2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

Beyond Functional Correctness: An Exploratory Study on the Time Efficiency of Programming Assignments

Yida Tao
Southern University of Science and Technology
Shenzhen, China
taoyd@sustech.edu.cn

Wenyan Chen
Southern University of Science and Technology
Shenzhen, China
12012437@mail.sustech.edu.cn

Qingyang Ye
Southern University of Science and Technology
Shenzhen, China
12012340@mail.sustech.edu.cn

Yao Zhao
Southern University of Science and Technology
Shenzhen, China
zhaoy6@sustech.edu.cn

ABSTRACT

Practical programming assignments are critical parts of programming courses in Computer Science education. Students are expected to translate programming concepts learned from lectures into executable implementations that solve the tasks outlined in the assignments. These implementations are primarily assessed based on their functional correctness, ensuring that students' code produces the expected output when provided with specific inputs.

However, functional correctness is not the only metric that evaluates the quality of programs. Runtime efficiency is a metric that is less frequently evaluated in programming courses, yet it holds significant importance in the context of professional software development. To investigate this gap and its potential ramifications, we conducted a large-scale empirical study on the time efficiency of 250 programming assignments that are evaluated solely on functional correctness. The results demonstrate that students' programming assignments exhibit significant variance in terms of execution time. We further identified 27 recurring inefficient code patterns from these assignments, and observed that most of the inefficient patterns can be optimized by automated tools such as PMD, IntelliJ IDEA and ChatGPT. Our findings provide actionable guidelines for educators to enhance the organization and integration of code performance topics throughout the programming course curriculum.

CCS CONCEPTS

• **Software and its engineering** → **Patterns**; • **Social and professional topics** → **Software engineering education**;

KEYWORDS

Programming Assignment, Code Performance, Tool Support

ICSE-SEET '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0498-7/24/04…$15.00
https://doi.org/10.1145/3639474.3640065

ACM Reference Format:
Yida Tao, Wenyan Chen, Qingyang Ye, and Yao Zhao. 2024. Beyond Functional Correctness: An Exploratory Study on the Time Efficiency of Programming Assignments. In *46th International Conference on Software Engineering: : Software Engineering Education and Training (ICSE-SEET '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639474.3640065

1 INTRODUCTION

Programming courses play a pivotal role in Computer Science (CS) education. Among the key factors of programming courses, practical programming assignments hold tremendous significance in fostering students' coding, debugging, and problem-solving skills. By working on practical programming assignments, students are able to apply theoretical knowledge to real-world scenarios, which empowers them with abilities and confidence to excel in the ever-evolving technology landscape.

Most programming assignments are assessed by *functional correctness* [23]. However, program performance is also a crucial metric to evaluate, especially for professional software development and business [14]. For instance, the runtime efficiency of high-traffic websites, such as e-commerce platforms and social media sites, directly affects user experience and therefore the businesses' overall competitiveness. In real-time applications such as healthcare and autonomous vehicle systems, even a millisecond delay could make a lethal impact. Hence, the importance of program performance should also be well recognized by CS students, and one way to promote this is for students to intentionally optimize the programming assignments for better runtime efficiency.

Time efficiency is a typical measurement for the runtime performance of programming assignments. However, this measurement is far less applied compared to functional correctness, which is the dominating measurement used in most systems of automated assignment assessment [23]. In such circumstances, where time consumption is not explicitly evaluated, would students still pay attention to the efficiency of their assignments? Or, if teachers have addressed performance concerns in lectures but performance is not a mandatory factor in assignment evaluation, would students make efforts to optimize their code?

To shed light on these questions, we studied the time efficiency of 250 programming assignments from an upper-level programming

course that are assessed solely on functional correctness. We conducted quantitative and qualitative analyses on these assignments to address the following research questions:

- **RQ1**: What is the time efficiency of programming assignments that are assessed solely on functional correctness?
- **RQ2**: What is the characteristics of programming assignments that have slower execution time?
- **RQ3**: Which methods can educators apply to help students develop time-efficient programs?

By profiling the 250 assignments in a consistent setting, we observed a significant variance in their execution times. In particular, 14.4% assignments are extremely inefficient for being orders of magnitude slower than their counterpart assignments. Next, we systematically analyzed slow assignments and their faster counterparts by leveraging static code analysis tools and manual inspections. As a result, we identified 27 inefficient code patterns at the statement level, which are related to *Strings*, *Variables*, *Collections*, and *File I/O* in Java programming. Educators can integrate these patterns into the course to improve students' knowledge of code performance.

The qualitative analysis of inefficient code further indicates potential deficiencies in our current pedagogical approach to this course. At present, we address code performance concerns mainly in the upper-level programming course; its introductory-level prerequisite course primarily emphasizes on fundamental programming concepts, with limited coverage of performance-related topics. Nevertheless, we observed that 58% of the inefficient code occurrences are related to topics covered in the introductory-level prerequisite course (e.g., strings and variables). This finding suggests that code performance concerns may not be addressed in isolation within a single course or a specific lecture, as a programming assignment, like real software applications, typically involves multiple concepts covered throughout various lectures in the programming course curriculum. Instead, performance concerns should be brought to students' attention simultaneously with the introduction of the corresponding programming concept. We also observed that 22% inefficient code are related to Java collections, whose internal implementations and performance implications are extensively discussed in the lectures of the upper-level programming course. This finding further underscores the importance of addressing code performance topics early in the programming course curriculum. Doing so not only raises students' awareness of performance issues but also fosters good coding practices, which can have a profound impact on their subsequent programming endeavors.

Lastly, we explored tools and approaches that may assist students in optimizing the time efficiency of their code. ChatGPT demonstrates promising results in fixing 96% inefficient code patterns. Nevertheless, ChatGPT may incur context switching overhead to students' workflow and produce inconsistent results due to its inherent unpredictability. IntelliJ IDEA, on the other hand, is the default Integrated Development Environment (IDE) for our course. It can be used together with the PMD plugin to fix 93% of the inefficient code patterns, while seamlessly integrating the code optimization process into students' regular programming workflow.

2 RELATED WORK

2.1 Automated Assessment and Feedback of Programming Assignments

Automating assignment assessment and feedback have been a long-standing subject of interest within the field of computer science education. In a systematic literature review compiled in 2022 [11], references published between 2005 and 2021 have been examined to investigate the techniques and tools employed for automating code assessment in educational contexts. Recent studies also incorporate advanced program analysis and other related techniques into automated code assessment. For example, *Gradeer* leverages *JUnit*, *PMD*, and *CheckStyle* to help tutors assess the semantic correctness and code quality of Java programs [9]. *GradeIT* applies program repair techniques on assignments that do not compile, thus awarding partial marks for these incomplete submissions [24]. Clegg investigated the application of mutation testing to enhance the evaluation of test suites used for assignment assessment [10].

