < .001$). A rank-biserial correlation test demonstrated a very large effect size, (r = 0.93, 95 CI [0.90, 0.94]), suggesting that feedback led to a substantial improvement in correctness in the context condition. A bar plot of correctness ratings pre- and post-feedback in the context condition is visualized in Figure 4.

These results indicate that LLM-generated feedback significantly improved the correctness of variable names in both conditions. The large effect sizes suggest that these improvements are substantial. Post-feedback correctness was slightly higher on average within the context condition compared to the no context condition, and this difference is investigated further in Section 5.3.

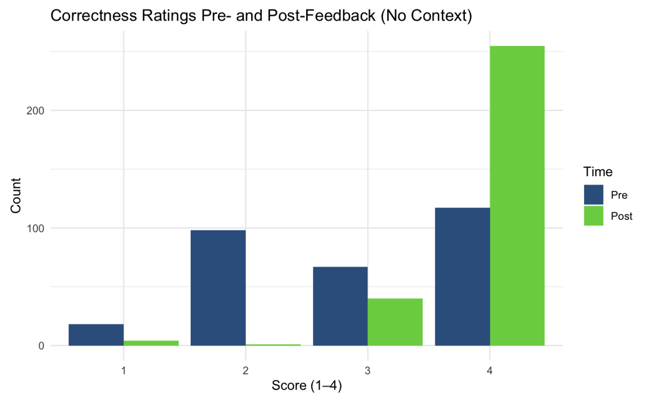

Figure 3: No Context Condition: Bar plot comparison of the count of correctness ratings before and after LLM feedback was applied.

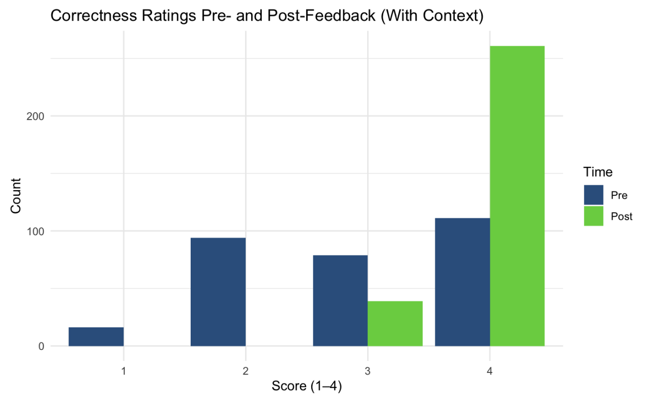

Figure 4: Context Condition: Bar plot comparison of the count of correctness ratings before and after LLM feedback was applied.

	Actionability (M, SD)	Justification (M, SD)
No Context	3.93, 0.29	3.70, 0.55
With Context	3.92, 0.29	3.66, 0.54

Table 2: Ratings of actionability and justification quality across conditions

5.2 RQ2: Actionability and Justification of Feedback

To assess the quality of the feedback itself, we examined two dimensions: justification and actionability. Mean ratings and standard deviations across both conditions are shown in Table 2. A box plot visualization for both metrics can be found in Figure 5.

Ratings were generally high across both dimensions, and across both context and no context conditions. These findings imply that LLM-generated feedback consistently was found to be actionable and well-justified. The lack of variation across conditions also suggests robustness in the quality of LLM feedback regardless of context. This is further explored in Section 5.3.

5.3 RQ3: Impact of Problem Context

To evaluate whether broader code context influenced the effectiveness of LLM feedback, we compared the amount of improvement in correctness between the two conditions. On average, variable name correctness improved by M = 0.92 (SD = 0.98) in the context condition and M = 0.88 (SD = 1.01) in the no context condition

[6]https://github.com/koutaroyumiba/var-namer/tree/main/data

Table 3: Snippet of Ratings (With Context Group).

Problem ID	C(PRE)-KOT	C(AFTER)-KOT	C(PRE)-OWEN	C(AFTER)-OWEN	C(PRE)-MAXINE	C(AFTER)-MAXINE
8	2	4	2	4	2	4
10	4	4	3	4	3	4
24	2	4	2	4	2	4
43	4	4	4	4	4	3
53	3	4	2	4	3	4
29	3	4	2	3	2	4
62	4	4	3	3	4	4

C(PRE) indicates correctness before, and C(AFTER) indicates correctness after.

Table 4: Snippet of Ratings (Without Context Group).

Problem ID	C(PRE)-KOT	C(AFTER)-KOT	C(PRE)-OWEN	C(AFTER)-OWEN	C(PRE)-MAXINE	C(AFTER)-MAXINE
8	2	4	2	3	2	4
10	4	4	4	4	4	4
24	2	4	2	4	2	4
43	4	4	4	4	3	3
53	2	4	1	4	2	4
29	3	4	3	4	3	4
62	4	4	3	4	4	3

C(PRE) indicates correctness before, and C(AFTER) indicates correctness after.