In addition to automatic code graders, researchers have also explored techniques that automatically provide informative and non-disruptive feedback to students. For example, static analysis has been applied to provide continuous feedback to students engaged in learning Scratch [13] and Python programming [16, 20]. Presler-Marshall et al. proposed *SQLRepair*, which identifies and repairs mistakes in students' written SQL queries [25]. Renzella and Cain proposed a cross-browser audio feedback feature, which is integrated into an open-source learning management system to improve student engagement [27].

While most of this line of work focuses on program's functional correctness, runtime performance has also been leveraged in automatic code graders and feedback systems. Karavirta and Ihantola measured the loading and running time of JavaScript exercises [18]. Cheang et al. described an automated grading system that gives students feedback on the execution outcomes of their programming assignments, including an assessment of whether the assignments have exceeded the time limit [8]. Memory consumption of assignment submissions has also been measured and provided as feedback in the *Pythia* learning platform [12].

In general, functional correctness is the predominant metric used for the assessment of programming assignments, while performance assessment is mostly applied for programming competitions but less used in educational contexts [11]. However, given the significance of performance metrics in software quality, it is crucial for us to understand the runtime performance of students' programming assignments, even when the primary evaluation criterion is functional correctness. Our work bridges this gap by providing empirical data and analysis on this issue.

2.2 Program Runtime Performance

Runtime performance is a crucial metric to evaluate software quality. While many factors affect the runtime performance of software programs, altering code implementations is one of the common approaches for performance optimization, a subject that has also been explored in related research. Kawrykow and Robillard examined Java applications and identified cases where API invocations can be substituted with more efficient alternatives [19]. Oliveira et

al. recommended energy-efficient alternatives for Java collection implementations using energy profiles and static analysis [21]. Tao et al. mined Stack Overflow to suggest faster code alternatives for data manipulation scripts written in Python [29]. Similar observations have also been made for other programming languages such as JavaScript [28] and Ruby [33]. However, to the best of our knowledge, limited research has delved into the runtime differences among alternative implementations in programming assignments. Our study aims to bridge this gap.

2.3 ChatGPT and Higher Education

ChatGPT is a conversational AI, trained from a large language model with massive data, that generates coherent, high-quality responses to users' questions [22]. In addition to engaging in authentic, general-purpose dialogues, ChatGPT also demonstrates great potential in tackling challenging problems within specific domains, such as programming, debugging, and testing. Xia and Zhang proposed a ChatGPT-based repair tool that achieves state-of-the-art repair performance on the Defects4J dataset [32]. Tian et al. conducted an empirical study that demonstrates the effectiveness of ChatGPT in performing code generation, code summary and code repair tasks [30].

Research endeavors have been undertaken to explore the opportunities and challenges of integrating ChatGPT in higher education, specifically in fields such as computer science. Jalil et al. reported that ChatGPT was able to answer over 55% of the questions from a widely used software testing textbook [17]. Geng et al. found that ChatGPT can attain a grade of B- in an introductory-level functional language programming course, outperforming nearly half of the enrolled students [15]. Wang et al. evaluated ChatGPT's performance on solving 187 problems that are collected from six undergraduate CS courses, aiming to discuss whether and how ChatGPT can be used as an AI assistant in CS education [31]. Our work also explores ChatGPT's capability of assisting students in improving program efficiency.

3 BACKGROUND

In this section, we provide an overview of the course and assignments utilized in the study.

3.1 Course Context

CS209 is an upper-level programming course that introduces advanced knowledge of application design and development to engineering students at our university. The course uses Java as the primary programming language and covers topics such as file I/O, collections, multithreading, network programming, and web development. By taking CS209, which is offered in both the spring and fall semesters, students are expected to develop a deeper understanding of design principles and best practices, while acquiring advanced programming proficiency for developing practical applications and software systems.

The prerequisite course for CS209 is CS109, an introductory-level programming course that prepares students with the basic knowledge on computer programming and the Java programming language. The topic of time efficiency, or program runtime performance in general, has been discussed in CS209 but not in CS109. For instance, in CS209, teachers address the performance difference between alternative implementations of Java collections, and introduce automated profiling tools for students to measure the time efficiency of their coding practices.

3.2 Assignment Setup

CS209 comprises weekly lab practices and three multi-week assignments designed to evaluate students' progress. While lab practices are typically small-scale tasks for students to finish onsite during lab sessions, assignments require students to develop a practical software system that addresses real-life problems. Here is a brief description of the three assignments:

- **Assignment I** is a *data analytics* system, which analyzes domain data to gain insights. Students are expected to apply knowledge of Java I/O, Collections, Generics, Lambda expressions, and Stream to finish this assignment.
- **Assignment II** is a client/server application built with socket programming, multithreading, and JavaFX.
- **Assignment III** is a web application built using Spring Boot, which aims to consolidate students' understanding on web development and associated concepts such as MVC (the Model-View-Controller pattern), ORM (Object-Relation Mapping), and dependency injection.

In this study, we used only assignment I as our subject. Assignments II and III are designed to be open-ended programming assignments, intended to foster students' creativity. As a result, these two assignments also introduce a multitude of confounding factors for program efficiency, making them unsuitable candidates for our study. For instance, different designs, GUIs, libraries and frameworks can be used for assignments II and III, which almost certainly influence the runtime behavior of students' submissions but offer limited insight into the actual time efficiency of students' own implementations.

Assignment I, on the other hand, divides a data analytics problem into a set of subtasks, each with clear functional requirements. Students are required to implement a method for each subtask, with the method input and output clearly specified in the assignment requirements through method signatures and return types. In addition, students are instructed to use Java built-in features such as file I/O, collections, and streams in this assignment, while third-party libraries are explicitly forbidden. Finally, a test suite is designed to evaluate the functional correctness for each method. In other words, assignment I provides a controlled setup where students use the same set of APIs to implement the same functions, which are evaluated by the same test suites. Therefore, this assignment allows us to conduct meaningful performance comparisons between functionally equivalent programs.

3.3 Data Description

We used the assignment I of CS209 from the 2022 Spring, 2022 Fall, and 2023 Spring semesters as our dataset. For all of these assignments, students submit their implementations as a set of `.java` files to an online judging (OJ) platform used internally at our university. Upon submission, the OJ platform automatically evaluates the functional correctness of the submission by executing the test suites deployed by the teaching team. Students get instant

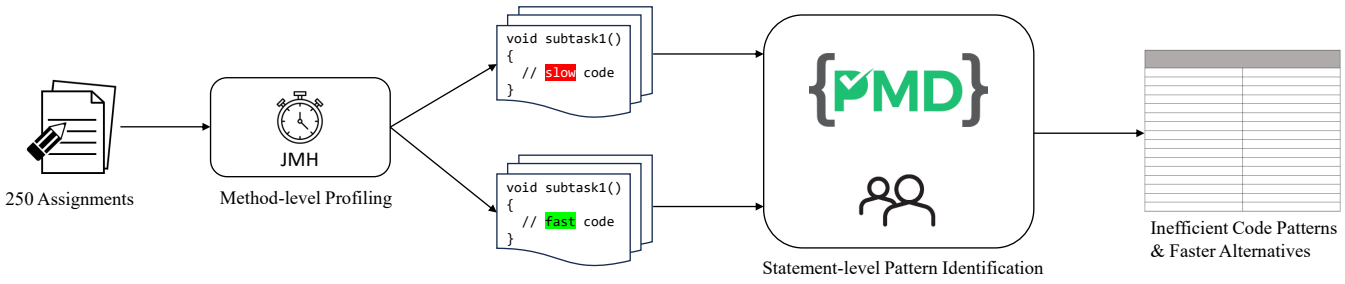


Figure 1: Overview of the methodology.

feedback on the test execution results and could make another submission based on the feedback. Finally, students should select only one of the attempts as the final submission, and the evaluation result of this selected attempt will be factored into the course grade.

It is worth mentioning that the OJ platform can also be configured to impose time and memory limits for the submitted assignments. At present, we have set relatively large values for these limits, and very few students have exceeded them. In other words, our current assessment primarily emphasizes the functional correctness of programming assignments.

We collected all of the assignment submissions from the OJ platform. We filtered out submissions with partial credit and retained only those with full credits for our study. This preprocessing ensures that all subject assignments are functionally equivalent in terms of the deployed test suites, and that any runtime difference between such assignments is most likely to be attributed to their different implementations. Table 1 summarizes the assignments and the respective number of submissions after the preprocessing step. A detailed description of each subtask is included in our artifact dataset [1]. All assignments are anonymized in our study.

4 METHODOLOGY

In this section, we outline our methodology for addressing the research questions proposed in Section 1. The overall workflow is shown in Figure 1.

4.1 Assignment Profiling

To address RQ1, we use Java profilers to automatically analyze the execution time of students' assignments. A Java profiler works by monitoring Java bytecode constructs and operations at the JVM level, and can be used to collect various runtime data during the execution of a Java program. While there are many Java profiler tools, we used Java Microbenchmark Harness (JMH) for our analysis, as JMH is a popular open-source Java profiler that is actively maintained and continuously supported [3]. In addition, JMH supports flexible configuration and automation, which is crucial for our analysis.

The assignment profiling consists of three steps. First, we developed a preprocessing script that leverages `javax.tools.JavaCompiler` to automatically compile all `.java` source code in the assignment dataset to `.class` bytecode. This step is necessary because JMH operates by analyzing bytecode, while the OJ platform only accepts submissions of `.java` source code.

Next, we implemented a program that leverages JMH to automatically profile students' assignments given the bytecode. As introduced in Section 3.2, a submission consists of several methods, each solves a single subtask in the assignment and serves as an atomic unit that is being tested for functional correctness. Therefore, we also use the method-level granularity for profiling. Specifically, each method in a submission is executed `N` times with the same test input we used for evaluating functional correctness. To obtain accurate estimation of time efficiency, we set `N=3` and used the average execution time as the profiling results.

Assignment profiling was conducted on a Intel Core i5 CPU (2GHz) machine with 16GB of memory running 64-bit macOS Big Sur 11.5.2 and JDK 17. The results were persisted in JSON files, which were further analyzed to study the time efficiency of students' assignments. We report the analysis results in Section 5.1.

4.2 Pattern Identification

To address RQ2, we aim to identify common syntax patterns in students' assignments that slow down program execution. Specifically, we extract inefficient code patterns at the *statement* level, which is the smallest unit for execution and optimization.

It is worth mentioning that students' choices on data structures and algorithms also affect the runtime performance of assignments. However, we cannot assume that students enrolled in CS209 have all taken courses on data structures and algorithms. In addition, algorithms are intricately tied to specific tasks, while our objective is to identify inefficient code patterns that are generally applicable. Hence, we focus on statement-level syntax patterns in this work.

We used *PMD* (version 7.0.0) for automatic pattern identification. PMD is an open-source static code analyzer for multiple programming languages including Java. It works by parsing source files into abstract syntax trees (AST) and running rules against the ASTs to find violations [5]. PMD provides a set of predefined rules that cover various code quality aspects such as coding styles, design issues, documentation, best practices, and performance optimization. Users may also define customized rules that are not included in the predefined rules. For the purpose of this work, we focus on the *performance violation* warnings issued by PMD. According to the official documentation, PMD has predefined 20 rules regarding performance and suboptimal code [6]. We implemented a program that runs PMD checks on the assignment dataset and obtains all of the performance warnings issued by PMD.

Table 1: Description on the assignments and submissions used in the study.

Semester	Assignment Description	# Sub-tasks	# Submissions
2022 Spring	A *Text Processor* that reads an English text file and performs analyses such as *word search*, *word count statistics*, and *fixing grammar mistakes*.	5	62
2022 Fall	A *Movie Analyzer* that reads a movie dataset and performs analyses such as *movie search*, *movie ranking* and *movie classification*	6	79
2023 Spring	A *Course Analyzer* that reads a online course dataset and performs analyses such as *computing participant statistics* and *course recommendation*.	6	109

In addition to automatic pattern identification, we also performed a manual inspection on the assignments in order to identify inefficient code patterns that are not discovered by PMD. For each subtask, we inspected the top N fastest submissions and top M slowest submissions based on the JMH profiling results (Section 4.1). The inspection stopped upon encountering consecutive implementations that exhibit similar patterns to those observed previously, which indicates that our inspection is likely to have already covered common implementation patterns. Hence, the values of N and M vary for each subtask. The manual inspection was conducted by two of the authors independently, and a third author validated their findings to reach a final conclusion.