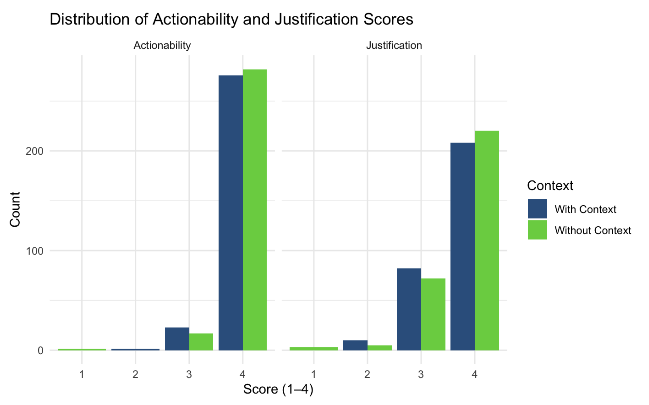

Figure 5: Bar plot comparison of actionability and justification scores across with context and no context conditions. Score appears to be high regardless of contextual information.

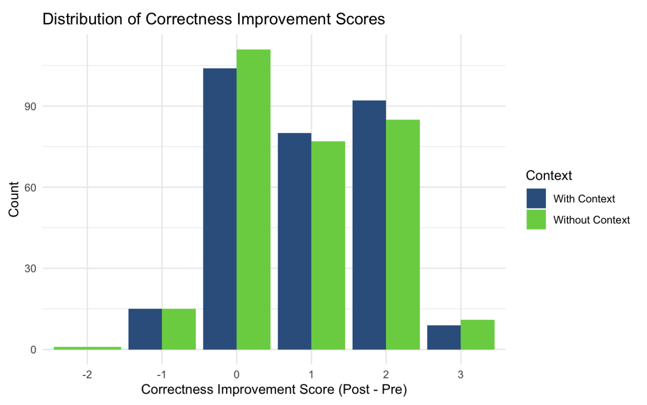

Figure 6: Bar plot demonstrating the distribution of score improvements across both context and no context conditions.

after LLM-generated feedback was applied. However, a Wilcoxon rank-sum test revealed that there was no significant difference between these improvements (W = 46132, p = .576), suggesting that providing context does not significantly increase the correctness of feedback. A box plot demonstrating the improvement in correctness can be found in Figure 6.

Furthermore, we investigated whether the presence of context can influence the quality of the feedback provided, measured by justification and actionability dimensions. Using the same test, we found that actionability of feedback did not see a significant difference between the two context conditions (W = 43183, p = .277), and neither did justification quality (W = 44104, p = .340). These findings suggest that providing broader context on code does not significantly influence the quality of LLM-generated feedback.

6 Discussion

6.1 Implications

Our results indicate that LLMs significantly enhance the correctness of variable naming. This improvement is consistent, even when contextual information varies. The ability of LLMs to perform well without full contextual information suggests that LLMs could be effectively integrated into autograders or code editors for use in CS1-level courses, where full context might not be always available.

Our results from Table 2 show that there is a high mean in both actionability and justification for LLM feedback. These findings suggest that LLMs can consistently provide high-quality feedback that students can understand and act upon, and that the changes to variable names are well-justified and reasonable. Consequently, such feedback holds strong potential for use in CS1-level educational settings, as correct, actionable, and well-explained feedback is likely to better support learners in developing more effective variable naming practices.

6.2 Limitations

Despite these findings, there were several limitations to our approach that should be acknowledged. First of all, the LLM occasionally provided unusual suggestions that lacked clear reasoning. For example, the LLM gave the following feedback for Problem 29 (Figure 7) and Problem 79 (Figure 8).

Problem 29 Feedback

- 1
- 2
- 3

Suggested Improvements
lst -> numbersList
total -> sumTotal
item -> currentNumber

Figure 7: The feedback generated for problem 29

Problem 79 Feedback

- The variable 'accumulator' clearly indicates its purpose, but 'sum' would be a more straightforward choice since it directly relates to the task of summing integers.
- The parameter 'n' is concise but could be more descriptive; consider using 'upper_limit' to clarify its role.
- The variable 'integer' within the loop could be renamed to 'current_integer' for better clarity, showing that it represents the current integer in the summation.