By combining the results of both automatic and manual identification, we gain a deeper understanding of the characteristics of inefficient source code in programming assignments. We report the analysis results in Section 5.2.

4.3 Tool Adoption

RQ3 aims to explore materials, tools, and techniques that can be leveraged by educators to enhance students' ability of writing time-efficient programs. In fact, the inefficient code patterns identified in the previous step (Section 4.2) can be directly applied as instructional materials in programming courses, offering detailed code syntax patterns for slow execution and their faster alternatives. PMD can also be used as a teaching aid that can automatically and effectively pinpoint inefficient source code in students' assignments. We elaborate the discussion in Section 6.

We also investigate tools and techniques that can be applied in the classroom and integrated into students' programming workflow to help them write efficient programs. To this end, we investigated *IntelliJ IDEA* and *ChatGPT*.

4.3.1 IntelliJ IDEA. IntelliJ IDEA (IDEA for short), the required IDE used in our course, is a powerful IDE that streamlines software development with a user-friendly interface [2]. Students implement all of the programming assignments and practices using IDEA , making it the ideal tool to explore if we want to enhance students' code efficiency without disrupting their familiar workflow.

One prominent feature of IDEA is *Inspections and Context Actions*, which checks over the quality of source code and provides fix suggestions based on a predefined set of standard Java inspections [4]. Basically, IDEA highlights potential code issues on the fly and users could fix the issue by simply clicking the suggested actions in the popup. If this builtin feature of IDEA already covers the inefficient code patterns we have identified, educators could promote its usage throughout the course and employ it as an effective teaching aid to improve the time efficiency of students' assignments.

For this reason, we study the inspection behavior of IDEA (version 2022.2.4) for inefficient assignment code. First, for each inefficient code pattern observed in the pattern identification process (Section 4.2), we randomly selected an assignment method that encloses the corresponding code pattern and used it as the input to IDEA. If IDEA could highlight statements that match our target code pattern, we consider it to successfully locate the inefficient code. Next, we clicked the suggested actions from the popup to automatically transform the code. We consider IDEA to successfully fix the inefficient code if the transformed code could still pass all of the original test cases within a shorter execution time.

4.3.2 ChatGPT. ChatGPT is a large language model created by OpenAI for advanced natural language processing tasks [22]. In addition to engaging in general human-like conversations, ChatGPT also demonstrates proficiency in performing advanced tasks such as programming and debugging [32]. Practitioners have further identified prospects for integrating ChatGPT within the realm of higher education, particularly in disciplines such as mathematics and computer science [26]. For these reasons, we also explore the capabilities of ChatGPT in terms of locating and optimizing inefficient code in programming assignments.

Consistent with the experiments with IDEA, we used the same code input of IDEA for ChatGPT.[1] Specifically, for each code input, which is an executable Java method from students assignments that represents an inefficient pattern, we constructed the sentence "*Modify the code to improve the efficiency but not to change the functionality*: `<inefficient code>`" and used it as the initial prompt to ChatGPT. We then manually inspected the response of ChatGPT to evaluate whether it has located and optimized our target inefficient code pattern. If so, the process was finished for this pattern and we moved on to the next pattern.

Otherwise, we continued the conversation by providing ChatGPT the second prompt, which elaborates on the specific *task* that ChatGPT should be looking for (e.g., "*Can you make some improvements on string concatenation?*"). If ChatGPT still could not generate an efficient alternative code, we provided a prompt that further points out the specific *API* to look for (e.g., `StringBuilder.append()`).

[1] GPT-3.5 accessed at August 27, 2023.

Table 2: Statistics of assignments execution time (ms).

	Min	Max	Mean	Std.	CoV
22 Spring	3.10	14031.48	547.86	1865.22	3.40
22 Fall	2.37	14253.51	6853.31	2082.15	3.04
23 Spring	0.04	59.05	5.51	10.71	1.94

If ChatGPT can generate an efficient alternative within three rounds of queries, we consider it to have succeeded. Otherwise, we consider ChatGPT to have failed the task. We also recorded the conversation length (i.e., the number of query-response cycles) for each pattern. Section 5.3 reports the results.

5 RESULTS

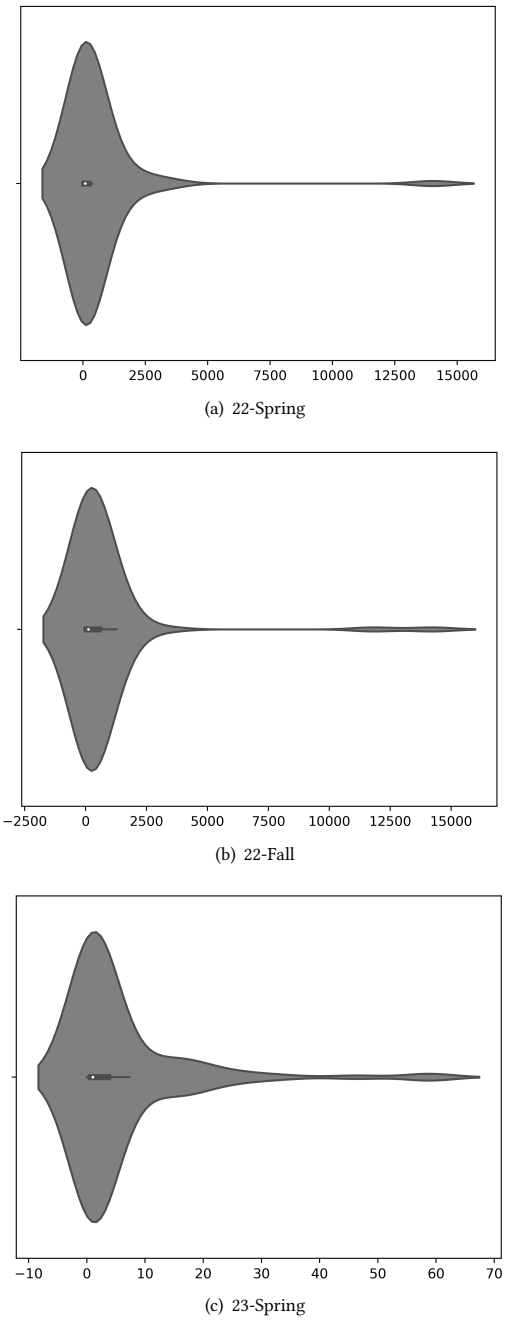
In this section, we present the study results regarding each of the three research questions.

5.1 Time Efficiency

As described in Section 4.1, we obtained the profiling results of programming assignments to examine their time efficiency. Table 2 shows the descriptive statistics of assignments' execution time. In each semester, a notable variance is observed, wherein the most efficient submission executes ~10^4x faster than the slowest submission. From Figure 2, which shows the runtime distribution of all assignments, we further observed that students' assignments tend to be bimodally distributed in terms of the execution time: the majority of submissions exhibit favorable time efficiency; however, 36 submissions (14.4%) take an extremely long time to execute, making them outliers within our dataset. Interestingly, only a few submissions demonstrate intermediate efficiency levels between these two extremes. This observation indicates that for certain tasks, there might exist two different implementation patterns that are both commonly used by users but with distinct time efficiency. We will elaborate on this issue in the next section.

To better quantify the efficiency variations of students' assignments, we compute the *Coefficient of Variation* (CoV), which is defined as the ratio of the standard deviation to the mean. Distributions with CoV > 1 are generally considered high-variance [7]. As shown in Table 2, the 22-Spring assignment exhibits the highest variation of time efficiency while the 23-Spring assignment exhibits the lowest variation. Nonetheless, all three assignments have a CoV larger than 1. We further computed the CoV values for all subtasks [1]. Similarly, all of the subtasks have a CoV larger than 1, revealing significant variations in terms of time efficiency.

Upon closer examination, we observed that subtasks with lower CoV tend to involve routine procedures such as sorting (e.g., 22S-subtask1 and 22F-subtask4). Students had a clear idea of which APIs to use for such tasks (e.g., `Collections.sort`), leading to relatively low deviations in the execution time of their respective implementations. On the other hand, subtasks with higher CoV tend to be more open-ended and flexible. For example, 22S-subtask3 requires students to identify the first letter of every sentence in a document; 22S-subtask5 asks students to compute the n-grams of a document. Both tasks can be solved with diverse approaches using different APIs, which is likely to result in greater variations in their runtime efficiencies.



(a) 22-Spring

(b) 22-Fall

(c) 23-Spring

Figure 2: The runtime distribution (ms) of assignments.

RQ1: Programming assignments that are evaluated solely on functional correctness demonstrate a substantial variability in time efficiency. In particular, 14.4% submissions are orders of magnitude slower than their counterparts. In addition, open-ended assignment tasks exhibit greater variations of time efficiency compared to routine tasks.

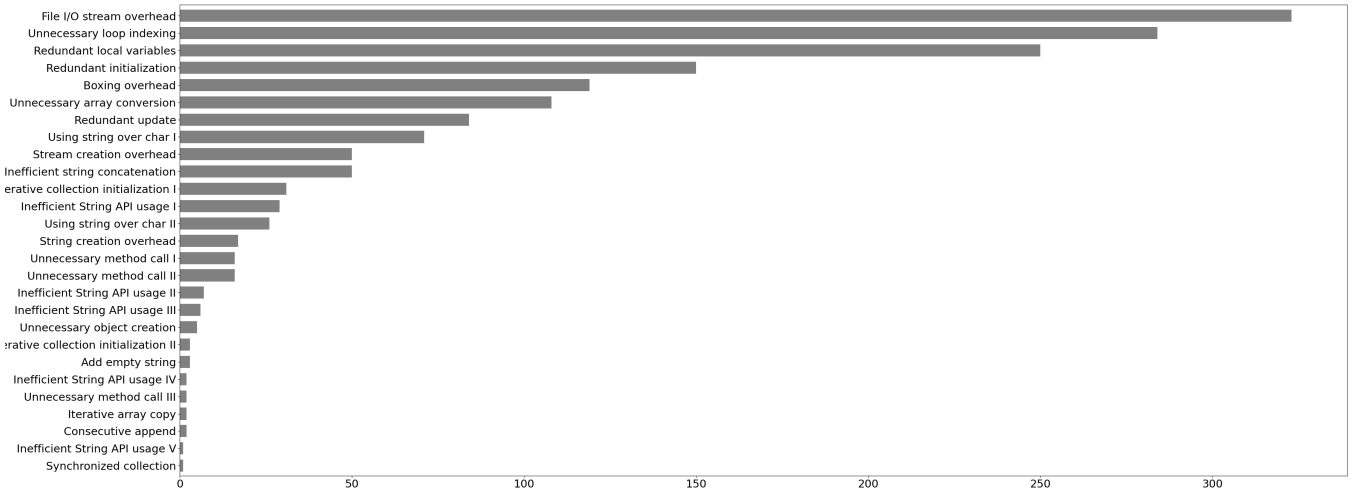


Figure 3: Occurrences of each inefficient code pattern.

Table 3: Categories of inefficient code patterns.

Category	Patterns	Occurrences
String	14	323
Collections	6	371
Variable	6	641
I/O	1	323
Total	27	1658

5.2 Code Characteristics

Using the approach described in Section 4.2, we identified 27 inefficient code patterns along with their faster alternatives. Among which, 13 patterns are identified automatically by PMD while 14 patterns are identified by manual inspections . Figure 3 shows the occurrences of each pattern.[2]

Based on the primary data types and operations used in the inefficient code, we further classified these patterns into four categories: *String*, *Collections*, *Variables*, and *I/O*.

5.2.1 The String Category. Patterns in this category operate on `String` or `StringBuffer` types and invoke their APIs. A typical reason for code inefficiency in this category is the *unawareness* of certain String-related APIs, which causes students to make unnecessary method calls. The sixth pattern (based on the pattern frequency) in Table 4 shows such an example: the inefficient code transforms a string to a character array and then queries the array to retrieve a specific character. However, the transformation using `String.toCharArray` is unnecessary as the `String.charAt` API combines these two steps into a single operation.

Even when students are aware of certain String APIs, the lack of comprehension regarding the internal workings of these APIs could also result in the development of inefficient code. Take the tenth pattern in Table 4 as an example. The "+=" operator for appending strings causes the JVM to create and use an internal `StringBuffer` [6]. If the concatenation occurs inside a loop, a new `StringBuffer` will be created for every iteration, which leads to potential performance degradation. Unfortunately, it is hard for students to spot this inefficiency if they do not understand the internal workings of the "+=" operator in the first place.