Suggested Improvements
accumulator -> sum
n -> upper_limit
integer -> current_integer

Figure 8: The feedback generated for problem 79

The feedback generated for Problem 29 lacked any justification for the proposed changes to the variable names, instead just providing a blank slate. In Problem 79, the LLM suggested replacing variable names such as 'total_sum' or 'accumulator' to 'sum'. While this may initially appear to be a reasonable simplification, it is important to note that 'sum' is a built-in function in Python. As such, using it as a variable name can lead to unexpected behavior and should generally be avoided. Such unusual feedback may mislead novice programmers into bad programming habits, and suggests that the information provided by LLMs should be carefully evaluated. It is worth noting that among the 200 feedback instances generated by the LLM, only one involved a major hallucination (as seen in Problem 29). The remaining hallucinations were minor (similar to that seen in Problem 79) and could likely be avoided by providing the LLM with additional context, such as specifying the programming language used in the code snippet.

Additionally, the student code utilized in this study was synthetically generated via LLMs, calling ecological validity into question. As the code was not created by actual students, the LLMs' performance in providing feedback on variable naming in student code may not be fully generalizable to CS1-level courses.

6.3 Future Research

While this study highlights the potential of using LLMs to improve variable naming in students' code, several avenues remain for future exploration. One important direction is the use of actual student code. In this study, LLM-generated code was used to evaluate feedback quality and variable naming performance. However, incorporating authentic student code in future research may offer a more accurate and ecologically valid assessment of the model's effectiveness in real-world educational contexts. Also, this study primarily focused on variable naming, leaving several other aspects of semantic code style unexplored. Future research could investigate additional dimensions of code style, such as commenting practices and use of appropriate constructs, to further evaluate the capabilities of LLMs in providing comprehensive feedback. In the future, conducting long-term studies to evaluate the impact of LLM-generated feedback on student learning outcomes, such as grades on code style assignments, would provide valuable insights into this approach to teach CS1-level students good code style practices.

7 Conclusion

In this paper, we investigated the potential of LLMs to provide effective feedback on variable naming in CS1-level student code. In particular, we explored the following research questions:

RQ1 *To what extent does LLM-generated feedback improve variable names?*
RQ2 *How actionable and well-justified is the feedback provided by LLMs on variable naming?*
RQ3 *How does the presence of broader context influence the effectiveness of LLM-generated feedback?*

In order to answer these questions, we developed a rubric for evaluating LLM-generated feedback, and collected ratings from evaluators across two different feedback conditions via a custom Python script. Our statistical analyses demonstrate that LLM-generated feedback can significantly improve the correctness of variable names in CS1-level code, with large effect sizes across both context and no context conditions. The improvement was consistent regardless of the condition, demonstrating that the LLM-provided feedback is accurate even when minimal context is available. Furthermore, the ratings for actionability and justification were also consistently high, indicating that feedback not only led to improvement, but was easy to follow and well-justified. These results support the potential of LLMs as a tool for automated feedback on semantic code style in introductory programming courses.

References

[1] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting Students in C++ Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education: Research* 3 (2004), 245–262. doi:10.28945/300

[2] Eran Avidan and Dror G. Feitelson. 2017. Effects of Variable Names on Comprehension: An Empirical Study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 55–65. doi:10.1109/ICPC.2017.27

[3] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martín Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python Coding Style Compliance on Stack Overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 210–214. doi:10.1109/MSR.2019.00042

[4] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. doi:10.1145/3344429.3372508

[5] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. 2017. Meaningful Identifier Names: The Case of Single-Letter Variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 45–54. doi:10.1109/ICPC.2017.18

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[7] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *2010 14th European Conference on Software Maintenance and Reengineering*. 156–165. doi:10.1109/CSMR.2010.27

[8] Roee Cates, Nadav Yunik, and Dror G. Feitelson. 2021. Does Code Structure Affect Comprehension? On Using and Naming Intermediate Variables. *CoRR* abs/2103.11008 (2021). arXiv:2103.11008 https://arxiv.org/abs/2103.11008

[9] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. doi:10.1145/3624720

[10] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proceedings of the 24th Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3511861.3511863

[11] Jack Forden, Alexander Gebhard, and Dennis Brylow. 2023. Experiences with TA-Bot in CS1. In *Proceedings of the ACM Conference on Global Computing Education Vol 1* (Hyderabad, India) *(CompEd 2023)*. Association for Computing Machinery, New York, NY, USA, 57–63. doi:10.1145/3576882.3617930

[12] Qian Fu, Yafeng Zheng, Mengyao Zhang, Lanqin Zheng, Junyi Zhou, and Bochao Xie. 2023. Effects of Different Feedback Strategies on Academic Achievements, Learning Motivations, and Self-Efficacy for Novice Programmers. *Educational technology research and development* 71, 3 (June 2023), 1013–1032. doi:10.1007/s11423-023-10223-2

[13] Gerhard Hagerer, Laura Lahesoo, Miriam Anschütz, Stephan Krusche, and Georg Groh. 2021. An Analysis of Programming Course Evaluations Before and After the Introduction of an Autograder. *CoRR* abs/2110.15134 (2021). arXiv:2110.15134 https://arxiv.org/abs/2110.15134