5.2.2 The Collections Category. Patterns in this category are related to the *Java Collections Framework*, which offers various data structures and algorithms for managing and manipulating collections of objects. For example, sorting a list can be achieved by `Collections.sort` while list concatenation can be achieved by `List.addAll`.

However, students may not adopt the collections utilities as expected. On one hand, students might employ the utilities excessively, resulting in unnecessary performance overhead. As an example, the ninth pattern in Table 4 turns a list to a stream before sorting, while the stream transformation is completely unnecessary. On the other hand, students may not fully exploit the convenience of the collections framework. For example, the eleventh pattern in Table 4 uses loop to concatenate elements to a list one at a time, which, however, could be replaced by the more efficient API call to `Collection.addAll`.

5.2.3 The Variable Category. In this category, code inefficiency is typically caused by redundant variable declaration or initialization. The fourth pattern in Table 4 shows an example of redundant variable initialization. Basically, while Java automatically initializes class fields with known default values, students still implement explicit initialization, which leads to three additional bytecode instructions per field [6]. We also observed non-trivial instances in student assignments in which variables are initialized but never used.

5.2.4 The I/O Category. Code in this category tackles the file I/O operations. The first row of Table 4 describes the pattern, where `Files.newInputStream(Paths.get(fileName))` should be used instead of `new FileInputStream(fileName)`, as `FileInputStream` contains a finalizer method which will cause garbage collection to

[2]A single assignment can contain multiple inefficient code patterns.

Table 4: Examples of inefficient code patterns and their efficient alternatives, ranked by frequency.

Rank	Inefficient code pattern	Efficient alternative	Type	Tools
1	`FileInputStream fis = new FileInputStream(fileName);`	`InputStream is = Files.newInputStream(Paths.get(fileName));`	I/O	PMD
4	`public class C { int i = 0; Object o = null; }`	`public class C { int i; Object o; }`	Variable	PMD IDEA ChatGPT
6	`String str = ...; char[] chars = str.toCharArray(); char c = chars[index];`	`String str = ...; char c = str.charAt(index);`	String	ChatGPT
9	`List list = ...; list.stream().sorted();`	`List list = ...; Collections.sort(list);`	Collections	ChatGPT
10	`String result = ""; for (int i = 0; i < 10; i++) { result += getStringFromSomewhere(); }`	`StringBuilder temp = new StringBuilder(); for (int i = 0; i < 10; i++) { temp.append(getStringFromSomewhere()); } String result = temp.toString();`	String	PMD IDEA ChatGPT
11	`List<Integer> list1 = ...; List<Integer> list2 = new ArrayList<>(List.of(1,2,3)); for(int i=0; i<list2.size(); i++){ list1.add(list2.get(i)); }`	`List<Integer> list1 = ...; List<Integer> list2 = new ArrayList<>(List.of(1,2,3)); list1.addAll(list2);`	Collections	IDEA ChatGPT

pause [6]. Again, identifying this inefficiency requires the knowledge of how `FileInputStream` works internally.

Table 3 reports the number of patterns in each category as well as the respective occurrences in students' submissions. We observed that the top inefficient code pattern (14/27) is associated with the *String* data type and API usages. In terms of frequency, 38.7% inefficient code occurrences are related to *Variables* while 22.4% are related to *Collections*. It is worth mentioning that although we identified only one pattern for file I/O, this particular pattern accounts for 19.5% of the inefficient code occurrences observed in our dataset.

RQ2: More than half of the inefficient code patterns involve String usages. Students also frequently write inefficient code regarding Collections, Variables and I/O, possibly due to their limited comprehension of the internal mechanisms of these concepts.

5.3 Teaching Aids

As described in Section 4.3, we investigated the performance of IDEA and ChatGPT in terms of locating and fixing inefficient code. Both tools accomplish the paired tasks of localizing inefficient code and fixing it through a single action. In other words, we did not observe cases where IDEA or ChatGPT only localizes the inefficient code without fixing it. In addition, for the 27 inefficient code patterns used in the experiments, neither IDEA nor ChatGPT generates new fixes that are distinct from the efficient alternatives we have identified.

As shown in Figure 4, which visualizes the number of inefficient code fixed by the three tools, PMD and IDEA complement each other effectively: both tools can fix inefficient code that the other might overlook, and together they can fix 93% (25 out of 27) inefficient code. On the other hand, nearly all code fixed by PMD and IDEA can also be fixed by ChatGPT, which alone could fix almost all the inefficient code (26 out of 27) used in our experiment dataset.

We did not observe prominent distinction between the three tools regarding the type of inefficient code they detect. Among the five inefficient code snippets that can be fixed by all of the three tools, two patterns are related to *Strings*, two others are related to *Collections*, and the remaining one is related to *Variables*. The fourth and tenth patterns in Table 4 present examples of inefficient code that can be fixed by all three tools.

Nonetheless, each tool still has its unique advantages. For example, PMD includes predefined rules for suboptimal code that are not supported by the other tools (e.g., the first I/O pattern in Table 4). IDEA, on the other hand, fixes redundant variable initializations better than PMD. In particular, if a variable is initialized but never used, IDEA issues a warning while PMD does not, probably because this pattern is overly general to be formulated as a specific performance violation rule.

Finally, we discuss the usability of these tools. First, since IDEA is the default IDE used in our course, it is highly convenient for students to utilize its built-in *Inspections and Context Action* feature for automatically fixing inefficient code. In other words, students do not have to install new plugins or learn new tools, as the fixing seamlessly integrates into their development process and incurs no cognitive interruption or context switching overhead.

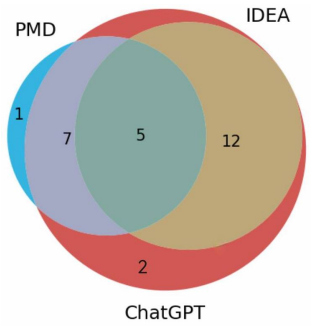


Figure 4: The number of inefficient code snippets fixed by PMD, IDEA and ChatGPT.

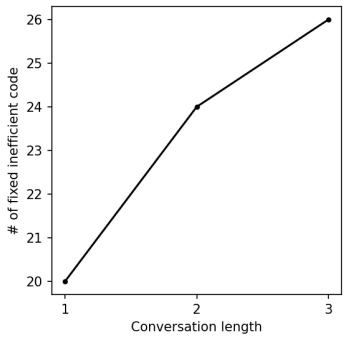


Figure 5: Effect of conversation length on ChatGPT's code fixing ability.