[14] Cruz Izu, Claudio Mirolo, Jürgen Börstler, Harold Connamacher, Ryan Crosby, Richard Glassey, Georgiana Haldeman, Olli Kiljunen, Amruth N. Kumar, David Liu, Andrew Luxton-Reilly, Stephanos Matsumoto, Eduardo Carneiro de Oliveira, SeÁn Russell, and Anshul Shah. 2025. Introducing Code Quality at CS1 Level: Examples and Activities. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 339–377. doi:10.1145/3689187.3709615

[15] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 513–523. doi:10.1109/ICSME.2019.00085

[16] James Jordon, Lukasz Szpruch, Florimond Houssiau, Mirko Bottarelli, Giovanni Cherubin, Carsten Maple, Samuel N. Cohen, and Adrian Weller. 2022. Synthetic Data – what, why and how? arXiv:2205.03257 [cs.LG] https://arxiv.org/abs/2205.03257

[17] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student Refactoring Behaviour in a Programming Tutor. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. ACM, Koli Finland, 1–10. doi:10.1145/3428029.3428043

[18] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (Sept. 2018), 43 pages. doi:10.1145/3231711

[19] Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2024. A Literature-Informed Model for Code Style Principles to Support Teachers of Text-Based Programming. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 134–143. doi:10.1145/3636243.3636258

[20] Charles Koutcheme, Nicola Dainese, Sami Sarsa, Arto Hellas, Juho Leinonen, and Paul Denny. 2024. Open Source Language Models Can Provide Feedback: Evaluating LLMs' Ability to Help Students Using GPT-4-As-A-Judge. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 52–58. doi:10.1145/3649217.3653612

[21] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 563–569. doi:10.1145/3545945.3569770

[22] David Liu, Jonathan Calver, and Michelle Craig. 2024. A Static Analysis Tool in CS1: Student Usage and Perceptions of PythonTA. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 172–181. doi:10.1145/3636243.3636262

[23] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 666–671. doi:10.1145/3287324.3287503

[24] Roope Luukkainen, Jussi Kasurinen, Uolevi Nikula, and Valentina Lenarduzzi. 2022. ASPA: A static analyser to support learning and continuous feedback on programming courses. An empirical validation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training* (Pittsburgh, Pennsylvania) *(ICSE-SEET '22)*. Association for Computing Machinery, New York, NY, USA, 29–39. doi:10.1145/3510456.3514149

[25] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. Style-Analyzer: Fixing Code Style Inconsistencies with Interpretable Unsupervised Algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 468–478. doi:10.1109/MSR.2019.00073

[26] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (Feb. 2024), 43 pages. doi:10.1145/3636515

[27] Yaneth Moreno, Anthony Montero, Francisco Hidrobo, and Saba Infante. 2023. *Synthetic Data Generator for an E-Learning Platform in a Big Data Environment*. 431–440. doi:10.1007/978-981-99-5414-8_39

[28] Stephen Nutbrown and Colin Higgins. 2016. Static Analysis of Programming Exercises: Fairness, Usefulness and a Method for Application. *Computer Science Education* 26, 2-3 (July 2016), 104–128. doi:10.1080/08993408.2016.1179865

[29] Tung Phung, Victor-Alexandru Pădurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 2* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3568812.3603476

[30] Liam Saliba, Elisa Shioji, Eduardo Oliveira, Shaanan Cohney, and Jianzhong Qi. 2024. Learning with Style: Improving Student Code-Style Through Better Automated Feedback. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1175–1181. doi:10.1145/3626252.3630889

[31] Eddie Antonio Santos and Brett A. Becker. 2024. Not the Silver Bullet: LLM-enhanced Programming Error Messages are Ineffective in Practice. In *Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research* (Manchester, United Kingdom) *(UKICER '24)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. doi:10.1145/3689535.3689554

[32] Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, and Jason P. Siegfried. 2021. Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning. In *2021 16th International Conference on Computer Science Education (ICCSE)*. 407–412. doi:10.1109/ICCSE51940.2021.9569444

[33] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* 1, FSE, Article 36 (July 2024), 23 pages. doi:10.1145/3643762

[34] Patricia Widjojo and Christoph Treude. 2023. Addressing Compiler Errors: Stack Overflow or Large Language Models? *CoRR* abs/2307.10793 (2023). doi:10.48550/ARXIV.2307.10793 arXiv:2307.10793

[35] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 41–50. doi:10.1145/3051457.3051469

[36] Juliette Woodrow, Ali Malik, and Chris Piech. 2024. AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1442–1448. doi:10.1145/3626252.3630773