PMD, on the other hand, is a standalone code analyzing tool. It also provides an official plugin for IDEA, which greatly minimizes the context switching overhead and allows students to work with both tools effortlessly within a unified environment. However, PMD only locates the inefficient code and suggests a better alternative. Students still need to fix the code manually.

Similar to PMD, ChatGPT is a standalone tool. Although there are unofficial ChatGPT plugins for IDEA, their primary user interfaces remain to be ChatGPT's text-based chat interface, which operates independently from IDEA's code editor where the actual programming takes place. As a result, students need to manually migrate ChatGPT's fixes to their code and performs necessary integration. In addition, ChatGPT may require multiple prompts before generating a correct fix. As shown in Figure 5, six inefficient code snippets are fixed with two or three prompts. In other words, ChatGPT's fixing mechanism is fundamentally different from that of PMD and IDEA, which fix inefficient code only in one shot.

> **RQ3**: ChatGPT fixes the most inefficient code snippets, but may require multiple attempts and incur context switching overhead in students' development process. IDEA seamlessly integrates with the development workflow of students, and could be further empowered by the PMD plugin to detect and fix more inefficient code.

6 LESSONS LEARNED

Our study reveals the prevalence of performance-related problems in students' programming assignments, prompting us to reevaluate the current course structure and investigate opportunities for enhancing future course offerings.

Performance topics are often underemphasized, and at times, entirely omitted, within introductory programming courses. As discussed in Section 5.2, a large number of inefficient code are related to `String`, which is introduced in a full lecture of CS109. However, this introductory programming course focuses on introducing the correct syntax of using `String`, without bringing students' attention to the internal mechanics of string operations and the associated performance implications. Variable initialization, which is another prevalent cause of inefficient code, is also covered in great detail in CS109. Similarly, teachers focused on the correct syntax and did not explain the performance overhead for redundant initialization, which may lead to the frequent appearance of such an inefficient code pattern in students' assignments.

The disregard of code performance issues may persistently influence students' learning outcomes in subsequent courses. As described in Section 3.1, CS209 extends CS109 with advanced programming concepts and principles. In particular, we incorporated a significantly greater amount of course material addressing performance-related concerns in CS209. Nevertheless, the neglect of performance topics in the prerequisite course keeps affecting the content delivery in the subsequent course. For instance, we do not address the performance concerns for `String` and variable initialization in CS209 because these two topics are delivered in CS109 and therefore are not even included in the syllabus of CS209. Moreover, it can be challenging to alter students' programming habits established early in the prerequisite course. For instance, although we have extensively discussed the internal implementations and performance implications of Java `Collections` in CS209, students still tend to use `Collections` in an inefficient loop, a habit that they developed early in CS109.

Instructions on code performance should be integrated throughout the entire programming course curriculum, instead of being confined to a specific lecture or delivered as an isolated topic. Our study shows that inefficient code patterns in students' assignments involve various programming concepts that are taught across different phases of the programming course curriculum. Unfortunately, our present course structure schedules the delivery of performance-related subjects in the latter stages, resulting in the aforementioned issues. To address these issues, we believe that performance subjects should be consistently addressed and integrated into the curriculum as needed throughout the course. For instance, when teachers introduce the concept of variable initialization, they could provide a supplementary explanation of the performance overhead for redundant variable initialization. Or, when teachers explain common APIs for `String` and I/O, they could discuss alternative implementations and compare their performance at the same time. Such supplementary explanations seamlessly align with the topic, ensuring that students become aware of the performance concerns as soon as they commence learning the concept, thereby contributing to the early development of their sound programming practices.

At our university, the *Software Engineering* course (CS304) is also part of the programming course curriculum, with either CS109 or CS209 as prerequisites. Code quality is one of the important topics in CS304, in which various code metrics, including performance metrics, are discussed. We believe that if students have acquired hands-on experience in enhancing code performance through prerequisite courses, they are more likely to accept and engage with the performance-related material presented in CS304.

Tools and metrics can be further employed to improve students' practical proficiency in writing efficient code. While supplementary explanations in lectures raise students' awareness of code performance, metrics and tools can further assist students in identifying and fixing performance issues in their code. As discussed in Section 5.3, PMD, IDEA, and ChatGPT are all viable tool options for detecting and fixing inefficient code. In particular, IDEA can be easily empowered by the PMD plugin to identify 93% inefficient code patterns. We intend to introduce these tools, preferably during the initial lab session when we first introduce IDEA, in order to equip students with an IDE that is "performance-conscious".

AI tools such as ChatGPT have generated valuable teaching and learning opportunities. However, challenges arise in defining boundaries between acceptable ChatGPT usages and academic dishonesty. Additionally, ensuring students genuinely comprehend the code generated by ChatGPT, rather than resorting to mechanical copy-and-paste practices, poses another significant challenge. Hence, instead of granting students unrestricted access to ChatGPT, we believe that implementing specific constraints is essential, especially for students enrolled in introductory-level programming courses. In the future, we plan to build a tool acting as a "buffer zone" between students and ChatGPT. Students should provide an assignment implementation requiring optimization as input to our tool. Subsequently, the tool would automatically generate queries to ChatGPT, process its responses, and present only the essential information back to the students.

Finally, code performance metrics should carry more weight in the assessment of programming assignments. For the introductory programming course, we intend to integrate a leaderboard into our OJ platform, which ranks students' submissions based on the execution time. The ranking may not necessarily factor into the final grades of students' assignments, but it could serve as an effective motivator for students to improve their code performance. For the upper-level programming course, we intend to add explicit performance requirements in the assessment of programming assignments, encouraging students to proactively optimize their code.

7 THREATS TO VALIDITY

7.1 Internal and Construct Validity

To address RQ1, we used JMH to profile the Java source code in students' assignments. The profiling results shown in Figure 2 are subject to JMH configuration as well as the runtime environment. To minimize this threat, we performed the profiling for all assignments under the same configuration and runtime environment. Research may yield varying absolute execution time under different runtime configurations, but the time differences between assignments should remain consistent as reported in this paper.

To address RQ2, we leveraged PMD and manual inspection to derive inefficient code patterns. However, performance inefficiency may also be caused by students' algorithmic implementation choices in the assignments. Furthermore, the derived patterns may be affected by tool deficiency and human judgement. To mitigate this threat, the manual inspection was initially conducted independently by two of the authors and subsequently checked by a third author. In addition, the derived patterns can also be detected by IDEA and ChatGPT, further validating their credibility.

To address RQ3, we explored the capabilities of PMD, IDEA, and ChatGPT with respect to fixing time-inefficient code. Nonetheless, the study results might be subject to the versions of these tools. In addition, the results generated by tools like ChatGPT can be significantly influenced by the formulation of the prompts. Hence, to facilitate study replication, we specify tool versions and other essential details (e.g., prompts for ChatGPT) in our paper.

7.2 External Validity

The data used in our study are individual programming assignments centered on data analysis tasks, all of which are written in Java. Hence, our findings may not generalize to group assignments, assignments designed for other tasks (e.g., web development), or assignments written in other programming languages. In addition, our dataset consists of programming assignments that are evaluated solely on functional correctness. Due to this design choice, our findings on RQ1 may not generalize to assignments evaluated using other metrics. Furthermore, our results may not generalize to other institutions and courses with different characteristics (e.g., the course enrollment capacity, students demographics, etc.).

8 CONCLUSIONS

In this study, we investigate the time efficiency of 250 programming assignments that are evaluated solely on functional correctness. We observed substantial variability in assignments' execution time, and identified 27 inefficient code patterns that fall into four categories. The study results not only provide actionable guidelines for optimizing inefficient code, but also demonstrate the necessity of using performance metrics in assignment evaluation. Our findings further suggest that code performance topics should be consistently and seamlessly integrated into the programming course curriculum in order to enhance students' awareness and foster good programming habits from the early stages of learning. Finally, we found that PMD, IDEA, and ChatGPT are all effective tools for detecting and fixing time-inefficient code. Educators may consider integrating these tools into the programming courses to help students write efficient code.

To facilitate future research, we have released the artifacts of this study [1], which include the assignment data, profiling scripts, and identified patterns.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive suggestions and insightful comments. This work is supported by the National Natural Science Foundation of China (Grant No. 62202213) and SUSTech Undergraduate Teaching Quality and Education Reform Project (Grant No. Y01271839).

REFERENCES

[1] 2023. Assignment Dataset and Other Artifacts. https://drive.google.com/file/d/1mfaJQqva3QZfOsM8TBehSVwTINhEwM08/view?usp=sharing

[2] 2023. IntelliJ IDEA – the Leading Java and Kotlin IDE. https://www.jetbrains.com/idea/

[3] 2023. Java Microbenchmark Harness (JMH). https://github.com/openjdk/jmh

[4] 2023. List of Java inspections. https://www.jetbrains.com/help/idea/list-of-java-inspections.html

[5] 2023. PMD: An extensible cross-language static code analyzer. https://pmd.github.io/

[6] 2023. PMD Predefined Rules: Performance. https://docs.pmd-code.org/latest/pmd_rules_java_performance.html

[7] LJ Anthony. 2003. The Cambridge dictionary of statistics. *Reference Reviews* 17, 1 (2003), 29–30.

[8] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. 2003. On automated grading of programming assignments in an academic institution. *Comput. Educ.* 41 (2003), 121–131.

[9] Benjamin Clegg, Maria-Cruz Villa-Uriol, Phil McMinn, and Gordon Fraser. 2021. Gradeer: An Open-Source Modular Hybrid Grader. (2021), 60–65. https://doi.org/10.1109/ICSE-SEET52601.2021.00015

[10] Benjamin Simon Clegg. 2021. *The application of mutation testing to enhance the automated assessment of introductory programming assignments.* Ph. D. Dissertation. University of Sheffield.

[11] Sébastien Combéfis. 2022. Automated code assessment for education: review, classification and perspectives on techniques and tools. *Software* 1, 1 (2022), 3–30.

[12] Sébastien Combéfis and Vianney le CLÉMENT de SAINT-MARCQ. 2012. Teaching Programming and Algorithm Design with Pythia, a Web-Based Learning Platform. *Olympiads in Informatics* 6 (2012).

[13] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. Litterbox: A linter for scratch programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 183–188.

[14] J. E. Gaffney. 1981. Metrics in Software Quality Assurance. In *Proceedings of the ACM '81 Conference (ACM '81)*. Association for Computing Machinery, New York, NY, USA, 126–130. https://doi.org/10.1145/800175.809854

[15] Chuqin Geng, Zhang Yihan, Brigitte Pientka, and Xujie Si. 2023. Can ChatGPT Pass An Introductory Level Functional Language Programming Course? *arXiv preprint arXiv:2305.02230* (2023).

[16] Austin Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 165–170. https://doi.org/10.1109/ICSE-SEET52601.2021.00026

[17] Sajed Jalil, Suzzana Rafi, Thomas D LaToza, Kevin Moran, and Wing Lam. 2023. Chatgpt and software testing education: Promises & perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 4130–4137.

[18] Ville Karavirta and Petri Ihantola. 2010. Automatic assessment of javascript exercises. *Proceedings of the 1st Educators' Day on Web Engineering Curricula, Vienna, Austria* (2010), 5–9.

[19] David Kawrykow and Martin P Robillard. 2009. Detecting inefficient API usage. In *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 183–186.

[20] 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 *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 29–39. https://doi.org/10.1145/3510456.3514149

[21] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 160–170.

[22] OpenAI. 2023. ChatGPT. https://openai.com/blog/chatgpt

[23] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages. https://doi.org/10.1145/3513140

[24] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. Association for Computing Machinery, 92–97. https://doi.org/10.1145/3059009.3059026

[25] Kai Presler-Marshall, Sarah Heckman, and Kathryn Stolee. 2021. SQLRepair: Identifying and Repairing Mistakes in Student-Authored SQL Queries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 199–210. https://doi.org/10.1109/ICSE-SEET52601.2021.00030

[26] Basit Qureshi. 2023. Exploring the use of chatgpt as a tool for learning and assessment in undergraduate computer science curriculum: Opportunities and challenges. *arXiv preprint arXiv:2304.11214* (2023).

[27] Jake Renzella and Andrew Cain. 2020. Enriching Programming Student Feedback with Audio Comments *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 173–183. https://doi.org/10.1145/3377814.3381712

[28] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.

[29] Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2021. Speeding up data manipulation tasks with alternative implementations: an exploratory study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–28.

[30] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[31] Tianjia Wang, Daniel Vargas-Diaz, Chris Brown, and Yan Chen. 2023. Towards Adapting Computer Science Courses to AI Assistants' Capabilities. *arXiv preprint arXiv:2306.03289* (2023).

[32] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[33] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*. 800–810.